

React Native

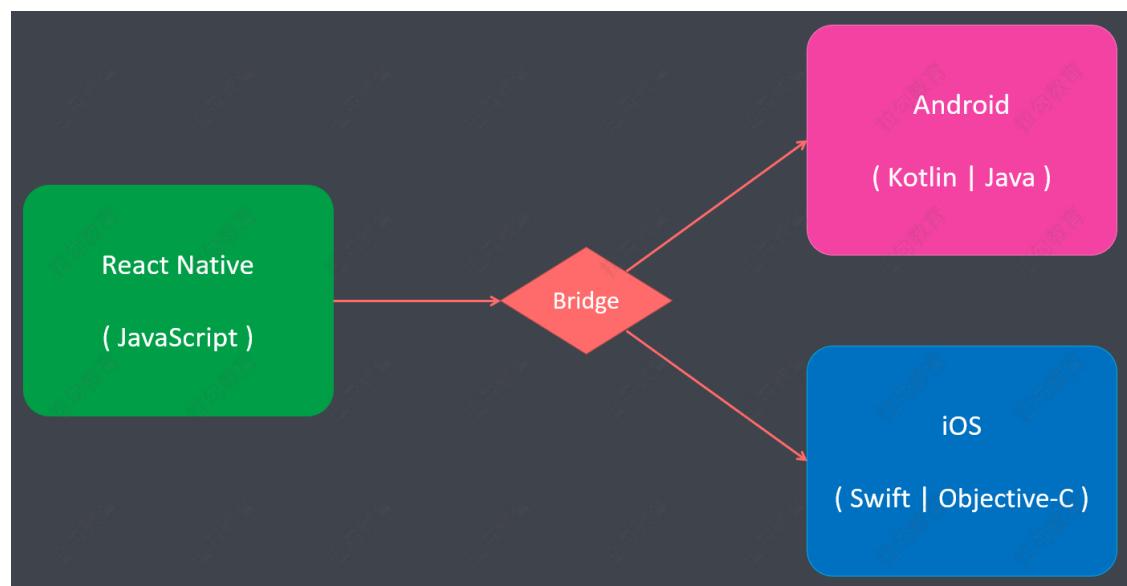
概述

简介

React Native (简称 RN) 是 Facebook 于2015年4月开源的跨平台移动应用开发框架，是 Facebook 早先开源的 JS 框架 React 在原生移动应用平台的衍生产物，支持 iOS 和 Android 两大平台。RN 使用 Javascript 语言，因此熟悉 Web 前端开发的技术人员，只需很少的学习就可以进入移动应用开发领域。

原理

RN 中的代码，经过转换后，会生成 Android 和 iOS 对应的原生代码。我们开发时，写的是 JavaScript，用的是 React 语法。项目编译后会生成两个环境下对应的代码。



环境搭建

RN 可以开发安卓和苹果应用。需要注意的是

- 在 Windows 下，只能搭建开发安卓应用的环境。而不能搭建开发苹果应用的环境
- 在 Mac 下，既可以搭建安卓应用的环境，又可以搭建开发苹果应用的环境

RN 的 Android 和 iOS 这两种环境的搭建，会有差异。

上课过程中，我们以 Windows 10 操作系统下，搭建 Android 开发环境为主。给大家演示环境搭建过程。

基础环境

必须安装的依赖有：Node.js、Yarn 和 React Native 脚手架

- Node.js
 - Node.js 的版本应大于等于 12，推荐安装 LTS 版本

14.15.0 长期支持版

推荐多数用户使用 (LTS)

[其它下载](#) | [更新日志](#) | [API 文档](#)

15.2.0 当前发布版

含最新功能

[其它下载](#) | [更新日志](#) | [API 文档](#)

- 安装之后，（为了提高下载速度）请设置 npm 镜像源

修改npm的镜像源

```
npm config set registry https://registry.npm.taobao.org
```

验证是否更改成功（查看镜像源）：

```
npm config get registry
```

- 安装 yarn

- yarn 是 Facebook 提供的替代 npm 的工具，可以加速 node 模块的下载

- npm install -g yarn

- 安装 React Native 脚手架

```
npm install -g react-native-cli
```

注意：本节安装的 React Native 版本是 0.63，由于不同的 RN 版本之前差异较大，请确保，你的 RN 版本与我一致（本文档是针对 0.63 写的，0.4x, 0.5x 的同学请绕行）

React Native 开发的 App，既可以发布为安卓应用，也可以发布为苹果应用。所以为了代码调试的需要，我们需要安装对应的两种开发环境。

注意：如果你只需要开发安卓应用，则不需要搭建 iOS 环境。同理，如果你只需要开发苹果应用，则无需搭建安卓开发环境。但一般用 React Native 开发，需要将应用发布到两端，所以，一般需要搭建两种环境。

安卓环境

请查看文档 [RN 安卓环境搭建.md](#)

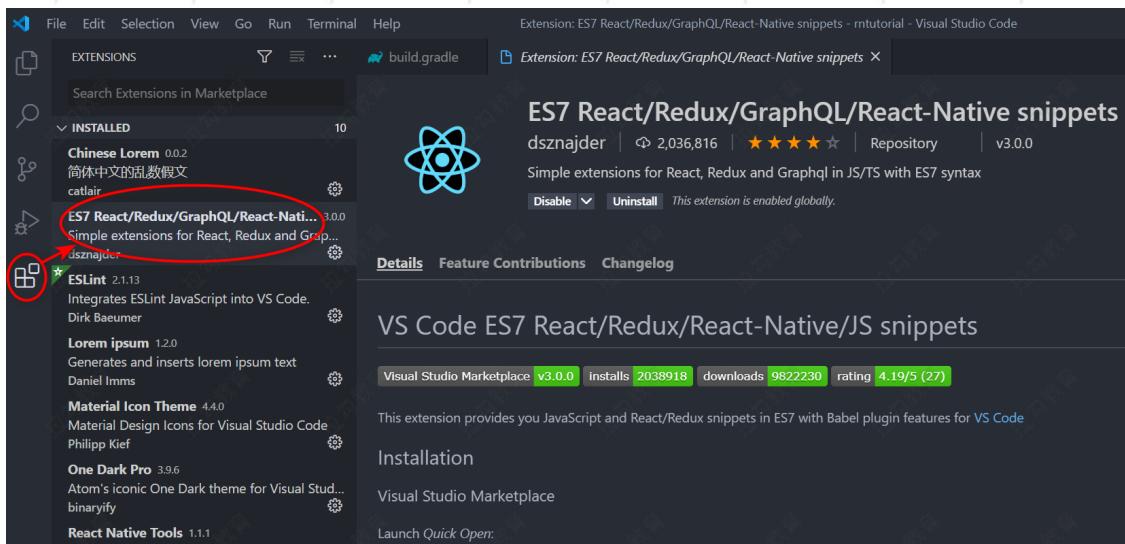
iOS环境

请查看文档 [RN iOS 环境搭建.md](#)

Vscode 插件

好的插件，在项目开发中，能起到事半功倍的效果。

这里推荐在 vscode 下安装插件：ES7 React/Redux/GraphQL/React-Native snippets。她具有很多有用的快捷命令。可以让我们快速，高效的写 RN 代码



快捷命令：

快捷命令可以帮助我们快速的创建代码，具体操作是输入 <快捷命令> 然后回车。下面给出了最长用的快捷命令

rcc (react create class)

```
import React, { Component } from 'react'

export default class FileName extends Component {
  render() {
    return <div>$2</div>
  }
}
```

rfc (react create function)

```
import React from 'react'

export default function $1() {
  return <div>$0</div>
}
```

rnc (react native class)

```
import React, { Component } from 'react'
import { Text, View } from 'react-native'

export default class FileName extends Component {
  render() {
    return (
      <View>
        <Text> $2 </Text>
      </View>
    )
  }
}
```

rnf (react native function)

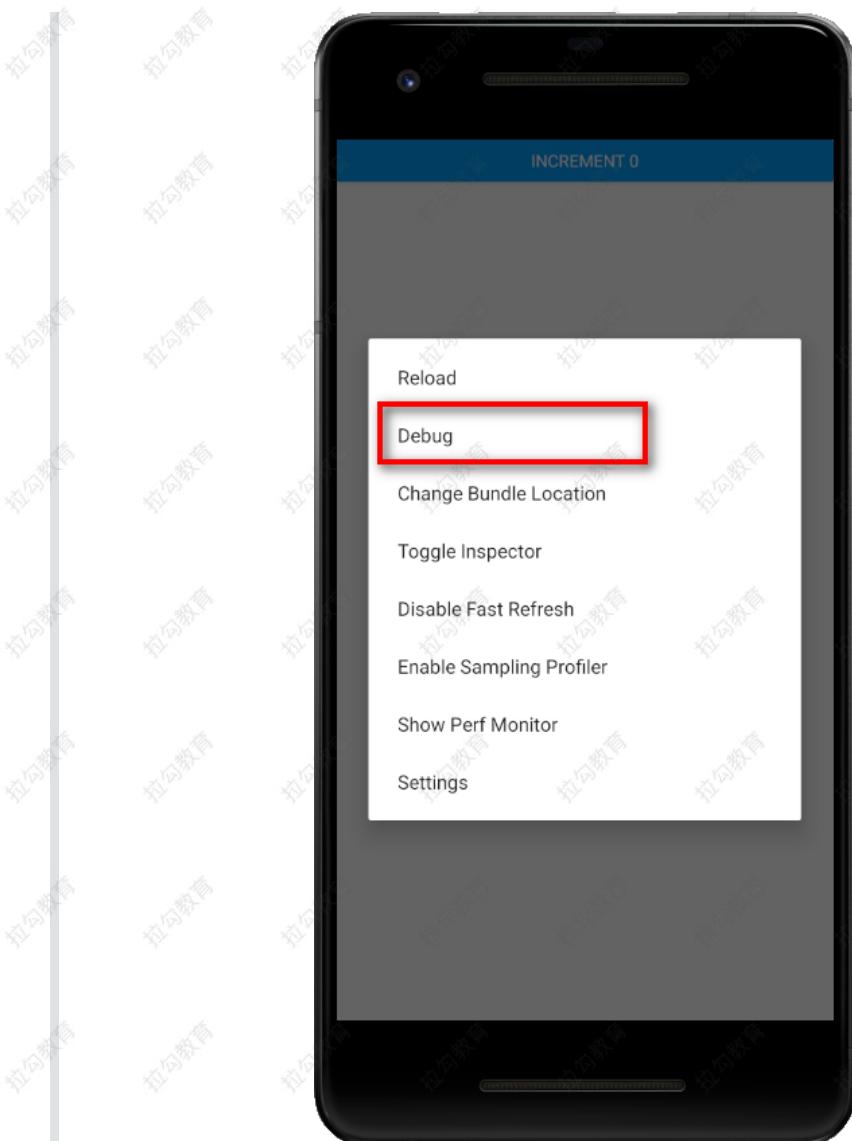
```
import React from 'react'
import { View, Text } from 'react-native'

export default function $1() {
  return (
    <View>
      <Text> $2 </Text>
    </View>
  )
}
```

调试工具

调试 RN 一般有两种方式：

- 浏览器调试
 - 点击模拟器（让模拟器获取焦点）
 - **ctrl+m** 快捷键，打开菜单，然后点选 debug



- 然后会跳转到浏览器，在浏览器上你会看到如下效果。然后，在页面上右键，点“检查”。就可以调试了

The screenshot shows the React Native Debugger UI at `localhost:8081/debugger-ui/`. It includes settings for 'Dark Theme' and 'Maintain Priority'. A note states that React Native JS code runs as a web worker inside this tab. It advises pressing `Ctrl+Shift+I` to open Developer Tools and enables `Pause On Caught Exceptions` for better debugging. It also suggests installing the standalone version of React Developer Tools. The status indicates a debugger session is active. A 'Reload app' button is present.

- 配置捕获网络请求

RN 发送的网络请求，默认是无法通过浏览器调试的。解决方案如下：

在入口文件（`index.js` 或 `App.js`）中加入这一行

```
GLOBAL.XMLHttpRequest = GLOBAL.originalXMLHttpRequest ||  
GLOBAL.XMLHttpRequest
```

- react-native-debugger

- 下载

下载地址：<https://github.com/jhen0409/react-native-debugger/releases>

- 安装

下载的压缩包中，有启动程序。这是绿色版。不需要安装。你只需要把解压后的文件放到一个合适的地方

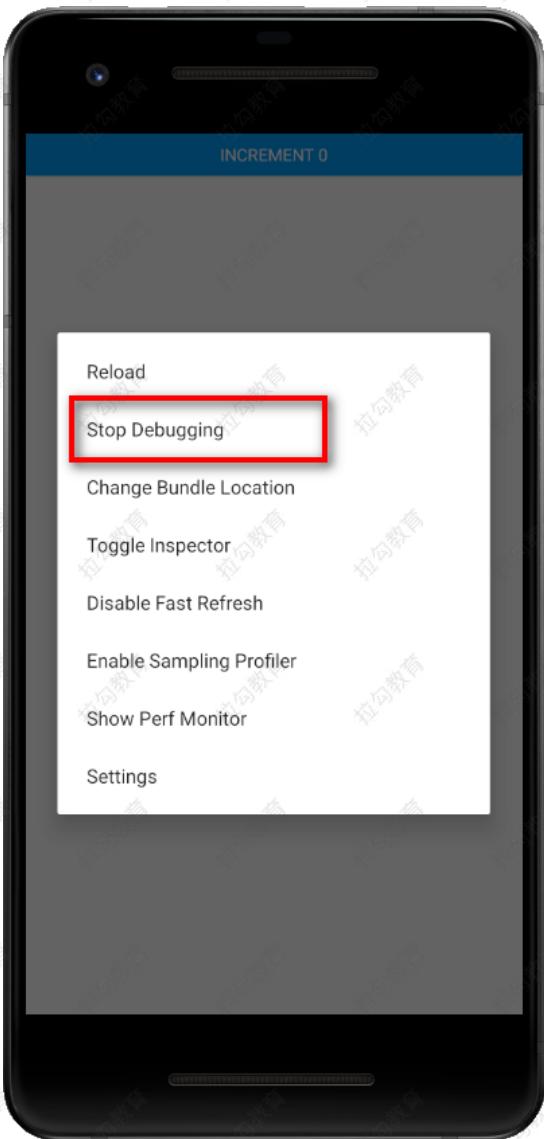
例如：我放到 D 盘下

- 使用

1. 先关掉之前的浏览器调试

有两步操作：

- 关闭调试
- 关闭浏览器



2. 然后双击打开，解压后的执行文件（react-native-debugger.exe）

StyleSheet

StyleSheet 是 RN 中用来声明样式的语法

RN 中的样式与 CSS 的区别

- RN 中的样式，没有继承性（子组件不能继承父组件的样式）

CSS: Cascading Style Sheet (级联样式表)

SS: Style Sheet (样式表)

- 样式名称使用小驼峰式命名

例如：CSS 中的 font-size，在 RN 中写成 fontSize

- 所有尺寸都是没有单位

例如：width: 100

这是因为 RN 中尺寸只有一个单位，dp（一种基于屏幕密度的抽象单位，默认省略。详情查看 [PixelRatio](#)）

```
import { PixelRatio } from 'react-native';
const dp2px = dp=>PixelRatio.getPixelSizeForLayoutSize(dp);
const px2dp = px=>PixelRatio.roundToNearestPixel(px);

// 按照下面的方式可实现 px 与 dp 之间的转换（比如 100px*200px 的 view）
<View style={{width:px2dp(100),height:px2dp(200),backgroundColor:"red"}} />
```

- RN 中有些样式的写法与 CSS 不同

例如: marginVertical

RN 样式的声明方式

- 通过 style 属性直接声明
 - 属性值是对象

```
// 基本用法
<Text style={{color: '#e33', fontsize: 30}}> Hello world </Text>

// 越靠后的样式，优先级越高；下面字体显示为蓝色
<Text style={{color: 'red', color: 'blue', }}> Hello World </Text>
```

- 属性值是数组

```
// 数组元素是对象，这样可以传入多个样式对象，方便构造复杂的样式效果
<Text style={[{fontSize: 40}, {color: 'green'}]}> Hello world </Text>
```

- 通过 StyleSheet 声明（推荐）

```
// 引入 stylesheet
import { Text, StyleSheet } from 'react-native'

<View style={styles.container}>
  <Text style={styles.red}>Hello world</Text>

  {/* 数组中可以传入多个样式 */}
  <Text style={[styles.red, styles.fontLarge]}>Hello RN</Text>
  <Text style={[styles.red, styles.fontMedium]}>Hello RN</Text>
  <Text style={[styles.red, styles.fontSmall]}>Hello RN</Text>
</View>

// 通过 stylesheet.create({}) 创建样式，与组件内容分开（使用 StyleSheet 之前要先引入）
const styles = StyleSheet.create({
  container: {
    marginTop: 50,
  },
  red: {
    color: 'red',
  },
  fontLarge: {
    fontSize: 40,
  },
  fontMedium: {
    fontSize: 30,
  },
  fontSmall: {
    fontSize: 20,
  },
});
```

```
fontMedium: {  
    fontsize: 30,  
},  
fontSmall: {  
    fontsize: 20,  
}  
};
```

另外，在实际开发中。样式值还可以通过变量的方式指定，例如：

```
<view  
    style={[styles.base, {  
        width: this.state.width,  
        height: this.state.width * this.state.aspectRatio  
    }]}  
/>
```

注意：RN 中的样式不支持 Less 或 Sass 中的 Mixin 特性（这也是一直被吐槽的）

Flexbox

背景

传统布局方式

- 文档流布局

根据代码出现的先后顺序布局，先出现的在前，后出现的在后

- 浮动

例如：左浮动，右浮动，清除浮动等；可以打破文档流的固有模式，实现更为丰富的布局

- 定位

绝对定位，相对定位，固定定位

传统布局方式，对于那些特殊布局非常不方便，比如，垂直居中就不容易实现。

基于此，2009年，W3C 提出了一种新的方案：Flex 布局，可以简便、完整、响应式地实现各种页面布局。

Flexbox 规范详情：<https://www.w3.org/TR/css-flexbox-1/>

简介

Flexbox 是 Flexible Box 的缩写，意为“弹性布局”，用来为盒状模型提供最大的灵活性。

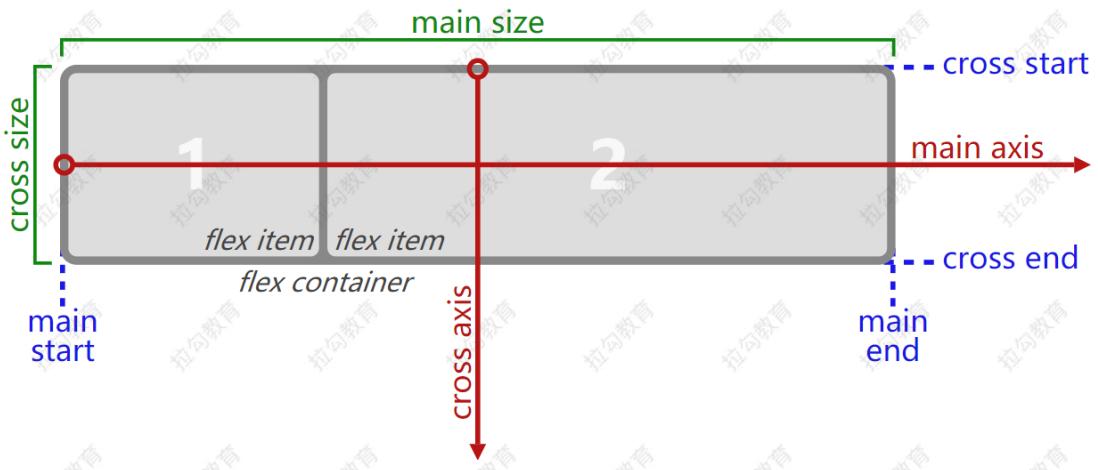
```
div {  
    display: flex;  
}
```

设为 Flex 布局以后，子元素的 float、clear 和 vertical-align 属性将失效

术语

容器：采用 Flex 布局的元素，称为 Flex 容器（flex container），简称“容器”。

项目：它的所有子元素自动成为容器成员，称为 Flex 项目（flex item），简称“项目”



容器默认存在两根轴：主轴（main axis）和交叉轴（cross axis）。

主轴的开始位置（与边框的交叉点）叫做 `main_start`，结束位置叫做 `main_end`；

交叉轴的开始位置叫做 `cross_start`，结束位置叫做 `cross_end`

项目默认沿主轴排列。单个项目占据的主轴空间叫做 `main_size`，占据的交叉轴空间叫做 `cross_size`

属性

flexDirection

`flexDirection` 用来指定主轴方向（CSS 中对应的属性名是: `flex-direction`）。有四种方式：

- `row`: 主轴是水平方向，从左向右
- `row-reverse`: 主轴是水平方向，从右向左
- `column`: 主轴是垂直方向，从上到下
- `column-reverse`: 主轴是垂直方向，从下往上

注意：**Web 开发中默认的主轴方向是 `row`，而 RN 中默认的主轴方向是 `column`**

`flexDirection` 的效果展示：

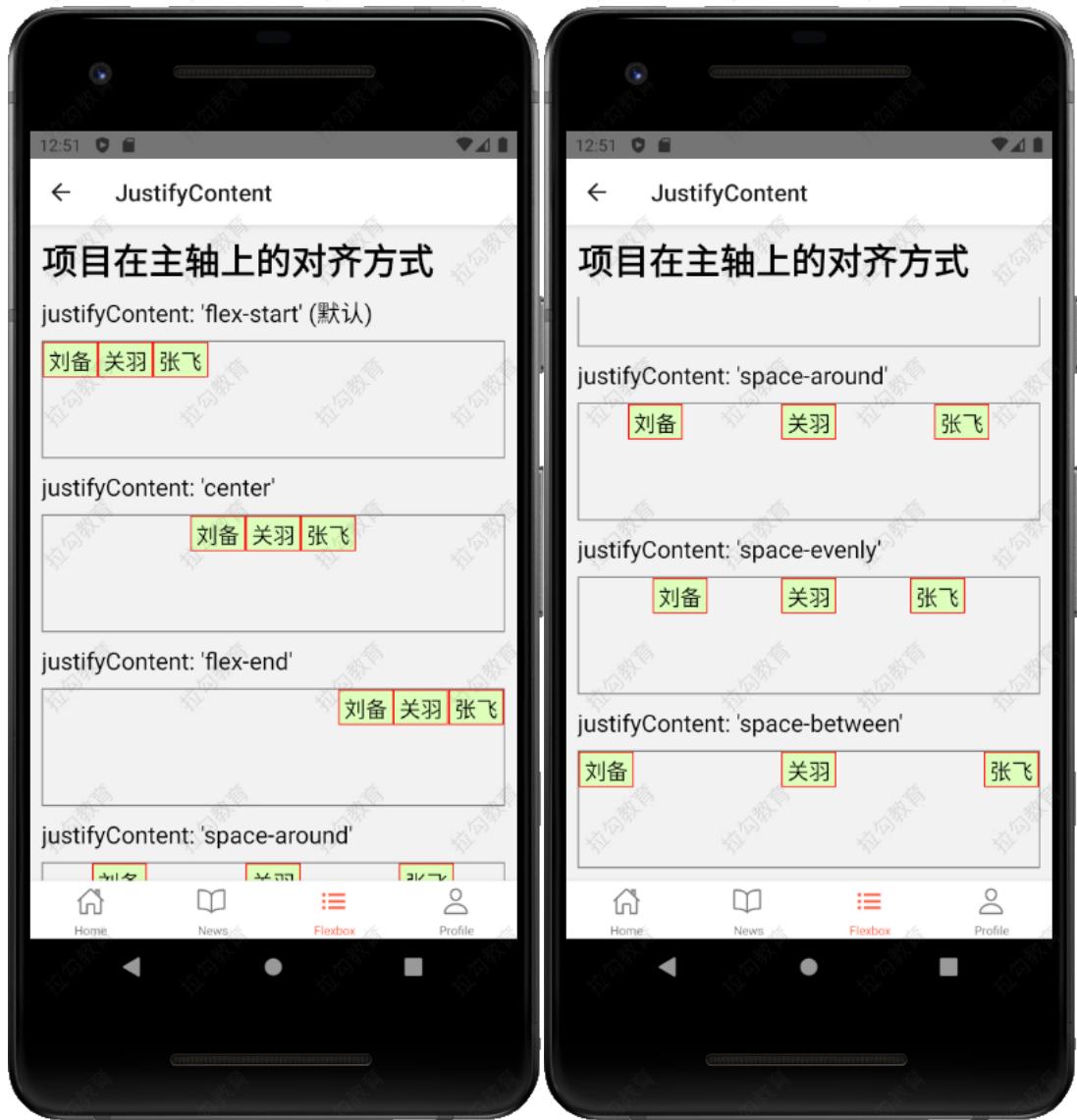


justifyContent

justifyContent (CSS 中对应的属性名是 justify-content) 用来指定项目在主轴上的对齐方式。有六种方式：

- flex-start：项目沿主轴开始位置对齐
- center：项目在主轴上居中对齐
- flex-end：项目沿主轴的结束位置对齐
- space-around：在主轴上，空格环绕对齐
- space-evenly：在主轴上，空格平均分布对齐
- space-between：在主轴上，两端对齐

justifyContent 的效果展示：



alignItems

alignItems (CSS 中对应的属性名是 align-items) 用来指定项目在交叉轴上的对齐方式。有五种方式：

- flex-start: 项目沿交叉轴的开始位置对齐
- center: 项目在交叉轴上居中对齐
- flex-end: 项目沿交叉轴的结束位置对齐
- stretch: 项目在交叉轴上拉伸对齐
- baseline: 项目在交叉轴上基线对齐 (沿着内容的底部基线)

alignItems 效果展示 (这里我们把主轴方向声明为 row, 交叉轴是垂直方向) :



最后一个，基线对齐，为了看到效果，我们故意把关羽和张飞的字体放大

flex

flex 是以数字的方式，来声明项目在主轴上的尺寸占比。

例如：我们把3个项目的 flex 值，分别设置为 1, 2, 3。则三个项目的整体尺寸是 $1+2+3 = 6$ 。所以，flex 值为3的项目就占 50% ($3/6 = 50\%$)

flex 效果展示（这里我们分别从 row 和 column 两个方向，来体验项目的尺寸占比）：



组件和API

简介

在 React Native 项目中，所有展示的界面，都可以看做是一个组件（Component），只是功能和逻辑上的复杂程度不同。

原生组件

在 Android 开发中是使用 Kotlin 或 Java 来编写视图；在 iOS 开发中是使用 Swift 或 Objective-C 来编写视图。在 React Native 中，则使用 React 组件通过 JavaScript 来调用这些视图。在运行时，React Native 为这些组件创建相应的 Android 和 iOS 视图。由于 React Native 组件就是对原生视图的封装，因此使用 React Native 编写的应用外观、感觉和性能与其他任何原生应用一样。我们将这些平台支持的组件称为**原生组件**。

作用	RN 组件	安卓视图	iOS 视图	HTML 标签
展示区块	View	ViewGroup	UIView	div
展示图片	Image	ImageView	UIImageView	img
展示文本	Text	TextView	UITextView	p
.....

核心组件

原生组件

React Native 允许您为 Android 和 iOS 构建自己的 Native Components (原生组件)。

核心组件

React Native 还包括一组基本的，随时可用的原生组件，您可以使用它们来构建您的应用程序。这些是 React Native 的 **核心组件**。（来自 react-native 的组件叫核心组件）

```
import {
  View,
  Text,
  TouchableOpacity,
  Dimensions,
  StyleSheet,
  StatusBar,
  Image,
  ImageBackground
} from 'react-native';
```

第三方组件

不在 react-native 中的，需要单独安装，然后才能使用的组件

自定义组件

一般指，具有特定功能的，由工程师自己写的，在项目中需要重复使用的组件。

常用组件

View

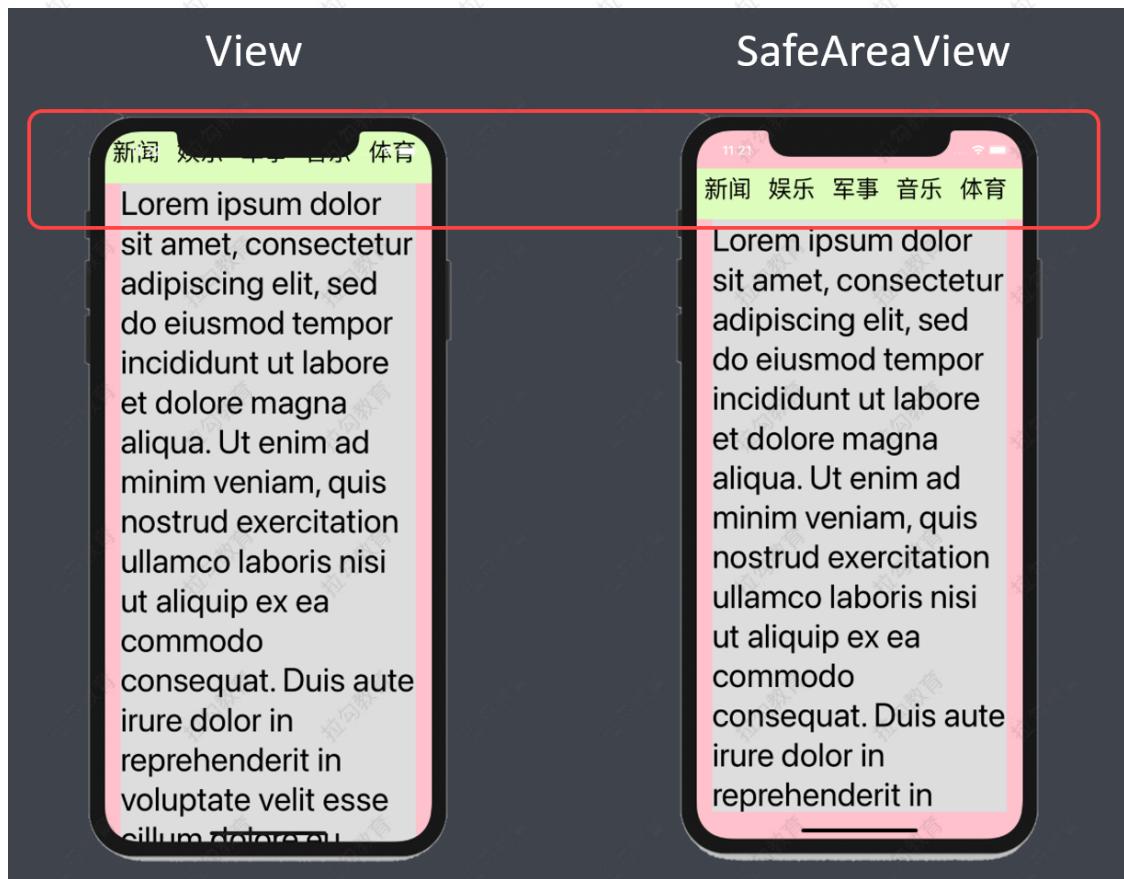
相当于 HTML 中的 div，用来展示内容

```
import { View } from 'react-native'

<View>{/* 内容 */}</View>
```

SafeAreaView

用法与 View 一致，只是 SafeAreaView 可以避开刘海（挖空屏幕）



Text

用来展示文本信息 (RN 中所有的文本，必须包含在 Text 标签中)

```
import { Text } from 'react-native'

<Text>文本内容</Text>

<Text
  style={[styles.newsItemHeader]} // 样式
  numberOfLines={2}           // 文本显示的行数
  ellipsizeMode="tail"        // 从文本的末尾进行截断
  onPress={() => alert('点击')}
  onLongPress={() => alert('长按点击')}
>
文本内容
</Text>
```

ellipsizeMode 的取值为 enum('head', 'middle', 'tail', 'clip')，用来设定当文本显示不下全部内容时，文本应该如何被截断，需要注意的是，它必须和 numberOfLines (文本显示的行数) 搭配使用，才会发挥作用。

- head：从文本的开头进行截断，并在文本的开头添加省略号，例如：...xyz。
- middle：从文本的中间进行截断，并在文本的中间添加省略号，例如：ab...yz。
- tail：从文本的末尾进行截断，并在文本的末尾添加省略号，例如：abcd....
- clip：文本的末尾显示不下的内容会被截断，并且不添加省略号，clip只适用于iOS平台。

Button

```
import { Button } from 'react-native'

<Button
  onPress={onPressLearnMore}
  title="Learn More"
  color="#841584"
/>
```

Button 组件不能使用 style 属性

Alert

引入

```
import { Alert } from 'react-native'
```

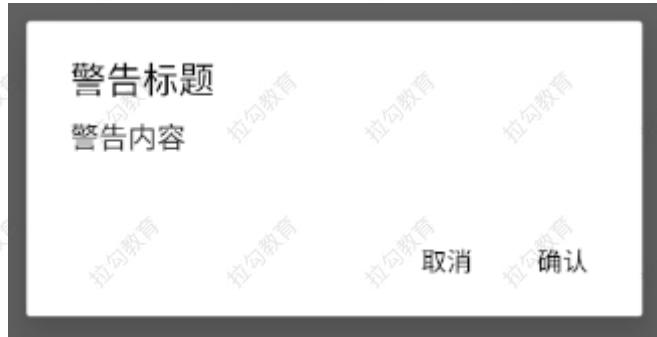
基础用法

```
Alert.alert('This is a button!')
```



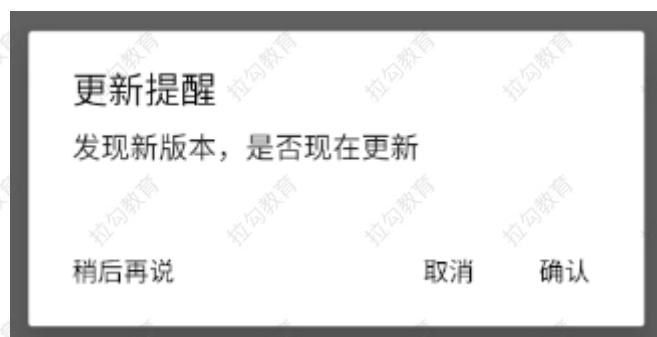
两个按钮

```
Alert.alert(
  "警告标题",
  "警告内容",
  [
    {
      text: "取消",
      onPress: () => console.log("Cancel Pressed"),
      style: "cancel"
    },
    { text: "确认", onPress: () => console.log("OK Pressed") }
  ]
);
```



三个按钮

```
Alert.alert(  
  "更新提醒",  
  "发现新版本，是否现在更新",  
  [  
    {  
      text: "稍后再说",  
      onPress: () => console.log("Ask me later pressed")  
    },  
    {  
      text: "取消",  
      onPress: () => console.log("Cancel Pressed"),  
      style: "cancel"  
    },  
    { text: "确认", onPress: () => console.log("OK Pressed") }  
  ]  
)
```



Switch

开关按钮，类似HTML 中的 CheckBox

引入

```
import { Switch } from 'react-native';
```

使用

```
<Switch  
    trackColor={{ false: "#999", true: "#666" }} // 背景色  
    thumbColor={this.state.hideStatusBar ? "red" : "white"} // 前景色  
    ios_backgroundColor="#3e3e3e"  
    value={this.state.hideStatusBar}  
    onValueChange={this.toggleStatusBar} // 开关处理函数  
/>
```

StatusBar

状态栏位于手机的顶部，一般用来显示网络信号，时间，电量等信息。在 RN 可以通过 StatusBar 来控制状态栏

引入

```
import { StatusBar } from 'react-native'
```

使用

```
<StatusBar  
    backgroundColor="blue" // 设置背景色，仅在 Android 下有效  
    animated={false}  
    hidden={this.state.hideStatusBar} // 是否隐藏 StatusBar  
/>
```

ActivityIndicator

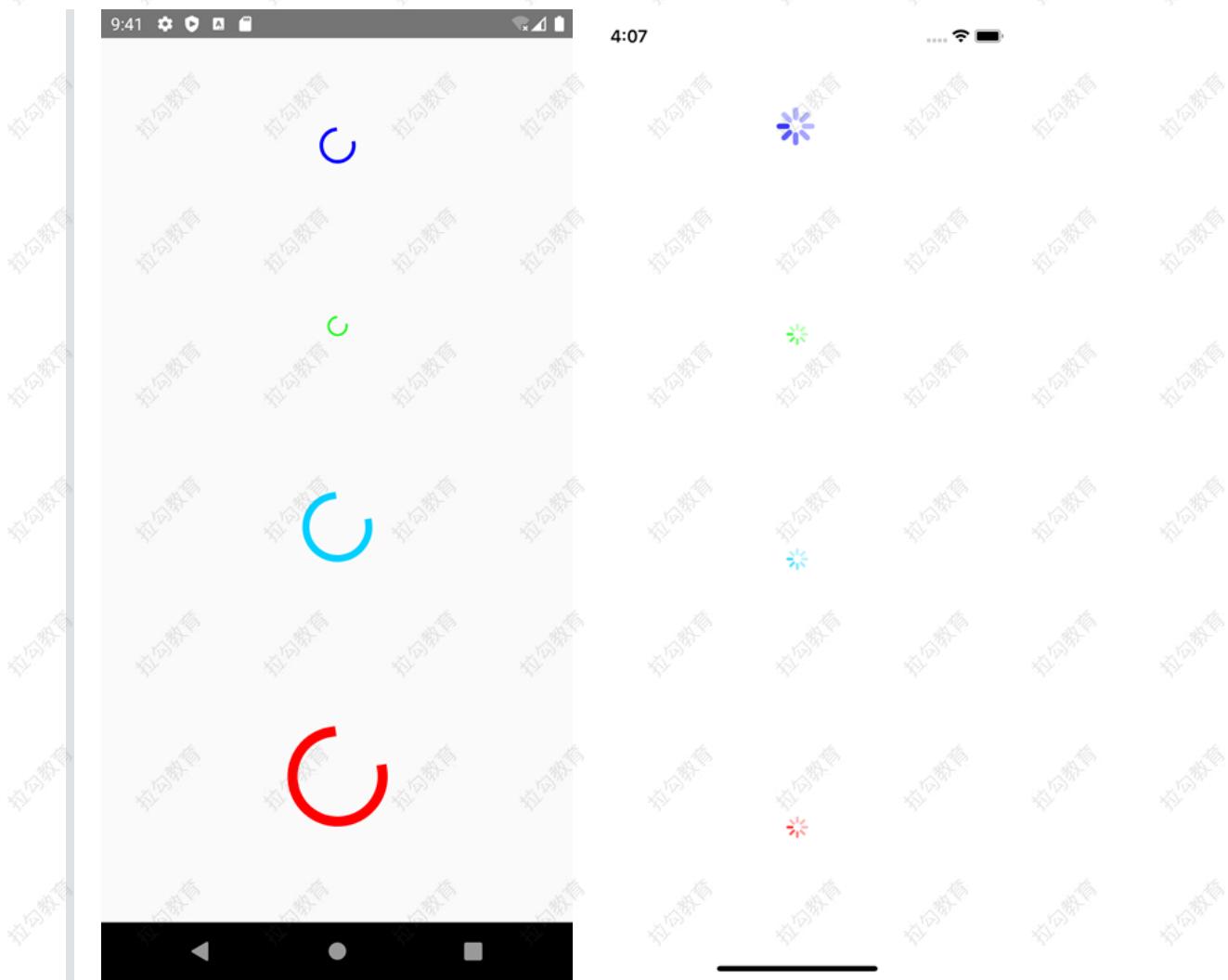
引入

```
import { ActivityIndicator } from 'react-native'
```

使用

```
<ActivityIndicator size="large" color="#0000ff" />  
<ActivityIndicator size="small" color="#00ff00" />  
<ActivityIndicator size={70} color="#00d0ff" />  
<ActivityIndicator size={100} color="red" />
```

Android VS iOS 下的效果（使用数字来声明大小，仅在 Android 下有效。）



Image

用来展示图片。图片路径有三种情况：

本地路径

```
<Image  
  style={styles.slideItem}  
  source={require('../images/1.jpg')}  
/>
```

图片地址只支持静态地址的写法。不支持变量拼接

例如，如下写法无效：

```
source={require('../images/' + imagename + '.jpg')}
```

URL 地址

```
<Image  
  style={styles.mediumLogo}  
  source={{  
    uri: 'https://reactnative.dev/img/tiny_logo.png',  
  }}  
/>
```

base64 字符串

```
<Image
  style={styles.logo}
  source={{

    uri:'data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAADMAAAAZCAYAAAA6oTAqAAAAEXRF
    WHRTbZ0d2FyzQBWbmdjcnVzaEB1SfMAAABQSURBVGje7dSxCQBACARB+2/ab8BEeQNhFi6WSYZLYud
    DQYGBgYGBgYGBgYGBgZmcvDqYGBgmhivGQYGBgYGBgYGBgYGBgbmQw+P/eMrC5UTVAAAAABJ
    RU5ErkJggg==',
  }}
/>>
```

TextInput

RN 中的 TextInput 支持多种表单类型，例如：普通输入框，密码框，文本域等。不同类型的表单是通过不同的属性来实现的。

```
import React, { Component } from 'react'
import { Text, StyleSheet, View, TextInput, Dimensions, Button } from 'react-native'

export default class index extends Component {
  constructor() {
    super()

    this.state = {
      username: '',
      password: ''
    }
  }

  doLogin = () => {
    alert(this.state.username)
  }

  render() {
    return (
      <View style={[styles.container]}>
        <TextInput
          style={[styles.input]} // 表单样式
          placeholder="请输入用户名" // 文字提示
          value={this.state.username} // 表单的值
          onChangeText={(val) => { // 内容变更的处理函数
            this.setState({
              username: val
            })
          }}
        />

        <TextInput
          style={[styles.input]}
          placeholder="请输入密码"
          value={this.state.password}
          secureTextEntry={true} // 启用加密效果
        />
      </View>
    )
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  input: {
    width: 300,
    height: 40
  }
})
```

```
        onChangeText={(val) => {
          this.setState({
            password: val
          })
        }}
      />

      <TextInput
        style={[styles.input]}
        placeholder="手机号"
        keyboardType="number-pad" // 使用数字键盘
      />

      <TextInput
        style={[styles.input]}
        placeholder="请输入自我介绍"
        multiline={true} // 启动多行
        numberOfLines={5} // 指定文本域的行数
        textAlignVertical="top" // 在 Android 中，使文本顶部对齐（默认垂直居中）
      />

      <View style={[styles.btn]}>
        <Button title="登陆" onPress={this.doLogin} />
      </View>
    </View>
  )
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center'
  },
  input: {
    width: Dimensions.get('window').width - 20,
    margin: 10,
    borderwidth: 1,
    borderColor: 'red',
    paddingHorizontal: 5
  },
  btn: {
    margin: 10
  }
})
```





Touchable

引入

```
import {  
  TouchableHighlight,      // 触碰高亮显示（背景高亮）  
  TouchableOpacity,        // 触碰透明显示（透明度发生变化）  
  TouchableWithoutFeedback // 触碰无反馈  
} from 'react-native'
```

使用

```
<TouchableHighlight onPress={() => this.highlight()}>  
  <View style={[styles.item]}>  
    <Text style={[styles.h3]}>触碰高亮</Text>  
  </View>  
</TouchableHighlight>  
<TouchableOpacity onPress={() => this.opacity()}>  
  <View style={[styles.item]}>  
    <Text style={[styles.h3]}>触碰透明度</Text>  
  </View>  
</TouchableOpacity>  
<TouchableWithoutFeedback onPress={() => this.withoutFeedback()}>  
  <View style={[styles.item]}>  
    <Text style={[styles.h3]}>触碰无反馈</Text>  
  </View>  
</TouchableWithoutFeedback>
```

触碰插件一般与事件一起使用

ScrollView

View 没有滚动功能。当内容超出可视区域后，就无法正常显示。此时，我们可以用滚动视图（ScrollView）来显示内容。

引入

```
import { ScrollView } from 'react-native';
```

使用

```
<ScrollView  
    style={{backgroundColor: '#dfb'}} // 指定样式  
    horizontal={true} // 是否水平方向滚动。默认是垂直方向  
    contentContainerStyle={{margin: 30}} // 设置内容样式  
    showsVerticalScrollIndicator={false} // 是否展示垂直方向的滚动条  
    showsHorizontalScrollIndicator={false} // 是否显示水平方向的滚动条  
>  
</ScrollView>
```

SectionList

SectionList 将列表分成若干个章节，每个章节有一个标题。支持下面这些常用的功能：

- 完全跨平台。
- 行组件显示或隐藏时可配置回调事件。
- 支持单独的头部组件。
- 支持单独的尾部组件。
- 支持自定义行间分隔线。
- 支持分组的头部组件。
- 支持分组的分隔线。
- 支持多种数据源结构
- 支持下拉刷新。
- 支持上拉加载。

引入

```
import { SectionList } from "react-native";
```

使用

```
const DATA = [  
  {  
    title: "魏国",  
    data: ["曹操", "司马懿", "张辽"]  
  },  
  {  
    title: "蜀国",  
    data: ["刘备", "关羽", "张飞", "诸葛亮"]  
  },  
  {  
    title: "吴国",  
    data: ["孙权", "周瑜", "鲁肃"]  
  }];  
// ...  
  
const Item = ({ title }) => (  
  <View style={styles.item}>  
    <Text style={styles.title}>{title}</Text>  
  </View>  
>  
);  
// ...
```

```
<SectionList
  sections={DATA} // 数据
  keyExtractor={({item, index}) => index} // 每一项的唯一索引
  renderItem={({item}) => <Item title={item} />} // 渲染项目的组件
  renderSectionHeader={({section: {title}}) => (
    <Text style={styles.header}>{title}</Text>
  )}
  ItemSeparatorComponent={() => {
    // 声明项目之间的分隔符
    return <View style={{borderBottomWidth:1,borderBottomColor: 'red'}}></View>
  }}
  ListEmptyComponent={() => {
    // 列表数据为空时，展示的组件
    return <Text style={{fontSize: 30}}>空空如也！</Text>
  }}
  // 下拉刷新
  refreshing={this.state.isFresh}
  onRefresh={this.loadData}

  // 上拉刷新
  onEndReachedThreshold={0.1} // 声明触底的比率，0.1表示距离底部还有10%
  onEndReached={() => {
    alert('到底了')
  }}
  ListHeaderComponent={() => {
    // 声明列表的头部组件
    return <Text style={{fontSize: 40}}>三国英雄榜</Text>
  }}
  ListFooterComponent={() => {
    // 声明列表的尾部组件
    return <Text style={{fontSize: 30}}>没有更多了</Text>
  }}
/>
```

效果如下



FlatList

FlatList 用来渲染列表。具有如下特点：

- 完全跨平台
- 支持垂直（默认）和水平两个方向的列表
- 可配置显示或隐藏的回调事件
- 支持自定义 Header
- 支持自定义 Footer
- 支持自定义行与行之间的分割线
- 下拉刷新
- 上拉刷新
- 支持跳到指定行
- 支持多列显示

代码演示

```
import React, { Component } from 'react'
import { Text, StyleSheet, View, FlatList, StatusBar, RefreshControl } from
'react-native'

export default class FlatListDemo extends Component {
```

```
constructor(props) {
  super(props)

  this.state = {
    isLoading: false,
    list: [
      {
        id: '1',
        title: 'Item 1',
      },
      {
        id: '2',
        title: 'Item 2',
      },
      {
        id: '3',
        title: 'Item 3',
      },
      {
        id: '4',
        title: 'Item 4',
      },
      {
        id: '5',
        title: 'Item 5',
      },
      {
        id: '6',
        title: 'Item 6',
      },
      {
        id: '7',
        title: 'Item 7',
      },
      {
        id: '8',
        title: 'Item 8',
      },
      {
        id: '9',
        title: 'Item 9',
      },
    ]
  }
}

renderItem = ({ index, item }) => {
  return (
    <View style={styles.item}>
      <Text style={styles.title}>{item.title}</Text>
    </View>
  );
}

reachBottom = () => {
  alert('到底了')
}
```

```
 loadData = () => {
  this.setState({
    isLoading: true
  })

  // 模拟网络请求
  setTimeout(() => {
    // 模拟请求数据
    alert('刷新请求数据')

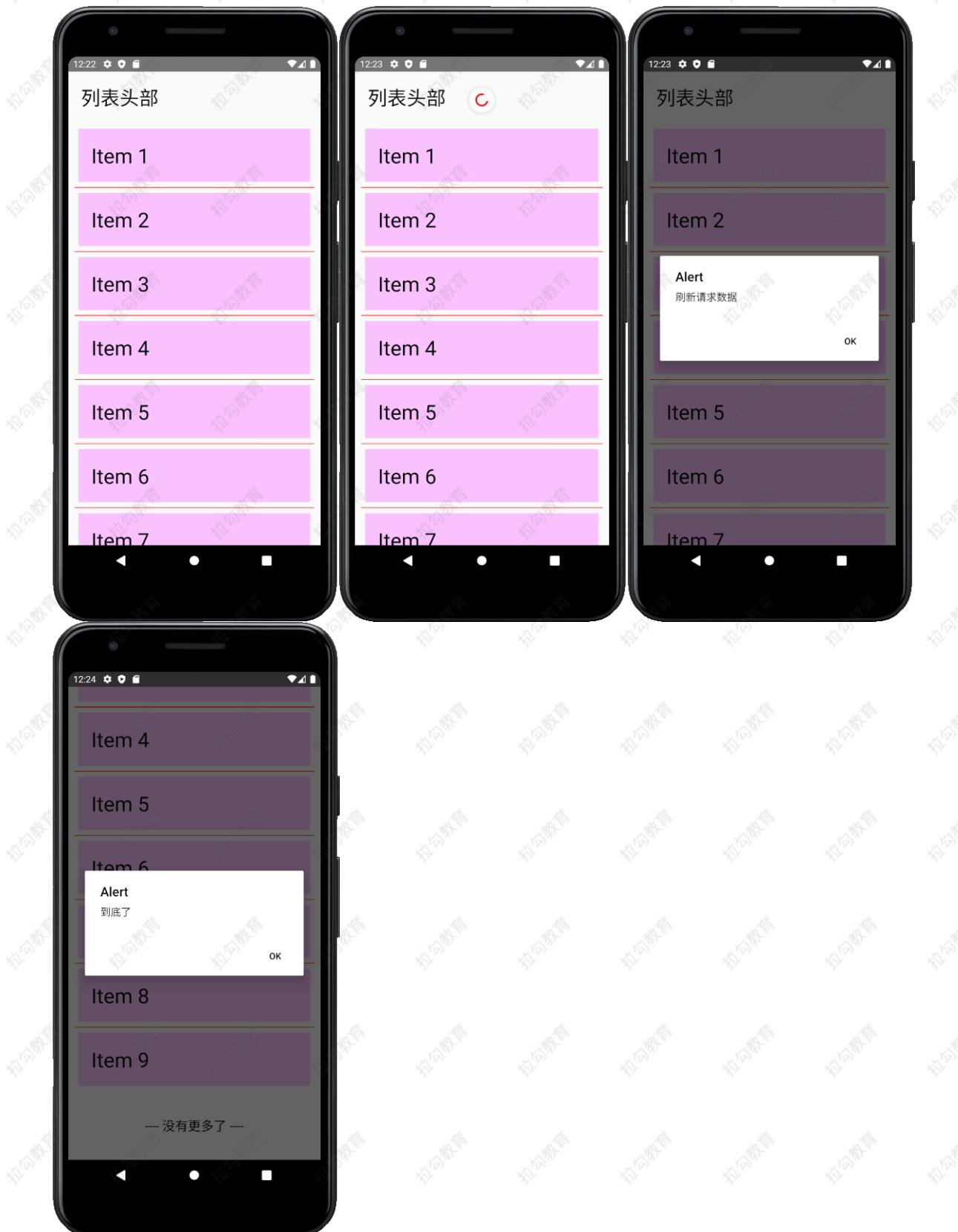
    this.setState({
      isLoading: false,
    })
  }, 2000);
}

render() {
  return (
    <FlatList
      ListHeaderComponent={() => { // 自定义头部
        return <View style={[styles.header]}>
          <Text style={{fontSize: 30}}>列表头部</Text>
        </View>
      }}
      // stickyHeaderIndices={[0]} // 置顶索引
      initialScrollIndex={0} // 初始索引
      data={this.state.list} // 数据
      renderItem={this.renderItem} // 渲染项目
      keyExtractor={item => item.id} // 唯一索引
      ItemSeparatorComponent={() => { // 分割线
        return <View style={[styles.separator]}></View>;
      }}
      onEndReached={() => { // 触底事件 (上拉刷新)
        this.reachBottom()
      }}
      onEndReachedThreshold={0.1} // 声明距离底部的比率
      horizontal={false} // 是否水平
      numColumns={1} // 列数
      // 下拉刷新
      refreshing={this.state.isLoading}
      onRefresh={() => {
        this.loadData(); // 下拉刷新加载数据
      }}
      // 设置刷新样式
      refreshControl={
        <RefreshControl
          title={"Loading"} // android中设置无效
          colors={[{"red"}]} // android
          tintColor={"red"} // ios
          titleColor={"red"}
          refreshing={this.state.isLoading}
          onRefresh={() => {
            this.loadData(); // 下拉刷新加载数据
          }}
        />
      }
      ListFooterComponent={() => { // 自定义尾部
        return <Text style={[styles.footer]}>---- 没有更多了 ----</Text>
      }}
    
```

```
        }
      />
    )
}
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: StatusBar.currentHeight || 0,
  },
  item: {
    backgroundColor: '#f9c2ff',
    padding: 20,
    marginVertical: 8,
    marginHorizontal: 16,
  },
  title: {
    fontsize: 32,
  },
  header: {
    fontsize: 20,
    margin: 20
  },
  footer: {
    fontsize: 20,
    textAlign: 'center',
    marginVertical: 40
  },
  separator: {
    borderBottomwidth: 1,
    borderBottomColor: 'red',
    marginHorizontal: 10
  }
});
```

效果图：



Animated

`Animated` 库旨在使动画变得流畅，强大并易于构建和维护。`Animated` 侧重于输入和输出之间的声明性关系，以及两者之间的可配置变换，此外还提供了简单的 `start/stop` 方法来控制基于时间的动画执行。

创建动画最基本的工作流程是：

- 先创建一个 `Animated.value`，将它连接到动画组件的一个或多个样式属性，

在 RN 中有**两种值**可以用于 Animated 中，分别是：

- Animated.value() -- 单个值
- Animated.valueXY() -- 向量值
- 然后使用 Animated.timing() 通过动画效果展示数据的变化。

Animated 提供了**三种动画类型**。

每种动画类型都提供了特定的函数曲线，用于控制动画值从初始值变化到最终值的变化过程：

- Animated.decay() **加速效果** - 以指定的初始速度开始变化，然后变化速度越来越慢直至停下。
- Animated.spring() **弹跳效果** - 提供一个简单的弹簧物理模型
- Animated.timing() **时间渐变效果** - 通过 easing 函数 在一定时间内，演绎一个动画

最常用的函数是 timing()

Animated 中默认导出了以下这些，可以直接使用的**动画组件**：

- Animated.Image
- Animated.ScrollView
- Animated.Text
- Animated.View

动画组合：动画还可以使用组合函数以复杂的方式进行组合：

- Animated.delay() 在给定延迟后开始动画。
- Animated.parallel() 同时启动多个动画。
- Animated.sequence() 按顺序启动动画，等待每一个动画完成后再开始下一个动画。
- Animated.stagger() 按照给定的延时间隔，顺序并行的启动动画。

引入

```
import { Animated, Text, View, StyleSheet, Button, Easing } from "react-native";
```

使用

```
const AnimatedDemo = () => {
  // fadeAnim 将会用作透明度的值。初始值为 0
  const fadeAnim = useRef(new Animated.Value(1)).current;

  const fadeIn = () => {
    // 在 1 到 5 秒内，改变 fadeAnim 的值
    Animated.timing(fadeAnim, {
      toValue: 1,
      duration: 5000,
      useNativeDriver: true,
    }).start();
  };

  const fadeOut = () => {
    // 在 0 到 5 秒内，改变 fadeAnim 的值
    Animated.timing(fadeAnim, {
```

```
        toValue: 0,
        duration: 5000,
        useNativeDriver: true,
    }).start();
};

return (
    <View style={styles.container}>
        <View>
            <Animated.View
                style={[ styles.fadingContainer,
                    {
                        opacity: fadeAnim // 将动画的值，绑定到透明度上
                    }
                ]}>
                <Text style={styles.fadingText}>淡入淡出的视图</Text>
            </Animated.View>
            <View style={styles.buttonRow}>
                <Button title="淡入" onPress={fadeIn} />
                <Button title="淡出" onPress={fadeOut} />
            </View>
        </View>
    </View>
);
}

export default AnimatedDemo;
```

WebView

[WebView](#) 相当于 RN 中的内置浏览器，我们写的 H5 的代码，可以直接在 WebView 中直接运行。该组件之前在 RN 核心中。现在已经单独维护了。

- 安装

```
yarn add react-native-webview
```

或

```
npm install --save react-native-webview
```

- 链接原生代码

React Native 模块包括 Objective-C, Swift, Java, or Kotlin 等原生代码，我们必须将其“链接”，然后，编译器才会在应用中使用。

```
react-native link react-native-webview
```

如果需要取消链接，请执行：react-native unlink react-native-webview

iOS 应用

如果你在 iOS 下，请在 `ios/` 或 `macos/` 目录下运行

```
cd ios && pod install && cd ../
```

Android 应用

如果 react-native-webview 的版本 < 6，则无需任何操作

如果 react-native-webview 的版本 >= 6，请确保 AndroidX 在项目中已启动。具体做法是在 **android/gradle.properties** 中添加如下两行

```
android.useAndroidX=true  
android.enableJetifier=true
```

上述链接操作完成后，我们就可以启动应用了

- iOS: yarn ios
- Android: yarn android

- 使用

直接填写网址

```
import React, { Component } from 'react';  
import { WebView } from 'react-native-webview';  
  
class MyWeb extends Component {  
  render() {  
    return (  
      <WebView  
        source={{ uri: 'https://infinite.red' }}  
        style={{ marginTop: 20 }}  
      />  
    );  
  }  
}
```

或者直接写 HTML 代码

```
import React, { Component } from 'react';  
import { WebView } from 'react-native-webview';  
  
class MyInlineweb extends Component {  
  render() {  
    return (  
      <WebView  
        originWhitelist={['*']}  
        source={{ html: '<h1>Hello world</h1>' }}  
      />  
    );  
  }  
}
```

Picker

[Picker](#) 相当于 HTML 中的下拉框。react-native@0.60.0 之前在 RN 核心中。现在已经单独维护了

- 安装

```
yarn add @react-native-picker/picker
```

在 iOS 下，还需要执行

```
pod-install
```

安卓没有额外操作

- 使用

```
import {Picker} from '@react-native-picker/picker';

// (...)

state = {
  gender: 0, // 在状态中指定性别
};

// (...)

<Picker
  selectedValue={this.state.gender} // 选中的值
  style={{height: 50, width: 100}}
  mode={"dialog"} // mode 属性只在 Android 下有效
  onChange={(itemValue, itemIndex) =>
    this.setState({gender: itemValue})
}>
  <Picker.Item label="保密" value="0" />
  <Picker.Item label="男" value="1" />
  <Picker.Item label="女" value="2" />
</Picker>
```

效果图：



Swiper

在 React Native 中，实现轮播图常用的组件是 [react-native-swiper](#)

- 安装

```
npm i react-native-swiper --save
```

- 使用

```
import React, { Component } from 'react'
import { StyleSheet, View, ScrollView, Image, Dimensions } from 'react-native'
import Swiper from 'react-native-swiper'

export default class SwiperDemo extends Component {
  render() {
    return (
      <View>
        <ScrollView>
          <Swiper style={[styles.wrapper]} showsButtons={true}>
            <Image
              source={require('../images/1.jpg')}
              style={[styles.slideImage]}
            />
            <Image
              source={require('../images/2.jpg')}
              style={[styles.slideImage]}
            />
            <Image
              source={require('../images/3.jpg')}
              style={[styles.slideImage]}
            />
          </Swiper>
        </ScrollView>
      </View>
    )
  }
}

const styles = StyleSheet.create({
  wrapper: {
    height: 200
  },
  slideImage: {
    width: Dimensions.get('window').width,
    height: 200
  }
})
```

注意：Swiper 要放在 ScrollView 组件中，否则显示不正常

AsyncStorage

[AsyncStorage](#) 是一个简单的、异步的、持久化的 Key-Value 存储系统，它对于 App 来说是全局性的。可用来代替 localStorage。

我们推荐您在 AsyncStorage 的基础上做一层抽象封装，而不是直接使用 AsyncStorage。

在 iOS 上，[AsyncStorage](#) 在原生端的实现是把较小值存放在序列化的字典中，而把较大值写入单独的文件。在 Android 上，[AsyncStorage](#) 会尝试使用 [RocksDB](#)，或退而选择 SQLite。

- 安装

```
yarn add @react-native-async-storage/async-storage
```

- 链接

在使用之前，我们需要将 AsyncStorage 链接到应用

安卓环境

如果你的 React-native ≥ 0.60 ，启动项目时，启动命令会自动链接必要的模块，我们不需要任何操作。

如果你的 React Native ≤ 0.59 ，在启动项目之前，需要先运行下面的命令。

```
react-native link @react-native-async-storage/async-storage
```

如果以上自动链接无效，再考虑使用下面的手动链接。

1. 在 android/settings.gradle 文件的尾部，添加如下内容

```
include ':@react-native-async-storage'  
project(':@react-native-async-storage').projectDir = new  
File(rootProject.projectDir, '../node_modules/@react-native-  
async-storage/async-storage/android')
```

2. 在 android/app/build.gradle 中添加依赖

```
dependencies {  
    ...  
    implementation project(':@react-native-async-storage')  
}
```

3. 在 android/app/src/main/java/your/package/MainApplication.java 中，添加如下代码

```
package com.myapp;  
  
import com.reactnativecommunity.asyncstorage.AsyncStoragePackage; // 添加  
这一行  
...  
  
@Override  
protected List<ReactPackage> getPackages() {  
    return Arrays.<ReactPackage>asList(  
        new MainReactPackage(),  
        new AsyncStoragePackage() // 添加这一行  
    );  
}
```

4. 启动安卓应用

```
yarn android
```

iOS 环境

1. 链接 AsyncStorage

```
cd ios && pod install && cd ../
```

2. 启动应用

```
yarn ios
```

- 使用

使用之前先引入组件

```
import AsyncStorage from '@react-native-async-storage/async-storage';
```

使用过程中，我们主要掌握 **增查删改** 四种操作

增：添加数据

查：获取数据

删：删除数据

改：修改，合并数据

- 添加数据

- 添加一条数据

添加数据我们使用 `setItem()`

如果给定的 key **不存在**，则 `setItem(key, value)` 用来**添加**数据；

如果给定的 key **已存在**，则 `setItem(key, value)` 用来**更新**数据；

- 字符串

```
const storeData = async (value) => {
  try {
    await AsyncStorage.setItem('@storage_Key', value)
  } catch (e) {
    // saving error
  }
}
```

- 对象

`AsyncStorage` 中只能存储字符串。所以，想要存储对象，需要通过 `JSON.stringify` 将其转成字符串。然后，在按照字符串的方式存储。

```
const storeData = async (value) => {
  try {
    const jsonValue = JSON.stringify(value)
    await AsyncStorage.setItem('@storage_Key', jsonValue)
  } catch (e) {
    // saving error
  }
}
```

- 添加多条数据

```
multiset = async () => {
  const firstPair = ["@MyApp_user", "value_1"]
  const secondPair = ["@MyApp_key", "value_2"]
  try {
    await AsyncStorage.multiset([firstPair, secondPair])
  } catch(e) {
    //save error
  }

  console.log("Done.")
}
```

- 获取数据

- 获取一条数据

- 字符串

```
const getData = async () => {
  try {
    const value = await AsyncStorage.getItem('@storage_Key')
    if(value !== null) {
      // value previously stored
    }
  } catch(e) {
    // error reading value
  }
}
```

- 对象

对象数据获取后是字符串，需要调用 JSON.parse 解析成对象。

```
const getData = async () => {
  try {
    const jsonvalue = await
    AsyncStorage.getItem('@storage_Key')
    return jsonvalue != null ? JSON.parse(jsonvalue) : null;
  } catch(e) {
    // error reading value
  }
}
```

- 获取多条数据

```
getMultiple = async () => {
  let values
  try {
    values = await AsyncStorage.multiGet(['@MyApp_user',
    '@MyApp_key'])
  } catch(e) {
    // read error
  }
  console.log(values)

  // example console.log output:
  // [ ['@MyApp_user', 'myUserValue'], ['@MyApp_key', 'myKeyValue'] ]
}
}
```

■ 获取所有数据

```
getAllKeys = async () => {
  let keys = []
  try {
    keys = await AsyncStorage.getAllKeys()
  } catch(e) {
    // read key error
  }

  console.log(keys)
  // example console.log result:
  // ['@MyApp_user', '@MyApp_key']
}
}
```

○ 删除数据

■ 删除一条数据

```
removeValue = async () => {
  try {
    await AsyncStorage.removeItem('@MyApp_key')
  } catch(e) {
    // remove error
  }

  console.log('Done.')
}
}
```

■ 删除多条数据

```
removeFew = async () => {
  const keys = ['@MyApp_USER_1', '@MyApp_USER_2']
  try {
    await AsyncStorage.multiRemove(keys)
  } catch(e) {
    // remove error
  }

  console.log('Done')
}
```

- 删除所有数据 (清空)

```
removeFew = async () => {
  const keys = ['@MyApp_USER_1', '@MyApp_USER_2']
  try {
    await AsyncStorage.multiRemove(keys)
  } catch(e) {
    // remove error
  }

  console.log('Done')
}
```

- 合并数据

- 合并一条数据

```
const USER_1 = {
  name: 'Tom',
  age: 20,
  traits: {
    hair: 'black',
    eyes: 'blue'
  }
}

const USER_2 = {
  name: 'Sarah',
  age: 21,
  hobby: 'cars',
  traits: {
    eyes: 'green'
  }
}

mergeUsers = async () => {
  try {
    //save first user
    await AsyncStorage.setItem('@MyApp_user',
    JSON.stringify(USER_1))

    // merge USER_2 into saved USER_1
    await AsyncStorage.mergeItem('@MyApp_user',
    JSON.stringify(USER_2))
  }
}
```

```
// read merged item
const currentUser = await AsyncStorage.getItem('@MyApp_user')

console.log(currentUser)

// console.log result:
// {
//   name: 'Sarah',
//   age: 21,
//   traits: {
//     eyes: 'green',
//     hair: 'black'
//   }
// }
```

■ 合并多条数据

```
const USER_1 = {
  name: 'Tom',
  age: 30,
  traits: {hair: 'brown'},
};

const USER_1_DELTA = {
  age: 31,
  traits: {eyes: 'blue'},
};

const USER_2 = {
  name: 'Sarah',
  age: 25,
  traits: {hair: 'black'},
};

const USER_2_DELTA = {
  age: 26,
  traits: {hair: 'green'},
};

const multiSet = [
  ["@MyApp_USER_1", JSON.stringify(USER_1)],
  ["@MyApp_USER_2", JSON.stringify(USER_2)]
]

const multiMerge = [
  ["@MyApp_USER_1", JSON.stringify(USER_1_DELTA)],
  ["@MyApp_USER_2", JSON.stringify(USER_2_DELTA)]
]

mergeMultiple = async () => {
  let currentlyMerged
```

```
try {
    await AsyncStorage.multiSet(multiSet)
    await AsyncStorage.multiMerge(multiMerge)
    currentlyMerged = await AsyncStorage.multiGet(['@MyApp_USER_1',
    '@MyApp_USER_2'])
} catch(e) {
    // error
}

console.log(currentlyMerged)
// console.log output:
// [
//   [
//     'USER_1',
//     {
//       name: "Tom",
//       age: 30,
//       traits: {
//         hair: 'brown'
//         eyes: 'blue'
//       }
//     }
//   ],
//   [
//     'USER_2',
//     {
//       name: 'Sarah',
//       age: 26,
//       traits: {
//         hair: 'green'
//       }
//     }
//   ]
// ]
```

- 代码实践

```
import React, { PureComponent } from 'react'
import { Text, View, TouchableHighlight } from 'react-native'
import AsyncStorage from '@react-native-async-storage/async-storage';

export default class App extends PureComponent {
  componentDidMount() {
    // 添加数据
    AsyncStorage.setItem('location', 'beijing')
  }

  getLocation = async () => {
    try {
      // 获取数据
      const location = await AsyncStorage.getItem('location')
      if (location !== null) {
        alert(location)
      }
    } catch(e) {
      // error reading value
    }
  }
}
```

```
        }
    }

    render() {
        return (
            <View style={{flex:1, marginTop: 100, alignItems: 'center'}}>
                <TouchableHighlight onPress={() => {
                    this.getLocation()
                }}>
                    <Text style={{fontSize: 30}}>点击获取数据</Text>
                </TouchableHighlight>
            </View>
        )
    }
}
```

效果图：



Geolocation

[Geolocation](#) 是 React Native 中用来定位的组件。

1. 安装

```
yarn add @react-native-community/geolocation
```

或

```
npm install @react-native-community/geolocation --save
```

2. 配置

- Android

在 `android/app/src/main/AndroidManifest.xml` 下添加允许授权的配置。

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION">
```

- iOS

- 对于 iOS 10+ 的环境，需要在 `ios/项目名称/Info.plist` 中添加 `NSLocationWhenInUseUsageDescription` 和 `NSLocationAlwaysAndWhenInUseUsageDescription` 等允许授权的配置。如果你的 iOS 小于 10，还需要添加 `NSLocationAlwaysUsageDescription`

如下图：

```
<plist version="1.0">
<dict>
  ...
  <key>NSLocationwhenInUseUsageDescription</key>
  <string>$(PRODUCT_NAME) use location once</string>
  <key>NSLocationAlwaysAndwhenInUseUsageDescription</key>
  <string>$(PRODUCT_NAME) always use location</string>
  <key>NSLocationAlwaysUsageDescription</key>
  <string>$(PRODUCT_NAME) always use location for ios10 and earlier</string>
</dict>
</plist>
```

如果上述配没有效果，请参考 [官方文档](#)

3. 使用

```
import Geolocation from '@react-native-community/geolocation';

Geolocation.getCurrentPosition(info => console.log(info));
```

返回结果

```
{  
  "coords": {  
    "accuracy": 70.6677474975586,  
    "altitude": 0,  
    "heading": 0,  
    "latitude": 39.981797, // 纬度  
    "longitude": 116.300916, // 经度  
    "speed": 0  
  },  
  "mocked": false,  
  "timestamp": 1606213656508  
}
```

Camera

[react-native-camera](#) 是 React Native 中调用摄像头的模块。

- 安装

```
npm install react-native-camera --save
```

- 链接

- RN > 0.60

```
cd ios && pod install && cd ..
```

- RN < 0.60

```
react-native link react-native-camera
```

- 配置

- 安卓环境

在 android/app/src/main/AndroidManifest.xml 文件下，添加权限相关设置

```
<!-- 必填项 -->  
<uses-permission android:name="android.permission.CAMERA" />  
  
<!-- 相册设置（非必要，不添加） -->  
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"  
/>  
<uses-permission  
  android:name="android.permission.WRITE_EXTERNAL_STORAGE" />  
  
<!-- 视频录制设置（非必要，不添加） -->  
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
```

在 android/settings.gradle 下，添加：

```
include ':react-native-camera'  
project(':react-native-camera').projectDir = new  
File(rootProject.projectDir, '../node_modules/react-native-  
camera/android')
```

在 `android/app/build.gradle` 下，添加设置

```
android {  
    ...  
    defaultConfig {  
        ...  
        missingDimensionStrategy 'react-native-camera', 'general' // <-- 添加  
这一行  
    }  
}  
  
dependencies {  
    ...  
    implementation project(':react-native-camera') // <-- 添加这一行  
}
```

◦ iOS 环境

在 `ios/项目名称/Info.plist` 文件中添加允许权限的配置

```
<!-- iOS 10+ -->  
<key>NSCameraUsageDescription</key>  
<string>第一次访问摄像头时，展示给用户的消息</string>  
  
<!-- 允许调用相册 iOS 11+ -->  
<key>NSPhotoLibraryAddUsageDescription</key>  
<string>第一次访问相册时，展示给用户的消息</string>  
  
<!-- 允许调用相册 -->  
<key>NSPhotoLibraryUsageDescription</key>  
<string>第一次访问相册时，展示给用户的消息</string>  
  
<!-- 允许使用麦克风录制视频 -->  
<key>NSMicrophoneUsageDescription</key>  
<string>第一次调用麦克风时，展示给用户的消息</string>
```

在 `ios/Podfile` 中添加如下代码

```
pod 'react-native-camera', path: '../node_modules/react-native-camera',  
subspecs: [  
    'BarcodeDetectorMLKit'  
]
```

如果想要使用人脸识别等功能，还需要单独安装 `Firebase`，请参考官方文档

安装依赖

```
cd ios && pod install && cd ..
```

如果上述配置不起作用，请参考 [官方文档](#) 进行手动配置

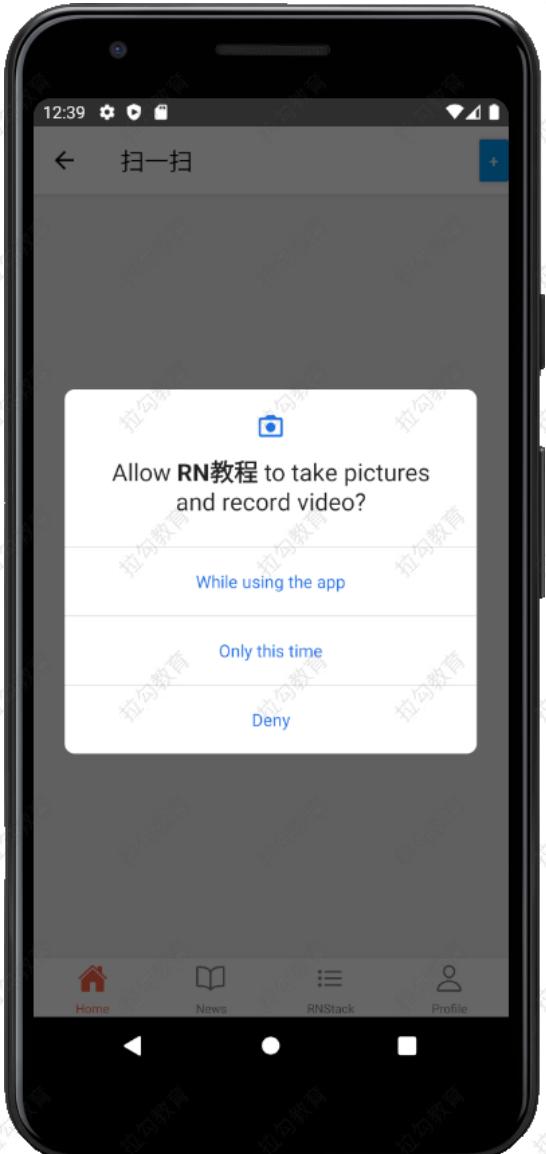
- 使用

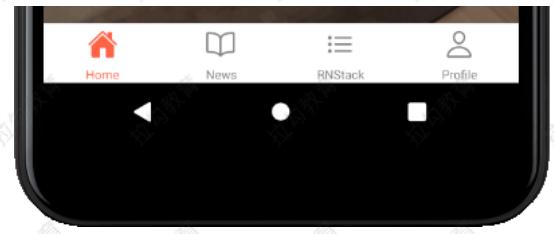
这里仅验证，扫码功能

```
import {RNCamera} from 'react-native-camera';

// .....

<RNCamera
  style={styles.preview}
  ratio={'16:9'}
  defaultVideoQuality={RNCamera.Constants.VideoQuality['720p']}
  scanAreaLimit={true}
  scanAreaX={(115 * winwidth) / 750}
  scanAreaY={(328 * winHeight) / 1334}
  scanAreaWidth={(522 * winwidth) / 750}
  scanAreaHeight={(521 * winHeight) / 1334}
  flashMode={
    this.state.torchState == 'off'
    ? RNCamera.Constants.FlashMode.off
    : RNCamera.Constants.FlashMode.torch
  }
  barCodeTypes={[
    RNCamera.Constants.BarCodeType.qr,
    RNCamera.Constants.BarCodeType.code39
  ]}
  onBarcodeRead={this._onBarcodeRead.bind(this)}
>
  <View style={styles.rectangleContainer}>
    <View style={styles.rectangle} />
    <Animated.View
      style={[
        styles.border,
        {
          transform: [
            {
              translateY: this.moveAnim.interpolate({
                inputRange: [0, 1],
                outputRange: [-200, 0],
              }),
            ],
          ],
        },
      ]}
    />
    <Text style={styles.rectangleText}>
      将二维码放入框内，即可自动扫描
    </Text>
  </View>
</RNCamera>
```





ImagePicker

[react-native-image-picker](#) 允许我们从设备中选择图片，或直接通过摄像头拍摄。

1. 安装

[官方安装文档](#)

```
yarn add react-native-image-picker  
  
# RN >= 0.60  
npx pod-install  
  
# RN < 0.60  
react-native link react-native-image-picker
```

2. 配置环境

1. Android

- 在 android/app/src/AndroidManifest.xml 下添加允许授权的配置（**如果之前已经设置了，请忽略**）

```
<uses-permission android:name="android.permission.CAMERA" />  
<uses-permission  
    android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

2. iOS

- 对于 iOS 10+ 的环境，需要在 **ios/项目名称/Info.plist** 中添加 NSPhotoLibraryUsageDescription , NSCameraUsageDescription , NSPhotoLibraryAddUsageDescription and NSMicrophoneUsageDescription` (如果允许视频) 等允许授权的配置

如下图：

```
<plist version="1.0">
<dict>
    ...
    <key>NSPhotoLibraryUsageDescription</key>
    <string>$(PRODUCT_NAME) would like access to your photo
gallery</string>
    <key>NSCameraUsageDescription</key>
    <string>$(PRODUCT_NAME) would like to use your camera</string>
    <key>NSPhotoLibraryAddUsageDescription</key>
    <string>$(PRODUCT_NAME) would like to save photos to your photo
gallery</string>
    <key>NSMicrophoneUsageDescription</key>
    <string>$(PRODUCT_NAME) would like to use your microphone (for
videos)</string>
</dict>
</plist>
```

如果上述配置不起作用, 请参考 [官方文档](#) 的手动配置,

3. 使用

react-native-image-picker 中有三个方法

- showImagePicker(options, callback)
 - 弹出一个对话框, 提供两个选项 (1. 去拍照, 2.在相册中选取图片)
- launchCamera(options, callback)
 - 直接调用摄像头去拍照
- launchImageLibrary(options, callback)
 - 直接访问本地相册

下面, 以 showImagePicker 为例, 演示 react-native-image-picker 的执行效果

```
import React, { Component } from 'react'
import { Text, StyleSheet, View, TouchableOpacity, Image } from 'react-
native'
import ImagePicker from 'react-native-image-picker';

const options = {
  title: '选择头像',
  storageOptions: {
    skipBackup: true,
    path: 'images',
  },
  cancelButtonTitle: '取消',
  takePhotoButtonTitle: '去拍照',
  chooseFromLibraryButtonTitle: '从手机相册中选择'
};

export default class Avatar extends Component {
  constructor(props) {
    super(props)

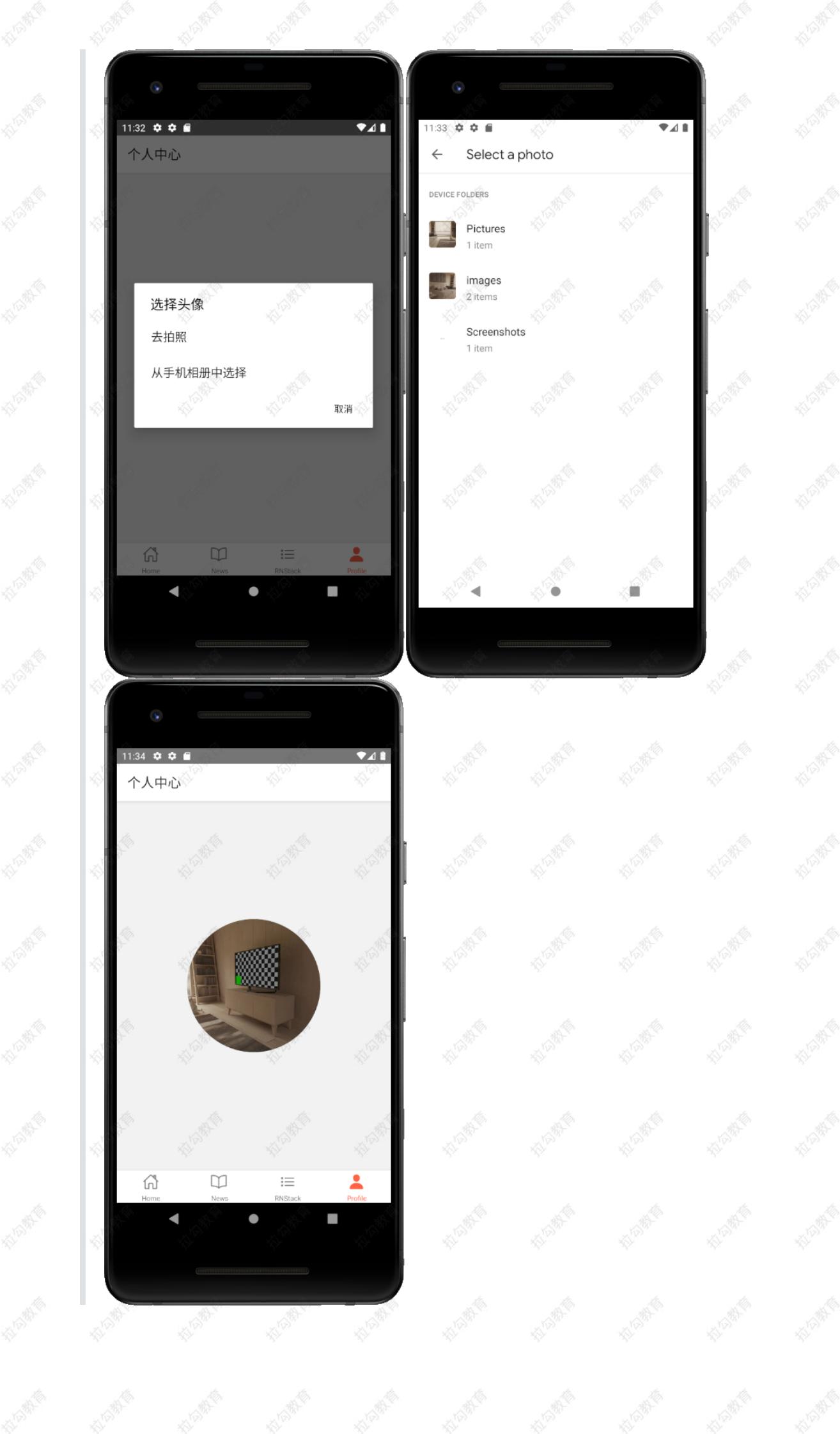
    this.state = {
      avatar: ''
    }
  }
```

```
handleSaveImg = () => {
  // 弹出窗口（去拍照 | 在相册中选择）
  ImagePicker.showImagePicker(options, (response) => {
    if (response.didCancel) {
      console.log('User cancelled video picker');
    } else if (response.error) {
      console.log('ImagePicker Error: ', response.error);
    } else if (response.customButton) {
      console.log('User tapped custom button: ', response.customButton);
    } else {
      const { uri } = response;
      this.setState({
        avatar: uri
      });
    }
  });
}

render() {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <TouchableOpacity onPress={() => this.handleSaveImg()}>
        <View
          style={{width: 200, height: 200}}
          {this.state.avatar === '' ? (
            <Text style={{width: 200,height: 200,borderRadius: 100}}>点击上传</Text>
          ) : (
            <Image style={{width: 200,height: 200,borderRadius: 100}}
              source={{ uri: this.state.avatar }} />
          )}
        </View>
      </TouchableOpacity>
    </View>
  );
}

```

效果图



其他

AppRegistry

AppRegistry 是所有 React Native 应用的 JS 入口。应用的根组件应当通过 `AppRegistry.registerComponent` 方法注册自己，然后原生系统才可以加载应用的代码包并且在启动完成之后通过调用 `AppRegistry.runApplication` 来真正运行应用。

```
import { Text, AppRegistry } from 'react-native';

const App = (props) => (
  <View>
    <Text>App1</Text>
  </View>
);

AppRegistry.registerComponent('Appname', () => App);
```

Dimensions

本模块用于获取设备屏幕的宽高

```
import { Dimensions } from 'react-native'

Dimensions.get('window').width
Dimensions.get('window').height
```

获取屏幕尺寸之后，我们可以根据屏幕尺寸做一些适配工作。例如：将屏幕三等分。

```
width: Dimensions.get('window').width / 3
```

Share

Share 模块用来完成分享功能

```
import React, { Component } from 'react';
import { Share, View, Button } from 'react-native';

class ShareExample extends Component {
  onShare = async () => {
    try {
      const result = await Share.share({
        message:
          'React Native | A framework for building native apps using React',
    });

      if (result.action === Share.sharedAction) {
        if (result.activityType) {
          // shared with activity type of result.activityType
        } else {
          // shared
        }
      }
    }
  }
}
```

```
        }
    } else if (result.action === Share.dismissedAction) {
        // dismissed
    }
} catch (error) {
    alert(error.message);
}
};

render() {
    return (
        <View style={{ marginTop: 50 }}>
            <Button onPress={this.onShare} title="Share" />
        </View>
    );
}
}

export default ShareExample;
```

Platform

Platform 模块可以区分平台，来完成针对平台的定制代码。例如：Platform.OS 会返回 ios 或 android

```
import { Platform, StyleSheet } from 'react-native';

const styles = StyleSheet.create({
    height: Platform.OS === 'ios' ? 200 : 100
});
```

路由与导航

简介

在 React 中实现路由，有两种方案，分别对应两个不同的场景

- React-Router
React-Router 适用于 Web 项目。
- React-Navigation

React Native 没有像浏览器那样的内置导航 API (location对象, history对象)。React Native 中的导航是通过 [React-Navigation](#) 来完成的。React Native 把导航和路由都集中到了 React-Navigation 中。

RN 0.44 之前，React-Navigation 在核心中维护，0.44 之后，与核心分离，独立维护。

官网: <https://reactnavigation.org/>

中文手册: <https://reactnavigation.org/docs/zh-Hans/getting-started.html>

注意：本节使用 React-Navigation 5.x 的版本来讲解

React-Navigation 常用的组件有四个（之后我们会依次介绍）

- `StackNavigator`

栈导航器在 React Native 中的作用，相当于 BOM 中的 history 对象。用来跳转页面和传递参数。与浏览器端导航不同的是。`StackNavigator` 还提供了路由之间的 **手势** 和 **动画**。

只有声明了栈导航之后，才能在 React Native 中执行跳转。

- `TabNavigator`

标签导航器（例如：底部标签栏），用来区分模块。

- `DrawerNavigator`

抽屉导航器，在 App 侧面划出的导航页面。

- `MaterialTopTabNavigator`

支持左右滑动的 Tab 菜单

基础安装

在正式开始学习 React Native 路由之前，我们需要先安装相关的组件和依赖

1. 安装组件

```
# 安装 react-navigation 核心组件  
yarn add @react-navigation/native
```

```
# 安装相关的依赖  
yarn add react-native-reanimated react-native-gesture-handler react-native-screens react-native-safe-area-context @react-native-community/masked-view
```

2. 链接组件

安装完成之后，我们还需要将相关组件和依赖，连接到操作系统平台中（Android 或 iOS）。

从 0.60 之后，[链接会自动执行](#)。因此，我们不再需要运行 `react-native link`

但是，**如果你是在 iOS 下，还需要运行下面的命令来完成链接**

```
npx pod-install ios
```

3. 添加头部组件

最后一步，你需要将如下代码，放到应用的头部（例如：放到 `index.js` 或 `App.js` 文件的头部）

```
import 'react-native-gesture-handler';  
// 其他引入
```

注意：如果你忽略了这一步，你的应用上线后可能会崩溃（虽然开发环境一切正常）

4. 添加导航容器

我们需要在入口文件中，把整个应用，包裹在导航容器（**NavigationContainer**）中（例如：在 `index.js` 或 `App.js` 文件中）。然后将其他应用代码，写在 `NavigationContainer` 组件中间。

```
import 'react-native-gesture-handler';
import * as React from 'react';
import { NavigationContainer } from '@react-navigation/native';

export default function App() {
  return (
    <NavigationContainer>
      {/* 具体的导航 */}
    </NavigationContainer>
  );
}
```

5. 使用具体的导航

完成以上 4 个步骤后，就可以在导航容器中使用具体的导航了。下面我们会一一介绍具体的导航。
主要包括：

- Stack 导航
- BottomTab 导航
- Drawer 导航
- MaterialTopTab 导航

Stack 导航

在浏览器中，我们可以通过 `标签`，来实现不同页面之间的跳转。当用户点击链接时，URL 地址会被推送到 `history` 的栈中。当用户点击回退按钮时，浏览器会从 `history` 栈的顶部弹出一项，然后我们所处的 **当前页面**，其实就是之前访问过的页面。但是，RN 中没有浏览器的内置 `history` 栈。而 React Navigation 的 **Stack 导航实现了类似浏览器端 history 栈的功能。可以在 RN 的不用屏幕之间进行跳转，并管理跳转地址。**

在 RN 中，如果想做跳转。必须先声明 Stack 导航

1. 安装组件

```
yarn add @react-navigation/stack
```

或

```
npm install @react-navigation/stack
```

2. 使用组件

```
import * as React from 'react';
import { View, Text, Button } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';

function HomeScreen({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text style={{ fontSize: 40 }}>Home Screen</Text>
      <Button onPress={() => navigation.navigate('Details')} title="跳转到详情页" />
    </View>
  );
}

const Stack = createStackNavigator();

function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="Details" component={DetailsScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}

export default App;
```

```
</view>
);
}

function DetailsScreen({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text style={{fontSize: 40}}>Details Screen</Text>
      <Button onPress={() => navigation.navigate('Home')} title="回首页" />
    </View>
  );
}

const Stack = createStackNavigator();

function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="Details">
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="Details" component={DetailsScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}

export default App;
```

我们可以通过 Screen 组件来声明路由。

Screen 组件有两个必选属性，分别是 name 和 component。

- name 是路由名称
- component 是组件名称（**不接受函数**）

路由声明之后，我们可以通过 navigate 方法来执行屏幕间的跳转。如下图：

类组件可以通过 **this.props.navigation.navigate(路由名称)** 方式来跳转

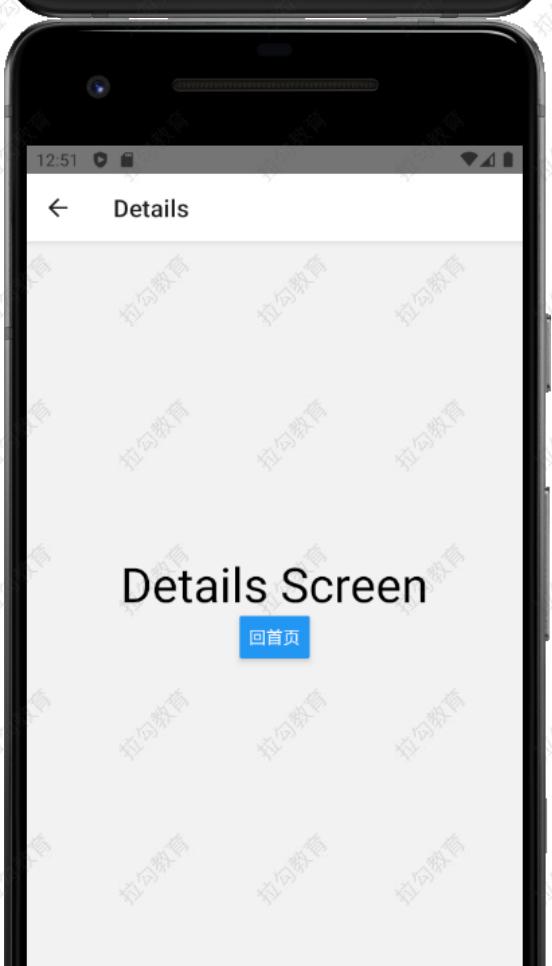
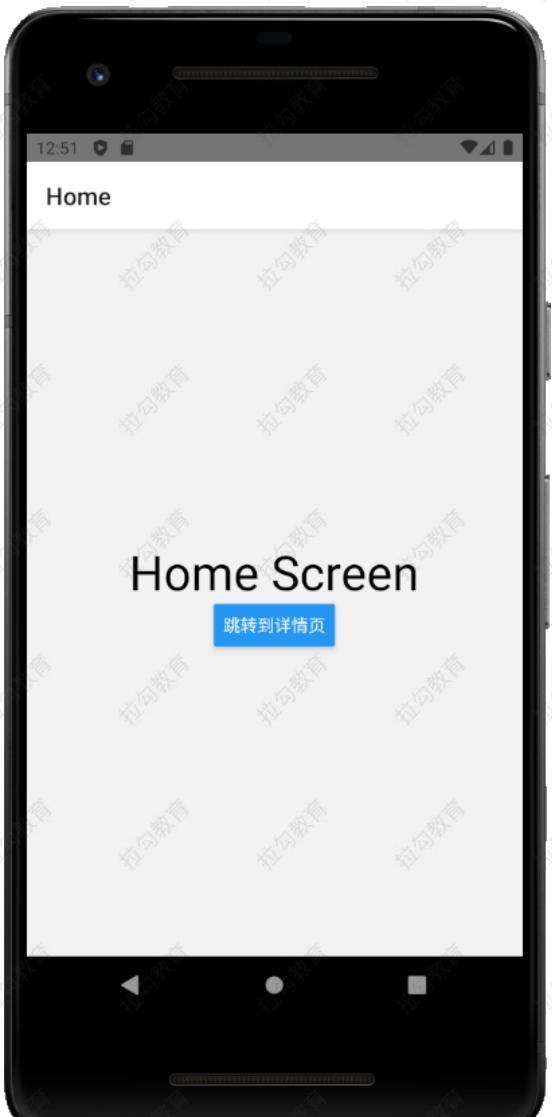
```
function HomeScreen({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text style={{fontSize: 40}}>Home Screen</Text>
      <Button onPress={() => navigation.navigate('Details')} title="跳转到详情页" />
    </View>
  );
}

function DetailsScreen({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text style={{fontSize: 40}}>Details Screen</Text>
      <Button onPress={() => navigation.navigate('Home')} title="回首页" />
    </View>
  );
}

const Stack = createStackNavigator();

function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="Det">
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="Details" component={DetailsScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

界面效果





3. 扩展属性

- Navigator 扩展

- initialRouteName 导航初始化路由名称（第一个加载的路由）。注意：**initialRouteName 发生改变时，需要重新启动应用。RN 的热更新对 initialRouteName 不起作用。**

如下图，虽然 Details 路由声明在 Home 路由之后，但会第一个加载

```
function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="Details">
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="Details" component={Detailsscreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

- headerMode

- float: iOS 的通用模式
- screen: Android 的通用模式
- none: 隐藏 header (包括 screen 的header)

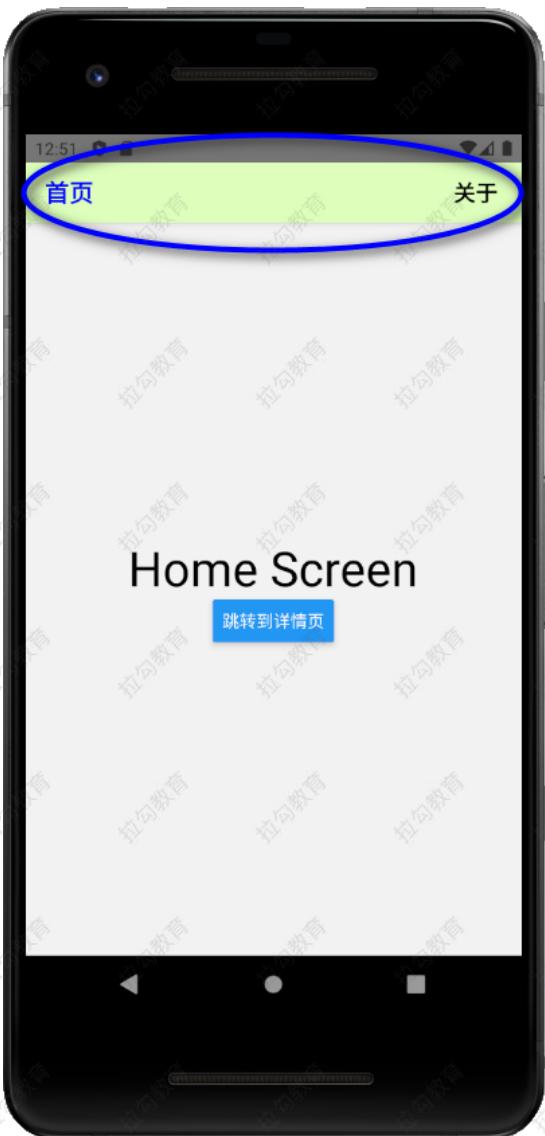
- Screen 扩展

- options

```
<Stack.Navigator initialRouteName="Home">
  <Stack.Screen
    name="Home"
    component={HomeScreen}
    options={{
      title: '首页', // 屏幕标题
      headerStyle: {
        backgroundColor: '#dfb', // 头部背景
        height: 50 // 头部高度
      },
      headerTitleStyle: { // 头部字体样式
        color: 'blue',
        fontsize: 20,
        fontWeight: 'bold',
      },
      headerShown: true, // 是否显示 header
      headerTitleAlign: "left", // header 标题排序方式 left | center
      // 设置头部右侧内容
      headerRight: () => (
        <TouchableOpacity onPress={() => alert('Hello')} >
          <Text
            style={{
              fontsize: 18,
              fontWeight: 'bold',
              color: 'black',
            }
          >Hello</Text>
        </TouchableOpacity>
      )
    }}
  </Stack.Screen>
</Stack.Navigator>
```

```
        marginRight: 20
    }
    >关于</Text>
</TouchableOpacity>
),
}
/>
<Stack.Screen name="Details" component={DetailsScreen} />
</Stack.Navigator>
```

options 选项，主要用来设置屏幕头部信息，例如：高度，颜色，字体大小等。效果如下图：



BottomTab 导航

1. 安装组件

```
yarn add @react-navigation/bottom-tabs
```

或

```
npm install @react-navigation/bottom-tabs
```

2. 使用组件

```
import * as React from 'react';
import { Text, View } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';

function HomeScreen() {
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text>Home!</Text>
    </View>
  );
}

function SettingsScreen() {
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text>Settings!</Text>
    </View>
  );
}

const Tab = createBottomTabNavigator();

export default function App() {
  return (
    <NavigationContainer>
      <Tab.Navigator>
        <Tab.Screen name="Home" component={HomeScreen} />
        <Tab.Screen name="Settings" component={SettingsScreen} />
      </Tab.Navigator>
    </NavigationContainer>
  );
}
```

效果图：





3. 为 Tab 导航菜单，设置小图标

1. 安装图标组件

[React-native-vector-icons](#) 是著名的图标组件，包含了世界各大公司的矢量图标。使用之前先安装：

```
npm install --save react-native-vector-icons
```

如下所示，里面有十多种图标库。

例如：蚂蚁金服团队推出的 AntDesign 图标库，也有 Bootstrap 中使用的 FontAwesome 图标库。

- [AntDesign](#) 蚂蚁金服团队 (**297** icons)
- [Entypo](#) by Daniel Bruce (**411** icons)
- [EvilIcons](#) by Alexander Madyankin & Roman Shamin (v1.10.1, **70** icons)
- [Feather](#) by Cole Bemis & Contributors (v4.28.0, **285** icons)
- [FontAwesome](#) Bootstrap 使用的图标库 (v4.7.0, **675** icons)
- [FontAwesome 5](#) by Fonticons, Inc. (v5.13.0, 1588 (free) **7842** (pro) icons)
- [Fontisto](#) by Kenan Gündoğan (v3.0.4, **615** icons)
- [Foundation](#) by ZURB, Inc. (v3.0, **283** icons)
- [Ionicons](#) Ionic 框架中的图标 (v5.0.1, **1227** icons)
- [MaterialIcons](#) by Google, Inc. (v4.0.0, **1547** icons)
- [MaterialCommunityIcons](#) by MaterialDesignIcons.com (v5.3.45, **5346** icons)
- [Octicons](#) by Github, Inc. (v8.4.1, **184** icons)
- [Zocial](#) by Sam Collins (v1.0, **100** icons)
- [SimpleLineIcons](#) by Sabbir & Contributors (v2.4.1, **189** icons)

2. 将图标文件关联到应用

不同环境下的关联方式不同，详情查看：<https://github.com/oblador/react-native-vector-icons>

■ iOS

项目根目录下运行：react-native link react-native-vector-icons

如果不能正常运行，请参考 [官方文档](#)

■ Android

编辑 android/app/build.gradle (不是 android/build.gradle) 并添加如下内容：

```
apply from: "../../node_modules/react-native-vector-
icons/fonts.gradle"
```

然后重新运行项目。

3. 代码实现

```
// 引入图标组件
import Ionicons from 'react-native-vector-icons/Ionicons';

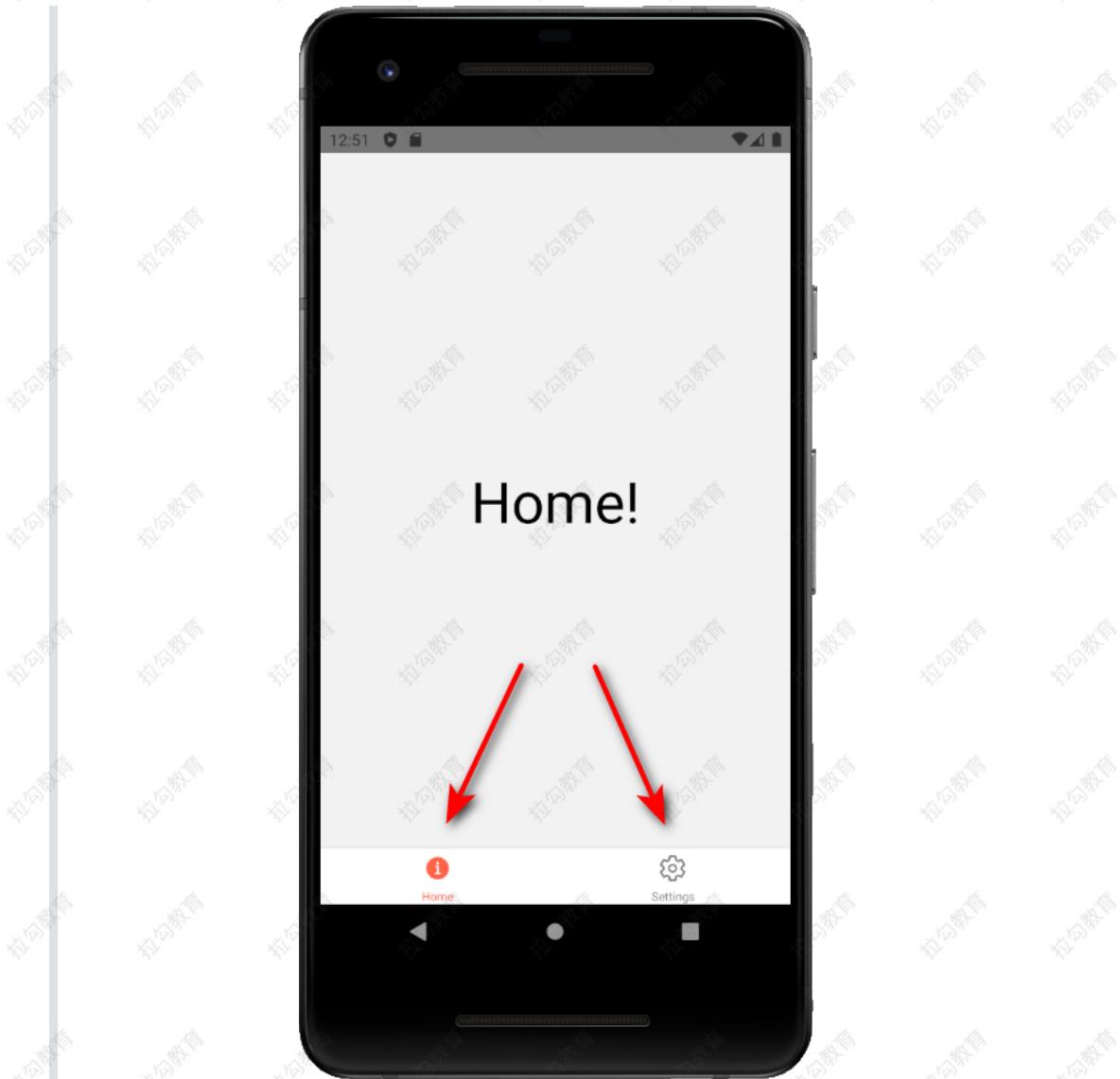
// (...)

export default function App() {
  return (
    <NavigationContainer>
      <Tab.Navigator
        screenOptions={({ route }) => ({
          tabBarIcon: ({ focused, color, size }) => {
            let iconName;

            if (route.name === 'Home') {
              iconName = focused
                ? 'ios-information-circle'
                : 'ios-information-circle-outline';
            } else if (route.name === 'Settings') {
              iconName = focused ? 'settings' : 'settings-outline';
            }

            // You can return any component that you like here!
            return <Ionicons name={iconName} size={size} color={color}>/>;
          },
        })}
        tabBarOptions={{
          activeTintColor: 'tomato',
          inactiveTintColor: 'gray',
        }}
      >
        <Tab.Screen name="Home" component={HomeScreen} />
        <Tab.Screen name="Settings" component={SettingssScreen} />
      </Tab.Navigator>
    </NavigationContainer>
  );
}
```

如下图，我们可以看到，在 Tab 菜单的文本上方，显示了图标：



Drawer 导航

1. 安装组件

```
yarn add @react-navigation/drawer
```

或

```
npm install @react-navigation/drawer
```

2. 使用组件

```
import * as React from 'react';
import { View, Text, Button } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createDrawerNavigator } from '@react-navigation/drawer';

function Feed({ navigation }) {
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text style={{ fontsize: 40 }}>Feed Screen</Text>
    </View>
  );
}

const App = () => {
  return (
    <NavigationContainer>
      <Drawer.Navigator>
        <Drawer.Screen name="Feed" component={Feed} />
        <Drawer.Screen name="Settings" component={Settings} />
      </Drawer.Navigator>
    </NavigationContainer>
  );
}

export default App;
```

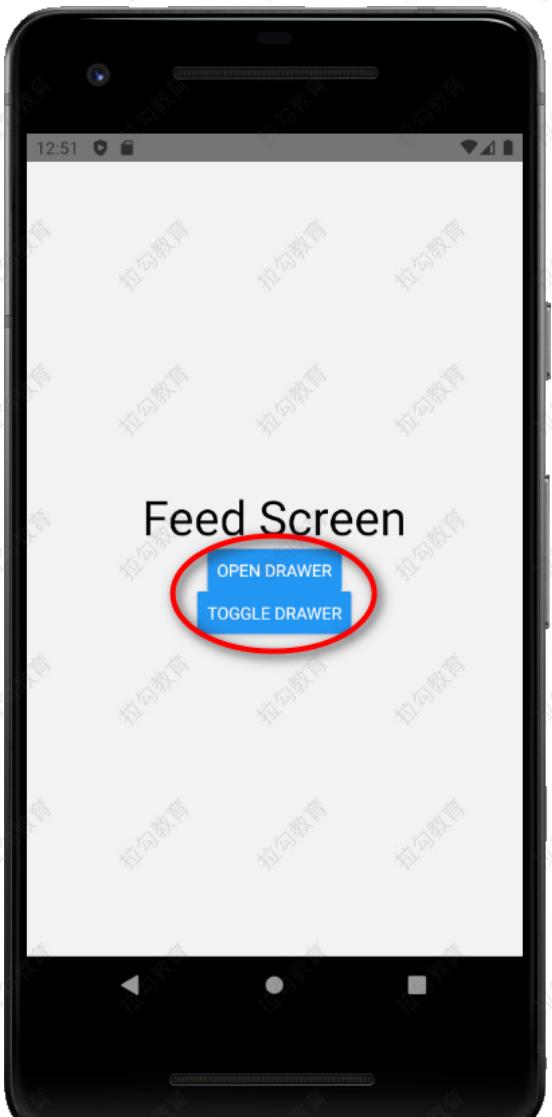
```
<Button title="Open drawer" onPress={() => navigation.openDrawer()} />
<Button title="Toggle drawer" onPress={() =>
navigation.toggleDrawer()} />
</View>
);
}

function Notifications() {
return (
<View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
<Text style={{fontSize: 40}}>Notifications Screen</Text>
</View>
);
}

const Drawer = createDrawerNavigator();

export default function App() {
return (
<NavigationContainer>
<Drawer.Navigator>
<Drawer.Screen name="Feed" component={Feed} />
<Drawer.Screen name="Notifications" component={Notifications} />
</Drawer.Navigator>
</NavigationContainer>
);
}
}
```

点击按钮，然后抽屉菜单就显示出来了。效果图：





3. 扩展属性

```
<Drawer.Navigator  
    drawerPosition={'right'} // 菜单右侧显示  
    drawerType="slide" // 设置抽屉菜单动画效果  
    drawerStyle={{  
        backgroundColor: '#cdb', // 设置抽屉菜单背景色  
        width: 180, // 设置抽屉菜单宽度  
    }}  
    drawerContentOptions={{  
        activeTintColor: '#e91e63', // 设置活跃字体颜色  
        itemStyle: { // 设置菜单项样式  
            marginVertical: 20 // 设置菜单的垂直外间距  
        },  
    }}  
>  
    <Drawer.Screen name="Feed" component={Feed} />  
    <Drawer.Screen  
        name="Notifications"  
        component={Notifications}  
        options={{  
            title: '通知' // 菜单标题  
        }}  
    />  
</Drawer.Navigator>
```



MaterialTopTab 导航

生成可以左右滑动的 Tab 导航

1. 安装

```
yarn add @react-navigation/material-top-tabs react-native-tab-view
```

或

```
npm install @react-navigation/material-top-tabs react-native-tab-view
```

2. 使用

```
import * as React from 'react';
import { Text, View } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createMaterialTopTabNavigator } from '@react-navigation/material-top-tabs';

function OrderUnpayScreen() {
  return (
    <View>
      <Text>OrderUnpay</Text>
    </View>
  );
}

const App = () => {
  return (
    <NavigationContainer>
      <MaterialTopTab.Navigator>
        <MaterialTopTab.Screen name="OrderUnpay" component={OrderUnpayScreen} />
        <MaterialTopTab.Screen name="OrderPay" component={OrderPayScreen} />
      </MaterialTopTab.Navigator>
    </NavigationContainer>
  );
}

export default App;
```

```
<view style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
    <Text style={{fontSize: 40}}>待付款!</Text>
</view>
);

}

function OrderPaidScreen() {
    return (
        <view style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
            <Text style={{fontSize: 40}}>待发货!</Text>
        </view>
    );
}

function OrderSentsScreen() {
    return (
        <view style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
            <Text style={{fontSize: 40}}>待收获!</Text>
        </view>
    );
}

function OrderFinishScreen() {
    return (
        <view style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
            <Text style={{fontSize: 40}}>待评价!</Text>
        </view>
    );
}

const Tab = createMaterialTopTabNavigator();

export default function App() {
    return (
        <NavigationContainer>
            <Tab.Navigator>
                <Tab.Screen
                    name="OrderUnpay"
                    component={OrderUnpayScreen}
                    options={{title: '待付款'}}
                />
                <Tab.Screen
                    name="OrderPaid"
                    component={OrderPaidScreen}
                    options={{title: "待发货"}}
                />
                <Tab.Screen
                    name="OrderSent"
                    component={OrderSentsScreen}
                    options={{title: "待收获"}}
                />
                <Tab.Screen
                    name="OrderFinish"
                    component={OrderFinishScreen}
                />
            </Tab.Navigator>
        </NavigationContainer>
    );
}
```

```
        options={{title:"待评价"}}
      />
    </Tab.Navigator>
  </NavigationContainer>
);
}
```

效果图（可以通过手指滑动，来切换屏幕）：







3. 属性配置

- Navigator 属性

- tabBarPosition

标签显示的位置，默认是 top，如果想把标签设置在底部，可以使用 bottom。

- tabBarOptions

包含 tabBar 组件属性的对象。

- activeTintColor - 当前标签的标签或图标颜色。
 - inactiveTintColor - 非当前标签的标签或图标颜色。
 - showIcon - 是否显示图标，默认是 false。
 - showLabel - 是否显示文字，默认是 true。
 - tabStyle - 标签样式对象。
 - labelStyle - 标签文字样式对象。这里指定的颜色，会覆盖 activeTintColor 和 inactiveTintColor 的值。
 - iconStyle - 图标样式对象。

```
<Tab.Navigator  
  tabBarPosition='bottom' // 显示位置 top | bottom  
  tabBarOptions={  
    labelStyle: { fontsize: 16 },  
    tabStyle: { // 标签样式  
      borderColor: '#dfb',  
      borderwidth: 1,  
    },  
    activeTintColor: 'red', // 当前标签的字体或图标颜色  
    inactiveTintColor: '#666', // 非当前标签的字体或图标颜色  
    showIcon: true, // 是否显示图标  
    showLabel: true, // 是否显示文字  
    style: { backgroundColor: 'powderblue' },  
  }  
>
```

- Screen 属性

- options

设置 Screen 组件的对象

- title - 设置标签文字
 - tabBarIcon - 设置标签图标。需要现在 Navigator 中指定 showIcon: true。

其值为函数，包含两个参数：{ focused: boolean, color: string }。

- focused 用来判断标签是否获取焦点，
 - color 为当前标签的颜色
 - tabBarLabel - 设置标签文字内容（当未定义时，会使用 title 字段）

其值为函数，包含两个参数：{ focused: boolean, color: string }。

```
<Tab.Screen  
  name="OrderPaid"  
  component={OrderPaidScreen}  
  options={{  
    title:"待发货",  
    tabBarIcon: ({focused, color}) => {  
      return (  
        <Ionicons name="arrow-redo-circle-outline" size={20} color=  
        {color} />  
      )  
    }  
  }}  
>
```

通过扩展属性，我们可以设置 tabBar 的位置，样式，图标等。如下图：



架构原理

为了更好的理解 React Native，我们需要了解 RN 的架构原理。这里主要介绍两个内容

- 现有架构

当前 RN 正在使用的架构

- 新架构

2018年6月，Facebook推出了RN的重构计划。我们需要了解下一代RN的架构原理。

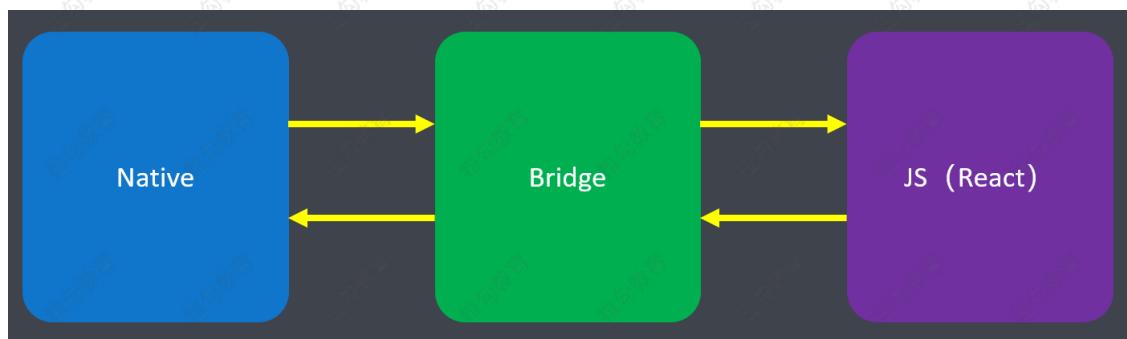
现有架构

架构模型

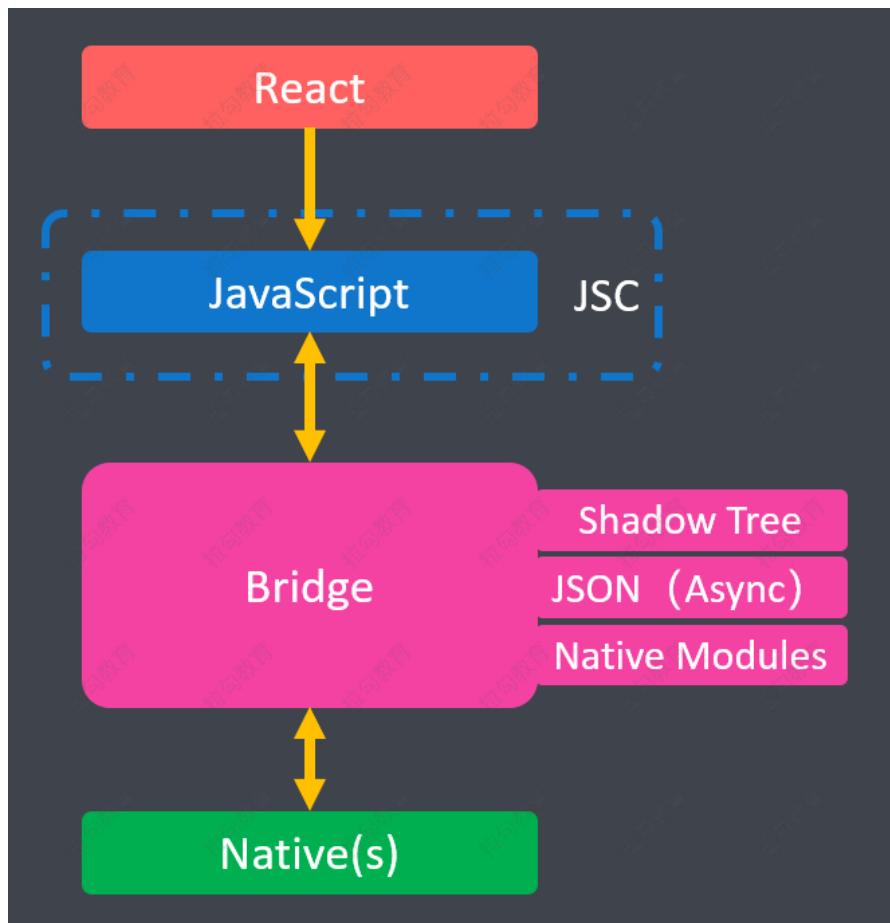
基本架构模型如下：

- Native是原生部分，例如：iOS原生或Android原生
- JS端主要是React语法
- Bridge用与Native和JS的通信

因为Native和JS相对独立。彼此通信是通过桥接器（Bridge）来实现。



详细一点的架构模型



- 最上层提供类React支持，运行在JSC提供的JavaScript运行时环境中
- Bridge层将JavaScript与Native世界连接起来。具体的，

- Shadow Tree 用来定义 UI 效果及交互功能，
- Native Modules 提供 Native 功能（比如蓝牙），
- 二者之间通过 JSON 消息相互通信

线程模型

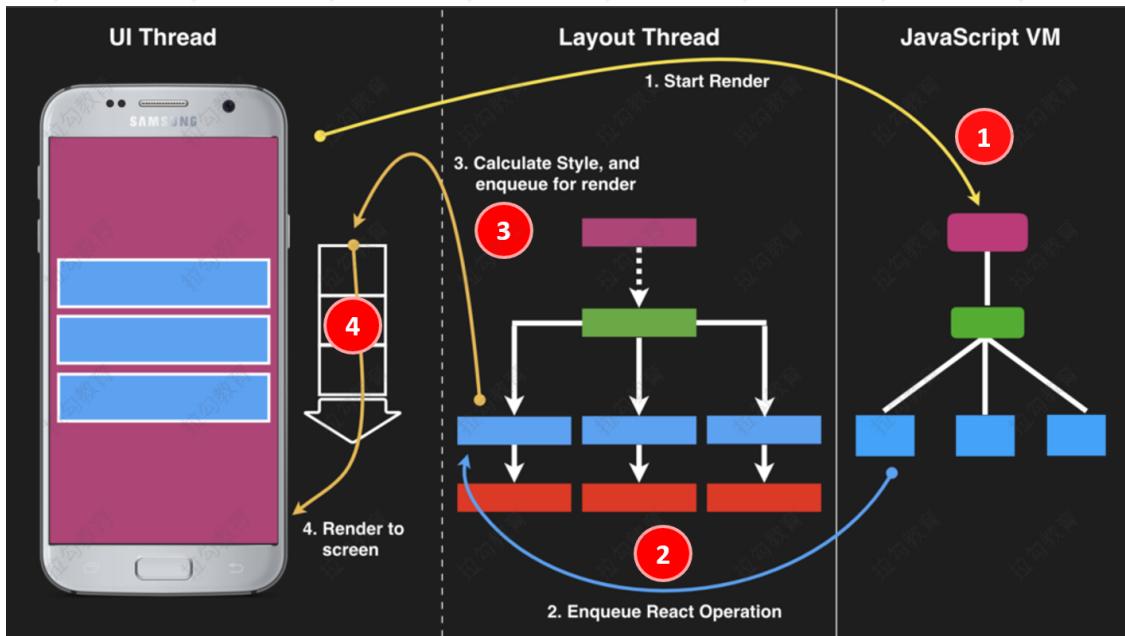
- JS 线程
 - JS 代码的执行线程，将源码通过 Metro 打包后，传给 JS 引擎进行解析
 - Main 线程（也称为 UI 线程或原生线程）
 - 主要负责原生渲染（Native UI）和调用原生模块（Native Modules）
 - Shadow 线程（也称为 Layout 线程）
 - 创建 Shadow Tree 来模拟 React 结构树（类似虚拟 DOM）
 - 再由 Yoga 引擎将 Flexbox 等样式，解析成原生平台的布局方式
- RN 使用 Flexbox 布局，但是原生是不支持，Yoga 用来将 Flexbox 布局转换为原生平台的布局方式。

渲染机制

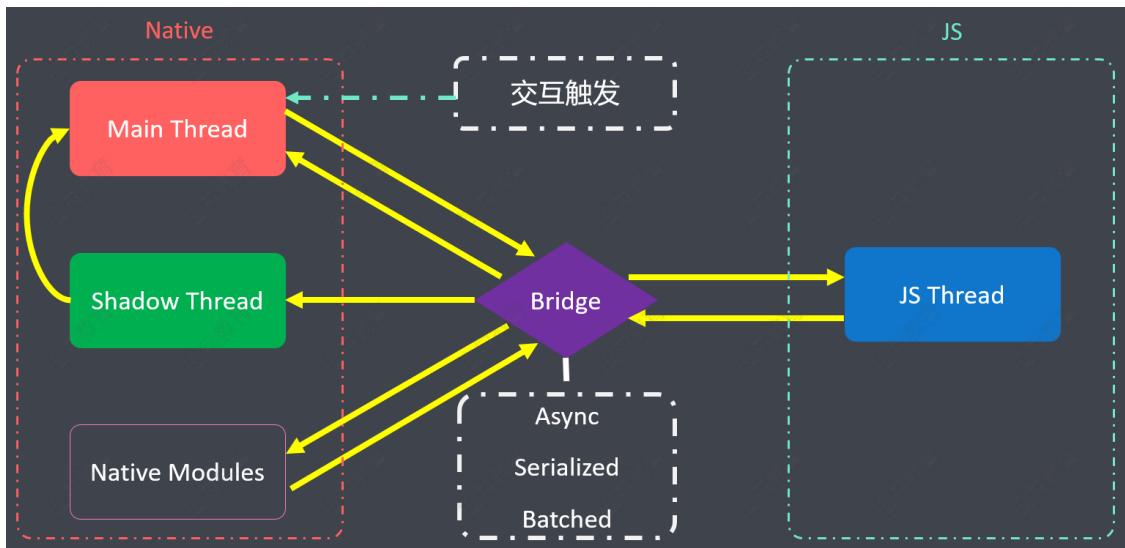
- JS 线程将视图信息（结构、样式、属性等）传递给 Shadow 线程，
- 创建出用于布局计算的 Shadow Tree，Shadow 线程计算好布局之后，再将完整的视图信息（包括宽高、位置等）传递给主线程
- 主线程据此创建 Native View (UI)



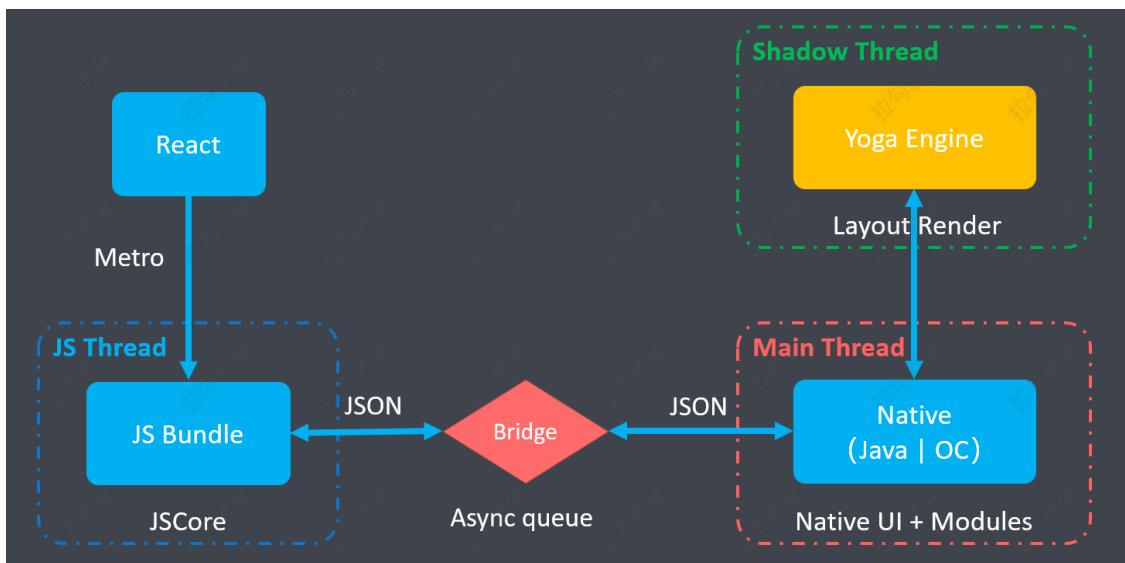
也可以通过下图，来理解渲染过程：

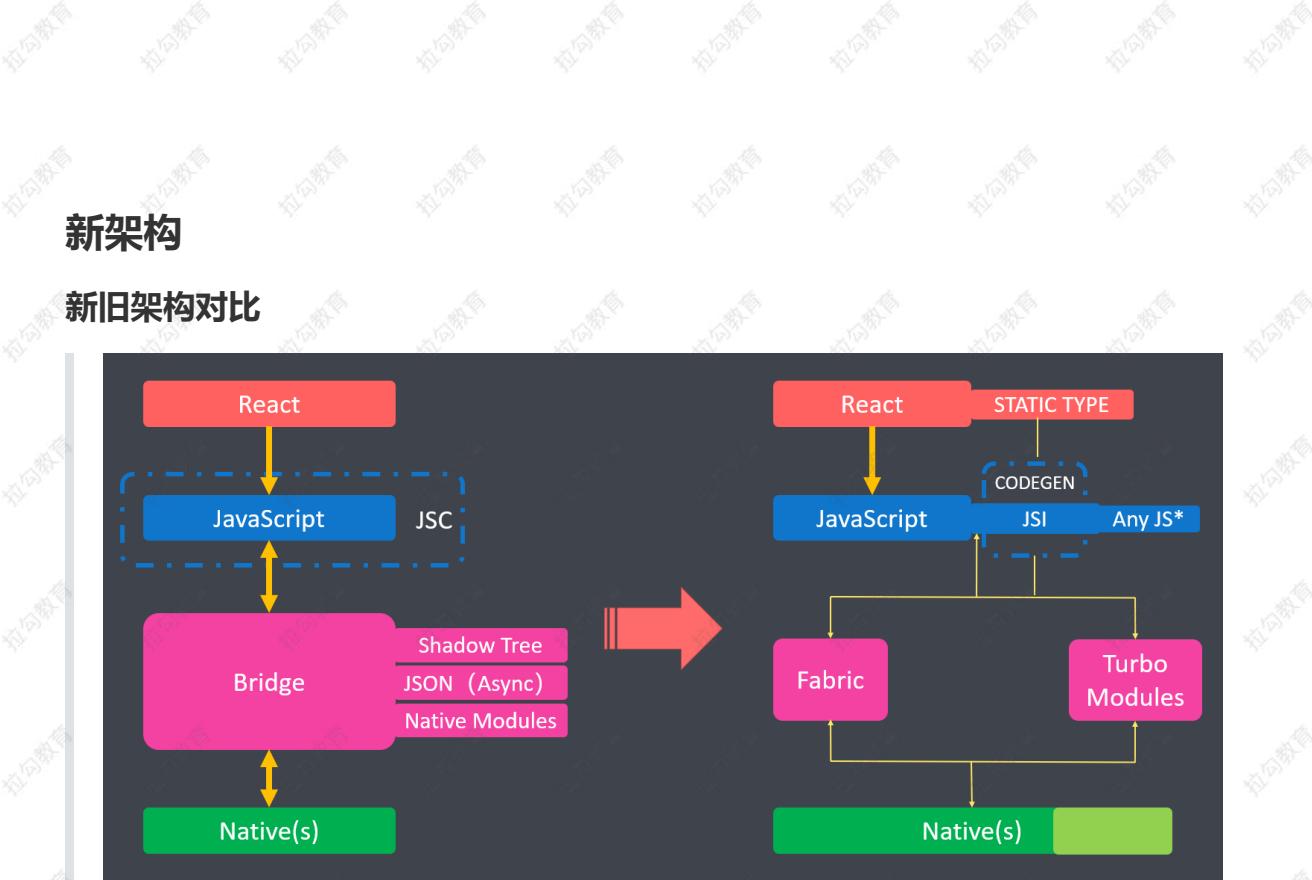


线程间通信



现有架构启动流程





新架构的主要改动

- JavaScript 层:
 - 支持 React 16+ 的新特征
 - 增强 JS 静态类型检查 (CodeGen)
 - 引入 JSI，允许替换不同的 JavaScript 引擎。支持 JS 与 Native 直接通信
- Bridge 层:
 - 划分成 Fabric 和 TurboModules 两部分，分别负责 UI 管理与 Native 模块
- Native 层:
 - 精简核心模块，将非核心部分拆分出去，作为社区模块，独立更新维护

增强类型检查

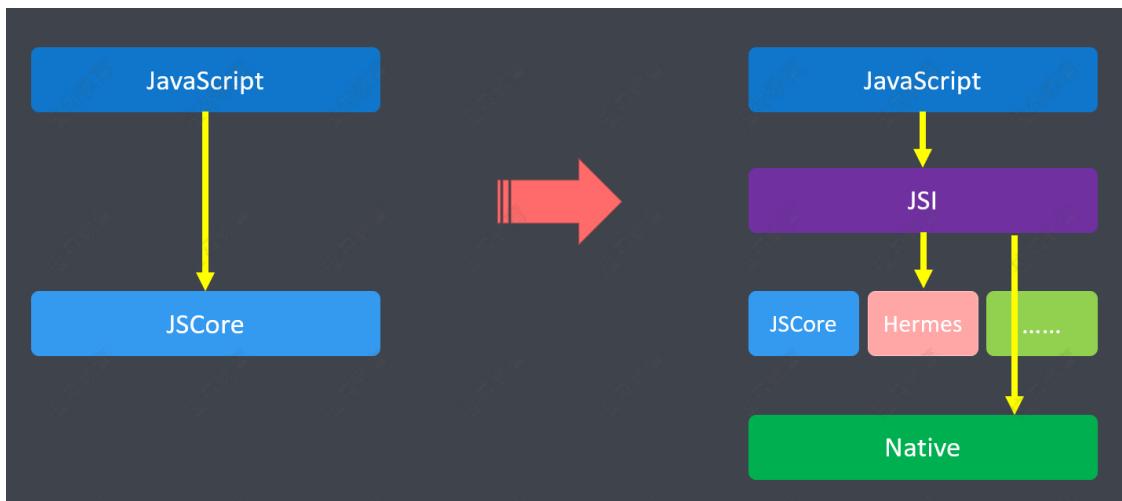
- CodeGen 是 Facebook 推出的代码生成工具
通过 CodeGen，自动将 Flow 或者 TypeScript 等有静态类型的 JS 代码翻译成 Fabric 和 TurboModules 使用的接口文件。
- 加入类型约束后的作用：
 - 减少了数据类型错误
 - 减少了数据验证的次数，提高了通信性能

举个例子：JS 中的数字经常被引号引起起来，从而将数字类型转成了字符串。将转换后的数字传递给 bridge 的时候，通常 iOS 下会静默失败，而 Android 会崩溃。

另外，类型约束对通信性能也有一定提升。因为，在加入类型约束之前，每次通信都需要进行数据验证。加载类型约束之后，我们就没有必要每次通信都进行数据验证了。减少了数据验证的次数，就会提高通信性能。

JSI (JavaScript Interface)

不同于之前直接将 JavaScript 代码输入给 JSC，新的架构中引入了一层 JSI (JavaScript Interface)，作为 JSC 之上的抽象



- JSI 是一个用C++写成的轻量级框架。其作用主要有两个：
 - 通过 JSI，可以实现 JS 引擎的更换
 - 通过 JSI，可以通过 JS 直接调用 Native
 - JS 对象可以直接获得 C++ 对象(Host Objects)引用，从而允许 JS 与 Native 的直接调用
 - 减少不必要的线程通信
 - 省去了序列化和反序列化的成本
 - 减轻了通信压力，提高了通信性能

优化 Bridge 层

- Fabric

- 简化了 UI 渲染

Fabric 简化了 React Native 渲染，简化之前渲染流程中，有复杂跨线程交互 (React -> Native -> Shadow Tree -> Native UI)。优化之后，直接在 C++ 层创建 JavaScript 与 Native 共享的 Shadow Tree，并通过 JSI 层将 UI 操作接口暴露给 JavaScript，允许 JavaScript 直接控制高优先级的 UI 操作，甚至允许同步调用 (应对列表快速滚动、页面切换、手势处理等场景)。这样避免了跨线程的操作，极大地提高了UI的响应速度。

- Turbo Modules

- 通过 JSI，可以让 JS 直接调用 Native 模块，实现同步操作
 - 实现 Native 模块按需加载，减少启动时间，提高性能

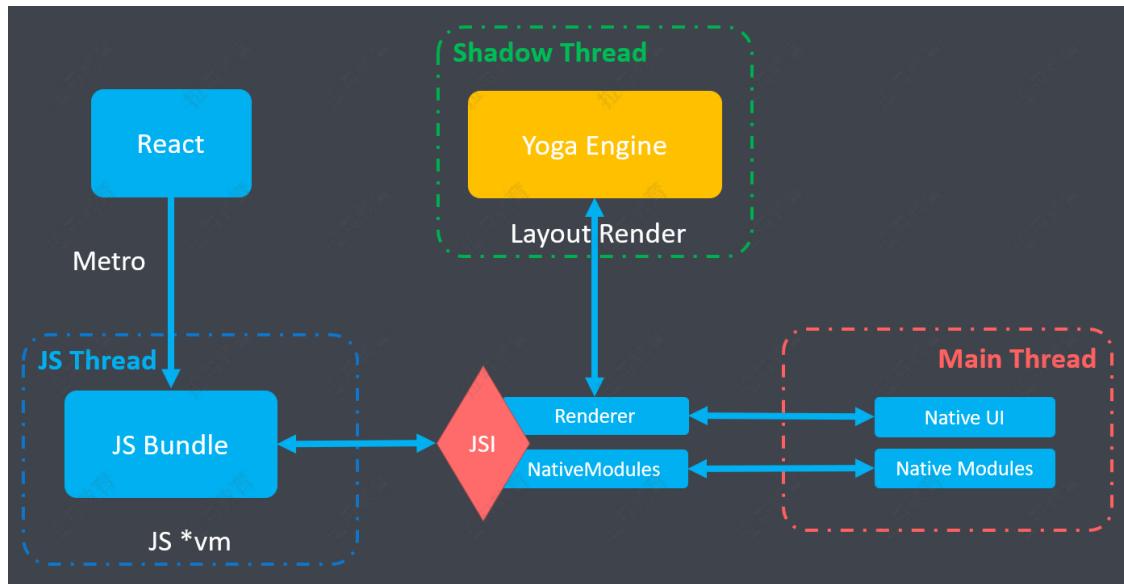
之前所有 Native Modules (无论是否需要用到) 都要在应用启动时进行初始化，因为 Native 不知道 JS 将会调用哪些功能模块。而新的 Turbo Modules 允许按需加载 Native 模块，并在模块初始化之后直接持有其引用，不再依靠消息通信来调用模块功能。因此，应用的启动时间也会有所提升

精简核心 (Lean Core)

- 将 react-native 核心包进行瘦身
 - RN 推出多年，其核心包太过臃肿
 - 有些包在项目中用不到，每次也要引入，造成资源浪费
- 非必要的包，移到社区模块，单独维护

- 例如： AsyncStorage、WebView 等

新架构启动流程



项目

项目简介

项目截图





项目规划

通过学过的 React Native 基础支持，完成一款接近实战的 App 开发。主要的功能点包括：

- 路由规划
- 数据接口
 - 申请数据接口
 - 调试数据接口
 - 调用数据接口
- UI 界面
 - 首页

获取地理位置，并根据地理位置调用相关接口
 - 新闻页

调用新闻列表接口，并展示
 - 用户页

用户登录，注册以及个人中心
- 状态管理
 - Redux
 - 路由鉴权

注意：

这里的项目代码，与之前的基础语法代码放在了一起。所以，使用之前的组件时，不再需要配置。
如果你的项目是重新初始化的，则你需要完成组件安装配置相关的所有工作。

例如：我之前已经将 react-native-vector-icons 组件配置好了。所以在项目中就不需要再次配置。
如果你的项目是重新初始化的，则需要重新安装和配置 react-native-vector-icons

数据接口

申请接口

第三方接口的申请方式大同小异。这里，我们以和风天气为例。来申请第三方接口

和风天气提供了免费的天气预报相关的 API。但是，使用之前，需要注册申请



1. 注册账号



2. 创建应用和KEY



3. 开发集成

1. 注册和风天气账号

<https://id.qweather.com/#/register>

2. 创建应用，获取 key

登陆账号，进入控制台

The screenshot shows the Heweather Control Panel interface. On the left is a sidebar with options like 'My Control Panel', 'Application Management', 'Resource Pack', 'Plugin Management', 'Data Statistics', 'Order Management', 'Consumption Record', 'Invoice Management', 'Contact Us', and 'Download APP'. The main area is titled 'Application Management' and shows a table for 'wechat' with three entries: 'weekend' (Public ID: HE19), 'sinasae' (Public ID: HE19), and 'lgmp' (Public ID: HE20). Each entry has 'Web API' type, 'Edit' and 'Delete' buttons, and a 'Type' column. At the top right of the main area is a blue 'Create Application' button, which is circled in red.

3. 开发集成

<https://dev.qweather.com/docs/start/#documentation>

调试接口

API 调试工具是用来对 API 接口进行调试的。比较出名的有 Postman, Insomnia、Postwoman 等。本节以 Insomnia 为例。

首先，需要跳转到 Insomnia 官网，下载 (Insomnia Core)：<https://insomnia.rest/download> 然后执行安装。

然后，配置开发环境。

The screenshot shows the Insomnia Core application window. On the left is a sidebar with 'Insomnia' and 'GET' selected. Below it are buttons for 'ACTIVATE ENVIRONMENT' (with 'Use hweather' checked), 'No Environment', and 'GENERAL'. At the bottom is a 'Manage Environments' button with a red border. To the right is a 'Manage Environments' dialog box. It has tabs for 'Base Environment' and 'Sub Environments'. Under 'Sub Environments', there is a list with 'Environment' selected. A code snippet is shown in a box, with the 'key' field highlighted and circled in red:

```
1 v {
2   "code": "200",
3   "updateTime": "2020-10-30T10:35+08:00",
4   "fxLink": "http://hfx.link/2ay1",
5   "daily": [
6     {
7       "fxDate": "2020-10-30",
8       "sunrise": "06:43",
9       "sunset": "17:12",
10      "moonrise": "16:55",
11      "moonset": "05:02",
12      "moonPhase": "盈凸月",
13      "tempMax": "19",
14      "tempMin": "7",
15      "iconDay": "101",
16      "textDay": "多云",
17      "iconNight": "305",
18      "textNight": "小雨",
19      "wind360Day": "180",
20      "windDirDay": "南风",
21      "windScaleDay": "1-2",
22      "windSpeedDay": "3",
23      "wind360Night": "180",
24      "windDirNight": "南风",
25    }
26  ]
27}
```

配置完之后，就可以根据接口（例如：和风天气接口）的使用规则，在 Insomnia 中调试接口了。使用效果如下：

The screenshot shows the Insomnia Core application window with a successful API call result. The title bar says 'Insomnia (hweather) - 3天预报'. The main interface shows a table with columns: 'Body', 'Auth', 'Query', 'Header', and 'Docs'. The 'Body' column shows the URL: 'https://devapi.qweather.com/v7/weather/3d?key=687e517f0684448a9f4695721414a07&location=116.29845,39.95933'. The 'Query' column shows 'key' and 'location' with their values. The 'Header' column shows 'Content-Type: application/json'. The 'Docs' column shows a preview of the JSON response. The response details are as follows:

```
200 OK | 92.5 ms | 530 B | Just Now
{
  "code": "200",
  "updateTime": "2020-10-30T10:35+08:00",
  "fxLink": "http://hfx.link/2ay1",
  "daily": [
    {
      "fxDate": "2020-10-30",
      "sunrise": "06:43",
      "sunset": "17:12",
      "moonrise": "16:55",
      "moonset": "05:02",
      "moonPhase": "盈凸月",
      "tempMax": "19",
      "tempMin": "7",
      "iconDay": "101",
      "textDay": "多云",
      "iconNight": "305",
      "textNight": "小雨",
      "wind360Day": "180",
      "windDirDay": "南风",
      "windScaleDay": "1-2",
      "windSpeedDay": "3",
      "wind360Night": "180",
      "windDirNight": "南风",
      "windScaleNight": "1-2"
    }
  ]
}
```

调用接口

React Native 提供了和 web 标准一致的 Fetch API，用于满足开发者访问网络的需求。

Get 调用

```
fetch(url, {
  method: 'GET'
}).then(function(response) {
  // 获取数据,数据处理
}).catch(function(err) {
  // 错误处理
});
```

POST 方式

```
let param = {user:'xxx',phone:'xxxxxx'};
fetch(url, {
  method: 'post',
  body: JSON.stringify(param)
}).then(function(response) {
  // 获取数据,数据处理
});
```

当然，也可以通过 [Axios](#) 进行接口调用。

UI 界面

UI 界面中用到的技术，大部分都是之前学习的。包括 RN 样式，RN 组件，RN 路由等。按照项目功能划分。包括：

- 首页
 - Tab 导航
 - header 设置
 - 响应式效果（通过 Dimensions 完成）
 - 轮播图（react-native-swiper）
 - 和风天气接口（获取地理位置，调用接口，FlatList 展示，线性渐变展示等）
- 新闻页
 - 调用新闻接口，通过 FlatList 展示
 - 通过 WebView，展示新闻详情（路由声明，路由跳转，路由传参）
- 用户页
 - 用户中心
 - 基本页面布局
 - 图标展示（Ionicons）
 - 路由声明与跳转
 - 用户登录
 - 背景图（ImageBackground）
 - 动画效果（react-native-linear-gradient）
 - 表单（常用属性和方法，数据验证与提示）
 - 用户注册
 - 所用技术与用户登录页面一致

其中，涉及到的新的组件，有以下两个

react-native-linear-gradient

[react-native-linear-gradient](#) 是用来声明线性渐变的组件

1. 安装

```
yarn add react-native-linear-gradient
```

2. 配置

如果 RN >= 0.60

- Android

不需要做任何操作，RN 启动时，会自动运行下面的链接命令

```
react-native link react-native-linear-gradient
```

- iOS

```
npx pod-install
```

如果 RN < 0.60 或 上述自动链接无效。可以参考官网的手动链接

<https://github.com/react-native-linear-gradient/react-native-linear-gradient>

3. 使用

```
import React, { Component } from 'react'
import { Text, Stylesheet, View } from 'react-native'
import LinearGradient from 'react-native-linear-gradient'

export default class LinearGradientDemo extends Component {
  render() {
    return (
      <View style={{flex: 1, justifyContent: 'center'}}>
        <LinearGradient
          colors={['#4c669f', '#3b5998', '#192f6a']}
          style={styles.linearGradient}
        >
          <Text style={styles.buttonText}> 垂直渐变 </Text>
        </LinearGradient>
        <LinearGradient
          start={{x: 0, y: 0}}
          end={{x: 1, y: 0}}
          colors={['#4c669f', '#3b5998', '#192f6a']}
          style={styles.linearGradient}
        >
          <Text style={styles.buttonText}> 水平渐变
        </Text>
      </LinearGradient>
    
```

```
        start={{x: 0.0, y: 0.2}}
        end={{x: 0.15, y: 1.2}}
        locations={[0,0.5,0.7]}
        colors=['#4c669f', '#3b5998', '#192f6a']}
        style={styles.linearGradient}
      >
    <Text style={styles.buttonText}>
      倾斜渐变
    </Text>
  </LinearGradient>
</view>
)
}
}

const styles = StyleSheet.create({
  linearGradient: {
    flex: 1,
    borderRadius: 55,
    margin: 10,
    justifyContent: 'center'
  },
  buttonText: {
    fontSize: 30,
    textAlign: 'center',
    color: '#fff',
  },
})
```

LinearGradient 中要的属性

- colors

必填项。声明颜色的数组，至少包含两个颜色值。

- start

可选项。对象格式，声明渐变的**开始**的坐标位置。{ x: number, y: number }

x 或 y 的取值范围是 0 - 1, 1就是100%，{x:0, y:0} 表示左上角，{x:1, y:0} 表示右上角

- end

可选项。对象格式，声明渐变的**结束**的坐标位置。{ x: number, y: number }

- locations

可选项，声明渐变颜色结束位置的数组。例如：

[0.1, 0.75, 1] 表示：

第一个颜色占据 0% - 10%，

第二个颜色占据 10% - 75%

最后一个颜色占据 75% - 100%

效果图



react-native-animatable

[react-native-animatable](#) 是非常流行的动画组件库。

1. 安装

```
yarn add react-native-animatable
```

2. 使用

```
import * as Animatable from 'react-native-animatable';

// 创建自定义组件
MyCustomComponent = Animatable.createAnimatableComponent(MyCustomComponent);

// 使用现有组件
<Animatable.View animation="动画名称" duration={持续时间} style={{}} >
  ...
</Animatable.View>
```

具体的 **动画名称 (animation)** 可以参考官网: <https://github.com/oblador/react-native-animatable>

例如：

- 弹跳

- 进入

- bounceIn
 - bounceInDown
 - bounceInUp
 - bounceInLeft
 - bounceInRight

- 离开

- bounceOut
 - bounceOutDown
 - bounceOutUp
 - bounceOutLeft
 - bounceOutRight

- 淡入淡出

- 进入

- fadeIn
 - fadeInDown
 - fadeInDownBig
 - fadeInUp
 - fadeInUpBig
 - fadeInLeft
 - fadeInLeftBig
 - fadeInRight
 - fadeInRightBig

- 离开

- fadeOut
 - fadeOutDown
 - fadeOutDownBig
 - fadeOutUp
 - fadeOutUpBig
 - fadeOutLeft
 - fadeOutLeftBig
 - fadeOutRight
 - fadeOutRightBig

其他 动画名称 (animation) 可以参考官网：<https://github.com/oblador/react-native-animated>

状态管理

Redux

Redux 是 React 中进行状态管理的工具，这里以复习回顾的方式。实现简单的计数器功能，打通 Redux 流程。

1. 安装 Redux

```
yarn add redux  
yarn add react-redux  
yarn add redux-thunk # 支持异步操作
```

2. 创建 store

redux/actions/actionTypes.js (用来集中管理 Action type)

```
/**  
 * 操作的细粒度划分  
 */  
export default {  
  COUNTER_INCREMENT: 'COUNTER_INCREMENT',  
  COUNTER_DECREMENT: 'COUNTER_DECREMENT',  
}
```

redux/reducers/Counter.js

```
import actionTypes from '../actions/actionTypes'  
  
const initState = {  
  num: 1  
}  
  
export default (state = initState, action) => {  
  switch (action.type) {  
    case actionTypes.COUNTER_INCREMENT:  
      return {  
        ...state,  
        num: state.num + action.payload  
      }  
    case actionTypes.COUNTER_DECREMENT:  
      return {  
        ...state,  
        num: state.num - action.payload  
      }  
    default:  
      return state  
  }  
}
```

redux/reducers/index.js

```
import { combineReducers } from 'redux'  
import Counter from './Counter'  
  
export default combineReducers({  
  Counter  
})
```

redux/store.js

```
import { createStore, applyMiddleware} from 'redux'
import reducers from './reducers'
import reduxThunk from 'redux-thunk'
const store = createStore(
  reducers,
  applyMiddleware(reduxThunk)
)

export default store
```

3. 将 store 挂载到 App 组件上

```
import { Provider } from 'react-redux'
import store from './redux/store'

export default class index extends Component {
  render() {
    return (
      <Provider store={store}>
        <Routes />
      </Provider>
    )
  }
}
```

4. 在组件内使用 redux 数据

```
import { connect } from 'react-redux'
import { increment, decrement } from '.././redux/actions/counter'

const mapStateToProps = state => {
  return {
    num: state.Counter.num
  }
}

class Counter extends Component {
  constructor(props) {
    super(props)
  }

  render() {
    return (
      <View style={{flex:1, flexDirection:'row', justifyContent:'space-around', alignItems: 'center'}}>
        <Button title={'-' } onPress={() => this.props.decrement(1)} />
        <Text>{this.props.num}</Text>
        <Button title={'+' } onPress={() => this.props.increment(1)} />
      </View>
    )
  }
}
```

```
    }

    export default connect(mapStateToProps, { increment, decrement })(Counter)
```

路由鉴权

大多数应用都有用户鉴权的要求，用户鉴权通过之后，才能访问与之相关的私有数据。

典型的鉴权流是这样的

- 用户打开应用
 - 初始会根据登录信息，从 Redux 中获取（此时可以将状态信息持久化存储）
 - 再次进入 App，从持久化存储（例如：AsyncStorage）中获取鉴权状态
- 当状态加载后，判断用户鉴权状态是否合法，合法跳转到首页，否则弹出鉴权页面
- 当用户退出应用后，我们清除鉴权状态，并跳回鉴权页面

代码示例：

```
import * as React from 'react';
import { Button, Text, TextInput, View } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
import AsyncStorage from '@react-native-async-storage/async-storage';

// createContext 创建上下文
const AuthContext = React.createContext();

function SplashScreen() {
  return (
    <View style={{flex:1, justifyContent: 'center', alignItems: 'center'}}>
      <Text>正在加载...</Text>
    </View>
  );
}

function HomeScreen() {
  // useContext 可以获取上下文中的数据（跨组件层级获取数据）
  const { signout } = React.useContext(AuthContext);

  return (
    <View>
      <Text>已经登陆!</Text>
      <Button title="退出" onPress={signout} />
    </View>
  );
}

function SignInScreen() {
  // useState 中接收唯一的参数，是状态的初始值
  // 返回值为数组，第一个元素是状态名，第二个是修改状态的方法，写法固定，以 set 开头
  const [username, setUsername] = React.useState('');
  const [password, setPassword] = React.useState('');
```

```
const { signIn } = React.useContext(AuthContext);

return (
  <View>
    <TextInput
      placeholder="用户名"
      value={username}
      onChangeText={setUsername}
    />
    <TextInput
      placeholder="密码"
      value={password}
      onChangeText={setPassword}
      secureTextEntry
    />
    <Button title="登陆" onPress={() => signIn({ username, password })} />
  </View>
);
}

const Stack = createStackNavigator();

export default function App({ navigation }) {

  // useReducer (reducer, initstate)
  const [state, dispatch] = React.useReducer(
    (prevState, action) => {
      switch (action.type) {
        case 'RESTORE_TOKEN':
          return {
            ...prevState,
            userToken: action.token,
            isLoading: false,
          };
        case 'SIGN_IN':
          return {
            ...prevState,
            isSignout: false,
            userToken: action.token,
          };
        case 'SIGN_OUT':
          return {
            ...prevState,
            isSignout: true,
            userToken: null,
          };
      }
    },
    {
      isLoading: true,
      isSignout: false,
      userToken: null,
    }
  );
  // useEffect 相当于类组件的生命周期
  React.useEffect(() => {

```

```
// 获取存储中的 token，然后导航到响应的位置
const bootstrapAsync = async () => {
  let userToken;

  try {
    userToken = await AsyncStorage.getItem('userToken');
  } catch (e) {
    // Restoring token failed
  }

  // 取到数据后，更新到状态中
  dispatch({ type: 'RESTORE_TOKEN', token: userToken });
};

bootstrapAsync();
}, []);

// useMemo 相当于 Vue 中的计算属性（computed），用来检测数据变化
const authContext = React.useMemo(
() => {
  signIn: async data => {
    // 执行登陆。可能与后端接口对接，也可以将登陆数据持久化存储到 AsyncStorage 中
    dispatch({ type: 'SIGN_IN', token: 'dummy-auth-token' });
  },
  signOut: () => dispatch({ type: 'SIGN_OUT' }),
  signUp: async data => {
    // 执行登陆。可能与后端接口对接，也可以将登陆数据持久化存储到 AsyncStorage 中
    dispatch({ type: 'SIGN_IN', token: 'dummy-auth-token' });
  },
},
[])

return (
  <AuthContext.Provider value={authContext}>
    <NavigationContainer>
      <Stack.Navigator>
        {state.isLoading ? (
          // 我们还未完成 token 的验证
          <Stack.Screen name="Splash" component={splashScreen} />
        ) : state.userToken == null ? (
          // 本地没有 token，说明用户未登陆
          <Stack.Screen
            name="SignIn"
            component={SignInScreen}
            options={{
              title: '登陆',
              animationTypeForReplace: state.isSignout ? 'pop' : 'push',
            }}
          />
        ) : (
          // 用户已经登陆
          <Stack.Screen name="Home" component={HomeScreen} />
        )}
      </Stack.Navigator>
    </NavigationContainer>
  </AuthContext.Provider>
);
```

```
}
```

项目优化

使用第三方 UI 组件

RN 官方组件太过简单，而自己去写样式又太浪费时间。而选择一些成熟的，第三方UI组件库。会让我们的项目开发事半功倍。

这里列出一些比较流行的

- [NativeBase](#) 目前在 Github 上有 14.5k 个星
官网: <https://nativebase.io/>
- [react-native-paper](#) 目前在 Github 上有 6.8k 个星
官网: <https://reactnativepaper.com/>

这里我们推荐 [react-native-paper](#)

1. 安装

```
yarn add react-native-paper
```

2. 使用

先将整个应用，通过 Provider 包裹起来。

```
import { Provider as PaperProvider } from 'react-native-paper';

export default function Main() {
  return (
    <PaperProvider>
      <App />
    </PaperProvider>
  );
}
```

然后再项目中使用具体的组件

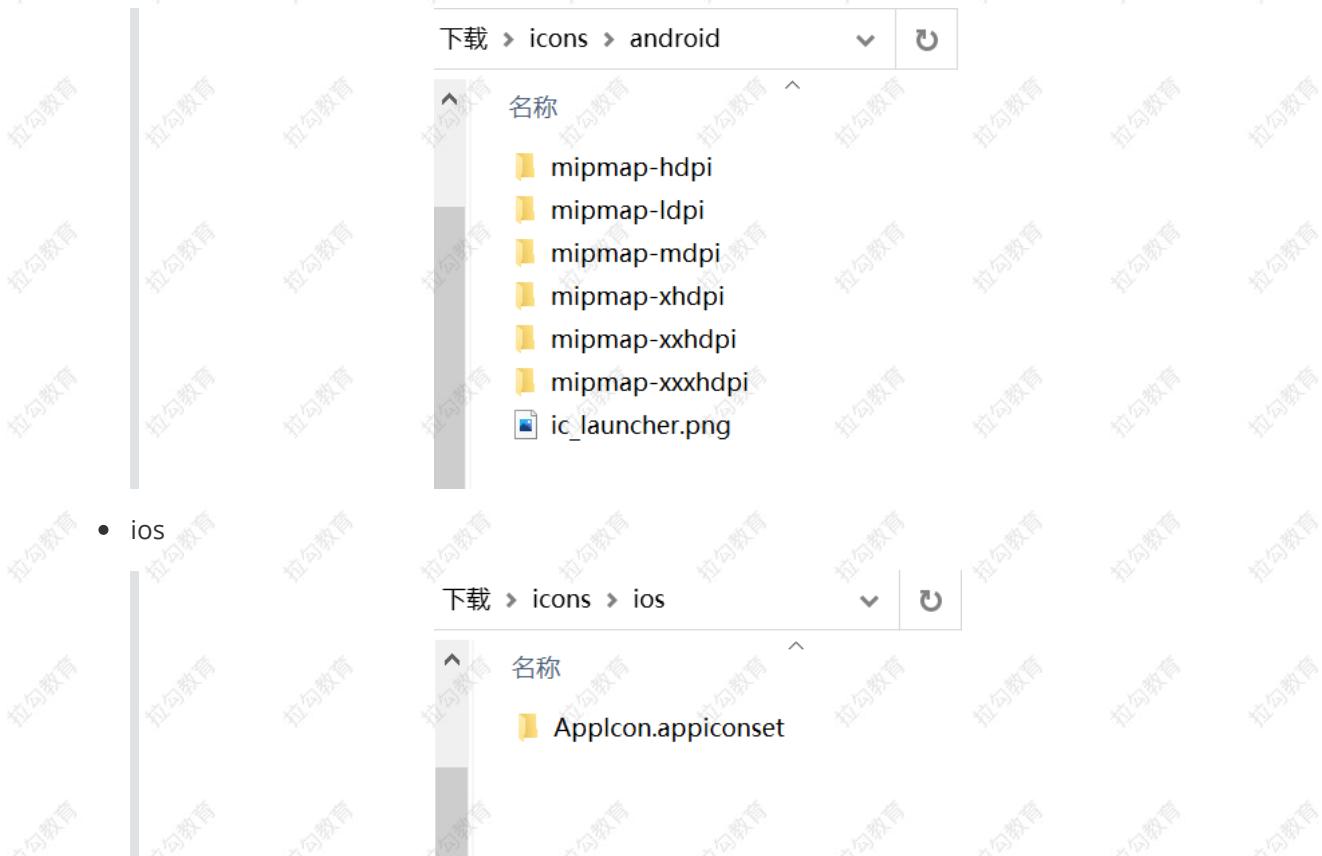
```
import { Button } from 'react-native-paper';

<Button icon="camera" mode="contained" onPress={() =>
  console.log('Pressed')}
  Press me
</Button>
```

修改应用 logo

应用图标对尺寸有要求，比较简单地方式是，准备一张 1024*1024 的图片，然后 [在线生成](#)。生成之后，我们可以将生成的图标下载下来。解压后，我们会得到两个目录：

- android



分别将上述目录，复制到 RN 项目对应的位置中

1. Android

替换 `android/app/src/main/res` 下对应的内容。

2. iOS

替换 `ios/项目名称/Images.xcassets/AppIcon.appiconset` 中的内容。

修改应用名称

• Android

编辑 `android/app/src/main/res/values/strings.xml`

```
<resources>
    <string name="app_name">你的应用名称</string>
</resources>
```

• iOS

编辑 `ios/项目名称/Info.plist` 文件，定位到 `CFBundleDisplayName`

```
<key>CFBundleDisplayName</key>
<string>你的应用名称</string>
....
```

