实战

React Native 精解与实战

邱鹏源　编著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

React Native 精解与实战 / 邱鹏源编著 . —北京：机械工业出版社，2018.6
（实战）

ISBN 978-7-111-60385-6

I. R… II. 邱… III. 移动终端－应用程序－程序设计 IV. TN929.53

中国版本图书馆 CIP 数据核字（2018）第 146531 号

React Native 精解与实战

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：陈佳媛 责任校对：李秋荣
印 刷：北京市荣盛彩色印刷有限公司 版 次：2018 年 8 月第 1 版第 1 次印刷
开 本：186mm×240mm 1/16 印 张：22
书 号：ISBN 978-7-111-60385-6 定 价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

Preface 前 言

从 2015 年 React Native 框架发布开始，我就关注 React Native 框架的发展，在得知 React Native 框架将同时支持 iOS 平台与 Android 平台的部署后，我们就开始着手将之前的项目从混合开发的方式慢慢转移到使用 React Native 框架开发的方式上来。当时国内的 React Native 方面的资料非常少，很多难题的解决都需要查阅大量的国外文档，正是在这个摸索的过程中，加深了对 React Native 框架的理解。

后来，我们的很多 Web 项目都在使用 React 框架，React 正是 React Native 最底层的技术框架，同时也深深体会到只有理解底层架构，才能对于很多表象的难题快速定位并找到解决方案。

本书基于我多年写的技术博客以及使用 React Native 框架的实战经验，我认为底层的原理永远比一些组件的使用方法更重要，所以本书介绍了大量框架底层原理。通过阅读本书你会发现掌握框架后，组件和 API 的使用将变得非常简单，希望你在学习时能体会到这种触类旁通的感觉。

本书主要内容

本书分为两大部分，第Ⅰ部分“入门”包括第 1～9 章，介绍 React Native 框架的基本原理与使用；第Ⅱ部分“进阶”包括第 10～15 章，介绍 React Native 框架的高阶开发与 App 部署相关知识。

第 1 章介绍 React 与 React Native 框架产生的背景与原理，以及开发优势。

第 2 章介绍 Node.js 框架，并实战演示了 React Native 开发环境的安装与配置。

第 3 章介绍 React Native 框架的构成、工作原理、组件间通信以及生命周期，包括代码实战演示。

第 4 章介绍 React Native 页面布局开发使用的 CSS Flex，帮助读者掌握好 React Native 框架中元素布局的基本方法。

第 5 章介绍 React Native 框架下 iOS 平台与 Android 平台环境配置与代码调试的方法，并对 React Native 框架的调试工具以及借助 Chrome 进行远程调试的方法进行了实战讲解。

第 6 章介绍 React Native 框架中常用的组件，如 View、TabBar、NavigatorIOS、Image、Text、TextInput、WebView、ScrollView 等。同时介绍了 iOS 平台与 Android 平台的适配以及更适合的第三方组件。

第 7 章介绍 React Native 框架重点 API 的使用，包括提示框、App 运行状态、异步存储、相机与相册、地理位置信息、设备网络信息等 API，希望读者熟练掌握这些基础 API，进而能举一反三。

第 8 章介绍 React Native 框架下的网络请求以及列表数据的绑定，这是开发 App 需要使用的技术重点。

第 9 章介绍 React Native 开发生态下一些常用的第三方组件，通过代码实战的方式进行讲解，并介绍了如何快速地找到自己的项目需要使用的第三方组件。

第 10 章结合 iOS 与 Android 平台深入讲解了 React Native 框架的底层运行原理，并分别介绍了两个平台的部署与测试方法。

第 11 章介绍 React Native 框架下 iOS 平台的混合开发方法，通过混合开发，你可以在 React Native 框架下访问任何 iOS 原生平台。

第 12 章介绍 React Native 框架下 Android 平台的混合开发方法，以及案例分析。

第 13 章详细讲解 React Native 框架下 iOS 平台与 Android 平台的消息推送原理，并介绍两个平台的消息推送实战。

第 14 章介绍项目最终打包前 App 的图标与启动图的设置，并介绍了如何通过第三方工具快速生成这些相关资源。

第 15 章介绍 React Native 性能调优的方法与技巧，以便在 App 上架前测试以及后期 App 遇到性能问题时可以快速定位到问题所在。

本书附录简单剖析了 React Native 的源码，希望能帮助你深入研究 React Native 框架的源码，以便能探究其底层本质。同时也分享了一些学习 React Native 框架的相关资源。

本书的读者对象

- 各类移动 App 开发人员，学习 React Native 框架可以使你开发一套 React Native 源码同时部署到 iOS 平台与 Android 平台。
- 想进入移动 App 开发领域的初学者，React Native 框架比 Android 和 iOS 两个原生平台的技术门槛低很多，只需掌握 HTML、CSS、JavaScript 相关知识点就可以动手开发跨平台的移动 App。
- 已经在使用 React Native 框架开发移动 App 的开发人员，书中讲解了 React Native 框架的底层原理，以及与 iOS 平台、Android 平台的高阶混合开发部分，完全用代码进行讲解，学习起来更加直观。

本书配套源代码

本书的配套源代码都可以在 https://github.com/ParryQiu/ReactNative-Book-Demo 下载。

后续如果遇到 React Native 框架的大升级，我同样会在此代码库中相应地更新实战演示的代码。

致谢

首先感谢多年来共事过的同事们、领导们，多年来容忍我在一些技术问题上吹毛求疵，给了我很多好的建议和思路启发。还要感谢吴怡编辑，从约稿到审稿，都体现了她的认真态度和专业知识，她能一针见血地指出问题所在，促使我更加认真、专业地去编写此书。

在编写的过程中，虽然对于每一个知识点我都查阅了大量的相关文档，但是本书涉及海量的知识点，难免会有疏漏，恳请各位读者斧正。

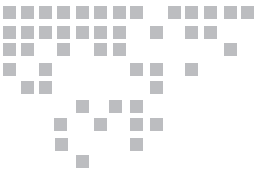
邱鹏源

2018 年 4 月

目　录 *Contents*

第 I 部分 *Part I*

入　门



Chapter 1 第 1 章

React 与 React Native 简介

这一章我们将对 React 与 React Native 的基本概念进行介绍。首先，详细介绍 React 产生的背景及 React 的框架，然后简单介绍 React 的底层实现原理，最后介绍 React Native 的基本概念。

你将对 React 与 React Native 框架的发展、框架之间的关系有一个基本的了解，具体的技术细节在后续的章节将有更加详细的讲解与实战解读。

1.1 React 简介

React 框架最早孵化于 Facebook 内部，Jordan Walke 是框架的创始人。React 作为内部使用的框架，在 2011 年的时候用于 Facebook 的新闻流（newsfeed），并于 2012 年用在了 Instagram 项目上。在 2013 年 5 月美国的 JSConf 大会上，Facebook 宣布 React 框架项目开源。

图 1-1 为 GitHub 上 React 的开源项目截图，地址为：https://github.com/facebook/react/。

React 框架产生的缘由是在当时的技术背景下，前端 MVC（Model-View-Controller）框架性能不能满足 Facebook 项目的性能需求以及扩展需求，所以 Jordan Walke 索性就自己着手开始写 React 框架，这种精神值得学习。

在当时 Facebook 内部极其复杂的项目中，面临的一个问题是，在 MVC 架构

的项目中当 Model 和 View 有数据流动时，可能会出现双向的数据流动，那么项目的调试以及维护将变得异常复杂。

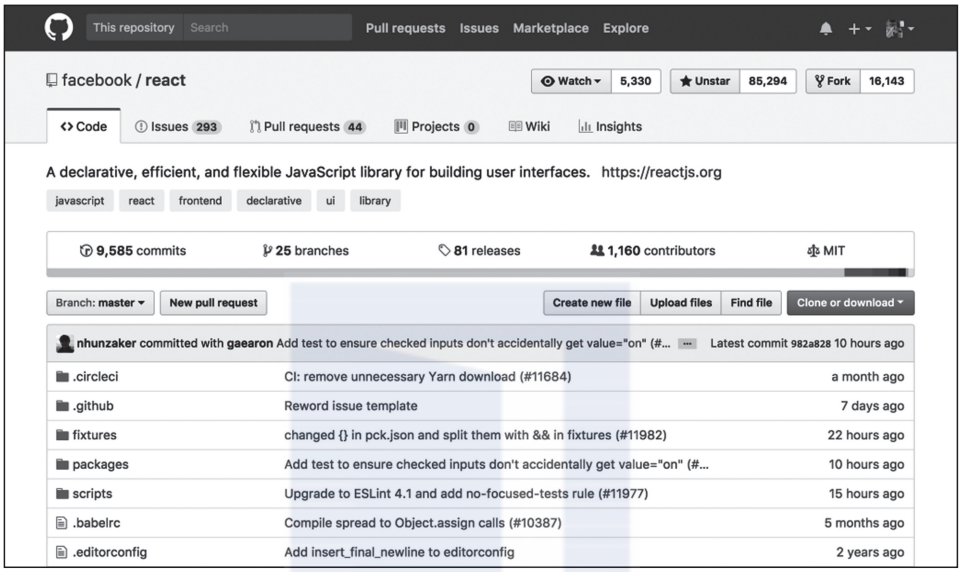


图 1-1　GitHub 上的 React 项目

React 官方也说自己不是一个 MVC 框架（https://reactjs.org/blog/2013/06/05/why-react.html），或者说 React 只专注于 MVC 框架设计模式中的 View 层面的实现。

为了大大减少传统前端直接操作 DOM 的昂贵花费，React 使用 Virtual DOM（虚拟 DOM）进行 DOM 的更新。

图 1-2 为 React 框架的基本结构，清晰明了地描述出了 React 底层与前端浏览器的沟通机制。

React 的组件是用户界面的最小元素，与外界的所有交互都通过 state 和 props 进行传递。通过这样的组件封装设计，使用声明式的编程方式，使得 React 的逻辑足够简化，并可以通过模块化开发逐步构建出项目的整体 UI。

React 框架中还有一个重要的概念是单向数据流，所有的数据流从父节点传递到子节点。假设父节点数据通过 props 传递到子节点，如果相对父节点（或者说相对顶层）传递的 props 值改变了，那么其所有的子节点（默认在没有使用

shouldComponentUpdate 进行优化的情况下）都会进行重新渲染，这样的设计使得组件足够扁平并且也便于维护。

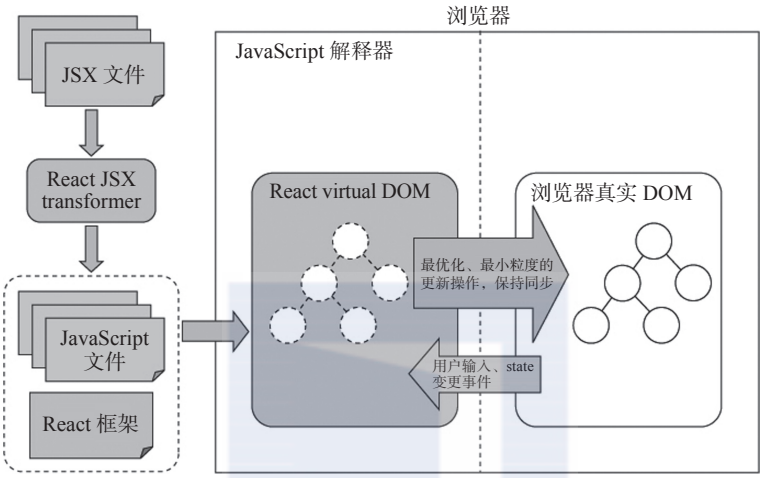


图 1-2 React 框架结构

以下的示例代码演示了 React 框架基本组件定义以及单向数据流的传递。

> 完整代码在本书配套源码的 01-01-02 文件夹。

我们在 index.js 文件中定义 React 项目的入口，render 函数中使用了子组件 BodyIndex，并通过 props 传递了两个参数，id 和 name，用于从父组件向子组件传递参数，这也是 React 框架中数据流的传递方式：

```
/**
 * 章节: 01-01-02
 * index.js 定义了 React 项目的入口
 * FilePath: /01-01-02/index.js
 * @Parry
 */

var React = require('react');
var ReactDOM = require('react-dom');
import BodyIndex from './components/bodyindex';
class Index extends React.Component {

  //生命周期函数 componentWillMount，组件即将加载
  componentWillMount(){
```

```
    console.log("Index - componentWillMount");
  }

  //生命周期函数 componentDidMount，组件加载完毕
  componentDidMount(){
    console.log("Index - componentDidMount");
  }

  //页面表现组件渲染
  render() {
    return (
      <div>
        <BodyIndex id={1234567890} name={"IndexPage"}/>
      </div>
    );
  }
}

ReactDOM.render(<Index/>, document.getElementById('example'));
```

在子组件 BodyIndex 中定义了 state 值，并通过 setTimeout 函数在页面加载 5 秒后进行 state 值的修改。页面表现层代码演示了如何读取自身的 state 值以及读取父组件传递过来的 props 值：

```
/**
 * 章节: 01-01-02
 * bodyindex.js 定义了一个名为 BodyIndex 的子组件
 * FilePath: /01-01-02/bodyindex.js
 * @Parry
 */

import React from 'react';
export default class BodyIndex extends React.Component {
  constructor() {
    super();
    this.state = {
      username: "Parry"
    };
  }

  render() {
    setTimeout(() => {
      //5秒后更改一下 state
      this.setState({username: "React"});
```

```
    }, 5000);

    return (
      <div>

        <h1>子组件页面</h1>

        <h2>当前组件自身的 state</h2>
        <p>username: {this.state.username}</p>

        <h2>父组件传递过来的参数</h2>
        <p>id: {this.props.id}</p>
        <p>name: {this.props.name}</p>

      </div>
    )
  }
}
```

项目的 package.json 文件配置和使用的相关框架版本如下所示：

```
{
  "name": "01-01-02",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "babel-preset-es2015": "^6.14.0",
    "babel-preset-react": "^6.11.1",
    "babelify": "^7.3.0",
    "react": "^15.3.2",
    "react-dom": "^15.3.2",
    "webpack": "^1.13.2",
    "webpack-dev-server": "^1.16.1"
  }
}
```

在命令行执行 webpack-dev-server 命令后，浏览器中的运行结果如图 1-3 所示，并且在 5 秒后子组件的 state 定义的 username 值由 Parry 变成了 React。具体的配置

方法及其意义将在后续章节讲解。

你可以直接在本地编写代码运行测试或直接下载本书配套源码运行，运行后，注意此 state 页面值更新的部分，整个页面没有进行任何的重新刷新加载，而只是进行了局部的更新，其原理详见下一节。

图 1-3　代码在浏览器中的执行结果

React 框架底层的核心为 Virtual DOM，也就是虚拟 DOM。本节将介绍它的底层特性，只有理解了 React 框架底层的本质，才能更好地帮助你理解 React 框架的前端表现，并为后续章节讨论 React Native 框架的性能优化进行一定的知识储备。

传统的 HTML 页面需要更新页面元素，或者说需要更新页面时，都是将整个页面重新加载实现重绘，执行这样的操作不管是从服务器代价还是从用户体验上来说，“代价”都是非常昂贵的。后来，有了 AJAX（Asynchronous JavaScript And XML）这样的局部更新技术，实现了页面局部组件的异步更新，不过 AJAX 在代码的编写、维护、性能以及更新粒度的控制上还是不太完美。

文档对象模型（Document Object Model，DOM），是 W3C 组织推荐的处理可扩展标志语言的标准编程接口。在 HTML 网页上，将构成页面（或文档）的对象元素组织在一个树形结构中，用来表示文档中对象的标准模型就称为 DOM。

React 在框架底层设计了一个虚拟 DOM，此虚拟 DOM 与页面上的真实 DOM 相互映射，当业务逻辑修改了 React 组件中的 state 部分，如上例中，子组件的 state 值，username 由 Parry 修改成了 React，React 框架底层的 diff 算法会通过比较虚拟 DOM 与真实 DOM 的差异，找出哪些部分被修改了最终只更新真实 DOM 与虚拟 DOM 差异的部分。此计算过程是在内存中进行的，所以 React 在前端中的高

性能表现正是来自于其底层的优良设计。

图 1-4 展示了 React 中的虚拟 DOM 与页面真实 DOM 之间的关系，其间的差异通过 React 框架底层的 diff 算法获取。

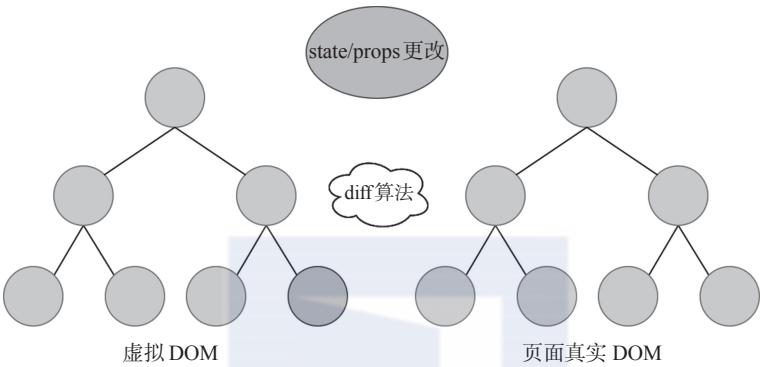


图 1-4 React 虚拟 DOM 与页面真实 DOM

要更加深入地了解 React 在源码级别的实现原理，可以参考我博客里从 React 的源码角度对其底层批量更新 state 策略的分析文章：

- 深入理解 React JS 中的 setState

http://blog.parryqiu.com/2017/12/19/react_set_state_asynchronously/

- 从源码的角度再看 React JS 中的 setState

http://blog.parryqiu.com/2017/12/29/react-state-in-sourcecode/

- 从源码的角度看 React JS 中批量更新 State 的策略（上）

http://blog.parryqiu.com/2018/01/04/2018-01-04/

- 从源码的角度看 React JS 中批量更新 State 的策略（下）

http://blog.parryqiu.com/2018/01/08/2018-01-08/

通过以上对 React 框架的简介、代码演示以及底层原理的剖析得知，React 最大的优势在于更新页面 DOM 时，对比于之前的前端更新方案，效率会大大提高。其实 React 并不会在 state 更改的第一时间就去执行 diff 算法并立即更新页面 DOM，而是将多次操作汇聚成一次批量操作，这样再次大大提升了页面更新重绘的效率。

使用 React 框架开发，我们不会通过 JavaScript 代码直接操作前端真实 DOM，而是完全通过 state 以及 props 的变更引起页面 DOM 的变更，相对于 jQuery 等框

架那样进行大量的 DOM 查找与操作要简单、高效得多。

React 框架在开源生态下，已经有大量的相关开源框架与组件可供使用，非常适合项目的快速开发。

1.2　React Native 简介

Facebook 曾致力于使用 HTML 5 进行移动端的开发，最终发现与原生的 App 相比，体验上还是有非常大的差距，并且这种差距越来越大，特别是在性能方面。最终，Facebook 放弃了 HTML 5 的技术路线，于 2015 年 3 月正式发布了 React Native 框架，此框架专注于移动端 App 的开发。

在最初发布的版本中，React Native 框架只用于开发 iOS 平台的 App，2015 年 9 月，Facebook 发布了支持 Android 平台的 React Native 框架。至此，React Native 框架真正实现了跨平台的移动 App 开发，此举简直就是移动 App 开发人员的福音。

图 1-5 为 GitHub 上 React Native 的开源项目，地址为 https://github.com/facebook/react-native/。

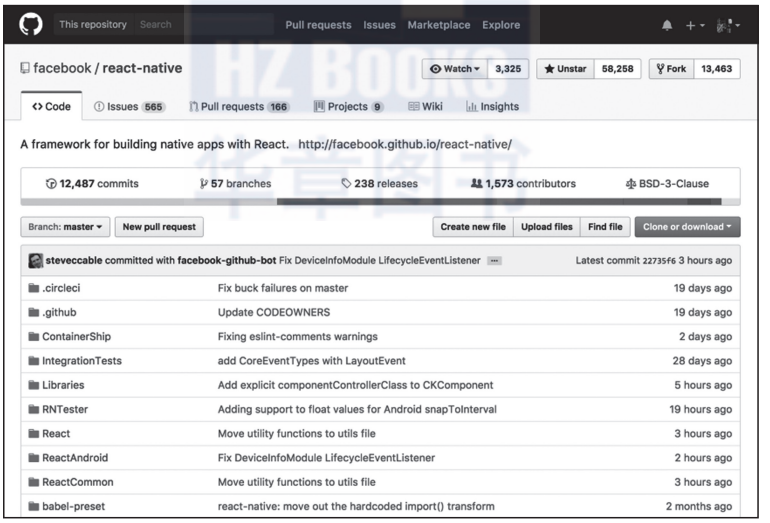


图 1-5　GitHub 上 React Native 开源项目

React Native 框架在 React 框架的基础上，底层通过对 iOS 平台与 Android 平台原生代码的封装与调用，结合前台的 JavaScript 代码，我们就可以编写出调用

iOS 平台与 Android 平台原生代码的 App，这样编写的 App 的性能远远优于使用 HTML 5 开发的 App 性能，因为 HTML 5 开发的 App 只是在 HTML 5 外部包裹上一个程序外壳后在移动平台上运行，在性能与功能上都不能达到 React Native 框架的水准。

React Native 框架提供了原生组件与底层 API 供开发者使用，这些自带的组件与 API 足以满足移动端 App 的开发需求，后续章节会详细讲解这些组件与 API 的概念与使用实战演示。

React Native 框架还提供了与 iOS 平台、Android 平台混合开发的接口，让开发者可以在 React Native 中调用 iOS 平台与 Android 平台中任意的原生 API 与代码，让可以在原生平台实现的任何功能都可以在 React Native 框架中得以实现。

在使用 React Native 框架开发移动平台 App 的过程中，我们可以直接使用 CSS 进行页面元素的布局，这之前是 iOS 与 Android 原生移动平台开发者简直不可想象的事情。

开发人员在具备了 React 框架基础知识后，可以更加快速地进行 React Native 框架的学习与开发。图 1-6 为 React Native 官网截图，代码展示了我们只需要使用类似 HTML 5（JSX）的代码就可以进行跨平台的移动 App 开发。

```
import React, { Component } from 'react';
import { Text, View } from 'react-native';

class WhyReactNativeIsSoGreat extends Component {
  render() {
    return (
      <View>
        <Text>
          If you like React on the web, you'll like React Native.
        </Text>
        <Text>
          You just use native components like 'View' and 'Text',
          instead of web components like 'div' and 'span'.
        </Text>
      </View>
    );
  }
}
```

图 1-6 React Native 演示代码

React Native 有以下优势：

- 底层采用 React 框架，减少了我们学习与开发的成本，React Native 框架可以让你真正跨越移动开发的鸿沟，不需要分开学习 iOS 平台与 Android 平台

的特定语法、页面布局以及各平台的特别处理技巧，使用一套 React Native 代码的部署就可以覆盖多个移动平台。

- React Native 性能优化很好，完全可以避开之前使用 HTML 5 开发移动 App 的性能障碍。
- React Native 框架的 JavaScript Core 底层，可以让 App 轻松实现更新操作，基本上更新一下 JavaScript 文件，整个 App 就完成了更新，非常适合用来开发 App 的热更新。
- React Native 框架同时也使得 App 的开发调试变得异常简单，不需要像之前在多个平台、多个语言、多个工具之间跳来跳去，React Native 开发的 App 在模拟器或真机中，只需要像刷新浏览器一样就可以即时查看到代码修改后的效果，并且还可以在 Chrome 浏览器中查看控制台输出、加断点、单步调试等，整个过程完全就是 JavaScript 开发调试的体验，非常畅快。

图 1-7 为使用 React Native 框架开发时，在 iOS 系统下的开发调试选项截图，功能非常强大、使用非常方便。Android 平台提供了同样的调试选项。

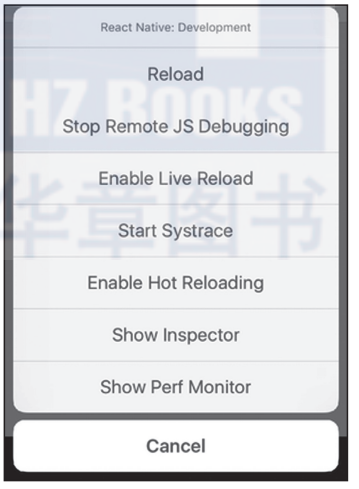


图 1-7　React Native 开发调试选项

1.3　React Native 前置知识点

在正式学习 React Native 框架前，我们梳理一下需要具备的基本知识，因为

React Native 毕竟是在多平台的移动 App 开发，涉及知识点比较多，而且底层的 React 框架有别于目前既有的一些前端框架知识。

- 掌握 HTML 5 的基本知识；
- 掌握 JavaScript 的基础知识，如果有 React 的基础知识学习起来会更加轻松；
- 掌握 CSS 布局的基本知识，在 React Native 中会使用 CSS 直接进行页面元素的布局与样式控制；
- 接触过移动端的开发更好，项目的后期会涉及两个平台的 App 打包、部署与上架，不过这些知识点后续章节都会讲解；
- Node.js 以及 npm 包管理的知识，这部分后续章节同样会有详细的讲解。

第 2 章 Chapter 2

Node.js 简介与开发环境配置

本章将对开发过程中依赖的基础框架 Node.js 进行介绍，并深入讲解为什么使用此框架。同时我们将开始配置 React Native 的开发环境，并对代码编辑器 Visual Studio Code 以及相关高效开发插件做详细的介绍。

2.1 Node.js 与 npm 简介

本节对 Node.js、npm 进行了介绍，以及对需要使用到 Node.js 框架的原因进行了介绍。

1. Node.js 简介

关于 Node.js，官网（https://nodejs.org）给出的定义如下。

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型，使其轻量又高效。Node.js 的包管理器 npm，是全球最大的开源库生态系统之一。

虽然只有几句话，但是已经很清楚地描述了 Node.js 以及 npm 的概念。

Node.js 本身并不是一个新的开发语言，也不是一个 JavaScript 框架，而是一个 JavaScript 运行时，底层为 Google Chrome V8 引擎，并在此基础上进行了封装，可用于创建快速、高效、可扩展的网络应用。Node.js 采用事件驱动与非阻塞 I/O 模型，以使得 Node.js 轻量并高效。

图 2-1 为 Node.js 的架构图，可以看到底层使用 C/C++ 编写。

- Chrome 的 V8 引擎是用 C++ 开发的，负责将 JavaScript 代码转换成机器码，所以引擎的整体执行效率非常高。Google Chrome 浏览器的底层使用的就是 V8 引擎；
- 线程池是完全使用 C 语言实现、全特性的异步 I/O 库 libeio，用于执行异步的输入 / 输出、文件描述符、数据处理、sockets 等；
- libev 是在 Node.js 内部运行的 event loop，最早用于类 Linux 系统。event loop 是一个让多线程执行更加高效的程序结构；
- 这些框架实现的语言不一样，有 C，有 C++，还有 JavaScript，那么将这些代码整合在一起就是 Node bindings 要做的事情；
- 最上层是使用 Node.js 开发时接触到的应用层，Node.js 提供了一系列标准的 JavaScript 类库供开发者使用。

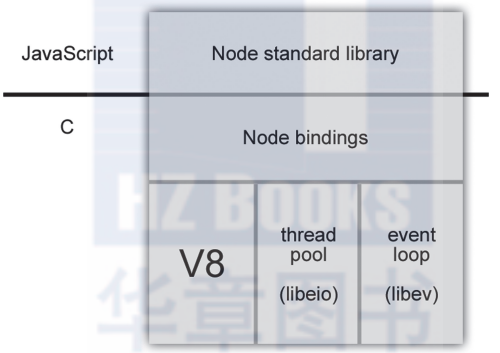


图 2-1 Node.js 架构图

2. npm 简介

npm 是 Node.js 的包生态系统，是最大的开源生态系统。可以理解为基于 Node.js 框架，全世界的开发者提交了各种各样的功能类库到 npm 中，其他开发者在开发过程中需要使用的大部分功能都可以在 npm 中找到已存在的库，完全不需要自己“造轮子”。

截至 2018 年 3 月 npm 官网（https://www.npmjs.com/）上已有 60 多万个包，是一个非常大的宝库，你可以下载、使用、学习各种类库，当然，也可以贡献自己的类库到 npm 中供其他开发者使用。

可以在 npm 中直接搜索你在开发过程中需要使用到的任何功能库，假设你需要一些关于 cookie 处理的 JavaScript 类库，图 2-2 就是在 npm 中搜索 cookie 相关类库的结果。使用 npm 库是你使用 React Native 开发 App 肯定会接触到的一个过程。

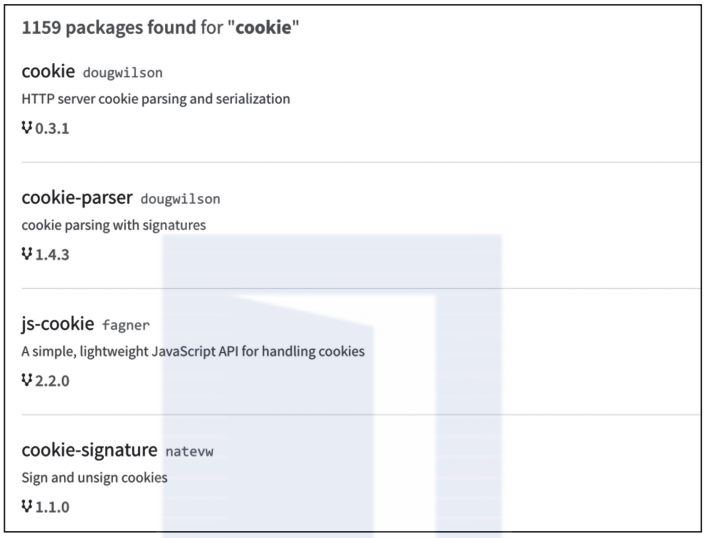


图 2-2　npm 中搜索类库

3. React Native 与 Node.js 的关系

Node.js 提供了很多的系统级的如文件操作、网络编程等特性，并且是事件驱动、异步编程的。React 构建于 Node.js 之上，其实本质上 React 也是 npm 包中的一个，React Native 也是 npm 包之一，只不过是功能非常强大的包而已。所以整个的框架都构建于 Node.js 之上，并且 Node.js 还提供了海量的类库，在这个完整的生态系统下开发，过程将变得更加高效，在后续的章节中将会慢慢体会到此生态系统的价值。

2.2　React Native 开发环境配置

本节我们将开始配置 React Native 的开发环境，包括 Node.js 的安装与 React Native 的安装，并介绍代码编辑器 Visual Studio Code 以及高效开发插件的安装，工欲善其事必先利其器，搭建一个完美的开发环境，学习起来才会更加顺畅。

2.2.1 安装 Node.js

Node.js 提供了多个平台的安装包，掌握了 Node.js，可以开发出很多跨平台的应用。

在图 2-3 的下载地址（https://nodejs.org/en/download/）中，显示了 Node.js 目前可以下载安装的平台。

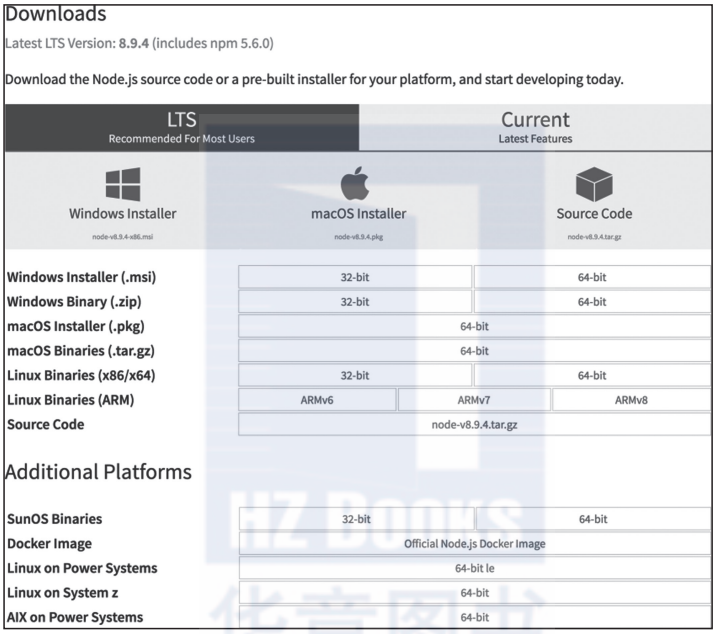


图 2-3 Node.js 下载页面

可以根据自己特定的开发环境，下载对应的版本安装即可。Node.js 官方推荐下载 LTS 版本，LTS 俗称长效版，框架整体的变更不频繁、稳定可靠，一般用于上线版本，当然学习环境的安装也推荐安装此版本。

如果需要安装其他的版本，在此下载页面的底部有 Previous Releases 链接，可查看到 Node.js 已发布的所有版本安装包。

下面以 macOS 系统为例，进行 Node.js 的安装。Windows 等其他平台下载对应的安装包安装即可，整个过程没有需要特别配置的地方，只要注意 Node.js 的安装包分为 32 位和 64 位，下载你电脑对应的安装包即可，且需要安装最低版本为 4.0 以上的 Node.js。

这里演示的是 Node.js 6.11.3 版本的安装，如图 2-4 所示，双击安装包进行安装，界面会有当前安装包包含的 Node.js 版本和 npm 版本的提示。

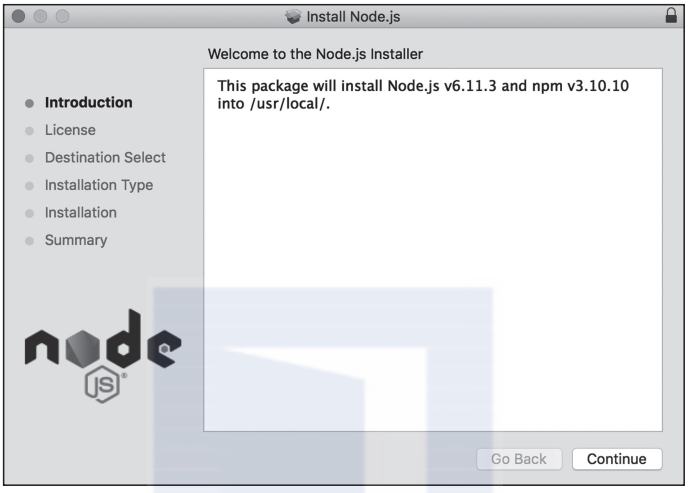


图 2-4　安装 Node.js 的版本提示

安装程序需要确认 Node.js 的 License，点击同意即可。同时会提示占用的系统空间，继续下一步。如图 2-5 所示，安装程序会进行安装，安装完成后，会在界面中提示 Node.js 和 npm 最终安装的路径，需要检查系统的全局变量是否已包含了对应的目录，一般都是默认配置好的。

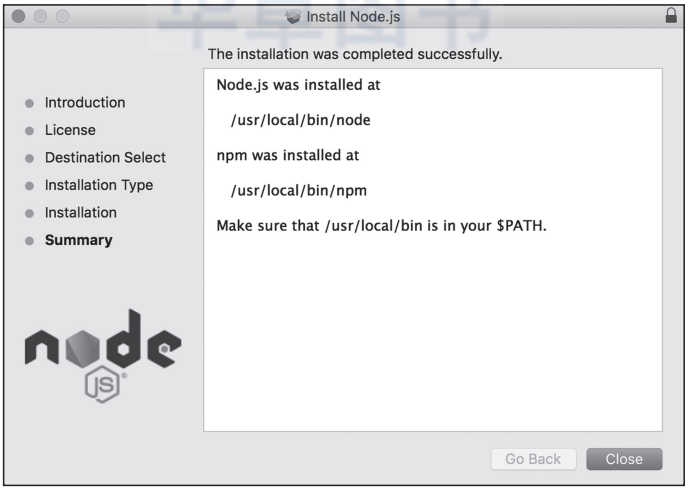


图 2-5　Node.js 安装完成的提示信息

在 Node.js 安装完成后，可以通过命令行检查 Node.js 以及 npm 是否安装成功。打开 macOS 的终端或者 Windows 系统的命令行工具，输入命令 node –v 可以查看到当前安装成功的 Node.js 版本信息，输入 npm –v 可以查看到当前连带安装的 npm 版本信息。

运行结果如图 2-6 所示，至此说明 Node.js 框架已安装成功。

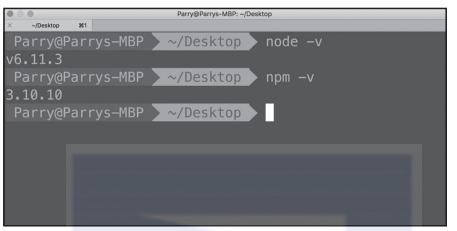


图 2-6 命令行查看 Node.js 和 npm 的版本信息

2.2.2 安装 React Native

在安装 React Native 框架之前，我们需要安装监控文件变更的组件 watchman，便于后期 React Native 项目的打包更新时提高性能之用。

在命令行中输入命令 brew install watchman 即可安装，前提是确保系统中已安装好了 Homebrew（https://brew.sh/）。

安装过程如图 2-7 所示，首次更新 Homebrew 的时间可能稍长，耐心等待片刻即可。

注意在 Windows 环境下不需要进行 Homebrew 和 watchman 的安装，跳过此安装步骤即可。

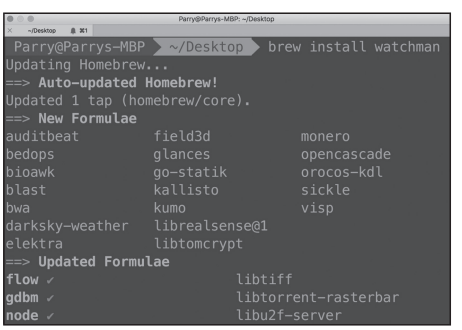


图 2-7 在 macOS 下安装 watchman

接下来我们开始安装 React Native，之前介绍 npm 时说过，React Native 也是一个 npm 的包，那么这里就可以通过 npm 命令进行 React Native 框架的安装。

图 2-8 为 React Native 在 npm 包中的项目页面，地址为：https://www.npmjs.com/package/react-native。

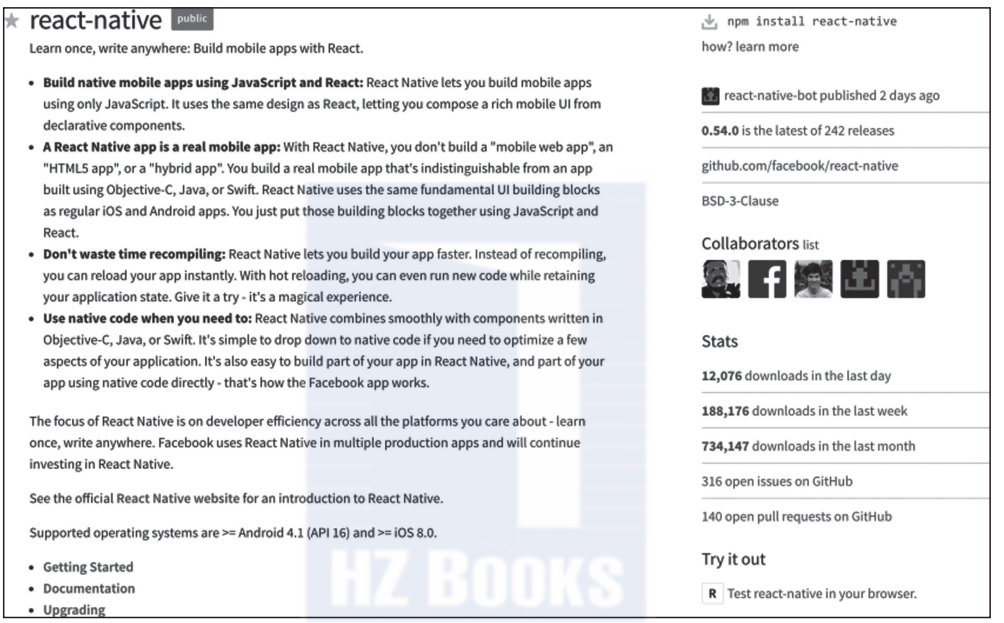


图 2-8　npm 中的 React Native 项目页面

在 npm 下安装一个包的命令格式为：npm install + 包的名称，如果加上参数 g，命令 npm install –g + 包的名称，就是全局安装，而不仅仅是在运行命令的当前目录中安装。

所以，我们通过 npm 命令执行安装 React Native CLI 的命令行工具即可，后续的 React Native 项目初始化都可以通过 React Native CLI 命令行工具执行。

安装命令为：npm install -g react-native-cli。

命令执行的结果如图 2-9 所示。

下面我们在本书配套源码的 02-02-02 文件夹中进行第一个 React Native 项目的初始化，执行命令：react-native init HelloReact。

命令执行过程如图 2-10 所示。

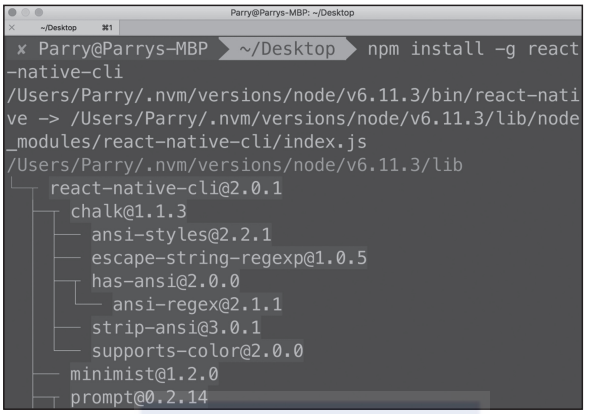


图 2-9 安装 React Native CLI

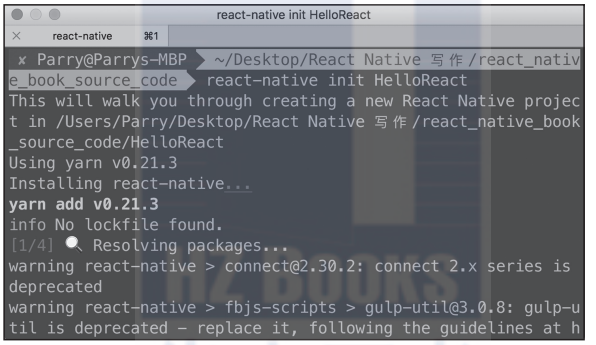


图 2-10 初始化 React Native 项目

初始化完成后，最终生成的项目文件结构如图 2-11 所示。

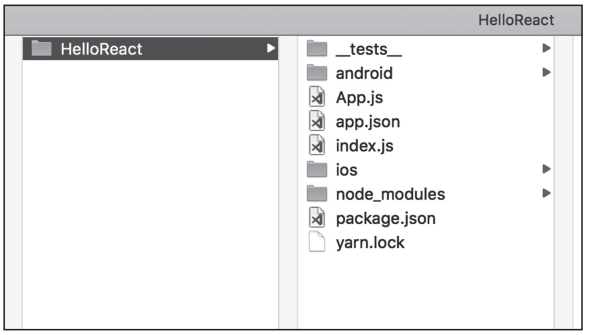


图 2-11 React Native 初始化项目结构

我们使用 Xcode 打开 ios 文件夹中的 iOS 项目以及使用 Android Studio 导入 android 文件夹中的 Android 项目。

iOS 平台的执行结果如图 2-12 所示，可以看到，React Native 构建的项目直接就适配好了 iPhone X，其他的 iOS 设备适配当然也没有任何问题。

图 2-12　项目在 iOS 平台下运行

Android 平台的执行结果如图 2-13 所示，同样，也可以完全适配。

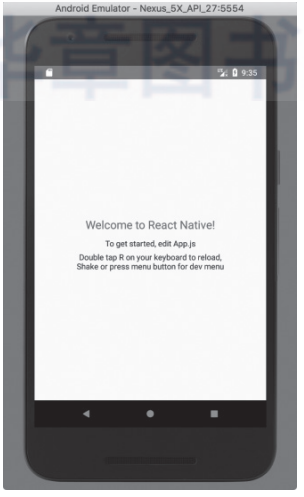


图 2-13　项目在 Android 平台下运行

到目前为止还没有写过任何一行代码，生成的 App 却已经可以完美地适配 iOS 与 Android 两个平台，这正是 React Native 平台的魅力所在，后续的实战章节我们还将继续领略此框架的魅力。

2.2.3 代码编辑器以及推荐插件

一个好的代码编辑器会让你的开发效率成倍地提升，这里从性能、界面、插件生态系统以及编辑器的更新迭代情况综合考虑，推荐大家使用微软推出的、免费的 Visual Studio Code，的确非常好用，可以说它是目前前端开发的首选编辑器。图 2-14 是 Visual Studio Code 编辑器的主界面，编辑器的左侧五个按钮依次为：项目文件浏览器、代码搜索、git 管理、调试工具、插件安装，右侧为代码编辑界面，最下面的状态栏包含了如 git 信息、代码定位、代码中的错误与警告、文件编码、代码格式等相关信息。

图 2-14 Visual Studio Code 编辑器

Visual Studio Code 还有一个很大的优势就是有很多提升开发效率的插件，这里推荐几个开发 React Native 项目必备的插件，这些插件会大大提高你的开发效率。你只需要直接在编辑器的插件选项中搜索名称即可安装。

Visual Studio Code 编辑器的插件安装界面如图 2-15 所示，在左侧的菜单按钮中选择插件，然后你可以在图中标示的搜索框中搜索需要安装插件支持的语言或直接输入插件名称，在搜索结果列表中选择对应的插件点击安装按钮即可。

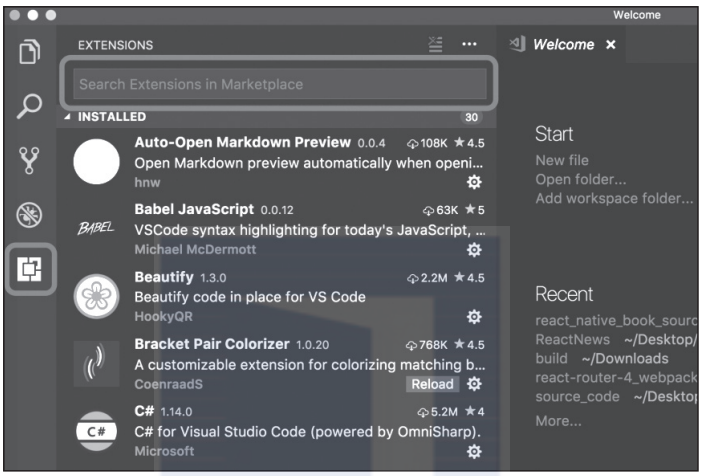


图 2-15　编辑器插件安装

1. React Native Tools

此插件提供 React Native 的开发环境，可以直接在编辑器中使用 React Native CLI 的命令，并可以对 React Native 框架提供的函数、参数、API 等进行智能提示，非常方便。

插件地址：https://marketplace.visualstudio.com/items?itemName=vsmobile.vscode-react-native/。

2. ES7 React / Redux / GraphQL / React-Native snippets

此插件在开发时提供 React Native 中会使用到的 ES 语法，可以快速生成输入。

如导入模块命令 import moduleName from 'module' 可以直接使用 imp 输入、导出模块命令 export default moduleName 只需要输入 exp 即可。

熟悉这些命令之后，可以省去很多书写特定代码的时间。

插件地址：https://marketplace.visualstudio.com/items?itemName=dsznajder.es7-react-js-snippets/。

3. react-beautify

此插件用于快速格式化 React Native 开发过程中的 JavaScript、JSX 等代码，免去很多手动整理代码格式的过程，保持规范、整洁、易读的代码格式，是一个优秀软件工程师的必备素养。

插件地址：https://marketplace.visualstudio.com/items?itemName=taichi.react-beautify/。

第 3 章 Chapter 3

React Native 工作原理与生命周期

本章将深入讲解 React Native 的底层原理，万丈高楼平地起，深入理解 React Native 底层的实现，有助于在开发中遇到难题时找到解决问题的思路。

本章介绍 React Native 的框架构成、工作原理、组件间通信，以及 React Native 中的生命周期。

如果需要直接进入 React Native 的开发与实战，可以从第 4 章开始学习。

3.1 React Native 框架及工作原理

React Native 框架内部提供了很多的内置组件，如图 3-1 所示。包括基本组件，如 View、Text 等，用于一些功能布局的 Button、Picker 等，iOS 平台与 Android 平台的特定组件、API 等。同时也提供了接口便于与原生平台进行交互。后续的章节我们会介绍与原生平台的混合实战开发。

在介绍 React 框架的章节，我们理解了如何将代码渲染至虚拟 DOM 并更新到真实 DOM 的过程。在 React Native 框架中，渲染到 iOS 平台与 Android 平台的过程如图 3-2 所示。

在 React 框架中，JSX 源码通过 React 框架最终渲染到了浏览器的真实 DOM 中，而在 React Native 框架中，JSX 源码通过 React Native 框架编译后，通过对应平台的 Bridge 实现了与原生框架的通信。如果我们在程序中调用了 React Native 提

供的 API，那么 React Native 框架就通过 Bridge 调用原生框架中的方法。

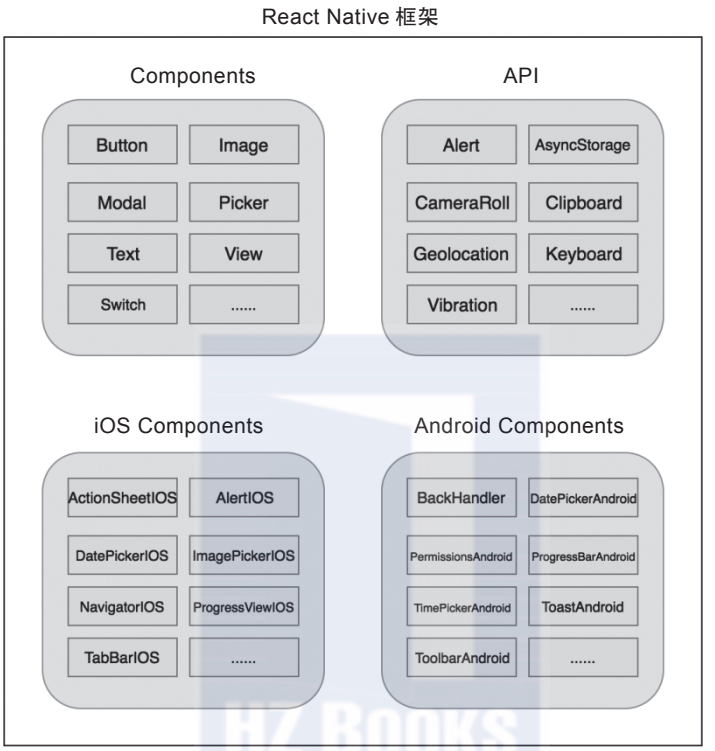


图 3-1 React Native 框架构成

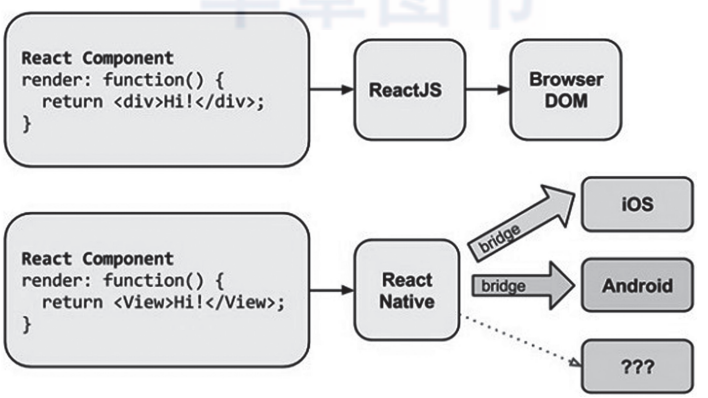


图 3-2 React Native 渲染

因为 React Native 的底层为 React 框架，所以，如果是 UI 层的变更，那么就映射为虚拟 DOM 后调用 diff 算法计算出变动后的 JSON 映射文件，最终由 Native 层将此 JSON 文件映射渲染到原生 App 的页面元素上，实现了在项目中只需控制 state 以及 props 的变更来引起 iOS 与 Android 平台的 UI 变更。

编写的 React Native 代码最终会被打包生成一个 main.bundle.js 文件供 App 加载，此文件可以存储在 App 设备本地，也可以存储于服务器上，以供 App 下载更新，后续章节讲解的热更新就会涉及 main.bundle.js 位置的设置问题。

3.1.1　React Native 与原生平台通信

React Native 在与原生框架通信中，如图 3-3 所示，采用了 JavaScriptCore 作为 JS VM，中间通过 JSON 文件与 Bridge 进行通信。若使用 Chrome 浏览器进行调试，那么所有的 JavaScript 代码都将运行在 Chrome 的 V8 引擎中，与原生代码通过 WebSocket 进行通信。

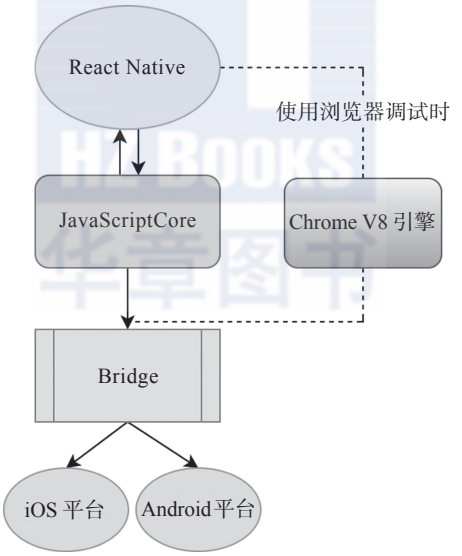


图 3-3　React Native 与原生平台的通信

3.1.2　组件间通信

React Native 开发最基本的元素就是组件，React Native 与 React 一样，也会涉及组件之间的通信，便于数据在组件之间传递，下面列出了几种常用的组件间通信方式。

1. 父子组件的通信

如同之前介绍 React 组件间参数传递一样，在 React Native 中，可以通过 props 的形式实现父组件向子组件传递值。

在下例中，父组件通过调用子组件并赋值子组件的 name 为 React，子组件通过 this.props.name 获取父组件传递过来的 name 的字符串值 React。

> 完整代码在本书配套源码的 03-04 文件夹。

```
/**
* 章节: 03-04
* 父子组件通信，在父组件中调用子组件
* FilePath: /03-04/parent-2-child.js
* @Parry
*/

<ChildComponent name='React'/>

/**
* 章节: 03-04
* 子组件实现，通过 props 获取父组件传递的值
* FilePath: /03-04/parent-2-child.js
* @Parry
*/

class ChildComponent extends Component {
  render() {
    return (
      <Text>Hello {this.props.name}!</Text>
    );
  }
}
```

2. 子父组件的通信

在开发过程中，还会有子组件向父组件通信传递值的需求，比如当子组件的某个值变更后，需要通知到父组件做相应的变更与响应，那么就需要子父组件之间的通信。

例如，在父组件的定义中，在调用子组件时，同样向子组件传递了一个参数，不过这个参数是一个函数，此函数用于接收后续子组件向父组件传递过来的数据，与之前父组件向子组件传递数据不太一样。

完整代码在本书配套源码的 03-04 文件夹。

```
/**
* 章节: 03-04
* 子父组件通信，父组件的实现
* FilePath: /03-04/child-2-parent.js
* @Parry
*/
import React, {Component} from 'react';
import ChildComponent from './ChildComponent'

class App extends Component {
  constructor(props) {
    super(props)
    this.state = {
      name: 'React'
    }
  }

  //传递到子组件的参数，不过参数是一个函数。
  handleChangeName(nickName) {
    this.setState({name: nickName})
  }

  render() {
    return (
      <div>
        <p>父组件的 name: {this.state.name}</p>
        <ChildComponent
          onChange={(val) => {
          this.handleChangeName(val)
        }}/>
      </div>
    );
  }
}

export default App;
```

下面为子组件的定义，子组件在页面中定义了一个按钮，点击此按钮调用自身的一个函数 handleChange，修改了自身 state 中的值 name 为 nickName 定义的值 Parry，那么此子组件的页面上的字符串将由之前的 Hello React! 变为 Hello Parry!，

同时使用了 this.props.changeName，也就是父组件调用时传递过来的函数，向父组件传递了 nickName 的值 Parry。

父组件在接收到子组件的调用后，调用了父组件自身的函数 handleChange Name 修改了自身的 state 中的 name 的值为 Parry，也就是子组件传递过来的 Parry，所以，父组件的页面上的值也同时由之前的 React 变更成了 Parry。代码如下：

```
/**
* 章节: 03-04
* 子父组件通信，子组件的实现
* FilePath: /03-04/child-2-parent.js
* @Parry
*/

import React, {Component} from 'react'

export default class ChildComponent extends Component {
  constructor(props) {
    super(props)

    this.state = {
      name: 'React'
    }
  }

  handleChange() {
    const nickName = 'Parry';
    this.setState({name: nickName})
    //调用父组件传递过来的函数参数，传递值到父组件去。
    this
      .props
      .changeName(nickName)
  }

  render() {
    const {name} = this.state;
    return (
      <div>
        <p>Hello {name}!</p>
        <Button
          onPress={this
          .handleChange
          .bind(this)}
```

```
37.            title="修改一下 name 为 Parry"/>
38.        </div>
39.     )
40.   }
41. }
```

3. 多级组件之间的通信

如果组件之间的父子层级非常多，需要进行组件之间的传递，这时候当然可以通过上面介绍的方法逐级传递，但这样的传递方法不是一个太好的方法。

因为组件之间通信冗长，嵌套逻辑太深，会导致用户体验不好，可以想象一下用户从最底层一层层操作返回到最顶层时的体验。

可以使用如 context 对象或 global 等方式进行多级组件间的通信，但是不推荐这种方式。最好不要让组件之间的层级关系太深。

4. 无直接关系组件间通信

前面提到的都是有层级关系的组件间的通信方式，如果组件间没有层级关系的话，则可以通过如 AsyncStorage 或 JSON 文件等方式进行通信。

当然，还可以使用 EventEmitter/EventTarget/EventDispatcher 继承或实现接口的方式、Signals 模式或 Publish/Subscribe 的广播形式，都可以达到无直接关系组件间的通信。

这些组件间的通信方式使得组件之间可以传递数据，后续的实战章节会有详细的代码实现，这里主要进行了理论的介绍。掌握这部分知识后才可以将 App 开发中的基本单位（也就是组件）串联起来。

3.2　React Native 中的生命周期

任何生命体都会经历从出生到消亡的过程，React Native 框架中的组件同样具有这样的属性。在组件生命周期的每个阶段，React Native 提供了多个生命周期函数，供开发者作为切入组件的钩子（hook），这样在对应的时间点程序就可以做对应的逻辑处理，从而实现相应的功能。

在 React Native 程序启动时，内部的虚拟 DOM 开始建立，生命周期就是建立在此虚拟 DOM 的整个生命周期之中，从虚拟 DOM 的初始化到虚拟 DOM 的卸载，React Native 为组件的不同状态建立了不同的生命周期。

在图 3-4 中，可以看到在 React Native 虚拟 DOM 的几个大的阶段中，都存在对应的生命周期函数。下面就分阶段介绍。

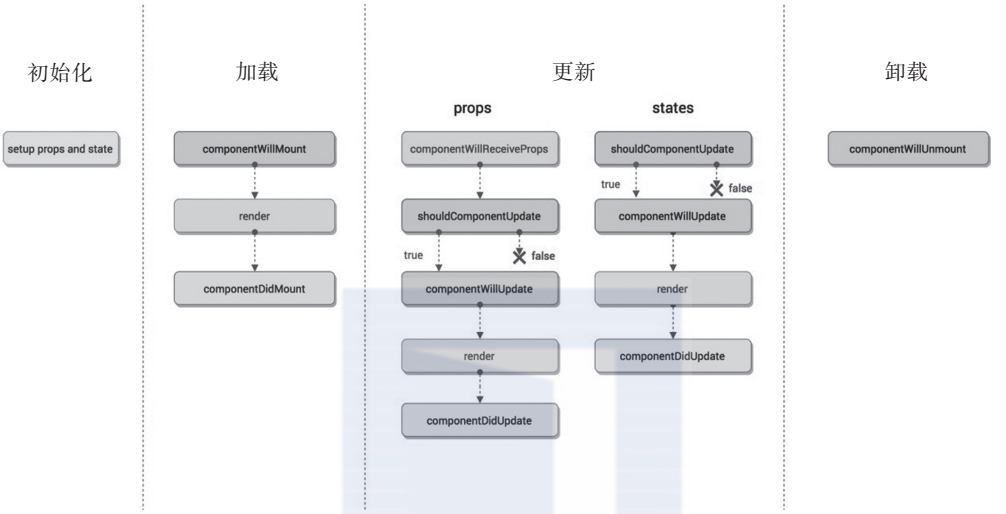


图 3-4 React Native 中的生命周期

1. 初始化阶段

此阶段设定组件 props 和 state 的默认值，可通过如下代码赋值：

```
1. static defaultProps = {
2.   autoPlay: false,
3.   maxLoop: 10,
4. };
```

2. 加载阶段

此阶段为组件开始实例化的阶段，比如当该组件被其他组件调用的时候。主要包含以下三个函数：

- componentWillMount：组件将要开始加载，若需要在组件开始加载前添加一些业务逻辑，那么就可以在此函数中添加。
- render：组件开始根据 props 和 state 生成页面的 DOM，并最终返回此 DOM。在此函数中不可以修改 props 和 state 的值，只可以读取，并且返回的 DOM 只能有一个顶层元素，比如说只能由一个 div 包裹所有的元素返回。
- componentDidMount：组件已加载完毕，在 render 函数之后立即执行此函数。

可以在这里进行网络请求，因为组件 UI 渲染好之后再进行网络请求，一般不会造成 UI 的错乱问题。在此生命周期函数中修改了 state 的值后，UI 会立即进行重新渲染，所以这是一个通过加载网络数据更新 UI 的好时机。

3. 更新阶段

当用户操作或者父组件有更新并且组件由于 props 或 state 的变更导致组件需要重新渲染时，会经历此阶段。而在更新渲染的几个重要时机，React Native 提供了如下的生命周期函数供开发者执行对应的逻辑操作：

- componentWillReceiveProps ：当接收到更新的 props 值时，会执行此函数，此时可以将接收到的 props 值赋值给 state。
- shouldComponentUpdate ：此函数用于判断新的 props 和 state 的变更需不需要引起组件的 UI 更新，默认是都会引起更新的，但是 React Native 提供了此函数供开发者自主决定是否需要更新。如果此函数返回 True，那么组件将进行更新，如果返回 False，那么组件就不更新。此函数在优化 App 性能时非常重要，因为可以通过此函数拦截掉很多不必要的组件 UI 更新。
- componentWillUpdate ：如果以上函数 shouldComponentUpdate 返回了 True，那么此函数将继续执行，表示组件即将进行更新操作。在更新操作前，还有时机进行相关的逻辑处理。但是从逻辑上你也应该明白，这里不可以再修改 state 的值，而只需做一些更新前的其他准备工作。
- componentDidUpdate ：组件更新完毕之后执行的函数。此函数有两个参数 prevProps 和 prevState，分别为更新前的 props 与 state。这里可以进行新旧值的比较，如果发现值有变化则可以进行一些网络请求、加载数据等操作。

4. 卸载阶段

- componentWillUnmount：此函数在组件被卸载和注销前执行，这里可以进行一些所谓的扫尾工作，如关闭之前的网络请求、清空一些不必要的存储、循环执行的定时器的清除等。

至此，React Native 一个组件的完整生命周期执行完毕，你可以通过下面的代码体会 React Native 每个阶段的执行过程。实际开发时只需要根据项目需求在对应的生命周期函数中添加上自己的业务逻辑即可。

```
/**
* 章节: 03-06
* React Native 中的生命周期
* FilePath: /03-06/lifecycle.js
* @Parry
*/

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

export default class LifeCycle extends Component {

  constructor(props) {
    super(props);
    this.state = {
      name: "React"
    }
  }

  //组件即将加载
  componentWillMount() {
    console.log("componentWillMount");
  }

  //开始组件渲染
  render() {
    console.log("render");
    return (
      <View>
        <Text>
          {this.state.name}
        </Text>
      </View>
    );
  }

  //组件加载完毕后
  componentDidMount() {
    console.log("componentDidMount");
  }

  //组件接收到新的 props
  componentWillReceiveProps(nextProps) {
    console.log("componentWillReceiveProps");
```

```
45.   }
46.
47.   //逻辑控制是否需要更新组件
48.   shouldComponentUpdate(nextProps, nextState) {
49.     console.log("shouldComponentUpdate");
50.   }
51.
52.   //组件即将更新重新渲染
53.   componentWillUpdate(nextProps, nextState) {
54.     console.log("componentWillUpdate");
55.   }
56.
57.   //组件重新渲染后
58.   componentDidUpdate(prevProps, prevState) {
59.     console.log("componentDidUpdate");
60.   }
61.
62.   //组件卸载注销前
63.   componentWillUnmount() {
64.     console.log("componentWillUnmount");
65.   }
66. }
67.
68. AppRegistry.registerComponent('LifeCycle', () => Main);
```

3.3　本章小结

如同武功修炼中必备的内功一样，本章看起来和使用 React Native 框架的关系不大，而且底层原理理解起来还有点晦涩难懂。不过，如果你想成为一个精通 React Native 框架的开发者，而不仅仅是一个使用者的话，这部分内容是非常重要的，而且在后期遇到此框架的难题时，你可以根据这部分底层原理性的知识快速找到问题的原因。其他软件开发语言的学习原则也是如此，希望能对你有所启发。