

Begin:

对JS Closure（闭包）的认识：

<https://stackoverflow.com/questions/111102/how-do-javascript-closures-work>

搞懂第一个作者列举的7个例子后，对闭包的理解就差不多了。

End

Begin:

两大事件head & body，两者内部都可以放置script,可以直接将js代码放在script中；也可以独立成.js文件，然后在<script src=””></script>中引用。引用路径可以是与当前页面相同文件夹，或者指定相对路径，或者指定完整路径皆可。

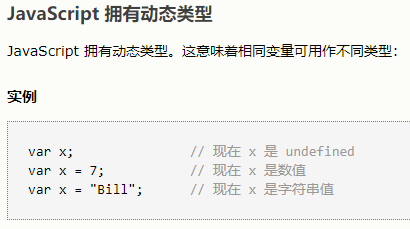
4种通过JS输出方法：

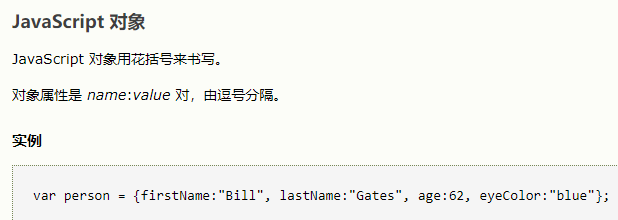


Js注释分单行和多行注释 // /\* \*/

Js只声明变量不赋值，那么默认值就是**unDefined**

Js数据类型：





Js函数：

Function funcName(){ \*\*\*\*\*\* }；

调用时funcName表示引用此函数对象，而带上()运算符后表示引用此函数执行后的结果

关于方法块中使用this的含义：





<button onClick = “……js code”/>

onClick表示事件属性，用可执行的js代码为属性赋值。当产生该事件时。

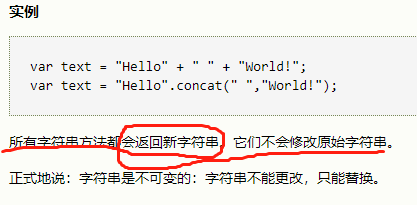
<*element* *event*='*一些 JavaScript*'>

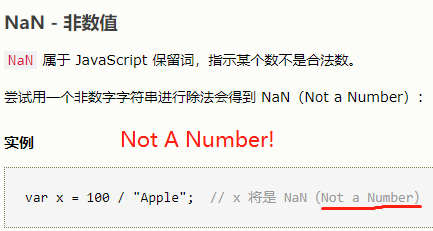
这样方便理解 与记忆！

JavaScript 对象无法进行对比，比较两个 JavaScript对象将始终返回 false。

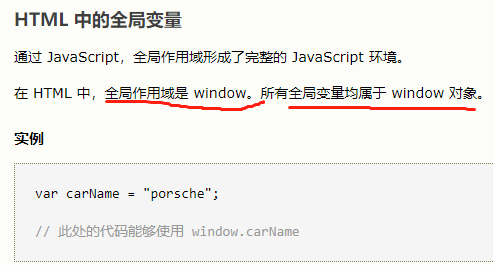
字符串搜索函数有两类：indexOf() and search().

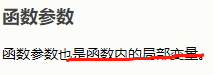
截取部分：slice() substring() subStr();





Break 跳出switch & for





JS Hoisting特性，即将**变量/方法的声明**提升至作用域最开始。代码中并没有变，产生这种变化其实是在编译阶段放到内存中时。Let Const修饰的变量和常量不会被提升。并且提升是只提升变量/方法的声明，而非初始化。



搞懂JS 中this所指的实际对象是谁。

* Let: 对比下for循环中的作用域：

for (let i = 0; i < 10; i++) {

// 一些语句

}

上面变量i的作用域仅为**for循环内**。**Block作用域**。

下面变量i的作用域为**全局**。

for (var i = 0; i < 10; i++) {

// 一些语句

}

**结论**：//足以证明for()括号内用var声明的变量作用域依旧是

//全局作用域。

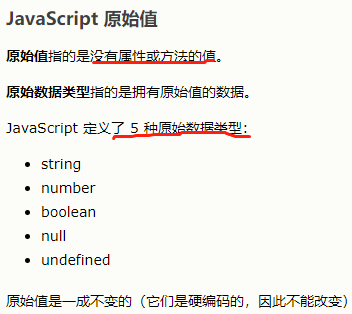
如果用let声明，那其作用域就只是for局部作用域。**外部甚至不可见**。RefrenceUnDefinedError.

Const表示常量，基本类似于java中的final特性。

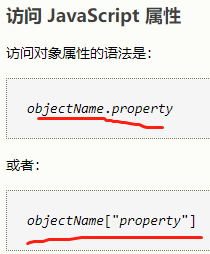
**Const let 都是声明局部作用域变量的！**

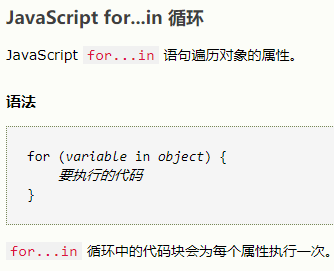
**Var定义的变量会被提升，但Const & let修饰的变量不会被提升。**

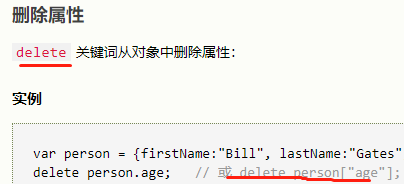


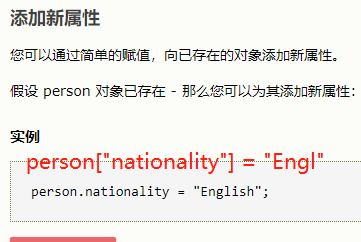


**JS是面向对象的语言。对象为王**！传递的是**对象引用**









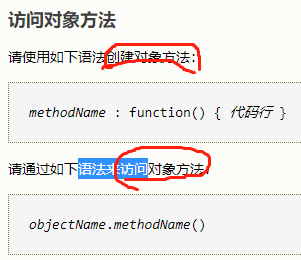
delete 关键词会同时删除属性的值和属性本身

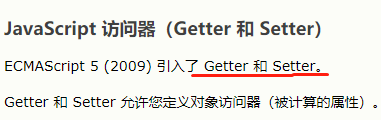
delete 操作符被**设计用于对象属性**。它对变量或函数没有影响。

JS对象也是有继承概念的

JavaScript 对象继承了它们的**原型的属性**。

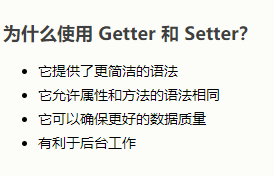
delete 关键词不会删除被继承的属性，但是如果您删除了某个原型属性，则将影响到所有从原型继承的对象。

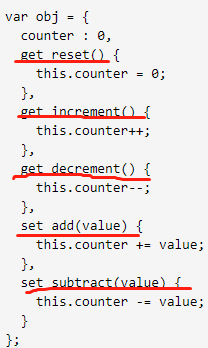




get getWhat(){ return sth}

可以针对不同的属性添加多个get/set方法。非常类似于java.





在 JavaScript 中，被称为 this 的事物是代码的“拥有者”。

请注意 this 并不是变量。它是关键词。您无法改变 this 的值。

JavaScript 提供原始数据类型字符串、数字和布尔的对象版本。但是并无理由创建复杂的对象。原始值快得多！



JS对象原型：

**所有 JavaScript 对象都从原型继承属性和方法**

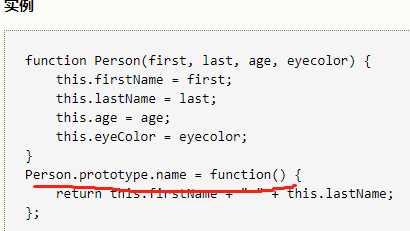
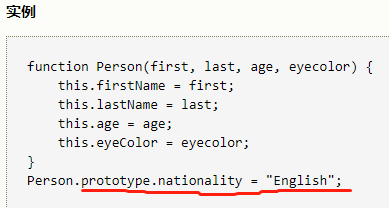
日期对象继承自 Date.prototype。数组对象继承自 Array.prototype。Person 对象继承自 Person.prototype。Object.prototype 位于原型继承链的顶端：

日期对象、数组对象和 Person 对象都继承自 Object.prototype。

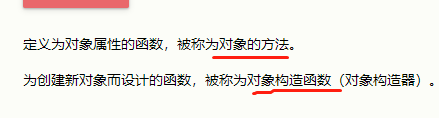
**使用 prototype 属性**

JavaScript prototype 属性允许您为对象构造器添加新属性：

JavaScript prototype 属性也允许您为对象构造器添加新方法：



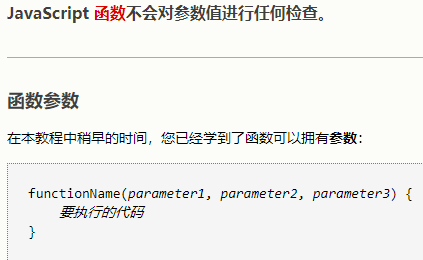
请只修改**您自己**定义的对象原型。绝不要修改标准 JavaScript 对象的原型



JavaScript ES5 对象方法 后面再看这节

**JavaScript 函数是通过 function 关键词定义的。**

**您可以使用函数声明或函数表达式。**



参数如果是对象，则传递进去的是该对象的引用地址；同java

参数如果是值，则传递进去的就是值；同Java

函数的调用与构造函数的调用（new ConstructorFunc()：

\*在 JavaScript 中，任何函数都是对象的方法。

\*如果一个函数不是 JavaScript 对象的方法，那么它就是**全局对象**的函数

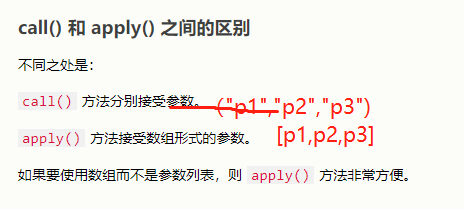
## 方法重用

使用 call() 方法，您可以编写能够在不同对象上使用的方法。



## 方法重用

通过 apply() 方法，您能够编写用于不同对象的方法。



拥有相同名称的全局变量和局部变量是不同的变量。修改一个，不会改变其他。

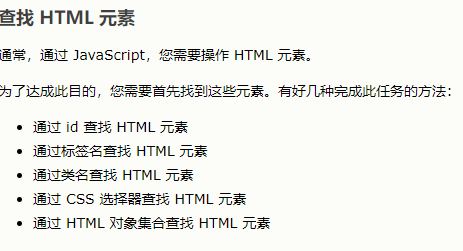
在网页中，全局变量属于 window 对象。全局变量活得和您的应用程序（窗口、网页）一样久。\

局部变量活得不长。它们在函数调用时创建，在函数完成后被删除

何为Html DOM?

当网页被加载时，浏览器会创建**页面的文档对象模型**（**D**ocument **O**bject **M**odel）。

在 DOM 中，所有 HTML 元素都被定义为**对象**。



改变html元素内容/属性值

Document.write(new content);

修改 HTML 文档内容最简单的方法是，使用 innerHTML 属性。

document.getElementById(id).innerHTML = new text

Document.getElementById(id).attribute = new value;

通过元素对象的style对象的属性来改变相应的css属性值。

document.getElementById(id).style.property = new style

Dom 动画 本节暂且略过

HTML DOM事件：

HTML DOM 允许 JavaScript 对 HTML 事件作出反应

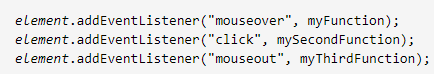
HTML DOM 事件监听器

addEventListener() 方法为元素附加事件处理程序而不会覆盖已有的事件处理程序

您能够向任何 DOM 对象添加事件处理程序而非仅仅 HTML 元素，例如 window 对象。

element.addEventListener(event, function, useCapture);

**注意：**请勿对事件使用 "on" 前缀；请使用 "click" 代替 "onclick"。





## 事件冒泡还是事件捕获？

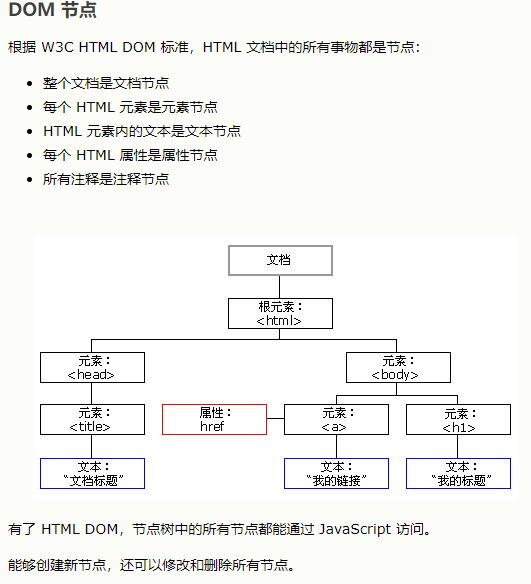
在 HTML DOM 中有两种事件传播的方法：**冒泡和捕获**。

事件传播是一种定义当发生事件时元素次序的方法。假如 <div> 元素内有一个 <p>，然后用户点击了这个 <p> 元素，应该首先处理哪个元素“click”事件？

在冒泡中，**最内侧元素的事件会首先被处理**，然后是更外侧的：首先处理 <p> 元素的点击事件，然后是 <div> 元素的点击事件。

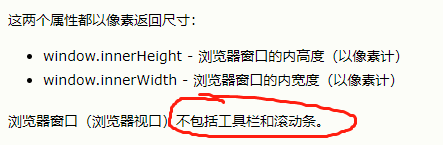
在捕获中，**最外侧元素的事件会首先被处理**，然后是更内侧的：首先处理 <div> 元素的点击事件，然后是 <p> 元素的点击事件

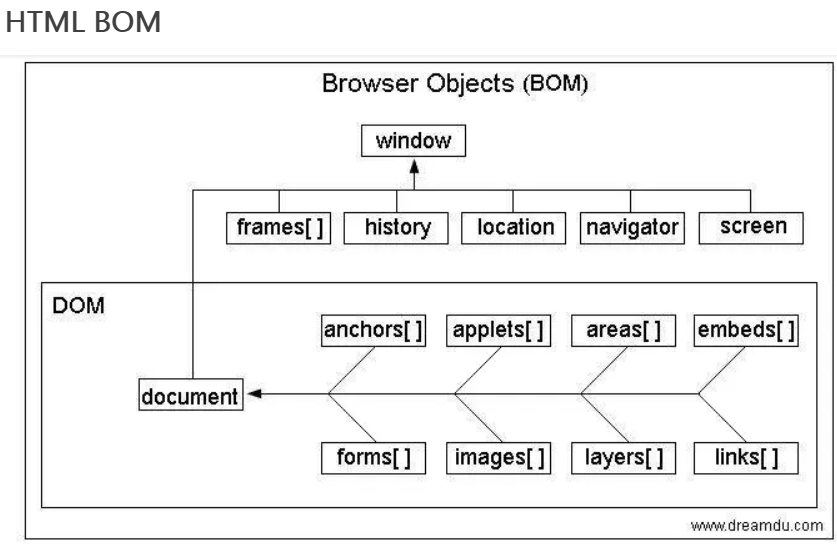




BOM和DOM

222222





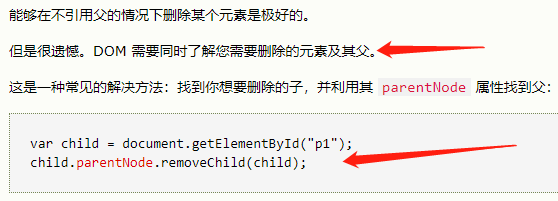




**JavaScript 有三种类型的弹出框：警告框、确认框和提示框。**

# JavaScript Timing 事件





DOM节点：可以添加，删除，替换任意节点。

一切皆为节点，元素<p>，属性<style>,文本<text>

## 什么是 AJAX？

AJAX = **A**synchronous **J**avaScript **A**nd **X**ML.

AJAX 并非编程语言。

AJAX 仅仅组合了：

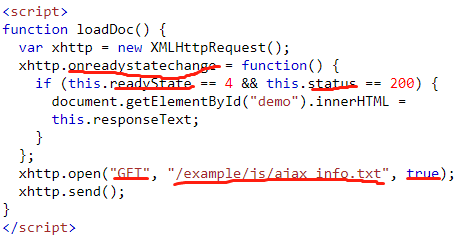
* 浏览器内建的 XMLHttpRequest 对象（从 web 服务器请求数据）
* JavaScript 和 HTML DOM（显示或使用数据）

AJAX XMLHttpRequest对象

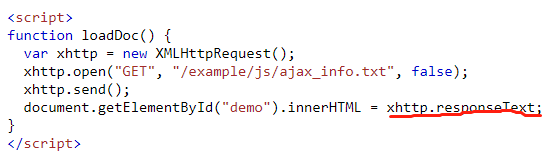
创建：variable = new XMLHttpRequest();



完整示例：

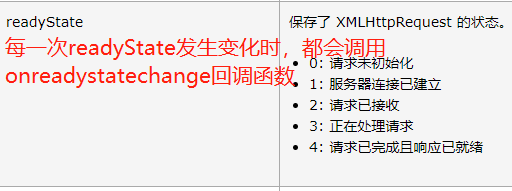


同步实例：



总是使用异步请求：

不推荐同步的 XMLHttpRequest (async = false)，因为 JavaScript 将停止执行直到服务器响应就绪。如果服务器繁忙或缓慢，应用程序将挂起或停止。

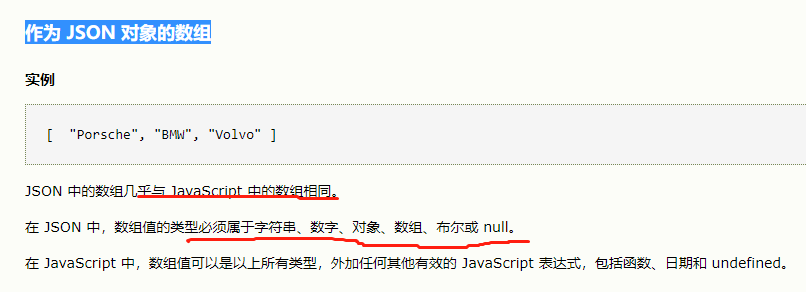




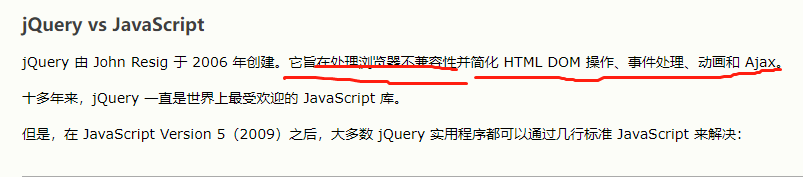
返回的响应有responseText和responseXML两种类型，而且XML HttpRequest 对象有一个內建的 XML 解析器,

ResponseXML 属性以 XML DOM 对象返回服务器响应。

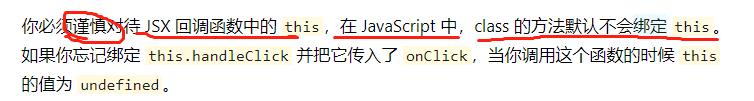




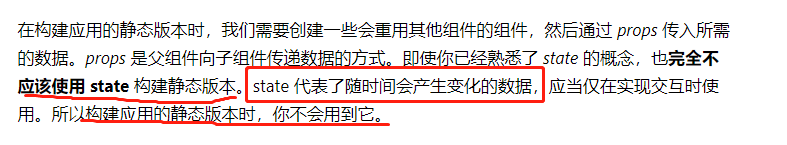
JQuery存在的优势：



ES6 class语法中，



React哲学：《非常重要的概念》



React中有两类模型数据：props和state. 清楚地理解两者的区别至关重要。

const element = <Welcome name="Sara" />;

当 React 元素为用户自定义组件时，它会将 JSX 所接收的属性（attributes）转换为单个对象传递给组件，这个对象被称之为 “props”。

**React 状态提升：**

通常，**多个组件需要反映相同的变化数据**，这时我们建议将共享状态提升到最近的共同父组件中去。让我们看看它是如何运作的。

\*每个组件中的state只有该组件能访问到，对其它任何组件（父组件/子组件）都是透明的。

如果两个同级组件需要使用共享的state来保存/读取信息，此时就用到了**“状态提升”**!

在 React 中，将多个组件中需要共享的 state 向上移动到它们的最近共同父组件中，便可实现共享 state。这就是所谓的“状态提升”。接下来，我们将 TemperatureInput 组件中的 state 移动至 Calculator 组件中去。

如何实现state属性向下传递，以及子组件向上更新父组件中的state,从而重新渲染自身呢？**通过props对象向下传递**（当然包括数据加工，但这已经不属于state存在的初衷）；通过**回调props传递过来函数进行回调**，从而改变state进行重新渲染。

JS对象中的key都是没有隐号的，而json中的key全部是隐号，而value有没有隐号，由其自身的数据类型决定的。数字没有，字符串有。

关于引号：

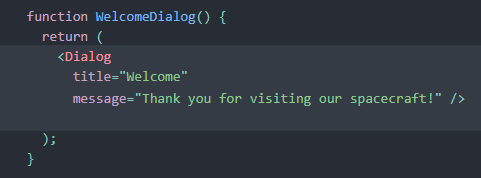
JSX value should be either an expression or a quoted JSX text (74:38)

对于JSX,属性值总是 {} 内加表达式，或者是双引号文本。单引号其实也可以的。不过写的不舒服，统一使用双引号吧.

**React组合与继承的对比：**

“特殊”组件可以通过 props 定制并渲染“一般”组件。具体怎么说呢？

WelcomeDialog可以说是Dialog的特殊实例，在java中我们可以继承并扩展Dialog，但是在react中，这么办：利用props向一般实例中传入特殊值

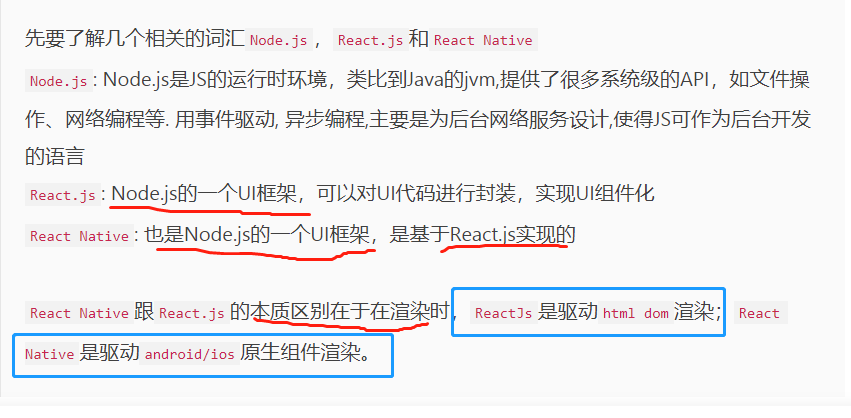


**在 React 中，我们也可以通过组合来实现这一点。**

**“特殊”组件可以通过 props 定制并渲染“一般”组件：**

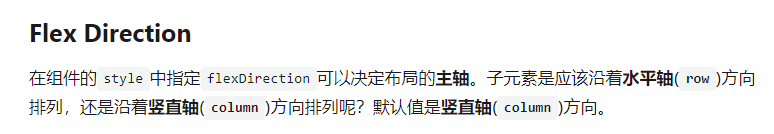


**组件：类似于js函数，接收任意入参props,返回元素！**



很多传统的应用框架会设计自有的模板语法，让你在结构标记中嵌入代码。React 反其道而行之，设计的 JSX 语法却是让你**在代码中嵌入结构标记**

**布局：**



**JustifyContent:在父容器中指定子元素沿主轴的排列方式**

**AlignItms:在父容器中指定子元素沿次轴的排列方式**

**AlignSelf:给子元素指定其在父容器中，沿次轴的排列方式**

**现成组件<Button/>的样式是不可定制，往往不符UI设计的，所以又给了四个自定义Button的组件，**

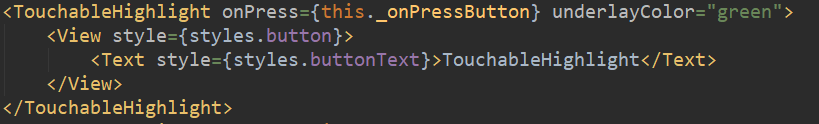
**<TouchableHighlight/>**

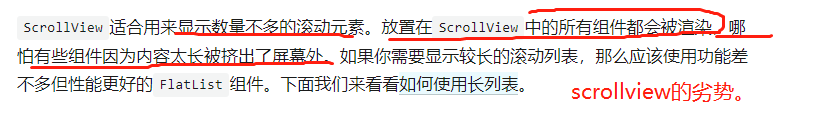
**<TouchableOpacity/>**

**<TouchableNativeFeedback/>**

**<TouchableWithoutFeedback/>**

**用法基本上都是一个模式：**

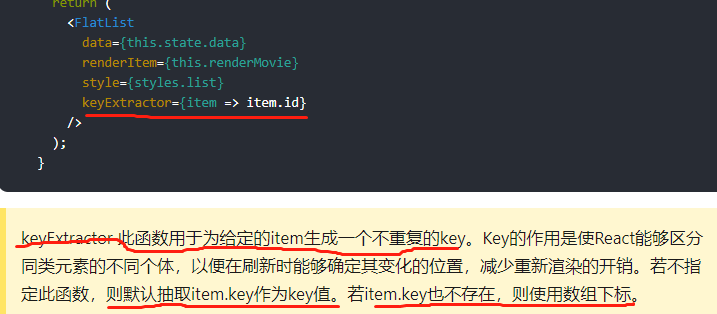




**因此引入下面的长列表以显示更多内容，而且性能更好：**



**FlatList需要两个prop: data, renderItem.**

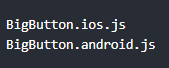


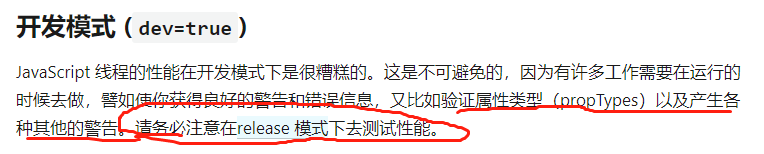
**箭头函数是ES6中新语法，简单：() => { };**

**React Native 提供了两种方法来区分平台：**

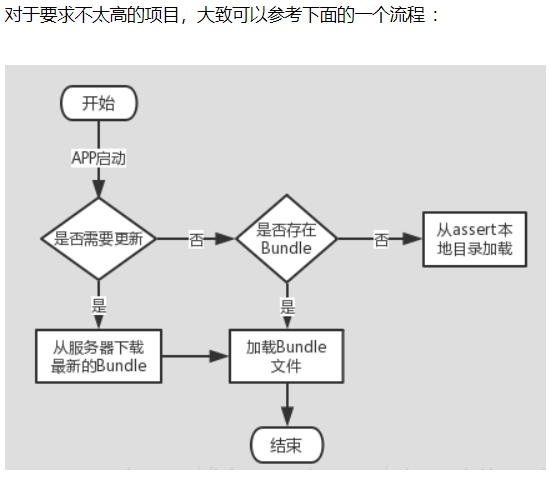
**使用Platform模块.**

**使用特定平台扩展名.**





**RN的热更新大概原理：**



说白了集成RN业务，就是集成RN离线包，解析并渲染。所以，RN热更新的根本原理就是更换js bundle文件和资源文件，并重新加载，新的内容就完美的展示出来了。

**RN 组件列表：**

**<View/>组件，容器作用，支持flexbox,borderradius,borderwith等各种方便的属性，相当于安卓上的android.View,web上的<div/>容器。**

**<Image/>组件**

**加载jsbundle中的静态资源，使用<Image source={require('./img/ori.jpg')}/>不需要大小**

**加载原生drawable下的图片时，只写图片名，不写扩展名，并且要指定大小：**

**<Image source={{uri:'ic\_launcher'}} style={{width: 80,height: 80,backgroundColor: 'transparent'}}/>**

**加载网络图片，也必须先指定容器大小：**

<Imagesource={{uri:'https://upload.wikimedia.org/wikipedia/commons/d/de/Bananavarieties.jpg'}}style={{width:80,height:80}}/>

**Image嵌套的Text的方式：实现背景底图**

**<Image ><Text></Text></Image>**

**<Image/>有很多属性可以使用呢**

**<Text/>组件:**

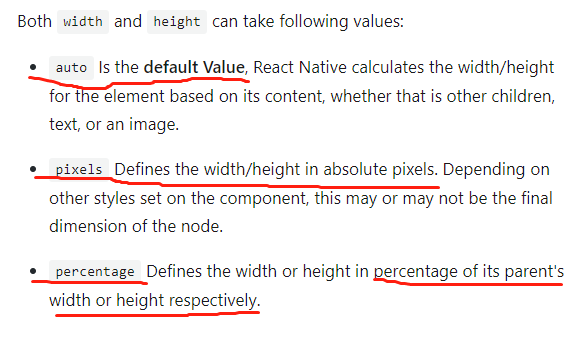
*嵌套特点:和Web上面一直的设计方案，我们通过嵌套包裹的方案，****相同的属性的文本可以用父标签进行包裹****，然后****内部特殊的文本采用子标签方案****，具体例子如下:*

Text嵌套时，注意子组件要不要继承/覆盖父组件的属性，默认是继承的。

**容器布局规则不同于<view/>容器，不支持flexbox弹性布局，默认会一个组件接前一个组件后面按序排列。**

**关于宽高：**

**宽高不设的话，默认取值是wrap\_content**



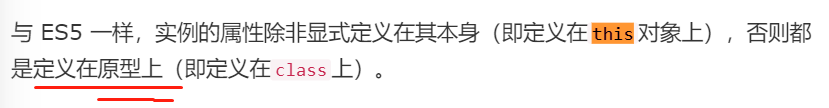
**flex** will define how your items are going to “fill” over the available space **along your main axis**. Space will be divided according to each element's flex property.

Flex定义了**沿着父容器的主轴**，子元素如何排布剩余空间，要搞清是**沿着主轴方向**分割剩余空间！

ScrollView组件：

ES6中class新特性：

注意，定义“类”的方法的时候，前面**不需要加上function这个关键字**，直接把函数定义放进去了就可以了。另外，方法之间不需要逗号分隔，加了会报错。而且，如果在类的方法中使用到了this关键，那么在构造器中要**显式进行绑定**：**this.method=this.method.bind(this)，目的是：生成一个绑定了当前对象为默认this的新方法。**



对于一个已定义好的function,可以用apply(),bind(),or,call()来改变方法中this的指向，但箭头函数不适用，因为箭头函数没有this**变量**属性（？）。

另外，由于箭头函数没有自己的this，所以当然也就不能用call()、apply()、bind()这些方法去改变this的指向。

根本原因是什么：

this指向的固定化，并不是因为箭头函数内部有绑定this的机制，实际原因是箭头函数根本没有自己的this，导致内部的this**就是外层代码块的**this。正是因为它没有this，所以也就不能用作构造函数。

**\***有一个问题，既然箭头函数的this,其实就是父代码块的this,那么问题是普通函数中this是什么？是如何确定的呢？

因为所有的内层函数都是箭头函数，都没有自己的this，它们的this其实都是最外层foo函数的this.

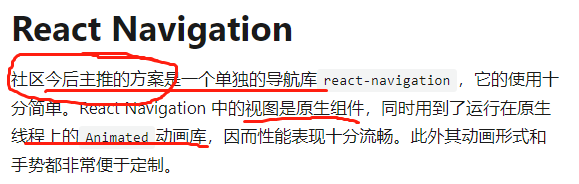
CSS和布局：

Flexbox works the same way in React Native as it does in CSS on the web, with a few exceptions. The defaults are different, with flexDirection defaulting to column instead of row, and the flex parameter only supporting a single number.

Flexbox工作原理和CSS中一样的。只是有一点不同点，默认主轴是y轴，交叉轴是x轴。默认布局方向是沿y轴布置的。

对于alignItems属性：默认值是stretch,即沿交叉轴自动伸长，但是子元素如果指定了具体的宽度/高度值，则此属性不生效了。同web css.

alignContent与flexWrap是搭配使用的。flexWrap指定主轴排列满时，是否多行显示其它子元素，如果多行显示子元素，alignContent生效，并且**alignConetnt指定的对齐方式是指多条轴线在与其垂直的方向上（交叉轴）方向上的排列方式。属性值同justifyContent.**



导航：Moving between Screens(重要的一节)

1. **Creating a stack navigator，所有的页面路由全部添加到这个里面**

const AppNavigator = **createStackNavigator**(

{

Home: HomeScreen,

Details: DetailsScreen,

},

{

initialRouteName: 'Home',

}

);

1. 方法原型：createStackNavigator(routeConfigMap，stackConfig);
2. 每一个注册的组件，其props都会接收到一个navigation对象，用来控制当前页面往哪跳转的路径。

TabNavigator:

Api

import { createBottomTabNavigator } from 'react-navigation-tabs';

**const tabNav =**

**createBottomTabNavigator**(RouteConfigs, TabNavigatorConfig);

1在TabNavigatorConfig中可以配置通用tabBar的通用属性；

2在具体每一个页面也可以具体配置当前页面及tab属性(navigationOptions)：如下（注意是static）

Static navigationOptions = {//单独设置tab

tabBarLabel: '我的',

headerTitle:'我的',

tabBarIcon:({focused})=>{

if(focused){

return <Image style={styles.tabBarIcon} source={require('./img/icon\_diamond.png')}/>

}else{

return <Image style={styles.tabBarIcon} source={require('./img/icon\_diamond\_gray.png')}/>

}

}

};

\*抓住一个最重要的结构：stackNavigator{ 嵌套普通screen , tabNavigator(也是一个screen) }

#####################

**\*一个**CreateBottomTabNavigator（）中**本身存在不能左右滑动的问题**：（坑了好长时间，是**这个控件本身就不支持**!!!）原因官方已给出：

<https://github.com/react-navigation/react-navigation/issues/4146>

<https://reactnavigation.org/blog/2018/05/07/react-navigation-2.0.html> 官方release note说明createBottomTabNavigator()中不支持swipteEnabled/animatedEnabled属性。

可以用createMaterialTopNavigatorf定制一下代替使用。

createMaterialBottomTabNavigator()需要引入新的依赖：react-navigation-material-bottom-tabs'

第三个createDrawerNavigator:同上面两种用法：

1.导包 2.引入3.createDrawerNavigator(routeMaps,configs)

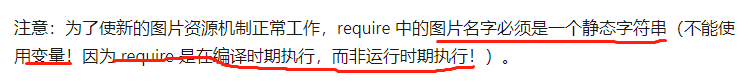
const homeDefaultPic = require('./img/tarbar\_icon\_home\_default.png');这也是对象

const homePressedPic = require('./img/tarbar\_icon\_home\_pressed.png');

另外，用require引用原生drawable下的资源时，**不带路径不带扩展名**，**只需要文件名**即可。

网络图片必须手动指定其大小，防止加载前后的大小跳动；

本地图片require(‘./…/xxx.png’)可以不指定大小，因为require是编译时期执行的，编译时就知道了本地已有图片的大小了。而非在运行时期执行！





碰到一个问题：drawer/tab无法响应手势滑动，并且其swipeEnabled:true，就是不生效呢？？

原因在于引入react-native-navigation时，需要同时引入react-native-gesture-handler，并且还有下面一步，与原生关联才能生效：（**原理就是给原生的activity加上了一层代理，接管了原生activity应该得到的触摸事件**）。



这样drawer/tab就可以滑动切换了！坑不少嘞！

**RN中的存储方式**：

AsyncStorage结合Promise的流式写法，不要太爽

匿名function:（error）=>{console.log(error)};

新写法: (p1,p2)=>void

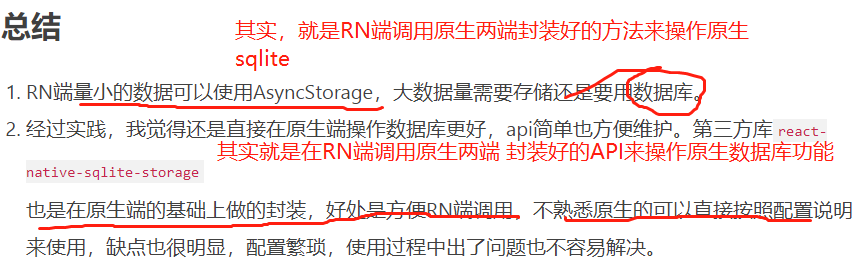
操作原生数据库：

两个选择：realm for RN , react-native-sqlite-storeage

react-native-sqlite-storeage配置时的坑:

project(':react-native-sqlite-storage').projectDir = new File(rootProject.projectDir, '../node\_modules/react-native-sqlite-storage/**platforms/android**')

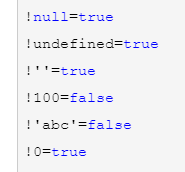
源工程的位置是在这里：/platforms/android,并不是文档上写的那个。



\*Promise代表一个未来的任务结果，这个任务有可能完成，有可能没有完成。**Promise模式唯一需要的一个接口是调用then方法**，用来注册当Promise完成或者失败时调用的回调函数

<https://www.jianshu.com/p/ec400befa594>

null , undefined , ’’ , 0 ===》false



### ****对于if(params){},不传参数，传的参数为null,传的参数为undefined,传的参数为""的时候if(obj)不成立false，反之则成立true.****

**网络请求fetch：**

**RN内部已经内置了俗称为AJAX的XMLHttpRequest**

**使用fetch API就可以。**

**网络状态的获取与监听：**

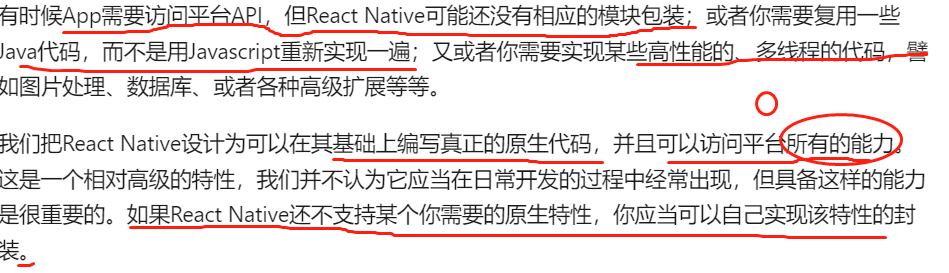


导入新包后再获取。

另外一种是用**react hook**来获取，暂时还不会用。后面再看。

**原生模块的封装和使用**：

官方对此的解释和定义：



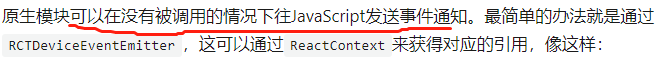
大概有以下步骤：

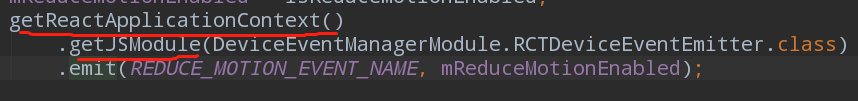
1. 编写原生代码module(ToastModule.class)，其中包含需要原生实现的功能(functions)
2. 将module注册到package,再在MainApplication中通过getPackages()将package返回，即将module注册到react-native,添加到NativeModules中.
3. 为了方便使用，可以编写一个.js文件，来封装下对原生module的调用。这一步他们叫导出一个js模块，**其实就是JS端中对应相应原生模块的js模块**。类似于跨进程的binder对象。远程代理对象/模块。(用户态与内核态)

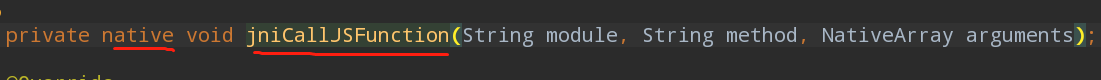
关于JS端与原生端之间的回调：

1. Callback
2. Promise
3. DeviceEventEmmitter

**前两种只能在原生端调用一次**，并且需要JS端调用时传递过来，而第三适用场景类似EventBus，一端发消息，另一端收消息。需要注册





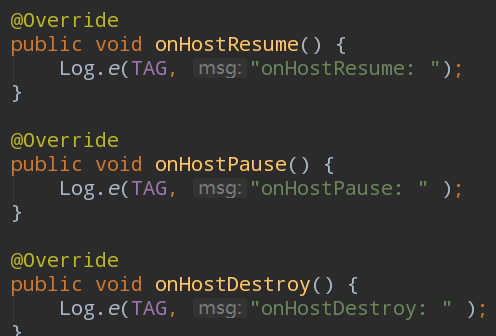


所有的getJSModule(class<?>),使用此功能，底层其实是从java层调用到本地Native层中的方法（中间用到了动态代理），进而转发到js中注册的监听器了（具体JS端注册原理流程后面再看）。

1. 从startActivityForResult中获取结果，官方文档上是有方法的。

GetcurrentActivity可以获得当前的宿主Activity。

5．监听activity的生命周期事件（比如onResume, onPause等等）。

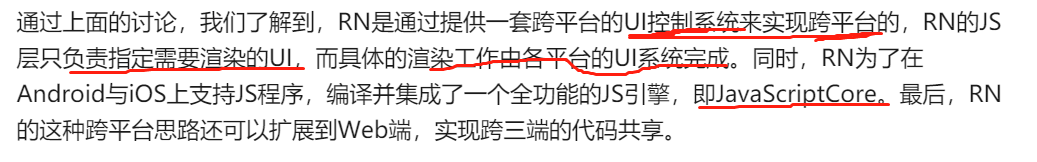


所以现在对，有能力访问原生平台的任何功能有更深的认识了！

Debug:



算是会使用chrome 开发者工具来调试RN的JS了。



End

Begin:

TODO-模仿一个知乎日报客户端RN版：

年前掌握RN开发，达到可以在项目中运用的水平。

End

Begin:

动画：

定义动画事件的 **Animated.timing(animateValue, conf<Object>)** 方法：config参数如下：easeing有很多预置函数。

{

duration: 动画持续的时间（单位是毫秒），默认为500。

easing：一个用于定义曲线的渐变函数。阅读Easing模块可以找到许多预定义的函数。iOS默认为Easing.inOut(Easing.ease)。

delay: 在一段时间之后开始动画（单位是毫秒），默认为0。

}

End

Begin:

ES6:变量的解构赋值

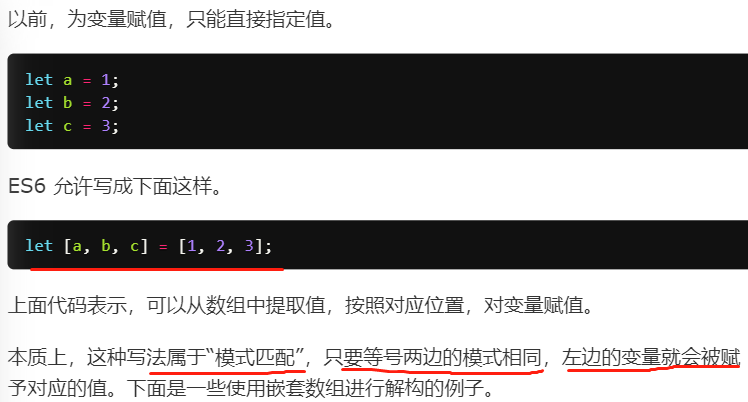
这到底是啥意思呢？ 数组解构是**严格按顺序的**，而对象则是**严格按属性名称解构的**。

**ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）。**

就是怎么给变量赋值呢？

本质上，这种写法属于“**模式匹配**”，只要等号两边的模式相同，左边的变量就会被赋予对应的值，不完全相等情况下，也能解构成功。

1. 数据的解构赋值





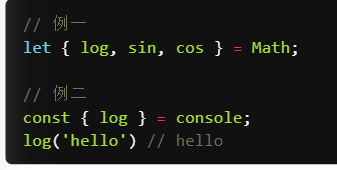
解构赋值允许指定默认值



注意，ES6 内部使用严格相等运算符（===），判断一个位置是否有值。所以，只有当一个数组成员严格等于undefined，默认值才会生效

1. 对象的解构赋值（这是一种赋值方式，通过解构已有对象，数组等任何可迭代的iterator，来为新变量赋值。

对象的解构赋值，可以很方便地**将现有对象的方法，变量**，**赋值到某个变量**。

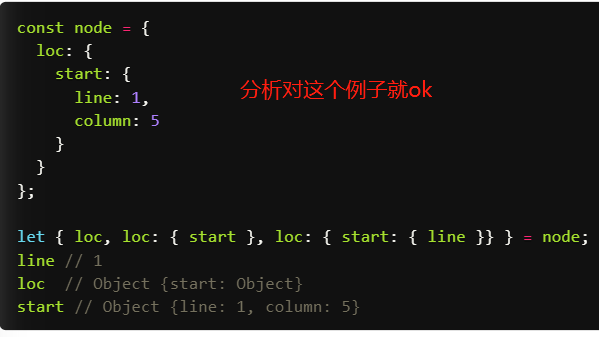




其实好记：

如果{foo,baz},这里匹配机制中，foo，baz即是要匹配的属性（名称一样），也是要赋值的新变量。

如果{foo:f , baz:b}，这时候，**分号前是匹配的模式**，**分号后才是真正要赋值的变量**。



1. 对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。

对象的解构也可以指定默认值。

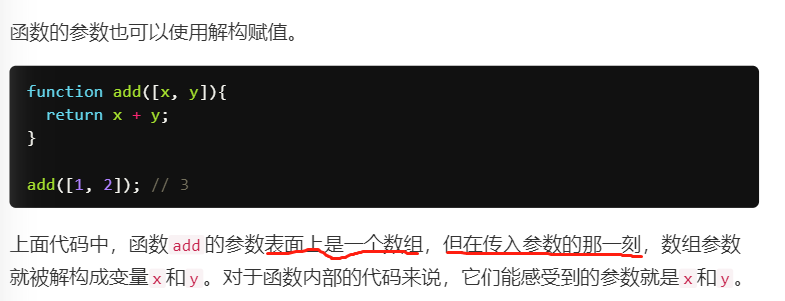
**默认值生效的条件是，对象的属性值严格等于undefined。===**

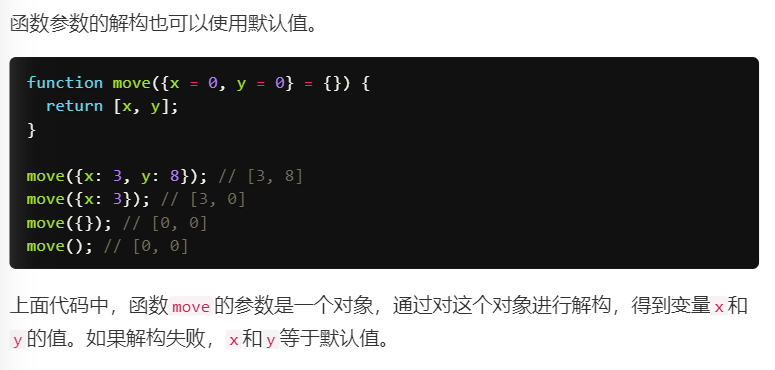
5.

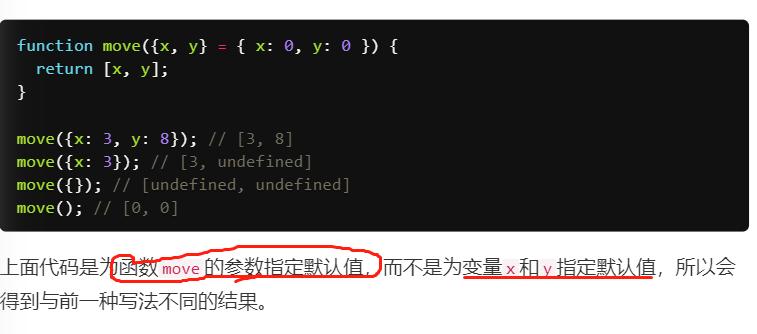


**\*解构赋值的规则是**，只要等号右边的值不是对象或数组，就先将其转为对象。由于undefined和null无法转为对象，所以对它们进行解构赋值，都会报错.

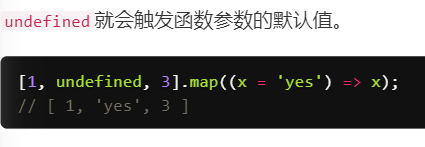
1. 函数参数的解构：





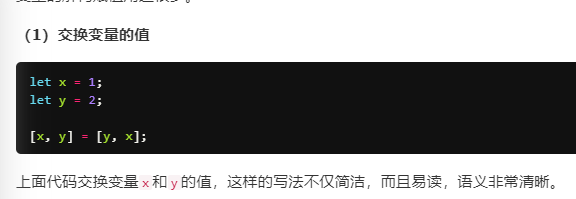


**一个是变量的默认值，一个是为函数的整体参数指定默认值 。两个东西**

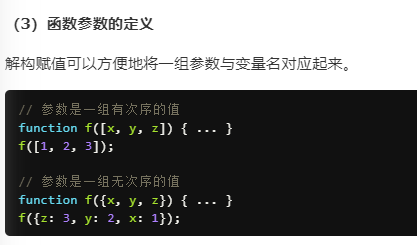


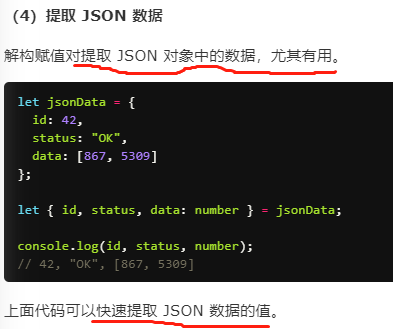
\*解构赋值的用途：

1.











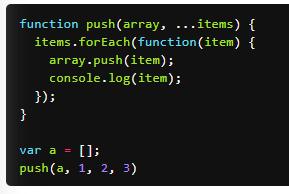


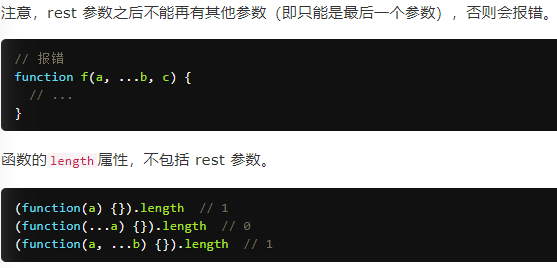


**\*ES6引入了rest参数**：

形式：（…变量名）

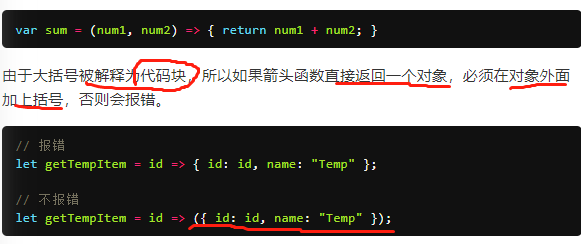
作用：**用于获取函数的多余参数**，这样就不需要使用arguments对象了。rest 参数搭配的变量是一个**数组**，该变量将多余的参数放入数组中.

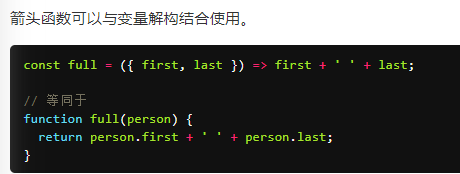


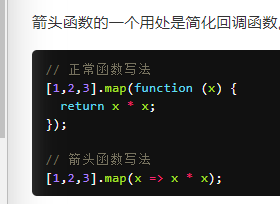


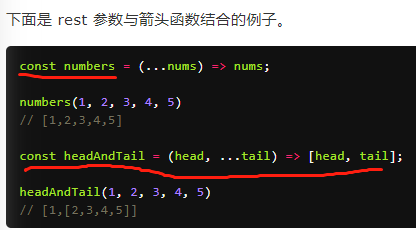
引入箭头函数：

基本形式：=>





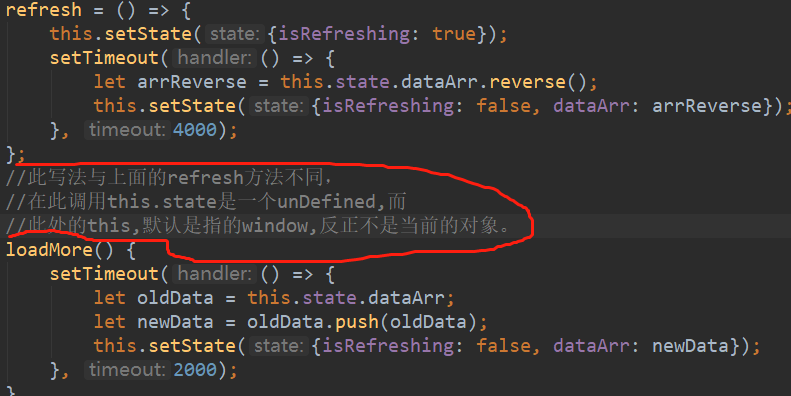


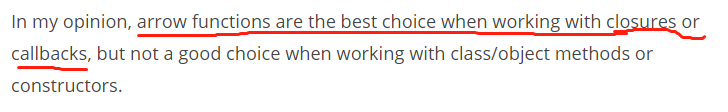


简要介绍：箭头函数中的this，指向与一般function定义的函数不同，

**箭头函数this的定义**：箭头函数中的this是在**定义函数的时候绑定(或者说是箭头函数定义生效时绑定)或者说箭头函数外部所在的作用scope执行时，来确定this.因为箭头函数没有this,在箭头函数中使用this，这个this是用它的父作用域的this引用，而父作用域的this引用只有在使用时才确定出来)**，而不是在执行函数的时候绑定。(?)

实际使用时遇到的一个实际问题？





这个作者认为箭头函数最好/合适的使用场景是：**回调**和**闭包**时。

In regular functions， the this keyword represented the object that **called the function**, which could be the window, the document, a button or whatever.

With arrow functions， the this keyword **always** represents the object that defined the arrow function.

**Because ‘this’, always references the owner of the function it is in**

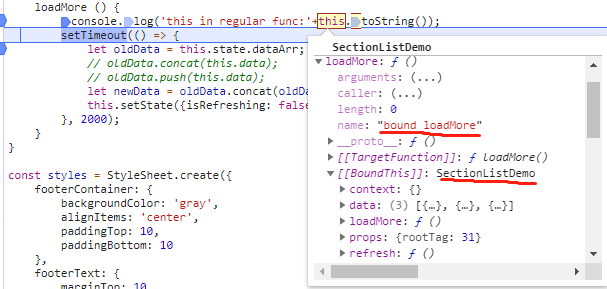
These are the reasons why we call that the value of this is determined by a function's **execution context. 认识还不深呢**

In arrow functions this will always take its value **from the outside**. In fact, the arrow function does not even have this.

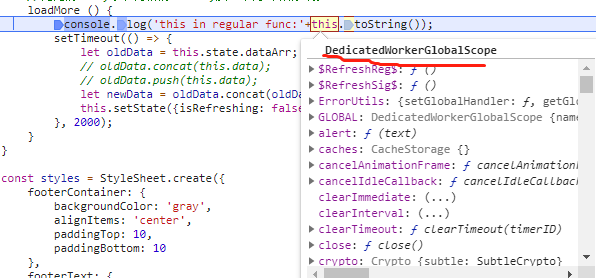
好像类中定义的函数，如果没有显式绑定this，隐式绑定的this，好像是window或者global对象呢！

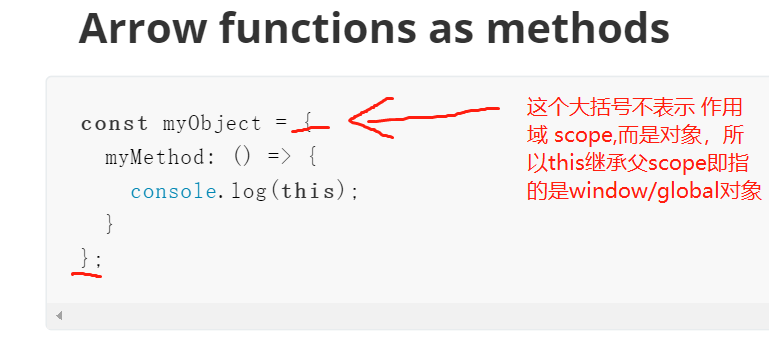
构造器中显式绑定了this:

this.loadMore = this.loadMore.bind(this);之后的变化如图：

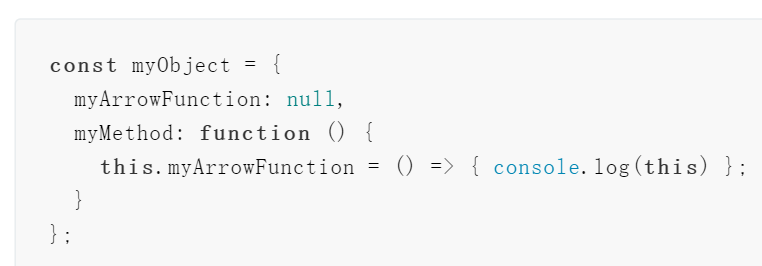


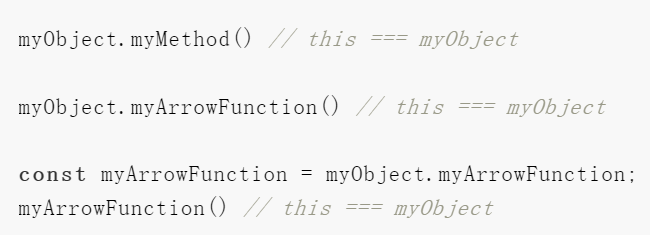
如果没有显式绑定呢？loadMore方法中的this,是一个Global对象：





第二个例子：





注释：Clearer now? When we call myObject.myMethod(), we initialize myObject.myArrowFunction with an arrow function which is inside of the method myMethod, so it will inherit its scope. We can clearly see a perfect use case, closures.

差不多感觉更多一点了：而且注意上面：所以现在对箭头函数的this绑定是在定义生效时确定/继承过来的，有一个新的认识了吧。

一旦确定后，再对其调用显式绑定(call,apply)是无用的；调用bind是抛异常的。

this.loadMore = this.loadMore.bind(this)，

绑定后产生的新方法才是包含新this的方法。默认此方法覆盖掉了此对象之前的那个loadMore方法。

<https://tylermcginnis.com/this-keyword-call-apply-bind-javascript/>



这样处理显式绑定之后，就可以使用普通函数作为回调函数并且在普通 函数中使用this正确指向了。

可能感觉有点感觉了：

另外，**一般函数**有隐式绑定this和显示绑定this(apply,call,bind),显式绑定，其实就是重新给函数的this赋值，使其指向其他对象，产生不同的结果。**箭头函数**因为其自身本身就没有this引用 ，所以也就没有显式绑定一说了。

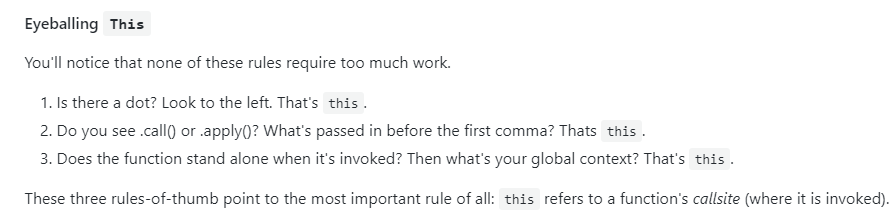
Begin:

关于window的函数类似setTimeout/setInterval等函数中传递回调函数时，回调函数中的this，从以前到现在的解决办法，看第一篇文章，讲的非常好。

<https://stackoverflow.com/questions/2130241/pass-correct-this-context-to-settimeout-callback>

<https://gist.github.com/zcaceres/2a4ac91f9f42ec0ef9cd0d18e4e71262>

上面两篇文章结合看，如何确定this到底是指的哪个对象的问题就弄清了。



4．对于.bind(obj),需要跳回去看是在哪里进行的bind

5．最后一个是回调函数callback,

Callbacks seem to introduce another layer of confusion to this, but they don't need to.

如何判断呢？

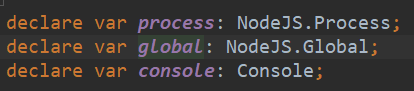
寻找回调方法的调用方(callsite).

总结：In sum: When in doubt, **look at the callsite**!

End

Redux:其实就是用来管理应用state状态的。三个核心东西,Store , Action , Reproducer.

管理应用数据流的。



三个全局变量，任何js中均可直接调用不用导入。

End

总结一下：今天主要学习了存储与网络模块的封装，storage对AsyncStorage封装，同时还有storage.async = {‘methodName’: method , ‘methodName’: method,},方法名必须和所要加载的数据的key相同。这样才会本地没有合格数据时，调用对应的网络方法去获取最新数据。这个项目里没有用Fetch()，而是用了Axio代替进行网络访问。效果一样：先是封装了通用的async GET/POST方法到一个js文件，然后封装了所有用到的api地址，封装到一个js文件中。然后在另外一个js文件中定义了实际的detail/before等实际用到的方法到一个对象中并导出，最终目的是在storage.js中对storage.async进行赋值等候调用。

涉及的几个文件：api.js + axios.js + storageAsync.js =>storage.js

还涉及到了好几个外部UI component的使用：ParallexScrollView + react-native-base等

算了不记了没有啥意义。

Begin:

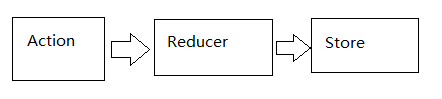
ES8中新引入的sync / async/await:它比Promise更进一步，更加方便。其异常捕捉也比较方便。

Async 修饰方法，await修饰具体某个需要异步的操作。

End

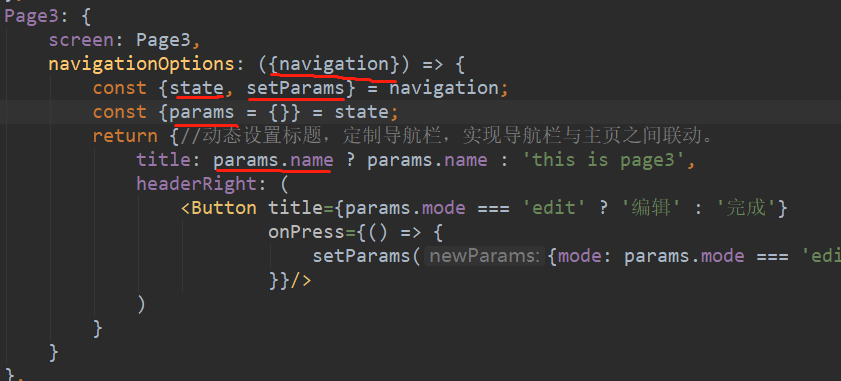
这个…是什么意思呢？





\*createAppContainer的作用：其实是对导航器进行wrapper,使其能作为一个组件Component在页面上显示。先这么记着就行。

createStackNavigator:参数使用与传递，



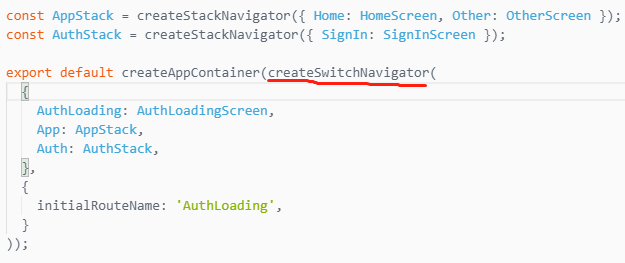
参数以键值对的形式存在state对象中的。

createBottomTabNavigator:

关键就是定义tabbarLabel & tabbarIcon这两个选项，一个是文字一个是图标

createSwitchNavigator:适用场景，RouteMap中的各个Route，相互不能通过按返回键再返回来到上个Route,只能通过navigate()，login/splash.下面是官方的使用例子：

screen,各种navigator都可以嵌套使用的。很有用的一个导航器。



一个导航器中的各个页面是共享同一个navigation对象的。不同的导航器间就不是同一个nav对象。

createDrawerNavigator:

\*自定义侧拉抽屉选项卡，这块需要提供给contentComponent一个组件。后面再看看

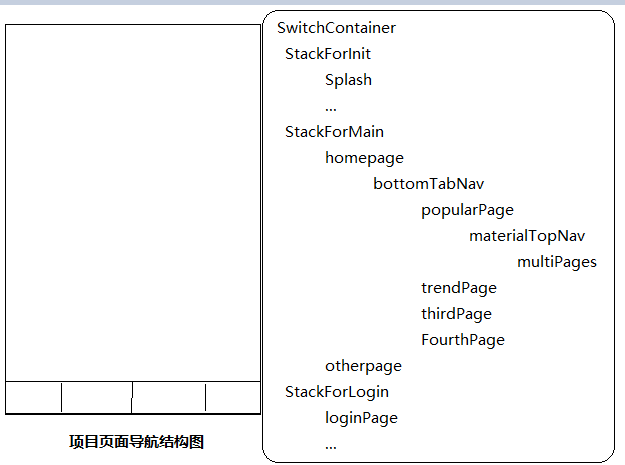
导航器创建好后要使用时，**必须通过createAppContainer()包装成组件后**，才可以象普通 组件 一样嵌入到view树中。

项目的主结构其实也简单：

createSwitchNavigator=createStackNavigatorForInit+ createStackNavigatorForMain+createStackNavigatorForLogin

createStackNavigatorForMain=HomePage

HomePage = createBottomTabNavigator(containers 4 screens);



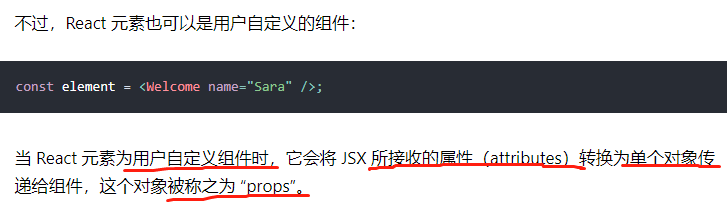
\*动态修改，底部导航栏tabBar的激活色：和上面抽屉的drawerContent自定义是一个思路。

实际上，JSX 仅仅只是 React.createElement(component, props, ...children) 函数的语法糖。

JSX 中的 Props:…这就是属性展开运算符

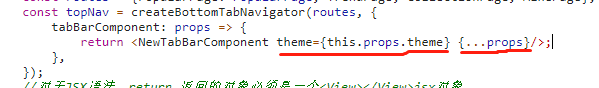


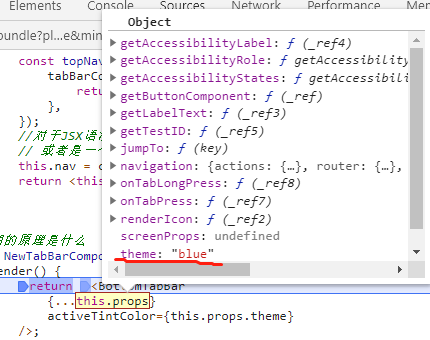
<https://react.docschina.org/docs/jsx-in-depth.html>



官网上说对于这些属性都会转换为props单个对象传给组件。那对于上面的写法，props={name:’sara’}, 而对于下面的写法呢？试试：下面的这种写法{ …props}是将原有对象展开复制到一个新对象中。并且，将单独的属性添加进了这个新对象中了：如下 图

（**将theme属性添加到了{…props}对象中去了**。）作了一个融合。**类似于Obj.assign({},souce1,source2…)**





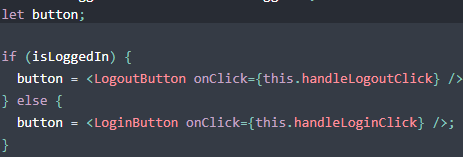




**React:条件渲染**

### 元素变量

将要渲染的元素储存在暂时储存在变量中，结合if使用



### 与运算符 &&

通过花括号包裹代码，你可以在 **JSX** 中**嵌入任何表达式**。

在 JavaScript 中，true && expression 总是会返回 expression, 而 false && expression 总是会返回 false

因此，如果条件是 true，&& 右侧的元素就会被渲染，如果是 false，React 会忽略并跳过它。

### 三目运算符

使用 JavaScript 中的三目运算符 condition ? true : false

### 阻止组件渲染

在极少数情况下，你可能希望能隐藏组件，即使它已经被其他组件渲染。若要完成此操作，你可以让 render 方法直接返回 null，而不进行任何渲染

**学习下ES6中的模板字符串：**



以$符开头如： `tab${index}`

看Component中的参数层级关系：

This.props

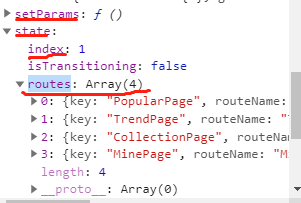
Navigation

setParams

state

index

routes Array



\*调用navigation对象的setParams(\*\*\*),它是给当前页面对象的route对象中添加了一个的对象参数。Routes[index][key] = value.

通过this.props.navigation.setParams()可以自动触发组件的render方法重新被回调的。

学习一个新函数：

作用：

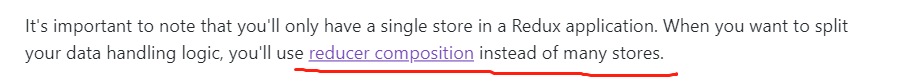
**Copy the values of all of the enumerable own properties from one or more source objects to a target object. Returns the target object；**

**ES6新特性，复制一个对象，还可以加入/覆盖新属性，后面的souce 覆盖掉前面的属性。**

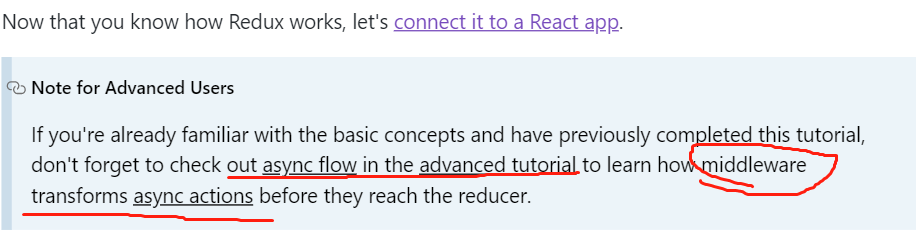
Object.asign(target,source1 , source2,source3)

Object.asign(target,source[])

关于reducer:

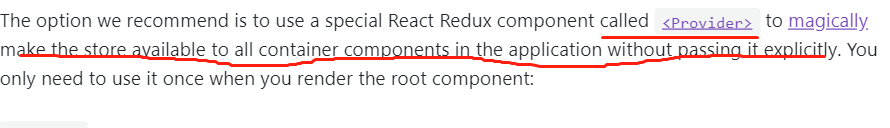


引入多个reducer进行组合，而不是许多store。



引入中间件的意义：

引入<Provider/>组件的意义：



Reducer:必须是一个纯函数，那何谓纯函数(pure function)呢？

符合特性：函数的输出结果只唯一取决于输入参数，同一输入参数必要产生相同的返回结果。因此对于Math.Random()它就不是一个纯函数。

一个关键点是redux是如何将组件与store中的state联动起来呢？而且还是双向联动呢？

有两个关键点：mapState2Props，指示如何将state中的变量传递给组件的props

MapDispatch2Props,指示如何将store中的dispatch(action)触发方法传递给某 个组件props中使用，以在该组件中触发某个action对象，以更新state tree.

而这两个变量要想生效，必须将组件与redux的store进行关联，如何关联呢？就是使用

‘react-redux’中的createConnect(mapState2Props, mapDispatch2Props)( 组件 ).

1. Const mapDispatch2Props = dispatch => ({//将dispatch方法对象映射成另外一个对象

onThemeChange: theme => dispatch(actions.onThemeChange(theme));

});

1. Const mapState2Props = state => ({//

theme: state.theme.theme;

});

先学会redux与现有项目结合，

Provider => store ＝＞combineReducers=>all reducers from /reducer/index.js

环环相扣

碰到一个问题：

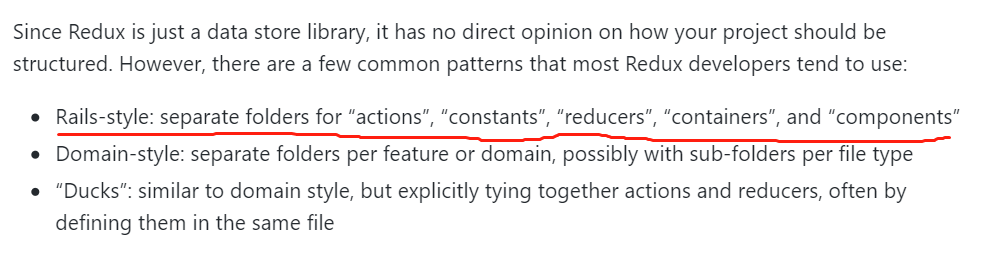
Error: Warning: React.createElement: type is invalid -- expected a string (for built-in components) or a class/function (for composite components) but got: <Provider />. **Did you accidentally export a JSX literal instead of a component?**

为啥这个对应于<DynamicTabBar/>组件的state中的key,为啥是changeTheme呢？

答：

在combineReducers()中传入的各个reducer所对应的key，就是生成的state tree中该state

theme:onThemeChange ,对应的state对象的结构为：{theme: { theme: ‘red’ } }.



以上是关于code structure的问题：倾向于第一种去分。

关于与redux结合后的渲染问题，组件何时被重新渲染或者重新构造一个全新的对象？

是否重新渲染的条件是进行一个浅比较，shallow check equlity,(**好像这个地方是可以性能优化的**)。

构建缓存模块：

ES6中引入class中的私有方法和私有属性：

1．\_function，从命令上加以区分。实际和普通方法没有什么区别。先这么用。

注意：this指向的问题，类的属性与方法：static function(){ }, Foo.prop = foo;

实例的属性与方法： 实际上，类的所有非static方法都定义在类的prototype属性上面，





如果不写constructor方法，JavaScript 引擎会自动为它添加一个空的constructor方法。

constructor方法默认返回实例对象（即this），完全可以指定返回另外一个对象。是不是可以使用代理呢

与 ES5 一样，实例的属性除非显式定义在其本身（即定义在this对象上，this.prop = ‘xxx’），否则都是定义在原型对象属性上（即定义在class上-其prototype对象上）。

基本上，ES6 的class可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。

Promise也有一些缺点。首先，无法取消Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。第三，当处于pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

如果改变已经发生了，你再对Promise对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

类似于rxJava中的调用：





**如果给then中传入了处理reject=>{ }回调函数时，catch中的回调就不再执行了；**

**不传入时，不管是reject还是throw new Error(),都会走catch中回调函数。**

因为 Promise 的状态一旦改变，就永久保持该状态，不会再变了142325199012211516



建议总是使用catch方法，而不使用then方法的第二个参数。

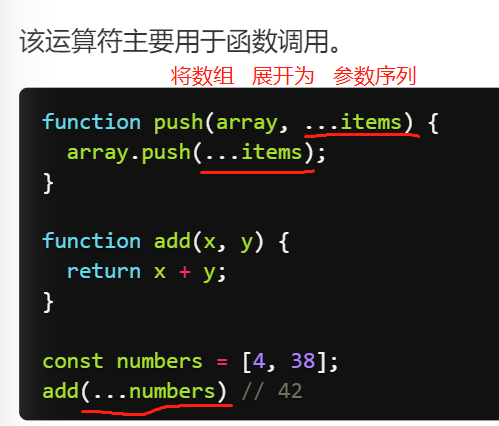
注意下‘其中**事件循环的概念与作用’，**

**一个事件循环的概念：**



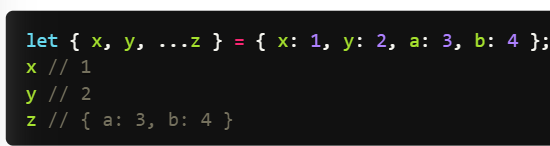
异步Action:

1. **…扩展操作符用于数组：**

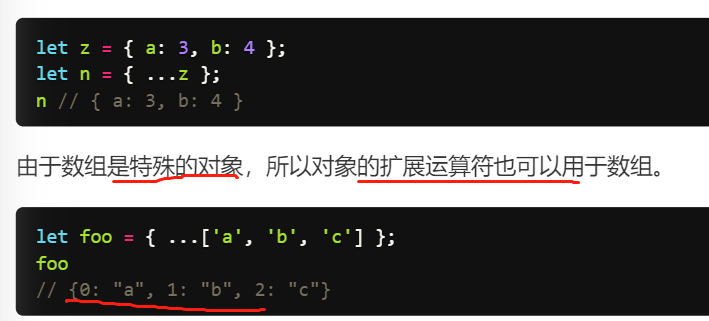


2.扩展操作符用于对象：

一种是：解构赋值



一种是：用于取出参数对象的所有可遍历属性，拷贝到当前对象之中（**确实特殊）**



**关于异步Action的写法**，一个指导思想是：

先定义所有同步的action Creator,然后再定义同步Action creator与异步操作相结合的function (返回值是一个函数).直接写到同一个js文件中即可。

**异步Action其实是一个函数。它可以dispatch 同步action而已。**

**它提供的是位于 action 被发起之后，到达 reducer 之前的扩展点**

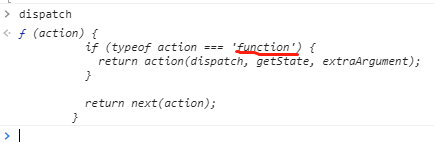
**总结一下：何为中间件？作用是什么？具体如何使用？**

**以一个页面容器为例，需要绑定state树中的某个State对象：**

1. **首先定义Action 的类型**
2. **其次定义同步的action生成函数，如果有异步操作，再定义异步函数，来将异步操作与同步的action生成函数结合起来。（直接在一个action.js文件中定义异步函数方便）**

**(dispatch)=>{dispatch(sync Action A);…;dispatch(sync Action B)}**

**Redux thunk 中遇到如果遇到是个函数，会自动调用此函数，并传入dispatch,state,args.截图如下：**

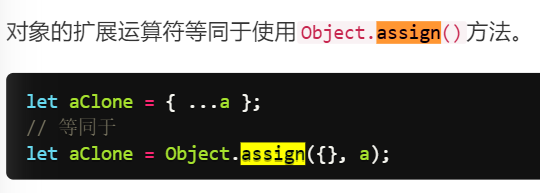


1. **定义与各同步action操作相对应的reducer函数（这个与是否异步无关）**
2. **聚合，将新定义的reducer函数，添加到combineReducers({})中进行聚合，它的key就是生成state树中对应属性的key.**
3. **构建完成之后，接下来是如何在组件中使用state树中，对应那一小部分state对象的属性呢？以及如何在组件中更新它呢？甚至说更新其它某个小部分的state对象呢？**
4. **封装展示型组件为容器型组件，通过connect()函数将组件与state关联起来，并且可以传递属性进来。（即将state注入到展示型组件中）map State to Props.**

**而且属性中有const {dispatch} = this.props;属性，使用dispatch可以在组件的任何地方dispatch任意的（同步的/异步的）action对象。**

**MVVM模式，要更新界面，只需要更新界面关联的数据即可。数据=》界面是单身影 响的。**

最后一个Object.assign()是 ES6 新增的。





Es6中export & export default的区别：

Export default xxx的实质是给要导出的xxx起了一个默认的名字，而且用于指定模块的默认输出。显然，**一个模块只能有一个默认输出，因此export default命令只能使用一次**。所以，import命令后面才不用加大括号，因为只可能唯一对应export default命令。

对于export xxx，导入时import {xxx} **名字必须一致**。

Flutter是框架**自带的一套渲染层**，不像RN使用的是JS驱动，中间转换为对原生渲染系统 的调用。

FlatList:列表滑动时 onEndReached={}被两次调用的解决办法：

<https://github.com/facebook/react-native/issues/14015>

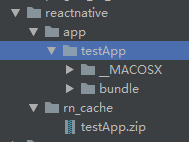
Begin:

已有项目集成RN开发环境与自更新：

利用某个Activity,作为初始化RN环境和页面容器即可。

根据已有项目中添加的（热）更新方式中可以学习到几点：

1. 整个更新流程是在启动app的StartActivity & TLifeHomeActivity中，获取目前服务器上最新的RN包的版本，并判断 是否需要重新加载更新本地已有的RN包：如果需要更新，直接开启下载zip 包，并且在下载完后自动解压到约定的路径(先删除已有的老zip和已有的解压文件)。到此就算更新完成了。
2. 网络传输RN包用zip，减少传输体积；代码中使用ZipInputStream解压zip包到指定夹
3. 如何在已有项目中以某个Activity作为RN环境初始化与页面的容器（使其同时具有纯RN项目中MainApplication与MainActivity的功能）
4. 下载后的zip先获取md5校验值与服务器下发的md5值一样后再解压使用。
5. 下载的压缩包在此进行解压到/data/data/packageName/files/reactnative/app/appId/



End



Begin:

学会封装组件，此次封装的是导航栏，其中含有状态栏与导航栏。关键在于掌握组件属性类型的定义方法PropTypes.以及默认属性的使用，只有在某个属性是undefined时才会触发使用默认属性值哦。

End