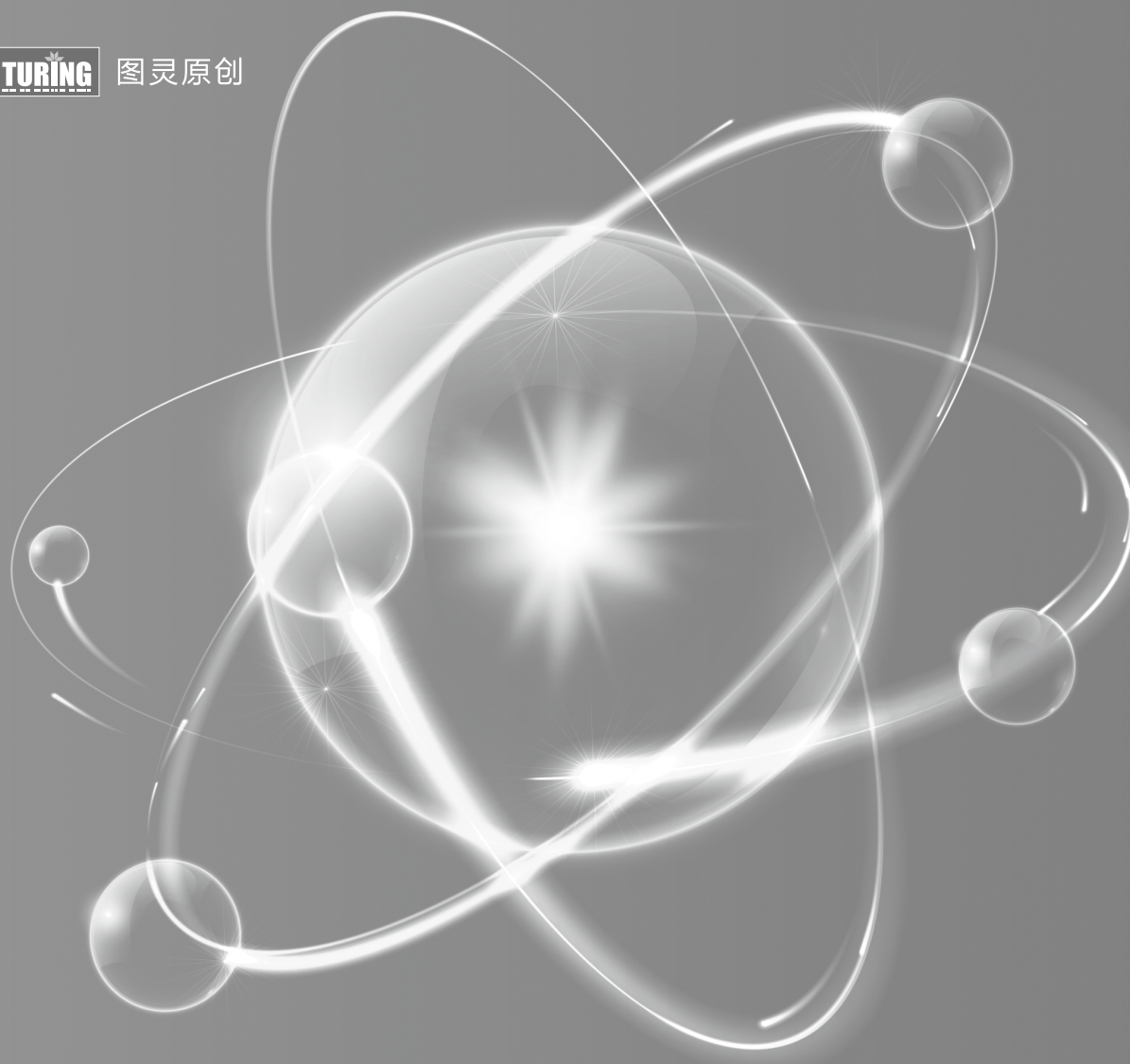




图灵原创



深入React技术栈

陈屹◎著

人民邮电出版社

北 京

图书在版编目 (C I P) 数据

深入React技术栈 / 陈屹著. -- 北京 : 人民邮电出版社, 2016. 11
(图灵原创)
ISBN 978-7-115-43730-3

I. ①深… II. ①陈… III. ①移动终端—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆CIP数据核字(2016)第243308号

内 容 提 要

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 React 与 server 的碰撞产生的一些思考。四是讲述它在可视化方面的优势与劣势。

本书适合有一定经验的前端开发人员阅读。

-
- ◆ 著 陈 屹
责任编辑 王军花
责任印制 彭志环
 - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京 印刷
 - ◆ 开本: 800×1000 1/16
印张: 22.75
字数: 538千字 2016年11月第1版
印数: 1-4 000册 2016年11月北京第1次印刷
-

定价: 79.00元

读者服务热线: (010)51095186转600 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京东工商广字第 8052 号

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

目 录

第 1 章 初入 React 世界	1	2.1.4 合成事件与原生事件混用	51
1.1 React 简介	1	2.1.5 对比 React 合成事件与 JavaScript 原生事件	54
1.1.1 专注视图层	1	2.2 表单	55
1.1.2 Virtual DOM	1	2.2.1 应用表单组件	55
1.1.3 函数式编程	2	2.2.2 受控组件	60
1.2 JSX 语法	3	2.2.3 非受控组件	61
1.2.1 JSX 的由来	3	2.2.4 对比受控组件和非受控组件	62
1.2.2 JSX 基本语法	7	2.2.5 表单组件的几个重要属性	63
1.3 React 组件	11	2.3 样式处理	64
1.3.1 组件的演变	11	2.3.1 基本样式设置	64
1.3.2 React 组件的构建	18	2.3.2 CSS Modules	66
1.4 React 数据流	21	2.4 组件间通信	74
1.4.1 state	21	2.4.1 父组件向子组件通信	74
1.4.2 props	23	2.4.2 子组件向父组件通信	75
1.5 React 生命周期	29	2.4.3 跨级组件通信	77
1.5.1 挂载或卸载过程	29	2.4.4 没有嵌套关系的组件通信	79
1.5.2 数据更新过程	30	2.5 组件间抽象	81
1.5.3 整体流程	33	2.5.1 mixin	81
1.6 React 与 DOM	34	2.5.2 高阶组件	86
1.6.1 ReactDOM	35	2.5.3 组合式组件开发实践	93
1.6.2 ReactDOM 的不稳定方法	36	2.6 组件性能优化	97
1.6.3 refs	38	2.6.1 纯函数	97
1.6.4 React 之外的 DOM 操作	40	2.6.2 PureComponent	100
1.7 组件化实例：Tabs 组件	41	2.6.3 Immutable	103
1.8 小结	47	2.6.4 key	109
第 2 章 漫谈 React	48	2.6.5 react-addons-perf	110
2.1 事件系统	48	2.7 动画	111
2.1.1 合成事件的绑定方式	48	2.7.1 CSS 动画与 JavaScript 动画	111
2.1.2 合成事件的实现机制	49	2.7.2 玩转 React Transition	113
2.1.3 在 React 中使用原生事件	51	2.7.3 缓动函数	116

2.8 自动化测试	121	4.4.3 设计 actionCreator	200
2.8.1 Jest	121	4.4.4 构建 controller-view	202
2.8.2 Enzyme	124	4.4.5 重构 view	203
2.8.3 自动化测试	125	4.4.6 添加单元测试	205
2.9 组件化实例：优化 Tabs 组件	125	4.5 解读 Flux	206
2.10 小结	133	4.5.1 Flux 核心思想	206
第 3 章 解读 React 源码	134	4.5.2 Flux 的不足	207
3.1 初探 React 源码	134	4.6 小结	207
3.2 Virtual DOM 模型	137	第 5 章 深入 Redux 应用架构	208
3.2.1 创建 React 元素	138	5.1 Redux 简介	208
3.2.2 初始化组件入口	140	5.1.1 Redux 是什么	208
3.2.3 文本组件	141	5.1.2 Redux 三大原则	209
3.2.4 DOM 标签组件	144	5.1.3 Redux 核心 API	210
3.2.5 自定义组件	150	5.1.4 与 React 绑定	211
3.3 生命周期的管理艺术	151	5.1.5 增强 Flux 的功能	212
3.3.1 初探 React 生命周期	152	5.2 Redux middleware	212
3.3.2 详解 React 生命周期	152	5.2.1 middleware 的由来	212
3.3.3 无状态组件	163	5.2.2 理解 middleware 机制	213
3.4 解密 setState 机制	164	5.3 Redux 异步流	217
3.4.1 setState 异步更新	164	5.3.1 使用 middleware 简化异步请求	217
3.4.2 setState 循环调用风险	165	5.3.2 使用 middleware 处理复杂异步流	221
3.4.3 setState 调用栈	166	5.4 Redux 与路由	224
3.4.4 初识事务	168	5.4.1 React Router	225
3.4.5 解密 setState	170	5.4.2 React Router Redux	227
3.5 diff 算法	172	5.5 Redux 与组件	229
3.5.1 传统 diff 算法	172	5.5.1 容器型组件	229
3.5.2 详解 diff	172	5.5.2 展示型组件	229
3.6 React Patch 方法	181	5.5.3 Redux 中的组件	230
3.7 小结	183	5.6 Redux 应用实例	231
第 4 章 认识 Flux 架构模式	184	5.6.1 初始化 Redux 项目	231
4.1 React 独立架构	184	5.6.2 划分目录结构	232
4.2 MV* 与 Flux	190	5.6.3 设计路由	234
4.2.1 MVC/MVVM	190	5.6.4 让应用跑起来	235
4.2.2 Flux 的解决方案	193	5.6.5 优化构建脚本	239
4.3 Flux 基本概念	194	5.6.6 添加布局文件	239
4.4 Flux 应用实例	198	5.6.7 准备首页的数据	242
4.4.1 初始化目录结构	198	5.6.8 连接 Redux	245
4.4.2 设计 store	198		

5.6.9 引入 Redux Devtools	250	6.5.5 replaceReducer	288
5.6.10 利用 middleware 实现 Ajax		6.6 解读 react-redux	288
请求发送	251	6.6.1 Provider	288
5.6.11 请求本地的数据	252	6.6.2 connect	290
5.6.12 页面之间的跳转	253	6.6.3 代码热替换	293
5.6.13 优化与改进	256	6.7 小结	294
5.6.14 添加单元测试	257		
5.7 小结	258	第 7 章 React 服务端渲染	295
第 6 章 Redux 高阶运用	259	7.1 React 与服务端模板	295
6.1 高阶 reducer	259	7.1.1 什么是服务端渲染	295
6.1.1 reducer 的复用	259	7.1.2 react-view	296
6.1.2 reducer 的增强	261	7.1.3 react-view 源码解读	296
6.2 Redux 与表单	262	7.2 React 服务端渲染	299
6.2.1 使用 redux-form-utils 减少		7.2.1 玩转 Node.js	300
创建表单的冗余代码	263	7.2.2 React-Router 和 Koa-Router	
6.2.2 使用 redux-form 完成表单的		统一	303
异步验证	265	7.2.3 同构数据处理的探讨	306
6.2.3 使用高阶 reducer 为现有模块		7.3 小结	307
引入表单功能	267	第 8 章 玩转 React 可视化	308
6.3 Redux CRUD 实战	268	8.1 React 结合 Canvas 和 SVG	308
6.3.1 准备工作	268	8.1.1 Canvas 与 SVG	308
6.3.2 使用 Table 组件完成“查”		8.1.2 在 React 中的 Canvas	310
功能	269	8.1.3 React 中的 SVG	311
6.3.3 使用 Modal 组件完成“增”		8.2 React 与可视化组件	316
与“改”	274	8.2.1 包装已有的可视化库	316
6.3.4 巧用 Modal 实现数据的删除		8.2.2 使用 D3 绘制 UI 部分	317
确认	277	8.2.3 使用 React 绘制 UI 部分	319
6.3.5 善用 promise 玩转 Redux 异步		8.3 Recharts 组件化的原理	322
事件流	278	8.3.1 声明式的标签	323
6.4 Redux 性能优化	279	8.3.2 贴近原生的配置项	325
6.4.1 Reselect	280	8.3.3 接口式的 API	326
6.4.2 Immutable Redux	282	8.4 小结	328
6.4.3 Reducer 性能优化	282	附录 A 开发环境	329
6.5 解读 Redux	284	附录 B 编码规范	345
6.5.1 参数归一化	285	附录 C Koa middleware	349
6.5.2 初始状态及 getState	286		
6.5.3 subscribe	286		
6.5.4 dispatch	287		

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，还是经验丰富的资深工程师，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。

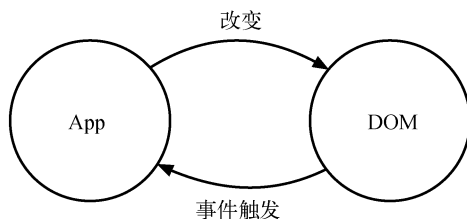


图 1-1 传统 DOM 更新

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。

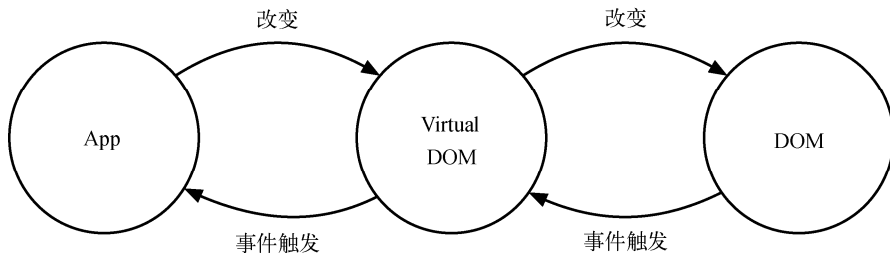


图 1-2 React DOM 更新

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 `f` 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。

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

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('blue', '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 = namevalue` 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 ``）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>  
);
```

但 HTML 中有一类特殊的注释——条件注释，它常用于判断浏览器的版本：

```
<!--[if IE]>  
  <p>Work in IE browser</p>  
<![endif]-->
```

上述方法可以通过使用 JavaScript 判断浏览器版本来替代：

```
{  
  (!!window.ActiveXObject || 'ActiveXObject' in window) ?  
  <p>Work in IE browser</p> : ''  
}
```

一般来说，条件注释的使用场景是在 `<head>` 中判断加载对应的脚本或样式。在服务端渲染中，我们还会遇到这样的场景，在 0.14 版本中可以使用 `<meta>` 标签来实现：

```

<meta dangerouslySetInnerHTML={
  _html: `
    <!--[if IE]>
      <script src="//example.org/app.js"></script>
    <![endif]-->
  `
} />

```

但在 15.0 版本中这已经不可用。因此，还是建议在 JavaScript 里判断浏览器版本，进行一些特有的操作。

● DOCTYPE

DOCTYPE 头是一个非常特殊的标志，一般会在使用 React 作为服务端渲染时用到。在 HTML 中，DOCTYPE 是没有闭合的，也就是说我们无法渲染它。

常见的做法是构造一个保存 HTML 的变量，将 DOCTYPE 与整个 HTML 标签渲染后的结果串连起来。第 7 章会详细讲到。

3. 元素属性

元素除了标签之外，另一个组成部分就是标签的属性。

在 JSX 中，不论是 DOM 元素还是组件元素，它们都有属性。不同的是，DOM 元素的属性是标准规范属性，但有两个例外——`class` 和 `for`，这是因为在 JavaScript 中这两个单词都是关键词。因此，我们这么转换：

- ❑ `class` 属性改为 `className`；
- ❑ `for` 属性改为 `htmlFor`。

而组件元素的属性是完全自定义的属性，也可以理解为实现组件所需要的参数。比如：

```

const Header = (title, children) => (
  <h3 title={title}>{children}</h3>
);

```

我们给 Header 组件加了一个 `title` 属性，那么可以这么调用：

```
<Header title="hello world">Hello world</Header>
```

当然，我们可以再给 Header 组件加上 `color` 等属性。可以看到，Header 和 h3 中两个 `title` 的不同之处，一个代表的是自定义标签的属性可以传递，一个是标签自带的属性无法传递。值得注意的是，在写自定义属性时，都由标准写法改为小驼峰写法。

此外，还有一些 JSX 特有的属性表达。

● Boolean 属性

省略 Boolean 属性值会导致 JSX 认为 `bool` 值设为了 `true`。要传 `false` 时，必须使用属性表达式。这常用于表单元素中，比如 `disabled`、`required`、`checked` 和 `readOnly` 等。

例如, `<Checkbox checked={true} />` 可以简写为 `<Checkbox checked />`, 反之 `<Checkbox checked={false} />` 就可以省略 `checked` 属性。

● 展开属性

如果事先知道组件需要的全部属性, JSX 可以这样来写:

```
const component = <Component name={name} value={value} />;
```

如果你不知道要设置哪些 props, 那么现在最好不要设置它:

```
const component = <Component />;
component.props.name = name;
component.props.value = value;
```

上述这样是反模式, 因为 React 不能帮你检查属性类型 (propTypes)。这样即使组件的属性类型有错误, 也不能得到清晰的错误提示。

这里, 可以使用 ES6 rest/spread 特性来提高效率:

```
const data = { name: 'foo', value: 'bar' };
const component = <Component name={name} value={value} />;
```

可以写成:

```
const data = { name: 'foo', value: 'bar' };
const component = <Component {...data} />;
```

● 自定义 HTML 属性

如果在 JSX 中往 DOM 元素中传入自定义属性, React 是不会渲染的:

```
<div d="xxx">content</div>
```

如果要使用 HTML 自定义属性, 要使用 `data-` 前缀, 这与 HTML 标准也是一致的:

```
<div data-attr="xxx">content</div>
```

然而, 在自定义标签中任意的属性都是被支持的:

```
<x-my-component custom-attr="foo" />
```

以 `aria-` 开头的网络无障碍属性同样可以正常使用:

```
<div aria-hidden={true}></div>
```

不论组件是用什么方法来写, 我们都需要知道, 组件的最终目的是输出虚拟元素, 也就是需要被渲染到界面的结构。其核心渲染方法, 或称为组件输出方法, 就是 `render` 方法。它是 React 组件生命周期的一部分, 也是最核心的函数之一。1.5 节将详细解释整个生命周期的运作。

4. JavaScript 属性表达式

属性值要使用表达式, 只要用 `{}` 替换 `"` 即可:

```
// 输入 (JSX) :
const person = <Person name={window.isLoggedIn ? window.name : ''} />;

// 输出 (JavaScript) :
const person = React.createElement(
  Person,
  {name: window.isLoggedIn ? window.name : ''}
);
```

子组件也可以作为表达式使用：

```
// 输入 (JSX) :
const content = <Container>{window.isLoggedIn ? <Nav /> : <Login />}</Container>;

// 输出 (JavaScript) :
const content = React.createElement(
  Container,
  null,
  window.isLoggedIn ? React.createElement(Nav) : React.createElement(Login)
);
```

5. HTML 转义

React 会将所有要显示到 DOM 的字符串转义，防止 XSS。所以，如果 JSX 中含有转义后的实体字符，比如 `©` (`©`)，则最后 DOM 中不会正确显示，因为 React 自动把 `©` 中的特殊字符转义了。有几种解决办法：

- ❑ 直接使用 UTF-8 字符 `©`；
- ❑ 使用对应字符的 Unicode 编码查询编码；
- ❑ 使用数组组装 `<div>[['cc ', ©, ' 2015']]</div>`；
- ❑ 直接插入原始的 HTML。

此外，React 提供了 `dangerouslySetInnerHTML` 属性。正如其名，它的作用就是避免 React 转义字符，在确定必要的情况下可以使用它：

```
<div dangerouslySetInnerHTML={{__html: 'cc &copy; 2015'}} />
```

1.3 React 组件

终于说到我们最为关心的 React 组件了。在 React 诞生之前，前端界对于组件的封装实现一直都处在摸索和实践的阶段。

1.3.1 组件的演变

在 MV* 架构出现之前，组件主要分为两种。

- ❑ 狭义上的组件，又称为 UI 组件，比如 Tabs 组件、Dropdown 组件。组件主要围绕在交互

动作上的抽象，针对这些交互动作，利用 JavaScript 操作 DOM 结构或 style 样式来控制。

- 广义上的组件，即带有业务含义和数据的 UI 组件组合。这类组件不仅有交互动作，更重要的是有数据与界面之间的交互。然而，这类组件往往有较大的争议。在规模较大的场景下，我们更倾向于采用分层的思想去处理。

以常用的 Tabs 组件为例，对于 UI 组件来说，一定会有 3 个部分组件：结构、样式和交互行为，分别对应着 HTML、CSS 和 JavaScript。一般情况下，我们会先构建组件的基本结构：

```
<div id="tab-demo">
  <div class="tabs-bar" role="tablist">
    <ul class="tabs-nav">
      <li role="tab" class="tabs-tab">Tab 1</li>
      <li role="tab" class="tabs-tab">Tab 2</li>
      <li role="tab" class="tabs-tab">Tab 3</li>
    </ul>
  </div>
  <div class="tabs-content">
    <div role="tabpanel" class="tabs-panel">
      第一个 Tab 里的内容
    </div>
    <div role="tabpanel" class="tabs-panel">
      第二个 Tab 里的内容
    </div>
    <div role="tabpanel" class="tabs-panel">
      第三个 Tab 里的内容
    </div>
  </div>
</div>
```

这个结构对我们来说非常熟悉，其中 `tabs-bar` 中的内容是组件的导航区域，而 `tabs-content` 中的内容自然就是组件的内容区域。利用 JavaScript 和 CSS 来控制对应索引的导航激活，且显示内容区域。

现在，我们就按照图 1-3 来定义组件的样式。



图1-3 组件样式

样式代码如下：

```
$class-prefix: "tabs";

.#{$class-prefix} {
  &-bar {
    margin-bottom: 16px;
  }

  &-nav {
    font-size: 14px;
  }
}
```



```

    &:after,
    &:before {
      display: table;
      content: " ";
    }

    &:after {
      clear: both;
    }
  }

  &-nav > &-tab {
    float: left;
    list-style: none;
    margin-right: 24px;
    padding: 8px 20px;
    text-decoration: none;
    color: #666;
    cursor: pointer;
  }

  &-nav > &-active {
    border-bottom: 2px solid #00C49F;
    color: #00C49F;
    cursor: default;
  }

  &-content &-panel {
    display: none;
  }

  &-content &-active {
    display: block;
  }
}

```

这里我们用 SCSS 来定义组件的样式，这样可以方便地定义 class 前缀，以达到定义一系列组件主题的目的。

最后是交互行为。我们引入 jQuery 方便操作 DOM，使用 ES6 classes 语法糖来替换早期利用原型构建面向对象的方法，以及使用 ES6 modules 替换 AMD 模块加载机制：

```

import $ from 'jquery';
import EventEmitter from 'events';

const Selector = (classPrefix) => ({
  PREFIX: classPrefix,
  NAV: `${classPrefix}-nav`,
  CONTENT: `${classPrefix}-content`,
  TAB: `${classPrefix}-tab`,
  PANEL: `${classPrefix}-panel`,

```

```
    ACTIVE: `${classPrefix}-active`,
    DISABLE: `${classPrefix}-disable`,
  });

class Tabs {
  static defaultOptions = {
    classPrefix: 'tabs',
    activeIndex: 0,
  };

  constructor(options) {
    this.options = $.extend({}, Tabs.defaultOptions, options);
    this.element = $(this.options.element);
    this.fromIndex = this.options.activeIndex;

    this.events = new EventEmitter();
    this.selector = Selector(this.options.classPrefix);

    this._initElement();
    this._initTabs();
    this._initPanels();
    this._bindTabs();

    if (this.options.activeIndex !== undefined) {
      this.switchTo(this.options.activeIndex);
    }
  }

  _initElement() {
    this.element.addClass(this.selector.PREFIX);
    this.tabs = $(this.options.tabs);
    this.panels = $(this.options.panels);
    this.nav = $(this.options.nav);
    this.content = $(this.options.content);

    this.length = this.tabs.length;
  }

  _initTabs() {
    this.nav && this.nav.addClass(this.selector.NAV);
    this.tabs.addClass(this.selector.TAB).each((index, tab) => {
      $(tab).data('value', index);
    });
  }

  _initPanels() {
    this.content.addClass(this.selector.CONTENT);
    this.panels.addClass(this.selector.PANEL);
  }

  _bindTabs() {
    this.tabs.click((e) => {
      const $el = $(e.target);
      if (!$el.hasClass(this.selector.DISABLE)) {
```

```

        this.switchTo($el.data('value'));
    }
});
}

events(name) {
    return this.events;
}

switchTo(toIndex) {
    this._switchTo(toIndex);
}

_switchTo(toIndex) {
    const fromIndex = this.fromIndex;
    const panelInfo = this._getPanelInfo(toIndex);

    this._switchTabs(toIndex);
    this._switchPanel(panelInfo);
    this.events.emit('change', { toIndex, fromIndex });

    this.fromIndex = toIndex;
}

_switchTabs(toIndex) {
    const tabs = this.tabs;
    const fromIndex = this.fromIndex;

    if (tabs.length < 1) return;

    tabs
        .eq(fromIndex)
        .removeClass(this.selector.ACTIVE)
        .attr('aria-selected', false);
    tabs
        .eq(toIndex)
        .addClass(this.selector.ACTIVE)
        .attr('aria-selected', true);
}

_switchPanel(panelInfo) {
    panelInfo.fromPanels
        .attr('aria-hidden', true)
        .hide();
    panelInfo.toPanels
        .attr('aria-hidden', false)
        .show();
}

_getPanelInfo(toIndex) {
    const panels = this.panels;
    const fromIndex = this.fromIndex;

    let fromPanels, toPanels;

```

```
    if (fromIndex > -1) {
      fromPanels = this.panels.slice(fromIndex, (fromIndex + 1));
    }

    toPanels = this.panels.slice(toIndex, (toIndex + 1));

    return {
      toIndex: toIndex,
      fromIndex: fromIndex,
      toPanels: $(toPanels),
      fromPanels: $(fromPanels),
    };
  }

  destroy() {
    this.events.removeAllListeners();
  }
}

export default Tabs;
```

初始化过程十分简洁，实例化组件并传入必要的几个参数就可以赋予交互：

```
const tab = new Tabs({
  element: '#tab-demo',
  tabs: '#tab-demo .tabs-nav li',
  panels: '#tab-demo .tabs-content div',
  activeIndex: 1,
});

tab.events.on('change', (o) => {
  console.log(o);
});
```

我们看到，组件封装的基本思路就是面向对象思想。交互基本上以操作 DOM 为主，逻辑上是结构上哪里需要变，我们就操作哪里。此外，对于 JavaScript 的结构，我们得到了几项规范标准组件的信息。

- ❑ **基本的封装性。**尽管说 JavaScript 没有真正面向对象的方法，但我们还是可以通过实例化的方法来制造对象。
- ❑ **简单的生命周期呈现。**最明显的两个方法 `constructor` 和 `destroy`，代表了组件的挂载和卸载过程。但除此之外，其他过程（如更新时的生命周期）并没有体现。
- ❑ **明确的数据流动。**这里的数据指的是调用组件的参数。一旦确定参数的值，就会解析传进来的参数，根据参数的不同作出不同的响应，从而得到渲染结果。

在这个阶段，前端在应用级别并没有过多复杂的交互，组件化发展缓慢。传统组件的主要问题在于结构、样式与行为没有很好地结合，不同参数下的逻辑可能会导致不同的渲染逻辑，这时就会存在大量 HTML 结构与 style 样式的拼装。比如，常见的 `show`、`hide` 与 `toggle` 方法，就

是通过改变 `class` 控制 `style` 来显示或隐藏。这样的逻辑一旦复杂，就存在大量的 DOM 操作，开发及维护成本相当高。

直到富客户端应用越来越多，传统组件化越来越无法满足开发者的需要，于是引进了分层的思想，此时就出现了 MVC 架构。View 只关心怎么输出变量，所以就诞生了各种各样的模板语言，比如 Smarty、Mustache、Handlebars 等。我们结合 Backbone 这样的架构一起使用。让模板本身可以承载逻辑，可以帮我们解决 View 上的逻辑问题。对于组件来说，可以减轻拼装 HTML 的逻辑部分，将这一部分解耦出去，解决了数据与界面耦合的问题。这时利用模板引擎可以在一定程度上实现组件化，不过这种组件化实现的还是字符串拼接级别的组件化。

对于模板，它更接近 HTML 表达方式，能更好地反映应用的语义结构，且易于从设计、布局和样式上思考，但模板作为一个 DSL，也有其局限性。我们需要重新思考到底什么才是组件的组成。直到这几年萌生的 Angular，我们看到了在 HTML 上定义指令的方式。

W3C 标准委员会最近才将类似的思想制定成了规范，称为 Web Components。顾名思义，这个规范是想统一 Web 端关于组件的定义。它通过定义 Custom Elements（自定义元素）的方式来统一组件。每个自定义元素可以定义自己对外提供的属性、方法，还有事件，内部可以像写一个页面一样，专注于实现功能来完成对组件的封装。

图 1-4 讲述了 Web Components 的 4 个组成部分：HTML Templates 定义了之前模板的概念，Custom Elements 定义了组件的展现形式，Shadow DOM 定义了组件的作用域范围、可以囊括样式，HTML Imports 提出了新的引入方式。

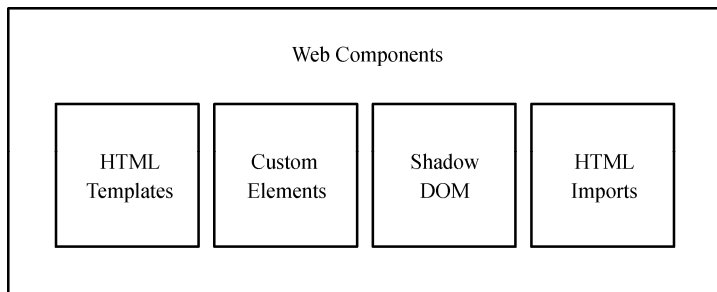


图 1-4 Web Components 组成

Web Components 定义了一切我们想要的组件化概念，现在还有 polymer 这个库可实现这一套理念，但事实上它还需要时间的考验。因为诸如如何包装在这套规范之上的框架，如何获得在浏览器端的全部支持、怎么与现代应用架构相结合等问题，目前都还没有统一的解法。但 Web Components 的确为组件化开辟了一条罗马大道，告诉了我们组件化可以这样做。

再说回 React，它的组件化是什么，又是怎么样构建的呢？

1.3.2 React 组件的构建

Web Components 通过自定义元素的方式实现组件化，而 React 的本质就是关心元素的构成，React 组件即为组件元素。组件元素被描述成纯粹的 JSON 对象，意味着可以使用方法或是类来构建。React 组件基本上由 3 个部分组成——属性（props）、状态（state）以及生命周期方法。这里我们从一张图来简单概括 React，如图 1-5 所示。

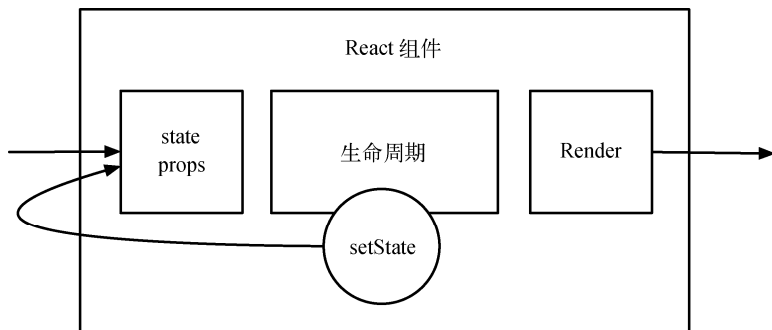


图 1-5 React 组件的组成

React 组件可以接收参数，也可能有自身状态。一旦接收到的参数或自身状态有所改变，React 组件就会执行相应的生命周期方法，最后渲染。整个过程完全符合传统组件所定义的组件职责。

1. React 与 Web Components

从 React 组件上看，它与 Web Components 传达的理念是一致的，但两者的实现方式不同：

- ❑ React 自定义元素是库自己构建的，与 Web Components 规范并不通用；
- ❑ React 渲染过程包含了模板的概念，即 1.2 节所讲的 JSX；
- ❑ React 组件的实现均在方法与类中，因此可以做到相互隔离，但不包括样式；
- ❑ React 引用方式遵循 ES6 module 标准。

可以说，React 还是在纯 JavaScript 上下了工夫，将 HTML 结构彻底引入到 JavaScript 中。尽管这种做法褒贬不一，但也有效解决了组件所要解决的问题之一。

2. React 组件的构建方法

React 组件基本上由组件的构建方式、组件内的属性状态与生命周期方法组成。在本节中，我们先来讨论创建 React 组件的构建方式，而属性状态与生命周期会在后面再介绍。

官方在 React 组件构建上提供了 3 种不同的方法：`React.createClass`、ES6 classes 和无状态函数（stateless function）。我们使用 1.1 节中的 Button 来分别介绍这 3 种方法。

● `React.createClass`

用 `React.createClass` 构建组件是 React 最传统、也是兼容性最好的方法。在 0.14 版本发布

之前，这一直都是 React 官方唯一指定的组件写法。示例代码如下：

```
const Button = React.createClass({
  getDefaultProps() {
    return {
      color: 'blue',
      text: 'Confirm',
    };
  },

  render() {
    const { color, text } = this.props;

    return (
      <button className={`btn btn-${color}`}>
        <em>{text}</em>
      </button>
    );
  }
});
```

从表象上看，`React.createClass` 方法就是构建一个组件对象。当另一个组件需要调用 `Button` 组件时，只用写成 `<Button />`，就可以被解析成 `React.createElement(Button)` 方法来创建 `Button` 实例，这意味着在一个应用中调用几次 `Button`，就会创建几次 `Button` 实例。

● ES6 classes

ES6 classes 的写法是通过 ES6 标准的类语法的方式来构建方法：

```
import React, { Component } from 'react';

class Button extends Component {
  constructor(props) {
    super(props);
  }

  static defaultProps = {
    color: 'blue',
    text: 'Confirm',
  }

  render() {
    return (
      <button className={`btn btn-${color}`}>
        <em>{text}</em>
      </button>
    );
  }
}
```

这里的直观感受是从调用内部方法变成了用类来实现。与 `createClass` 的结果相同的是，调用类实现的组件会创建实例对象。

再说起继承，我们很容易联想到在组件抽象过程中也可以使用继承的思路。如果我们学过面向对象的知识，就知道继承与组合的不同，它们可以用 IS-A 与 HAS-A 来区别。在实际应用 React 的过程中，我们极少让子类去继承功能组件。试想在 UI 层面小的修改就会影响到整体交互或样式，牵一发而动全身，用继承来抽象往往是事倍功半。在 React 组件开发中，常用的方式是将组件拆分到合理的粒度，用组合的方式合成业务组件，也就是 HAS-A 的关系。但在高阶组件构建中，我们可以用反向继承的方法来实现，具体内容请阅读 2.5 节。

说明 React 的所有组件都继承自顶层类 `React.Component`。它的定义非常简洁，只是初始化了 `ReactComponent` 方法，声明了 `props`、`context`、`refs` 等，并在原型上定义了 `setState` 和 `forceUpdate` 方法。内部初始化的生命周期方法与 `createClass` 方式使用的是同一个方法创建的。具体解读可参见 3.2.2 节。

● 无状态函数

使用无状态函数构建的组件称为无状态组件，这种构建方式是 0.14 版本之后新增的，且官方颇为推崇。示例代码如下：

```
function Button({ color = 'blue', text = 'Confirm' }) {
  return (
    <button className={`btn btn-${color}`}>
      <em>{text}</em>
    </button>
  );
}
```

无状态组件只传入 `props` 和 `context` 两个参数；也就是说，它不存在 `state`，也没有生命周期方法，组件本身即上面两种 React 组件构建方法中的 `render` 方法。不过，像 `propTypes` 和 `defaultProps` 还是可以通过向方法设置静态属性来实现的。

在适合的情况下，我们都应该且必须使用无状态组件。无状态组件不像上述两种方法在调用时会创建新实例，它创建时始终保持了一个实例，避免了不必要的检查和内存分配，做到了内部优化。

3. 用 React 实现 Tabs 组件

这里我们趁热打铁，运用已经掌握的组件构建方法来实现一个组件。首先，用 ES6 classes 的写法来初始化 Tabs 组件的“骨架”：

```
import React, { Component, PropTypes } from 'react';

class Tabs extends Component {
  constructor(props) {
    super(props);
  }
}
```



```
// ...  
  
render() {  
  return <div className="ui-tabs"></div>;  
}  
};  
  
export default Tabs;
```

从这一节起，我们就以 Tabs 组件为例慢慢介绍 React 组件的主要组成部分，看看它到底有什么不同之处。

1.4 React 数据流

在 React 中，数据是自顶向下单向流动的，即从父组件到子组件。这条原则让组件之间的关系变得简单且可预测。

state 与 props 是 React 组件中最重要的概念。如果顶层组件初始化 props，那么 React 会向下遍历整棵组件树，重新尝试渲染所有相关的子组件。而 state 只关心每个组件自己内部的状态，这些状态只能在组件内改变。把组件看成一个函数，那么它接受了 props 作为参数，内部由 state 作为函数的内部参数，返回一个 Virtual DOM 的实现。

1.4.1 state

在使用 React 之前，常见的 MVC 框架也非常容易实现交互界面的状态管理，比如 Backbone。它们将 View 中与界面交互的状态解耦，一般将状态放在 Model 中管理。但在 React 没有结合 Flux 或 Redux 框架前，它自身也同样可以管理组件的内部状态。在 React 中，把这类状态统一称为 state。

当组件内部使用库内置的 `setState` 方法时，最大的表现行为就是该组件会尝试重新渲染。这很好理解，因为我们改变了内部状态，组件需要更新了。比如，我们实现了一个计数器组件：

```
import React, { Component } from 'react';  
  
class Counter extends Component {  
  constructor(props) {  
    super(props);  
  
    this.handleClick = this.handleClick.bind(this);  
  
    this.state = {  
      count: 0,  
    };  
  }  
}
```

```
handleClick(e) {
  e.preventDefault();

  this.setState({
    count: count + 1,
  });
}

render() {
  return (
    <div>
      <p>{this.state.count}</p>
      <a href="#" onClick={this.handleClick}>更新</a>
    </div>
  );
}
```

在 React 中常常在事件处理方法中更新 state，上述例子就是通过点击“更新”按钮不断地更新内部 count 的值，这样就可以把组件内状态封装在实现中。

值得注意的是，setState 是一个异步方法，一个生命周期内所有的 setState 方法会合并操作。关于 setState 的实现原理，请参见 3.4 节。

有了这个特性，让 React 变得充满了想象力。我们完全可以只用 React 来完成对行为的控制、数据的更新和界面的渲染。然而，随着内容的深入，我们并不推荐开发者滥用 state，过多的内部状态会让数据流混乱，程序变得难以维护。

我们再来看 Tabs 组件的 state。从前一节的经验中得到两个可能的内部状态——activeIndex 和 prevIndex，它们分别表示当前选中 tab 的索引和上一次选中 tab 的索引。而需要特别注意的一点是，当前选中的索引亦是组件本身需要的参数之一。

这里我们针对 activeIndex 作为 state，就有两种不同的视角。

- ❑ **activeIndex 在内部更新。**当我们切换 tab 标签时，可以看作是组件内部的交互行为，被选择后通过回调函数返回具体选择的索引。
- ❑ **activeIndex 在外部更新。**当我们切换 tab 标签时，可以看作是组件外部在传入具体的索引，而组件就像“木偶”一样被操控着。

这两种情形在 React 组件的设计中非常常见，我们形象地把第一种和第二种视角写成的组件分别称为智能组件（smart component）和木偶组件（dumb component）。

当然，实现组件时，可以同时考虑兼容这两种。我们来看下 Tabs 组件中初始化时的实现部分：

```
constructor(props) {
  super(props);

  const currProps = this.props;

  let activeIndex = 0;
```

```

if ('activeIndex' in currProps) {
  activeIndex = currProps.activeIndex;
} else if ('defaultActiveIndex' in currProps) {
  activeIndex = currProps.defaultActiveIndex;
}

this.state = {
  activeIndex,
  prevIndex: activeIndex,
};
}

```

这里我们定义了两种 state——`activeIndex` 和 `prevIndex`。

对于 `activeIndex` 来说，既可能来源于使用内部更新的 `defaultActiveIndex` prop，即我们不需要外组件控制组件状态，也可能来源于需要外部更新的 `activeIndex` prop。如图 1-6 所示，我们只能通过切换外组件的状态来更新。



图1-6 切换组件的状态

不过，不论组件是内部更新还是外部更新，我们都需要 `activeIndex` 这个 state 来更新渲染。那么，如何做到外部更新时让状态更新呢？这个问题会在 1.6 节中详解。

这里，我们反复提到的 props 是不是就是指传入参数呢？继续看下一节。

1.4.2 props

props 是 React 中另一个重要的概念，它是 `properties` 的缩写。props 是 React 用来让组件之间互相联系的一种机制，通俗地说就像方法的参数一样。在 1.2 节中，我们已经接触过它们了。

props 的传递过程，对于 React 组件来说是非常直观的。React 的单向数据流，主要的流动管道就是 props。props 本身是不可变的。当我们试图改变 props 的原始值时，React 会报出类型错误的警告，组件的 props 一定来自于默认属性或通过父组件传递而来。如果说要渲染一个对 props 加工后的值，最简单的方法就是使用局部变量或直接在 JSX 中计算结果。

我们在 1.4.1 节中讨论了 Tabs 组件 state 的设置情况。假设 Tabs 组件的数据都是通过 `data` prop 传入的，即 `<Tabs data={data} />`。那么，Tabs 组件的 props 还会有哪些。根据之前的经验，它一定会有以下几项。

- ❑ `className`: 根节点的 `class`。为了方便覆盖其原始样式，我们都会在根节点上定义 `class`，这一点会在 2.3 节中详细说明。

- ❑ **classPrefix**: **class** 前缀。对于组件来说, 定义一个统一的 **class** 前缀, 对样式与交互分离起了非常重要的作用。
- ❑ **defaultActiveIndex** 和 **activeIndex**: 默认的激活索引, 这在 1.4.1 节中已说明。
- ❑ **onChange**: 回调函数。当我们切换 **tab** 时, 外组件需要知道组件内部的信息, 尤其是当前 **tab** 的索引号的信息。它一般与 **activeIndex** 搭配使用。

React 为 **props** 同样提供了默认配置, 通过 **defaultProps** 静态变量的方式来定义。当组件被调用的时候, 默认值保证渲染后始终有值。在 **render** 方法中, 可以直接使用 **props** 的值来渲染。这里, 我们只需要默认设置 **classPrefix** 和 **onChange** 即可。因为 **defaultActiveIndex** 和 **activeIndex** 我们需要保持只取其中一个条件。相关代码如下:

```
static defaultProps = {  
  classPrefix: 'tabs',  
  onChange: () => {},  
};
```

但 **Tabs** 组件的信息全由一个对象传进来的方式真的好么? 对于 **React** 组件来说, 我们考虑设计组件一定要满足一大原则——直观。把基本设置与数据一起定义成一个数组或对象是初学者最容易犯的一个错误, 如果说组件能够分解, 那我们一定要分解, 并使用子组件的方式来处理。

再一次仔细观察 **Tabs** 组件在 **Web** 界面的特征, 一般来说, 会看到两个区域: 切换区域与内容区域。那么, 我们就定义两个子组件, 其中 **TabNav** 组件对应切换区域, **TabContent** 组件对应内容区域。这两个区域组件都存放了一个有序数组, 都可以进一步拆分。到这里, 我们就想得到两种组织的方式。

- ❑ 在 **Tabs** 组件内把所有定义的子组件都显式展示出来。这种方式的好处在于非常易于理解, 可自定义能力强, 但调用过程显得过于笨重。**React-Bootstrap** 和 **Material UI** 组件库中的 **Tabs** 组件采用的是这种形式。调用方式近似如下形式:

```
<Tabs classPrefix='tabs' defaultActiveIndex={0}>  
  <TabNav>  
    <TabHead>Tab 1</TabHead>  
    <TabHead>Tab 2</TabHead>  
    <TabHead>Tab 3</TabHead>  
  </TabNav>  
  <TabContent>  
    <TabPane>第一个 Tab 里的内容</TabPane>  
    <TabPane>第二个 Tab 里的内容</TabPane>  
    <TabPane>第三个 Tab 里的内容</TabPane>  
  </TabContent>  
</Tabs>
```

- ❑ 在 **Tabs** 组件内只显示定义内容区域的子组件集合, 头部区域对应内部区域每一个 **TabPane** 组件的 **props**, 让其在 **TabNav** 组件内拼装。这种方式的调用写法简洁, 把复杂的逻辑留

给了组件去实现。Ant Design 组件库中的 Tabs 组件采用的就是这种形式。调用方式近似如下形式：

```
<Tabs classPrefix={'tabs'} defaultActiveIndex={0}>
  <TabPane key={0} tab={'Tab 1'}>第一个 Tab 里的内容</TabPane>
  <TabPane key={1} tab={'Tab 2'}>第二个 Tab 里的内容</TabPane>
  <TabPane key={2} tab={'Tab 3'}>第三个 Tab 里的内容</TabPane>
</Tabs>
```

在本章中，我们通过后一种方式讲解。基本的结构确定后，我们需要想一下怎么渲染这个结构的内容。显然，并不是所有参数都由 Tabs 组件承载。只有两个 props 放在了 Tabs 组件上，而其他参数直接放到 TabPane 组件中，由它的父组件 TabContent 隐式对 TabPane 组件拼装。

那么，这个一直在说的子组件是什么呢，我们到底怎么对它进行拼装渲染呢？

1. 子组件 prop

在 React 中有一个重要且内置的 prop——children，它代表组件的子组件集合。children 可以根据传入子组件的数量来决定是否是数组类型。上述调用 TabPane 组件的过程，翻译过来即是：

```
<Tabs classPrefix={'tabs'} defaultActiveIndex={0} className="tabs-bar"
  children=[
    <TabPane key={0} tab={'Tab 1'}>第一个 Tab 里的内容</TabPane>,
    <TabPane key={1} tab={'Tab 2'}>第二个 Tab 里的内容</TabPane>,
    <TabPane key={2} tab={'Tab 3'}>第三个 Tab 里的内容</TabPane>,
  ]
>
</Tabs>
```

实现的基本思路就以 TabContent 组件渲染 TabPane 子组件集合为例来讲，其中渲染 TabPane 组件的方法如下：

```
getTabPanels() {
  const { classPrefix, activeIndex, panels, isActive } = this.props;

  return React.Children.map(panels, (child) => {
    if (!child) { return; }

    const order = parseInt(child.props.order, 10);
    const isActive = activeIndex === order;

    return React.cloneElement(child, {
      classPrefix,
      isActive,
      children: child.props.children,
      key: `tabpanel-${order}`,
    });
  });
}
```

上述代码讲述了子组件集合是怎么渲染的。通过 React.Children.map 方法遍历子组件，

将 `order` (渲染顺序)、`isActive` (是否激活 `tab`)、`children` (`Tabs` 组件中传下的 `children`) 和 `key` 利用 `React` 的 `cloneElement` 方法克隆到 `TabPane` 组件中, 最后返回这个 `TabPane` 组件集合。这也是 `Tabs` 组件拼装子组件的基本原理。

其中, `React.Children` 是 `React` 官方提供的一系列操作 `children` 的方法。它提供诸如 `map`、`forEach`、`count` 等实用函数, 可以为我们处理子组件提供便利。

最后, `TabContent` 组件的 `render` 方法只需要调用 `getTabPanes` 方法即可渲染:

```
render() {
  return (<div>{this.getTabPanes()}</div>);
}
```

假如我们把 `render` 方法中的 `this.getTabPanes` 方法中对子组件的遍历直接放进去, 就会变成如下形式:

```
render() {
  return (<div>{React.Children.map(children, (child) => {...})}</div>);
}
```

这种调用方式称为 `Dynamic Children` (动态子组件)。它指的是组件内的子组件是通过动态计算得到的。就像上述对子组件的遍历一样, 我们一样可以对任何数据、字符串、数组或对象作动态计算。

用声明式编程的方式来渲染数据, 这种做法和关心所有细节的命令式编程相比, 会让我们轻松许多。当然, 除了数组的 `map` 函数, 还可以用其他实用的高阶函数, 如 `reduce`、`filter` 等函数。值得注意的是, 与 `map` 函数相似但不返回调用结果的 `forEach` 函数不能这么使用。

2. 组件 props

当然, `React` 的强大之处不止于此, 我们观察 `TabPane` 组件中的 `tab` prop:

```
<TabPane key={0} tab={'Tab 1'}>第一个 Tab 里的内容</TabPane>
```

它现在传入的是一个字符串。那么, 假如可以传入节点呢, 是不是就可以自定义 `tab` 头展示的形式了。这就是 `component props`。对于子组件而言, 我们不仅可以直接使用 `this.props.children` 定义, 也可以将子组件以 `props` 的形式传递。一般我们会用这种方法来让开发者定义组件的某一个 `prop`, 让其具备多种类型, 来做到简单配置和自定义配置组合在一起的效果。

在 `Tabs` 组件中, 我们就用到了这样的功能, 调用方式如下所示:

```
<Tabs classPrefix={'tabs'} defaultActiveIndex={0} className="tabs-bar">
  <TabPane
    order="0"
    tab={<span><i className="fa fa-home"></i>&nbsp;Home</span><}>
      第一个 Tab 里的内容
    </TabPane>
  <TabPane
    order="1"
```

```

      tab={<span><i className="fa fa-book"></i>&nbsp;Library</span>>
        第二个 Tab 里的内容
      </TabPane>
      <TabPane
        order="2"
        tab={<span><i className="fa fa-pencil"></i>&nbsp;Applications</span>>
          第三个 Tab 里的内容
        </TabPane>
      </Tabs>

```

这里我们使用 font-awesome 的图标。渲染后，每一个 tab 上的文字前都会有一个图标，如图 1-7 所示。



图1-7 文字前加上了图标

当然，我们也可以加入更多的自定义元素，可以是多行的，甚至可以插入动态数据。这听上去有些复杂，但实现过程其实非常简单。下面是写在 TabNav 组件中简化的渲染子组件集合的方法：

```

getTabs() {
  const { classPrefix, activeIndex, panels } = this.props;

  return React.Children.map(panels, (child) => {
    if (!child) { return; }

    const order = parseInt(child.props.order, 10);

    let classes = classNames({
      [`${classPrefix}-tab`]: true,
      [`${classPrefix}-active`]: activeIndex === order,
      [`${classPrefix}-disabled`]: child.props.disabled,
    });

    return (
      <li>{child.props.tab}</li>
    );
  });
}

```

其实现看上去与 getTabPanels 方法非常像，关键在于通过遍历 TabPane 组件的 tab prop 来实现我们想要的功能。不论 tab 是以字符串的形式还是以虚拟元素的形式存在，都可以直接在 标签中渲染出来。

3. 用 function prop 与父组件通信

现在我们发现对于 state 来说，它的通信集中在组件内部；对于 props 来说，它的通信是父组件向子组件的传播。相关代码如下：

```

handleTabClick(activeIndex) {
  // ...

```

```

    this.props.onChange({activeIndex, prevIndex});
  }

```

我们通过点击事件 `handleTabClick` 触发了 `onChange` prop 回调函数给父组件必要的值。对于兄弟组件或不相关组件之间的通信，具体请看 2.4 节。

4. propTypes

众所周知，JavaScript 不是强类型语言，我们对在没有保证的环境下写 JavaScript 已经习以为常了。强类型还是弱类型，正是一个开发时的约束问题。React 对此作了妥协，便有了 `propTypes`。

`propTypes` 用于规范 props 的类型与必需的状态。如果组件定义了 `propTypes`，那么在开发环境下，就会对组件的 props 值的类型作检查，如果传入的 props 不能与之匹配，React 将实时在控制台里报 warning。在生产环境下，这是不会进行检查的。

我们来分析下 Tabs 组件中的情况，并写出对应的 `propTypes`。Tabs 组件包括父组件 Tabs 与子组件 TabPane，下面我们分开来讨论两者的 `propTypes`。现在，我们先来看 Tabs 组件的 `propTypes`：

```

static propTypes = {
  classPrefix: React.PropTypes.string,
  className: React.PropTypes.string,
  defaultActiveIndex: React.PropTypes.number,
  activeIndex: React.PropTypes.number,
  onChange: React.PropTypes.func,
  children: React.PropTypes.oneOfType([
    React.PropTypes.arrayOf(React.PropTypes.node),
    React.PropTypes.node,
  ]),
};

```

我们很清晰地列举了所有可能的 props，并对它们的类型进行定义。再来看看 TabPane 组件的 `propTypes`：

```

static propTypes = {
  tab: React.PropTypes.oneOfType([
    React.PropTypes.string,
    React.PropTypes.node,
  ]).isRequired,
  order: React.PropTypes.string.isRequired,
  disable: React.PropTypes.bool,
};

```

在 TabPane 组件的 props 中，对 `tab` 和 `order` prop 除了定义类型，还定义了是否必要。因此，如果在写 TabPane 组件时，没有定义 `order` prop，浏览器就会主动报一个类型错误的提示：

```
Warning: Failed propType: Required prop `order` was not specified in `TabPane`.
```

值得注意的是，在 `propTypes` 支持的基本类型中，函数类型的检查是 `propTypes.func`，而不是 `propTypes.function`。对于布尔类型的检查是 `propTypes.bool`，而不是 `propTypes.boolean`。这是因为 `function` 和 `boolean` 在 JavaScript 里是关键词。

`propTypes` 有很多类型支持，不仅有基本类型，还包括枚举和自定义类型。

1.5 React 生命周期

生命周期（life cycle）的概念广泛运用于各行各业。从广义上来说，生命周期泛指自然界和人类社会中各种客观事物的阶段性变化及其规律。自然界的生命周期，可分为出生、成长、成熟、衰退直到死亡。而不同体系下的生命周期又都可以从上述规律中演化出来，运用到软件开发的生命周期上，这二者看似相似，事实上又有所不同。生命体的周期是单一方向不可逆的过程，而软件开发的生命周期会根据方法的不同，在完成前重新开始。

React 组件的生命周期根据广义定义描述，可以分为挂载、渲染和卸载这几个阶段。当渲染后的组件需要更新时，我们会重新去渲染组件，直至卸载。

因此，我们可以把 React 生命周期分成两类：

- ❑ 当组件在挂载或卸载时；
- ❑ 当组件接收新的数据时，即组件更新时。

1.5.1 挂载或卸载过程

下面我们简要介绍一下组件的挂载和卸载过程。

1. 组件的挂载

组件挂载是最基本的过程，这个过程主要做组件状态的初始化。我们推荐以下面的例子为模板写初始化组件：

```
import React, { Component, PropTypes } from 'react';

class App extends Component {
  static propTypes = {
    // ...
  };

  static defaultProps = {
    // ...
  };

  constructor(props) {
    super(props);

    this.state = {
      // ...
    };
  }

  componentWillMount() {
    // ...
  }
}
```

```
componentDidMount() {  
  // ...  
}  
  
render() {  
  return <div>This is a demo.</div>;  
}  
}
```

我们看到 `propTypes` 和 `defaultProps` 分别代表 `props` 类型检查和默认类型。这两个属性被声明成静态属性,意味着从类外面也可以访问它们,比如可以这么访问: `App.propTypes` 和 `App.defaultProps`。

之后会看到两个明显的生命周期方法,其中 `componentWillMount` 方法会在 `render` 方法之前执行,而 `componentDidMount` 方法会在 `render` 方法之后执行,分别代表了渲染前后的时刻。

这个初始化过程没什么特别的,包括读取初始 `state` 和 `props` 以及两个组件生命周期方法 `componentWillMount` 和 `componentDidMount`,这些都只会在组件初始化时运行一次。

如果我们在 `componentWillMount` 中执行 `setState` 方法,会发生什么呢?组件会更新 `state`,但组件只渲染一次。因此,这是无意义的执行,初始化时的 `state` 都可以放在 `this.state`。

如果我们在 `componentDidMount` 中执行 `setState` 方法,又会发生什么呢?组件当然会再次更新,不过在初始化过程就渲染了两次组件,这并不是一件好事。但实际情况是,有一些场景不需要 `setState`,比如计算组件的位置或宽高时,就不得不让组件先渲染,更新必要的信息后,再次渲染。

2. 组件的卸载

组件卸载非常简单,只有 `componentWillUnmount` 这一个卸载前状态:

```
import React, { Component, PropTypes } from 'react';  
  
class App extends Component {  
  componentWillMount() {  
    // ...  
  }  
  
  render() {  
    return <div>This is a demo.</div>;  
  }  
}
```

在 `componentWillUnmount` 方法中,我们常常会执行一些清理方法,如事件回收或是清除定时器。

1.5.2 数据更新过程

更新过程指的是父组件向下传递 `props` 或组件自身执行 `setState` 方法时发生的一系列更新动作。这里我们屏蔽了初始化的生命周期方法,以便观察更新过程的生命周期:

```
import React, { Component, PropTypes } from 'react';

class App extends Component {
  componentWillMount(nextProps) {
    // this.setState({})
  }

  shouldComponentUpdate(nextProps, nextState) {
    // return true;
  }

  componentWillUpdate(nextProps, nextState) {
    // ...
  }

  componentDidUpdate(prevProps, prevState) {
    // ...
  }

  render() {
    return <div>This is a demo.</div>;
  }
}
```

如果组件自身的 state 更新了，那么会依次执行 `shouldComponentUpdate`、`componentWillUpdate`、`render` 和 `componentDidUpdate`。

`shouldComponentUpdate` 是一个特别的方法，它接收需要更新的 props 和 state，让开发者增加必要的条件判断，让其在需要时更新，不需要时不更新。因此，当方法返回 `false` 的时候，组件不再向下执行生命周期方法。

`shouldComponentUpdate` 的本质是用来进行正确的组件渲染。怎么理解呢？我们需要先从初始化组件的过程开始说起，假设有如图 1-8 所示的组件关系，它呈三级的树状结构，其中空心圆表示已经渲染的节点。

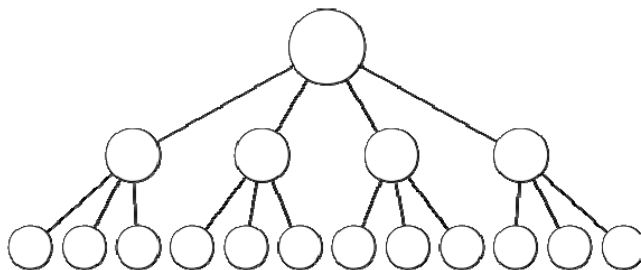


图 1-8 初始化渲染结构

当父节点 props 改变的时候，在理想情况下，只需渲染在一条链路上有相关 props 改变的节点即可，如图 1-9 所示。

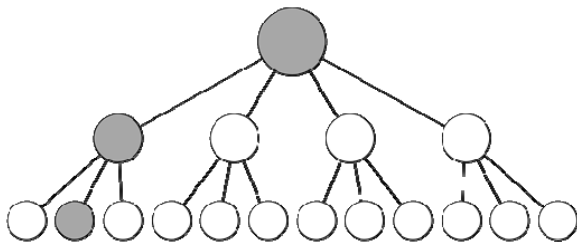


图 1-9 props 改变时 React 节点的渲染路径

而默认情况下，React 会渲染所有的节点，因为 `shouldComponentUpdate` 默认返回 `true`。正确的组件渲染从另一个意义上说，也是性能优化的手段之一。

值得注意的是，无状态组件是没有生命周期方法的，这也意味着它没有 `shouldComponentUpdate`。渲染到该类组件时，每次都会重新渲染。当然，不少开发者在使用无状态组件时会纠结这一点。为了更放心地使用，我们可以选择引用 `Recompose` 库的 `pure` 方法：

```
const OptimizedComponent = pure(ExpensiveComponent);
```

事实上，`pure` 方法做的事就是将无状态组件转换成 `class` 语法加上 `PureRender` 后的组件。关于性能优化相关的内容，我们会在 2.6 节中详解。

`componentWillUpdate` 和 `componentDidUpdate` 这两个生命周期方法很容易理解，对应的初始化方法也很容易知道，它们代表在更新过程中渲染前后的时刻。此时，我们可以想到 `componentWillUpdate` 方法提供需要更新的 `props` 和 `state`，而 `componentDidUpdate` 提供更新前的 `props` 和 `state`。

这里需要注意的是，你不能在 `componentWillUpdate` 中执行 `setState`。如果你对此很感兴趣，想一探究竟，可以直接跳至 3.3 节，那里有更加深入的解释。

如果组件是由父组件更新 `props` 而更新的，那么在 `shouldComponentUpdate` 之前会先执行 `componentWillReceiveProps` 方法。此方法可以作为 React 在 `props` 传入后，渲染之前 `setState` 的机会。在此方法中调用 `setState` 是不会二次渲染的。

回想之前介绍 `Tabs` 组件实现时留下的一个问题：如果 `Tabs` 组件的 `activeIndex` `prop` 只由外组件来更新，那是怎么做到的呢？秘密就在 `componentWillReceiveProps` 方法上，相关代码如下：

```
componentWillReceiveProps(nextProps) {  
  if ('activeIndex' in nextProps) {  
    this.setState({  
      activeIndex: nextProps.activeIndex,  
    });  
  }  
}
```

这样的设置就是让传入的 `props` 判断是否存在 `activeIndex`。如果用了 `activeIndex` 初始化组件，那么每次组件更新前都会去更新组件内部的 `activeIndex` `state`，达到更新组件的目的。

然后，在 tab 点击事件上，对是否存在 `defaultActiveIndex` prop 进行判断即可达到在传入 `defaultActiveIndex` 时使用内部更新，当传入 `activeIndex` 时使用外部传入的 props 更新。相关代码如下：

```
handleTabClick(activeIndex) {
  const prevIndex = this.state.activeIndex;

  if (this.state.activeIndex !== activeIndex &&
    'defaultActiveIndex' in this.props) {
    this.setState({
      activeIndex,
      prevIndex,
    });
  }

  this.props.onChange([ activeIndex, prevIndex ]);
}
}
```

1.5.3 整体流程

我们用一张流程图（如图 1-10 所示）来理清生命周期方法之间的关系，以及关键 API 调用的反馈。

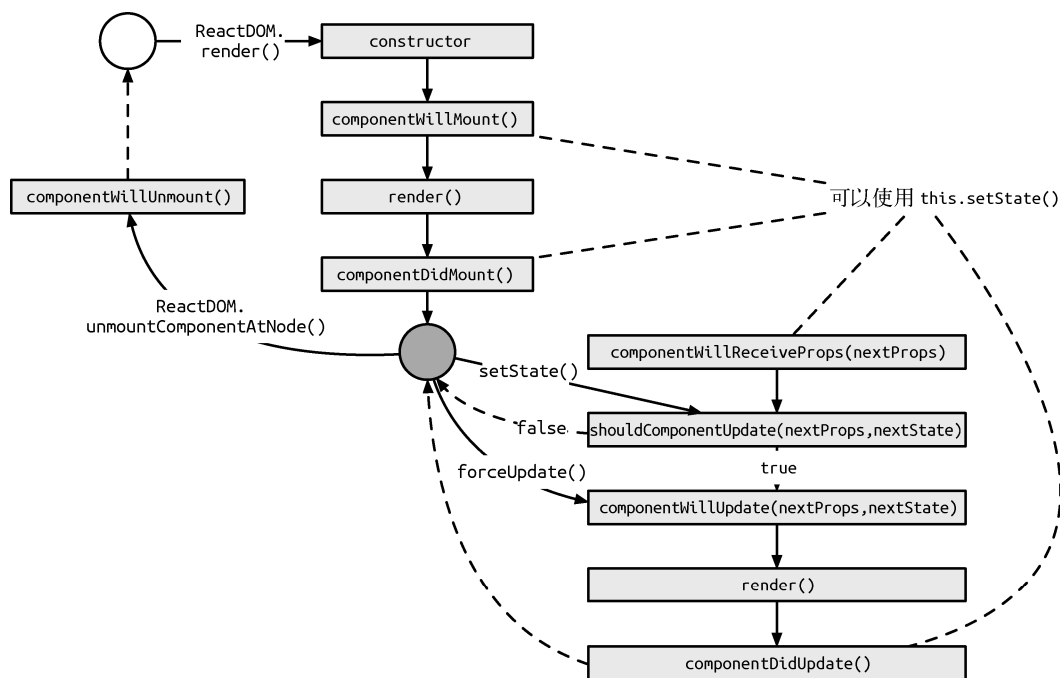


图 1-10 React 生命周期整体流程图

此外，我们在 1.3 节中提到用 `createClass` 来构建组件时，生命周期稍有不同。这里我们对还在用 `createClass` 方式的开发者们，简要说明方法级别上的不同，如图 1-11 所示。

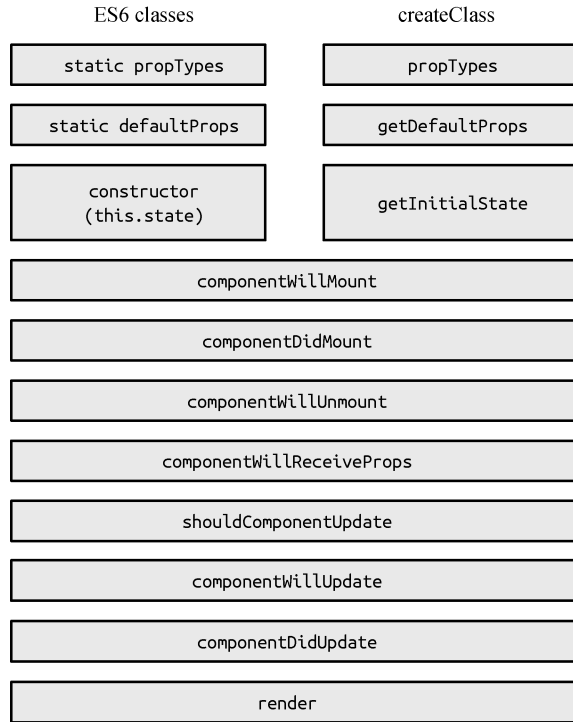


图 1-11 使用 ES6 classes 与 createClass 构建组件方法的异同

我们看到初始化方法有所不同，但生命周期方法均没有变化。此外，ES6 classes 中的静态方法用静态关键词 `static` 声明即可，如 `static customMethod() {}`；`mixin` 属性被移除，可以使用高阶组件（higher-order component）替代。

在源码中，生命周期的调用其实也是复用的代码。为推行 ECMAScript 标准，我们更倾向于使用 ES6 classes 的方式来构建组件。

1.6 React 与 DOM

前面已经介绍完组件的组成部分了，但还缺少最后一环，那就是将组件渲染到真实 DOM 上。从 React 0.14 版本开始，React 将 React 中涉及 DOM 操作的部分剥离开，目的是为了抽象 React，同时适用于 Web 端和移动端。ReactDOM 的关注点在 DOM 上，因此只适用于 Web 端。

在 React 组件的开发实现中，我们并不会用到 ReactDOM，只有在顶层组件以及由于 React 模型所限而不得不操作 DOM 的时候，才会使用它。

1.6.1 ReactDOM

ReactDOM 中的 API 非常少，只有 `findDOMNode`、`unmountComponentAtNode` 和 `render`。下面我们就从 API 的角度来讲讲它们的用法。

1. findDOMNode

上一节我们已经讲过组件的生命周期，DOM 真正被添加到 HTML 中的生命周期方法是 `componentDidMount` 和 `componentDidUpdate` 方法。在这两个方法中，我们可以获取真正的 DOM 元素。React 提供的获取 DOM 元素的方法有两种，其中一种就是 ReactDOM 提供的 `findDOMNode`：

```
DOMNode findDOMNode(ReactComponent component)
```

当组件被渲染到 DOM 中后，`findDOMNode` 返回该 React 组件实例相应的 DOM 节点。它可以用于获取表单的 `value` 以及用于 DOM 的测量。例如，假设要在当前组件加载完时获取当前 DOM，则可以使用 `findDOMNode`：

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

class App extends Component {
  componentDidMount() {
    // this 为当前组件的实例
    const dom = ReactDOM.findDOMNode(this);
  }

  render() {}
}
```

如果在 `render` 中返回 `null`，那么 `findDOMNode` 也返回 `null`。`findDOMNode` 只对已经挂载的组件有效。

涉及复杂操作时，还有非常多的原生 DOM API 可以用。但是需要严格限制场景，在使用之前多问自己为什么要操作 DOM。

2. render

为什么说只有在顶层组件我们才不得不使用 ReactDOM 呢？这是因为要把 React 渲染的 Virtual DOM 渲染到浏览器的 DOM 当中，就要使用 `render` 方法了：

```
ReactDOM.render(
  ReactElement element,
  DOMElement container,
  [function callback]
)
```

该方法把元素挂载到 `container` 中，并且返回 `element` 的实例（即 `refs` 引用）。当然，如果是无状态组件，`render` 会返回 `null`。当组件装载完毕时，`callback` 就会被调用。

当组件在初次渲染之后再次更新时，React 不会把整个组件重新渲染一次，而会用它高效的 DOM diff 算法做局部的更新。这也是 React 最大的亮点之一！

此外，与 render 相反，React 还提供了一个很少使用的 unmountComponentAtNode 方法来进行卸载操作。

1.6.2 ReactDOM 的不稳定方法

ReactDOM 中有两个不稳定方法，其中一个方法与 render 方法颇为相似。讲起它，还得从我们常用的 Dialog 组件在 React 中的实现讲起。

我们先来回忆一下 Dialog 组件的特点，它是不在文档流中的弹出框，一般会绝对定位在屏幕的正中央，背后有一层半透明的遮罩。因此，它往往直接渲染在 document.body 下，然而我们并不知道如何在 React 组件外进行操作。这就要从实现 Dialog 的思路以及涉及 DOM 部分的实现讲起。

这里我们引入 Portal 组件，这是一个经典的实现，最初的实现来源于 React Bootstrap 组件库中的 Overlay Mixin，后来使用越来越广泛。我们截取关键部分的源码：

```
import React from 'react';
import ReactDOM, { findDOMNode } from 'react-dom';
import CSSPropertyOperations from 'react/lib/CSSPropertyOperations';

export default class Portal extends React.Component {
  constructor() {
    // ...
  }

  openPortal(props = this.props) {
    this.setState({ active: true });
    this.renderPortal(props);
    this.props.onOpen(this.node);
  }

  closePortal(isUnmounted = false) {
    const resetPortalState = () => {
      if (this.node) {
        ReactDOM.unmountComponentAtNode(this.node);
        document.body.removeChild(this.node);
      }
      this.portal = null;
      this.node = null;
      if (isUnmounted !== true) {
        this.setState({ active: false });
      }
    };

    if (this.state.active) {
      if (this.props.beforeClose) {
        this.props.beforeClose(this.node, resetPortalState);
      }
    }
  }
}
```



```

    } else {
      resetPortalState();
    }

    this.props.onClose();
  }
}

renderPortal(props) {
  if (!this.node) {
    this.node = document.createElement('div');
    // 在节点增加到 DOM 之前, 执行 CSS 防止无效的重绘
    this.applyClassNameAndStyle(props);
    document.body.appendChild(this.node);
  } else {
    // 当新的 props 传下来的时候, 更新 CSS
    this.applyClassNameAndStyle(props);
  }

  let children = props.children;
  // https://gist.github.com/jimfb/d99e0678e9da715ccf6454961ef04d1b
  if (typeof props.children.type === 'function') {
    children = React.cloneElement(props.children, { closePortal: this.closePortal });
  }

  this.portal = ReactDOM.unstable_renderSubtreeIntoContainer(
    this,
    children,
    this.node,
    this.props.onUpdate
  );
}

render() {
  if (this.props.openByClickOn) {
    return React.cloneElement(this.props.openByClickOn, { onClick: this.handleWrapperClick });
  }
  return null;
}
}

```

从 Portal 组件可以看出, 我们实现了一个“壳”, 其中包括触发事件、渲染的位置以及暴露的方法, 但它并不关心子组件的内容。当我们使用它的时候, 可以这么写:

```

<Portal ref="myPortal">
  <Modal title="My modal">
    Modal content
  </Modal>
</Portal>

```

这个组件可以说是 Dialog 实现的精髓, 我们为 Dialog 的行为抽象了 Portal 这个父组件。

当调用上述代码时，可以注意到在运行到 `componentDidMount` 生命周期方法时，最后调用了 `this.renderPortal()` 方法，这个方法把子组件里的内容插入到 `document.body` 下，这就实现了子组件不在标准文档流的渲染。

这就说到了 ReactDOM 中不稳定的 API 方法 `unstable_renderSubtreeIntoContainer`。它的作用很简单，就是更新组件到传入的 DOM 节点上，我们在这里使用它完成了在组件内实现跨组件的 DOM 操作。

这个方法与 `render` 方法很相似，但 `render` 方法缺少一个插入某个节点的参数。从最终 ReactDOM 方法实现的源代码 `react/src/renderers/dom/client/ReactMount.js` 中可以了解到，`unstable_renderSubtreeIntoContainer` 与 `render` 方法对应调用的方法如下。

- ❑ `render`: `ReactMount._renderSubtreeIntoContainer(null, nextElement, container, callback)`。
- ❑ `unstable_renderSubtreeIntoContainer`: `ReactMount._renderSubtreeIntoContainer(parentComponent, nextElement, container, callback)`。

源码证明了我们的猜想，这也说明了两者的区别在于是否传入父节点。

此外，另一个 ReactDOM 中的不稳定方法 `unstable_batchedUpdates` 是关于 `setState` 的更新策略，我们会在 3.4.5 中详细介绍。

1.6.3 refs

刚才我们已经详述了 ReactDOM 的 `render` 方法，比如我们渲染了一个 App 组件到 root 节点下：

```
const myAppInstance = ReactDOM.render(<App />, document.getElementById('root'));
myAppInstance.doSth();
```

我们利用 `render` 方法得到了 App 组件的实例，然后就可以对它做一些操作。但在组件内，JSX 是不会返回一个组件的实例的！它只是一个 `ReactElement`，只是告诉 React 被挂载的组件应该长什么样：

```
const myApp = <App />;
```

`refs` 就是为此而生的，它是 React 组件中非常特殊的 `prop`，可以附加到任何一个组件上。从字面意思来看，`refs` 即 `reference`，组件被调用时会新建一个该组件的实例，而 `refs` 就会指向这个实例。

它可以是一个回调函数，这个回调函数会在组件被挂载后立即执行。例如：

```
import React, { Component } from 'react';

class App extends Component {
  constructor(props){
    super(props);

    this.handleClick = this.handleClick.bind(this);
  }
}
```

```

handleClick() {
  if (this.myTextInput !== null) {
    this.myTextInput.focus();
  }
}

render() {
  return (
    <div>
      <input type="text" ref={(ref) => this.myTextInput = ref} />
      <input
        type="button"
        value="Focus the text input"
        onClick={this.handleClick}
      />
    </div>
  );
}
}

```

在这个例子中，我们得到 `input` 组件的真正实例，所以可以在按钮被按下后调用输入框的 `focus()` 方法。这个例子把 `refs` 放到原生的 DOM 组件 `input` 中，我们可以通过 `refs` 得到 DOM 节点；而如果把 `refs` 放到 React 组件，比如 `<TextInput />`，我们获得的就是 `TextInput` 的实例，因此就可以调用 `TextInput` 的实例方法。

`refs` 同样支持字符串。对于 DOM 操作，不仅可以使使用 `findDOMNode` 获得该组件 DOM，还可以使用 `refs` 获得组件内部的 DOM。比如：

```

import React, { Component } from 'react';

class App extends Component {
  componentDidMount() {
    // myComp 是 Comp 的一个实例，因此需要用 findDOMNode 转换为相应的 DOM
    const myComp = this.refs.myComp;
    const dom = findDOMNode(myComp);
  }

  render() {
    return (
      <div>
        <Comp ref="myComp" />
      </div>
    );
  }
}

```

要获取一个 React 组件的引用，既可以使用 `this` 来获取当前 React 组件，也可以使用 `refs` 来获取你拥有的子组件的引用。

我们回到 1.6.2 节中 `Portal` 组件里暴露的两个方法 `openPortal` 和 `closePortal`。这两个方法的调用方式为：

```
this.refs.myPortal.openPortal();  
this.refs.myPortal.closePortal();
```

这种命令式调用的方式，尽管说并不是 React 推崇的，但我们仍然可以使用。原则上，在组件状态维护中不建议用这种方式。

为了防止内存泄漏，当卸载一个组件的时候，组件里所有的 `refs` 就会变为 `null`。

值得注意的是，`findDOMNode` 和 `refs` 都无法用于无状态组件中，原因在前面已经说过。无状态组件挂载时只是方法调用，没有新建实例。

对于 React 组件来说，`refs` 会指向一个组件类的实例，所以可以调用该类定义的任何方法。如果需要访问该组件的真实 DOM，可以用 `ReactDOM.findDOMNode` 来找到 DOM 节点，但我们并不推荐这样做。因为这在大部分情况下都打破了封装性，而且通常都能用更清晰的办法在 React 中构建代码。

1.6.4 React 之外的 DOM 操作

DOM 操作可以归纳为对 DOM 的增、删、改、查。这里的“查”指的是对 DOM 属性、样式的查看，比如查看 DOM 的位置、宽、高等信息。而要对 DOM 进行增、删、改，就要先到 DOM 中查询元素。

React 的声明式渲染机制把复杂的 DOM 操作抽象为简单的 `state` 和 `props` 的操作，因此避免了很多直接的 DOM 操作。不过，仍然有一些 DOM 操作是 React 无法避免或者正在努力避免的。

举一个明显的例子，如果要调用 HTML5 Audio/Video 的 `play` 方法和 `input` 的 `focus` 方法，React 就无能为力了，这时只能使用相应的 DOM 方法来实现。

React 提供了事件绑定的功能，但是仍然有一些特殊情况需要自行绑定事件，例如 `Popup` 等组件，当点击组件其他区域时可以收缩此类组件。这就要求我们对组件以外的区域（一般指 `document` 和 `body`）进行事件绑定。例如：

```
componentDidUpdate(prevProps, prevState) {  
  if (!this.state.isActive && prevState.isActive) {  
    document.removeEventListener('click', this.hidePopup);  
  }  
  
  if (this.state.isActive && !prevState.isActive) {  
    document.addEventListener('click', this.hidePopup);  
  }  
}  
  
componentWillUnmount() {  
  document.removeEventListener('click', this.hidePopup);  
}  
  
hidePopup(e) {
```

```

    if (!this.isMounted()) { return false; }

    const node = ReactDOM.findDOMNode(this);
    const target = e.target || e.srcElement;
    const isInside = node.contains(target);

    if (this.state.isActive && !isInside) {
      this.setState({
        isActive: false,
      });
    }
  }
}

```

React 中使用 DOM 最多的还是计算 DOM 的尺寸（即位置信息）。我们可以提供像 width 或 height 这样的工具函数：

```

function width(el) {
  const styles = el.ownerDocument.defaultView.getComputedStyle(el, null);
  const width = parseFloat(styles.width.indexOf('px') !== -1 ? styles.width : 0);

  const boxSizing = styles.boxSizing || 'content-box';
  if (boxSizing === 'border-box') {
    return width;
  }

  const borderLeftWidth = parseFloat(styles.borderLeftWidth);
  const borderRightWidth = parseFloat(styles.borderRightWidth);
  const paddingLeft = parseFloat(styles.paddingLeft);
  const paddingRight = parseFloat(styles.paddingRight);

  return width - borderRightWidth - borderLeftWidth - paddingLeft - paddingRight;
}

```

但上述计算方法并不能完全覆盖所有情况，这需要付出不少的成本去实现。值得高兴的是，React 正在自己构建一个 DOM 排列模型，来努力避免这些 React 之外的 DOM 操作。我们相信在不久的将来，React 的使用者就可以完全抛弃掉 jQuery 等 DOM 操作库。

可以说在 React 组件开发中，还有很多意料之外的情形。在这些情形中，应该如何运用 React 的方式优雅地解决问题是我们需要一直思考的。

1.7 组件化实例：Tabs 组件

前面我们穿插介绍了 Tabs 组件的关键实现，现在将把完整的例子展示出来：

```

import React, { Component, PropTypes, cloneElement } from 'react';
import classNames from 'classnames';
import style from './tabs.scss';

```

这段代码是最基本的引用。除了引用 React 之外，还引用了操作 class 的库 classNames 以及

样式文件。这归功于 webpack 强大的加载机制，详情请参考附录 A 中对 webpack 配置的讲解。

Tabs 组件的封装逻辑在之前已经讲解得很清晰了，即把必要的 props 克隆到 TabNav 或 TabContent 组件中，并把它们组装到一起渲染出来。

值得注意的是，我们在 Tabs 组件中设计了切换 tab 时的 onChange 函数，通过传递 onChange prop 到 TabNav 子组件中，在子组件中完成对节点上事件的绑定：

```
class Tabs extends Component {
  static propTypes = {
    // 在主节点上增加可选 class
    className: PropTypes.string,
    // class 前缀
    classPrefix: PropTypes.string,
    children: PropTypes.oneOfType([
      PropTypes.arrayOf(PropTypes.node),
      PropTypes.node,
    ]),
    // 默认激活索引，组件内更新
    defaultActiveIndex: PropTypes.number,
    // 默认激活索引，组件外更新
    activeIndex: PropTypes.number,
    // 切换时回调函数
    onChange: PropTypes.func,
  };

  static defaultProps = {
    classPrefix: 'tabs',
    onChange: () => {},
  };

  constructor(props) {
    super(props);

    // 对事件方法的绑定
    this.handleTabClick = this.handleTabClick.bind(this);

    const currProps = this.props;

    let activeIndex;
    // 初始化 activeIndex state
    if ('activeIndex' in currProps) {
      activeIndex = currProps.activeIndex;
    } else if ('defaultActiveIndex' in currProps) {
      activeIndex = currProps.defaultActiveIndex;
    }

    this.state = {
      activeIndex,
      prevIndex: activeIndex,
    };
  }
}
```

```

componentWillReceiveProps(nextProps) {
  // 如果 props 传入 activeIndex, 则直接更新
  if ('activeIndex' in nextProps) {
    this.setState({
      activeIndex: nextProps.activeIndex,
    });
  }
}

handleTabClick(activeIndex) {
  const prevIndex = this.state.activeIndex;

  // 如果当前 activeIndex 与传入的 activeIndex 不一致,
  // 并且 props 中存在 defaultActiveIndex 时, 则更新
  if (this.state.activeIndex !== activeIndex &&
    'defaultActiveIndex' in this.props) {
    this.setState({
      activeIndex,
      prevIndex,
    });

    // 更新后执行回调函数, 抛出当前索引和上一次索引
    this.props.onChange({ activeIndex, prevIndex });
  }
}

renderTabNav() {
  const { classPrefix, children } = this.props;

  return (
    <TabNav
      key="tabBar"
      classPrefix={classPrefix}
      onTabClick={this.handleTabClick}
      panels={children}
      activeIndex={this.state.activeIndex}
    />
  );
}

renderTabContent() {
  const { classPrefix, children } = this.props;

  return (
    <TabContent
      key="tabcontent"
      classPrefix={classPrefix}
      panels={children}
      activeIndex={this.state.activeIndex}
    />
  );
}

render() {

```

```
const { className } = this.props;
// classnames 用于合并 class
const classes = classnames(className, 'ui-tabs');

return (
  <div className={classes}>
    {this.renderTabNav()}
    {this.renderTabContent()}
  </div>
);
}
```

我们看到，两个子组件 TabNav 和 TabContent 的渲染起到了至关重要的作用。而 TabNav 组件与 TabContent 组件处理的逻辑类似，不同的是前者是从 TabPane 组件的 tab prop 中取得内容，后者是从 TabPane 组件的 children 中取得内容。

我们来看一下 TabNav 组件的实现：

```
class TabNav extends Component {
  static propTypes = {
    classPrefix: React.PropTypes.string,
    panels: PropTypes.node,
    activeIndex: PropTypes.number,
  };

  getTabs() {
    const { panels, classPrefix, activeIndex } = this.props;

    return React.Children.map(panels, (child) => {
      if (!child) { return; }

      const order = parseInt(child.props.order, 10);

      // 利用 class 控制显示和隐藏
      let classes = classnames({
        [`${classPrefix}-tab`]: true,
        [`${classPrefix}-active`]: activeIndex === order,
        [`${classPrefix}-disabled`]: child.props.disabled,
      });

      let events = {};
      if (!child.props.disabled) {
        events = {
          onClick: this.props.onTabClick.bind(this, order),
        };
      }

      const ref = {};
      if (activeIndex === order) {
        ref.ref = 'activeTab';
      }
    });
  }
}
```



```

    return (
      <li
        role="tab"
        aria-disabled={child.props.disabled ? 'true' : 'false'}
        aria-selected={activeIndex === order ? 'true' : 'false'}
        {...events}
        className={classes}
        key={order}
        {...ref}
      >
        {child.props.tab}
      </li>
    );
  });
}

render() {
  const { classPrefix } = this.props;

  const rootClasses = classNames({
    [`${classPrefix}-bar`]: true,
  });

  const classes = classNames({
    [`${classPrefix}-nav`]: true,
  });

  return (
    <div className={rootClasses} role="tablist">
      <ul className={classes}>
        {this.getTabs()}
      </ul>
    </div>
  );
}
}

```

然后是 TabContent 组件，仔细对比它与前者的不同。再次推敲 TabContent 组件中的 `getTabPanels` 方法，看似简单，实则精妙：

```

class TabContent extends Component {
  static propTypes = {
    classPrefix: React.PropTypes.string,
    panels: PropTypes.node,
    activeIndex: PropTypes.number,
    isActive: PropTypes.bool,
  };

  getTabPanels() {
    const { classPrefix, activeIndex, panels, isActive } = this.props;

    return React.Children.map(panels, (child) => {
      if (!child) { return; }

```

```
const order = parseInt(child.props.order, 10);
const isActive = activeIndex === order;

return React.cloneElement(child, {
  classPrefix,
  isActive,
  children: child.props.children,
  key: `tabpanel-${order}`,
});
});
}

render() {
  const { classPrefix } = this.props;

  const classes = classNames({
    [`${classPrefix}-content`]: true,
  });

  return (
    <div className={classes}>
      {this.getTabPanes()}
    </div>
  );
}
```

最后是 TabPane 组件，它是最末端的节点，只有最基本的渲染：

```
class TabPane extends Component {
  static propTypes = {
    tab: PropTypes.oneOfType([
      PropTypes.string,
      PropTypes.node,
    ]).isRequired,
    order: PropTypes.string.isRequired,
    disable: PropTypes.bool,
    isActive: PropTypes.bool,
  };

  render() {
    const { classPrefix, className, isActive, children } = this.props;

    const classes = classNames({
      [className]: className,
      [`${classPrefix}-panel`]: true,
      [`${classPrefix}-active`]: isActive,
    });

    return (
      <div
        role="tabpanel"
        className={classes}
      >
```

```
        aria-hidden={!isActive}>
        {children}
      </div>
    );
  }
}
```

自此，Tabs 组件就开发完毕了。

1.8 小结

本章通过穿插 Tabs 组件的实现介绍了 React 的主要概念及 API，为读者开启了通向 React 的大门。

随着章节的深入，我们会陆续介绍 React 高阶使用方法、背后的运行机制、处理数据的架构 Flux 与 Redux。相信从现在开始，你已经做好在 React 的海洋里遨游的准备了。