

react native教程

编者 xiedrsz



中科软信息系统有限公司

sinosoft

目录

[简介 2](#_Toc476041853)

[概述 2](#_Toc476041854)

[搭建开发环境 2](#_Toc476041855)

[Hello World 2](#_Toc476041856)

# 简介

## 概述

React Native看起来很像React，只不过其基础组件是原生组件而非web组件。要理解React Native应用的基本结构，首先需要了解一些基本的React的概念，比如JSX语法、组件、state状态以及props属性。如果你已经了解了React，那么还需要掌握一些React Native特有的知识，比如原生组件的使用。

# 搭建开发环境

windows Android react native 开发环境已搭建完成，详见《react native开发环境搭建》

# Hello World

程序代码：

import React, { Component } from 'react';

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

class HelloWorldApp extends Component {

render() {

return (

<Text>Hello world!</Text>

);

}

}

// 注意，这里用引号括起来的'HelloWorldApp'必须和你init创建的项目名一致

AppRegistry.registerComponent('HelloWorldApp', () => HelloWorldApp);

你可以新建一个项目，然后用上面的代码覆盖你的index.ios.js或是index.android.js 文件，然后运行看看。

React Native内置了对**ES2015**标准的支持，你可以放心使用而无需担心兼容性问题。上面的示例代码中的import、from、class、extends、以及() =>箭头函数等新语法都是ES2015中的特性。

上面的代码定义了一个名为HelloWorldApp的新的组件（**Component**），并且使用了名为**AppRegistry**的内置模块进行了“注册”操作。你在编写React Native应用时，肯定会写出很多新的组件。而一个App的最终界面，其实也就是各式各样的组件的组合。组件本身结构可以非常简单——唯一必须的就是在render方法中返回一些用于渲染结构的JSX语句。

AppRegistry模块则是用来告知React Native哪一个组件被注册为整个应用的根容器。你无需在此深究，因为**一般在整个应用里AppRegistry.registerComponent这个方法只会调用一次**。上面的代码里已经包含了具体的用法，你只需整个复制到index.ios.js或是index.android.js文件中即可运行。

# Props属性

大多数组件在创建时就可以使用各种参数来进行定制。用于定制的这些参数就称为props（属性）。

以常见的基础组件Image为例，在创建一个图片时，可以传入一个名为source的prop来指定要显示的图片的地址，以及使用名为style的prop来控制其尺寸。

代码如下：

import React, { Component } from 'react';

import { AppRegistry, Image } from 'react-native';

class Bananas extends Component {

render() {

let pic = {

uri: 'https://upload.wikimedia.org/wikipedia/commons/d/de/Bananavarieties.jpg'

};

return (

<Image source={pic} style={{width: 193, height: 110}} />

);

}

}

AppRegistry.registerComponent('Bananas', () => Bananas);

注：在iOS上使用http链接的图片地址可能不会显示，参见：

<https://segmentfault.com/a/1190000002933776>

请注意{pic}外围有一层括号，我们需要用括号来把pic这个变量嵌入到JSX语句中。括号的意思是括号内部为一个js变量或表达式，需要执行后取值。因此我们可以把任意合法的JavaScript表达式通过括号嵌入到JSX语句中。

自定义的组件也可以使用props。通过在不同的场景使用不同的属性定制，可以尽量提高自定义组件的复用范畴。只需在render函数中引用this.props，然后按需处理即可。下面是一个例子：

import React, { Component } from 'react';

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

class Greeting extends Component {

render() {

return (

<Text>Hello {this.props.name}!</Text>

);

}

}

class LotsOfGreetings extends Component {

render() {

return (

<View style={{alignItems: 'center'}}>

<Greeting name='Rexxar' />

<Greeting name='Jaina' />

<Greeting name='Valeera' />

</View>

);

}

}

AppRegistry.registerComponent('LotsOfGreetings', () => LotsOfGreetings);

上面的例子出现了一样新的名为View的组件。View 常用作其他组件的容器，来帮助控制布局和样式。

# State状态

我们使用两种数据来控制一个组件：**props**和**state**。props是在父组件中指定，而且一经指定，在被指定的组件的生命周期中则不再改变。 对于需要改变的数据，我们需要使用state。

一般来说，你需要在**constructor**中初始化state（译注：这是ES6的写法，早期的很多ES5的例子使用的是getInitialState方法来初始化state，这一做法会逐渐被淘汰），然后在需要修改时调用**setState**方法。

假如我们需要制作一段不停闪烁的文字。文字内容本身在组件创建时就已经指定好了，所以文字内容应该是一个prop。而文字的显示或隐藏的状态（快速的显隐切换就产生了闪烁的效果）则是随着时间变化的，因此这一状态应该写到state中：

import React, { Component } from 'react';

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

class Blink extends Component {

constructor(props) {

super(props);

this.state = { showText: true };

// 每1000毫秒对showText状态做一次取反操作

setInterval(() => {

this.setState({ showText: !this.state.showText });

}, 1000);

}

render() {

// 根据当前showText的值决定是否显示text内容

let display = this.state.showText ? this.props.text : ' ';

return (

<Text>{display}</Text>

);

}

}

class BlinkApp extends Component {

render() {

return (

<View>

<Blink text='I love to blink' />

<Blink text='Yes blinking is so great' />

<Blink text='Why did they ever take this out of HTML' />

<Blink text='Look at me look at me look at me' />

</View>

);

}

}

AppRegistry.registerComponent('BlinkApp', () => BlinkApp);

实际开发中，我们一般不会在定时器函数（setInterval、setTimeout等）中来操作state。典型的场景是在接收到服务器返回的新数据，或者在用户输入数据之后。你也可以使用一些“状态容器”比如Redux来统一管理数据流（译注：但我们不建议新手过早去学习redux <http://redux.js.org/index.html>）。

State的工作原理和React.js完全一致。

# 样式

在React Native中，你并不需要学习什么特殊的语法来定义样式。我们仍然是使用JavaScript来写样式。所有的核心组件都接受名为**style**的属性。这些样式名基本上是遵循了web上的CSS的命名，只是按照JS的语法要求使用了驼峰命名法，例如将background-color改为backgroundColor。

style属性可以是一个普通的JavaScript**对象**。这是最简单的用法，因而在示例代码中很常见。你还可以传入一个**数组**——在数组中位置居后的样式对象比居前的优先级更高，这样你可以间接实现样式的继承。

实际开发中组件的样式会越来越复杂，我们建议使用**StyleSheet.create**来集中定义组件的样式。比如像下面这样：

import React, { Component } from 'react';

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

class LotsOfStyles extends Component {

render() {

return (

<View>

<Text style={styles.red}>just red</Text>

<Text style={styles.bigblue}>just bigblue</Text>

<Text style={[styles.bigblue, styles.red]}>bigblue, then red</Text>

<Text style={[styles.red, styles.bigblue]}>red, then bigblue</Text>

</View>

);

}

}

const styles = StyleSheet.create({

bigblue: {

color: 'blue',

fontWeight: 'bold',

fontSize: 30,

},

red: {

color: 'red',

},

});

AppRegistry.registerComponent('LotsOfStyles', () => LotsOfStyles);

常见的做法是按顺序声明和使用style属性，以借鉴CSS中的“层叠”做法（即后声明的属性会覆盖先声明的同名属性）。

# 高度与宽度

组件的高度和宽度决定了其在屏幕上显示的尺寸。

## 指定宽高

最简单的给组件设定尺寸的方式就是在样式中指定固定的width和height。React Native中的尺寸都是无单位的，表示的是与设备像素密度无关的逻辑像素点。

import React, { Component } from 'react';

import { AppRegistry, View } from 'react-native';

class FixedDimensionsBasics extends Component {

render() {

return (

<View>

<View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />

<View style={{width: 100, height: 100, backgroundColor: 'skyblue'}} />

<View style={{width: 150, height: 150, backgroundColor: 'steelblue'}} />

</View>

);

}

};

// 注册应用(registerComponent)后才能正确渲染

// 注意：只把应用作为一个整体注册一次，而不是每个组件/模块都注册

AppRegistry.registerComponent('AwesomeProject', () => FixedDimensionsBasics);

## 弹性（Flex）宽高

在组件样式中使用flex可以使其在可利用的空间中动态地扩张或收缩。一般而言我们会使用flex:1来指定某个组件扩张以撑满所有剩余的空间。如果有多个并列的子组件使用了flex:1，则这些子组件会平分父容器中剩余的空间。如果这些并列的子组件的flex值不一样，则谁的值更大，谁占据剩余空间的比例就更大（即占据剩余空间的比等于并列组件间**flex**值的比）。

组件能够撑满剩余空间的前提是其父容器的**尺寸不为零**。如果父容器既没有固定的width和height，也没有设定flex，则父容器的尺寸为零。其子组件如果使用了flex，也是无法显示的。

import React, { Component } from 'react';

import { AppRegistry, View } from 'react-native';

class FlexDimensionsBasics extends Component {

render() {

return (

// 试试去掉父View中的`flex: 1`。

// 则父View不再具有尺寸，因此子组件也无法再撑开。

// 然后再用`height: 300`来代替父View的`flex: 1`试试看？

<View style={{flex: 1}}>

<View style={{flex: 1, backgroundColor: 'powderblue'}} />

<View style={{flex: 2, backgroundColor: 'skyblue'}} />

<View style={{flex: 3, backgroundColor: 'steelblue'}} />

</View>

);

}

};

AppRegistry.registerComponent('AwesomeProject', () => FlexDimensionsBasics);

# 使用Flexbox布局

我们在React Native中使用flexbox规则来指定某个组件的子元素的布局。Flexbox可以在不同屏幕尺寸上提供一致的布局结构。

一般来说，使用flexDirection、alignItems和 justifyContent三个样式属性就已经能满足大多数布局需求。

React Native中的Flexbox的工作原理和web上的CSS基本一致，当然也存在少许差异。首先是默认值不同：flexDirection的默认值是column而不是row，而flex也只能指定一个数字值。

## Flex Direction

在组件的style中指定**flexDirection**可以决定布局的**主轴**。子元素是应该沿着水平轴(**row**)方向排列，还是沿着竖直轴(**column**)方向排列呢？默认值是竖直轴(column)方向。

import React, { Component } from 'react';

import { AppRegistry, View } from 'react-native';

class FlexDirectionBasics extends Component {

render() {

return (

// 尝试把`flexDirection`改为`column`看看

<View style={{flex: 1, flexDirection: 'row'}}>

<View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />

<View style={{width: 50, height: 50, backgroundColor: 'skyblue'}} />

<View style={{width: 50, height: 50, backgroundColor: 'steelblue'}} />

</View>

);

}

};

AppRegistry.registerComponent('AwesomeProject', () => FlexDirectionBasics);

## Justify Content

在组件的style中指定**justifyContent**可以决定其子元素沿着主轴的**排列方式**。子元素是应该靠近主轴的起始端还是末尾段分布呢？亦或应该均匀分布？对应的这些可选项有：**flex-start**、**center**、**flex-end**、**space-around**以及**space-between**。

import React, { Component } from 'react';

import { AppRegistry, View } from 'react-native';

class JustifyContentBasics extends Component {

render() {

return (

// 尝试把`justifyContent`改为`center`看看

// 尝试把`flexDirection`改为`row`看看

<View style={{

flex: 1,

flexDirection: 'column',

justifyContent: 'space-between',

}}>

<View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />

<View style={{width: 50, height: 50, backgroundColor: 'skyblue'}} />

<View style={{width: 50, height: 50, backgroundColor: 'steelblue'}} />

</View>

);

}

};

AppRegistry.registerComponent('AwesomeProject', () => JustifyContentBasics);

## Align Items

在组件的style中指定**alignItems**可以决定其子元素沿着**次轴**（与主轴垂直的轴，比如若主轴方向为row，则次轴方向为column）的**排列方式**。子元素是应该靠近次轴的起始端还是末尾段分布呢？亦或应该均匀分布？对应的这些可选项有：**flex-start**、**center**、**flex-end**以及**stretch**。

注意：要使stretch选项生效的话，子元素在次轴方向上不能有固定的尺寸。以下面的代码为例：只有将子元素样式中的width: 50去掉之后，alignItems: 'stretch'才能生效。

import React, { Component } from 'react';

import { AppRegistry, View } from 'react-native';

class AlignItemsBasics extends Component {

render() {

return (

// 尝试把`alignItems`改为`flex-start`看看

// 尝试把`justifyContent`改为`flex-end`看看

// 尝试把`flexDirection`改为`row`看看

<View style={{

flex: 1,

flexDirection: 'column',

justifyContent: 'center',

alignItems: 'center',

}}>

<View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />

<View style={{width: 50, height: 50, backgroundColor: 'skyblue'}} />

<View style={{width: 50, height: 50, backgroundColor: 'steelblue'}} />

</View>

);

}

};

AppRegistry.registerComponent('AwesomeProject', () => AlignItemsBasics);

# 处理文本输入

**TextInput**是一个允许用户输入文本的基础组件。它有一个名为**onChangeText**的属性，此属性接受一个函数，而此函数会在文本变化时被调用。另外还有一个名为**onSubmitEditing**的属性，会在文本被提交后（用户按下软键盘上的提交键）调用。

import React, { Component } from 'react';

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

class PizzaTranslator extends Component {

constructor(props) {

super(props);

this.state = {text: ''};

}

render() {

return (

<View style={{padding: 10}}>

<TextInput

style={{height: 40}}

placeholder="Type here to translate!"

onChangeText={(text) => this.setState({text})}

/>

<Text style={{padding: 10, fontSize: 42}}>

{this.state.text.split(' ').map((word) => word && '🍕').join(' ')}

</Text>

</View>

);

}

}

// 注册应用(registerComponent)后才能正确渲染

// 注意：只把应用作为一个整体注册一次，而不是每个组件/模块都注册

AppRegistry.registerComponent('PizzaTranslator', () => PizzaTranslator);

# ScrollView

**ScrollView**是一个通用的可滚动的容器，你可以在其中放入多个组件和视图，而且这些组件并不需要是同类型的。ScrollView不仅可以垂直滚动，还能水平滚动（通过**horizontal**属性来设置）。

ScrollView适合用来显示数量不多的滚动元素。如果你需要显示较长的滚动列表，那么应该使用功能差不多但性能更好的ListView组件。