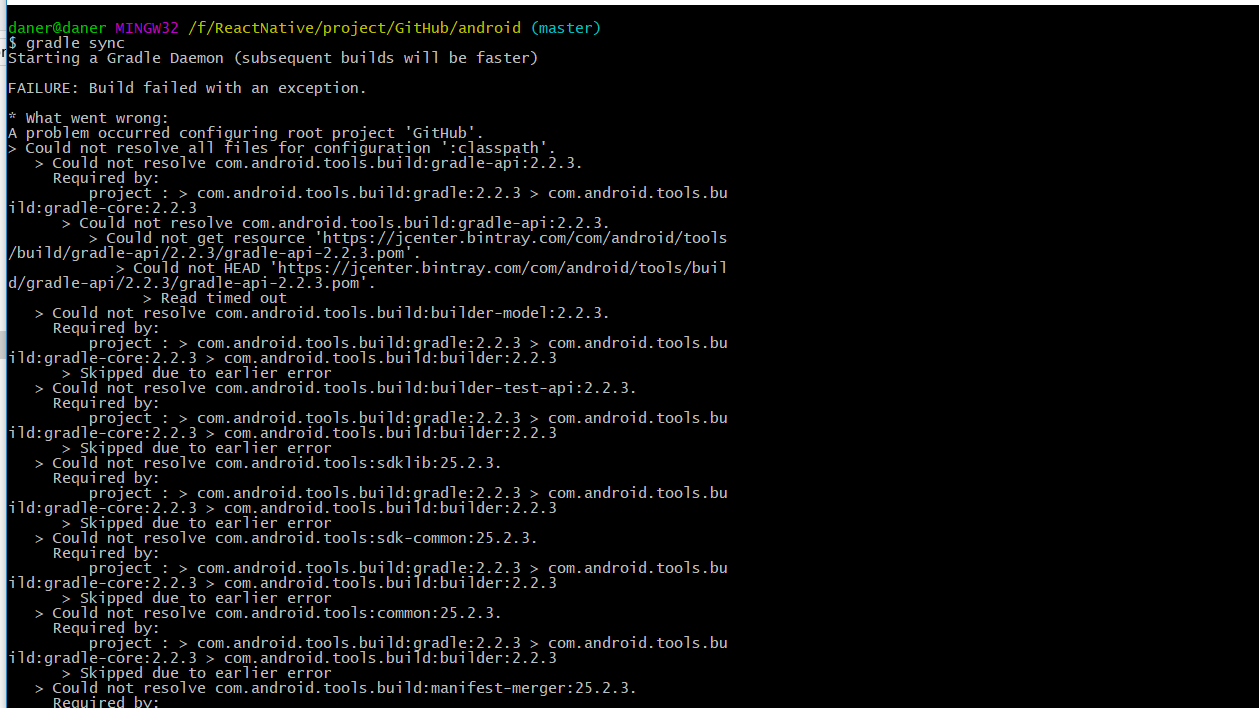
# 第一部分 ReactNative基础篇

## 一 新建ReactNative项目

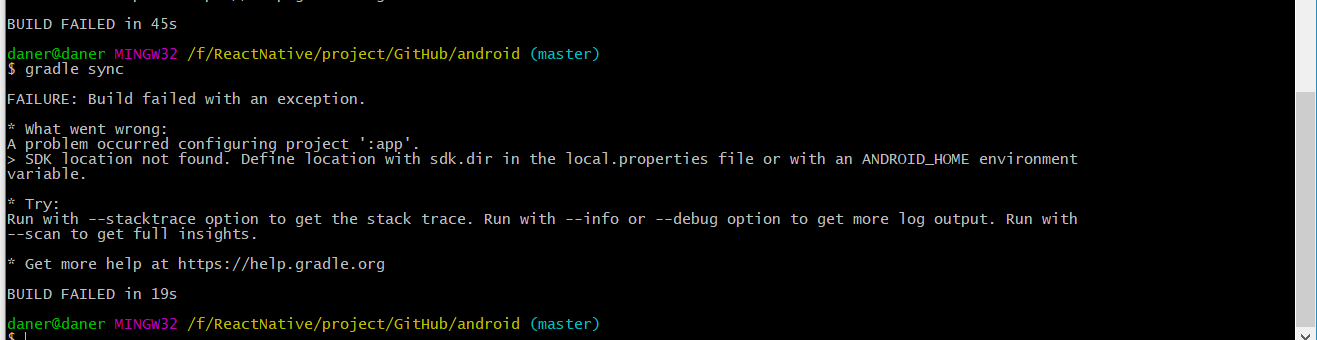
参考博客3 新建RN的工程并创建项目运行

react-native init GitHub

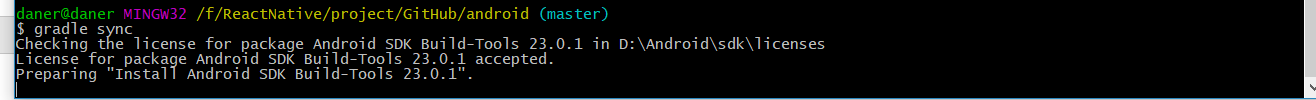


打开梯子

gradle sync

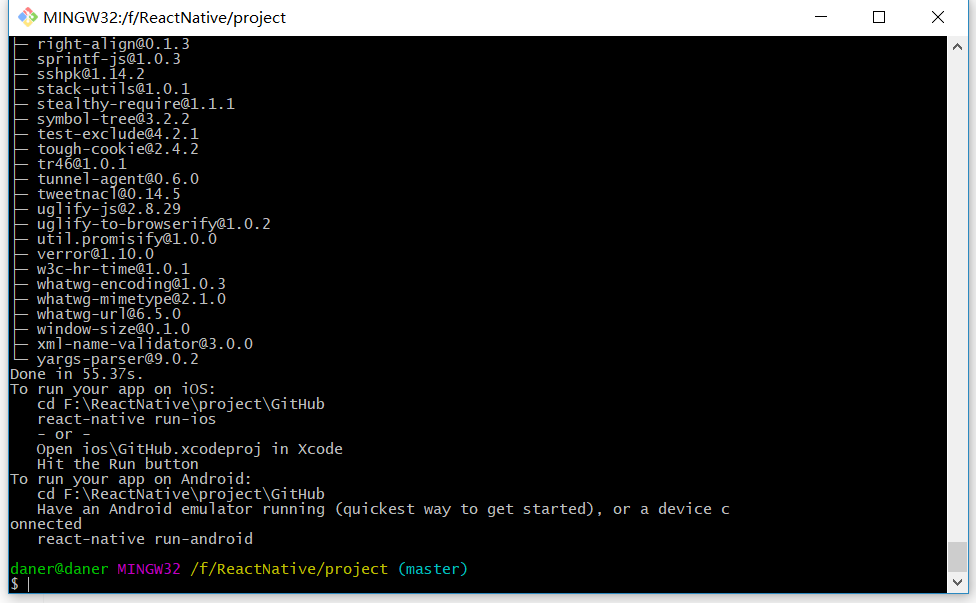


拷贝其他项目的local.properties到项目中





https://stackoverflow.com/questions/47495066/the-compileoptions-bootclasspath-property-has-been-deprecated

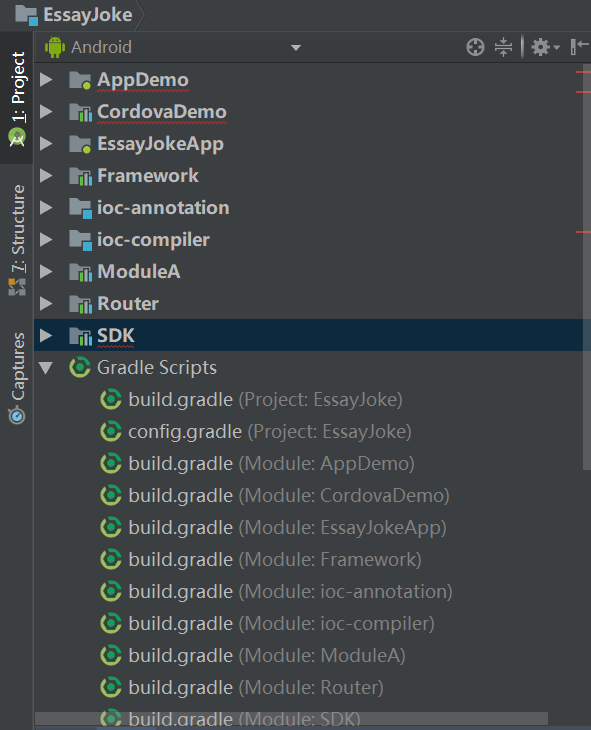


## 二 Android已有项目集成React Native项目

### 1 AS中集成ReactNative步骤

首先搭建好Android开发环境，并有一个新建的或者已经存在的Android项目

在未创建ReactNative项目大概是这个样子

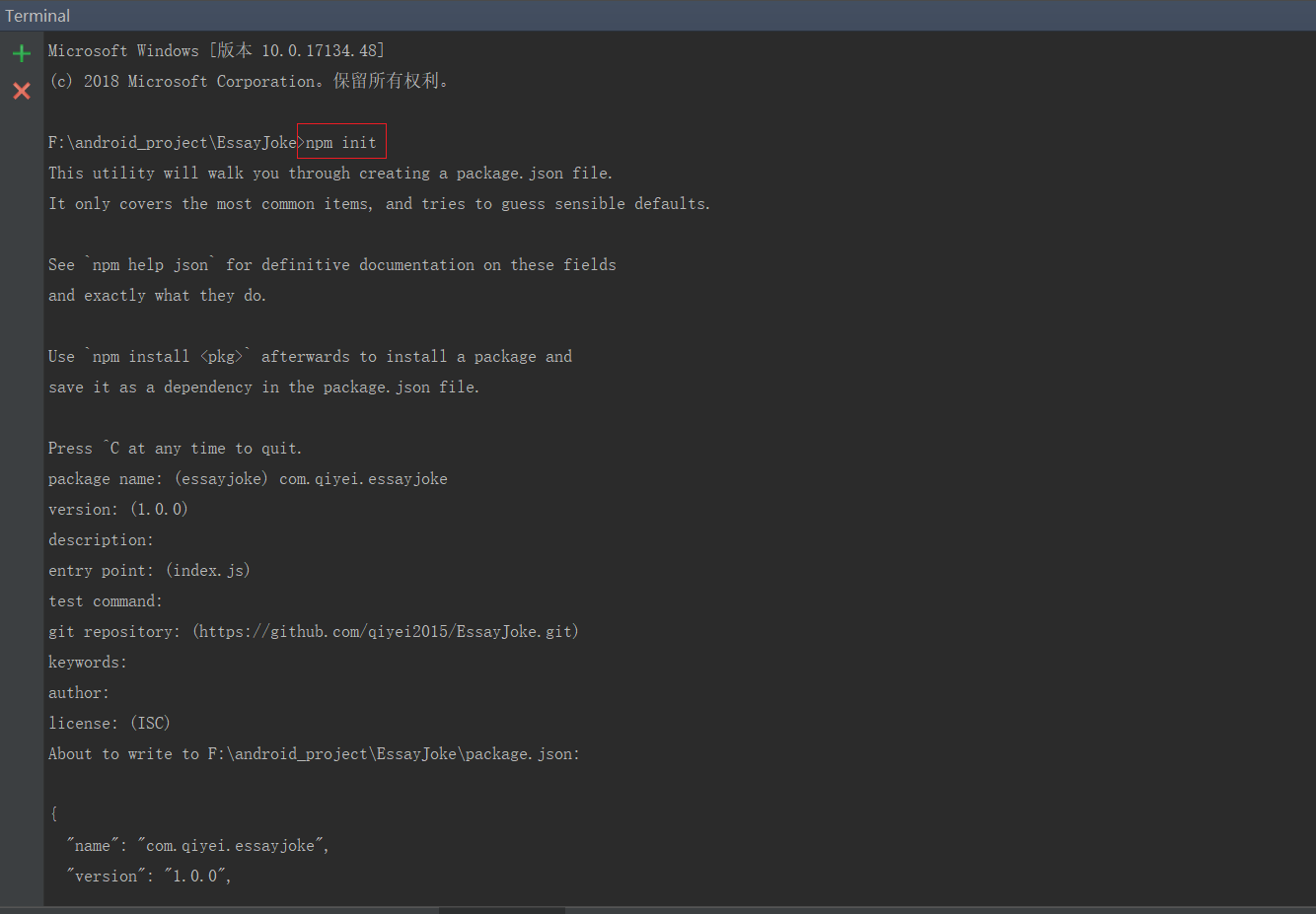


接着创建ReactNative Module用于存放我们RN相关的内容，极大程度的与原生开发解耦

#### 1 在应用中添加js代码

在项目根目录输入以下命令

npm init



在这中间要求输入包名，其他的默认即可

这一步完成之后，在项目的根目录下就会生成package.json这个文件

接下来还需要把启动脚本放进去：

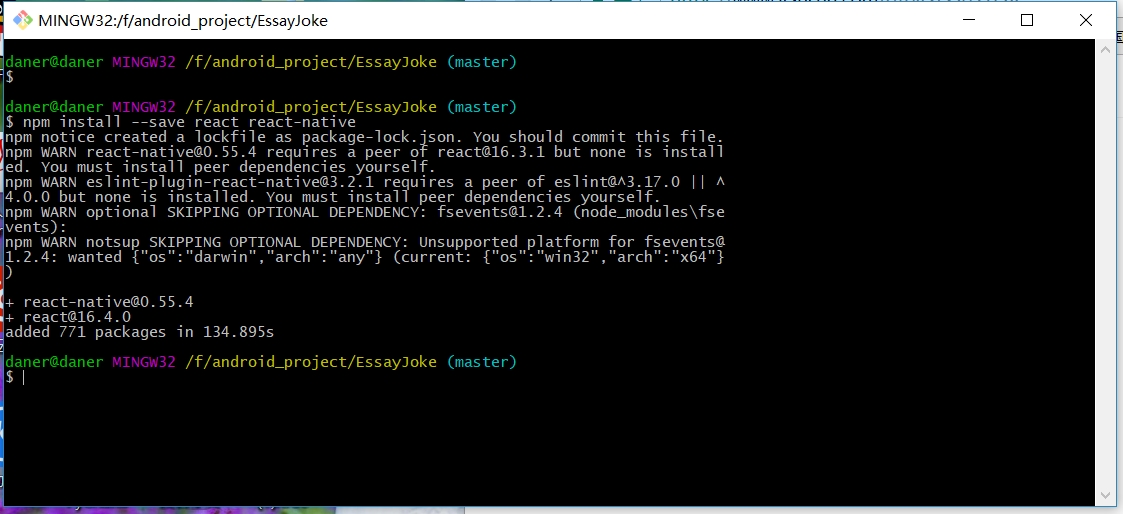
"start": "node node\_modules/react-native/local-cli/cli.js start"

注意下启动脚本的路径，如果集成的项目目录结构不一样的话会报找不到启动脚本之类的错误，到时根据具体目录结构修改就好了。

#### 2 安装React 和React Native

根目录输入以下命令

npm install --save react react-native

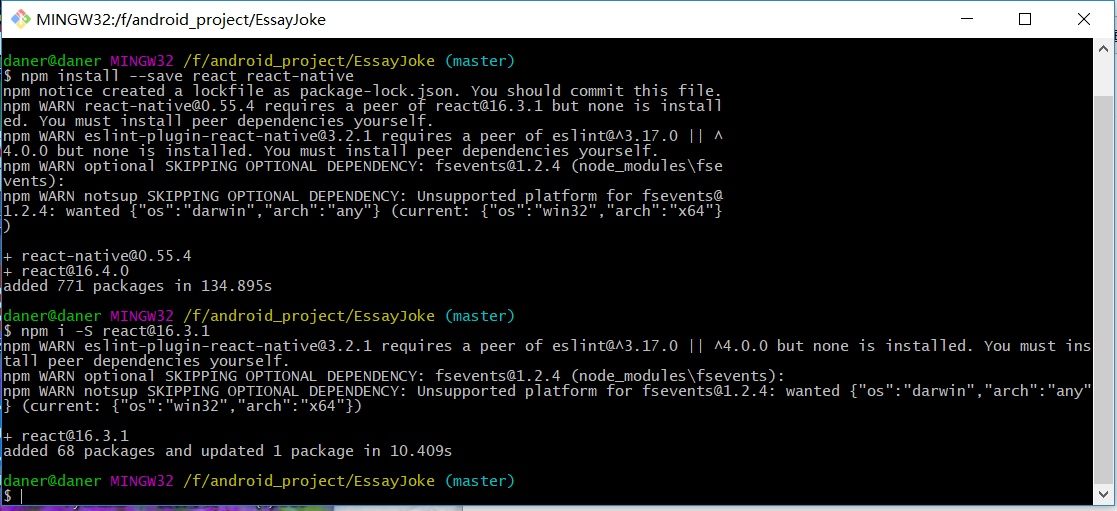


大约一两分钟的样子（如果卡到这里了，看看安装时是不是忘了配置镜像），完成之后你的根目录下会多了一个node\_modules的文件夹，里面存放了下载好的React 和React Native。

要是控制台输出了版本不一致的警告信息，例如：

npm WARN react-native@0.45.1 requires a peer of react@16.3.1 but none was installed.

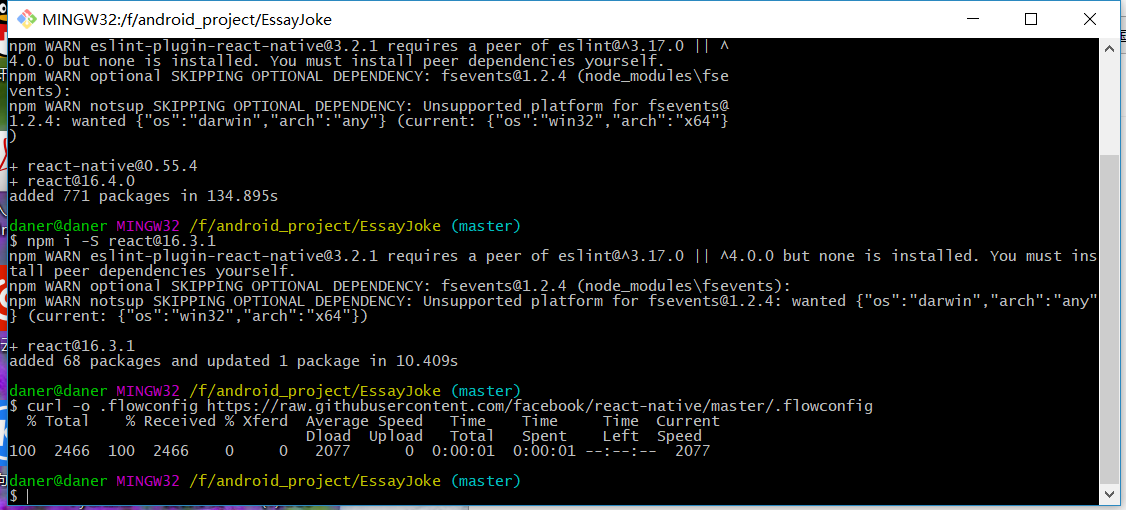
则继续执行 npm i -S react@16.3.1 (这里版本跟警告信息一致)。



#### 3 下载.flowconfig文件(非必须)

curl -o .flowconfig https://raw.githubusercontent.com/facebook/react-native/master/.flowconfig

第三步curl命令，其实质是下载.flowconfig配置文件，这个文件用于约束js代码的写法。非必需，可跳过。



#### 4 新建index.android.js

项目根目录下创建index.android.js文件，目录内容如下：

'use strict';

import React from 'react';

import {

AppRegistry,

StyleSheet,

Text,

View

} from 'react-native';

class EssayJokeReactNativeDemo extends React.Component {

render() {

return (

<View style={styles.container}>

<Text style={styles.hello}>Hello, EssayJoke React-NativeDemo</Text>

</View>

)

}

}

var styles = StyleSheet.create({

container: {

flex: 1,

justifyContent: 'center',

},

hello: {

fontSize: 20,

textAlign: 'center',

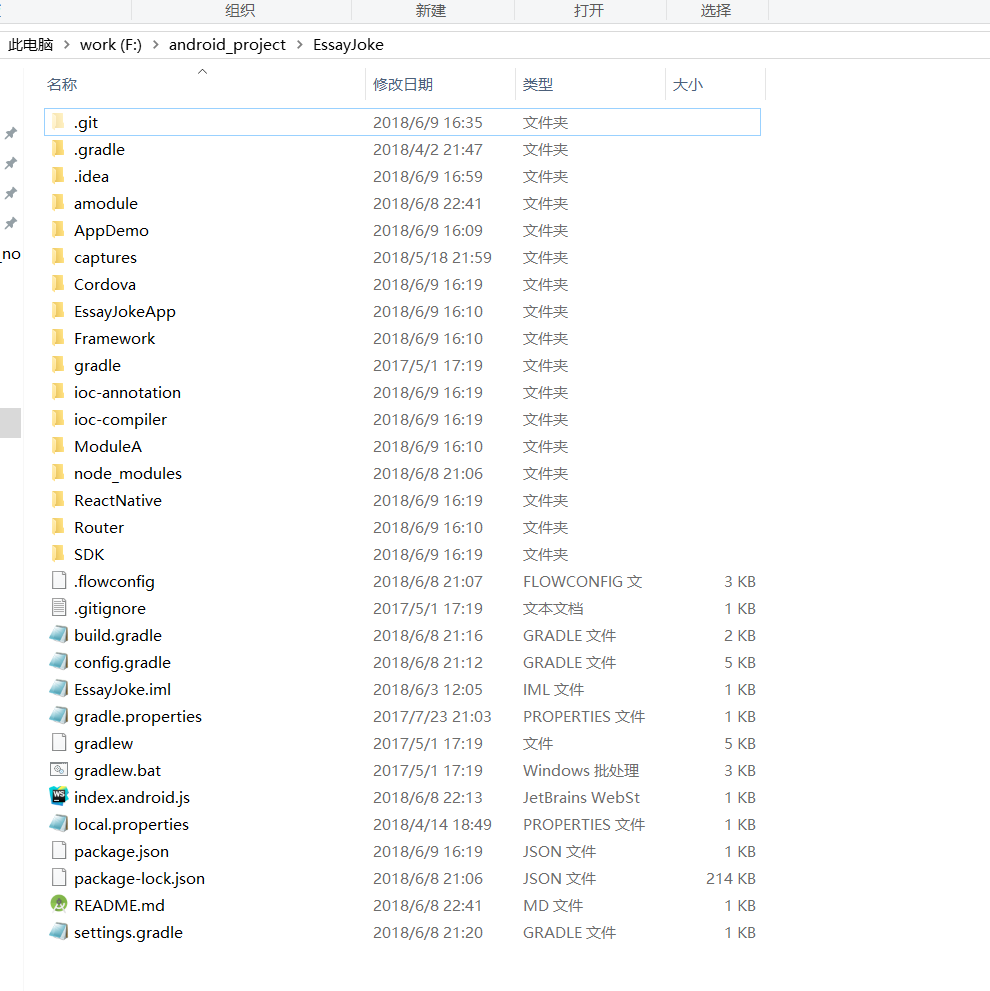
margin: 10,

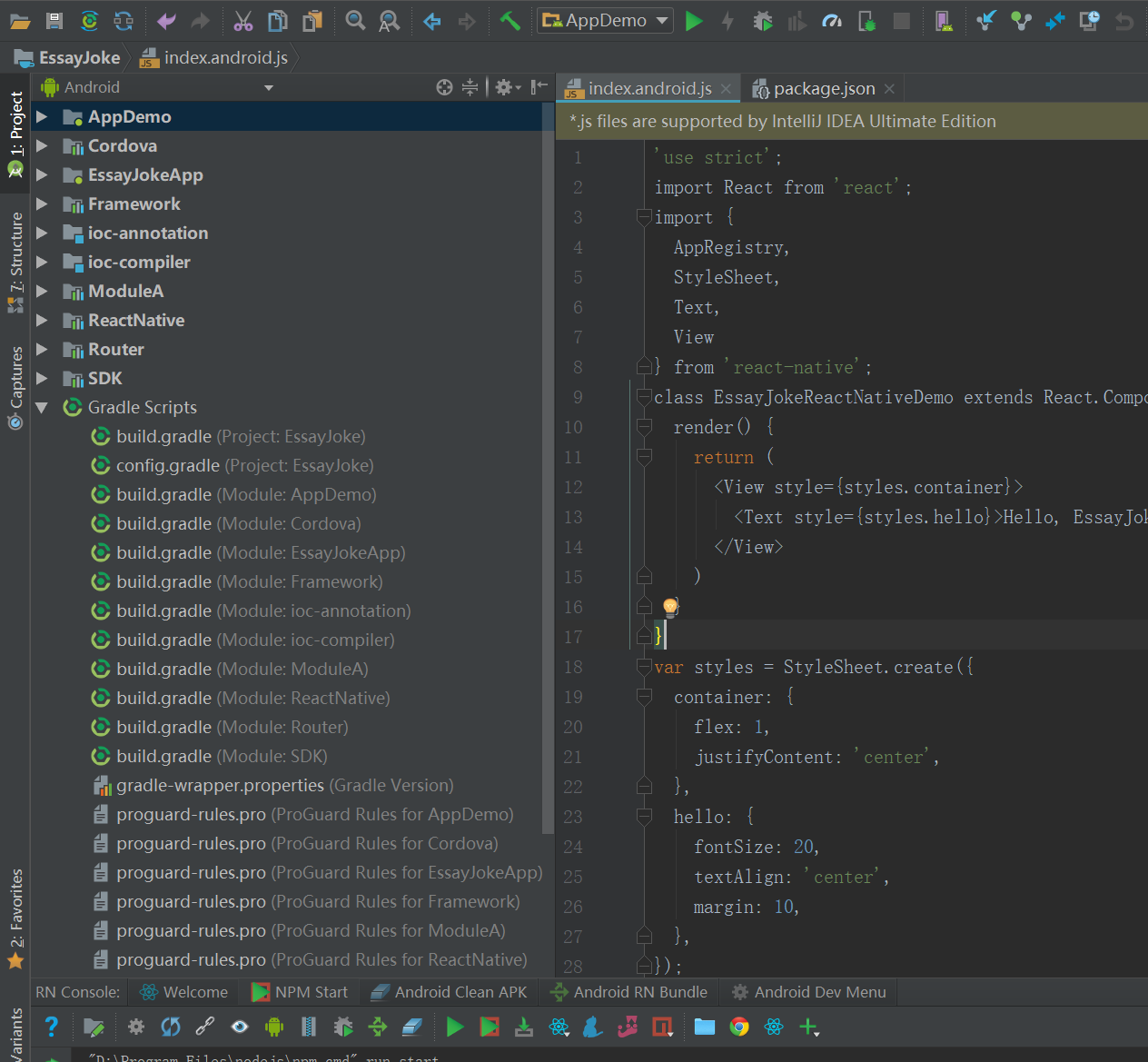
},

});

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

到这里js部分的都弄的差不多了，现在来看整个项目的目录结构如下：





ReactNative Module就是我们ReactNative相关开发的内容都放在该module下

#### 5 ReactNative Module引入依赖RN依赖

在build.gradle引入以下依赖

com\_facebook\_react\_react\_native = "com.facebook.react:react-native:+"

api com\_facebook\_react\_react\_native // From node\_modules.

在项目根目录build.gradle添加依赖仓库地址

allprojects {

repositories {

…

maven {

// All of React Native (JS, Android binaries) is installed from npm

url "$rootDir/node\_modules/react-native/android"

}

}

}

这里maven路径官网上写的是$rootDir/../node\_modules/react-native/android，这个跟目录结构有关，跟我一样在根目录上操作的话是不需要$rootDir后面的..的，它指的路径就是项目根目录下的node\_modules下的路径

#### 6 新建Activity作为原生与RN的跳转

在较老的RN版本中，我们新建一个Activity实现DefaultHardwareBackBtnHandler接口，大体类似下面

public class MyReactActivity extends AppCompatActivity implements DefaultHardwareBackBtnHandler {

private ReactRootView mReactRootView;

private ReactInstanceManager mReactInstanceManager;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

mReactRootView = new ReactRootView(this);

mReactInstanceManager = ReactInstanceManager.builder()

.setApplication(getApplication())

.setBundleAssetName("index.android.bundle")

.setJSMainModuleName("index.android")

.addPackage(new MainReactPackage())

.setUseDeveloperSupport(BuildConfig.DEBUG)

.setInitialLifecycleState(LifecycleState.RESUMED)

.build();

// 注意这里的HelloWorld必须对应“index.android.js”中的

// “AppRegistry.registerComponent()”的第一个参数

mReactRootView.startReactApplication(mReactInstanceManager, "test", null);

setContentView(mReactRootView);

}

@Override

public void invokeDefaultOnBackPressed() {

super.onBackPressed();

}

@Override

protected void onPause() {

super.onPause();

if (mReactInstanceManager != null) {

mReactInstanceManager.onHostPause(this);

}

}

@Override

protected void onResume() {

super.onResume();

if (mReactInstanceManager != null) {

mReactInstanceManager.onHostResume(this, this);

}

}

@Override

protected void onDestroy() {

super.onDestroy();

if (mReactInstanceManager != null) {

mReactInstanceManager.onHostDestroy();

}

}

@Override

public void onBackPressed() {

if (mReactInstanceManager != null) {

mReactInstanceManager.onBackPressed();

} else {

super.onBackPressed();

}

}

@Override

public boolean onKeyUp(int keyCode, KeyEvent event) {

if (keyCode == KeyEvent.KEYCODE\_MENU && mReactInstanceManager != null) {

mReactInstanceManager.showDevOptionsDialog();

return true;

}

return super.onKeyUp(keyCode, event);

}

}

较新的RN版本，例如RN0.55 我们直接继承ReactActivity即可

/\*\*

\* @author Created by qiyei2015 on 2018/6/8.

\* @version: 1.0

\* @email: 1273482124@qq.com

\* @description:

\*/

public class ReactNativeDemoActivity extends ReactActivity{

private static final String MAIN\_COMPONENT = "EssayJokeReactNativeDemo";

/\*\*

\* 返回在index.android.js 中注册的组件名

\* @return

\*/

@Nullable

@Override

protected String getMainComponentName() {

return MAIN\_COMPONENT;

}

}

新建一个Activity，让其继承ReactActivity，并重写getMainComponentName(),返回我们在index.android.js

中注册的HelloWorld这个组件。

在Application中初始化一下

/\*\*

\* Email: 1273482124@qq.com

\* Created by qiyei2015 on 2017/5/8.

\* Version: 1.0

\* Description:

\*/

public class BaseApplication extends Application implements ReactApplication{

private static final String TAG = " Sophix";

/\*\*

\* ReactNative的host

\*/

private final ReactNativeHost mReactNativeHost = new ReactNativeHost(this) {

@Override

public boolean getUseDeveloperSupport() {

return BuildConfig.DEBUG;

}

@Override

protected List<ReactPackage> getPackages() {

return Arrays.<ReactPackage>asList(

new MainReactPackage()

);

}

};

@Override

public void onCreate() {

super.onCreate();

....

}

/\*\*

\* ReactNative的host

\* @return

\*/

@Override

public ReactNativeHost getReactNativeHost() {

return mReactNativeHost;

}

....

}

在manifest.xml配置如下：

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.qiyei.reactnative">

<application>

<activity android:name="com.facebook.react.devsupport.DevSettingsActivity" />

<activity android:name=".ReactNativeDemoActivity" />

</application>

</manifest>

由于需要访问 DevSettingsActivity 界面，也需要在 AndroidManifest.xml 中声明

#### 7 同步编译

该步骤可能会遇到以下问题

Warning:Conflict with dependency 'com.google.code.findbugs:jsr305'. Resolved versions for app (3.0.0) and test app (2.0.1) differ. See http://g.co/androidstudio/app-test-app-conflict for details.

在build.bundle里面加入：

allprojects {

...

configurations.all {

resolutionStrategy.force 'com.google.code.findbugs:jsr305:3.0.0'

}

}

#### 8 在ReactNative Module新建assets目录

### 2 在AS中运行RN项目

有了以上准备，我们就直接来运行项目吧

#### 1 启动npm server

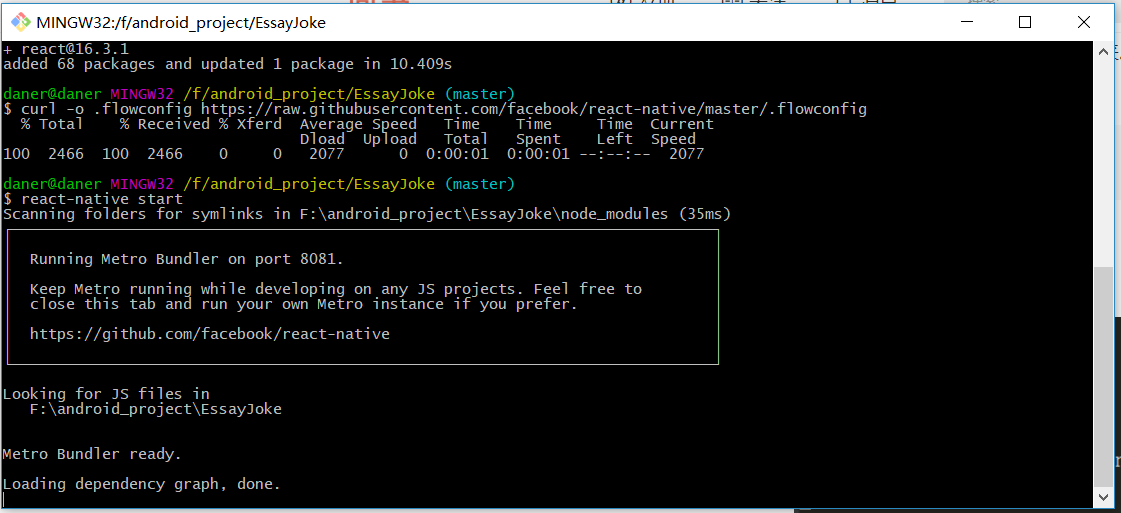
在根目录下输入以下命令

npm start

或者

react-native start

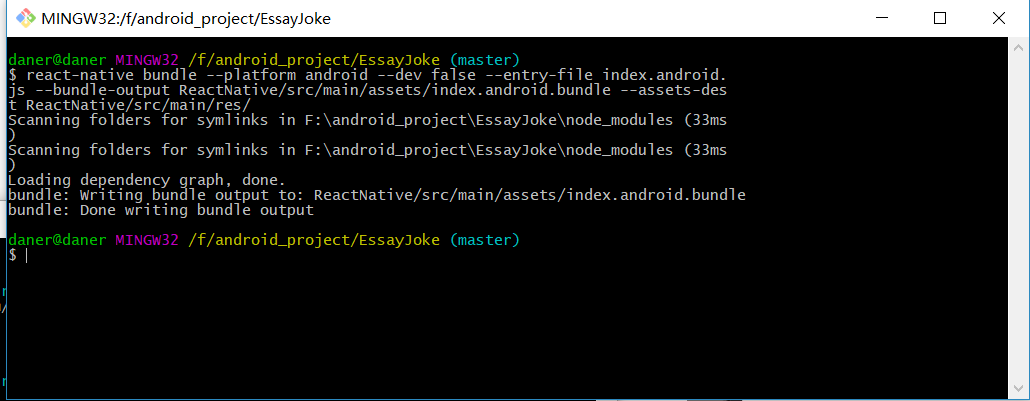
都可以启动后台服务



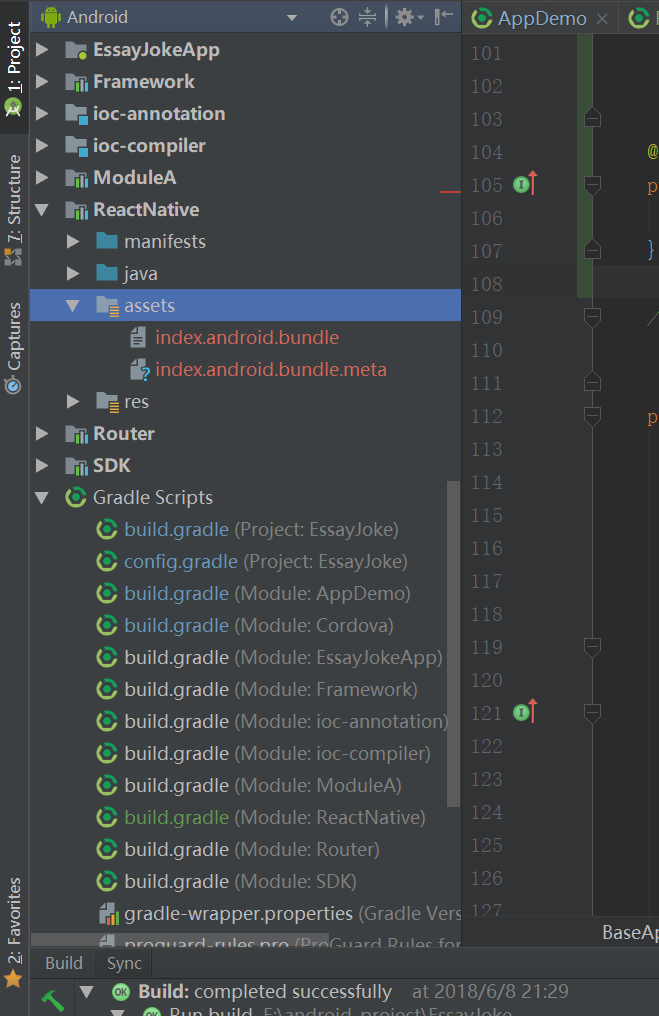
如果卡在了以上界面，直接在根目录下执行以下命令

react-native bundle --platform android --dev false --entry-file index.android.js --bundle-output ReactNative/src/main/assets/index.android.bundle --assets-dest ReactNative/src/main/res/

注意，需要将index.android.bundle生成的目录指定对



完成之后assets目录下会生成以下两个文件



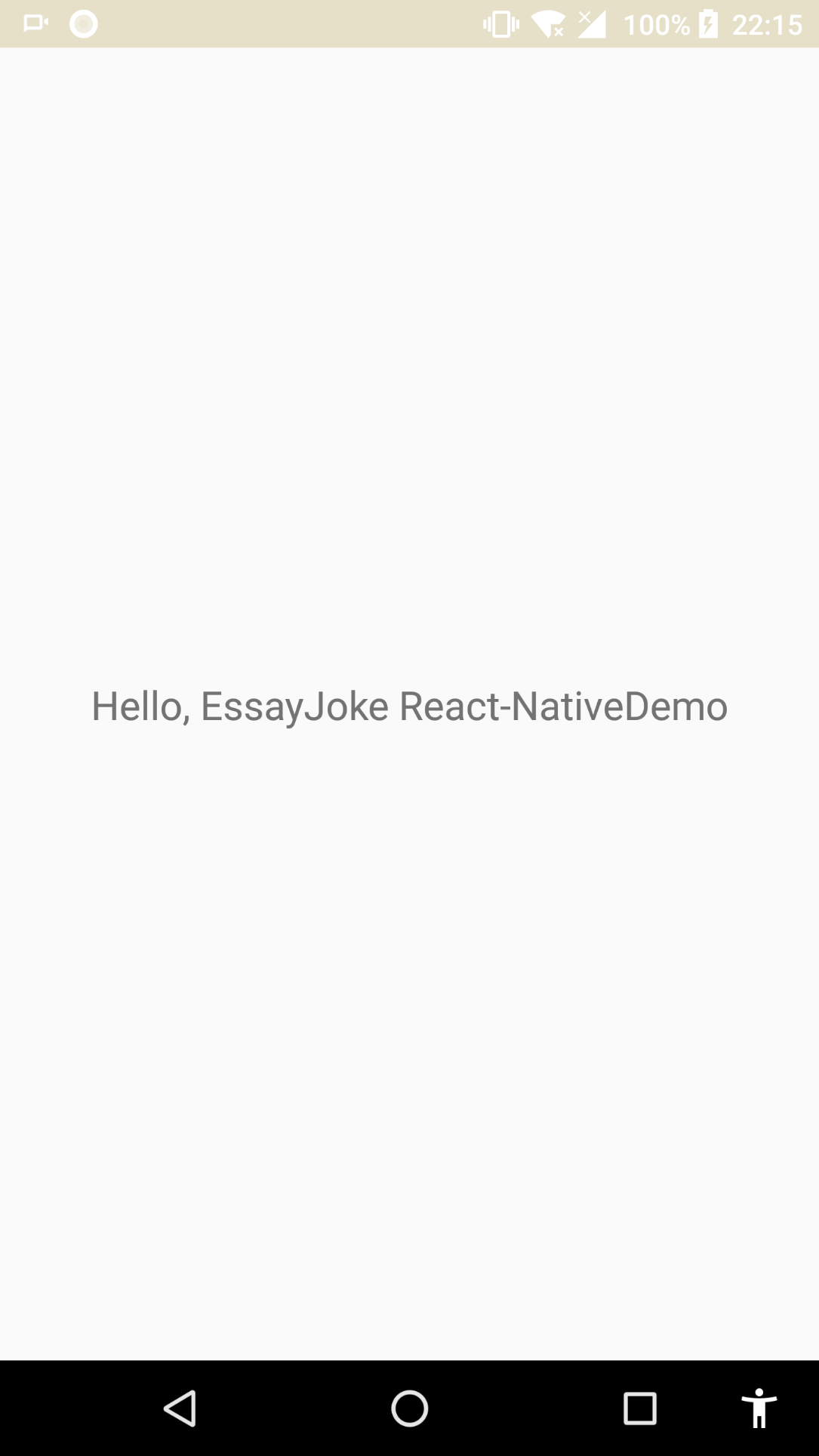
#### 2 监测npm server是否可以访问

在电脑的浏览器中访问以下地址，如果可以访问，一般来说server就启动成功了

http://localhost:8081/index.android.bundle?platform=android

#### 3 运行APP

点击跳转到RN界面，运行效果如下：



注意，运行过程中可能会有一些不顺利的地方，可以搜索网络上的答案解决

4

react-native bundle --platform android --dev false --entry-file index.js --bundle-output android/app/src/main/assets/index.android.bundle --assets-dest android/app/src/main/res/

## 三 调试RN项目

1 chrome安装React Developer Tools

2 webstorm 中点击debug

3 启动android 模拟器

查看模拟器 emulator -list-avds

emulator –avd Pixel\_2\_XL\_API\_27

打开调试 ctrl + M 打开调试菜单

ctrl + shift + I 打开chrome 调试菜单

<http://idea.codebeta.cn>

## 四 Atom环境搭建

1 安装以下插件

emmet

atom-beautify

autocomplete-paths

api-docs

linter

linter-ui-default

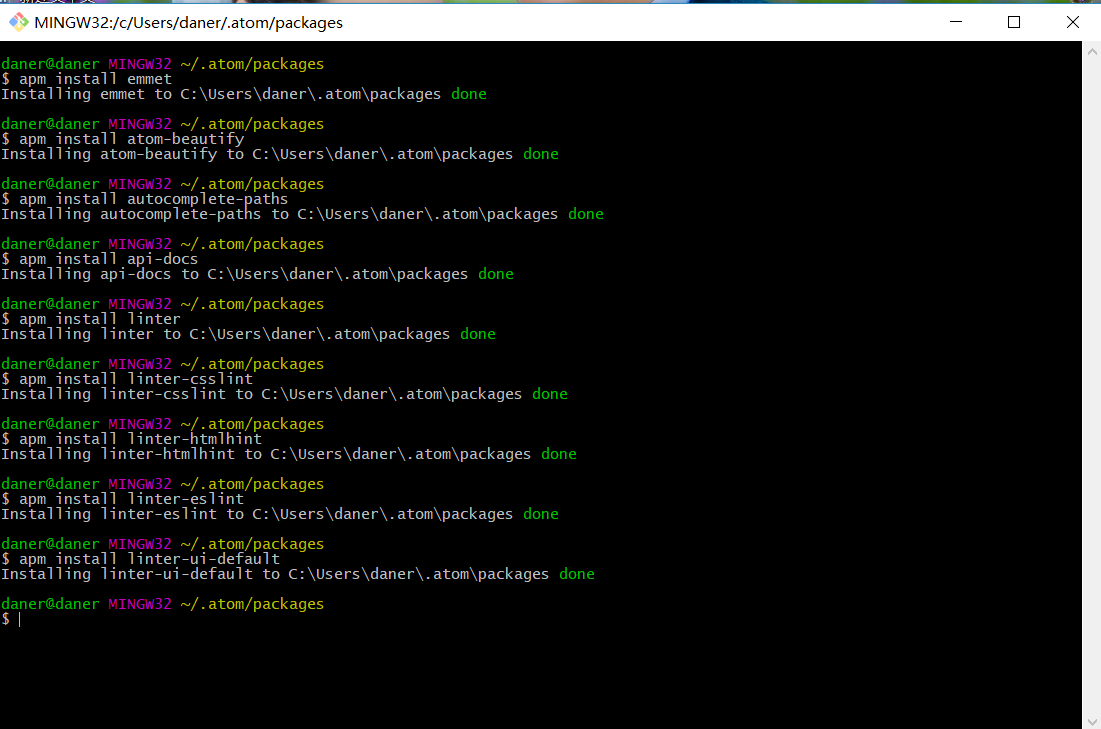
linter-csslint

linter-htmlhint

linter-eslint

atom-runner

atom-ternjs



intentions, busy-signal

安装browser-sync

npm install –g browser-sync

2 windows 命令

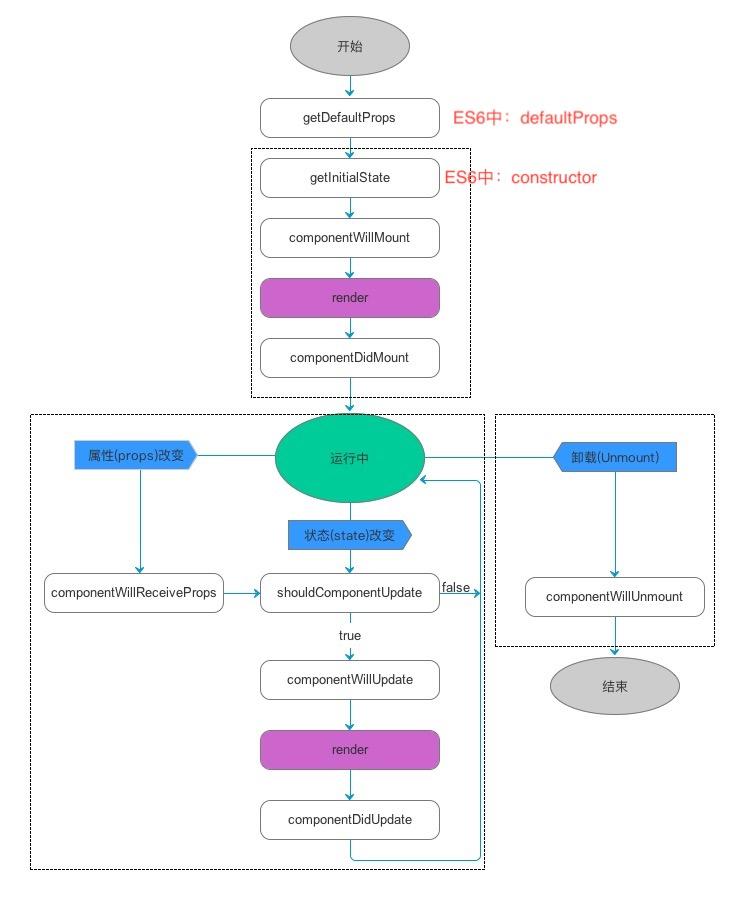
echo %cd%

# 第二部分 ReactNative 基础

## 一 React-Native 组件生命周期

RN 是由很多组件构成，在React Native中使用组件来封装界面模块时，整个界面就是一个大的组件，开发过程就是不断优化和拆分界面组件、构造整个组件树的过程。

理解组件生命周期很重要，下面是RN组件的整个示意图



组件的生命周期分为四个阶段 创建阶段 实例化阶段 运行(更新)阶段 销毁阶段

### 1 创建阶段

该阶段主要发生在创建组件类的时候，在这个阶段中会初始化组件的属性类型和默认属性。

#### 1 defaultProps

这里会初始化一些默认的属性，通常会将固定的内容放在这个过程中进行初始化和赋值，一个控件可以利用this.props获取在这里初始化它的属性，由于组件初始化后，再次使用该组件不会调用defaultProps函数，所以组件自己不可以自己修改props（即：props可认为是只读的），只可由其他组件调用它时在外部修改。

在ES6里，可以统一使用static成员来实现.

//定义默认属性

static defaultProps = {

autoPlay:false,

maxLoops:10,

};

//定义proTypes

static propTypes = {

autoPlay:PropTypes.bool.isRequired,

maxLoops:PropTypes.number.isRequired,

};

### 2 实例化阶段

该阶段主要发生在组件类被调用(实例化)的时候。

组件类被实例化的时候，触发一系列流程：

#### 1 constructor(props)

这里是对控件的一些状态进行初始化，由于该函数不同于defaultProps，在以后的过程中，会再次调用，所以可以将控制控件的状态的一些变量放在这里初始化，如控件上显示的文字，可以通过this.state来获取值，通过this.setState来修改state值。

在ES6里，通过constructor(构造器)对状态进行初始化

//组件实例化时调用 可调用多次

constructor(props){

super(props);

console.log("constructor");

this.state = {

loopsRemaining:this.props.maxLoops,

};

}

#### 2 componentWillMount()

准备加载组件

这个调用时机是在组件创建，并初始化了状态之后，在第一次绘制 render() 之前。可以在这里做一些业务初始化操作，也可以设置组件状态。这个函数在整个生命周期中只被调用一次。

如果在这个函数里面调用setState，本次的render函数可以看到更新后的state，并且只渲染一次。

//实例化 准备加载组件

componentWillMount() {

console.log('componentWillMount');

}

#### 3 render()

render是一个组件必须有的方法，形式为一个函数，渲染界面，并返回JSX或其他组件来构成DOM，和Android的XML布局、WPF的XAML布局类似，只能返回一个顶级元素。

render() {

console.log("render()");

return (

<View style={styles.container}>

<Text style={styles.welcome}>

Welcome to React Native!

</Text>

<Text style={styles.instructions}>

To get started, edit App.js

</Text>

<Text style={styles.instructions}>

{instructions}

</Text>

<Greeting name='React 1'/>

<Greeting name='React 2'/>

<Greeting name='React 3'/>

<TouchableHighlight onPress={this.\_onPressButton}>

<Text>Button</Text>

</TouchableHighlight>

</View>

);

}

#### 4 componentDidMount()

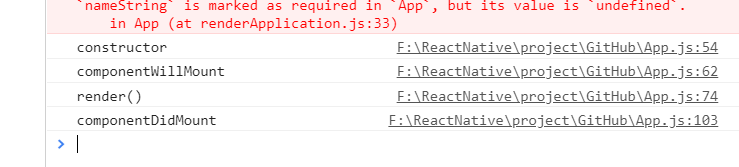
在组件第一次绘制之后，会调用 componentDidMount()，通知组件已经加载完成。这个函数调用的时候，其虚拟 DOM 已经构建完成，你可以在这个函数开始获取其中的元素或者子组件了。需要注意的是，RN 框架是先调用子组件的 componentDidMount()，然后调用父组件的函数。从这个函数开始，就可以和 JS 其他框架交互了，例如设置计时 setTimeout 或者 setInterval，或者发起网络请求。这个函数也是只被调用一次。这个函数之后，就进入了稳定运行状态，等待事件触发。

componentDidMount() {

console.log('componentDidMount');

}

运行结果如下：



### 3 运行(更新)阶段

该阶段主要发生在用户操作之后，或者父组件有更新的时候，此时会根据用户的操作行为，进行相应的界面结构调整。

触发的流程如下：

#### 1 componentWillReceiveProps(nextProps)

当组件接收到新的props时，会触发该函数。在该函数中，通常可以调用setState()来完成对state的修改。

输入参数 nextProps 是即将被设置的属性，旧的属性还是可以通过 this.props 来获取。在这个回调函数里面，你可以根据属性的变化，通过调用 this.setState() 来更新你的组件状态，这里调用更新状态是安全的，并不会触发额外的 render() 调用。如下

/\*\*

\* 在组件渲染完成之后，当reactNative组件接受到新的props时，这个函数将被调用，这个函数接受一个object参数，object里时新的props。

\* @param nextProps 新的props

\*/

componentWillReceiveProps(nextProps) {

this.setState({loopsRemaining:nextProps.maxLoops});

console.log('componentWillReceiveProps' + nextProps.value);

}

#### 2 shouldComponentUpdate(nextProps, nextState)

返回布尔值（决定是否需要更新组件）

输入参数 nextProps 和上面的 componentWillReceiveProps 函数一样，nextState 表示组件即将更新的状态值。这个函数的返回值决定是否需要更新组件，如果 true 表示需要更新，继续走后面的更新流程。否者，则不更新，直接进入等待状态。

默认情况下，这个函数永远返回 true 用来保证数据变化的时候 UI 能够同步更新。在大型项目中，你可以自己重载这个函数，通过检查变化前后属性和状态，来决定 UI 是否需要更新，能有效提高应用性能。

//返回布尔值（决定是否需要更新组件）

shouldComponentUpdate(nextProps,nextState){

console.log("shouldComponentUpdate");

return true;

}

#### 3 componentWillUpdate(nextProps, nextState)

houldComponentUpdate返回true或者调用forceUpdate之后，就会开始准更新组件，并调用 componentWillUpdate()。

//组件将要更新

componentWillUpdate(nextProps, nextState){

console.log("componentWillUpdate");

}

输入参数与 shouldComponentUpdate 一样，在这个回调中，可以做一些在更新界面之前要做的事情。需要特别注意的是，在这个函数里面，你就不能使用 this.setState 来修改状态。这个函数调用之后，就会把 nextProps 和 nextState 分别设置到 this.props 和 this.state 中。紧接着这个函数，就会调用 render() 来更新界面了。

#### 4 render()

再确定需要更新组件时，调用render，根据diff算法，渲染界面，生成需要更新的虚拟DOM数据。

#### 5 componentDidUpdate()

虚拟DOM同步到DOM中后，执行该方法，可以在这个方法中做DOM操作。

//组件已经更新

componentDidUpdate() {

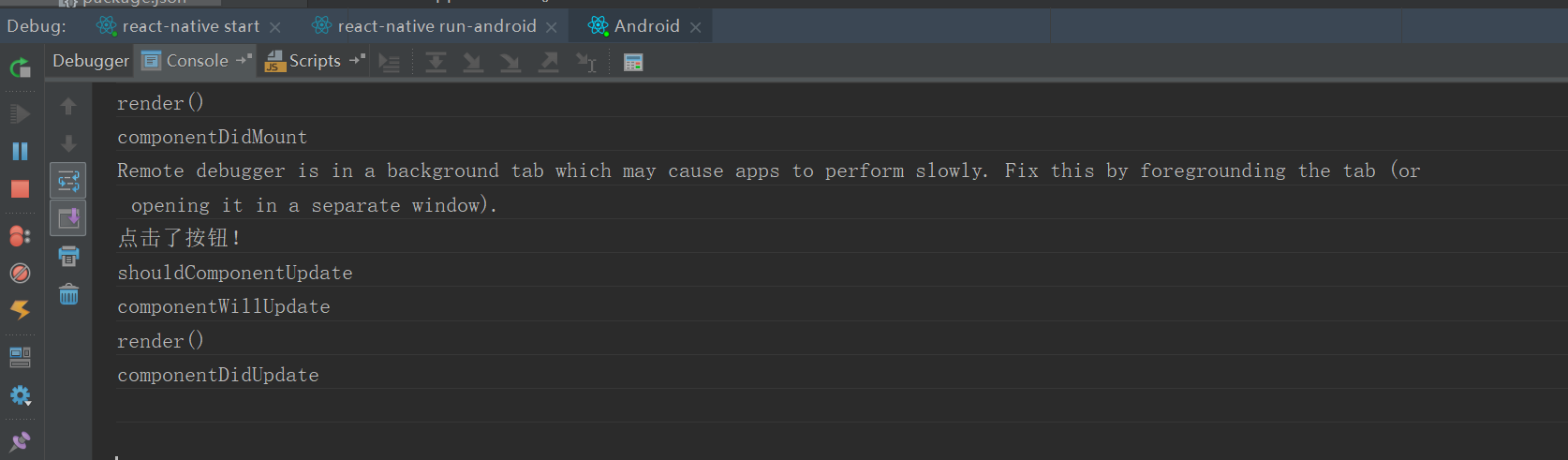
console.log("componentDidUpdate");

}

除了首次render之后调用componentDidMount，其它render结束之后都是调用componentDidUpdate。

componentWillMount、componentDidMount和componentWillUpdate、componentDidUpdate可以对应起来。区别在于，前者只有在挂载的时候会被调用；而后者在以后的每次更新渲染之后都会被调用。

**注意：绝对不要在componentWillUpdate和componentDidUpdate中调用this.setState方法，否则将导致无限循环调用。**



### 4 销毁阶段

#### 1 componentWillUnmount()

该阶段主要发生组件销亡的时候，触发componentWillUnmount。当组件需要从DOM中移除的时候，通常需要做一些取消事件绑定，移除虚拟DOM中对应的组件数据结构，销毁一些无效的定时器等工作，都可以在这个方法中处理。

当组件要被从界面上移除的时候，就会调用 componentWillUnmount。

//销毁阶段 组件将要销毁

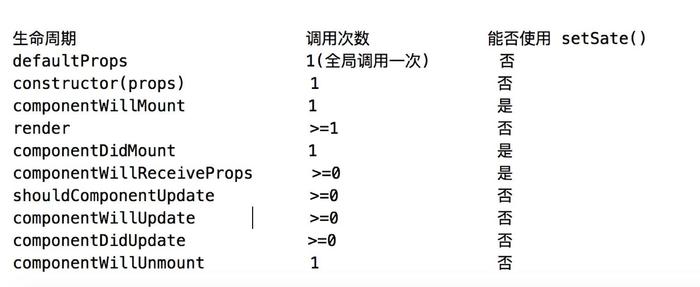
componentWillUnmount(){

console.log("componentWillUnmount");

}

在这个函数中，可以做一些组件相关的清理工作，例如取消计时器、网络请求等。

### 5 总结



## 二 RN Props与State

### 1 Props属性

属性（props）：组件的props是不可变的，它只能从其他的组件（例如父组件）传递过来。

实例：

import React, { Component } from 'react';

import {Text} from "react-native";

import { PropTypes} from 'prop-types';

export default class PropsTest extends Component{

//默认属性

static defaultProps = {

name:"",

};

//属性检查

static propTypes = {

name:PropTypes.string.isRequired, //必须传递

};

render(){

return(

<Text>name:{this.props.name}</Text>

);

}

}

使用：

<PropsTest name={12356}></PropsTest>

技巧：使用ES6的延展操作符来传递属性

### 2 State状态

state是组件的状态描述，一般为组件私有。

组件的state是可变的，它负责处理与用户的交互。在通过用户点击事件等操作以后，如果使得当前组件的某个state发生了改变，那么当前组件就会触发render()方法刷新自己。

import React, { Component } from 'react';

import {Image, Text, View, StyleSheet} from "react-native";

import { PropTypes} from 'prop-types';

export default class StateTest extends Component{

//默认属性

static defaultProps = {

name:"",

};

//属性检查

static propTypes = {

name:PropTypes.string.isRequired, //必须传递

};

//方式2

state={

size:80,

};

constructor(props){

super(props);

//方式1

// this.state={

// size:80,

// };

}

render(){

return(

<View>

<Text style={{fontSize:30}} onPress={this.press1.bind(this)}>吹气球</Text>

<Image style={{width:this.state.size, height:this.state.size,}} source={require('../../res/img/qiqiu.jpg')}></Image>

</View>

);

}

//点击事件

press1(){

this.setState({

size:this.state.size+10

})

}

}

const styles = StyleSheet.create({

container:{

},

});

### 3 ref使用

用于虚拟结点加载后的引用结点，从而获取结点的属性等

import React, { Component } from 'react';

import {Image, Text, View, StyleSheet} from "react-native";

import { PropTypes} from 'prop-types';

import Counter from "../component/Counter";

export default class RefTest extends Component{

//默认属性

static defaultProps = {

name:"",

};

//属性检查

static propTypes = {

name:PropTypes.string.isRequired, //必须传递

};

//方式2

state={

count:0,

};

constructor(props){

super(props);

}

render(){

return(

<View>

<Counter ref={"counterRef"}/>

<Text style={{fontSize:30}} onPress={this.press1.bind(this)}>Counter:{this.state.count}</Text>

</View>

);

}

//点击事件

press1(){

var count = this.refs.counterRef.getCount();

this.setState({

count:count

})

}

}

const styles = StyleSheet.create({

container:{

},

});

## 三 Flex布局

### 1 RN的FlexBox与Web CSS FlexBox不同

值得一提的是，React Native中的FlexBox 和Web CSSS上FlexBox工作方式是一样的。但有些地方还是有些出入的，如：

#### 1 flexDirection

React Native中默认为flexDirection:'column'，在Web CSS中默认为flex-direction:'row'

RN默认是纵向排列的，Web是横向

#### 2 alignItems

React Native中默认为alignItems:'stretch'，在Web CSS中默认align-items:'flex-start'

#### 3 flex

Web CSS的flex接受多参数，如:flex: 2 2 10%;，但在 React Native中flex只接受一个参数

#### 4 不支持属性

RN中不支持align-content，flex-basis，order，flex-basis，flex-flow，flex-grow，flex-shrink

### 2 width与height

在学习FlexBox之前首先要清楚一个概念“宽和高”。一个组件的高度和宽度决定了它在屏幕上的尺寸，也就是大小。

#### 1 像素无关

在React Native中尺寸是没有单位的，它代表了设备独立像素

<View style={ {width:100,height:100,margin:40,backgroundColor:'gray'}}>

<Text style={ {fontSize:16,margin:20}}>尺寸</Text>

</View>

上述代码，运行在Android上时，View的长和宽被解释成：100dp 100dp单位是dp，字体被解释成16sp 单位是sp，运行在iOS上时尺寸单位被解释称了pt，这些单位确保了布局在任何不同dpi的手机屏幕上显示不会发生改变

### 3 父视图属性(容器属性)

容器属性主要有以下四个

**flexDirection** enum(‘row’, ‘column’,’row-reverse’,’column-reverse’)

**flexWrap** enum(‘wrap’, ‘nowrap’)

**justifyContent** enum(‘flex-start’, ‘flex-end’, ‘center’, ‘space-between’, ‘space-around’)

**alignItems** enum(‘flex-start’, ‘flex-end’, ‘center’, ‘stretch’)

#### 1 flexDirection

属性定义了父视图中的子元素沿横轴或侧轴方片的排列方式。

* row: 从左向右依次排列
* row-reverse: 从右向左依次排列
* column(default): 默认的排列方式，从上向下排列
* column-reverse: 从下向上排列

#### 2 flexWrap

flexWrap属性定义了子元素在父视图内是否允许多行排列，默认为nowrap。

* nowrap flex的元素只排列在一行上，可能导致溢出。
* wrap flex的元素在一行排列不下时，就进行多行排列。

#### 3 justifyContent

justifyContent属性定义了如何分配顺着父容器主轴的弹性（flex）元素之间及其周围的空间，默认为flex-start

* flex-start(default) 从行首开始排列。每行第一个弹性元素与行首对齐，同时所有后续的弹性元素与前一个对齐。
* flex-end 从行尾开始排列。每行最后一个弹性元素与行尾对齐，其他元素将与后一个对齐。
* center 伸缩元素向每行**居中分配**。每行第一个元素到行首的距离将与每行最后一个元素到行尾的距离相同。
* space-between 在每行上**均匀分配**弹性元素。相邻元素间距离相同。每行第一个元素与行首对齐，每行最后一个元素与行尾对齐。
* space-around 在每行上均匀分配弹性元素。相邻元素间距离相同。**每行第一个元素到行首的距离和每行最后一个元素到行尾的距离将会是相邻元素之间距离的一半。**

注：这里的对齐是指纵向对齐，对齐策略由alignItems指定

#### 4 alignItems

alignItems属性以与justify-content相同的方式在侧轴方向上将当前行上的弹性元素对齐，默认为stretch。

* flex-start 元素向侧轴起点对齐。
* flex-end 元素向侧轴终点对齐。
* center 元素在侧轴居中。如果元素在侧轴上的高度高于其容器，那么在两个方向上溢出距离相同。
* stretch 弹性元素被在侧轴方向被拉伸到与容器相同的高度或宽度。

### 4 子视图属性

#### 1 alignSelf

lignSelf属性以属性定义了flex容器内被选中项目的对齐方式。注意：alignSelf 属性可重写灵活容器的 alignItems 属性。

* auto(default) 元素继承了它的父容器的 alignitems 属性。如果没有父容器则为 “stretch”。
* stretch 元素被拉伸以适应容器。
* center 元素位于容器的中心。
* flex-start 元素位于容器的开头。
* flex-end 元素位于容器的结尾。

#### 2 flex

属性定义了一个可伸缩元素的能力，默认为0。

### 5 其他布局

#### 1 视图边框

这个主要是CSS的属性

borderBottomWidth number 底部边框宽度

borderLeftWidth number 左边框宽度

borderRightWidth number 右边框宽度

borderTopWidth number 顶部边框宽度

borderWidth number 边框宽度

border<Bottom Left Right Top>Color 个方向边框的颜色

borderColor 边框颜色

#### 2 外边距

margin number 外边距

marginBottom number 下外边距

marginHorizontal number 左右外边距

marginLeft number 左外边距

marginRight number 右外边距

marginTop number 上外边距

marginVertical number 上下外边距

#### 3 内边距

padding number 内边距

paddingBottom number 下内边距

paddingHorizontal number 左右内边距

paddingLeft number 做内边距

paddingRight number 右内边距

paddingTop number 上内边距

paddingVertical number 上下内边距

#### 4 边缘

left number 属性规定元素的左边缘。该属性定义了定位元素左外边距边界与其包含块左边界之间的偏移。

right number 属性规定元素的右边缘。该属性定义了定位元素右外边距边界与其包含块右边界之间的偏移

top number 属性规定元素的顶部边缘。该属性定义了一个定位元素的上外边距边界与其包含块上边界之间的偏移。

bottom number 属性规定元素的底部边缘。该属性定义了一个定位元素的下外边距边界与其包含块下边界之间的偏移。

#### 5定位(position)

position enum(‘absolute’, ‘relative’)属性设置元素的定位方式，为将要定位的元素定义定位规则。

absolute：生成绝对定位的元素，元素的位置通过 “left”, “top”, “right” 以及 “bottom” 属性进行规定。

relative：生成相对定位的元素，相对于其正常位置进行定位。因此，”left:20” 会向元素的 LEFT 位置添加 20 像素。

## 四 事件处理

### 1 处理触摸事件

React Native提供了可以处理常见触摸手势（例如点击或滑动）的组件， 以及可用于识别更复杂的手势的完整的手势响应系统。

#### 1 可点击的组件

使用"Touchable"开头的一系列组件。这些组件通过onPress属性接受一个点击事件的处理函数。当一个点击操作开始并且终止于本组件时（即在本组件上按下手指并且抬起手指时也没有移开到组件外），此函数会被调用。

class MyButton extends Component {

\_onPressButton() {

console.log("You tapped the button!");

}

render() {

return (

<TouchableHighlight onPress={this.\_onPressButton}>

<Text>Button</Text>

</TouchableHighlight>

);

}

}

# 第三部分 React Native常用组件

## 一 Touchable组件

### 1 Touchable组件简介

Touchable组件是RN的按钮组件，一共有四大类

TouchableWithoutFeedback：不带任何反馈的可触摸组件

TouchableHighlight：在TouchableWithoutFeedback的基础上添加了当按下时背景会变暗的效果。

TouchableOpacity：相比TouchableHighlight在按下去会使背景变暗的效果，TouchableOpacity会在用户手指按下时降低按钮的透明度，而不会改变背景的颜色。

TouchableNativeFeedback：在Android上还可以使用TouchableNativeFeedback，它会在用户手指按下时形成类似水波纹的视觉效果。注意，此组件只支持Android。

### 2 Touchable组件使用

Touchable组件主要用作按钮使用，可以处理一下事件

#### 1 onPress function

当触摸操作结束时调用，但如果被取消了则不调用（譬如响应者被一个滚动操作取代），一般可用作点击事件

render(){

return(

<View style={styles.container}>

<TouchableOpacity onPress={this.\_onPress.bind(this)} }>

<Text>onPress</Text>

</TouchableOpacity>

</View>

);

}

\_onPress(){

console.log("onPress");

}

#### 2 onLongPress function

长按事件，长按按钮时触发

<TouchableOpacity onLongPress={this.\_onLongPress.bind(this)}>

<Text>onLongPress</Text>

</TouchableOpacity>

\_onLongPress(){

console.log("onLongPress");

}

#### 3 disabled bool

禁止使能，长用于禁止按下等操作，是Touchable组件的属性之一

<TouchableOpacity disabled={true}>

<Text>无法点击</Text>

</TouchableOpacity>

#### 4 onPressIn function

表示用户点击时开始按下时，类似于Android中的Action\_down事件

#### 5 onPressOut function

表示用户点击时将要结束时，类似于Android中的Action\_up事件

onPressIn 与onPressOut结合可以计算用户按下的时间。

<TouchableOpacity onPressIn={this.\_pressIn.bind(this)} onPressOut={this.\_pressOut.bind(this)}>

<Text>测试按下的时间</Text>

</TouchableOpacity>

\_pressIn(){

console.log("pressIn");

}

\_pressOut(){

console.log("pressOut");

}

## 二 Image组件

### 1 Image组件介绍

RN中Image组件主要用于加载图片，可加载静态图片，网络图片，以及原生图片，本地文件系统中图片资源

### 2 Image组件功能

加载图片一般使用Image的source属性

#### 1 加载静态图片

静态图片是指js下的图片资源，例如放在源码下的res的目录

加载静态图片主要方式是：使用Image的source属性，使用require来指定路径

<Image source={require(image\_path)}/>

注意，这里的path一般是相对路径，起点是当前js文件所在的路径

import React, { Component } from 'react';

import {Image, View} from "react-native";

const image\_path1 = '../../res/img/qiqiu.jpg';

export default class ImageDemo extends Component{

render(){

return (

<View>

<Image source={require(image\_path1)}/>

</View>

);

}

}

加载本地图片资源可以不指定大小，默认以图片资源大小。

#### 2 加载网络图片

加载网络图片资源主要用到source的uri属性

<Image source={{uri:image\_url}} style={{width:size,height:size}}/>

注意，加载网络图片必须制定image的大小，否则无法加载

import React, { Component } from 'react';

import {Image, View} from "react-native";

const image\_path1 = '../../res/img/qiqiu.jpg';

const imgae\_url = 'https://timgsa.baidu.com/timg?image&quality=80&size=b9999\_10000&sec=1531544047510&di=c76803f98e7e5e7b86ef403716ecd670&imgtype=0&src=http%3A%2F%2Fs8.sinaimg.cn%2Fmw690%2F006LDoUHzy7auXu0wVp67%26690';

export default class ImageDemo extends Component{

render(){

return (

<View>

<Image source={require(image\_path1)} style={{width:100,height:100,margin:5}}/>

<Image source={{uri:imgae\_url}} style={{width:100,height:100,margin:5}}/>

</View>

);

}

}

#### 3 加载原生图片

加载原生图片是指能加载Android中drawable中的图片，也能加载IOS中的图片，这里以一张在Android中drawable中logo.png

<Image source={{uri:image\_native\_path}} style={{width:100,height:100}}/>

注意，加载原生图片也必须制定大小，并且需要重新运行react-native run android

import React, { Component } from 'react';

import {Image, View} from "react-native";

const image\_path1 = '../../res/img/qiqiu.jpg';

const imgae\_url = 'https://timgsa.baidu.com/timg?image&quality=80&size=b9999\_10000&sec=1531544047510&di=c76803f98e7e5e7b86ef403716ecd670&imgtype=0&src=http%3A%2F%2Fs8.sinaimg.cn%2Fmw690%2F006LDoUHzy7auXu0wVp67%26690';

const image\_native\_1 = 'qiqiu';

export default class ImageDemo extends Component{

render(){

return (

<View>

<Image source={require(image\_path1)} style={{width:100,height:100,margin:5}}/>

<Image source={{uri:imgae\_url}} style={{width:200,height:200,margin:5}}/>

<Image source={{uri:image\_native\_1}} style={{width:100,height:100,margin:5}}/>

</View>

);

}

}

#### 4 加载文件系统图片

### 3 Image图片加载技巧

#### 1 图片缩放resizeMode

表示图片缩放模式

cover: 均匀缩放图像（保持图像的纵横比），使图像的尺寸（宽度和高度）等于或大于视图的相应尺寸（减去填充）。

contain: 均匀缩放图像（保持图像的纵横比），使图像的尺寸（宽度和高度）等于或小于视图的相应尺寸（减去填充）。

stretch: 独立缩放宽度和高度，这可能会改变图片的宽高比

repeat: 重复图像以覆盖视图的框架。 图像将保持其大小和纵横比，除非它大于视图，在这种情况下，它将被均匀缩小以使其包含在视图中。IOS支持，Android不支持

center: 使图像沿两个维度居中。 如果图像大于视图，请将其均匀缩小以使其包含在视图中。

默认属性是cover

## 三 使用导航器navigation

### 1 安装react-navigation

yarn add react-navigation或者npm install --save react-navigation

### 2 StackNavigator导航组件

#### 1 navigation prop

当导航器中的屏幕被打开时，它会收到一个navigation prop，navigation prop是整个导航环节的关键一员，接下来就详细讲解一下navigation的作用。

navigation包含一下功能：

navigate：跳转到其他界面；

state：屏幕的当前state；

setParams：改变路由的params；

goBack：关闭当前屏幕；

dispatch：向路由发送一个action；

1使用navigate进行界面之间的跳转

navigate(routeName, params, action)

routeName：要跳转到的界面的路由名，也就是在导航其中配置的路由名；

params：要传递给下一个界面的参数；

action：如果该界面是一个navigator的话，将运行这个sub-action

2 使用state的params

可以通过this.props.state.params来获取通过setParams()，或navigation.navigate()传递的参数。

3 使用setParams 改变route params

setParams: function setParams(params)： 我们可以借助setParams来改变route params，比如，通过setParams来更新页面顶部的标题，返回按钮等；

#### 2 navigationOptions props(屏幕导航选项)

navigationOptions：配置StackNavigator的一些属性。

title：标题，如果设置了这个导航栏和标签栏的title就会变成一样的，不推荐使用

header：可以设置一些导航的属性，如果隐藏顶部导航栏只要将这个属性设置为null

headerTitle：设置导航栏标题，推荐

headerBackTitle：设置跳转页面左侧返回箭头后面的文字，默认是上一个页面的标题。可以自定义，也可以设置为null

headerTruncatedBackTitle：设置当上个页面标题不符合返回箭头后的文字时，默认改成"返回"

headerRight：设置导航条右侧。可以是按钮或者其他视图控件

headerLeft：设置导航条左侧。可以是按钮或者其他视图控件

headerStyle：设置导航条的样式。背景色，宽高等

headerTitleStyle：设置导航栏文字样式

headerBackTitleStyle：设置导航栏‘返回’文字样式

headerTintColor：设置导航栏颜色

headerPressColorAndroid：安卓独有的设置颜色纹理，需要安卓版本大于5.0

gesturesEnabled：是否支持滑动返回手势，iOS默认支持，安卓默认关闭

StackNavigator 导航组件可以用来作为我们的页面导航使用。导航组件主要使用react-navigation的StackNavigator函数来创建。

相关源码如下：

const StackNavigator = (routeConfigs, config = {}) => {

const navigator = createStackNavigator(routeConfigs, config);

return createNavigationContainer(navigator);

};

export default StackNavigator;

RouteConfig配置如下：

screen：对应界面名称，需要填入import之后的页面

navigationOptions：同navigationOptions属性

mode：定义跳转风格

card：使用iOS和安卓默认的风格

modal：iOS独有的使屏幕从底部画出。类似iOS的present效果

headerMode：返回上级页面时动画效果

float：iOS默认的效果

screen：滑动过程中，整个页面都会返回

none：无动画

cardStyle：自定义设置跳转效果

transitionConfig： 自定义设置滑动返回的配置

onTransitionStart：当转换动画即将开始时被调用的功能

onTransitionEnd：当转换动画完成，将被调用的功能

path：路由中设置的路径的覆盖映射配置

initialRouteName：设置默认的页面组件，必须是上面已注册的页面组件

initialRouteParams：初始路由参数

我们创建的页面如下：

//创建页面路由

const App = createStackNavigator({

LauncherPage:{

screen:LauncherPage,

navigationOptions:{

gesturesEnabled:false, //手势打开

header:null,

headerText:null, //去除标题

}

},

FlexBox:{

screen:FlexBox,

navigationOptions:{

gesturesEnabled:false, //手势打开

header:null, //去除header

headerText:null, //去除标题

}

},

DemoTestPage:{

screen:DemoTestPage,

navigationOptions:{

gesturesEnabled:false, //手势打开

header:null,

headerText:null, //去除标题

}

},

MyTab:{

screen:MyTab,

navigationOptions:{

gesturesEnabled:false, //手势打开

header:null,

headerText:null, //去除标题

}

}

});

export default App

分别创建了三个页面的导航组件，

#### 3 创建StackNavigator路由表

使用StackNavigator创建路由表非常简单

const AppNavigator = StackNavigator({

HomePage:{

screen:HomePage,

navigationOptions:{

title:"HomePage",

}

},

Page1:{

screen:Page1,

//参数是props,{navigation} 表示解构赋值。箭头函数返回值作为navigationOptions对象,需要小括号括起来

navigationOptions: ({navigation}) => ({

//这里是TAB键上面那个

title:`${navigation.state.params.name}页面名`,

})

},

Page2:{

screen:Page2,

navigationOptions:{

title:"Page2",

}

},

Page3:{

screen:Page3,

}

},{

//全局配置参数

navigationOptions:{

// header:null,

}

});

这里创建了4个页面的路由，其中主页面是HomePage。会被默认加载，其他页面需要使用navigation导航过去

我们看HomePage页面

import React,{Component} from "react";

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

export default class HomePage extends Component{

//覆盖路由里面设置

static navigationOptions = {

title:"Home",

headerBackTitle:"主页返回",

};

render(){

//解构赋值 等价于const navigation = this.props.navigation;

const {navigation} = this.props;

return (

<View style={styles.container}>

<Text>HomePage</Text>

<Button

title="Go to Page1"

onPress={() => {

//动态传递属性到Page1

navigation.navigate('Page1',{name:"动态的"});

}}

/>

<Button

title="Go to Page2"

onPress={() => {

navigation.navigate('Page2',{name:"page2的页面"});

}}

/>

<Button

title="Go to Page3"

onPress={() => {

navigation.navigate('Page3',{title:"哈哈"});

}}

/>

</View>

);

}

}

const styles = StyleSheet.create({

container:{

flex:1,

justifyContent:"center",

alignItems:'center',

},

});

可以看到可以从主页面跳转到各个分页面

### 3 TabNavigator底部导航组件

### 4 DrawerNavigator导航组件

# 第五部分 RN开发中的问题

## 一 开发疑难点

1 Cannot resolve symbol 'PropTypes' less... (Ctrl+F1)

Validate ES6 imports

Cannot resolve symbol 'PropTypes' less... (Ctrl+F1)

Validate ES6 imports

## 二 环境问题

1 Remote debugger is in a background tab which may cause apps to perform slowly. Fix this by foregrounding the tab (or opening it in a separate window).

# 第六部分 React Native开发实战—GitHub

## 一 GitHub开发路线

### 1 技术路线

### 2 技术栈

1 JS

2 ECS6

3 React

4 Flexbox

5 AsyncStorage 数据存储

6 Native Modules 原生组件封装

第三方库

@react-native-check-box

@react-native-easy-toast

@react-native-splash -screen

@react-native-htmlview

@react-native

## 二 Github 项目结构

react-native-tab-navigator

<https://www.npmjs.com/package/react-native-tab-navigator>