

---

# Table of Contents

|                            |       |
|----------------------------|-------|
| 序言                         | 1.1   |
| 第一章. React Native介绍        | 1.2   |
| 第二章. React Native基础        | 1.3   |
| 第三章. 启动你的App               | 1.4   |
| 第四章. 一些一定要知道的组件            | 1.5   |
| 第五章. 是时候说说样式与布局了           | 1.6   |
| 第六章. 储存                    | 1.7   |
| 第一节 AsyncStorage           | 1.7.1 |
| 第二节 Realm                  | 1.7.2 |
| 第七章. ListView与FlatList     | 1.8   |
| 第八章. 导航器(React Navigation) | 1.9   |
| 第九章. 网络                    | 1.10  |
| 第十章. 打包应用                  | 1.11  |
| 附录：样式                      | 1.12  |

## 序言

嘿，欢迎进入React Native的世界，相信我，这会是一种酷酷的App开发方式。在过去的一段时间里，React Native成为了越来越多开发者的App开发方式，选择React Native就是选择了一个大社区，有什么问题可以较快的得到解决。

在React Native发布后，作者君特别想尝试一下这种新的开发方式，不过很不幸，那个时候只能用来开发IOS应用。所以尝鲜这种事拖了好久。使用之后发现，这确实是个好东西，至少不用为了开发个小应用而去学原生应用的开发，用些前端的知识就能搞定了。

本想做些视频来分享的，不过周围的环境总是很嘈杂，安静的地方很难找的到。所以，假期过后视频就没怎么录了。另一方面，React Native版本更新很快，做视频很容易就落后了，想来想去，还是书这种形式比较好，更新也很是方便。

React Native并不难学，不要给自己一些奇怪的暗示。

本书为入门向，所以有些更加深入的东西需要自己去探索。现在，翻到下一页，React Native之旅即将开始~

## React Native介绍

虽然，你可能已经对React Native有一定的了解了，但是按照惯例，还是要简单介绍一下。

React是由Facebook的工程师们在2013年开源的一款前端框架，现在已成为最火热的前端框架之一。React Native则是使用React来开发移动应用的方式。React Native底层使用JS与原生进行通信，这使得其性能比Hybird强了不少，如果觉得Hybird使你的应用变得卡顿，那么投入React Native的怀抱是一个好的选择~

React Native在底层通过一套机制天然的支持ES6,这意味着你可以在你的应用中愉快的使用ES6语法，而不用头疼兼容问题。

使用React Native意味着更快的调试速度，当你在配置中开启一些选项时，React Native可以做到保存后自动更新，摆脱耗时巨长的编译过程。还可以在Chrome中获取调试信息，这使排除异常的过程更加轻松。

React Native具有自动更新的能力。前一阵子苹果警告了一些使用热更新的应用，不用担心，这并不是针对React Native，那些被警告的应用使用了有害的API，而React Native热更新是从网络获取JS bundle，不会影响安全性。

虽然React Native很棒，但是有些事情还是要知道的。

1.React Native抛弃了HTML，所以需要使用一种叫做JSX的东西，如果觉得有些畏惧，那么请放心，JSX还是很易于掌握的。

2.React Native并非万能，有些事情它不能做到。举个栗子，如果我想做一个桌面小部件，那么我在这个时候就需要一些原生知识了。

3.React Native不能完全取代原生应用，它们之间应当是相辅相成的。

# React Native基础

## 本章提要

- 1.初识JSX
- 2.组件
- 3.PropTypes和State

## 1.初识JSX

React Native抛弃了HTML，所以只好用JSX喽，在这里，我们来简单的了解一下JSX。

JSX是一个JS的语法糖，写起来就像XML。嗯，如果你不知道什么是XML的话，就脑补HTML吧，它们都需要一些标签。比如这样：

```
<Text></Text>
```

还有这样：

```
<Image />
```

怎么样，有没有一丝熟悉的感觉？JSX就是这样简单，我们只需要一个`<`加上组件名字和`/>`就可以了（其中Text，Image就是组件名）

## 2.组件

在React Native中，有两种组件，一种是无状态组件一种是高阶组件。无状态组件，顾名思义，就是没有状态的组件，写一个无状态组件就和写一个函数一样简单。由于无状态组件的性能较高，我们一般会在App性能优化的时候使用无状态组件。下面是一个无状态组件的例子：

```
const Hello = () => {
  return (
    <Text>我是一个无状态组件</Text>
  )
}
```

其中，Hello是组件的名字，这里有个友情提示，在React Native中，组件的名字最好都是首字母大写，这样可以避免一些奇怪的BUG。然后在return中写组件的内容，注意，使用文字的时候，一定要在文字外面套上Text组件，否则会红屏报错。如果想要用Props(属性)，要传一个props参数，就像这样

```
const Hello = (props) => { ... }
```

如果不知道什么是Props的话，别着急，一会儿会说到的。

接下来是高阶组件。为啥叫高阶组件嘞？因为它能做的事情比无状态组件多(◦ω◦)。写起来是酱紫的，稍微复杂一点：

```

class Hello extends Component {
  constructor(props) {
    super(props);
    this.state = {};
  }
  render() {
    return (
      <View>
        <Text>Hi,很高兴认识你</Text>
      </View>
    )
  }
}

```

这是一个比较完整的组件，包含了Props和State(状态)。在高阶组件中使用state和props，我们需要使用constructor来初始化一下~这里的super(props)是为了可以在constructor使用 this.props，当然，如果不需要这些，则不用写（虽然没写，但是按照ES6规范，class需要有一个constructor，所以constructor会被自动加上）。就像这样：

```

class Hello extends Component {
  render() {
    return (
      <View>
        <Text>略略略/Text>
      </View>
    )
  }
}

```

### 3.Props和State

嗯哼，刚才稍微提了一下Props和State，是不是很想知道这俩是啥嘞~别着急，我这就说(◦ω◦) Props的意思是属性，在React中，我们可以实现组件的复用(毕竟不复用组件太浪费了)。为了实现组件的复用，我们希望可以对组件做些定制，这个时候，我们就需要使用Props了。往组件中传些不同的参数，从而使组件具有不同的功能。我们称这些参数为Props。如果不太理解，没关系，来看这个栗子。

设计师为衣服厂设计了一些衣服，但是为了照顾到不同的人群，所以要有不同的型号。高的，矮的，胖的，瘦的。虽然有不同的型号，但是是同一款衣服，我们可以认为高矮胖瘦是参数，衣服的款式是一个组件。

我们来看一下Props怎么用。首先写一个叫做Eat的组件：

```

class Eat extends Component {
  render() {
    return (
      <Text>我想吃</Text>
    )
  }
}

```

在组件里使用Props，我们需要一对花括号 {} (写其他的JS语句也需要花括号)就像这样:

```

return (
  <Text>我想吃{this.props.food}</Text>
)

```

在props后面跟着的是属性的名字

接下来，我们写另一个组件Dinner，并引入Eat组件：

```
class Dinner extends Component {
  render() {
    return (
      <Eat />
    )
  }
}
```

然后使用props:

```
<Eat food='鱼' />
```

这样最终会显示我想吃鱼，这里的鱼是food的值。到这里，Props的栗子就结束了，我们来看看State。

一般来说，组件是需要和用户进行交互的，组件会随着交互而发生变化，这里的变化指的是State也就是状态。

下面是一个用到了状态的组件:

```
class NeedState extends Component {
  constructor(props) {
    super(props);
    this.state = { text: '' };
  }
  render() {
    return (
      <View>
        <Text onPress={() => { this.setState({ text: '状态变了' })}}>点我！</Text>
        <Text>期待会发生的事情</Text>
      </View>
    )
  }
}
```

首先，我们需要constructor函数来初始化状态，别忘记用 this.state 写入状态的名字。onPress 可以帮我们在点了某个组件之后执行一定的动作。我们在这里用它来触发 setState 来改变状态。

如果没有奇怪的错误的话，点击 点我 这个文本之后，下面的文字会变成 状态变了。

State大概就是这么用的，没什么难度。在使用React Native的过程中，我们会时常用到Props和State，所以记下它们的用法还是很有必要的。

# 启动你的App

本章提要

- 1.搭建React Native环境
- 2.启动App

## 1. 搭建React Native环境

啊哈，我们即将近距离接触React Native了。不过，我们得在这之前安装一下React Native环境。

首先，我们来安装Node.js环境。如果你的系统是Windows，那么到Node.js官网上下载安装包，一路next就行了。如果是Linux或MacOS，那么作者君推荐使用神奇的nvm。nvm的地址是: <https://github.com/creationix/nvm> 按照说明安装即可。

然后执行

```
nvm install x.xx.x(要安装的版本号)
```

这样Node.js就安装完毕了。

接下来，我们来安装Java环境。去Oracle的官网上下载安装

包<http://www.oracle.com/technetwork/java/javase/downloads/index.html> 然后安装。

同样的，Windows用户一路next然后设置JAVA\_HOME环境变量即可。对于Linux和MacOS用户来说，会稍稍麻烦一些。首先把安装包解压到opt目录，然后在/etc/profile里加上

```
export JAVA_HOME=/opt/(这里是安装包解压后的路径，别填错)
export JRE_HOME=${JAVA_HOME}/jre
export CLASSPATH=.:${JAVA_HOME}/lib:${JRE_HOME}/lib
export PATH=${JAVA_HOME}/bin:$PATH
```

最后别忘了执行一下 `source /etc/profile`

如果执行 `java -version` 可以正常输出的话，那么恭喜啦，对很多人来说不怎么好装的Java环境被你安装好了哟

然后安装安卓环境，我们先去Google的开发者网站<https://developer.android.google.cn/studio/index.html> 上下载Android Studio。安装好Android Studio，我们需要安装SDK及SDK Tools。首先，我们来安装SDK。在SDK Platforms，选择Show Package Details，然后勾选Android 6.0下的Google APIs，Android SDK Platform 23，Intel x86 Atom\_64 System Image还有Google APIs Intel x86 Atom\_64 System Image。接下来，在SDK Tools窗口中选择Show Package Details，然后选中有23.0.1的选项并下载。最后，我们需要在/etc/profile中写入环境变量

```
export ANDROID_HOME='/home/elven/Android/Sdk'
```

Windows用户也需要设置 `ANDROID_HOME` 这个环境变量。

如果是老司机的话，Android Studio是不需要安装的，不过为了及时，方便的更新SDK，还是装上吧。

再来三步，环境就安装好啦，如果感觉有些焦灼，给自己冲杯茶，咖啡或果汁是很好的选择~

接下来，我们需要安装React Native。在终端中执行

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

然后安装Yarn

```
npm install yarn -g
```

最后安装Watchman，按照这里(<https://facebook.github.io/watchman/docs/install.html#build-install>)的说明一步步安装。如果有报错的话，多半是有什么没装，按照报错里的信息装一下就Ok了。另外，Windows系统不需要安装Watchman（其实是Watchman不能在在这个系统运行）  
到这里，开发环境就搭建好了。

## 2. 启动App

在启动App之前，我们得创建一个新的项目（啥都没有拿什么启动，哈哈）在终端输入

```
react-native init 项目名
```

等待一会儿，进入项目目录。好了，拿出你的小手机和数据线并与电脑相连。新开一个终端，然后进入项目目录，执行

```
react-native start
```

回到原来的终端，执行

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

如果是Windows的话，只要执行 `run-android` 这条命令就可以，packager可以自行启动，这里分开是为了保证App可以正常启动。

闭一会儿眼睛，稍稍休息下之后，你就会在手机上看到启动好的App了！

作者君猜，你可能不知道项目中的`index.android.js`里的最后一句是什么意思，所以就在代码里小小的解释一下：

```
AppRegistry.registerComponent('项目名称', () => 组件名称);
```

## 一些一定要知道的组件

### 本章提要

- View组件
- Text组件
- Button组件
- Image组件
- 第三方组件

很多组件是开发应用的基石，如果没有这些组件，那么是写不出来一个应用的，在这一章，我们来学习一些一定要知道的组件。

在使用组件的时候，忘记引入组件是不行的，我们来看一下怎么引入组件。

```
import {  
  Text,  
  View,  
  Button,  
  ...其它的组件名字  
} from 'react-native';
```

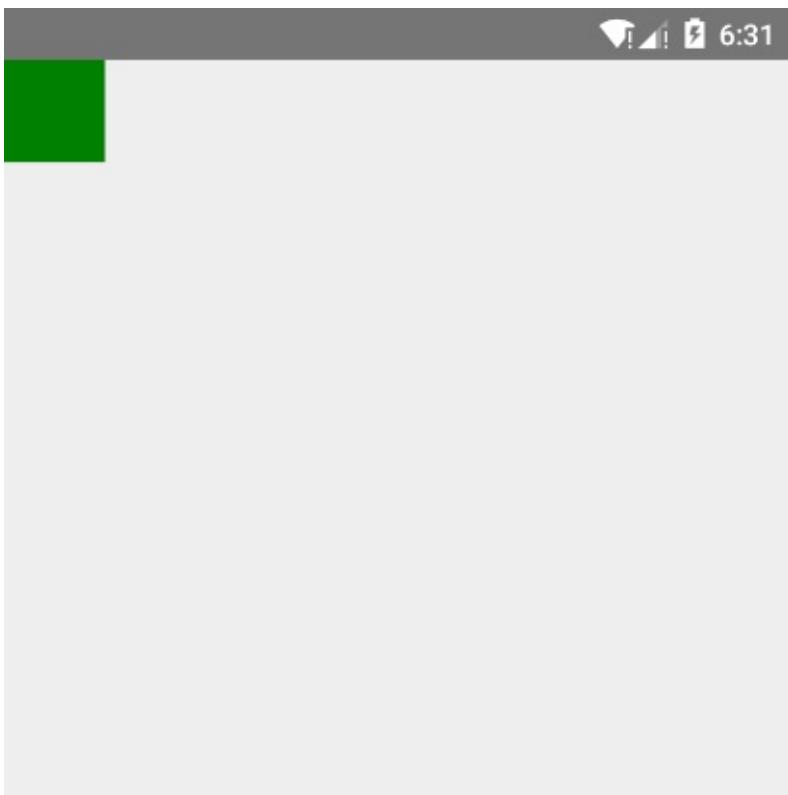
这样就可以依次引入Text，View和Button组件了~

## View组件

这么说吧，第一个就介绍View组件是有原因的。View组件可是最基础的组件，我们经常需要拿View组件当容器使用~ View自己没有固定的样式，我们需要自己为它定义样式。

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

这样，在屏幕上便可以出现一个绿色的，边长为50dp的正方形



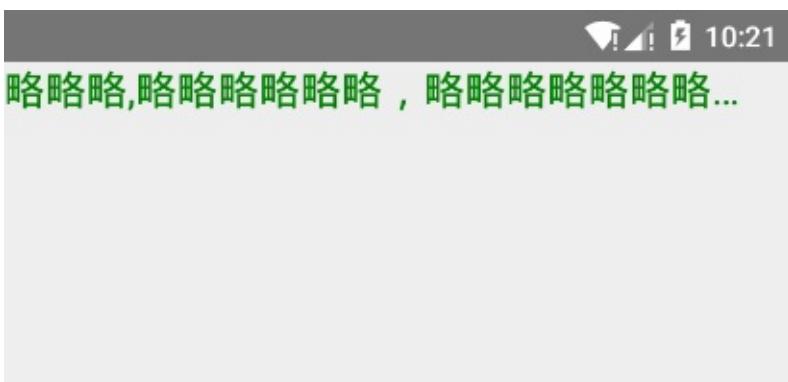
样式什么的，我们会在下一章讲到，在这里简单看看就好。

## Text组件

一般来说，一个应用总是离不开文字的，在React Native中，文字外面需要套上Text组件才可以， 我们可以通过Text组件控制文字大小及颜色，文字行数，还可以通过点击Text组件来触发一些动作（函数）。栗子如下：

```
<Text numberOfLines={ 1 } style={{ fontSize:20, color:'green' }}>  
略略略,略略略略略略, 略略略略略略略略略略略略  
</Text>
```

我们通过numberOfLines来控制行数，超过设定的行数便会在设定好的最后一行显示省略号。通过fontSize和color设定了文字的大小及颜色。效果如下：



## Button组件

Button组件大概是在0.37版加入的，在那之前按钮什么的要自己写，如果对官方出的这个不满意的话，那就自己写或找一些第三方组件吧~ 我们写两个按钮当作栗子：

```
<View>
  <Button color="green" title="我是一个正常的按钮"/>
  <Button disabled={true} title="我是一个不可用的按钮"/>
</View>
```



由于写了两个组件，所以要在外面包裹一层容器，这里我们用View。这两个按钮一个是正常状态，一个是不可用状态，我们通过控制disabled属性来控制是否可用(默认false)。按钮中的文字写在title里，color属性可以控制颜色。由于按钮需要和onPress一起用，所以会有一个这样的警告：

Warning: Failed prop type: The prop `onPress` is marked as required in `Button`, but its value is `undefined`....

不过暂时没什么问题，咱先不用管，以后用到的时候再写~

## Image组件

在以前，用React Native引用组件总是很坑，不过现在比以前好多了。由于作者君木有苹果电脑，所以不怎么清楚IOS版是怎么用的，我们就只说安卓的吧。图片的来源有两种，一种是本地图片，一种是网络图片，这两种图片在引入的时候稍稍有些差别，我们先看引入本地图片的例子：

```
<Image source={require('./images/avatar.png')} style={{ height:50, width:50 }} />
```

我们在项目根目录创建一个叫images的文件夹，并在里面放一个叫avatar的图片。然后require图片地址，使用图片组件要给它宽和高，不然无法正常显示。还有，如果图片没显示的话，记得重新执行

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

然后是网络图片：

```
<Image source={{uri: 'http://qiniu.com/xxx/png'}} style={{ height:50, width:50 }} />
```

在使用网络图片的时候，要将require改成uri，而且不写高度和宽度也没问题~

## 第三方组件

当你觉得官方组件有限，自己写又心好累，这时该怎么办呢？

当然是使用第三方组件啦~React Native社区越来越大，我们可以用的组件也越来越多，在这里，作者君推荐两个搜索组件的地方：

- js.coach(<https://js.coach/react-native>)
- GitHub(<https://github.com/>)

其实js.coach的组件来源也是GitHub 23333

# 是时候说说样式与布局了

## 本章提要

- 样式
- 布局
- Dimensions

## 样式

在前几章说了一些样式，是不是有点懵了呢？没关系，我们一起看看样式在React Native中的使用。

在React Native中，写样式和写CSS差不多，只不过是把`-`的连接方式换成了小驼峰式。又把`;`换成了`,`，有的时候要把值加上分号，比如这样：

```
background-color: green; ---> backgroundColor: 'green',
```

实际上，React Native的样式算是CSS的一个子集，当发现有些样式不能用的话不要惊讶，试着换用别的方式来实现就好~

作者君总结了一些常用的样式放到了附录里，可以作为写样式时的参考。

我们知道，代码总是会越写越多，样式也一样。所以为了效率，为了样式的整洁，我们写样式用`StyleSheet.create`比较好，不过只是写着玩玩的话，怎么写都好。下面我们来看看这种定义样式的方法

```
const styles = StyleSheet.create({
  content: {
    backgroundColor: '#fff',
    height: 50,
    width: 100
  },
  text: {
    color: 'blue',
    textAlign: 'center'
  }
})
```

然后在组件中这样使用即可：

```
<View style={styles.content}>
  <Text style={styles.text}>哈，我是小整洁</Text>
  ....
</View>
```

这样写后，你的代码可读性会提高不少，毕竟样式都像：

```
<Text style={{ fontStyle:'italic', fontSize: 20, ... }}>
```

就会变得乱糟糟一团团了。

## 布局

### Flexbox

在React Native中，你可以愉快的使用Flexbox布局，完全不用考虑兼容性（放心，绝对不需要考虑IE6），如果在浏览器端写过的话，在React Native上继续写这种布局会很是得心应手的。如果没接触过也没关系，这里会很详细的讲的（毕竟很重要）。

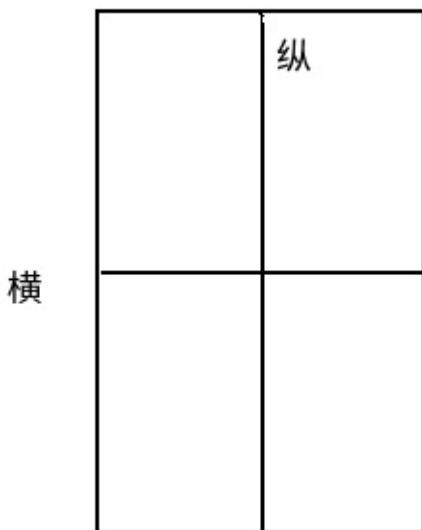
首先，我们先瞅瞅Flexbox布局需要用到什么样式：

```
flex, flexDirection, justifyContent, alignItems, alignSelf, flexWrap, flexShrink, flexGrow
```

嗯，先把这些列在这里，有个大概印象就成，我们一会儿再说。

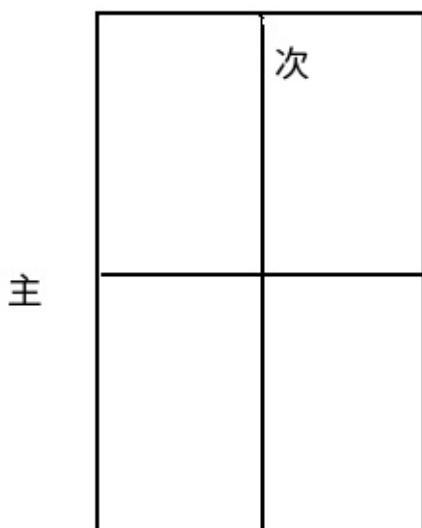
对于Flexbox来说，有一个很重要的概念——轴。理解了轴的概念，使用Flexbox也不会有什么问题了。

对于一个平面来说，它有两个轴，横轴和纵轴，就是下面这样：

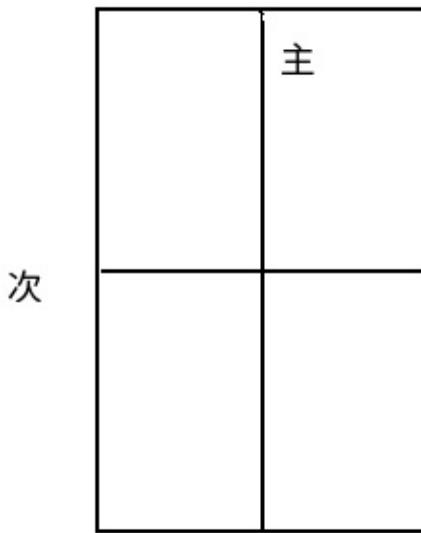


Flexbox用操作这两个轴的方式来完成布局，不过不叫横纵而是叫主次。

当横轴为主轴时，次轴是纵轴：



当纵轴为主轴时，次轴是横轴：



看完预备知识，我们可以看看Flexbox是怎么用的了。

首先，要有一个flex容器，如果不指定高度和宽度，它会占满能够占领的最大空间，如果给了高度和宽度，那么会占满你给它的空间。

我们用 `flex` 告诉React Native说这是一个使用了弹性布局的组件，就像这样：

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

`flex`后面的数字可以简单的理解为比例

不过呢，我们一般不会只写个 `flex: 1`，我们还得写一些别的东西，就像下面这样：

```
<View style={{ flex: 1, flexDirection: 'column', justifyContent:'center', alignItems:'flex-end' }}>
</View>
```

除了 `flex: 1`，剩下的其实就是用来操作横纵轴的。用 `flexDirection` 声明主轴是纵轴，次轴是横轴，用 `justifyContent` 声明元素处在主轴的中心位置，用 `alignItems` 声明元素处在次轴的末尾。

我们在 `View` 容器里填充一些元素进去，就可以很直观的看到效果：

```
<View style={{ flex: 1, flexDirection: 'column', justifyContent:'center', alignItems:'flex-end' }}>
  <Text style={{ backgroundColor:'blue'}} >1</Text>
  <Text style={{ backgroundColor:'red' }}>2</Text>
  <Text style={{ backgroundColor:'green' }}>3</Text>
</View>
```

效果是这样（虽然比较建议你自己敲一下代码，但是还是把图放在这里比较好）：

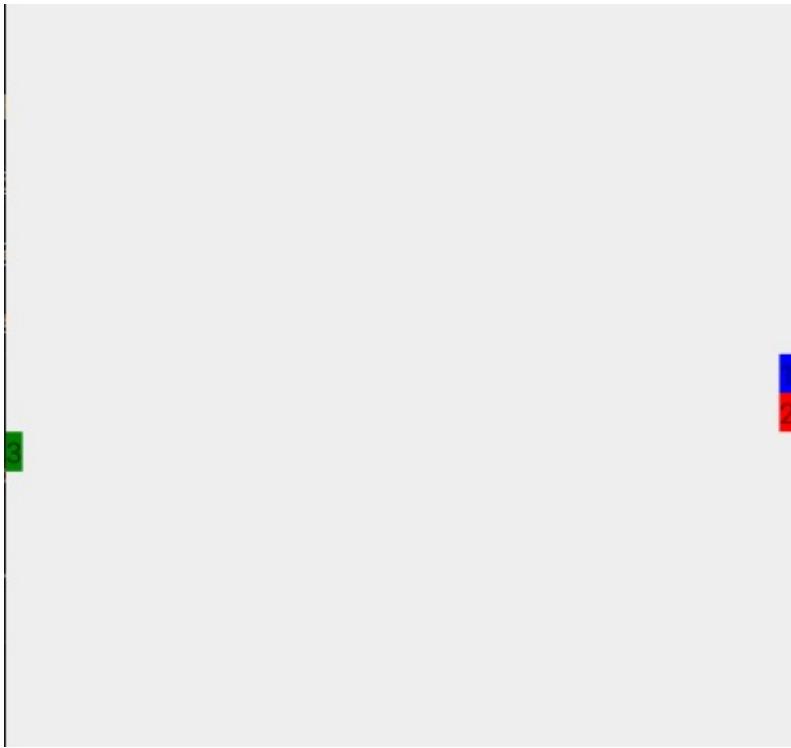


至于为啥这个图看起来有些大，是因为作者君把整个模拟器的屏幕截过来了，毕竟要突出元素在主轴中间的效果嘛。

这些还不是全部，毕竟不可能每个整体都是统一的，内部总是要有些不一样的。拿上面的那个例子来说：如果我不想让每个元素都在次轴的末尾，那该怎么办呢？很简单，这个时候我们就可以使用 `alignSelf` 了。`alignSelf` 可以使容器里的元素和父样式不一样，我们来简单的改造一下上面的例子：

```
<Text style={{ backgroundColor:'green', alignSelf:'flex-start' }}>3</Text>
```

这样就可以使第三个 `Text` 在前面了。就像这样：



我们再说回 `flex`。我们知道，`flex` 后面是接数字的，数字是这个容器所占的比例，也就是说，后面的数字越大，容器所占比例也就越大。

不过，如果只有 `flex: 1` 的话，它会自己占领所有地盘，谁叫没有其它的容器和它抢地盘呢。

## position

对于一个App来说，不可能只用到Flexbox，还得用到其他的布局方式。

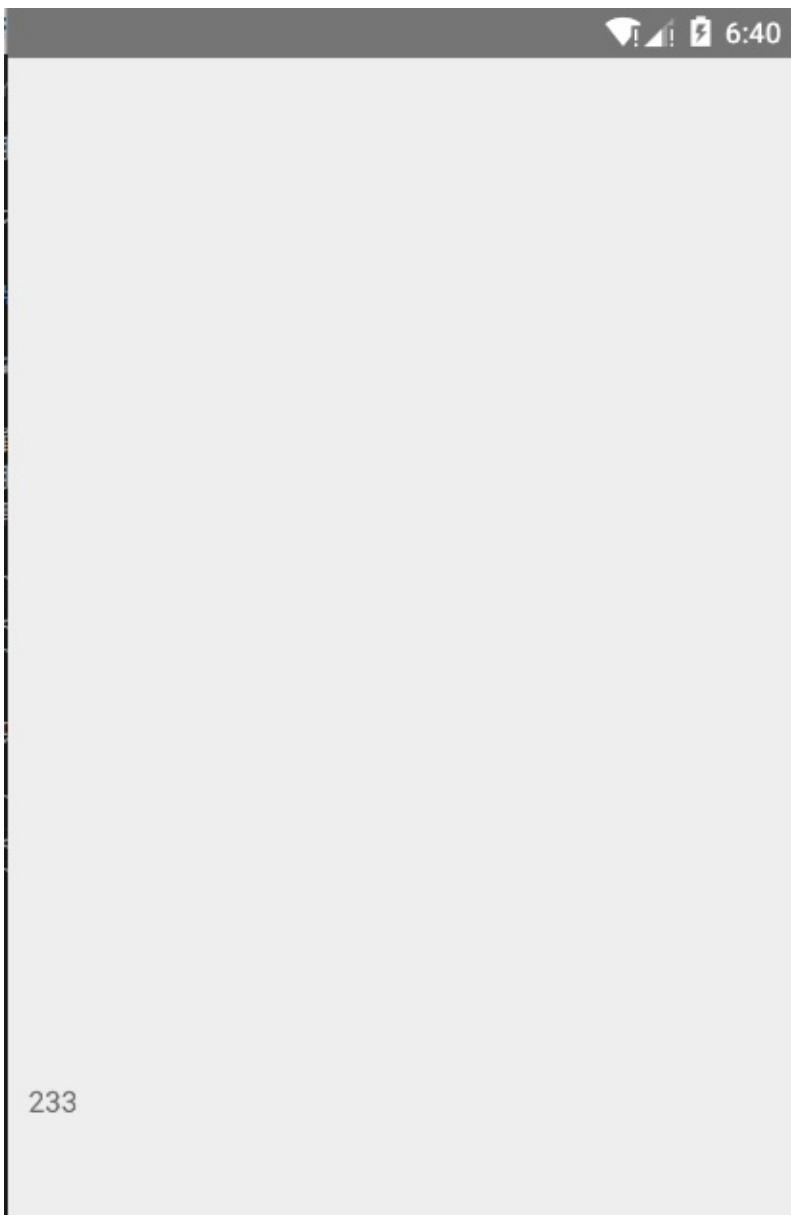
首先来看绝对布局`absolute`，绝对布局这名字听起来挺霸气的，实际上也确实霸气，规定元素在那里，那元素就得在那里待着。我们使用 `top, bottom, left, right` 来声明元素的上下左右距容器有多少距离，从而将元素定在那个位置。我们在用绝对布局的时候需要告诉元素它的布局方式是绝对布局，就像这样：

```
<Text style={{ position: 'absolute', width: 50, height: 50 }}></Text>
```

只不过我们得事先声明元素的宽度和高度，不然样式会变得很奇怪，然后来定位元素的位置：

```
<Text style={{ position: 'absolute', width:50, height: 50, bottom: 20, left:10 }}>233</Text>
```

就像这样：

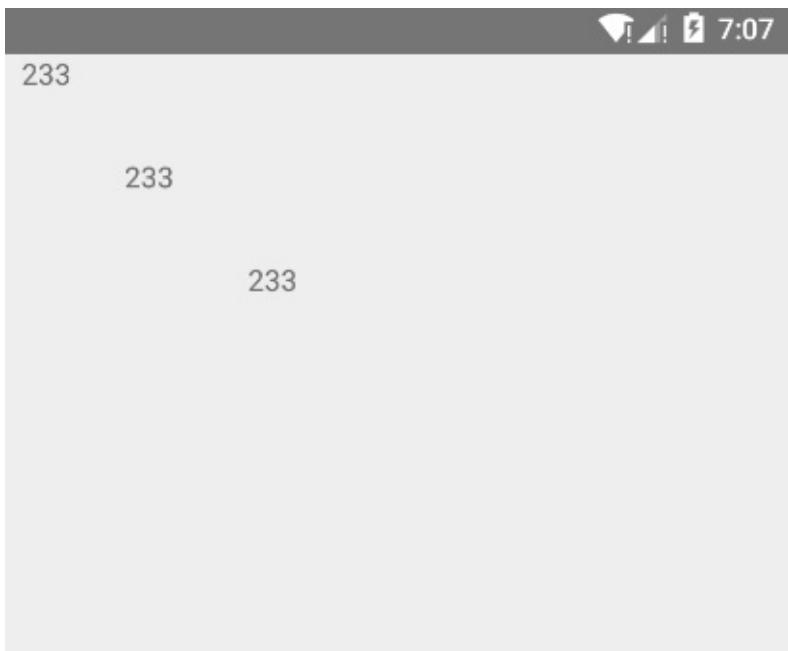


还有一点得注意，我们需要给父容器一个合适的高度和宽度，比如这样：

```
<View style={{ height:500,width:500 }}>
  <Text style={{ position: 'absolute', bottom: 20, left:10, width:50, height: 50 }}>233</Text>
  <Text style={{ position: 'absolute', bottom: 20, left:80, width:50, height: 50 }}>233</Text>
</View>
```

既然有绝对布局，就有相对布局，只不过可能会有些难用。你可以把相对布局理解成一个流，只不过 `bottom, right` 不能用。可以使用 `top, left` 更改元素在流中的上，左偏移大小（`top, left` 初始值为 0）。下面是个例子：

```
<View style={{height:500,width:500}}>
  <Text style={{ position: 'relative', left:10, width:50, height: 50 }}>233</Text>
  <Text style={{ position: 'relative', left:60, width:50, height: 50 }}>233</Text>
  <Text style={{ position: 'relative', left:120, width:50, height: 50 }}>233</Text>
</View>
```



## Dimensions

在React Native中，我们不能用 `width: 60%` 之类的方式来进行布局，那该怎么办嘞？嗯，我们可以换个方式，那就是 `Dimensions`，我们来请Dimensions出场：

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

然后获取宽度和高度，这里我们用解构赋值的方式：

```
const { width, height } = Dimensions.get("window");
```

然后就可以在样式中使用喽：

```
<View style={{ width: width * 0.5 ,height: height * 0.5 }}></View>
```

# 储存

## 本章提要

- AsyncStorage
- Realm 一个应用总是要存些东西的，我们要怎么做才能把数据存起来呢？我们这里有两个选择，一个是使用官方提供的 AsyncStorage 另一个是使用小型数据库。

当数据量不大，结构很简单的时候，我们来使用 AsyncStorage 这个 key-value 储存系统，当数据量较大的时候，我们就需要使用小型数据库了，这里我们会学习一个叫做 Realm 的数据库。

首先，我们来看看 AsyncStorage 吧~

## AsyncStorage

与数据打交道，免不了对数据进行增加，删除，更改和查找。我们来看看如何用 AsyncStorage 实现这些。

增加数据：

```
save(key, value) {
  AsyncStorage.setItem(key,value).then(
    (errs)=>{
      if (!errs) {
        //xxx
      }else {
        //xxx
      }
    })
}
```

在这里，我们使用了 promise，这样会使代码看起来舒服很多。虽然看起来很多，实际上，只要知道

```
AsyncStorage.setItem(key,value)
```

就可以了，我们用 `setItem` 来将 key 和 value 存储起来，看到这里想必你已经知道了为什么 AsyncStorage 是 key-value 系统了。

由于有了 key，所以我们可以用 key 进行查询：

```
get(key) {
  AsyncStorage.getItem(key)
    .then(
      (result)=> {
        if (!result) {
          //xxx
        }
        //返回结果
      })
    .catch((error)=> {
      //错误处理
    })
}
```

数据我们存完了，但是如果数据没用了怎么办？删掉呗。和查询数据很像，删除也需要用到 key：

```
del(key) {
  AsyncStorage.removeItem(key).
    then(
      ()=>{
        //删除成功
      })
    .catch((error)=> {
      //错误处理
    })
}
```

至于修改数据，其实写法和增加数据一样，当你需要修改一项数据的时候，直接传入 key 和一个新的值就可以了。

## Realm

前面的AsyncStorage只能存些小东西，数据量一大就开始捉襟见肘了，到了这种时候就该用Realm了。其实用SQLite也行，不过不怎么好用，索性就不用了。当初作者君在用Realm的时候版本号才是0.14转眼间就到1.0.0了，所以请放心的使用Realm吧。

Realm并不是React Native自己就有的，所以我们需要安装Realm：

```
npm install --save realm
```

安装好后大概是这样：

```
> realm@1.2.0 install /home/elven/Work/book/node_modules/realm
> node-pre-gyp install --fallback-to-build

[realm] Success: "/home/elven/Work/book/node_modules/realm/compiled/node-v48_linux_x64/realm.node" is installed via remote
book@0.0.1 /home/elven/Work/book
└── realm@1.2.0
    ├── extract-zip@1.6.0
    ├── concat-stream@1.5.0
    ├── readable-stream@2.0.6
    │   ├── isarray@1.0.0
    │   ├── typedarray@0.0.6
    │   ├── debug@0.7.4
    │   ├── mkdirp@0.5.0
    │   ├── minimist@0.0.8
    │   ├── yauzl@2.4.1
    │   └── fd-slicer@1.0.1
    ├── pend@1.2.0
    ├── ini@1.3.4
    ├── nan@2.6.0
    ├── node-pre-gyp@0.6.34
    │   ├── nopt@4.0.1
    │   ├── abbrev@1.1.0
    │   ├── osenv@0.1.4
    │   ├── npmlog@4.0.2
    │   └── console-control-strings@1.1.0
    ├── gauge@2.7.3
    └── aproba@1.1.1
        └── signal-exit@3.0.2
```

然后执行

```
react-native link realm
```

```
Scanning 584 folders for symlinks in /home/elven/Work/book/node_modules (16ms)
rnpm-install info Linking realm android dependency
rnpm-install info Android module realm has been successfully linked
rnpm-install info Linking realm ios dependency
rnpm-install warn ERRGROUP Group 'Frameworks' does not exist in your XCode project. We have created it automatically for you
rnpm-install info iOS module realm has been successfully linked
```

装完Realm之后得重新构建一下App：

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

*Tips*：由于网络的问题呢，可能会构建失败，多试几次就可以构建完成了

接下来测试一下Realm是不是安装成功了，先是引入Realm：

```
import Realm from 'realm';
```

下面是Realm官方文档的例子的：

```

export default class <project-name> extends Component {
  render() {
    let realm = new Realm({
      schema: [{name: 'Dog', properties: {name: 'string'}}]
    });

    realm.write(() => {
      realm.create('Dog', {name: 'Rex'});
    });
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          Count of Dogs in Realm: {realm.objects('Dog').length}
        </Text>
      </View>
    );
  }
}

```

<project-name> 是你的项目名称，别弄错了。然后刷新一下，界面上的数字就会加1。如果程序可以顺利运行，Realm就已经安装好了。

先简单的解释一下上面的代码，首先我们新建了一个叫做Dog的Schema，下面的 write 是写入数据，组件中的 realm.objects('Dog').length 会返回Dog中的数据个数。每次刷新App，都会写入一条数据，所以数字会加一。

有了一个简单的认识之后，让我们来详细的了解Realm吧！

## Schema

在使用Realm的时候离不开Schema，一个大些的App一般会用到多个Schema，所以在正式使用Realm的时候不能像上面的例子一样敷衍，我们需要进行一个简单的封装。新建一个叫Schema.js文件，它可以在app文件夹下（没有app文件夹就自己新建一个），

在文件头部引入Realm：

```
import Realm from 'realm';
```

然后创建一个叫Todo的class：

```
class Todo extends Realm.Object {}
```

不喜欢这个名字的话，你可以自己随便起一个名字。

然后来写Schema：

```

Todo.schema = {
  name:'Todo',
  primaryKey: 'id',
  properties: {
    title: {type: 'string'},
    id: {type: 'string', indexed: true},
  }
}

```

这里我们用id来作为主键，由于这个id具有唯一性，我们一会儿会写一个小程序来生成id。

然后导出模块：

```
export default new Realm({schema: [Todo]})
```

如果你需要多个Schema，在后面加上就好：

```
export default new Realm({schema: [Todo, Category, xxx, xxx]})
```

然后在你需要使用储存的地方引入Schema.js。

比如这样引入：

```
import realm from './Schema';
//注意文件位置
```

然后写一个生成id(为了不生成重复的id，我们使用guid，别管这是啥，只要知道这东西一般不会重复就好了)的函数：

```
const guid = () => {
  return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'
    .replace(/[xy]/g, (c) => {
      var r = Math.random() * 16 | 0, v = c == 'x' ? r : (r & 0x3 | 0x8);
      return v.toString(16);
    });
}
```

接下来，我们来看如何去用Realm（也就是增删改查）。

我们来写一个输入框来收集数据：

```
<TextInput
  style={{ height: 80, width: 200, marginBottom: -20 }}
  placeholder="输入"
  onChangeText={(title) => this.setState({title})}
/>
<Button title="添加" />
```



不出意外的话，下边会有一个黄色的提示（前边的章节有说过），先不要管它，我们一会儿解决。为了储存数据，我们需要把输入的文字放到state里面，上面的 `onChangeText` 就是干这个的，对了，别忘记加一个叫`title`的状态：

```
constructor(props){
  super(props);
  this.state = {
    text: null
  };
}
```

Ok，搞定这些后我们来写一个写入数据的方法，我们起名叫write，write应该写在constructor下边：

```
write(){
  realm.write(()=>{
    realm.create('Todo',{ title: this.state.title, id: guid() });
  });
}
```

写入数据需要用到 `realm.write()` 和 `realm.create()`，`Todo`是Schema，后边跟着的是要写入的数据，`title`是从 `state`中取出来的，然后我们调用 `guid()` 来生成一个不重复的id。接下来把这个方法绑到按钮上，实现点击添加数据：

```
<Button title="添加" onPress={ ()=> this.write() }/>
```

别忘记写this哦，write是内部的方法，不用this会引用不到的。

我们可以在下边写一个文本来显示Todo中数据的条数，就和上面的例子一样，我们在Button下面写个换行符：

```
<Text>{\'\n'}</Text>
```

然后把数据放到state里：

```
this.state = {
  text: null,
  data: realm.objects('Todo')
};
```

在换行符下面继续写：

```
<Text>{ this.state.data.length }</Text>
```

然后才是重点，Realm是可以使数据实时更新的，只要增加一个监听就好了，我们把这个监听写在构造里面：

```
this.state = {...}
realm.addListener('change', () => {
  this.forceUpdate()
});
```

然后输入一个数据，保存，下面的数字就可以实时的变化了~

然后我们来看查询是怎样的，界面的话把上面的代码复制过去，改几个字就好：

```
<TextInput
  style={{ height: 80, width: 200, marginBottom: -20 }}
  placeholder="输入"
  onChangeText={(find) => this.setState({find})}
/>
<Button title="查询" />
```

然后把find放到state里，正如我们之前做过的一样：

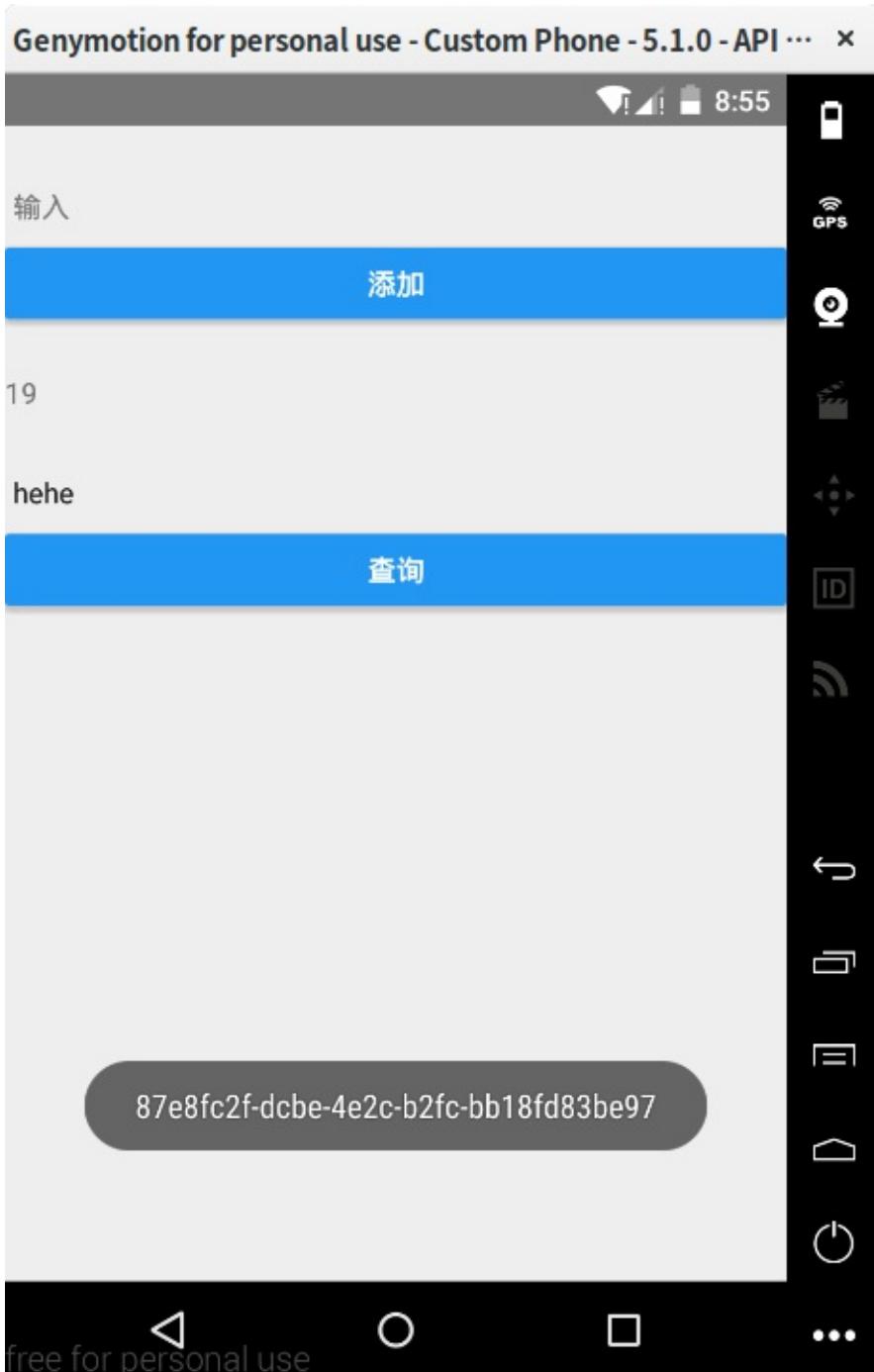
```
this.state = {  
  text: null,  
  find: null,  
  data: realm.objects('Todo')  
};
```

然后我们写一个叫find的方法,这个方法稍稍有些复杂：

```
find () {  
  realm.objects('Todo')  
    .filtered('title = ' + '"' + this.state.find + '"')  
    .forEach(result => ToastAndroid.show(result.id, ToastAndroid.SHORT));  
}
```

我们借助了一个叫做 `ToastAndroid` 的组件来实现显示查找结果的id，所以别忘记引入这个组件。`filtered` 是一个过滤器，我们在里面写查询代码，注意喽，查询的值应该是一个字符串所以在外面包裹了双引号。由于返回的结果稍稍有些复杂，所以我们用 `forEach` 来处理一下，使其获得id。

效果应该是这个样子的：



至于修改和删除就很简单了，所以这里只给出核心代码，作者君知道你很聪明，所以来想想具体怎么用把~

修改：

```
realm.write(() => {
    realm.create('Todo', { id: xxx, title: xxx }, true);
});
```

删除：

```
realm.write(() => {
    const todo = realm.objects('Todo.filtered("id = xxxxx")');
    realm.delete(todo);
});
```

小提示：

唔？ID真的没有用吗？具体可以起到什么作用呢？

想明白这两点，修改和删除就没问题了。另外，有没有发现修改，增加，删除都是在write中进行的呢？

## ListView与FlatList

当你需要给用户在一个页面展示一堆数据的时候，一个一个去渲染对性能来说真的不好，所以我们使用React Native内置的列表来渲染数据。在0.43版本之前，我们一直使用ListView来渲染数据，不过它的性能经常被人吐槽，社区里也不断提出解决方案。就连Realm都内置了一个性能更好些的ListView。不过自从官方发布了0.43版之后，这种局面应该会离我们远去了，因为官方给出了更好的方案FlatList。不过，ListView虽然老旧了一些，但依然还有学习的必要，所以我们从ListView开始学起

### ListView

对于ListView来说，我们需要一个数据源，这样ListView才有渲染视图的原料。这些数据可以放到一个数组里面(从Realm取出的数据可以直接用ListView渲染)，首先定义取值策略，我们一般采用这种方式(上下数据不相等时更新数据)：

```
const ds = new ListView.DataSource({ rowHasChanged: (r1, r2) => r1 !== r2 });
```

然后我们来创建一个数组来储存数据：

```
const data = [
  {
    id: 1,
    name: 'meow'
  },
  {
    id: 2,
    name: 'meow2'
  },
  {
    id: 3,
    name: 'meow3'
  },
  {
    id: 4,
    name: 'meow4'
  }
]
```

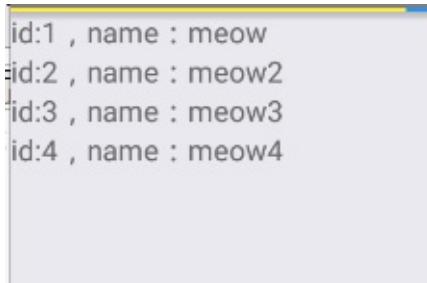
准备好后，我们开始使用ListView，别忘记引用它：

```
import {
  ...
  ListView
} from 'react-native';
```

然后随便找个视图丢个ListView进去，然后用刚才初始化好的ds来接受数据并使用renderRow渲染数据：

```
<ListView
  dataSource={ds.cloneWithRows(data)}
  renderRow={(rowData) => <Text>id:{rowData.id}, name:{rowData.name}</Text>}
/>
```

没加任何样式，不过数据愉快的渲染出来了：



我们可以用列表渲染文字，图片，在列表里嵌套组件等等，还可以像寻常组件一样为其添加各种样式。因为写样式就像寻常组件一样，所以这里就不再说了。

除了传递 `rowData`，还可以传递 `sectionID`, `rowID` 来获取section id和 row id，如果对这两个参数感兴趣就自己动手做下试验。

再提一个和性能有关的属性 `initialListSize`，这个属性可以指定首屏渲染多少数据，避免造成性能损耗。

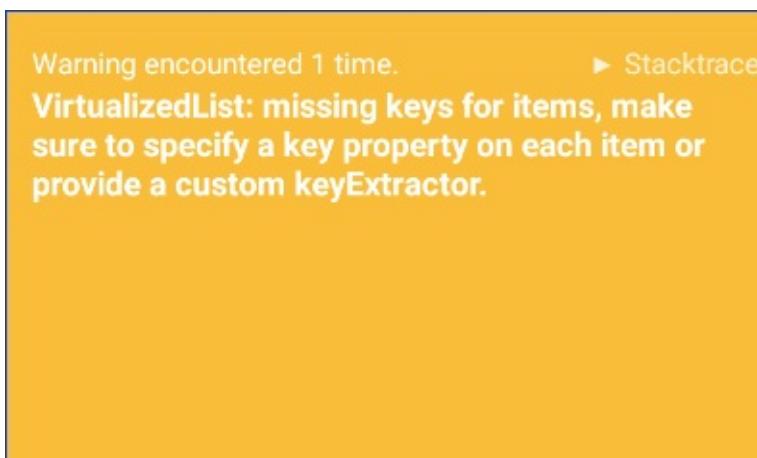
```
<ListView
  ...
  initialListSize={123}
/>
```

## FlatList

FlatList是一个高性能的列表，为什么要使用它我们在开篇介绍已经讲过，下面我们来看一个简单的例子：

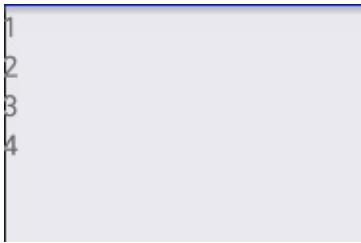
```
<FlatList
  data={data}
  renderItem={({ item }) => <Text>{item.id}</Text> }
/>
```

这里的data是在ListView例子中用过的data，不过呢，这段代码并不能运行，因为FlatList需要每行都有一个不同的key，所以我们需要炮制一下data。至于这个提示，先看前半句：



之前在创建data的时候我们用了`const`，为了后边能够修改data，我们将`const`改成`let`。然后写一小段代码为data添加key：

```
for (let i = 0; i < data.length; i++) {
  data[i].key = i
}
```



然后data就可以丢到FlatList中使用了(使用新组件别忘记引入)。另外，在现在的版本中data只能是一个普通数组，不可以是别的复杂数据结构(为了简单)。

接下来，我们来看FlatList支持的属性。

FlatList默认竖式布局，但是只要设置 `horizontal` 属性为true，列表就会变成横的：

```
horizontal={true}
```

刚才为data添加key的那一步其实并没有必要，因为有 `keyExtractor` 属性，我们可以通过这一属性来指定key，拿刚才的data来举例：

```
keyExtractor={(item, index) => item.id }
```

然后是三个比较像的组件 `ListHeaderComponent`，`ItemSeparatorComponent`，`ListFooterComponent`，在这三个属性中，我们向其中分别会放头部组件，分隔组件，尾部组件。举一个简单的例子：

```
ListHeaderComponent={()=>{return <Text>我是头部</Text>}}
ItemSeparatorComponent={()=>{return <Text>我是分隔组件</Text>}}
ListFooterComponent={()=>{return <Text>我是尾部</Text>}}
```



实际使用的时候记得把组件单独拿出来，这里这么写只是为了方便。FlatList为了优化内存占用并保证流畅滑动会在屏幕外异步绘制列表内容（看不到的地方用空白代替items），所以如果用户滑动过快就会看到空白内容，对于这个问题，我们可以使用 `getItemLayout` 属性来优化，不过你得知道内容的行高才能用：

```
//xxx是行高
getItemLayout={(data, index) =>
  { length: xxx, offset: xxx * index, index }
)}
```

介绍的最后一个组件在当前版本（0.43）有点小问题 `onEndReached` 有时候会不触发，所以只好期待下一个版本喽

`onEndReached` 是当列表滑到尾部会调用的一个属性（距离尾部不足这个 `onEndReachedThreshold` 的大小），我们可以用它来更新列表数据什么的：

```
onEndReached={()=>{
//更新数据
}}
```

与 `onEndReachedThreshold` 是个好基友，我们在里边放些数字，如果滑动列表正好触发 `onEndReached` 的话，我们需要设置 `onEndReachedThreshold` 的值为0

```
onEndReachedThreshold={0}
```

## 导航器

一个应用（普通的那种），总是有好多个界面，只有一个界面的应用怎么能留得住用户呢？（虽然有很多界面的也不一定留得住）所以说，做应用一般不会只做一个页面。我们知道，网页之间可以自由跳转，通过一个 `<a></a>` 就可以搞定了，但是开发应用不是写网页，React Native中也没有 `<a>`，那当我们需要做界面跳转的时候该怎么办呢？答案是使用 `Navigator` 组件。哈哈，逗你的，在0.42版本前使用 `Navigator` 来实现导航效果没问题，不过呢，自从0.43版本之后React Native将停止维护这个组件了，因为出现了更好用的 `React Navigation`。

首先我们来安装它：

```
npm install --save react-navigation
```

然后重新启动你的应用

作者君在安装这个库的时候，它的版本是beta7,有一个小bug需要处理一下，删掉 `node_modules/react-navigation/src/views/Header.js` 的第12行，

```
import ReactComponentWithPureRenderMixin from 'react/lib/ReactComponentWithPureRenderMixin';
```

不然直接红屏报错。

安装完毕后，我们来一个简单的例子，首先我们需要引入它：

```
import { StackNavigator } from 'react-navigation';
```

然后写两个组件，一个叫 `HelloScreen`，一个叫 `UserScreen`：

```
class HelloScreen extends Component {
  static navigationOptions = {
    title: 'Hello',
  };
  render() {
    const { navigate } = this.props.navigation;
    return (
      <View>
        <Text>{\'\n'}</Text>
        <Text>按下边的按钮</Text>
        <Button
          onPress={() => navigate('User')}
          title="Go"
        />
      </View>
    );
  }
}

class UserScreen extends Component {
  static navigationOptions = {
    title: '这里是~',
  };
  render() {
    return (
      <View>
        <Text>{\'\n'}</Text>
        <Text>嘿，猜猜这里是誰？</Text>
      </View>
    );
  }
}
```

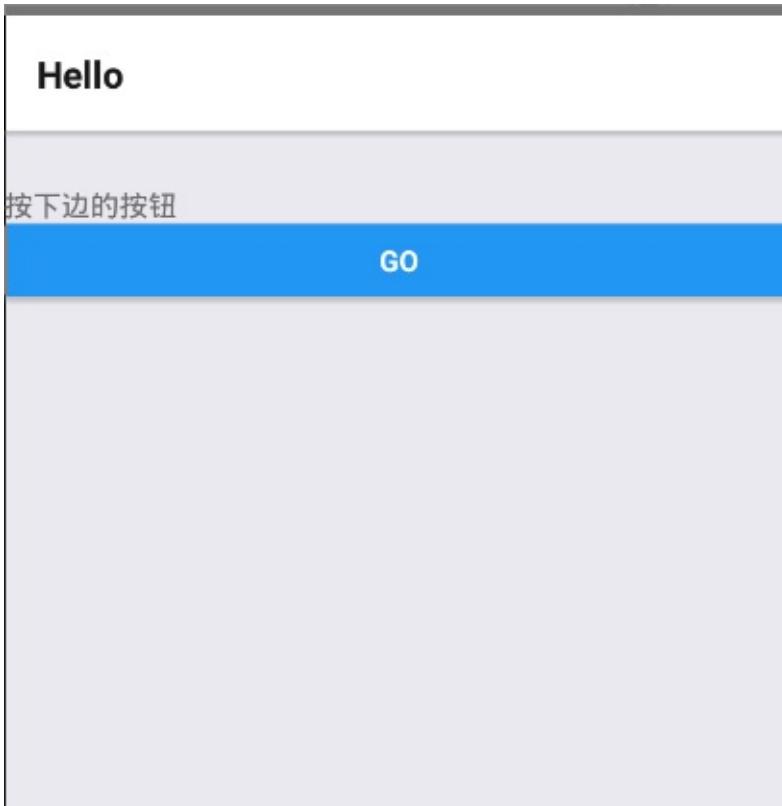
这两个组件和正常的组件差不多，不过是多了些东西罢了。其中，`navigationOptions` 数组里可以放些配置信息，一般用来配置Header部分。HelloScreen里的`const { navigate } = this.props.navigation`是为了可以在组件中使用`navigate`方法。`navigate(...)`理解成跳转到那个界面就好。不过，光有这两个组件可看不出来什么，我们还需要一个`StackNavigator`来使导航可以正常运作。（可以把这个当成一个特殊的组件，更好理解）：

```
const MyApp = StackNavigator({
  Home: { screen: HelloScreen },
  User: { screen: UserScreen },
});
```

这里的`Home`和`User`是路由别名，在使用`navigate`时需要用到。然后改一下注册的组件，就像下边那样：

```
AppRegistry.registerComponent('AppName', () => MyApp);
```

好啦，我们刷新一下应用：



然后点一下Go按钮：



就像这样，几行代码轻轻松松便实现了导航效果，甚至连头部都帮你做好了~

## 参数传递

很多时候，我们需要像要跳转的界面传递些参数，比如，某条数据的id，用户的名字。我们只需要把上边的例子小小的修改一下就可以传递参数了：

```
//修改navigate方法
navigate('User', { id: '233333333', name: 'nico' });
```

然后接受参数：

```
render() {
  const { params } = this.props.navigation.state;
  return (
    <View>
      <Text>{\'\n'}</Text>
      <Text>嘿，这里是{params.name}，id是：{params.id}</Text>
    </View>
  );
}
```

这样就实现参数的传递啦，快刷新一下看看效果吧~

## 定制头部

前边说过，`navigationOptions` 里面会放些配置信息，我们可以用这个来定制头部，比如说在右边加一个按钮：

```
static navigationOptions = {
  title: 'hahaha',
  headerRight: <Button title="你好" />,
}
```

理论上来说，它会在右边显示一个按钮，但是并不行，一脸茫然的作者君去翻了下GitHub，发现别人也遇到了这些问题，然后默默的更新了一下`react-navigation`，从beta7升到了beta9：

```
npm update react-navigation
```

然后刷新一下应用就可以发现按钮乖乖的显示在那里了：



除了按钮，你还可以丢个之类的过去，都是可以的。

## Tab Navigator

如果需要Tab导航，我们使用其内置的 `TabNavigator` 即可，用法和 `StackNavigator` 差不多，只要在最后使用 `TabNavigator` 即可就像这样：

```
const XXX = TabNavigator({
  .....
})
```

如果想要混用 `StackNavigator` 和 `TabNavigator` 则稍稍麻烦一点，首先，我们写两个新的组件：

```
class HiScreen extends Component {
  render() {
    return <Text>Hi</Text>
  }
}

class WorldScreen extends Component {
  render() {
    return <Text>World</Text>
  }
}
```

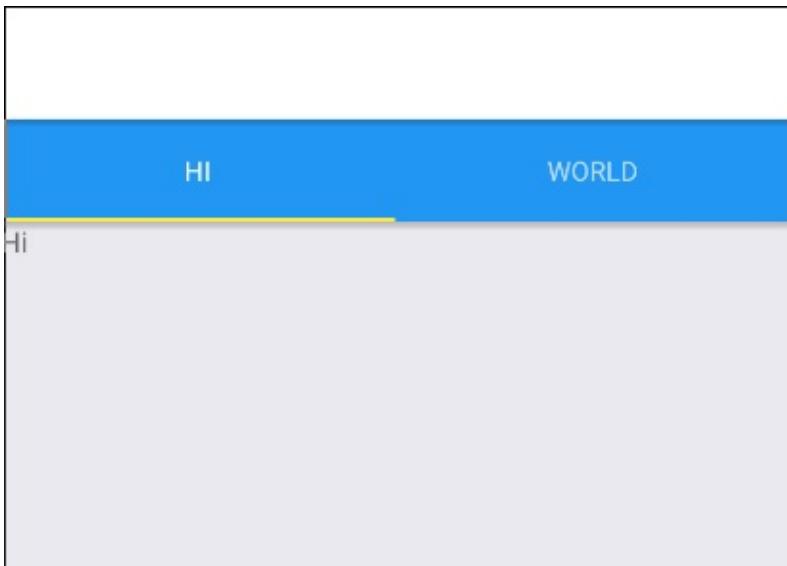
然后使用 `TabNavigator`：

```
const TabScreen = TabNavigator({ Hi: { screen: HiScreen }, World: { screen: WorldScreen }, });
```

这里的 `Hi`，`world` 会被作为选项卡 (Tab) 的名字，然后修改MyApp:

```
const MyApp = StackNavigator({
  Home: { screen: TabScreen },
  User: { screen: UserScreen },
});
```

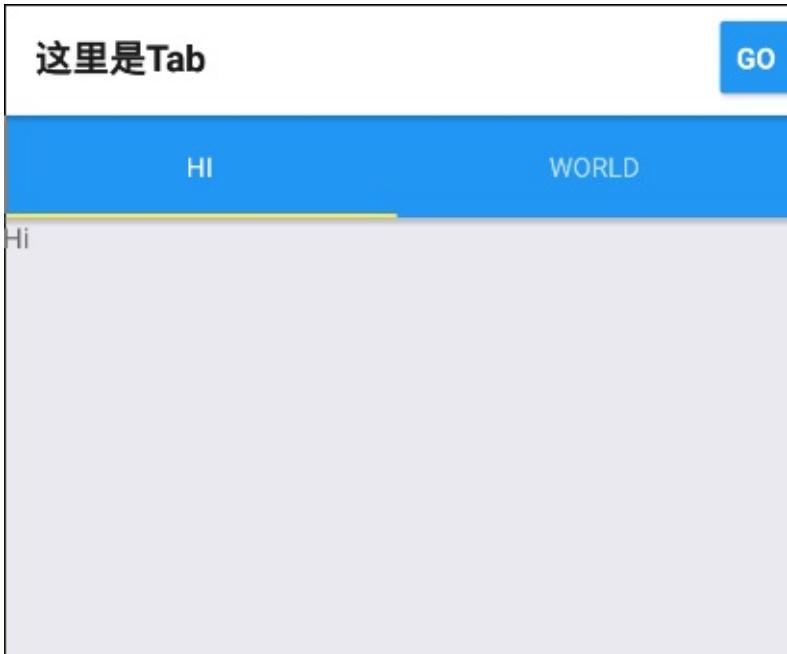
刷新一下应用，会发现公用的头部是空白的：



没关系，我们来添加一下：

```
TabScreen.navigationOptions = {  
  title: '这里是Tab'  
};
```

和别的导航组件，你也可以丢个按钮进去：



最后，我们来写一个可以跳转到UserScreen的按钮(别忘记navigate)：

```
//HiScreen  
const { navigate } = this.props.navigation;  
return (  
  <View>  
    <Text>Hi</Text>  
    <Button  
      onPress={() => navigate('User', { id: '233333333', name: 'nico' })}  
      title="Go"  
    />  
  </View>  
)
```

然后刷新一下看看是否成功吧~

## 网络

这一章呢，不会只讲网络哦~

### Mock Api

Mock Api是一项非常棒的服务，它可以帮助我们快速的模拟后端的REST API并且不需要你自己的服务器，甚至代码都不用写，它的网址是 <http://www.mockapi.io/>，可以注册也可以使用GitHub帐号登陆。

点击  来创建一个项目：



填写项目名称和Api前缀(随便写就行)：

 New Mock Project

**Mock Project Name**  
Ex: Todoapp, github, secretproject.

**React-Native**

**API Prefix (optional)**  
Add API prefix to all endpoints in this project (Ex: /api/v1)

**/api**

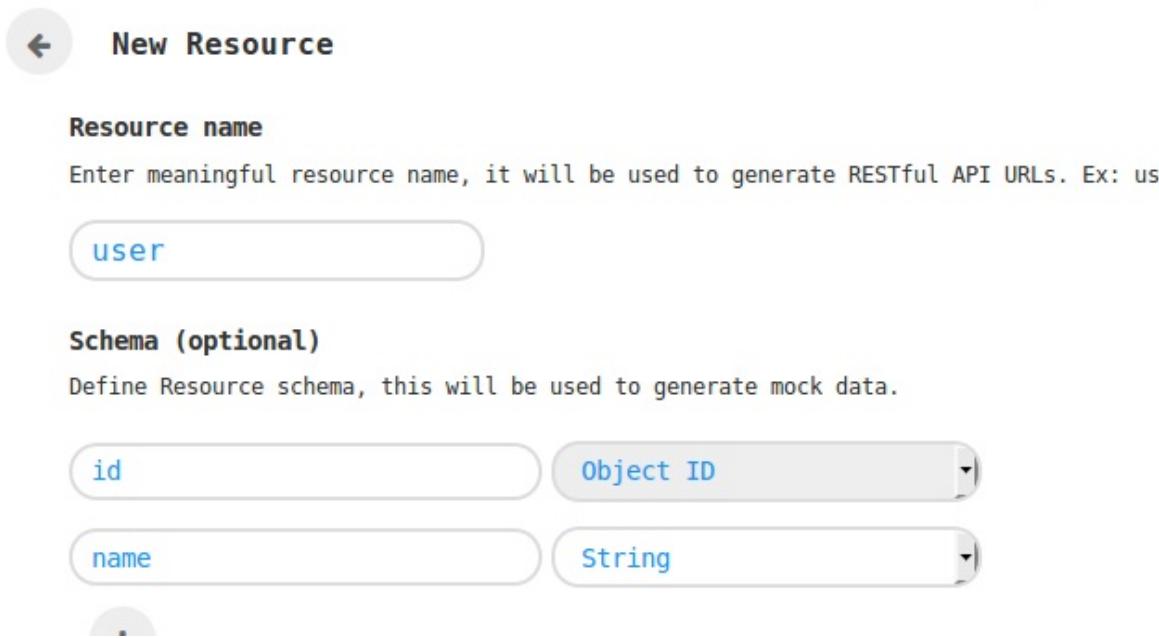
**Access**  
Others can search for this mock project and clone it.

 Public

接下来点击Create Resource来创建资源：



然后填写资源名称，以及修改Schema，为了方便，我们就留下id和name：



然后点击next按钮，这一步完了之后，我们就可以通过 `xxxxxx.mockapi.io/api/user` 使用REST Api了。

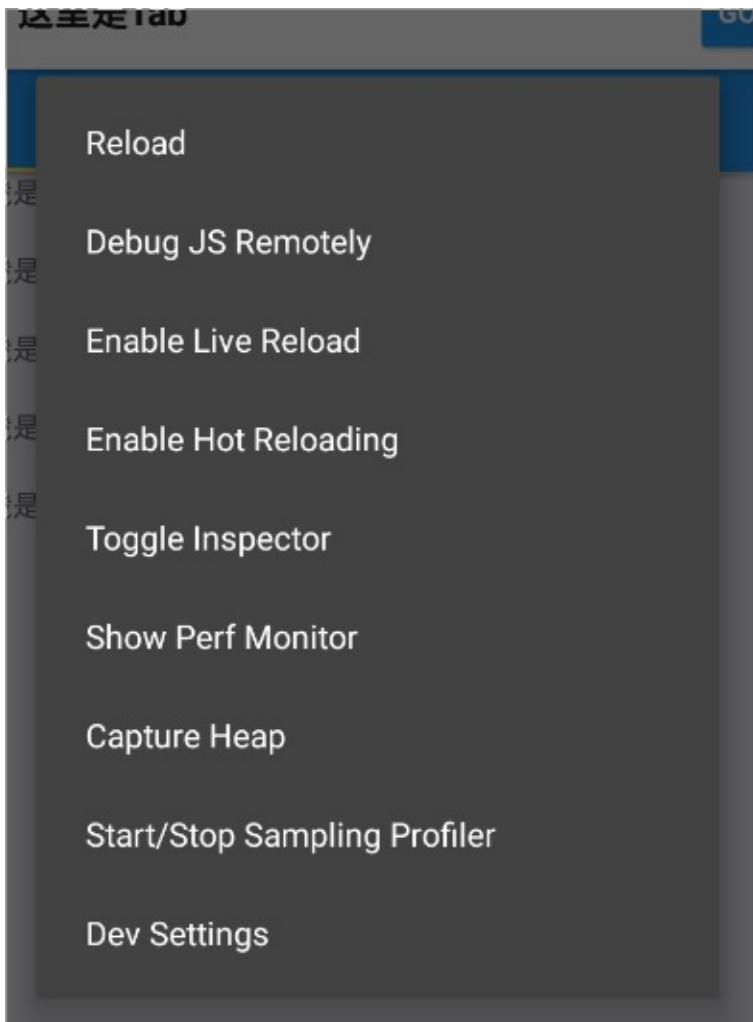
## Fetch

在React Native中，我们可以使用Fetch来进行网络请求，由于Fetch API天然支持Promise规范，所以我们的代码可以变得很优雅。

执行Get操作应该是最为简单的了，我们看下代码：

```
fetch('http://xxx.mockapi.io/api/user')
  .then((response) => {
    console.log(response);
  })
  .catch((err) => console.error(err));
```

这样便可以在控制台打印出user下的所有内容。你可能会想：喵喵喵？React Native还有控制台？当然有啦，不过我们得打开远程调试，首先打开菜单：



然后选择Debug Js Remotely，之后便会自动打开Chrome。自己打开Chrome的控制台便可以看到 console.log 的内容了。

实际应用中，我们会在组件渲染的时候把数据丢到状态或别的地方，然后使用这些数据(一般会用作列表渲染)。不管怎么说，不会直接console就是了。

然后是Post：

```
fetch('http://xxxx.mockapi.io/api/user', {
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    name: 'meow',
  })
})
.then((xxxx))
.catch((err) => console.error(err))
```

我们可以加个then来处理在Post过程中要做的事情，并用catch来获取异常。

至于Put和Delete参考Post就好啦~

## 打包应用

假设你的应用做完了，总不能还在后台开着packager就把应用发布出去吧。很明显，我们需要让应用可以离线，下边所要讲的就是一个让应用脱离packager的过程

### 生成签名

没签名直接打包应用是不行的，我们需要实现生成一个签名。在Linux下，我们执行这条命令：

```
keytool -genkey -v -keystore RN-key.keystore -alias RN-key-alias -keyalg RSA -keysize 2048 -validity 10000
```

然后会让你设置最少6位的密码，密码一定要记好，作者君的建议是两个密码最好不要设置成一样的~

至于Windows，我们得进到JDK目录下执行此命令，或者使用一些傻瓜式的签名生成软件。

### 修改gradle变量

将生成的RN-key.keystore放到android/app文件夹中，然后修改在android目录下的gradle.properties，

```
MYAPP_RELEASE_STORE_FILE=RN-key.keystore
MYAPP_RELEASE_KEY_ALIAS=RN-key-alias
MYAPP_RELEASE_STORE_PASSWORD=your password
MYAPP_RELEASE_KEY_PASSWORD=your password
```

### 添加配置

修改android/app目录下的build.gradle，在 defaultConfig 代码块下增加：

```
signingConfigs {
    release {
        storeFile file("RN-key.keystore")
        storePassword your password
        keyAlias RN-key-alias
        keyPassword your password
    }
}
```

在buildTypes中的release下添加：

```
signingConfig signingConfigs.release
```

整体就像这样：

```

signingConfigs {
    release {
        storeFile file("RN-key.keystore")
        storePassword "your password"
        keyAlias "RN-key-alias"
        keyPassword "your password"
    }
}
splits {
    abi {
        reset()
        enable enableSeparateBuildPerCPUArchitecture
        universalApk false // If true, also generate a universal APK
        include "armeabi-v7a", "x86"
    }
}
buildTypes {
    release {
        minifyEnabled enableProguardInReleaseBuilds
        proguardFiles getDefaultProguardFile("proguard-android.txt"), "proguard-rules.pro"
        signingConfig signingConfigs.release
    }
}

```

## 打包应用

进入项目所在目录，执行以下命令（packger应该处于开启状态）：

```
cd android && ./gradlew assembleRelease
```

Windows用户不用写`./`，漫长的等待过后，我们便可以在`android/app/build/outputs/apk`下找到我们的应用了，并且这个可以发布到各大应用市场。

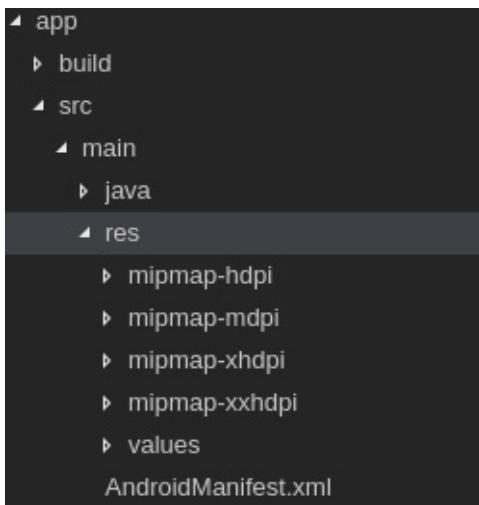
## 混淆及安全

为了安全（也为了缩小apk大小），我们需要启用代码混淆。进入`android/app`目录，然后修改`build.gradle`文件，将`def enableProguardInReleaseBuilds = false`改为`def enableProguardInReleaseBuilds = true`

如果不准备同时支持ARM和X86，那么将`enableSeparateBuildPerCPUArchitecture`改为`true`可以给安装包减少4mb大小~

## 修改icon

进入`android/app/src/main/res`目录，我们可以看到有这么几个子目录：



然后把这四个目录下的图片换成别的就可以啦，尺寸要对上哟~

## 修改应用名称

假设我们想将应用的名字换成 茄蒂 要怎么办呢？很简单，还是在res目录，我们进入values目录，修改strings.xml即可：

```
<resources>
  <string name="app_name">茄蒂</string>
</resources>
```

## 边框：

```
颜色
borderColor: color
borderBottomColor: color
borderLeftColor: color
borderTopColor: color
borderRightColor: color
圆角
borderRadius: number
borderBottomLeftRadius: number
borderBottomRightRadius: number
borderTopLeftRadius: number
borderTopRightRadius: number

宽度
borderWidth: number
borderBottomWidth: number
borderLeftWidth: number
borderRightWidth: number
borderTopWidth: number
样式
borderStyle: enum('solid', 'dotted', 'dashed')
```

## 边距：

```
内边距
padding: number
paddingBottom: number
paddingLeft: number
paddingRight: number
paddingTop: number
paddingHorizontal: number (左右内边距)
paddingVertical: number (上下内边距)

外边距
margin: number
marginBottom: number
marginLeft: number
marginRight: number
marginTop: number

marginHorizontal: number (左右内边距)
marginVertical: number (上下外边距)
```

## 文字

```
color: color
fontFamily: string
fontSize: number
fontWeight: enum('normal', 'italic')
fontStyle: enum('normal', 'bold', '100', '200', '300', '400', '500', '600', '700', '800', '900')
textDecorationColor: color (文本修饰颜色)
textDecorationLine: enum('none', 'underline', 'line-through', 'underline line-through')
textDecorationStyle: enum('solid', 'double', 'dashed', 'dotted')
letterSpacing: number
lineHeight: number
textAlign: enum('auto', 'left', 'right', 'center', 'justify')
```

## 布局

```
backgroundColor: color  
  
flex: number  
flexDirection: enum('row', 'row-reverse', 'column', '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')  
alignSelf: enum('auto', 'flex-start', 'flex-end', 'center', 'stretch')  
  
position: enum('absolute', 'relative')  
right: number  
left: number  
top: number  
bottom: number  
width: number  
height: number
```

## 图形变换

```
scaleX: 水平方向缩放  
scaleY: 垂直方向缩放  
rotation: 旋转  
translateX: 水平方向平移  
translateY: 水平方向平移
```

## 阴影

说个悲惨的故事，阴影只能在**ios**用

```
shadowColor: color  
shadowOffset: {width: number, height: number}  
shadowOpacity: number  
shadowRadius: number
```