React-Navite

React-Navite 是 Facebook 开源的一款可以使用 JavaScript 构建原生应用的框架。

React-Navite 简介

- 性能远远优于 HTML5，几乎可以达到 Native 一致的体验
- 使用 JavaScript 开发，Learn once, write anywhere，一份代码搞定 iOS & Android 两个平台
- 无需编译成二进制可执行文件，支持动态发布

React-Navite 前景

Facebook 官方投入较大，社区趋于活跃，使用 React-Native 的公司越来越多：携程、阿里、腾讯、新美大、去哪儿等。

环境搭建

- 硬件环境
- 软件环境

硬件环境

电脑

- Mac OS（推荐）
- Windows
- Linux

移动设备

- iOS （iPhone/iPad）
- android 手机/Pad

软件环境

必须安装的软件

- Homebrew（Mac）/ Chocolatey（Windows）
- Node，npm
- react-native-cli
- Xcode（Mac）/ Android Studio（Windows）

软件环境

推荐安装的辅助工具

- Watchman
- Flow

软件环境

IDE、编辑器

- Nuclide
- Sublime
- Visual Studio Code

使用 react-native-cli 创建 HelloWorld工程

```
react-native init HelloWolrd
```

使用 react-native-cli 运行 HelloWorld 工程

```
react-native run-ios
```

or

```
react-native run-android
```

HelloWorld 代码分析

```
// index.ios.js
import React, { Component } from 'react';
import { AppRegistry, Text } from 'react-native';

class HelloWorld extends Component {
  render() {
    return (
      <Text>Hello world!</Text>
    );
  }
}

AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

HelloWorld 代码修改

把 `<Text>Hello world!</Text>` 改为 `<Text>Hello React-Native!</Text>`

在模拟器中按 `⌘-R` 就可以看到最新的修改。

React-Native 开发基础

- ES6
- JSX
- React
- flexbox 布局

ES6(ECMAScript 6/ECMAScript 2015)

ECMAScript 是 JavaScript 的标准。

ES6 相对 ES5 的重要改进：

- 新增 let 与 const 命令
- 变量的解构（Destructuring）
- 扩展运算（...）
- Class
- 箭头函数
- Promise 对象

JSX

JSX 是 JavaScript 的语法扩展，看起来有点像 XML。

JSX 是伴随着 React 而产生的。

React

React 是 Facebook 推出的一个用来构建用户界面的 JavaScript 库。

Facebook 的内部项目，因为对市场上所有 JavaScript MVC 框架都不满意，就决定自己写一套，用来架设 Instagram 的网站。

2013年5月开源。

衍生出 React Native 项目。

React 核心

- 虚拟 DOM
- 组件化
- 状态机

虚拟 DOM

组件并不是真实的 DOM 节点，而是存在于内存之中的一种数据结构，叫做虚拟 DOM （virtual DOM）。只有当它插入文档以后，才会变成真实的 DOM 。根据 React 的设计，所有的 DOM 变动，都先在虚拟 DOM 上发生，然后再将实际发生变动的部分，反映在真实 DOM上，这种算法叫做 DOM diff ，它可以极大提高网页的性能表现。

组件化

在DOM树上的节点被称为元素，在这里则不同，Virtual DOM上称为 commponent。Virtual DOM的节点就是一个完整抽象的组件，它是由 commponents 组成。

React 允许将代码封装成组件（component），然后像插入普通 HTML 标签一样，在网页中插入这个组件。

```
let HelloMessage = React.createClass({
  render: function() {
    return <h1>Hello {this.props.name}</h1>;
  }
});

ReactDOM.render(
  <HelloMessage name="John" />,
  document.getElementById('example')
);
```

状态机

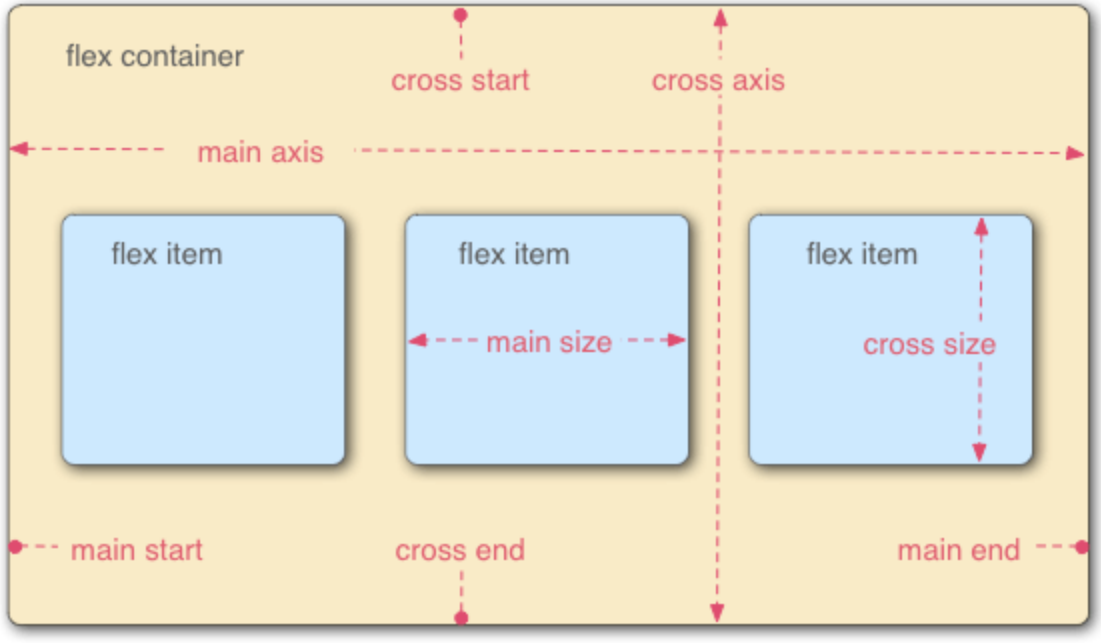
组件免不了要与用户互动，React 的一大创新，就是将组件看成是一个状态机。

除了接受输入数据（通过 this.props ），组件保持内部状态数据（通过 this.state ）。

当一个组件的状态数据的变化，展现的 UI 将被重新调用 render() 更新。

flexbox 布局

React-Native 的局是用 css 中的 flexbox 布局，与 Android 传统的布局样式有点像。



API 与 Component

Facebook 官方对 React-Native 的接口分为两类：Component 与 API。

Component 能够插入到 Vsiual DOM 中渲染 UI 的。

还有另一类接口是与 Visual DOM 无关的，只是一些简单的调用然后返回结果（可能会与 UI 相关，但是不会插入到 JS 层的 Visual DOM 中，而是直接调起了 Native UI）,这类接口 Facebook 官方把它们归类为 API。

常用API

- AppRegistry
- Platform
- Alert
- AsyncStorage
- CameraRoll

AppRegistry

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

AppRegistry.registerComponent(
                                'HelloWorld',
                                () => HelloWorld
                               );
```

Platform

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

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

if(Platform.Version === 21){
  console.log('Running on Lollipop!');
}
```

Alert

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

Alert.alert(
            'Alert Title',
            alertMessage,
          );
```

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。它用来代替LocalStorage。

AsyncStorage Persisting data:

```
try {
  await AsyncStorage.setItem('someKey', 'Some value');
} catch (error) {
  // Error saving data
}
```

AsyncStorage Fetching data:

```
try {
  const value = await AsyncStorage.getItem('someKey');
  if (value !== null){
    // We have data!!
    console.log(value);
  }
} catch (error) {
  // Error retrieving data
}
```

AsyncStorage API List

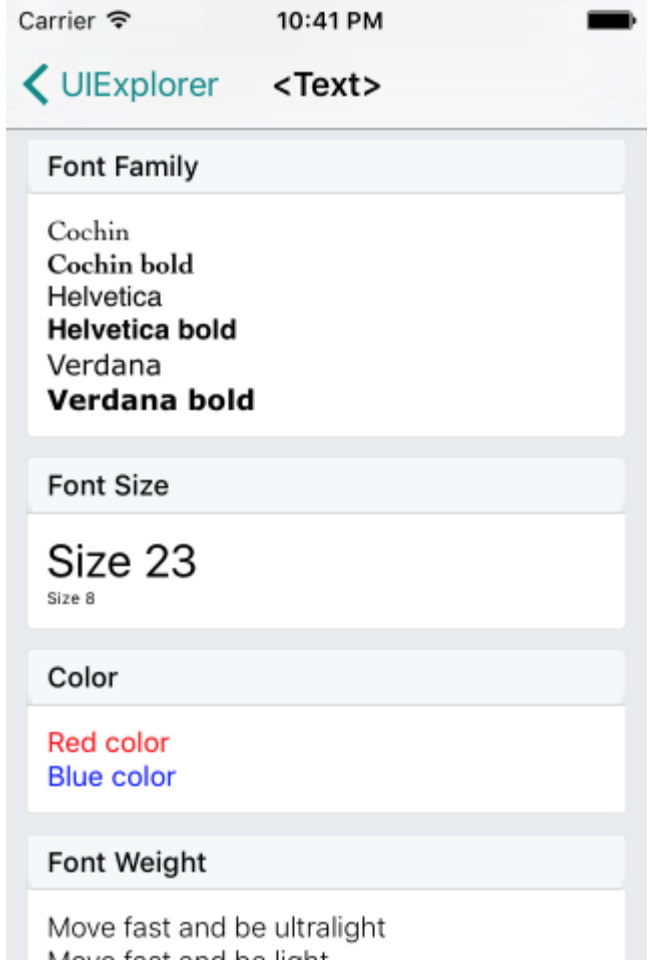
```
static getItem(key, callback?)
static setItem(key, value, callback?)
static removeItem(key, callback?)
static mergeItem(key, value, callback?)
static clear(callback?)
static getAllKeys(callback?)
static flushGetRequests()
static multiGet(keys, callback?)
static multiSet(keyValuePairs, callback?)
static multiRemove(keys, callback?)
static multiMerge(keyValuePairs, callback?)
```

常用组件

- Text
- TextInput
- Touchable 类组件
- Image
- WebView

Text

一个用于显示文本的React组件，并且它也支持嵌套、样式，以及触摸处理。

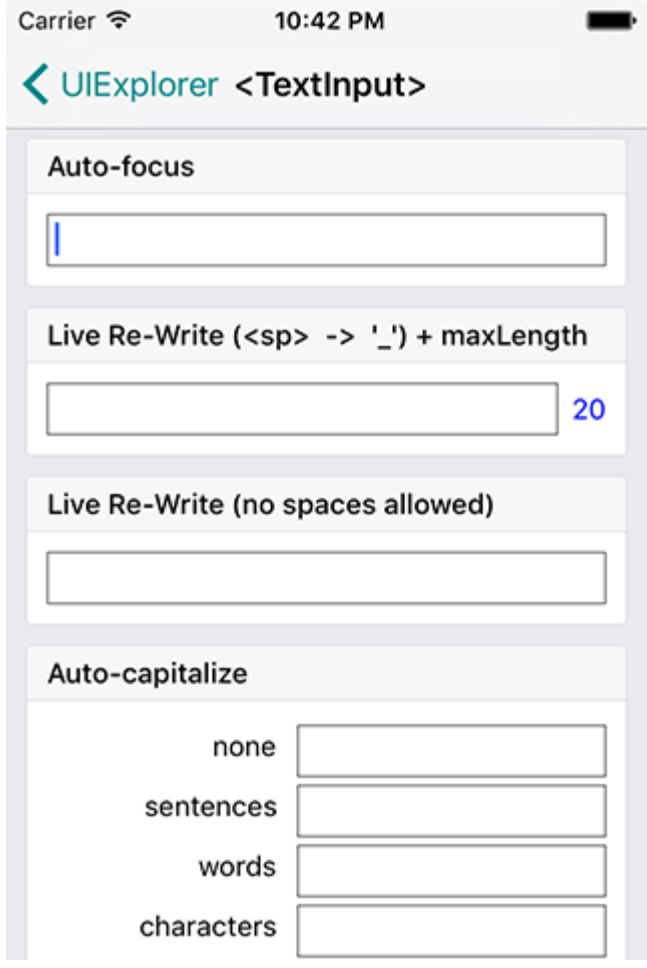


Text 示例代码

```
<Text>
        <Text onPress={() => {console.log('text pressed.');} }>
                文本组件支持触摸事件。
        </Text>
        <Text numberOfLines={2}>
                文本组件支持多行文本，也支持嵌套。
        </Text>
</Text>
```

TextInput

TextInput是一个允许用户在应用中通过键盘输入文本的基本组件。本组件的属性提供了多种特性的配置，譬如自动完成、自动大小写、占位文字，以及多种不同的键盘类型（如纯数字键盘）等等。



TextInput 示例代码

```
<TextInput
    style={{height: 40, borderColor: 'gray', borderWidth: 1}}
    onChangeText={(text) => this.setState({text})}
    value={this.state.text}
  />
```

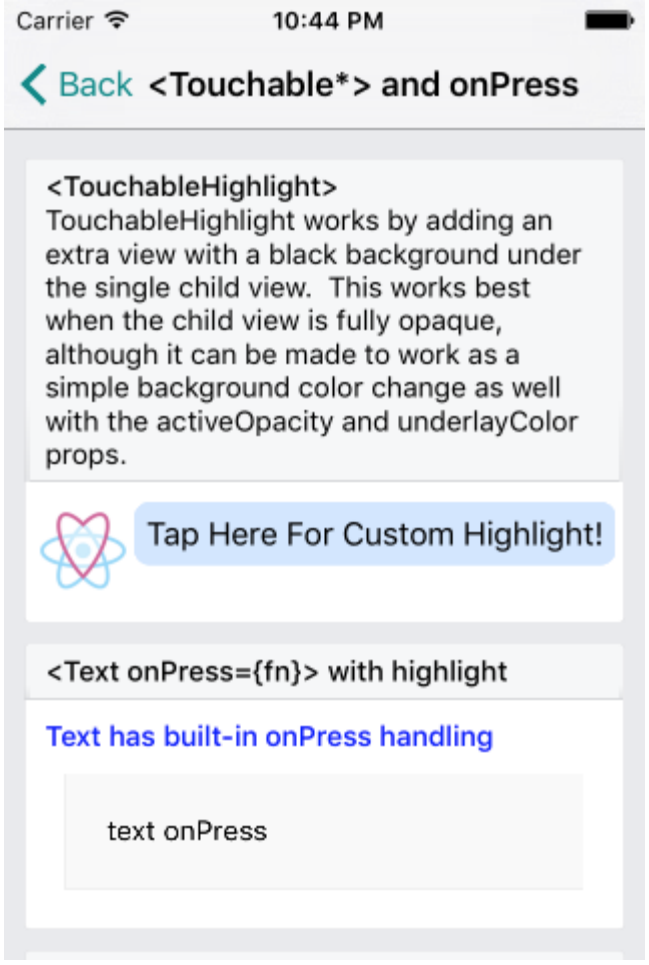
Touchable 类组件

React-Native 提供了下面三个组件与 Native 的 Button 控件类似功能：

- TouchableHighlight
- TouchableOpacity
- TouchableWithoutFeedback

TouchableHighlight 组件

当按下的时候，封装的视图的不透明度会降低，同时会有一个底层的颜色透过而被用户看到，使得视图变暗或变亮。



TouchableHighlight 组件示例代码

```
renderButton: function() {
  return (
    <TouchableHighlight onPress={this._onPressButton}>
      <Image
        style={styles.button}
        source={require('./button.png')}
      />
    </TouchableHighlight>
  );
},
```

TouchableOpacity 组件

当按下的时候，封装的视图的不透明度会降低。这个过程并不会真正改变视图层级，大部分情况下很容易添加到应用中而不会带来一些奇怪的副作用。

（此组件与TouchableHighlight的区别在于并没有额外的颜色变化，更适于一般场景）

TouchableOpacity 组件示例代码

```
renderButton: function() {
  return (
    <TouchableOpacity onPress={this._onPressButton}>
      <Image
        style={styles.button}
        source={require('image!myButton')}
      />
    </TouchableOpacity>
  );
},
```

TouchableWithoutFeedback 组件

（仅限Android平台）。在Android设备上，这个组件利用原生状态来渲染触摸的反馈。目前它只支持一个单独的View实例作为子节点。在底层实现上，实际会创建一个新的RCTView结点替换当前的子View，并附带一些额外的属性。

为提高平台通用性应避免使用这个组件。

TouchableWithoutFeedback 组件示例代码

```
renderButton: function() {
  return (
    <TouchableNativeFeedback
        onPress={this._onPressButton}
        background={TouchableNativeFeedback.SelectableBackgrour
      <View style={{width: 150, height: 100, backgroundColor: '
        <Text style={{margin: 30}}>Button</Text>
      </View>
    </TouchableNativeFeedback>
  );
},
```

Image 组件

用于显示多种不同类型图片，包括:

- 网络图片、
- 静态资源、
- 临时的本地图片、
- 以及本地磁盘上的图片（如相册）等。
- base64 编码的图片

Image 组件

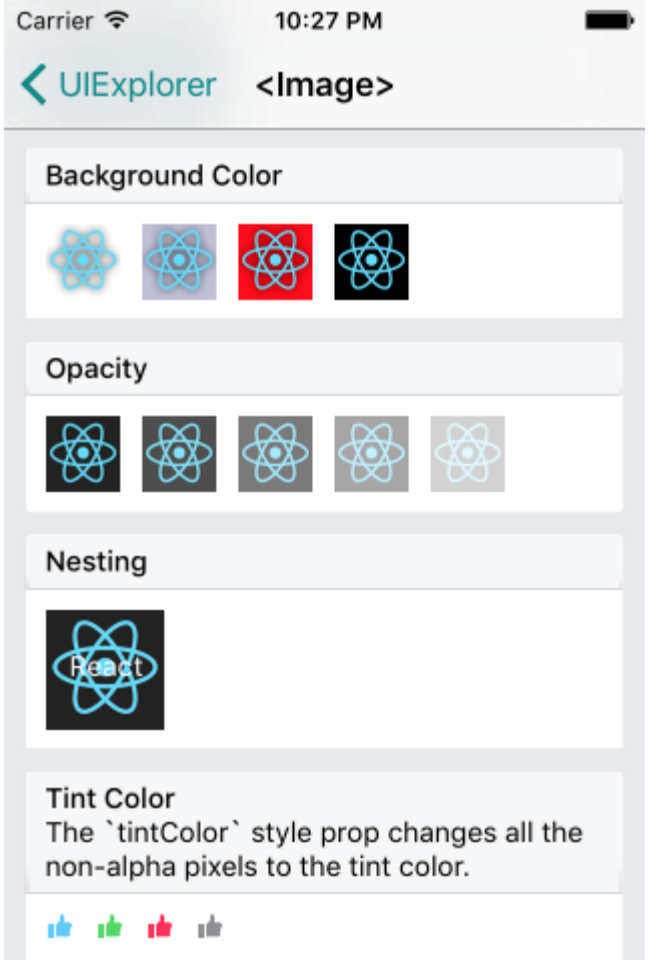
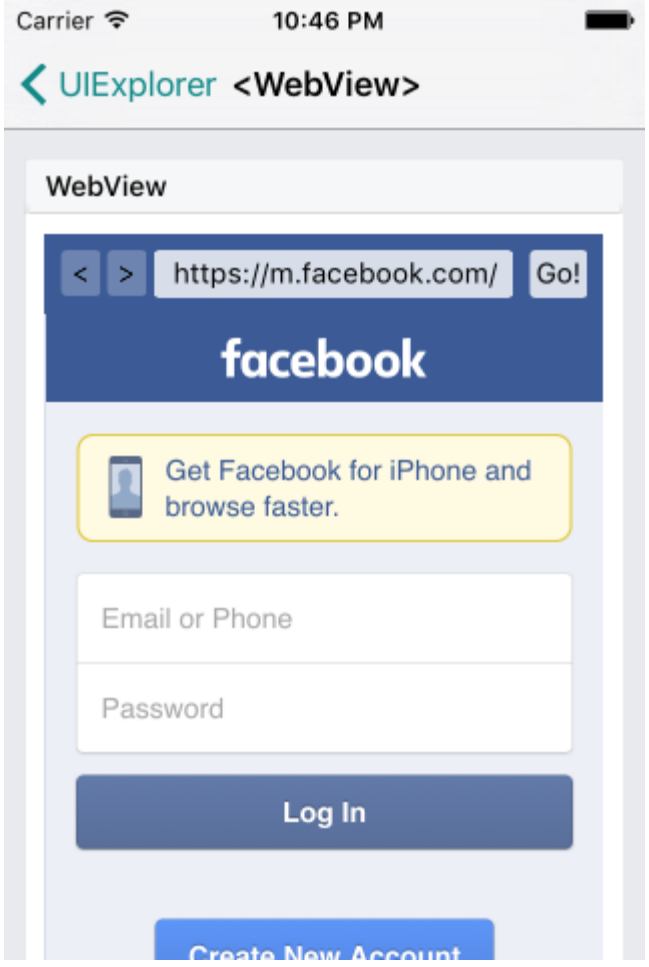

Carrier
10:27 PM
UIExplorer
<Image>
Background Color
Opacity
Nesting
React
Tint Color
The `tintColor` style prop changes all the non-alpha pixels to the tint color.

Image 组件示例代码

```
renderImages() {
  return (
    <View>
      <Image
        style={styles.icon}
        source={require('./icon.png')}
      />
      <Image
        style={styles.logo}
        source={{uri: 'http://facebook.github.io/react/img/logo
      />
    </View>
  );
}
```

WebView

WebView 可以用于访问一个网页。



WebView

```
<WebView
          ref={WEBVIEW_REF}
          automaticallyAdjustContentInsets={false}
          style={styles.webView}
          source={{uri: this.state.url}}
          javaScriptEnabled={true}
          domStorageEnabled={true}
          decelerationRate="normal"
          onNavigationStateChange={this.onNavigationStateChange
          onShouldStartLoadWithRequest={this.onShouldStartLoadW
          startInLoadingState={true}
          scalesPageToFit={this.state.scalesPageToFit}
        />
```

ScrollView 和 ListView

ScrollView 和 ListView 为视图提供了滚动的能力。

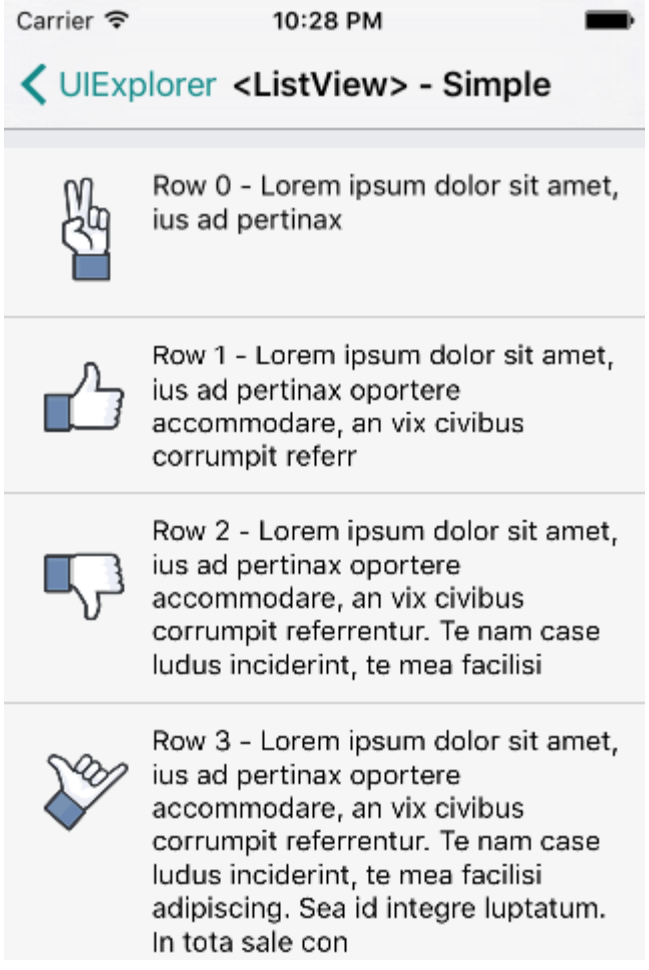
ScrollView

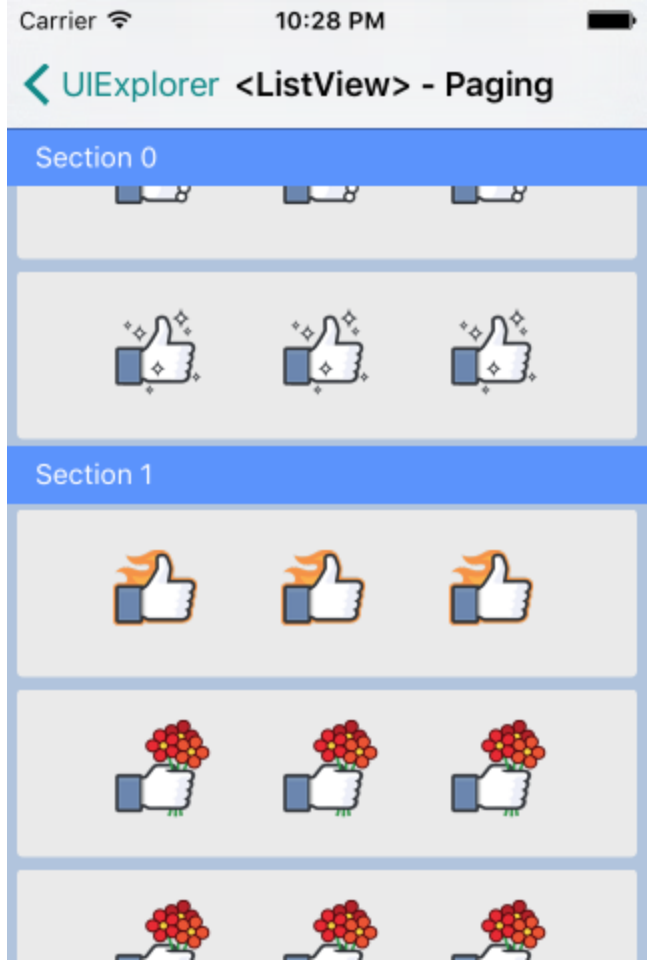
ScrollView 是 Native 暴露出来的组件。

```
return (
    <ScrollView contentContainerStyle={styles.contentContainer}
    </ScrollView>
  );
  ...
  var styles = StyleSheet.create({
    contentContainer: {
      paddingVertical: 20
    }
  });
```

ListView

ListView 用于高效地显示一个可以垂直滚动的变化的数据列表。





ListView 基本用法

1. 创建一个ListView.DataSource数据源，给它传递一个普通的数据数组
2. 使用数据源来实例化一个ListView组件
3. 定义它的renderRow回调函数，这个函数会接受数组中的每个数据作为参数，返回一个可渲染的组件（作为listview的每一行）

ListView 最简单的例子：

```
constructor(props) {
  super(props);
  var ds = new ListView.DataSource({rowHasChanged: (r1, r2) =>
  this.state = {
    dataSource: ds.cloneWithRows(['row 1', 'row 2']),
  };
}
render() {
  return (
    <ListView
      dataSource={this.state.dataSource}
      renderRow={(rowData) => <Text>{rowData}</Text>}
    />
  );
}
```

组件生命周期

组件生命周期图示


开始
getDefaultProps
getInitialState
componentWillMount
render
componentDidMount
运行中
属性(props)改变
状态(state)改变
卸载(Unmount)
componentWillReceiveProps
shouldComponentUpdate
false
true
componentWillUpdate
render
componentWillUnmount
结束

导航组件与导航栏组件

- Navigator & NavigatorIOS
- NavigationBar

Navigator

- 纯 JS 实现
- iOS 和 Android 通用

NavigatorIOS

- Native 实现
- 只能在 iOS 上使用
- 并非由 Facebook 官方开发组维护
- 默认包含一个导航栏：这个导航栏不是 React Native 视图组件，因此只能稍微修改样式。

导航组件的选择

如果纯js的方案能够满足需求，建议你选择 Navigator 组件

导航栏

- NavigatorIOS 只能轻量级配置导航栏选项
- Navigator 包含一个简单的导航栏 Navigator.NavigationBar，定制程度较高

网络

- Fetch
- XMLHttpRequest
- WebSocket

Fetch

用来发送 HTTP(S) 网络请求。

```
fetch('https://mywebsite.com/endpoint/')
```

第二个参数对象是可选的，它可以自定义HTTP请求中的一些部分。

```
fetch('https://mywebsite.com/endpoint/', {
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    firstParam: 'yourValue',
    secondParam: 'yourOtherValue',
  })
})
```

WebSocket

WebSocket 是一种基于TCP连接的全双工通讯协议。

```
let ws = new WebSocket('ws://host.com/path');

ws.onopen = () => {  // 建立连接
  ws.send('something');
};

ws.onmessage = (e) => {  // 收到了消息
  console.log(e.data);
};

ws.onerror = (e) => {  // 有错误发生
  console.log(e.message);
};

ws.onclose = (e) => {  // 连接关闭
  console.log(e.code, e.reason);
};
```

XMLHttpRequest

XMLHttpRequest 基于 iOS 网络 API 实现。需要注意与网页环境不同的地方就是安全机制：你可以访问任何网站，没有跨域的限制。

```
var request = new XMLHttpRequest();
request.onreadystatechange = (e) => {
  if (request.readyState !== 4) {
    return;
  }

  if (request.status === 200) {
    console.log('success', request.responseText);
  } else {
    console.warn('error');
  }
};

request.open('GET', 'https://mywebsite.com/endpoint.php');
request.send();
```

XMLHttpRequest

通常不应该直接使用 XMLHttpRequest，因为它的API用起来非常冗长。
不过这一 API 的实现完全兼容浏览器，因而你可以使用很多 npm 上已有的第三方库，例如 frisbee或是 axios。

(不过官方还是推荐你使用fetch)

React-Native 通讯机制及混合开发基础

- React-Native 通讯机制
- 模块配置映射表
- Native 调用 JS
- JS 调用 Native
- Native 模块暴露
- Native UI 组件暴露
- 混合开发中的多线程

React-Native 通讯机制

- Native 调 JS :
  有现成的接口，类似 webview 提供的 - stringByEvaluatingJavaScriptFromString 方法可以直接在当前 context 上执行一段JS脚本；
  并且可以获取执行后的返回值，这个返回值就相当于 JS 向 Native 传递信息。

- JS 调 Native :
  JS 并不能主动调到 Native 层，所以 JS 对 Native 的调用是先存到一个 Queue 里面，Native 调 JS 时才会把 Queue 里的方法返回给 Native 层去执行。

React Native也是以此为基础，通过各种手段，实现了在 Native 定义一个模块方法，JS可以直接调用这个模块方法并还可以无缝衔接回调。

模块配置映射表

Native 端和 JS 端各有一个 bridge 保存了同样一份模块配置表，配置表里包括了所有模块和模块里方法的信息，JS 调用 Native 模块方法时，通过 bridge 里的配置表把模块方法转为模块 ID 和方法 ID 传给 Native，Native 通过 bridge 的模块配置表找到对应的方法执行之:。

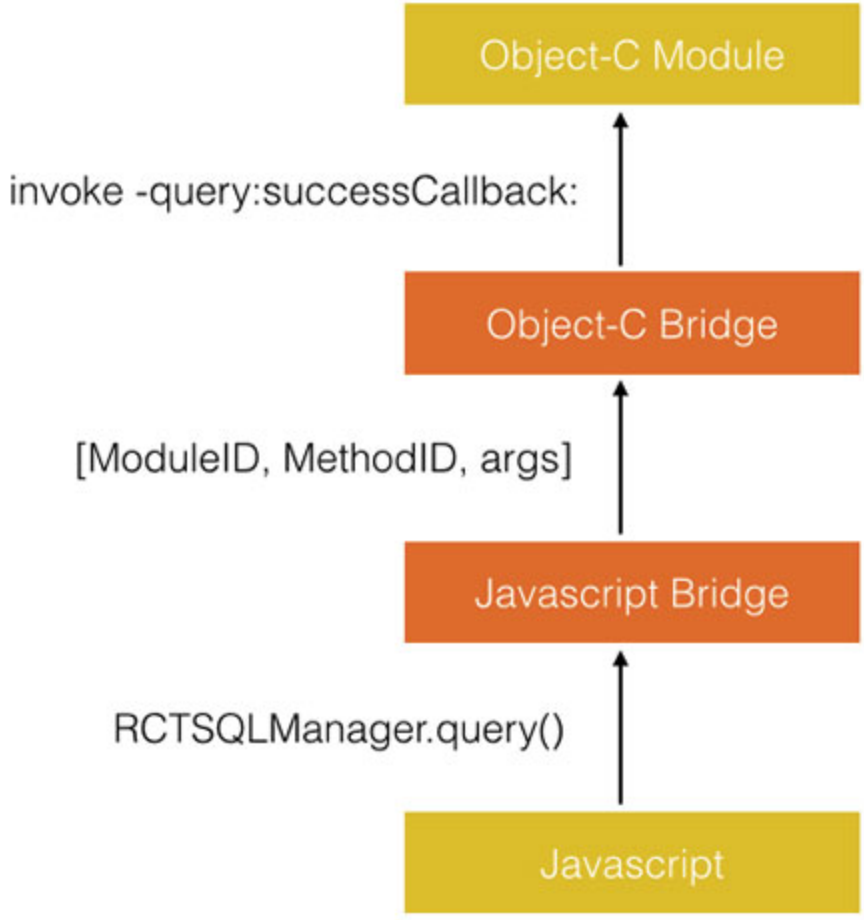
```
{
    "remoteModuleConfig": {
        "RCTSQLManager": {
            "methods": {
                "query": {
                    "type": "remote",
                    "methodID": 0
                }
            },
            "moduleID": 4
        },
        ...
     },
}
```

Native 调用 JS

1. Native Bridge 查找模块配置表。
2. 在对应 JS Context 上执行JS。
3. JS Bridge 查找 模块配置表，找到对应的方法并执行之。

JS 调用 Native（先不考虑回调）

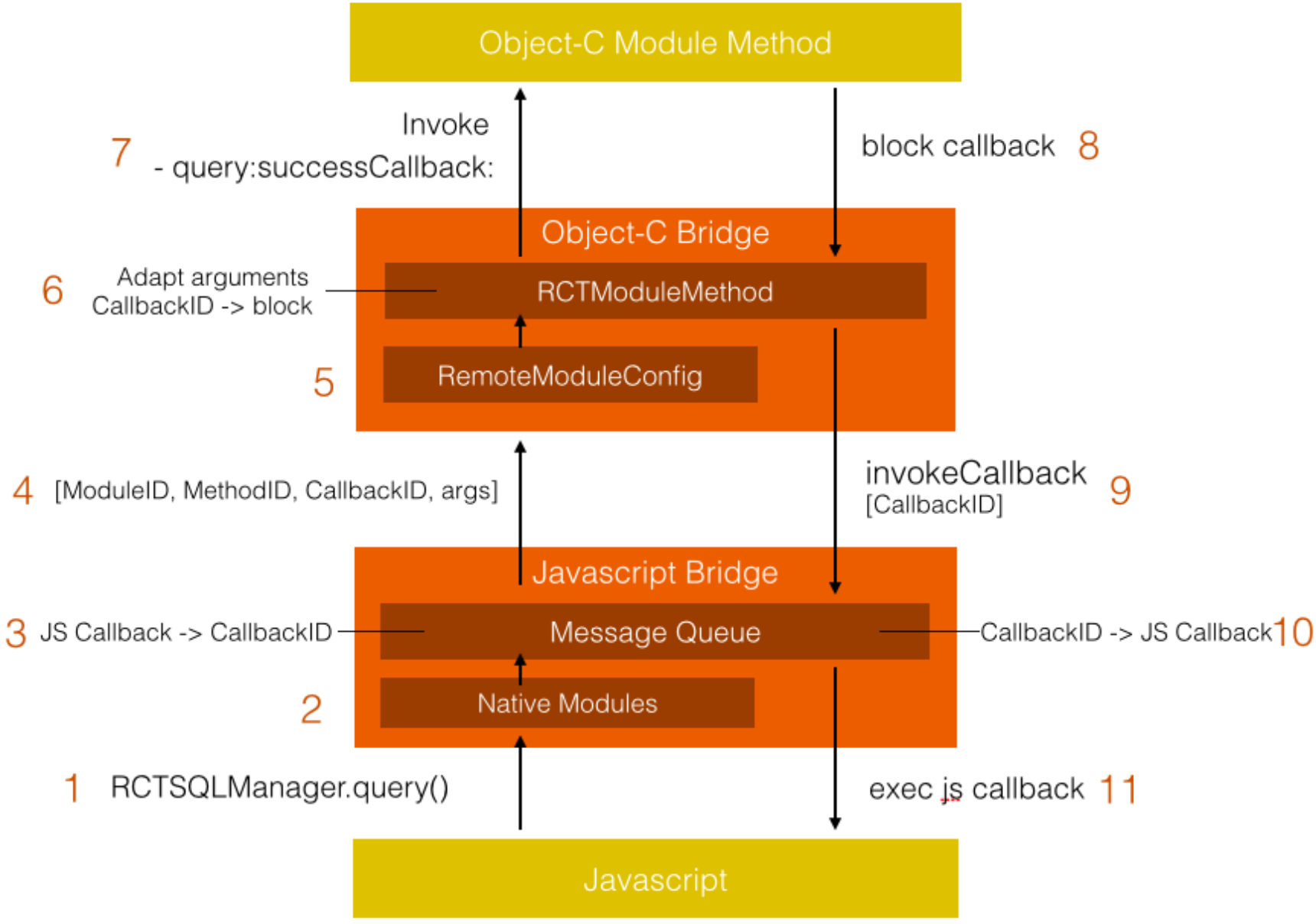
图示



JS 调用 Native 方法（带回调）

图示

JS 调用 Native 方法（带回调）



Native 模块暴露

在React Native中，一个“原生模块”就是一个实现了“RCTBridgeModule”协议的Objective-C类。

```
// CalendarManager.h
#import "RCTBridgeModule.h"

@interface CalendarManager : NSObject <RCTBridgeModule>
@end

// CalendarManager.m
@implementation CalendarManager

RCT_EXPORT_MODULE();

RCT_EXPORT_METHOD(addEvent:(NSString *)name location:(NSString
{
  RCTLogInfo(@"Pretending to create an event %@ at %@", name, l
}
@end
```

Native 模块暴露

JS 中调用：

```
let CalendarManager =  require('react-native').NativeModules.Ca
CalendarManager.addEvent('Birthday Party', '4 Privet Drive, Sur
```

Native 导出常量

原生模块可以导出一些常量，这些常量在JavaScript端随时都可以访问。用这种方法来传递一些静态数据，可以避免通过bridge进行一次来回交互。

```
- (NSDictionary *)constantsToExport
{
  return @{ @"firstDayOfTheWeek": @"Monday" };
}
```

Javascript端可以随时同步地访问这个数据：

```
console.log(CalendarManager.firstDayOfTheWeek);
```

Native 组件暴露

- 创建一个 `RCTViewManager` 子类
- 添加RCT_EXPORT_MODULE()标记宏
- 实现 `-(UIView *)view` 方法

Native 组件暴露 - Native 代码示例

```
// RCTMapManager.m
#import <MapKit/MapKit.h>

#import "RCTViewManager.h"

@interface RCTMapManager : RCTViewManager
@end

@implementation RCTMapManager

RCT_EXPORT_MODULE()

- (UIView *)view
{
  return [[MKMapView alloc] init];
}

@end
```

Native 组件暴露 - JS 调用示例

```
// MapView.js

var { requireNativeComponent } = require('react-native');

// requireNativeComponent 自动把这个组件提供给 "RCTMapManager"
module.exports = requireNativeComponent('RCTMap', null);
```

混合开发中的多线程

- 原生模块不应对自己被调用时所处的线程做任何假设。如果调用了 Native UI 渲染相关任务应该手动调度到主线程执行。
- 如果一个操作需要耗时较长（如磁盘IO）则不应该阻塞 React 本身的消息队列，而是应该手动调度到一个专门的线程去处理。

Swift 支持

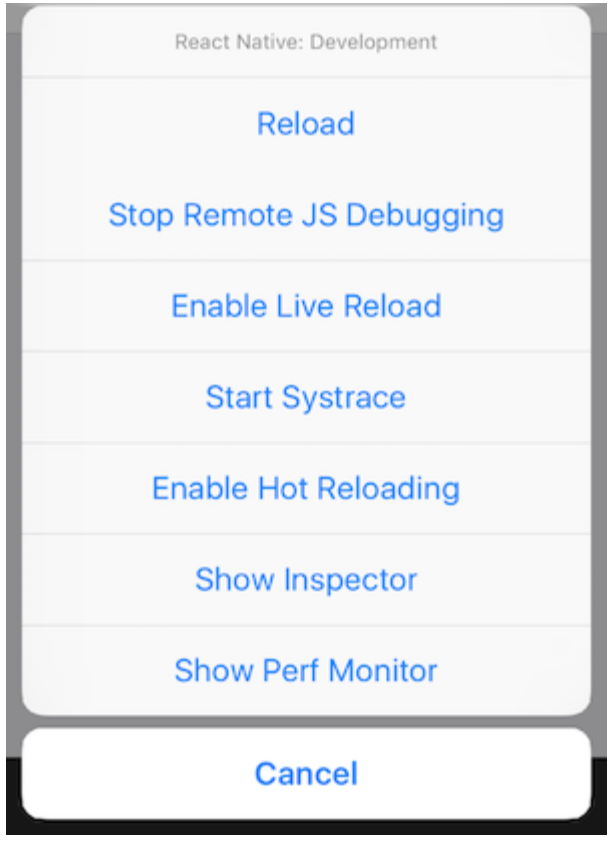
React-Native 本身不支持 Swift，但是可以通过 Swift-OC 混合编程来实现 React-Native 支持 Swift，但是这种方式成本较高，不太推荐。

性能实践与调优

- 工具
- 常见性能问题及优化方案

工具

React-Native SDK 内置了性能相关工具，开发者模式下摇一摇手机即可打开。



性能问题常见原因

- 开发模式 (dev=true)
- 导航器(Navigator)切换卡顿
- ListView初始化渲染太慢以及列表过长时滚动性能太差

导航器(Navigator)切换卡顿

Navigator 的动画是由 JavaScript 线程所控制的。

解决方案：动画完成后再渲染复杂 UI

```
InteractionManager.runAfterInteractions
```

ListView初始化渲染太慢以及列表过长时滚动性能太差

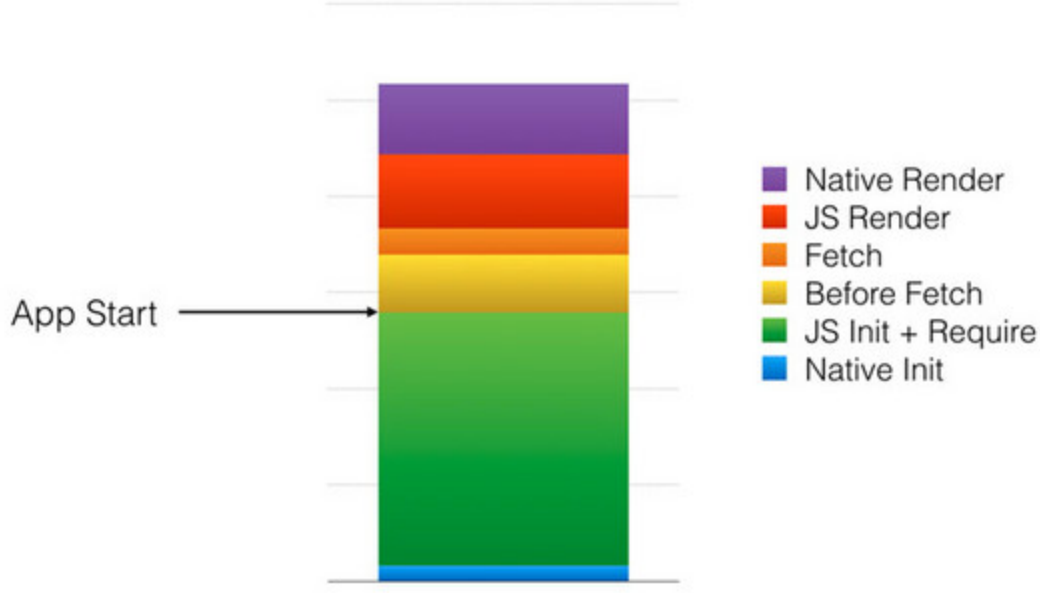
ListView 并不是基于原生的 UITableView 或 ListView 封装的，而是基于 ScrollView 封装的，
所以 Cell 复用是在 JS 层实现的。

优化方案：

- 设置首次渲染行数 initialListSize
- 设置每一帧渲染行数 pageSize
- 设置超过视野范围的渲染行数 scrollRenderAheadDistance
- 设置 `overflow:hidden`

RN性能分析

- iOS:加载0.5s
- Android：加载0.8-1s
  时间占用分析如下



RN加载优化

- 预加载common包优化
- Android JSC与view分离
- iOS &Android Native Require优化
- 打包拆包优化

Thanks

Q & A