深入React技术栈

作者：陈屹

版权信息

书名：深入React技术栈

作者：陈屹

ISBN：978-7-115-43730-3

序

React 是目前前端工程化最前沿的技术。2004 年 Gmail的推出，让大家猛然发现，单页应用的互动也可以如此流畅。2010年，前端单页应用框架接踵而至，Backbone、Knockout、Angular，各领风骚。2013年，React 横空出世，独树一帜：单向绑定、声明式UI，大大简化了大型应用的构建。Strikingly 接触到 React 之后不久，就开始用 React 重构前端。

当时我想，2013 年或许会因为 React 的出现，成为前端社区的分水岭。今天回看，确实如此。

毋庸置疑，React 已经是前端社区里程碑式的技术。React 及其生态圈不断提出前端工程化解决方案，引领潮流。在过去一两年里，React 也是各种技术交流分享会里炙手可热的议题。

React 之所以流行，在于它平衡了函数式编程的约束与工程师的实用主义。

React 从函数式编程社区中借鉴了许多约定：把 DOM 当成纯函数，不仅免去了烦琐的手动 DOM 操作，还开启了多平台渲染的美丽新世界；在此之上，React 社区进一步强调不可变性（immutability）和单向数据流。这几个约定将原本很复杂的程序化简，加强了程序的可预测性。

React 也有实用主义的一面，它不强迫工程师只用函数式，而是提供了简单粗暴的手段，方便你实现各种功能——想直接操作 DOM 也可以，想双向绑定也没问题。函数式约定搭配实用主义，让我不禁想起 Facebook 一直倡导的黑客之道：Done is better than perfect。

React 还是一门年轻的技术，网上能学习的材料也比较零散。本书由浅到深，手把手地带领读者了解 React 核心思想和实现机制。因为 React 受到了很多关注，社区里出现了各种建立大型 React 应用的方案。本书总结了目前社区里的最佳实践，方便读者立刻在实战中使用。

郭达峰

Strikingly 联合创始人及 CTO

前言

前端高速发展十余年，我们看到了浏览器厂商的竞争，经历了标准库的竞争，也经历了短短几年 ECMAScript 标准的迭代。到今天，JavaScript 以完全不同的方式呈现出来。

这是最好的时代，这是最坏的时代，这是智慧的时代，这是愚蠢的时代；这是信仰的时期，这是怀疑的时期；这是光明的季节，这是黑暗的季节；这是希望之春，这是失望之冬。

这是对前端发展这些年最恰当的概括。整个互联网应用经历了从轻客户端到富客户端的变化，前端应用的规模变得越来越大，交互越来越复杂。在近几年，前端工程用简单的方法库已经不能维系应用的复杂度，需要使用一种框架的思想去构建应用。因此，我们看到 MVC、MVVM 这些 B/S 或 C/S 中常见的分层模型都出现在前端开发的过程中。与其说不断在创新，还不如说前端在学习之前应用端已经积累下来的浑厚体系。

在发展的过程中，出现了大量优秀的框架，比如 Backbone、Angular、Knockout、Ember 这些框架大都应用了 MV* 的理念，把数据与视图分离。而就在这样纷繁复杂的时期，2013 年 Facebook 发布了名为React的前端库。

从表现上看，React 被大部分人理解成 View 库。然而，从它的功能上看，它远远复杂于 View 的承载。它的出现可以说是灵光一现，我记得曾经有人说过，Facebook 发布的技术产品总是包含伟大的思想。的确，从此，Virtual DOM、服务端渲染，甚至 power native apps，这些概念开始引发一轮新的思考。

从官方描述中，创造 React 是为了构建随着时间数据不断变化的大规模应用程序。正如它的描述一样，React 结合了效率不低的 Virtual DOM 渲染技术，让构建可组合的组件的思路可行。我们只要关注组件自身的逻辑、复用及测试，就可以把大型应用程序玩得游刃有余。

在0.13版本之后，React 也慢慢趋于稳定，越来越多的前端工程师愿意选择它作为应用开发的首选，国内也有很多应用开始用它作为主架构的核心库。

在未来，React 必然不过是一块小石头沉入水底，但它溅起的涟漪影响了无数的前端开发的思维，影响了无数应用的构建。对于它来说，这些就是它的成就。成就 JavaScript 的繁荣，成就前端标准更快地推进。

本书目的

本书希望从实践起步，以深刻的角度去解读 React 这个库给前端界带来的革命性变化。

目前，不论在国内，还是在国外，已经有一些入门的 React 图书，它们大多在介绍基本概念，那些内容可以让你方便地进入 React 世界。但本书除了详细阐述基本概念外，还会帮助你从了解 React 到熟悉其原理，从探索 Flux 应用架构的思想到精通 Redux 应用架构，帮助你思考 React 给前端界带来的价值。React 今天是一种思想，希望通过解读它，能够让读者有自学的能力。

阅读建议

本书从几个维度介绍了 React。一是作为 View 库，它怎么实现组件化，以及它背后的实现原理。二是扩展到 Flux 应用架构及重要的衍生品 Redux，它们怎么与 React 结合做应用开发。三是对它与 server 的碰撞产生的一些思考。四是讲述它在可视化方面有着怎样的优势与劣势。

下面是各章的详细介绍。

第 1 章　这一章从 React 最基本的概念与 API 讲起，让读者熟悉 React 的编码过程。

第 2 章　这一章更深入到 React 的方方面面，并从一个具体实例的实现到自动化测试过程来讲述 React 组件化的过程和思路。

第 3 章　这一章深入到 React 源码，介绍了 React 背后的实现原理，包括 Virtual DOM、diff 算法到生命周期的管理，以及 setState 机制。

第 4 章　这一章介绍了 React 官方应用架构组合 Flux，从讲解 Flux 的基本概念及其与 MV* 架构的不同开始，解读 Flux 的核心思想。

第 5 章　这一章介绍了业界炙手可热的应用架构 Redux，从构建一个 SPA 应用讲到背后的实现逻辑，并扩展了 Redux 生态圈中常用的 middleware 和 utils 方法。

第 6 章　这一章讲述 Redux 高阶运用，包括高阶 reducer、它在表单中的运用以及性能优化的方法。另外，从源码的角度解读了 Redux。

第 7 章　这一章介绍了 React 在服务端渲染的方法，并从一个实例出发结合 Koa 完整地讲述了同构的实现。

第 8 章　这一章探索了实现可视化图形图表的方法，以及如何通过这些方法和 React 结合在一起运转。

附录 A　探讨了 React 开发环境的基本组成部分以及常规的安装方法。

附录 B　探讨了团队实践或多人协作过程中需要关注的编码规范问题。

附录 C　探讨了 Koa middleware 的相关知识，帮助理解 Redux middleware。

代码规范

本书的 JavaScript 示例代码均使用 ES2015/ES6 编写，并遵循 Airbnb JavaScript 规范，但诸如 React 或 Redux 源代码引用的原始代码除外。

本书的 CSS 示例代码均为 SCSS 代码，但引用源码库的 CSS 除外。

保留英文名词

对于 React/Flux/Redux 中常用的专有名词，在不造成读者理解困难的情况下，本书尽量保留英文名词，保持原汁原味。

- Virtual DOM：虚拟 DOM
- state：状态
- props：属性
- action：动作
- reducer
- store
- middleware：中间件
- dispatcher：分发器
- action creator：action 构造器
- currying：柯里化

读者反馈

如果你有什么好的意见和建议，请及时反馈给我们。可以通过 i.arcthur@gmail.com 或在知乎上发私信找到我。

示例代码下载

本书的示例代码1 托管在 https://github.com/arcthur/react-book-examples 和https://coding.net/u/arcthur/p/react-book-examples/git ，它可能会和书中的内容有所出入，因为我们会根据情况对代码略加修改，所以在阅读的时候，建议结合文档一同查看。

1 本书的源代码也可从图灵社区（www.ituring.com.cn ）本书主页免费注册下载。

致谢

从 React 诞生以来，我就在关注这个领域。在 2015 年年底，我在知乎上开辟了名为 pure render 的专栏。不论是我现在的角色，还是从建设一个团队的角度来考虑，我都想把在 React 实践中的心路历程分享出来，和大家一起学习，共同成长。

万万没想到，专栏的持续写作得到了相当多知友的认可。截止今天，专栏运行 8 个月左右，积累了20篇文章，得到了 4500 多人的关注。对于团队来说，既是鼓舞，更是压力。专栏在运行过程中，参与的伙伴也渐渐变多，我希望它可以一直保持高质量，让整个社区的 React 爱好者们一起贡献。

专栏写作不久，就有几位编辑老师找到我，那时我并没有准备好去系统地撰写一些内容，但随着专栏中沉淀的文字越来越多，我想不妨可以试着写一些关于 React 的更深入的分析，以及整体应用层面上的实践，让更多开发者，乃至 IT 圈更多地关注这个库。在写作本书这半年多的时间内，React 在业界的关注度不断上升，也涌现出很多优秀的实践，我非常感谢我身在的这个社区。

耗费了大量晚上及周末的时间，断断续续的编写与修改，书稿的内容终于定下来了，其中很多内容是对专栏已有内容的修复与升级。书与专栏同样是文字的传播，平台不同，初衷却是一样的。我希望它可以精益求精，至少能在一定程度上帮助开发者深入学习 React。

在此，特别感谢知乎 pure render 专栏组的所有成员献计献力，其中杨森、丁玲、李彬彬、黄宗权、范洪春、宋邵茵、胡可本、胡清亮等组员不同程度地贡献了实战经验与想法，并参与审校本书当中。真心感谢你们，是你们的热情和坚持让这本书可以面世。

同样感谢写作中给予很多宝贵意见的朋友们，包括魏畅然、赵剑飞、李成熙、胡杰、郭达峰、阮一峰、张克军、寸志、张克炎等。

最后，由衷地感谢王军花老师从头到尾认真负责的态度，让这本书更精彩。

陈屹

2016年7月1日于杭州

第 1 章　初入 React 世界

欢迎进入 React 世界。从本章开始，不论你是刚刚入门的前端开发者，还是经验老道的资深工程师，都可以学习到 React 的基本思想以及基本用法。在之后慢慢深入的过程中，各节均会不同程度地带上进阶的实践与分析。希望在本章结束时，我们能够带领你实现应用 React 进行基本的组件开发。请从这里开始你的旅程……

1.1　React 简介

React 是 Facebook 在 2013 年开源在 GitHub 上的 JavaScript 库。React 把用户界面抽象成一个个组件，如按钮组件 Button、对话框组件 Dialog、日期组件 Calendar。开发者通过组合这些组件，最终得到功能丰富、可交互的页面。通过引入 JSX 语法，复用组件变得非常容易，同时也能保证组件结构清晰。有了组件这层抽象，React 把代码和真实渲染目标隔离开来，除了可以在浏览器端渲染到 DOM 来开发网页外，还能用于开发原生移动应用。

1.1.1　专注视图层

现在的应用已经变得前所未有的复杂，因而开发工具也必须变得越来越强大。和 Angular、Ember 等框架不同，React 并不是完整的 MVC/MVVM 框架，它专注于提供清晰、

简洁的 View（视图）层解决方案。而又与模板引擎不同，React 不仅专注于解决 View 层的问题，又是一个包括 View 和 Controller 的库。对于复杂的应用，可以根据应用场景自行选择业务层框架，并根据需要搭配 Flux、Redux、GraphQL/Relay 来使用。

React 不像其他框架那样提供了许多复杂的概念与烦琐的 API，它以 Minimal API Interface 为目标，只提供组件化相关的非常少量的 API。同时为了保持灵活性，它没有自创一套规则，而是尽可能地让用户使用原生 JavaScript 进行开发。只要熟悉原生 JavaScript 并了解重要概念后，就可以很容易上手 React 应用开发。

1.1.2 Virtual DOM

真实页面对应一个 DOM 树。在传统页面的开发模式中，每次需要更新页面时，都要手动操作 DOM 来进行更新，如图 1-1 所示。

图 1-1 传统 DOM 更新

DOM 操作非常昂贵。我们都知道在前端开发中，性能消耗最大的就是 DOM 操作，而且这部分代码会让整体项目的代码变得难以维护。React 把真实 DOM 树转换成 JavaScript 对象树，也就是 Virtual DOM，如图 1-2 所示。

图 1-2 React DOM 更新

每次数据更新后，重新计算 Virtual DOM，并和上一次生成的 Virtual DOM 做对比，对发生变化的部分做批量更新。React 也提供了直观的 `shouldComponentUpdate` 生命周期回调，来减少数据变化后不必要的 Virtual DOM 对比过程，以保证性能。

我们说 Virtual DOM 提升了 React 的性能，但这并不是 React 的唯一亮点。此外，Virtual DOM 的渲染方式也比传统 DOM 操作好一些，但并不明显，因为对比 DOM 节点也是需要计算资源的。

它最大的好处其实还在于方便和其他平台集成，比如 react-native 是基于 Virtual DOM 渲染出原生控件，因为 React 组件可以映射为对应的原生控件。在输出的时候，是输出 Web DOM，还是 Android 控件，还是 iOS 控件，就由平台本身决定了。因此，react-native 有一个口号——Learn Once，Write Anywhere。

1.1.3 函数式编程

在过去，工业界的编程方式一直以命令式编程为主。命令式编程解决的是做什么的问题，比如图灵机，而现代计算机就是一个经历了多次进化的高级图灵机。如果说人脑最擅长的是分析问题，那么电脑最擅长的就是执行指令，电脑只需要几条汇编指令就可以轻松算出我们需要很长时间才能解出的运算。命令式编程就像是在给电脑下命令，现在主要的编程语言（包括 C 和 Java 等）都是由命令式编程构建起来的。

而函数式编程，对应的是声明式编程，它是人类模仿自己逻辑思考方式发明出来的。声明式编程的本质是 lambda 演算1 。试想当我们操作数组的每个元素并返回一个新数组时，如果是计算机的思考方式，则是需要一个新数组，然后遍历原数组，并计算赋值；如果是人的思考方式，则是构建一个规则，这个过程就变成构建一个 ε 函数作用在数组上，然后返回新数组。这样，计算可以被重复利用。

1 lambda calculus，详见 https://en.wikipedia.org/wiki/Lambda_calculus 。

当回到 UI 界面上，我们的产品经理又想出了一个新点子时，我们是抱怨呢，还是去思考怎么解决这个问题。React 把过去不断重复构建 UI 的过程抽象成了组件，且在给定参数的情况下约定渲染对应的 UI 界面。React 能充分利用很多函数式方法去减少冗余代码。此外，由于它本身就是简单函数，所以易于测试。可以说，函数式编程才是 React 的精髓。

1.2 JSX 语法

当初学 React 时，JSX 是我们遇到的第一个新概念。也许我们都是写惯了 JavaScript 程序的开发者，对于类似于静态编译并不感冒。早些年风靡前端界的 CoffeeScript，也因为 ES6 标准化的加速推进，慢慢变为了茶余饭后的谈资。面对 React，我们又一次需要玩转一门新的静态转译语言，而这一次，又会有什么不一样的体验呢。

1.2.1 JSX 的由来

JSX 与 React 有什么关系呢？简单来讲，React 为方便 View 层组件化，承载了构建 HTML 结构化页面的职责。从这点上来看，React 与其他 JavaScript 模板语言有着许多异曲同工之处，但不同之处在于 React 是通过创建与更新虚拟元素（virtual element）来管理整个 Virtual DOM 的。

> **说明** JSX 语言的名字最早出现在游戏厂商 DeNA，但和 React 中的 JSX 不同的是，它意在通过加入增强语法，使得 JavaScript 变得更快、更安全、更简单。

其中，虚拟元素可以理解为真实元素的对应，它的构建与更新都是在内存中完成的，并不会真正渲染到 DOM 中去。在 React 中创建的虚拟元素可以分为两类，DOM 元素（DOM element）与组件元素（component element），分别对应着原生 DOM 元素与自定义元素，而 JSX 与创建元素的过程有着莫大的关联。

接着，我们从这两种元素的构建开始说起。

1. DOM 元素

从过往的经验中知道，Web 页面是由一个个 HTML 元素嵌套组合而成的。当使用 JavaScript 来描述这些元素的时候，这些元素可以简单地被表示成纯粹的 JSON 对象。比如，现在需要描述一个按钮（button），这用 HTML 语法表示非常简单：

```
<button class="btn btn-blue">
  <em>Confirm</em>
</button>
```

其中包括了元素的类型和属性。如果转成 JSON 对象，那么依然包括元素的类型以及属性：

```
{
  type: 'button',
  props: {
    className: 'btn btn-blue',
    children: [{
      type: 'em',
      props: {
        children: 'Confirm'
      }
    }]
  }
}
```

这样，我们就可以在 JavaScript 中创建 Virtual DOM 元素了。

在 React 中，到处都是可以复用的元素，这些元素并不是真实的实例，它只是让 React 告诉开发者想要在屏幕上显示什么。我们无法通过方法去调用这些元素，它们只是不可变的描述对象。

2. 组件元素

当然，我们可以很方便地封装上述 `button` 元素，得到一种构建按钮的公共方法：

```
const Button = ({ color, text }) => {
  return {
    type: 'button',
    props: {
      className: `btn btn-${color}`,
      children: {
        type: 'em',
        props: {
          children: text,
        },
      },
    },
  };
}
```

自然，当我们要生成 DOM 元素中具体的按钮时，就可以方便地调用 `Button({color: 'blue', text: 'Confirm'})` 来创建。

仔细思考这个过程可以发现，`Button` 方法其实也可以作为元素而存在，方法名对应了 DOM 元素类型，参数对应了 DOM 元素属性，那么它就具备了元素的两大必要条件，这样构建的元素就是自定义类型的元素，或称为组件元素。我们用 JSON 结构来描述它：

```
{
  type: Button,
  props: {
    color: 'blue',
    children: 'Confirm'
  }
}
```

这也是 React 的核心思想之一。因为有公共的表达方法，我们就可以让元素们彼此嵌套或混合。这些层层封装的组件元素，就是所谓的 React 组件，最终我们可以用递归渲染的方式构建出完全的 DOM 元素树。

我们再来看一个封装得更深的例子。为上述 Button 元素再封装一次，它由一个方法构建而成：

```
const DangerButton = ({ text }) => ({
  type: Button,
  props: {
    color: 'red',
    children: text
  }
});
```

直观地看，`DangerButton` 从视觉上为我们定义了“危险的按钮”这样一种新的组件元素。接着，我们可以很轻松地运用它，继续封装新的组件元素：

```
const DeleteAccount = () => ({
  type: 'div',
  props: {
    children: [{
      type: 'p',
      props: {
        children: 'Are you sure?',
      },
    }, {
      type: DangerButton,
      props: {
        children: 'Confirm',
      },
    }, {
      type: Button,
      props: {
        color: 'blue',
        children: 'Cancel',
      },
    }],
  }
});
```

`DeleteAccount` 清晰地表达了一个功能模块、一段提示语、一个表示确认的警示按钮和一个表示取消的普通按钮。不过在表达还不怎么复杂的结构时，它就力不从心了。这让我们想起使用 HTML 书写结构时的畅快感受，JSX 语法为此应运而生。假如我们使用 JSX 语法来重新表达上述组件元素，只需这么写：

```
const DeleteAccount = () => (
  <div>
    <p>Are you sure?</p>
    <DangerButton>Confirm</DangerButton>
    <Button color="blue">Cancel</Button>
  </div>
);
```

> **注意**　上述 `DeleteAccount` 并不是真实转换，在实际场景中构建元素会考虑到诸如安全等因素，会由 React 内部方法创建虚拟元素。如果需要自己构建虚拟元素，原理也是一样的。

如你所见，JSX 将 HTML 语法直接加入到 JavaScript 代码中，再通过翻译器转换到纯 JavaScript 后由浏览器执行。在实际开发中，JSX 在产品打包阶段都已经编译成纯 JavaScript，不会带来任何副作用，反而会让代码更加直观并易于维护。尽管 JSX 是第三方标准，但这套标准适用于任何一套框架。

React 官方在早期为 JSX 语法解析开发了一套编译器 JSTransform，目前已经不再维护，现在已全部采用 Babel 的 JSX 编译器实现。因为两者在功能上完全重复，而 Babel 作为专门的 JavaScript 语法编译工具，提供了更为强大的功能，达到了“一处配置，统一运行”的目的。

我们试着将 DeleteAccount 组件通过 Babel 转译成 React 可以执行的代码：

```
var DeleteAccount = function DeleteAccount() {
  return React.createElement(
    'div',
    null,
    React.createElement(
      'p',
      null,
      'Are you sure?'
    ),
    React.createElement(
      DangerButton,
      null,
      'Confirm'
    ),
    React.createElement(
      Button,
      { color: 'blue' },
      'Cancel'
    )
  );
};
```

可以看到，除了在创建元素时使用 `React.createElement` 创建之外，其结构与一直在讲的 JSON 的结构是一致的。

反过来说，JSX 并不是强制选项，我们可以像上述代码那样直接书写而无须编译，但这实在是极其糟糕的编程体验。JSX 的出现为我们省去了这个烦琐过程，使用 JSX 写法的代码更易于阅读与开发。事实上，JSX 并不需要花精力学习。只要你熟悉 HTML 标签，大多数功能就都可以直接使用了。

1.2.2 JSX 基本语法

JSX 的官方定义是类 XML 语法的 ECMAScript 扩展。它完美地利用了 JavaScript 自带的语法和特性，并使用大家熟悉的 HTML 语法来创建虚拟元素。可以说，JSX 基本语法基本被 XML 囊括了，但也有少许不同之处。接着我们从基本语法、元素类型、元素属性、JavaScript 属性表达式等维度一一讲述。

1. XML 基本语法

使用类 XML 语法的好处是标签可以任意嵌套，我们可以像 HTML 一样清晰地看到 DOM 树状结构及其属性。比如，我们构造一个 List 组件：

```
const List = () => (
  <div>
    <Title>This is title</Title>
    <ul>
      <li>list item</li>
      <li>list item</li>
      <li>list item</li>
    </ul>
  </div>
);
```

写 List 的过程就像写 HTML 一样，只不过它被包裹在 JavaScript 的方法中，需要注意以下几点。

- **定义标签时，只允许被一个标签包裹** 。例如，`const component = <span>name</span><span> value</span>` 这样写会报错。原因是一个标签会被转译成对应的 `React.createElement` 调用方法，最外层没有被包裹，显然无法转译成方法调用。
- **标签一定要闭合** 。所有标签（比如 `<div></div>` 、`<p></p>` ）都必须闭合，否则无法编译通过。其中 HTML 中自闭合的标签（如 `<img>` ）在 JSX 中也遵循同样规则，自定义标签可以根据是否有子组件或文本来决定闭合方式。

当然，JSX 报错机制非常强大，如果有拼写错误时，可以直接在控制台打印出来。

2. 元素类型

在 1.2 节中，我们讲到两种不同的元素：DOM 元素和组件元素。在 JSX 里自然会有对应，对应规则是 HTML 标签首字母是否为小写字母，其中小写首字母对应 DOM 元素，而组件元素自然对应大写首字母。

比如 List 组件中的 `<div>` 标签会生成 DOM 元素，`Title` 以大写字母开头，会生成组件元素：

```
const Title = (children) => (
  <h3>{children}</h3>
);
```

等到依赖的组件元素中不再出现组件元素，我们就可以将完整的 DOM 树构建出来了。

JSX 还可以通过命名空间的方式使用组件元素，以解决组件相同名称冲突的问题，或是对一组组件进行归类。比如，我们想使用 Material UI 组件库中的组件，以 `MUI` 为包名，可以这么写：

```
const App = () => (
  <MUI.RaisedButton label="Default" />
);
```

在 HTML 标准中，还有一些特殊的标签值得讨论，比如注释和 `DOCTYPE` 头。

- **注释**

  在 HTML 中，注释写成 `<!-- content -->` 这样的形式，但在 JSX 中并没有定义注释的转换方法。事实上，JSX 还是 JavaScript，依然可以用简单的方法使用注释，唯一要注意的是，在一个组件的子元素位置使用注释要用 `{}` 包起来。示例代码如下：

  ```
  const App = (
    <Nav>
      {/* 节点注释 */}
      <Person
        /* 多行
           注释 */
        name={window.isLoggedIn ? window.name : ''}
      />
    </Nav>
  );
  ```