O'REILLY®

(第二版)

React 学习手册

Learning React Second Edition



[美] 亚历克斯·班克斯(Alex Banks)
伊夫·波尔切洛(Eve Porcello) 著
安道 译

中国电力出版社
CHINA ELECTRIC POWER PRESS

第二版

React学习手册

[美] 亚历克斯·班克斯（Alex Banks）和
伊夫·波尔切洛（Eve Porcello）著

安道 译

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社
CHINA ELECTRIC POWER PRESS

Copyright © 2020 Alex Banks and Eve Porcello. All rights reserved.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2020. Authorized translation of the English edition, 2020 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2020。

简体中文版由中国电力出版社出版 2020。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

图书在版编目（CIP）数据

React 学习手册 /（美）亚历克斯·班克斯（Alex Banks）和伊夫·波尔切洛（Eve Porcello）著；安道译 . — 北京：中国电力出版社，2021.6

书名原文：Learning React, Second Edition

ISBN 978-7-5198-5654-0

I. ① R… II. ①亚… ②伊… ③安… III. ①移动终端－应用程序－程序设计－技术手册 IV. ① TN929.53-62

中国版本图书馆 CIP 数据核字 (2021) 第 096951 号

北京市版权局著作权合同登记 图字：01-2021-2208 号

出版发行：中国电力出版社

地　　址：北京市东城区北京站西街 19 号（邮政编码 100005）

网　　址：http://www.cepp.sgcc.com.cn

责任编辑：刘 炽（liuchi1030@163.com）

责任校对：王小鹏

装帧设计：Karen Montgomery，张 健

责任印制：杨晓东

印　　刷：北京天宇星印刷厂

版　　次：2021 年 6 月第一版

印　　次：2021 年 6 月北京第一次印刷

开　　本：750 毫米 ×980 毫米 16 开本

印　　张：23.5

字　　数：486 千字

印　　数：0001—3000 册

定　　价：98.00元

版 权 专 有 侵 权 必 究

本书如有印装质量问题，我社营销中心负责退换

O'Reilly Media, Inc.介绍

O'Reilly以“分享创新知识、改变世界”为己任。40多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O'Reilly业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来O'Reilly图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

目录

前言 1

第 1 章 初识 React 7

1.1 坚实的基础 8

1.2 React 的过去和未来 9

1.3 资源文件 10

1.3.1 资源仓库 10

1.3.2 React 开发者工具 10

1.3.3 安装 Node.js 12

第 2 章 JavaScript 新特性 15

2.1 声明变量 16

2.1.1 const 关键字 16

2.1.2 let 关键字 17

2.1.3 模板字符串 20

2.2 创建函数 21

2.2.1 函数声明 21

2.2.2 函数表达式 22

2.2.3 默认参数 24

2.2.4 箭头函数 25

2.3 编译 JavaScript 28

2.4 对象和数组 29
2.4.1 析构对象 29
2.4.2 析构数组 32
2.4.3 对象字面量增强 32
2.4.4 展开运算符 34
2.5 JavaScript 异步编程 36
2.5.1 使用 fetch 处理简单的 promise 37
2.5.2 async/await 38
2.5.3 构建 promise 39
2.6 类 40
2.7 ES6 模块 42

第 3 章 JavaScript 函数式编程 45

3.1 函数式编程是什么意思 46
3.2 命令式和声明式 48
3.3 函数式编程基本概念 51
3.3.1 不可变性 51
3.3.2 纯函数 54
3.3.3 数据转换 57
3.3.4 高阶函数 65
3.3.5 递归 66
3.3.6 合成 70
3.3.7 综合应用 71

第 4 章 React 运行机制 77

4.1 页面设置 77
4.2 React 元素 78
4.3 ReactDOM 81
4.4 React 组件 87

第 5 章 在 React 中使用 JSX 95

5.1 使用 JSX 创建 React 元素 95
5.1.1 JSX 小贴士 96

5.1.2 使用 JSX 映射数组 98
5.2 Babel 98
5.3 使用 JSX 编写菜谱应用 100
5.4 React 片段 108
5.5 webpack 简介 109
5.5.1 创建项目 111
5.5.2 加载构建包 122
5.5.3 源码映射 123
5.5.4 Create React App 124
第 6 章 React 状态管理 127
6.1 构建一个星级评价组件 128
6.2 useState 钩子 130
6.3 为提高可重用性而重构 134
6.4 组件树中的状态 136
6.4.1 沿组件树向下发送状态 137
6.4.2 沿组件树向上发送交互 141
6.5 构建表单 145
6.5.1 使用 ref 146
6.5.2 受控组件 148
6.5.3 自定义钩子 150
6.5.4 把颜色添加到状态中 152
6.6 React 上下文 153
6.6.1 把颜色放入上下文 155
6.6.2 使用 useContext 获取颜色 157
6.6.3 有状态的上下文供应组件 158
6.6.4 使用上下文自定义钩子 160
第 7 章 使用钩子增强组件 165
7.1 useEffect 介绍 165
7.1.1 依赖数组 168
7.1.2 深入检查依赖 173

7.1.3 何时使用 useLayoutEffect 180
7.1.4 钩子使用规则 182
7.1.5 使用 useReducer 改进代码 186
7.1.6 使用 useReducer 处理复杂的状态 189
7.1.7 提升组件性能 192
7.1.8 shouldComponentUpdate 和 PureComponent 195
7.1.9 何时重构 196

第 8 章 处理数据 199

8.1 请求数据 199
8.1.1 随请求发送数据 201
8.1.2 使用 fetch 上传文件 202
8.1.3 授权请求 202
8.1.4 在本地保存数据 204
8.1.5 处理 promise 状态 208
8.2 渲染属性 210
8.3 虚拟化列表 214
8.3.1 创建 useFetch 钩子 219
8.3.2 创建 Fetch 组件 222
8.3.3 处理多个请求 224
8.3.4 备忘值 225
8.3.5 瀑布式请求 229
8.3.6 限制网络速度 233
8.3.7 并行请求 235
8.3.8 等待值出现 237
8.3.9 取消请求 238
8.4 GraphQL 简介 242
8.4.1 GitHub GraphQL API 242
8.4.2 发起一个 GraphQL 请求 245

第 9 章 Suspense 253

9.1 错误边界组件 255

9.2 代码分拆 259
9.2.1 Suspense 组件介绍 262
9.2.2 使用 Suspense 加载数据 263
9.2.3 抛出 promise 267
9.2.4 构建兼容 Suspense 的数据源 272
9.2.5 Fiber 277

第 10 章 React 测试 281

10.1 ESLint 281
10.2 Prettier 288
10.2.1 在项目中配置 Prettier 289
10.2.2 在 VSCode 中使用 Prettier 291
10.3 对 React 应用做类型检查 292
10.3.1 PropTypes 292
10.3.2 Flow 296
10.3.3 TypeScript 301
10.4 测试驱动开发 304
10.5 使用 Jest 305
10.6 测试 React 组件 311
10.6.1 查询 315
10.6.2 测试事件 316
10.6.3 代码覆盖度 319

第 11 章 React Router 321

11.1 使用 React Router 322
11.2 React Router 属性 327
11.3 重定向 332

第 12 章 React 服务器端渲染 341

12.1 同构与普适 342
12.2 服务器端渲染 React 345
12.3 使用 Next.js 做服务器端渲染 353

12.4 Gatsby 359
12.5 React 未来展望 361
作者介绍 363
封面介绍 363

前言

本书是为那些想学习 React 库，以及想学习当前 JavaScript 语言最新技术的开发人员而写的。当下是 JavaScript 开发者的黄金时代，整个生态系统蓬勃发展，不断涌现新工具、句法和最佳实践，以期解决开发中遇到的很多问题。笔者写这本书旨在整合相关技术，让你顺利掌握 React。本书涵盖状态管理、React 路由、测试和服务器端渲染，不仅介绍基础知识，还将进行实战演练。

阅读本书不要求你事先了解 React 的任何知识。笔者将从头开始介绍 React 的基础知识。同样，也不要求你用过最新的 JavaScript 句法。这部分知识在第 2 章介绍，作为后续章节的基础。

如果你熟悉 HTML、CSS 和 JavaScript，阅读本书就没有什么障碍。专攻一个 JavaScript 库时，熟悉这三大技术总是好的。

在阅读本书的过程中，请随时查阅配套的 GitHub 仓库（*http://github.com/moonhighway/learning-react*）。本书的所有示例都在这个仓库中，你可以自己动手实践。

排版约定

本书采用下述排版约定。

斜体（*Italic*）

表示新术语、URL、电子邮件地址、文件名和扩展名。

等宽字体（`Constant Width`）

表示程序清单，在段落中出现则表示程序元素，例如变量、函数名、数据库、数据类型、环境变量、语句和关键字。

斜体等宽字体（*`Constant Width Italic`*）

表示应该由用户原封不动输入的命令或其他文本。



表示提示或建议。



表示一般性说明。



表示警告。

使用代码示例

本书的补充材料（代码示例、练习题等）可到 *https://github.com/moonhighway/learning-react* 下载。

如有技术疑问，或者在使用代码示例的过程中遇到问题，请发电子邮件到 *bookquestions@oreilly.com*。

本书是帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用到你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用书中的几个代码片段写一个程序就无须获得许可，销售或分发O'Reilly旗下书中的示例则需要获得许可；引用书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和ISBN，比如："Learning React by Alex Banks and Eve Porcello (O'Reilly). Copyright 2020 Alex Banks and Eve Porcello, 978-1-492-05172-5."。

如果你觉得自己对示例代码的使用超出了上述许可范围，请通过 *permissions@oreilly.com* 与我们联系。

O'Reilly Online Learning



40年间，O'Reilly Media为众多公司提供技术和商业培训，提升知识储备和洞察力，为企业的成功助力。

我们有一群独家专家和创新者，他们通过图书、文章、会议和在线学习平台分享知识和技术。O'Reilly的在线学习平台提供按需访问的直播培训课程、详细的学习路径、交互式编程环境，以及由O'Reilly和其他200多家出版社出版的书籍和视频。详情请访问 *http://oreilly.com*。

联系方式

请将你对本书的评价和问题发给出版社：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

勘误、示例和其他信息可到 *https://oreil.ly/learningReact_2e* 上获取。

对本书的评论或技术疑问，可以发电子邮件到 *bookquestions@oreilly.com*。

欲了解本社图书、课程、会议和新闻等更多信息，请访问我们的网站 *http://oreilly.com*。

我们的 Facebook：*http://facebook.com/oreilly*。

我们的 Twitter：*http://twitter.com/oreillymedia*。

我们的 YouTube：*http://www.youtube.com/oreillymedia*。

致谢

笔者接触 React 纯粹是机缘巧合。为 Yahoo 内部的一个 JavaScript 全栈培训项目编写材料时，笔者用的是 YUI，然而在 2014 年 8 月，YUI 终止开发了。我们不得不修改所有课程文件，可是改成什么呢？现在前端应该使用什么呢？答案是 React。我们没有立即喜欢上 React，几个小时之后才被它吸引。当时我们猜测，React 将产生翻天覆地的影响。非常幸运，我们在早期就选定了它。

感谢 Angela Rufino 和 Jennifer Pollock 对本书第二版的帮助和支持。同时，感谢 Ally MacDonald 对第一版的编辑工作。感谢几位技术审校：Scott Iwako、Adam Rackis、Brian Sletten、Max Firtman 和 Chetan Karande。

没有 Sharon Adams 和 Marilyn Messineo，这本书不可能得以存世。Alex 的第一台电脑，一台 Tandy TRS 80 彩色计算机就是他们二人合买的。倘若没有 Jim 和

Lorri Porcello 及 Mike 和 Sharon Adams 的关爱、支持和鼓励，本书不会最终得以付梓。

我们还要感谢加利福尼亚州 Tahoe 市的 Connexion 咖啡馆，感谢这家店为我们供应完成本书所需的咖啡。同时感谢店主 Robin，他的鞭策给了我们无限动力："写一本编程书？听着就无聊！"

第 1 章

初识 React

一个 JavaScript 库，怎样才算优秀？取决于 GitHub 上的星标数、npm 上的下载数，还是 Twitter 上业界领军人物日常的发推数？怎样选择最合适的工具来创建一流作品呢？我们怎么知道是否值得投入时间？又怎么知道选择的工具是合适的呢？

React 发布初期，围绕它的讨论很多，有人说它好，也有人持怀疑态度。那时 React 刚刚问世，新事物总是让人望而却步。

为了回应这些声音，React 团队的 Pete Hunt 写了一篇文章，题为“Why React?”。这篇文章建议“给它（React）五分钟时间”。他的目的是鼓励人们先尝试一下 React，不要盲目定调说开发团队的方式太疯狂。

是的，React 是一个小型库，可能没有内置你构建应用所需的一切功能，但请给它五分钟时间。

是的，使用 React 写出的代码像是 JavaScript 代码中的 HTML。当然，这些标签要经过预处理才能在浏览器中运行。为此，可能需要用到构建工具，例如 webpack。请给 React 五分钟时间。

React 问世已经十年，很多团队认定它是优秀的，这一切都始于最初的五分钟。我们指的是 Uber、Twitter、Airbnb 和 Twitter，大量公司尝试了 React，事实证明，React 确实可以帮助团队以更快的速度构建更好的产品。说到底，这不就是我们所追求的吗？不为发推，不为星标数，也不为下载数。我们的初心是使用自己喜欢的工具构建传世的产品，我们的初心是推出可让我们自豪地说出口的产品。如果这也是你的初心，或许你就会爱上 React。

1.1 坚实的基础

不管你是 React 新手，还是想通过本书学习 React 的最新特性，笔者都希望本书能为你在以后使用这个库的过程中建立坚实的基础。本书的目标是帮你理清学习过程，按照学习路线图，一步一步攀上高峰。

深入学习 React 之前，一定要熟悉 JavaScript。不是说你要精通 JavaScript 的方方面面，掌握每一种模式，而是要熟悉数组、对象和函数，这样在阅读本书时才能一帆风顺。

下一章介绍较新的 JavaScript 句法，带你熟悉最新的 JavaScript 特性，尤其是 React 经常用到的那部分。随后，介绍 JavaScript 函数式编程，以便理解深入 React 骨髓的范式。在使用 React 的过程中，合理使用模式写出的代码可读性更高，方便重用且易于测试，这对 JavaScript 开发人员有很大的助益。你会慢慢习惯，彻底改头换面。

接下来，介绍使用组件构建用户界面需要掌握的 React 基础知识。随后学习综合应用组件，通过属性和状态添加逻辑。我们将介绍 React Hooks，以便在多个组件之间重用有状态的逻辑。

了解基础知识之后，我们将构建一个应用，供用户添加、编辑和删除颜色。我们将学习如何借助 Hooks 和 Suspense 获取数据。在构建那个应用的过程中，我们将介绍来自 React 生态系统中的一些工具，处理常见的问题，例如路由、测试和服务器端渲染。

希望通过这种方式能让你更快地熟悉 React 生态系统，不仅了解表面知识，更要掌握在实际工作中构建 React 应用所需的工具和技能。

1.2 React 的过去和未来

React 最初由 Facebook 的一名软件工程师 Jordan Walke 开发。2011 年，Facebook 的新闻摘要使用了这个库；2012 年，Facebook 收购 Instagram 后也用上了这个库。在 2013 年的 JSConf 上，React 开源了，成为众多 UI 库中的一员，与 jQuery、Angular、Dojo、Meteor 等并列一类。那时，React 被看作“MVC 中的 V”。也就是说，React 组件起到视图层的作用，即 JavaScript 应用的用户界面。

自那时起，社区开始广泛使用 React。2015 年 1 月，Netflix 宣布使用 React 开发 UI。那个月晚些时候，一个使用 React 构建移动应用的库 React Native 发布了。Facebook 这还发布了 ReactVR，这个工具把 React 带入了更广阔的应用领域。2015 年和 2016 这两年出现了很多后来广受欢迎的工具，例如 React Router、Redux 和 Mobx，用于处理路由和状态管理等任务。毕竟，React 是一个库，专注于实现特定的功能集，而不大包大揽，涵盖所用使用场景。

这条时间线上的另一个重要事件是 2017 年发布的 React Fiber。Fiber 重写了 React 充满神秘色彩的渲染算法，把 React 的内核推倒重建，可是几乎没有改变任何公开 API。重写之后，React 更加现代化，性能更好，而且对用户没有什么影响。

时间继续向前推进，2019 年发布的 Hooks，这是为组件添加有状态的逻辑及在组件之间共享逻辑的新方式。此外，我们还迎来了 Suspense，优化了 React 的异步渲染。

可以预见，未来我们还将看到更多变化，不过 React 成功的原因之一是这些年在背后辛勤开发项目的团队。这个团队雄心万丈，但又小心谨慎，不断推进前瞻性的优化，同时兼顾改动可能对整个社区产生的影响。

在 React 和相关工具发展的过程中，有时会引入破坏性变动。其实，这些工具的未来版本也可能会导致本书中的示例代码无法运行。不过，你依然可以继续使用书中的代码示例。我们将在 *package.json* 文件中提供具体的版本信息，以便你安装各个包的正确版本。

除了本书，你还可以关注 React 的官方博客（*https://facebook.github.io/react/blog*），跟踪变化。发布 React 的新版本时，核心团队会撰写一篇详细的文章和更改日志来说明变化。官方博客已被翻译成多种语言，如果你的母语不是英语，可以在文档网站的语言页面中找到本地化版本（*https://reactjs.org/languages*）。

第二版的变化

这是本书第二版。过去几年，React 不断发展，笔者觉得有必要更新一下。本书重点关注 React 团队目前推荐的最佳实践，不过也会提及一些弃用的特性，毕竟现在还有很多几年前编写的 React 代码需要运行和维护。放心，笔者将在边注栏中提及这些特性，以防你需要维护旧的 React 应用。

1.3 资源文件

本节说明如何使用本书的资源文件，以及如何安装一些有用的 React 工具。

1.3.1 资源仓库

本书配套的 GitHub 仓库（*https://github.com/moonhighway/learning-react*）中有每一章的代码文件。

1.3.2 React 开发者工具

笔者强烈建议安装 React 开发者工具，来辅助你开发 React 项目。这些工具可通过扩展在 Chrome 和 Firefox 中安装，可作为 Safari、IE 和 React Native 的独立应用程序使用。借助开发者工具可以审查 React 组件树、查看属性和状态详情，

甚至还可以判断哪些线上网站使用的是 React。这在调试上有很大的帮助，还可以让我们了解其他项目是如何使用 React 的。

若想安装，请访问这个 GitHub 仓库（*https://oreil.ly/5tizT*）。仓库中有 Chrome（*https://oreil.ly/Or3pH*）和 Firefox 扩展（*https://oreil.ly/uw3uv*）的链接。

安装之后，你就可以看出哪些网站正在使用 React 了。只要浏览器工具栏中 React 的图标点亮了（见图 1-1），就说明那个网站使用的是 React。

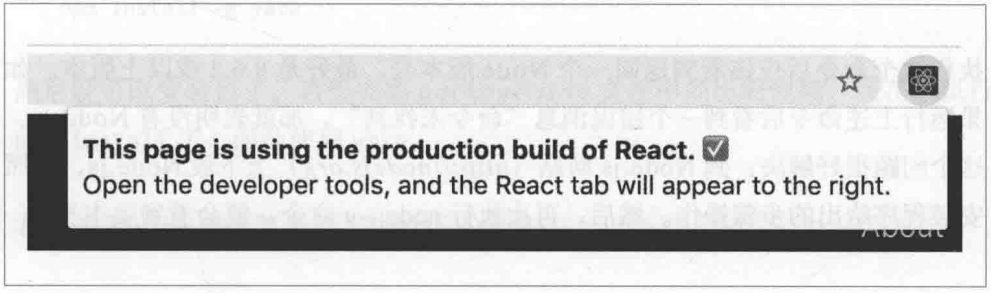

图 1-1：在 Chrome 中使用 React 开发者工具

此时，打开浏览器的开发者工具，你会看到一个名为“React”的新标签页，如图 1-2 所示。点开这个标签页，就会看到构成当前查看页面的所有组件。

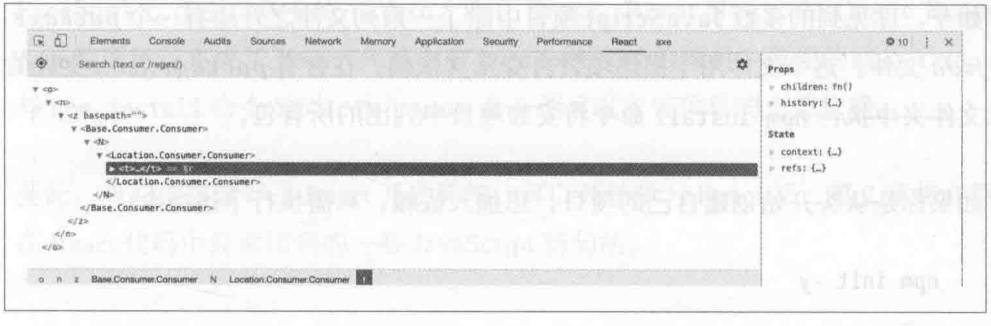

图 1-2：使用 React 开发者工具审查 DOM

1.3.3 安装 Node.js

Node.js 是一个 JavaScript 运行时环境，用于构建全栈应用。Node 是开源的，可以在 Windows、macOS、Linux 等平台中安装。第 12 章构建 Express 服务器时会用到 Node。

使用 React 必须安装 Node，但不要求精通 Node。如果你不确定自己的设备中是否安装了 Node.js，请打开一个终端或命令提示符窗口，输入下述命令：

```
node -v
```

执行这个命令后应该看到返回一个 Node 版本号，最好是 8.6.2 或以上版本。如果运行上述命令后看到一个错误消息“命令未找到”，那就表明没有 Node.js。这个问题很好解决：到 Node.js 网站（*http://nodejs.org*）上下载 Node.js，按照安装程序给出的步骤操作。然后，再次执行 `node -v` 命令，便会看到版本号。

npm

安装 Node.js 顺带就安装了 npm，即 Node 包管理器。在 JavaScript 社区中，工程师喜欢分享自己的开源代码项目，以免重复编写框架、库或辅助函数。React 就是一个有用的 npm 库。在本书的后续章节中，我们将使用 npm 安装各种包。

如今，你见到的多数 JavaScript 项目中除了一系列文件之外还有一个 *package.json* 文件。这个文件用于描述项目自身及其依赖。在含有 *package.json* 文件的文件夹中执行 `npm install` 命令将安装项目中列出的所有包。

如果你是从头开始创建自己的项目，想加入依赖，只需执行下述命令：

```
npm init -y
```

这个命令的作用是初始化项目，创建 *package.json* 文件。然后，你就可以使用 npm 安装依赖了。使用 npm 安装包的方法是执行下述命令：

```
npm install package-name
```

若想使用 npm 删除某个包，执行下述命令：

```
npm remove package-name
```

Yarn

npm 的一个替代品是 Yarn。这个工具由 Facebook 联合 Exponent、Google 和 Tilde 于 2016 年发布。Facebook 等公司使用这个工具高效管理依赖。熟悉 npm 的工作流程之后，上手 Yarn 易如反掌。首先，使用 npm 全局安装 Yarn：

```
npm install -g yarn
```

然后就可以安装包了。若想安装 *package.json* 文件中列出的依赖，现在不执行 `npm install` 了，而是使用 `yarn`。

若想使用 `yarn` 安装某个包，执行下述命令：

```
yarn add package-name
```

删除依赖的命令也不陌生：

```
yarn remove package-name
```

Facebook 已在生产环境使用 Yarn，而且 React、React Native 和 Create React App 等项目都集成了 Yarn。凡是含有 *yarn.lock* 文件的项目，使用的就是 Yarn。与 `npm install` 命令类似，执行 `yarn` 命令便可以安装项目的全部依赖。

至此，我们便搭建好了 React 开发环境，可以开始学习 React 了。第 2 章将介绍在 React 代码中经常用到的一些 JavaScript 新句法。

第 2 章

JavaScript 新特性

JavaScript 自 1995 年发布以来历经很多变化。一开始，我们使用 JavaScript 为网页添加交互元素，例如按钮点击效果、悬停状态、表单验证等。然后，有了 DHTML 和 AJAX 的携持，JavaScript 变得更加强大。如今，借助 Node.js，JavaScript 已经变成一门真正的软件开发语言，用于构建全栈应用。JavaScript 无处不在。

JavaScript 的演进由一群来自使用 JavaScript 的公司、浏览器厂商和社区领袖的人引导。这些年负责指导 JavaScript 发展的委员会是 European Computer Manufacturers Association（ECMA，欧洲电脑制造商协会）。对语言的更改由社区驱动，来自社区成员撰写的提案。任何人都可以向 ECMA 委员会提交提案（*https://tc39.github.io/process-document*）。ECMA 委员会的职责是管理和安排各项提案，决定每一版规范包含哪些特性。

ECMAScript 的第一版于 1997 年发布，名为 ECMAScript1。随后在 1998 年又发布了 ECMAScript2。ECMAScript3 于 1999 年发布，增加了正则表达式、字符串处理功能等。ECMAScript4 的审定过程受政治因素的影响，混乱不堪，最终未能通过，也就没有发布。2009 年，ECMAScript5（ES5）发布，带来了新的数组方法、对象属性和支持 JSON 的库。

自此之后，JavaScript 发展势头迅猛。2015 年发布 ES6（ES2015）之后，每一年都有新的 JavaScript 特性发布。尚在讨论中的提案通常称为 ESNext，表示未来的 JavaScript 规范将包含的特性。

提案历经几个既定的阶段，先是阶段 0，表示新提交的提案，最后是阶段 4，表示提案已完成。当一个提案受到关注，要不要实现提案中的特性，取决于 Chrome 和 Firefox 等浏览器厂商。以 `const` 关键字为例，以前，创建变量都使用 `var`。ECMA 委员会认为，应该有个 `const` 关键字用于声明常量（本章后文详述）。可是，刚引入 `const` 时，我们不能直接在 JavaScript 代码中使用，这样在浏览器中是无法正常运行的。现在之所以可以直接使用，是因为浏览器厂商已经提供了支持。

本章即将讨论的很多特性在最新版浏览器中已经得到支持，不过我们也将介绍如何编译 JavaScript 代码，即把浏览器尚不支持的新句法转换成可以正确识别的旧句法。kangax 编制的兼容表（*https://oreil.ly/oe7la*）是一个很好的参考资源，可以查明最新的 JavaScript 特性在各浏览器中的支持程度。

本章说明在阅读本书过程中能用到的 JavaScript 句法。希望本章的内容能让你了解足够的 JavaScript 句法知识，顺利使用 React。如果你一直没有下定决心转用最新句法，现在是个好机会。如果你已经熟悉最新的语言特性，请跳到下一章。

2.1 声明变量

在 ES2015 之前，声明变量只能使用 `var` 关键字。而现在，我们有几种不同的选择，而且功能得到了增强。

2.1.1 const 关键字

常量是不能覆盖的变量，一旦声明，其值不能再更改。在 JavaScript 中，我们会用到很多不应该覆盖的变量，因此会经常用到 `const`。一些语言很早之前就这么做了，不过 JavaScript 在 ES6 中才引入常量。

常量出现之前，我们只能使用变量，而变量是可以覆盖的。

```
var pizza = true;
pizza = false;
console.log(pizza); // false
```

常量的值不可以重置，倘若试图覆盖常量的值，控制台会报错（见图 2-1）。

```
const pizza = true;
pizza = false;
```

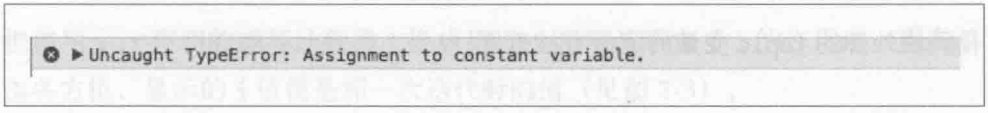

图 2-1：试图覆盖常量

2.1.2 let 关键字

现在，JavaScript 变量具有词法作用域。在 JavaScript 中，代码块使用花括号（{}）标识。对函数来说，花括号会限定使用 `var` 声明的变量的作用域。然而，来看 `if/else` 语句的情况。从其他语言转过来的人可能会认为这种语句块也会限定变量的作用域。实际情况是，直到 `let` 出现之后才是如此。

在 `if/else` 块中创建的变量，其作用域不被限定在块中。

```
var topic = "JavaScript";

if (topic) {
  var topic = "React";
  console.log("block", topic); // block React
}

console.log("global", topic); // global React
```

`if` 块中的 `topic` 变量重置了块外部的 `topic` 变量的值。

使用 let 关键字，可以把变量的作用域限定在任何代码块中。使用 let 可以保住全局变量的值：

```
var topic = "JavaScript";

if (topic) {
  let topic = "React";
  console.log("block", topic); // React
}

console.log("global", topic); // JavaScript
```

代码块外部的 topic 变量的值没有被重置。

在 for 循环中，花括号也不限定变量的作用域。

```
var div,
  container = document.getElementById("container");

for (var i = 0; i < 5; i++) {
  div = document.createElement("div");
  div.onclick = function() {
    alert("This is box #" + i);
  };
  container.appendChild(div);
}
```

在这个循环中，我们创建了 5 个 div 元素，都放入一个容器中。每个 div 元素都赋予了一个 onclick 处理程序，创建一个弹出框，显示索引。for 循环中声明的 i 是全局变量，经过几次迭代，其值变成 5。单击任何一个 div 元素，弹出框中显示的 i 值都是 5，因为全局变量 i 当前的值就是 5（见图 2-2）。

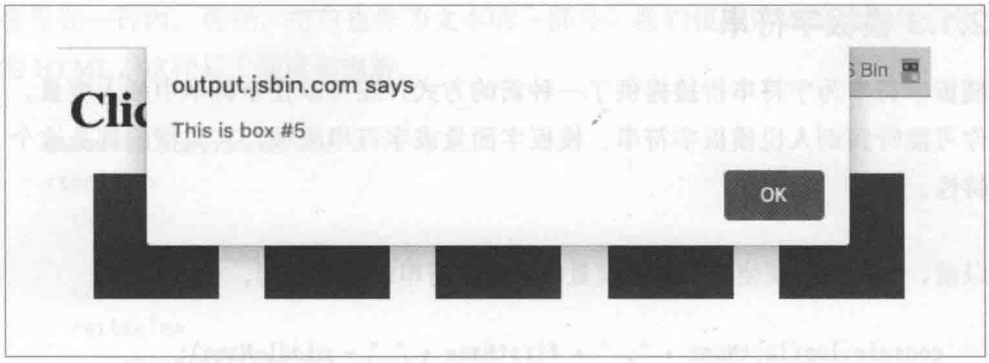

图 2-2：对每个方框来说，i 都等于 5

把使用 `var` 声明的循环计数器 `i` 换成用 `let` 声明就限定了 `i` 的作用域。现在单击各方框，显示的 `i` 值便是那一次迭代时的值（见图 2-3）。

```
const container = document.getElementById("container");
let div;
for (let i = 0; i < 5; i++) {
  div = document.createElement("div");
  div.onclick = function() {
    alert("This is box #: " + i);
  };
  container.appendChild(div);
}
```

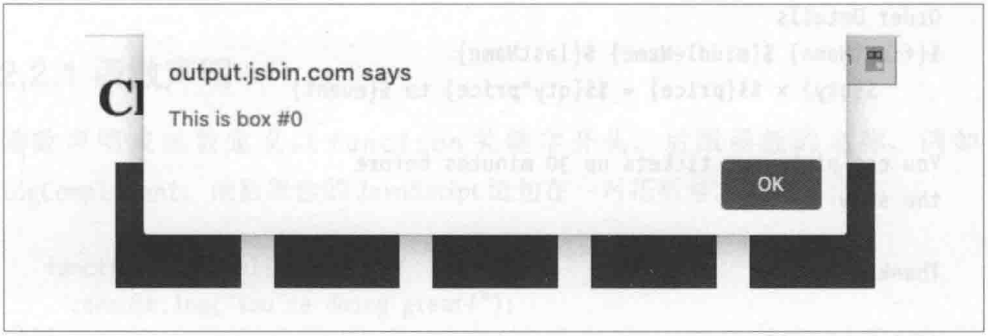

图 2-3：i 的作用域得到 let 的保护

i 的作用域得到 let 的保护。

2.1.3 模板字符串

模板字符串为字符串拼接提供了一种新的方式，还可以在字符串中插入变量。你可能听到别人说模板字符串、模板字面量或字符串模板，其实指的都是这个特性。

以前，字符串拼接使用加号把变量的值和字符串组合在一起。

```
console.log(lastName + ", " + firstName + " " + middleName);
```

有了模板，我们在创建的字符串中使用 ${ } 插入变量的值。

```
console.log(`${lastName}, ${firstName} ${middleName}`);
```

任何返回一个值的 JavaScript 代码都可以放入 ${ }，添加到模板字符串中。

模板字符串保留空白，适合用于创建电子邮件模板、代码示例，以及其他包含空白的内容。现在，一个字符串可以横跨多行，而不会导致代码出错。

```
const email = `
Hello ${firstName},

Thanks for ordering ${qty} tickets to ${event}.

Order Details
${firstName} ${middleName} ${lastName}
     ${qty} x $${price} = $${qty*price} to ${event}

You can pick your tickets up 30 minutes before
the show.

Thanks,

${ticketAgent}
```

以前，在 JavaScript 代码中直接使用 HTML 字符串不是一件易事，要把所有内

容写在一行内。现在，空白也作为文本的一部分，我们便可以直接插入格式化的 HTML，这样易于阅读和理解。

```
document.body.innerHTML = `
<section>
  <header>
      <h1>The React Blog</h1>
  </header>
  <article>
      <h2>${article.title}</h2>
      ${article.body}
  </article>
  <footer>
      <p>copyright ${new Date().getYear()} | The React Blog</p>
  </footer>
</section>
`;
```

注意，页面标题和文章内容还可以使用变量指定。

2.2 创建函数

只要想让 JavaScript 重复执行一项任务，就可以使用函数。本节说明用于创建函数的不同句法，以及各自的适用场景。

2.2.1 函数声明

函数声明或函数定义以 `function` 关键字开头，后跟函数的名称，例如 `logCompliment`。函数所含的 JavaScript 语句在一对花括号之间定义。

```
function logCompliment() {
  console.log("You're doing great!");
}
```

函数在声明之后便可以调用、执行。

```
function logCompliment() {
  console.log("You're doing great!");
}

logCompliment();
```

调用后，你会在控制台中看到那句表扬你的话。

2.2.2 函数表达式

另一种选择是使用函数表达式。这种方式通过变量创建函数。

```
const logCompliment = function() {
  console.log("You're doing great!");
};

logCompliment();
```

结果没变，控制台输出“You're doing great!”。

在函数声明和函数表达式之间抉择时要明确一点：函数声明的作用域会被提升，而函数表达式不会。换句话说，在编写函数声明之前可以调用函数，而在创建函数表达式之前不能调用函数，否则会报错。例如：

```
// 在声明函数之前调用函数
hey();
// 函数声明
function hey() {
  alert("hey!");
}
```

这样做是可以的。你会在浏览器中看到弹出框。这是因为函数的作用域被提升了，移到了文件作用域的顶部。如果以同样的方式调用函数表达式创建的函数则会报错。

```
// 在声明函数之前调用函数
hey();
```

```
// 函数表达式
const hey = function() {
  alert("hey!");
};
TypeError: hey is not a function
```

这个简单的示例很明显，不过在项目中导入文件和函数时偶尔也会遇到这种TypeError。遇到这种错误，重构一下，改成函数声明即可。

传递参数

前面声明的logCompliment函数不接受任何参数。如果想为函数提供动态变量，可以把具名参数传给函数，放在圆括号之间即可。先添加一个firstName变量：

```
const logCompliment = function(firstName) {
  console.log(`You're doing great, ${firstName}`);
};

logCompliment("Molly");
```

现在，调用logCompliment函数后，传入的firstName值将出现在控制台消息中。

在此之上，我们还可以再添加一个名为message的参数，不再硬编码消息，而是传入动态的值。

```
const logCompliment = function(firstName, message) {
  console.log(`${firstName}: ${message}`);
};

logCompliment("Molly", "You're so cool");
```

函数返回值

logCompliment函数把消息输出到控制台中，不过更多时候，我们希望函数返回一个值。下面为这个函数添加一个return语句。return语句的作用是指定函数的返回值。我们将把这个函数的名称改为createCompliment。

```
const createCompliment = function(firstName, message) {
  return `${firstName}: ${message}`;
};

createCompliment("Molly", "You're so cool");
```

如果你想检查函数有没有按预期执行，可以把函数调用放入 console.log 中：

```
console.log(createCompliment("You're so cool", "Molly"));
```

2.2.3 默认参数

C++ 和 Python 等语言允许开发者为函数的参数声明默认值。默认参数是 ES6 规范引入的特性，如果没有为参数提供值，将使用默认值。

例如，可以为 name 和 activity 两个参数设置默认的字符串值：

```
function logActivity(name = "Shane McConkey", activity = "skiing") {
  console.log(`${name} loves ${activity}`);
}
```

如果没有为 logActivity 函数提供参数，它仍能正常运行，只不过使用的是默认值。默认参数可以是任何类型，不限于字符串。

```
const defaultPerson = {
  name: {
    first: "Shane",
    last: "McConkey"
  },
  favActivity: "skiing"
};

function logActivity(person = defaultPerson) {
  console.log(`${person.name.first} loves ${person.favActivity}`);
}
```

2.2.4 箭头函数

箭头函数是 ES6 新增的一个特性，非常有用。有了箭头函数，创建函数不再需要使用 function 关键字了。另外，也经常不使用 return 关键字。例如，我们想创建一个函数，接受一个 firstName 参数，返回一个字符串，把一个人变成一地的领主。借助这个函数，任何人都可以变成领主。

```
const lordify = function(firstName) {
  return `${firstName} of Canterbury`;
};

console.log(lordify("Dale")); // Dale of Canterbury
console.log(lordify("Gail")); // Gail of Canterbury
```

使用箭头函数可以极大地简化句法：

```
const lordify = firstName => `${firstName} of Canterbury`;
```

现在，整个函数在一行内就声明完毕了，去掉了 function 关键字，也去掉了 return 关键字，箭头指明的就是要返回的值。这种写法的另一个好处是，如果函数只接受一个参数，可以省略参数两侧的圆括号。

不过，参数超过一个时要放入圆括号中。

```
// 常规函数
const lordify = function(firstName, land) {
  return `${firstName} of ${land}`;
};
// 箭头函数
const lordify = (firstName, land) => `${firstName} of ${land}`;

console.log(lordify("Don", "Piscataway")); // Don of Piscataway
console.log(lordify("Todd", "Schenectady")); // Todd of Schenectady
```

这个函数可以写在一行内，毕竟只需要返回一个语句。如果有多行，要使用花括号。

```
const lordify = (firstName, land) => {
  if (!firstName) {
    throw new Error("A firstName is required to lordify");
  }

  if (!land) {
    throw new Error("A lord must have a land");
  }

  return `${firstName} of ${land}`;
};

console.log(lordify("Kelly", "Sonoma")); // Kelly of Sonoma
console.log(lordify("Dave")); // ! JAVASCRIPT ERROR
```

上述代码片段中的 `if/else` 语句放在花括号内，由于使用的是箭头函数，句法依然很简洁。

返回对象

如果想返回一个对象呢？假如有个名为 `person` 的函数，根据传入的 `firstName` 和 `lastName` 参数构建一个对象。

```
const person = (firstName, lastName) =>
    {
        first: firstName,
        last: lastName
    }

console.log(person("Brad", "Janson"));
```

现在运行这段代码，你会看到一个错误：`Uncaught SyntaxError: Unexpected token :`。修正的方法是，把要返回的对象放在圆括号中。

```
const person = (firstName, lastName) => ({
  first: firstName,
  last: lastName
});
```

```
console.log(person("Flad", "Hanson"));
```

这里缺少的圆括号是 JavaScript 和 React 应用中无数 bug 的根源，千万别忘了！

箭头函数和作用域

常规的函数不限定 this 的作用域。以下述代码为例，在 setTimeout 回调中，this 变成了别的东西，不再是 tahoe 对象。

```
const tahoe = {
  mountains: ["Freel", "Rose", "Tallac", "Rubicon", "Silver"],
  print: function(delay = 1000) {
    setTimeout(function() {
      console.log(this.mountains.join(", "));
    }, delay);
  }
};

tahoe.print(); // Uncaught TypeError: Cannot read property 'join' of undefined
```

出现这个错误的原因在于试图在 this 上调用 .join 方法。在控制台中输出 this，可以看到它引用的是 Window 对象。

```
console.log(this); // Window {}
```

为了解决这个问题，我们可以使用箭头函数句法保全 this 的作用域。

```
const tahoe = {
  mountains: ["Freel", "Rose", "Tallac", "Rubicon", "Silver"],
  print: function(delay = 1000) {
    setTimeout(() => {
      console.log(this.mountains.join(", "));
    }, delay);
  }
};

tahoe.print(); // Freel, Rose, Tallac, Rubicon, Silver
```

现在正常了，我们可以使用逗号把几个度假村连在一起了。务必时刻考虑作用域。箭头函数不限定 this 的作用域。

```
const tahoe = {
  mountains: ["Freel", "Rose", "Tallac", "Rubicon", "Silver"],
  print: (delay = 1000) => {
    setTimeout(() => {
      console.log(this.mountains.join(", "));
    }, delay);
  }
};

tahoe.print(); // Uncaught TypeError: Cannot read property 'join' of undefined
```

把 print 函数改成箭头函数后，this 引用的就是窗口了。

2.3 编译 JavaScript

一项 JavaScript 新特性提出并获得一定的支持后，社区通常想在所有浏览器都支持之前就开始使用。为了确保你写出的代码能正常在浏览器中运行，要把代码转换成兼容性更好的版本。这个过程叫作编译（compile）。最常用于编译 JavaScript 代码的工具之一是 Babel（*http://www.babeljs.io*）。

过去，若想使用最新的 JavaScript 特性，只有等待，几周、几个月，甚至几年，直到浏览器支持。现在，借助 Babel 可以立即就使用最新的 JavaScript 特性。加入编译步骤之后，JavaScript 与其他语言更像了。然而，这与常规的编译还不一样，代码不被编译成二进制文件，而是转换成更多浏览器可以解析的句法。另外，JavaScript 现在有源码了，这意味着，项目中有些文件不会在浏览器中运行。

以一个接受默认参数的箭头函数为例：

```
const add = (x = 5, y = 10) => console.log(x + y);
```

使用 Babel 编译这段代码，生成的代码如下所示。

```
"use strict";

var add = function add() {
  var x =
    arguments.length <= 0 || arguments[0] === undefined ? 5 : arguments[0];
  var y =
    arguments.length <= 1 || arguments[1] === undefined ? 10 : arguments[1];
  return console.log(x + y);
};
```

Babel 添加了“use strict”声明，在严格模式下运行代码。变量 x 和 y 通过 `arguments` 数组（你可能熟悉这个技术）模拟成默认参数。编译得到的 JavaScript 支持范围更广。

如果想进一步学习 Babel，请访问文档网站中的 Babel REPL（*https://babeljs.io/repl*）。在左侧输入一些新句法，这个工具就会给出旧句法。

编译 JavaScript 代码的过程通常由 webpack 或 Parcel 等构建工具自动操作。详见后文。

2.4 对象和数组

自 ES2016 起，JavaScript 句法新增了一些在对象和数组中限定变量作用域的创造性方式。这些创造性技术在 React 社区中广泛使用。本节介绍其中一些新语法，包括析构、对象字面量增强和展开运算符。

2.4.1 析构对象

析构赋值语句把对象中字段的作用域限定在本地，以及声明将用到哪些值。以下面的 `sandwich` 对象为例。这个对象有四个键，不过我们只想使用其中两个值。为此，可以把 `bread` 和 `meat` 的作用域限定在本地。

```
const sandwich = {
  bread: "dutch crunch",
  meat: "tuna",
```

```
  cheese: "swiss",
  toppings: ["lettuce", "tomato", "mustard"]
};

const { bread, meat } = sandwich;

console.log(bread, meat); // dutch crunch tuna
```

这段代码从对象中取出 bread 和 meat，创建两个局部变量。另外，由于这两个析构的变量是使用 let 声明的，因此 bread 和 meat 的变化对原 sandwich 对象没有影响。

```
const sandwich = {
  bread: "dutch crunch",
  meat: "tuna",
  cheese: "swiss",
  toppings: ["lettuce", "tomato", "mustard"]
};

let { bread, meat } = sandwich;

bread = "garlic";
meat = "turkey";

console.log(bread); // garlic
console.log(meat); // turkey

console.log(sandwich.bread, sandwich.meat); // dutch crunch tuna
```

此外，还可以析构传给函数的参数。以下述在控制台中输出领主名字的函数为例：

```
const lordify = regularPerson => {
  console.log(`${regularPerson.firstName} of Canterbury`);
};

const regularPerson = {
  firstName: "Bill",
  lastname: "Wilson"
};
```

```
lordify(regularPerson); // Bill of Canterbury
```

与其使用点号句法深入对象内部，我们可以从 regularPerson 对象中析构出需要使用的值：

```
const lordify = ({ firstName }) => {
  console.log(`${firstName} of Canterbury`);
};

const regularPerson = {
  firstName: "Bill",
  lastname: "Wilson"
};

lordify(regularPerson); // Bill of Canterbury
```

下面我们更进一步，修改一下数据。现在，regularPerson 对象内部嵌套了一个对象，键名为 spouse：

```
const regularPerson = {
  firstName: "Bill",
  lastname: "Wilson",
  spouse: {
    firstName: "Phil",
    lastname: "Wilson"
  }
};
```

如果想把配偶的名字也变成领主的形式，要稍微调整一下函数的析构参数：

```
const lordify = ({ spouse: { firstName } }) => {
  console.log(`${firstName} of Canterbury`);
};

lordify(regularPerson); // Phil of Canterbury
```

使用冒号和嵌套的花括号可以从 spouse 对象中析构出 firstname。

2.4.2 析构数组

值也可以从数组中析构出来。假设我们想把数组中的第一个值赋给一个变量。

```
const [firstAnimal] = ["Horse", "Mouse", "Cat"];

console.log(firstAnimal); // Horse
```

此外，还可以使用逗号跳过不需要的值，这叫列表匹配（list matching）。在需要跳过的元素位置上使用逗号执行的就是列表匹配。还以前面的数组为例，在前两个值的位置上使用逗号就可以获取第三个值：

```
const [, , thirdAnimal] = ["Horse", "Mouse", "Cat"];

console.log(thirdAnimal); // Cat
```

本节后面将更进一步，同时使用数组析构和展开运算符。

2.4.3 对象字面量增强

对象字面量增强与析构相反，它指把对象重新组合成一体。使用对象字面量增强可以把全局作用域中的变量添加到一个对象中。

```
const name = "Tallac";
const elevation = 9738;

const funHike = { name, elevation };

console.log(funHike); // {name: "Tallac", elevation: 9738}
```

name 和 elevation 现在是 funHike 对象的键。

此外，还可以使用对象字面量增强或重组创建对象方法。

```
const name = "Tallac";
const elevation = 9738;
const print = function() {
```

```
  console.log(`Mt. ${this.name} is ${this.elevation} feet tall`);
};

const funHike = { name, elevation, print };

funHike.print(); // Mt. Tallac is 9738 feet tall
```

注意，我们使用 this 访问对象的键。

定义对象方法时无须使用 function 关键字。

```
// 旧句法
var skier = {
  name: name,
  sound: sound,
  powderYell: function() {
    var yell = this.sound.toUpperCase();
    console.log(`${yell} ${yell} ${yell}!!!`);
  },
  speed: function(mph) {
    this.speed = mph;
    console.log("speed:", mph);
  }
};

// 新句法
const skier = {
  name,
  sound,
  powderYell() {
    let yell = this.sound.toUpperCase();
    console.log(`${yell} ${yell} ${yell}!!!`);
  },
  speed(mph) {
    this.speed = mph;
    console.log("speed:", mph);
  }
};
```

使用对象字面量增强可以把全局变量放到对象中，还可以省略 function 关键字，减少输入量。

2.4.4 展开运算符

展开运算符是三个点号（...），可用于执行几项不同的任务。首先，可以使用展开运算符合并数组的内容。假如有两个数组，我们可以把它们合并成一个数组，创建第三个数组：

```
const peaks = ["Tallac", "Ralston", "Rose"];
const canyons = ["Ward", "Blackwood"];
const tahoe = [...peaks, ...canyons];

console.log(tahoe.join(", ")); // Tallac, Ralston, Rose, Ward, Blackwood
```

peaks 和 canyons 中的所有元素都放入了新数组 tahoe 中。

下面来看展开运算符可以解决的另一个问题。以前例中的 peaks 数组为例，这一次我们不想获取第一个元素，而是获取最后一个元素。为此，可以使用 Array.reverse 方法反转数组，再析构数组：

```
const peaks = ["Tallac", "Ralston", "Rose"];
const [last] = peaks.reverse();

console.log(last); // Rose
console.log(peaks.join(", ")); // Rose, Ralston, Tallac
```

看到了吗？ reverse 函数其实改变了原数组。在有展开运算符的情况中，我们无须改变原数组，而是创建一个副本，再做反转。

```
const peaks = ["Tallac", "Ralston", "Rose"];
const [last] = [...peaks].reverse();

console.log(last); // Rose
console.log(peaks.join(", ")); // Tallac, Ralston, Rose
```

由于我们使用展开运算符创建了一个副本，peaks 数组完好无损，仍可以原来的形式继续使用。

展开运算符也可用于获取数组中剩余的元素：

```
const lakes = ["Donner", "Marlette", "Fallen Leaf", "Cascade"];

const [first, ...others] = lakes;

console.log(others.join(", ")); // Marlette, Fallen Leaf, Cascade
```

我们还可以使用三个点号句法把函数的参数收集到一个数组中。在函数中，这叫剩余参数（rest parameters）。这里，我们使用展开运算符定义一个可接受 *n* 个参数的函数，然后使用这些参数打印一些控制台消息。

```
function directions(...args) {
  let [start, ...remaining] = args;
  let [finish, ...stops] = remaining.reverse();

  console.log(`drive through ${args.length} towns`);
  console.log(`start in ${start}`);
  console.log(`the destination is ${finish}`);
  console.log(`stopping ${stops.length} times in between`);
}

directions("Truckee", "Tahoe City", "Sunnyside", "Homewood", "Tahoma");
```

directions 函数通过展开运算符接受传入的各个参数。第一个参数赋值给 start 变量，最后一个参数赋值给 finish 变量（借助 Array.reverse）。然后，我们使用 arguments 数组的长度显示要经过多少小镇。中途停靠的小镇数量是 arguments 数组的长度减去 finish 这一站。这样做十分灵活，可以使用 directions 函数处理任意多个停靠站。

展开运算符还可用于处理对象（参见 GitHub 中 Rest/Spread Properties 项目的页面，*https://oreil.ly/kCpEL*）。使用展开运算符处理对象的方式与处理数组类似。在下面的示例中，就像前面使用它合并两个数组一样，不过这一次是合并对象。

```
const morning = {
  breakfast: "oatmeal",
  lunch: "peanut butter and jelly"
};

const dinner = "mac and cheese";

const backpackingMeals = {
  ...morning,
  dinner
};

console.log(backpackingMeals);

// {
//   breakfast: "oatmeal",
//   lunch: "peanut butter and jelly",
//   dinner: "mac and cheese"
// }
```

2.5 JavaScript 异步编程

截止目前，本章给出的代码示例都是同步的。我们编写的同步 JavaScript 代码是一系列按照顺序立即执行的指令。比如，使用 JavaScript 做简单的 DOM 处理，可能会写出如下的代码。

```
const header = document.getElementById("heading");
header.innerHTML = "Hey!";
```

这些就是指令。“去选择 ID 为 heading 的元素。然后，把那个元素内的 HTML 设为‘Hey’吧。”这一切都是同步的。一项操作在执行的过程中，不执行其他任何操作。

在现代的 Web 中，离不开异步任务。这种任务通常需要等到其他作业结束才能完成。我们可能需要访问数据库，可能需要缓存视频或音频流内容，可能需要从 API 中获取数据。在 JavaScript 中，异步任务不阻塞主线程。在等待 API 返回数据的时段内，JavaScript 是空闲的，可以去做其他事。过去几年，JavaScript

取得了长足的发展，进一步简化了异步操作的处理机制。本节介绍与异步编程有关的部分特性。

2.5.1 使用 fetch 处理简单的 promise

以前，请求 REST API 是一件非常复杂的事情。为了把数据加载到应用中，要编写 20 多行嵌套的代码。后来，`fetch()` 函数的出现拯救了我们。感谢 ECMAScript 委员会的这一善举。

我们从 randomuser.me API 获取一些数据。这个 API 提供了电子邮件地址、姓名、电话号码、地理位置等成员信息，可用作虚拟数据。fetch 函数只接受一个参数，即资源的 URL。

```
console.log(fetch("https://api.randomuser.me/?nat=US&results=1"));
```

在控制台中可以看到输出了一个挂起的 promise。*promise* 为理解 JavaScript 中的异步行为提供了一种方式。promise 是一个对象，表示异步操作的状态是挂起、已完成或是失败。这就好像是浏览器告诉你，“嗨，我会尽自己所能获取这个数据。不管成功与否，我都会回过头来告诉你进展的。”

回来再看 `fetch` 的结果。挂起的 promise 表示获取数据之前的状态。我们要串接一个名为 `.then()` 的函数。这个函数接受一个回调函数，在前一步操作执行成功后运行。也就是先获取一些数据，然后再做其他操作。

这里，我们想做的其他操作是把响应变成 JSON 格式。

```
fetch("https://api.randomuser.me/?nat=US&results=1").then(res =>
  console.log(res.json())
);
```

`then` 方法在 promise 得到成功处理后调用回调函数。不管回调函数返回什么，都将作为下一个 `then` 函数的参数。因此，我们可以串接多个 `then` 函数处理成功的 promise。

```
fetch("https://api.randomuser.me/?nat=US&results=1")
  .then(res => res.json())
  .then(json => json.results)
  .then(console.log)
  .catch(console.error);
```

首先，使用 `fetch` 向 randomuser.me 发起一个 GET 请求。如果请求成功，把响应主体转换成 JSON 格式。接下来，返回得到的 JSON 结构，再把结果发给 `console.log` 函数，输出到控制台。最后有个 `catch` 函数，在 `fetch` 没有得到成功处理时调用指定的回调。在从 randomuser.me 获取数据的过程中出现的所有错误都将根据这个回调进行处理。这里，我们只是使用 `console.error` 把错误输出到控制台。

2.5.2 async/await

处理 promise 的另一种常用方式是创建异步函数。一些开发者倾向使用异步函数句法，因为所用的句法较熟悉，就像同步函数的代码一样。此时，我们不等待 promise 成功处理后返回结果再调用一系列 `then` 函数了，而是让异步函数暂停执行函数中的代码，直到 promise 处理成功。

下面再发起一个 API 请求，不过这一次把整个过程放到一个异步函数中。

```
const getFakePerson = async () => {
  let res = await fetch("https://api.randomuser.me/?nat=US&results=1");
  let { results } = res.json();
  console.log(results);
};

getFakePerson();
```

注意，`getFakePerson` 函数是使用 `async` 关键字声明的。这样定义的才是异步函数，才会在成功处理 promise 之前暂停执行后续代码。promise 调用前面要加上 `await` 关键字，让异步函数等待 promise 得到成功处理。上述代码完成的任务与前一节使用 `then` 函数的那段代码完全一样。好吧，几乎完全一样……

```
const getFakePerson = async () => {
  try {
    let res = await fetch("https://api.randomuser.me/?nat=US&results=1");
    let { results } = res.json();
    console.log(results);
  } catch (error) {
    console.error(error);
  }
};

getFakePerson();
```

好了，现在这段代码完成的任务真正与前一节使用 then 函数的那段代码完全一样了。如果 fetch 调用处理成功，把结果输出到控制台；如果不成功，使用 console.error 把错误输出到控制台。使用 async 和 await 时，要把 promise 调用放在 try...catch 块中，处理 promise 失败时可能出现的错误。

2.5.3 构建 promise

异步请求的结果有两种可能：一切如我们所愿，或者出现错误。成功和失败的请求有多种不同的类型。例如，我们可以不断尝试获取数据，直至成功。收到的错误也可能有多种类型。promise 提供了一种处理方式，可以把这种复杂的情况简化为一次成功或失败。

下面的 getPeople 函数返回一个新的 promise。这个 promise 向 API 发起请求，如果成功，加载数据；如果不成功，返回错误。

```
const getPeople = count =>
  new Promise((resolves, rejects) => {
    const api = `https://api.randomuser.me/?nat=US&results=${count}`;
    const request = new XMLHttpRequest();
    request.open("GET", api);
    request.onload = () =>
      request.status === 200
        ? resolves(JSON.parse(request.response).results)
        : reject(Error(request.statusText));
    request.onerror = err => rejects(err);
```

```
    request.send();
  });
```

这种情况下，虽然创建了 promise，但并没有使用。若想使用这个 promise，要调用 `getPeople` 函数，传入要加载的成员数量。我们还可以串接 `then` 函数，在这个 promise 处理成功后做些操作。如果 promise 被拒，相关细节会发回给 `catch` 函数或 `async/await` 句法中的 `catch` 块。

```
getPeople(5)
  .then(members => console.log(members))
  .catch(error => console.error(`getPeople failed: ${error.message}`))
);
```

promise 简化了异步请求的处理过程，这是一件好事，因为在 JavaScript 中经常需要处理异步任务。对现代的 JavaScript 工程技术人员来说，扎实地理解异步行为是必备技能。

2.6 类

在 ES2015 之前，JavaScript 规范没有官方的类句法。引入类之后，很多人非常高兴，因为类的句法与传统的面向对象语言（例如 Java 和 C++）十分相似。过去几年，React 库慢慢向类倾斜，大量使用类构建用户界面组件。然而，现在 React 开始抛弃类了，转而使用函数构建组件。但是，你还会经常见到类，尤其是在旧的 React 代码和 JavaScript 世界中。那就简单介绍一下类吧。

JavaScript 使用原型继承实现一种类似面向对象的结构。例如，我们可以创建一个需要使用 `new` 运算符调用的 `Vacation` 构造函数。

```
function Vacation(destination, length) {
  this.destination = destination;
  this.length = length;
}

Vacation.prototype.print = function() {
```

```
  console.log(this.destination + " | " + this.length + " days");
};

const maui = new Vacation("Maui", 7);

maui.print(); // Maui | 7 days
```

这段代码实现的效果有点类似面向对象语言中的自定义类型。Vacation 有属性（destination、length），有一个方法（print）。maui 实例通过原型继承 print 方法。如果你习惯了较为标准的类，这或许会让你火冒三丈。为了平息不满，ES2015 引入了类声明句法，可现实是，JavaScript 的运作方式并没有变。函数还是对象，继承仍通过原型处理。类就像是在糟糕的原型句法上套了一层句法糖。

```
class Vacation {
  constructor(destination, length) {
    this.destination = destination;
    this.length = length;
  }

  print() {
    console.log(`${this.destination} will take ${this.length} days.`);
  }
}
```

创建类时，类名通常使用首字母大写形式。声明好类之后，可以使用 new 关键字创建类的实例。然后就可以调用类中定义的方法了。

```
const trip = new Vacation("Santiago, Chile", 7);

trip.print(); // Chile will take 7 days.
```

创建一个类对象之后，可以任意创建新的度假行程，想使用多少次都可以。类还可以扩展。扩展类得到的子类自超类继承属性和方法。子类也可以按照自己的方式处理属性和方法，不过默认是直接继承。

你可以把 Vacation 当成一个抽象类，在此基础上创建不同的度假类型。例如，可以扩展 Vacation 类得到 Expedition 类，增加装备。

```
class Expedition extends Vacation {
  constructor(destination, length, gear) {
    super(destination, length);
    this.gear = gear;
  }

  print() {
    super.print();
    console.log(`Bring your ${this.gear.join(" and your ")}`);
  }
}
```

这是简单的继承，子类从超类继承属性。除了只调用 Vacation 类的 print 方法，我们还可以在 Expedition 类的 print 方法中添加新的打印内容。创建实例的方法不变，使用 new 关键字创建一个变量即可。

```
const trip = new Expedition("Mt. Whitney", 3, [
  "sunglasses",
  "prayer flags",
  "camera"
]);

trip.print();

// Mt. Whitney will take 3 days.
// Bring your sunglasses and your prayer flags and your camera
```

2.7 ES6 模块

JavaScript 模块（module）是一组可重用的代码，方便插入其他 JavaScript 文件而不产生变量冲突。JavaScript 模块保存在单独的文件中，一个文件一个模块。创建和导出模块有两种选择：从一个模块中导出多个 JavaScript 对象，或者从一个模块中导出一个 JavaScript 对象。

下面的 *text-helpers.js* 模块导出了两个函数：

```
export const print=(message) => log(message, new Date())
```

```
export const log=(message, timestamp) =>
  console.log(`${timestamp.toString()}: ${message}`)
```

export 可以导出供其他模块使用的任何 JavaScript 类型。在这个示例中，导出的是 print 和 log 两个函数。*text-helpers.js* 中声明的其他变量都只存在于那个模块。

模块也可以只导出一个主变量。此时，使用 export default。例如，*mt-freel.js* 可以导出一个特定的探险队：

```
export default new Expedition("Mt. Freel", 2, ["water", "snack"]);
```

如果只想导出一个类型，就可以把 export 换成 export default。同样，export 和 export default 都可用于导出任何 JavaScript 类型：原始类型、对象、数组和函数。

在其他文件中使用一个模块，通过 import 语句导入。导出多个对象的模块可以充分利用对象析构。使用 export default 的模块导入为一个变量：

```
import { print, log } from "./text-helpers";
import freel from "./mt-freel";

print("printing a message");
log("logging a message");

freel.print();
```

导入的模块变量在当前模块中可以放在其他变量名下：

```
import { print as p, log as l } from "./text-helpers";

p("printing a message");
l("logging a message");
```

此外，也可以使用 * 把一切都导入为一个变量：

```
import * as fns from './text-helpers'
```

import 和 export 句法还没有得到所有浏览器及 Node 的完全支持。然而，与其他新兴的 JavaScript 句法一样，Babel 支持。这意味着，你可以在自己的源码中使用这些语句，Babel 知道在何处寻找你要使用的模块，把找到的模块添加到编译得到的 JavaScript 代码中。

CommonJS

CommonJS 是所有 Node 版本都支持的模块模式（参见 Node.js 模块文档，*https://oreil.ly/CN-gA*）。Babel 和 webpack 也支持这种模块。在 CommonJS 中，JavaScript 对象使用 module.exports 导出。

例如，在 CommonJS 中可以把 print 和 log 函数导出为对象：

```
const print(message) => log(message, new Date())

const log(message, timestamp) =>
console.log(`${timestamp.toString()}: ${message}`)

module.exports = {print, log}
```

CommonJS 不支持 import 语句，模块使用 require 函数导入。

```
const { log, print } = require("./txt-helpers");
```

JavaScript 发展十分迅速，适应了工程技术人员对这门语言日益增长的要求，而且浏览器也会积极实现新特性。如果想了解最新的兼容信息，请查看 ESNext 兼容表（*https://oreil.ly/rxTcg*）。本章介绍了 JavaScript 最新句法的很多特性，它们都支持函数式编程技术。在函数式 JavaScript 编程中，我们把组成应用的代码分解为一系列函数。下一章深入探讨函数式编程技术，并说明使用这些技术的缘由。

第 3 章

JavaScript 函数式编程

踏上 React 之旅后，你会发现函数式编程概念随处可见。越来越多的 JavaScript 项目开始采用函数式编程技术，React 项目尤甚。

不经意间，你可能已经编写过函数式 JavaScript 代码。如果你使用 map 或 reduce 处理过数组，那么你就已经走上了函数式 JavaScript 程序员的道路。函数式编程技术不仅是 React 的核心思想，也是 React 生态系统中很多库的重要编程手段。

如果你想溯源函数式编程风格，笔者来告诉你：一切始于 20 世纪 30 年代发明的 *lambda* 演算（lambda calculus 或 λ-calculus）。[注 1] 自 17 世纪诞生以来，函数始终是微积分的一个部分。函数可以发给函数，用作参数，也可以当作结果由函数返回。更为复杂的函数称为高阶函数（higher-order function），可以处理函数，把函数当作参数或返回结果，或者二者兼具。20 世纪 30 年代，Alonzo Church 在普林斯顿大学求学时，在使用高阶函数做实验的过程中发明了 lambda 演算。

20 世纪 50 年代后期，John McCarthy 在 λ 演算的基础上发明了一门新编程语言，

注 1：Dana S. Scott，“λ-Calculus: Then & Now”，*https://oreil.ly/k0EpX*。

即 Lisp。Lisp 实现了高阶函数的概念，把函数当作一等成员（或称一等公民）。如果函数可以声明为变量，作为参数发给函数，那这个函数就是一等成员。这种函数甚至还可以由函数返回。

本章探讨函数式编程的一些重要概念，还将介绍如何使用 JavaScript 实现函数式编程技术。

3.1 函数式编程是什么意思

JavaScript 支持函数式编程，因为在 JavaScript 中，函数是一等成员。这意味着，变量可以做到的事函数都能做到。最新的 JavaScript 句法对这门语言做了一些改进，使用函数式编程技术更方便了，例如箭头函数、promise 和展开运算符。

在 JavaScript 中，函数可以表示应用的数据。你应该已经注意到，函数可以使用 var、let 或 const 关键字声明，就像声明字符串、数字等变量一样。

```
var log = function(message) {
  console.log(message);
};

log("In JavaScript, functions are variables");

// In JavaScript, functions are variables
```

这个函数还可以使用箭头函数句法编写。函数式程序员经常编写小巧的函数，箭头函数句法大大简化了这个过程。

```
const log = message => {
  console.log(message);
};
```

由于函数是变量，那就可以把函数添加到对象中。

```
const obj = {
  message: "They can be added to objects like variables",
```

```
  log(message) {
    console.log(message);
  }
};

obj.log(obj.message);

// They can be added to objects like variables
```

这两种写法的效果是一样的，都把一个函数存储在一个名为 log 的变量中。不过，第二个函数是使用 const 关键字声明的，从而避免被覆盖。

此外，还可以把函数添加到 JavaScript 数组中。

```
const messages = [
  "They can be inserted into arrays",
  message => console.log(message),
  "like variables",
  message => console.log(message)
];

messages[1](messages[0]); // They can be inserted into arrays
messages[3](messages[2]); // like variables
```

函数可以像常规变量一样作为参数发给其他函数。

```
const insideFn = logger => {
  logger("They can be sent to other functions as arguments");
};

insideFn(message => console.log(message));

// They can be sent to other functions as arguments
```

函数也可以像变量一样由其他函数返回。

```
const createScream = function(logger) {
  return function(message) {
    logger(message.toUpperCase() + "!!!");
```

```
  };
};

const scream = createScream(message => console.log(message));

scream("functions can be returned from other functions");
scream("createScream returns a function");
scream("scream invokes that returned function");

// FUNCTIONS CAN BE RETURNED FROM OTHER FUNCTIONS!!!
// CREATESCREAM RETURNS A FUNCTION!!!
// SCREAM INVOKES THAT RETURNED FUNCTION!!!
```

上述两个示例用到了高阶函数，即接受函数为参数或返回其他函数。高阶函数 createScream 还可以使用箭头句法表示：

```
const createScream = logger => message => {
  logger(message.toUpperCase() + "!!!");
};
```

如果你在函数声明中见到多个箭头，那就说明使用的是高阶函数。

可以断定的是，JavaScript 支持函数式编程，因为函数是一等公民。这意味着，函数是数据，可以像变量一样保存、读取或在应用中流动。

3.2 命令式和声明式

函数式编程是一个更大编程范式的一部分，即声明式编程（declarative programming）。声明式编程是一种编程风格，采用这种风格开发的应用有个显著特点：重点描述该做什么，而不管怎么做。

为了充分理解声明式编程，要与命令式编程（imperative programming）对比一下。命令式编程风格只关注如何使用代码得到结果。以一个常见的任务为例：让字符串符合 URL 格式要求。通常，我们要把字符串中的空格替换成连字符，因为空格不符合 URL 的格式要求。先使用命令式风格解决这个任务。

```
const string = "Restaurants in Hanalei";
const urlFriendly = "";

for (var i = 0; i < string.length; i++) {
  if (string[i] === " ") {
    urlFriendly += "-";
  } else {
    urlFriendly += string[i];
  }
}

console.log(urlFriendly); // "Restaurants-in-Hanalei"
```

在这个示例中，我们迭代字符串中的每个字符，把找到的空格替换掉。这个程序的结构只关注如何解决此项任务。我们用到了一个 `for` 循环和一个 `if` 语句，通过等值运算符设定值。表面上看，从代码中得不到多少信息。采用命令式编程风格的程序需要大量注释，这样才能理解来龙去脉。

下面来看使用声明式编程风格完成同一项任务的代码。

```
const string = "Restaurants in Hanalei";
const urlFriendly = string.replace(/ /g, "-");

console.log(urlFriendly);
```

这里，我们使用 `string.replace`，通过正则表达式把所有空格替换成连字符。使用 `string.replace` 就是在描述应该做什么操作：把字符串中的空格替换掉。具体如何处理空格则隐藏在 `replace` 函数中。在采用声明式编程风格的程序中，句法本身描述的是应该做什么事，至于怎么做则被隐藏起来了。

采用声明式编程风格的程序易于理解，因为代码自身就说明了在做什么。请阅读下述示例中的句法。这段代码详细描述了从 API 中加载 members 后做了什么。

```
const loadAndMapMembers = compose(
  combineWith(sessionStorage, "members"),
  save(sessionStorage, "members"),
  scopeMembers(window),
```

```
  logMemberInfoToConsole,
  logFieldsToConsole("name.first"),
  countMembersBy("location.state"),
  prepStatesForMapping,
  save(sessionStorage, "map"),
  renderUSMap
);

getFakeMembers(100).then(loadAndMapMembers);
```

声明式方式易于阅读，因此也更容易理解。各函数是如何实现的都隐藏起来了。这些小巧的函数命名合理，而且以一种清晰明了的方式组合在一起，一眼就能看出成员数据在加载后保存起来了，然后又经过映射再打印出来。这种方式不需要多少注释。本质上说，采用声明式编程开发出来的应用更易于理解，因而也就更易于弹性伸缩。关于声明式编程范式的进一步说明参见“Declarative Programming”的维基（*https://oreil.ly/7MbkB*）。

下面再来看一个任务：构建文档对象模型（document object model，DOM，*https://www.w3.org/DOM*）。命令式方式关注的是如何构建 DOM：

```
const target = document.getElementById("target");
const wrapper = document.createElement("div");
const headline = document.createElement("h1");

wrapper.id = "welcome";
headline.innerText = "Hello World";

wrapper.appendChild(headline);
target.appendChild(wrapper);
```

这段代码涉及创建元素、设置元素，以及把元素添加到文档中。这样一来就很难做出改动、添加功能，或者扩充到 10000 行代码。

下面来看如何使用 React 组件以声明式方式构建 DOM：

```
const { render } = ReactDOM;
```

```
const Welcome = () => (
  <div id="welcome">
    <h1>Hello World</h1>
  </div>
);

render(<Welcome />, document.getElementById("target"));
```

React 是声明式的。这里，Welcome 组件描述了如何渲染 DOM。render 函数使用组件中声明的指令构建 DOM，隐藏了如何渲染 DOM 的细节信息。可以清楚地看出，我们想做的是在 ID 为 target 的元素中渲染 Welcome 组件。

3.3 函数式编程基本概念

现在，你对函数式编程有了一定的了解，也知道“函数式”或“声明式”的意思。接下来介绍函数式编程的一些核心概念：不可变性、纯函数、数据转换、高阶函数和递归。

3.3.1 不可变性

可变指可以更改，那么不可变就是不可更改。在采用函数式风格的程序中，数据是不可变的，永不更改。

如果你需要对外公开自己的出生证明，但是想去掉隐私信息，你有两种选择：拿一只大号马克笔把出生证明上的隐私信息涂掉，或者找一台复印机。复制一份出生证明，然后用大号马克笔涂改副本是更好的选择。这样既处理了出生证明，也保障了原件完好无损。

应用中的不可变数据就是这个意思。我们不直接更改原始数据结构，而是创建数据结构的副本，所有操作都使用副本。

为了理解不可变思想，我们来看更改数据是什么情况。假如有一个对象表示草绿色：

```
let color_lawn = {
  title: "lawn",
  color: "#00FF00",
  rating: 0
};
```

我们可以构建一个函数为颜色打分，使用这个函数更改 color 对象的评分。

```
function rateColor(color, rating) {
  color.rating = rating;
  return color;
}

console.log(rateColor(color_lawn, 5).rating); // 5
console.log(color_lawn.rating); // 5
```

在 JavaScript 中，函数的参数是对真正数据的引用。像前例那样设置颜色的评分会更改或改变原颜色对象（假如你委托一个商家处理和分享了自己的出生证明，而商家归还的出生证明原件上用黑色马克笔遮盖住了重要的信息。你肯定希望商家有一定的常识，知道应该复制一份出生证明，归还完好无损的原件）。我们可以重新编写 rateColor 函数，不损坏原件（color 对象）。

```
const rateColor = function(color, rating) {
  return Object.assign({}, color, { rating: rating });
};

console.log(rateColor(color_lawn, 5).rating); // 5
console.log(color_lawn.rating); // 0
```

这里，我们使用 Object.assign 更改颜色的评分。Object.assign 就是一台复印机。它创建一个空对象，复制颜色对象，然后覆盖副本的评分。这样，我们便得到了一个更改了评分的颜色对象，而原对象没有被更改。

这个函数还可以使用箭头函数句法和对象展开运算符编写。这一版 rateColor 函数使用展开运算符把颜色对象复制到一个新对象中，然后覆盖评分：

```
const rateColor = (color, rating) => ({
```

```
    ...color,
    rating
  });
```

这一版 rateColor 函数的作用与前一版完全一样，把颜色视为不可变对象，句法更少，看着稍微清爽一些。注意，返回的对象在一对圆括号内。使用箭头函数时一定要这样做，因为箭头不能直接指向一个对象的花括号。

再来看一个存储颜色名称的数组：

```
let list = [{ title: "Rad Red" }, { title: "Lawn" }, { title: "Party Pink" }];
```

我们可以定义一个函数，使用 Array.push 把颜色添加到这个数组中：

```
const addColor = function(title, colors) {
  colors.push({ title: title });
  return colors;
};

console.log(addColor("Glam Green", list).length); // 4
console.log(list.length); // 4
```

然而，Array.push 不是一个不可变函数。这个 addColor 函数为数组添加了新元素，更改了原数组。为了不更改 colors 数组，必须使用 Array.concat：

```
const addColor = (title, array) => array.concat({ title });

console.log(addColor("Glam Green", list).length); // 4
console.log(list.length); // 3
```

Array.concat 的作用是拼接数组。这里，concat 方法是接受一个由新颜色的标题构成的对象，把它添加到原数组的副本中。

与复制对象一样，我们还可以使用展开运算符拼接数组。下述代码片段使用最新的 JavaScript 句法重新定义前面的 addColor 函数：

```
const addColor = (title, list) => [...list, { title }];
```

这一版把原列表复制到一个新数组中，然后把一个包含颜色标题的对象添加到副本中。这就体现了不可变性。

3.3.2 纯函数

纯函数（pure function）指基于参数做计算，并返回一个值的函数。纯函数至少接受一个参数，而且始终返回一个值或另一个函数。这种函数没有副作用、不设置全局变量，也不更改应用的状态。纯函数把参数视为不可变数据。

为了理解纯函数，先来看一个非纯函数：

```
const frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite: false
};

function selfEducate() {
  frederick.canRead = true;
  frederick.canWrite = true;
  return frederick;
}

selfEducate();
console.log(frederick);

// {name: "Frederick Douglass", canRead: true, canWrite: true}
```

`selfEducate`函数不是纯函数。它不接受任何参数，也没有返回一个值或一个函数，而且还更改了作用域之外的一个变量，即`frederick`。只要调用`selfEducate`函数，“外面的世界”就变了。这个函数有副作用：

```
const frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite: false
};
```

```
const selfEducate = person => {
  person.canRead = true;
  person.canWrite = true;
  return person;
};

console.log(selfEducate(frederick));
console.log(frederick);

// {name: "Frederick Douglass", canRead: true, canWrite: true}
// {name: "Frederick Douglass", canRead: true, canWrite: true}
```



纯函数易于测试

纯函数可以轻而易举地测试。纯函数不改变环境（外部世界），因此测试前后不需要复杂的设置。纯函数所需的一切都通过参数获取。测试纯函数时，你可以自主控制参数，预估执行结果。前例中的 selfEducate 函数有副作用，不是纯函数。调用该函数会更改传给它的对象。倘若能把发给该函数的参数视为不可变数据，那它就是纯函数。

下面我们为这个函数添加一个参数：

```
const frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite: false
};

const selfEducate = person => ({
  ...person,
  canRead: true,
  canWrite: true
});

console.log(selfEducate(frederick));
console.log(frederick);
```

```
// {name: "Frederick Douglass", canRead: true, canWrite: true}
// {name: "Frederick Douglass", canRead: false, canWrite: false}
```

终于，这一版 selfEducate 变成了纯函数。该函数基于传入的值（即 person）做计算，返回一个新 person 对象，而不更改传入的参数，因此没有副作用。

来看一个更改 DOM 的非纯函数：

```
function Header(text) {
  let h1 = document.createElement("h1");
  h1.innerText = text;
  document.body.appendChild(h1);
}

Header("Header() caused side effects");
```

Header 函数使用指定的文本创建一个标题元素，然后把它添加到 DOM 中。这个函数不是纯函数，因为它没有返回一个函数或一个值，而且有副作用，即更改了 DOM。

在 React 中，UI 使用纯函数表达。下述示例中的 Header 函数是纯函数，其作用与前例一样，即创建一个 h1 元素。然而，这个函数自身没有副作用，因为没有更改 DOM。该函数虽然创建了 h1 元素，但是却把更改 DOM 的操作交给了应用的其他组件。

```
const Header = props => <h1>{props.title}</h1>;
```

纯函数也是函数式编程的核心概念。使用纯函数可以让你的生活轻松不少，因为纯函数不影响应用的状态。建议遵守下面三条规则编写函数：

1. 函数应该至少接受一个参数。
2. 函数应该返回一个值或另一个函数。
3. 函数不应该更改任何参数。

3.3.3 数据转换

如果数据都是不可变的，信息在一个应用中如何流动呢？在函数式编程中，数据从一种格式变成另一种格式。我们要做的是使用函数产出转换后的副本。这样的函数减少了代码的命令式属性，从而能降低复杂度。

你不需要任何框架就可以根据一个数据集产出另一个数据集，JavaScript 在语言层面上就内置了完成这项任务所需的工具。若想精通 JavaScript 函数式编程，必须掌握两个核心函数，即 `Array.map` 和 `Array.reduce`。

本节介绍如何使用这两个函数及其他核心函数把数据从一种类型转换成另一种类型。

以下述高中校名数组为例：

```
const schools = ["Yorktown", "Washington & Liberty", "Wakefield"];
```

我们可以使用 `Array.join` 函数得到一个以逗号分隔的清单或其他格式的字符串：

```
console.log(schools.join(", "));

// "Yorktown, Washington & Liberty, Wakefield"
```

`Array.join` 是 JavaScript 内置的数组方法，作用是从数组中提取出以指定符号分隔的字符串。原数组完好无损，`join` 只是为数组提供了另一种表示形式。至于字符串是如何得到的，程序员就无须关心了。

如果想定义一个函数，创建一个新数组，存储名称以“W”开头的学校，可以使用 `Array.filter` 方法：

```
const wSchools = schools.filter(school => school[0] === "W");

console.log(wSchools);
// ["Washington & Liberty", "Wakefield"]
```

Array.filter 是 JavaScript 内置的函数，根据源数组产出一个新数组。该函数只接受一个参数，这个参数是一个断言（predicate），即一个始终返回布尔值 true 或 false 的函数。Array.filter 在数组的每个元素上调用断言，元素作为参数传给断言，返回值决定是否把元素添加到新建的数组中。这里，我们使用 Array.filter 检查每个学校的名称是否以“W”字母开头。

若想从数组中删除元素，不建议使用 Array.pop 或 Array.splice，应该使用 Array.filter，因为 Array.filter 执行的是不可变操作。在下面的示例中，cutSchool 函数返回筛除特定校名后得到的新数组。

```
const cutSchool = (cut, list) => list.filter(school => school !== cut);

console.log(cutSchool("Washington & Liberty", schools).join(", "));

// "Yorktown, Wakefield"

console.log(schools.join("\n"));

// Yorktown
// Washington & Liberty
// Wakefield
```

这里，我们调用 cutSchool 函数返回一个不包含“Washington & Liberty”的新数组。然后，在新数组上调用 join 函数，创建由余下两个校名组成的字符串。cutSchool 是纯函数，它接受的参数为一组校名和想从中删除的校名，返回值为一个不含那个校名的新数组。

对函数式编程十分重要的另一个数组函数是 Array.map。这个函数接受的参数不是断言，而是一个函数。这个函数在数组的每个元素上调用一次，不管返回什么都添加到新数组中。

```
const highSchools = schools.map(school => `${school} High School`);

console.log(highSchools.join("\n"));
```

```
// Yorktown High School
// Washington & Liberty High School
// Wakefield High School

console.log(schools.join("\n"));

// Yorktown
// Washington & Liberty
// Wakefield
```

这里，我们使用 map 函数在每个校名后面添加“High School”。schools 数组依然完好无损。

在前面的示例中，我们从一个字符串数组中产出一个字符串数组。其实，map 函数可以产出任何 JavaScript 类型的数组，包括对象数组、值数组、嵌套数组和函数数组。下面的示例使用 map 函数返回每个学校的对象。

```
const highSchools = schools.map(school => ({ name: school }));

console.log(highSchools);

// [
// { name: "Yorktown" },
// { name: "Washington & Liberty" },
// { name: "Wakefield" }
// ]
```

我们从一个由字符串组成的数组产出了一个由对象组成的数组。

如果你想创建一个纯函数，更改对象数组中的一个对象，也可以使用 map。在下面的示例中，我们把校名“Stratford”改成了“HB Woodlawn”，但没有更改 schools 数组。

```
let schools = [
  { name: "Yorktown" },
  { name: "Stratford" },
  { name: "Washington & Liberty" },
```

```
  { name: "Wakefield" }
];

let updatedSchools = editName("Stratford", "HB Woodlawn", schools);

console.log(updatedSchools[1]); // { name: "HB Woodlawn" }
console.log(schools[1]); // { name: "Stratford" }
```

schools 是一个对象数组。updatedSchools 变量调用 editName 函数，传入想更新的校名、新校名和 schools 数组。相关改动是在新数组上操作的，原数组原封未动。

```
const editName = (oldName, name, arr) =>
  arr.map(item => {
    if (item.name === oldName) {
      return {
        ...item,
        name
      };
    } else {
      return item;
    }
  });
```

在 editName 函数中，map 函数用于根据原数组创建一个新对象数组。editName 函数在一行之内完全可以写完。下述示例使用简写的 if/else 语句编写这个函数：

```
const editName = (oldName, name, arr) =>
  arr.map(item => (item.name === oldName ? { ...item, name } : item));
```

如果想把数组转换成对象，可以结合 Array.map 和 Object.keys。Object.keys 方法从对象中获取所有键，返回由键构成的数组。

假如我们想把下面的 schools 对象转换成数组：

```
const schools = {
  Yorktown: 10,
```

```
  "Washington & Liberty": 2,
  Wakefield: 5
};

const schoolArray = Object.keys(schools).map(key => ({
  name: key,
  wins: schools[key]
}));

console.log(schoolArray);

// [
// {
// name: "Yorktown",
// wins: 10
// },
// {
// name: "Washington & Liberty",
// wins: 2
// },
// {
// name: "Wakefield",
// wins: 5
// }
// ]
```

在这个示例中，Object.keys 返回一个由校名构成的数组，我们可以在这个数组上调用 map，产出一个长度相同的新数组。新对象的 name 字段使用键名设置，wins 字段设为对应的值。

目前，我们学习了如何使用 Array.map 和 Array.filter 转换数组，还学习了如何结合 Object.keys 和 Array.map 把数组变成对象。我们的函数式装备库中需要的最后一个工具是把数组转换成原始类型和其他对象的能力。

reduce 和 reduceRight 函数可用于把数组转换成任何值，包括数字、字符串、布尔值、对象，甚至是函数。

假如我们想找出一个数字数组中最大的那个数，为此要把数组转换成数字，这时要使用 reduce：

```
const ages = [21, 18, 42, 40, 64, 63, 34];

const maxAge = ages.reduce((max, age) => {
  console.log(`${age} > ${max} = ${age > max}`);
  if (age > max) {
    return age;
  } else {
    return max;
  }
}, 0);

console.log("maxAge", maxAge);

// 21 > 0 = true
// 18 > 21 = false
// 42 > 21 = true
// 40 > 42 = false
// 64 > 42 = true
// 63 > 64 = false
// 34 > 64 = false
// maxAge 64
```

ages 数组被归约为一个值了，即最大年龄 64。reduce 函数接受两个参数：一个回调函数和一个初始值。在这个示例中，初始值是 0，即先把最大值设为 0。回调函数在数组中的每个元素上调用一次。首次调用回调函数时，age 等于 21，即数组中的第一个值，max 等于 0，即设置的初始值。回调函数返回这两个数中较大的那一个，即 21，下次迭代将把这个值赋给 max。每次迭代比较各个 age 值与 max 值，返回二者中较大的那个。最后，比较数组中最后一个数和前一次调用回调函数返回的数。

如果把前例中的 console.log 语句去掉，再换用简写的 if/else 语句，使用下述句法就可以计算任何数字数组中的最大值。

```
const max = ages.reduce((max, value) => (value > max ? value : max), 0);
```



Array.reduceRight

Array.reduceRight 的作用与 Array.reduce 相同，只不过不从数组开头开始归约，而是从数组末尾开始。

有时，我们需要把数组转换成对象。下述示例使用 reduce 把一个颜色数组转换成一个散列（hash）。

```
const colors = [
  {
    id: "xekare",
    title: "rad red",
    rating: 3
  },
  {
    id: "jbwsof",
    title: "big blue",
    rating: 2
  },
  {
    id: "prigbj",
    title: "grizzly grey",
    rating: 5
  },
  {
    id: "ryhbhsl",
    title: "banana",
    rating: 1
  }
];

const hashColors = colors.reduce((hash, { id, title, rating }) => {
  hash[id] = { title, rating };
  return hash;
}, {});

console.log(hashColors);
```

```
// {
// "xekare": {
// title:"rad red",
// rating:3
// },
// "jbwsof": {
// title:"big blue",
// rating:2
// },
// "prigbj": {
// title:"grizzly grey",
// rating:5
// },
// "ryhbhsl": {
// title:"banana",
// rating:1
// }
// }
```

在这个示例中，传给 reduce 函数的第二个参数是一个空对象。这是创建散列的初始值。每次迭代，回调函数使用方括号句法为散列添加一个键，把键的值设为数组中 id 字段的值。我们可以像这样使用 Array.reduce，把一个数组归约为一个值（本例是一个对象）。

使用 reduce 甚至可以把数组转换成完全不同的数组。假如我们想把一个包含重复值的数组归约为一个值都唯一的数组，那就可以使用 reduce 方法完成这项任务。

```
const colors = ["red", "red", "green", "blue", "green"];

const uniqueColors = colors.reduce(
  (unique, color) =>
    unique.indexOf(color) !== -1 ? unique : [...unique, color],
  []
);

console.log(uniqueColors);

// ["red", "green", "blue"]
```

这个示例把 colors 数组归约为一个没有重复值的数组。传给 reduce 函数的第二个参数是一个空数组，作为 unique 的初始值。如果 unique 数组中没有指定的颜色，才把那个颜色添加到数组中。否则，跳过那个颜色，返回当前的 unique 数组。[译注 1]

map 和 reduce 是函数式程序员不可缺少的重要武器，JavaScript 也不例外。如果你想成为一名熟练的 JavaScript 工程师，那就必须掌握这两个函数。由一个数据集创建另一个数据集是一项必备技能，任何编程范式都能用到。

3.3.4 高阶函数

高阶函数对函数式编程也是十分重要的。前面已经提及高阶函数，本章甚至还用过几次。高阶函数指用于处理其他函数的函数，其参数可以是函数，也可以返回函数，或者二者兼而有之。

第一类高阶函数是接受其他函数为参数的函数。Array.map、Array.filter 和 Array.reduce 都接受函数作为参数，因此它们是高阶函数。

下面来看如何实现高阶函数。在下面的示例中，我们定义了一个 invokeIf 函数，它测试一个条件，在返回真值时调用一个回调函数，在返回假值时调用另一个回调函数。

```
const invokeIf = (condition, fnTrue, fnFalse) =>
  condition ? fnTrue() : fnFalse();

const showWelcome = () => console.log("Welcome!!!");

const showUnauthorized = () => console.log("Unauthorized!!!");

invokeIf(true, showWelcome, showUnauthorized); // "Welcome!!!"
invokeIf(false, showWelcome, showUnauthorized); // "Unauthorized!!!"
```

invokeIf 接受两个函数为参数，一个针对真值，另一个针对假值。为了演示该

译注 1：根据上下文，前一段中原文所说的“distinct”数组应该是“unique”，翻译时已更正。

函数的用法，我们传入了 showWelcome 和 showUnauthorized。条件为真时，调用 showWelcome，为假时则调用 showUnauthorized。

返回一个函数的高阶函数有助于我们处理 JavaScript 复杂的异步操作，创建我们在必要时需要使用或重用的函数。

柯里化（currying）就是需要用到高阶函数的一种函数式技术。

下面是柯里化的一例。userLogs 函数保存一些信息（username），返回一个函数，以便在其他信息（message）可用时使用或重用。在这个示例中，所有日志消息前面都会加上对应的用户名。注意，这里用到了第 2 章返回一个 promise 的 getFakeMembers 函数。

```
const userLogs = userName => message =>
  console.log(`${userName} -> ${message}`);

const log = userLogs("grandpa23");

log("attempted to load 20 fake members");
getFakeMembers(20).then(
  members => log(`successfully loaded ${members.length} members`),
  error => log("encountered an error loading members")
);

// grandpa23 -> attempted to load 20 fake members
// grandpa23 -> successfully loaded 20 members

// grandpa23 -> attempted to load 20 fake members
// grandpa23 -> encountered an error loading members
```

userLogs 是高阶函数。log 函数由 userLogs 得来，每次调用 log 函数都会在消息前面加上“grandpa23”。

3.3.5 递归

递归技术指创建重新调用自身的函数。如果要解决的问题涉及循环，往往就可以转而使用递归函数。假如有一项任务是从 10 开始倒计时。我们可以通过一

个 for 循环解决这个问题，此外也可以使用一个递归函数。在下面的示例中，countdown 是递归函数。

```
const countdown = (value, fn) => {
  fn(value);
  return value > 0 ? countdown(value - 1, fn) : value;
};

countdown(10, value => console.log(value));

// 10
// 9
// 8
// 7
// 6
// 5
// 4
// 3
// 2
// 1
// 0
```

countdown 函数的参数为一个数字和一个函数。在这个示例中，调用该函数时传入参数是值 10 和一个回调函数。调用 countdown 函数时即调用回调函数，把当前值输出到控制台。接下来，countdown 检查值是否大于 0。如果大于 0，countdown 调用自身，传入递减后的值。最终，值将变为 0，此时 countdown 直接返回 0，整个调用栈结束。

递归模式特别适合在异步操作中使用。函数在做好准备时调用自身，例如在数据可用时，或者当计时器结束时。

我们可以修改一下 countdown 函数，为倒计时加入延迟。修改后的版本可用于创建倒计时钟表。

```
const countdown = (value, fn, delay = 1000) => {
  fn(value);
  return value > 0
```

```
    ? setTimeout(() => countdown(value - 1, fn, delay), delay)
    : value;
};

const log = value => console.log(value);
countdown(10, log);
```

在这个示例中，我们创建了一个10秒钟的倒计时，只在一开始调用了countdown函数一次，传入数字10和一个记录倒数读秒的函数。countdown没有立即调用自身，而是像真正的钟表一样，等待一秒钟再调用。

递归技术适合在搜索数据结构中使用。你可以递归迭代子文件夹，直到找出仅包含文件的文件夹为止。也可以递归迭代HTML DOM，直至找到一个没有子节点的元素为止。下面的示例递归迭代一个对象，深入对象内部获取一个嵌套的值。

```
const dan = {
  type: "person",
  data: {
    gender: "male",
    info: {
      id: 22,
      fullname: {
        first: "Dan",
        last: "Deacon"
      }
    }
  }
};

deepPick("type", dan); // "person"
deepPick("data.info.fullname.first", dan); // "Dan"
```

我们可以使用deepPick获取存储在第一个对象中的Dan的类型，也可以深入内部嵌套的对象，找出Dan的名字。使用点号记法的字符串可以指定要寻找的值在对象中的嵌套层级。

```
const deepPick = (fields, object = {}) => {
  const [first, ...remaining] = fields.split(".");
```

```
  return remaining.length
    ? deepPick(remaining.join("."), object[first])
    : object[first];
};
```

deepPick 函数要么返回一个值，要么调用自身，不过终将返回一个值。首先，这个函数把点号记法的字段字符串分解为一个数组，通过数组析构把第一个值与其余的值分开。如果还有余值，deepPick 调用自身，传入稍微不同的数据，继续深入下一层。

这个函数不断调用自身，直到字段字符串中没有点号，即后面没有字段为止。通过下面的示例可以看出，在 deepPick 迭代的过程中 first、remaining 和 object[first] 的值是如何变化的。

```
deepPick("data.info.fullname.first", dan); // "Dan"

// First Iteration
// first = "data"
// remaining.join(".") = "info.fullname.first"
// object[first] = { gender: "male", {info} }

// Second Iteration
// first = "info"
// remaining.join(".") = "fullname.first"
// object[first] = {id: 22, {fullname}}

// Third Iteration
// first = "fullname"
// remaining.join("." = "first"
// object[first] = {first: "Dan", last: "Deacon" }

// Finally...
// first = "first"
// remaining.length = 0
// object[first] = "Deacon"
```

递归是一项强大的函数式技术，实现起来是一种享受。

3.3.6 合成

函数式程序是把逻辑拆分成一系列关注特定任务的小型纯函数。最终，你要把这些小型函数整合到一起。具体来说，你可能要综合运用，按一定顺序或并行调用，或者合成一些大型函数，直到构建出一个应用为止。

合成的方式、模式和技术有很多种，你可能熟悉的一种是串接。在 JavaScript 中，函数可以使用点号记法串接在一起，依次处理前一个函数的返回值。

我们知道，字符串有 replace 方法。replace 方法返回一个模板字符串，而模板字符串也有 replace 方法。因此，可以使用点号记法把 replace 方法串接在一起处理字符串。

```
const template = "hh:mm:ss tt";
const clockTime = template
  .replace("hh", "03")
  .replace("mm", "33")
  .replace("ss", "33")
  .replace("tt", "PM");

console.log(clockTime);

// "03:33:33 PM"
```

在这个示例中，template 是一个字符串。在 template 字符串后面串接一系列 replace 方法，可以把字符串中的时分秒替换成新的时间值。template 自身完好无损，可以继续用于创建要显示的其他时间。

下面的 both 函数充当了在两个函数之间传递数据的管道。civilianHours 函数的输出作为 appendAMPM 函数的输入，把两个函数的功能合二为一，修改日期。

```
const both = date => appendAMPM(civilianHours(date));
```

然而，这样的句法难以理解，也不易维护和扩充。试想，如果需要通过 20 个函数发送一个值呢？

一种更为优雅的方法是创建一个高阶函数，把多个函数合成为一个大型函数。

```
const both = compose(
  civilianHours,
  appendAMPM
);

both(new Date());
```

这种方法看起来好多了。这样易于扩充，随时都可以添加更多函数。另外，这种方法也方便修改组合函数的顺序。

compose 函数是一个高阶函数，接受的参数是函数，只返回一个值。

```
const compose = (...fns) => arg =>
  fns.reduce((composed, f) => f(composed), arg);
```

compose 接受的参数是函数，返回一个函数。在上述实现中，我们使用展开运算符把通过参数传入的函数转换成一个名为 fns 的数组。返回的函数接受一个参数，即 arg。调用这个函数时，首先填充 fns 数组的是想通过该函数发送的参数。这个参数作为 compose 的初始值，随后各值是每次迭代 reduce 的回调返回的值。注意，回调函数接受两个参数：composed 和一个函数 f。调用每个函数时传入的参数为 composed，即前一个函数的输出结果。最终，将调用最后一个函数，返回最后的结果。

这是一个简单的示例，目的是通过 compose 函数演示合成技术的用法。如果需要处理多个参数，或者处理的参数不是函数，那情况就复杂了。

3.3.7 综合应用

现在，我们已经学习了函数式编程的核心概念。下面我们将这些概念用于构建一个小型 JavaScript 应用。

我们的目标是构建一个时钟，以民用时间格式显示时分秒和上下午。每个字段

都由两个数字组成，如果时间值只有一个数字，则在前面添上零。这个时钟还要走秒，每隔一秒变一次。

首先，来看使用命令式风格写出的时钟。

```
// 每秒记录一次时钟时间
setInterval(logClockTime, 1000);

function logClockTime() {
  // 获取民用格式的时间字符串
  let time = getClockTime();

  // 清空控制台，输出时间
  console.clear();
  console.log(time);
}

function getClockTime() {
  // 获取当前时间
  let date = new Date();
  let time = "";

  // 序列化时钟时间
  let time = {
    hours: date.getHours(),
    minutes: date.getMinutes(),
    seconds: date.getSeconds(),
    ampm: "AM"
  };

  // 转换成民用格式
  if (time.hours == 12) {
    time.ampm = "PM";
  } else if (time.hours > 12) {
    time.ampm = "PM";
    time.hours -= 12;
  }

  // 为小时数添加一个 0，变成两位数
  if (time.hours < 10) {
```

```
    time.hours = "0" + time.hours;
  }

  // 为分钟数添加一个 0，变成两位数
  if (time.minutes < 10) {
    time.minutes = "0" + time.minutes;
  }

  // 为秒数添加一个 0，变成两位数
  if (time.seconds < 10) {
    time.seconds = "0" + time.seconds;
  }

  // 把时钟时间格式化为“hh:mm:ss tt”形式的字符串
  return time.hours + ":" + time.minutes + ":" + time.seconds + " " + time.ampm;
}
```

这样做是可以，我们可以借助注释理解代码的作用。然而，这里用到的函数又大又复杂，每个函数做的事情太多了，没有注释的帮助很难理解，而且不易维护。下面再看使用函数式风格构建出来的应用是多么灵便。

我们的目标是把应用的逻辑切分成一系列小的构成部分，即函数。每个函数专注于一项任务，然后再组合成大型函数，用于创建时钟。

首先，我们要创建几个函数，用于获取值和管理控制台。我们需要一个返回一秒钟的函数、一个返回当前时间的函数，以及把消息输出到控制台和清理控制台的函数。在函数式程序中，应该尽量使用函数，而非值。需要时，我们可以调用函数来获取值。

```
const oneSecond = () => 1000;
const getCurrentTime = () => new Date();
const clear = () => console.clear();
const log = message => console.log(message);
```

接下来，需要一些转换数据的函数。我们将使用下面三个函数更改 Date 对象，把它变成创建时钟所需的对象。

`serializeClockTime`

把传入的日期对象处理成包含时分秒的时钟时间。

`civilianHours`

把传入的时钟时间对象中的小时数转换成民用时间格式。例如，1300 变成 1:00。

`appendAMPM`

在传入的时钟时间对象上添加上下午标识（AM 或 PM）。

```
const serializeClockTime = date => ({
  hours: date.getHours(),
  minutes: date.getMinutes(),
  seconds: date.getSeconds()
});

const civilianHours = clockTime => ({
  ...clockTime,
  hours: clockTime.hours > 12 ? clockTime.hours - 12 : clockTime.hours
});

const appendAMPM = clockTime => ({
  ...clockTime,
  ampm: clockTime.hours >= 12 ? "PM" : "AM"
});
```

这三个函数用于转换数据，把参数视作不可变的对象，不修改原数据。

接下来，需要定义几个高阶函数。

`display`

接受一个函数，返回一个函数，把时间发给参数指定的函数。在这个示例中，目标函数是 `console.log`。

`formatClock`

接受一个模板字符串，返回根据指定条件格式化的时钟时间。在这个示例中，模板是“hh:mm:ss tt”。`formatClock` 将把模板中的时分秒和上下午替换成具体的值。

prependZero

接受对象中的一个键为参数，为该键名下存储的值添加前导零。接受指定字段的键，如果键对应的值小于 10，就在值前添加一个零。

```
const display = target => time => target(time);

const formatClock = format => time =>
  format
    .replace("hh", time.hours)
    .replace("mm", time.minutes)
    .replace("ss", time.seconds)
    .replace("tt", time.ampm);

const prependZero = key => clockTime => ({
  ...clockTime,
  key: clockTime[key] < 10 ? "0" + clockTime[key] : clockTime[key]
});
```

我们将调用这几个高阶函数创建格式化，每一次走秒后的时钟时间所需重复使用的函数。formatClock 和 prependZero 只在开头调用一次，设置所需的模板和键。二者返回的函数每一秒调用一次，把时间格式化成显示所需的格式。

现在，构建时钟所需的函数都定义好了，接下来该合成了。我们将使用前一节定义的 compose 函数处理合成。

convertToCivilianTime

一个独立函数，接受时钟时间为参数，把它转换成民用时间格式。

doubleDigits

一个独立函数，接受民用时钟时间为参数，添加所需的前导零，确保时分秒都显示为两位数。

startTicking

设置一个时间间隔，每隔一秒调用一次回调，启动时钟。回调是使用我们自己定义的函数合成的。每隔一秒清理控制台、获取当前时间、转换格式、变成民用格式、格式化，最后显示。

```
const convertToCivilianTime = clockTime =>
  compose(
    appendAMPM,
    civilianHours
  )(clockTime);

const doubleDigits = civilianTime =>
  compose(
    prependZero("hours"),
    prependZero("minutes"),
    prependZero("seconds")
  )(civilianTime);

const startTicking = () =>
  setInterval(
    compose(
      clear,
      getCurrentTime,
      serializeClockTime,
      convertToCivilianTime,
      doubleDigits,
      formatClock("hh:mm:ss tt"),
      display(log)
    ),
    oneSecond()
  );

startTicking();
```

这个声明式版本实现的效果与命令式版本一样。然而，这一版却有几个优点。首先，这些函数都易于测试和重用，可用于制作其他时钟或数显功能。其次，这个程序易于扩充。这里没有副作用，函数外部没有全局变量。bug 不可避免，至少容易找到。

本章介绍了函数式编程的基本概念。本书后文在讨论 React 的最佳实践时还会说明很多 React 组件是如何利用函数式编程技术的。下一章正式开始探讨 React，深入了解一些起指导作用的开发理念。

第 4 章

React 运行机制

目前，我们学习了最新的句法，还探讨了指导 React 开发的函数式编程模式。这些都是为下一步学习做准备的，即本章要讲的 React 的运行机制。我们要开始编写真正的 React 代码了。

使用 React 构建应用几乎离不开 JSX。这是一种基于标签的 JavaScript 句法，看起来很像 HTML。JSX 句法在下一章详细说明，本书后续章节会不断用到。然而，若想完全理解 React，必须要弄懂 React 最基本的构成单元，即 React 元素。随后，我们将探讨 React 元素。接着，讨论 React 组件，学习如何创建组成其他组件和元素的自定义组件。

4.1 页面设置

为了在浏览器中使用 React，我们要引入两个库：React 和 ReactDOM。前者用于创建视图，后者则具体负责在浏览器中渲染 UI。这两个库都可以通过 unpkg CDN 引用（链接见下述代码片段）。我们来创建一个 HTML 文档：

```
<!DOCTYPE html>
<html>
  <head>
```

```
    <meta charset="utf-8" />
    <title>React Samples</title>
  </head>
  <body>
    <!-- 目标容器 -->
    <div id="root"></div>

    <!-- React 和 ReactDOM 库（开发版本）-->
    <script
  src="https://unpkg.com/react@16/umd/react.development.js">
  </script>
    <script
  src="https://unpkg.com/react-dom@16/umd/react-dom.development.js">
  </script>

    <script>
      // 纯 React 和 JavaScript 代码
    </script>
  </body>
</html>
```

这是在浏览器中使用 React 需要做的最少工作。你可以把 JavaScript 代码放在单独的文件中，不过在页面中加载时必须放到 React 后面。为了在浏览器控制台中看到所有错误消息和警告，我们使用的是 React 的开发版本。你也可以使用 *react.production.min.js* 和 *react-dom.production.min.js*，换成供生产环境使用的简化版本，只是看不到警告消息了。

4.2 React 元素

简单来说，HTML 是一系列指令，让浏览器构建 DOM。浏览器加载 HTML 并渲染用户界面时，构成 HTML 文档的元素变成 DOM 元素。

比如说你想为一个菜谱构建 HTML 层次结构。这项任务可能的一种实现方案如下所示。

```
<section id="baked-salmon">
  <h1>Baked Salmon</h1>
```

```
    <ul class="ingredients">
      <li>2 lb salmon</li>
      <li>5 sprigs fresh rosemary</li>
      <li>2 tablespoons olive oil</li>
      <li>2 small lemons</li>
      <li>1 teaspoon kosher salt</li>
      <li>4 cloves of chopped garlic</li>
    </ul>
    <section class="instructions">
      <h2>Cooking Instructions</h2>
      <p>Preheat the oven to 375 degrees.</p>
      <p>Lightly coat aluminum foil with oil.</p>
      <p>Place salmon on foil</p>
      <p>Cover with rosemary, sliced lemons, chopped garlic.</p>
      <p>Bake for 15-20 minutes until cooked through.</p>
      <p>Remove from oven.</p>
    </section>
  </section>
```

在HTML中，元素之间呈现一种层次结构，就像家谱一样。我们可以说根元素（在本例中是一个区块）有三个子代：一个标题、一个配料无序列表以及一个操作说明区块。

过去，网站由相互独立的HTML页面组成。用户访问这些页面时，浏览器请求并加载各HTML文档。AJAX（Asynchronous JavaScript and XML）的发明引入了单页应用（single-page application，SPA，*https://oreil.ly/z6Saj*）的概念。由于浏览器可以通过AJAX请求并加载少量数据之后，整个Web应用便可以精简成一个页面，依靠JavaScript更新用户界面。

在单页应用中，浏览器只在开始时加载一个HTML文档。在用户浏览网站的过程中，始终待在同一个页面。用户不断与应用产生交互，JavaScript则不断销毁和创建新的用户界面。这给人的感觉像是从一个页面跳转到另一个页面，但其实你仍然在相同的HTML页面上，背后繁重的工作都交给JavaScript了。

DOM API（*https://mzl.la/2m1oQDJ*）是一系列对象，供JavaScript与浏览

器交互，修改 DOM。如果你曾用过 `document.createElement` 或 `document.appendChild`，那你就用过 DOM API。

React 是代我们更新浏览器 DOM 的一个库。我们不再需要关心构建高性能单页应用相关的复杂问题了，一切都交给 React。有了 React，我们不再直接与 DOM API 交互，而是指明想让 React 构建什么，React 收到指令后会帮我们渲染和协调元素。

浏览器 DOM 由 DOM 元素组成，类似地，React DOM 由 React 元素组成。DOM 元素和 React 元素看着类似，其实区别很大。React 元素是对真正的 DOM 元素的描述。换句话说，React 元素是如何创建浏览器 DOM 的指令。

我们可以使用 `React.createElement` 创建一个表示 `h1` 的 React 元素：

```
React.createElement("h1", { id: "recipe-0" }, "Baked Salmon");
```

第一个参数指定想创建的元素的类型。这里，我们想创建的是一个 `h1` 元素。第二个参数设定元素的属性。这个 `h1` 有个 `id` 属性，值为 `recipe-0`。第三个参数指定元素的子节点，即插入起始和结束标签之间的节点（这里只插入了一些文本）。

渲染时，React 将把这个元素转换成真正的 DOM 元素：

```
<h1 id="recipe-0">Baked Salmon</h1>
```

属性以类似的方式赋予新创建的 DOM 元素，添加到标签的属性中，而子节点则作为元素内的文本。React 组件其实就是 JavaScript 字面量，告知 React 如何构建 DOM 元素。

在控制台中输出这个元素，你会看到如下内容：

```
{
  $$typeof: Symbol(React.element),
  "type": "h1",
```

```
  "key": null,
  "ref": null,
  "props": {id: "recipe-0", children: "Baked Salmon"},
  "_owner": null,
  "_store": {}
}
```

这是一个 React 元素的结构，其中一些字段是供 React 使用的，包括 `_owner`、`_store` 和 `$$typeof`。`key` 和 `ref` 字段对 React 元素很重要，稍后再讲。现在先来看 `type` 和 `props` 字段。

React 元素的 `type` 属性告知 React 要创建的是什么类型的 HTML 或 SVG 元素。`props` 属性表示构建一个 DOM 元素所需的数据和子元素。`children` 属性则表示嵌套显示在元素内的文本。



创建元素

这里给出的是 `React.createElement` 返回的对象。我们其实不会自己动手创建 React 元素，而是使用 `React.createElement` 函数。

4.3 ReactDOM

创建 React 元素后，我们希望在浏览器中看到它的效果。ReactDOM 就提供了在浏览器中渲染 React 元素所需的工具。渲染所需的 `render` 方法在 ReactDOM 中。

React 元素，包括它的子元素，使用 `ReactDOM.render` 在 DOM 中渲染。我们想渲染的元素通过第一个参数传入，第二参数是目标节点，即指明在哪里渲染元素。

```
const dish = React.createElement("h1", null, "Baked Salmon");

ReactDOM.render(dish, document.getElementById("root"));
```

这段代码在 DOM 中渲染标题元素，把一个 h1 元素添加到 id 为 root 的 div 元素（已在 HTML 中定义）中。这个 div 元素在 body 标签内构建。

```
<body>
  <div id="root">
    <h1>Baked Salmon</h1>
  </div>
</body>
```

在 DOM 中与渲染元素有关的一切功能都在 ReactDOM 包中。在 React 16 之前的版本中，只能在 DOM 中渲染一个元素。而现在，还可以渲染数组。在 ReactConf 2017 上宣布这个功能时，所有人都兴奋地鼓掌和尖叫。这个功能十分重要，具体用法如下。

```
const dish = React.createElement("h1", null, "Baked Salmon");
const dessert = React.createElement("h2", null, "Coconut Cream Pie");

ReactDOM.render([dish, dessert], document.getElementById("root"));
```

上述代码在 root 容器中把两个元素渲染为同辈元素。但愿你也鼓掌和尖叫了。

下一节说明如何使用 props.children。

子元素

React 使用 props.children 渲染子元素。前一节把一个文本元素渲染为 h1 元素的子元素，因此 props.children 的值是 Baked Salmon。此外，也可以把其他 React 元素渲染为子元素，创建一个元素树。之所以用“元素树”这个术语，是因为这种结构有一个根元素，从根元素上又发出很多枝杈。

以呈现配料的那个无序列表为例：

```
<ul>
  <li>2 lb salmon</li>
  <li>5 sprigs fresh rosemary</li>
  <li>2 tablespoons olive oil</li>
```

```
    <li>2 small lemons</li>
    <li>1 teaspoon kosher salt</li>
    <li>4 cloves of chopped garlic</li>
  </ul>
```

在这个示例中，根元素是无序列表，它有6个子元素。使用React.createElement表示这个ul及其子元素的方式如下：

```
React.createElement(
  "ul",
  null,
  React.createElement("li", null, "2 lb salmon"),
  React.createElement("li", null, "5 sprigs fresh rosemary"),
  React.createElement("li", null, "2 tablespoons olive oil"),
  React.createElement("li", null, "2 small lemons"),
  React.createElement("li", null, "1 teaspoon kosher salt"),
  React.createElement("li", null, "4 cloves of chopped garlic")
);
```

传给createElement函数的各个额外参数就是一个子元素。React将创建一个由子元素构成的数组，把props.children的值设为这个数组。

审查得到的React元素，你会看到每个列表项目都是一个React元素，出现在props.children名下的数组里。在控制台中输出这个元素：

```
const list = React.createElement(
  "ul",
  null,
  React.createElement("li", null, "2 lb salmon"),
  React.createElement("li", null, "5 sprigs fresh rosemary"),
  React.createElement("li", null, "2 tablespoons olive oil"),
  React.createElement("li", null, "2 small lemons"),
  React.createElement("li", null, "1 teaspoon kosher salt"),
  React.createElement("li", null, "4 cloves of chopped garlic")
);

console.log(list);
```

结果如下所示：

```
{
    "type": "ul",
    "props": {
    "children": [
    { "type": "li", "props": { "children": "2 lb salmon" } … },
    { "type": "li", "props": { "children": "5 sprigs fresh rosemary"} … },
    { "type": "li", "props": { "children": "2 tablespoons olive oil" } … },
    { "type": "li", "props": { "children": "2 small lemons"} … },
    { "type": "li", "props": { "children": "1 teaspoon kosher salt"} … },
    { "type": "li", "props": { "children": "4 cloves of chopped garlic"} … }
    ]
    …
    }
}
```

可以看到，每个列表项目都是一个子元素。本章前面给出了整个菜谱的HTML，根元素是一个section。若想在React中创建这个结构，要使用一系列createElement调用：

```
React.createElement(
  "section",
  { id: "baked-salmon" },
  React.createElement("h1", null, "Baked Salmon"),
  React.createElement(
    "ul",
    { className: "ingredients" },
    React.createElement("li", null, "2 lb salmon"),
    React.createElement("li", null, "5 sprigs fresh rosemary"),
    React.createElement("li", null, "2 tablespoons olive oil"),
    React.createElement("li", null, "2 small lemons"),
    React.createElement("li", null, "1 teaspoon kosher salt"),
    React.createElement("li", null, "4 cloves of chopped garlic")
  ),
  React.createElement(
    "section",
    { className: "instructions" },
    React.createElement("h2", null, "Cooking Instructions"),
    React.createElement("p", null, "Preheat the oven to 375 degrees."),
```

```
    React.createElement("p", null, "Lightly coat aluminum foil with oil."),
    React.createElement("p", null, "Place salmon on foil."),
    React.createElement(
      "p",
      null,
      "Cover with rosemary, sliced lemons, chopped garlic."
    ),
    React.createElement(
      "p",
      null,
      "Bake for 15-20 minutes until cooked through."
    ),
    React.createElement("p", null, "Remove from oven.")
  )
);
```



React 中的 className

有 class 属性的 HTML 元素，在 React 中使用的是 className，而不是 class。这是因为 class 是 JavaScript 的保留字，所以不得不使用 className 定义 HTML 元素的 class 属性。这个示例演示的是纯粹使用 React 的情况，浏览器最终运行的就是纯 React。React 应用是由一系列 React 元素构成的树状结构，一切都发端于一个根元素。React 使用 React 元素发出的指令在浏览器中构建 UI。

使用数据构建元素

React 的主要优势是能把数据与 UI 元素分开。由于 React 本身是 JavaScript，我们可以借助 JavaScript 逻辑构建 React 组件树。例如，可以把配料存储在一个数组中，然后映射那个数组，构建 React 组件。

回过头再看一下那个无序列表：

```
React.createElement(
  "ul",
  null,
```

```
    React.createElement("li", null, "2 lb salmon"),
    React.createElement("li", null, "5 sprigs fresh rosemary"),
    React.createElement("li", null, "2 tablespoons olive oil"),
    React.createElement("li", null, "2 small lemons"),
    React.createElement("li", null, "1 teaspoon kosher salt"),
    React.createElement("li", null, "4 cloves of chopped garlic")
  );
```

这个配料列表用到的数据可以轻易使用一个 JavaScript 数组表示：

```
const items = [
  "2 lb salmon",
  "5 sprigs fresh rosemary",
  "2 tablespoons olive oil",
  "2 small lemons",
  "1 teaspoon kosher salt",
  "4 cloves of chopped garlic"
];
```

我们希望使用这些数据生成全部列表项目，但自己不用动手一个一个编写。为此，我们可以映射这个数组，为各个配料创建列表项目：

```
React.createElement(
  "ul",
  { className: "ingredients" },
  items.map(ingredient => React.createElement("li", null, ingredient))
);
```

上述句法为数组中的每个配料创建一个 React 元素。各字符串作为文本显示在列表项目的子节点中。每个配料的名称显示为列表项目。

运行这段代码，你会看到控制台发出一个警告，如图 4-1 所示。

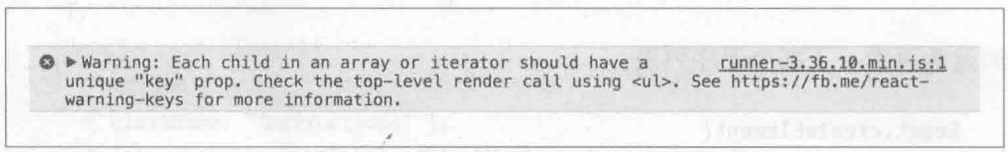

图 4-1：控制台警告

通过迭代数组构建一组子元素时，React 预期每个元素都有 key 属性。有了 key 属性，React 才能高效率地更新 DOM。为每个列表项目元素添加一个独一无二的 key 属性后，这个警告便会消失。我们可以使用每个配料的数组索引作为这个独一无二的值：

```
React.createElement(
  "ul",
  { className: "ingredients" },
  items.map((ingredient, i) =>
    React.createElement("li", { key: i }, ingredient)
  )
);
```

讲解 JSX 时将进一步探讨键，现在只需为每个列表项目添加独一无二的键，让控制台中的警告消失。

4.4 React 组件

不管用户界面的体量多大、内容是什么，或者是用什么技术创建的，都是由不同的部件构成的。按钮、列表、标题，各个部件组合在一起构成完整的用户界面。假如一个菜谱应用中有三份菜谱，虽然每一部分显示的数据不一样，但是创建每一个菜谱的部件是一样的（见图 4-2）。

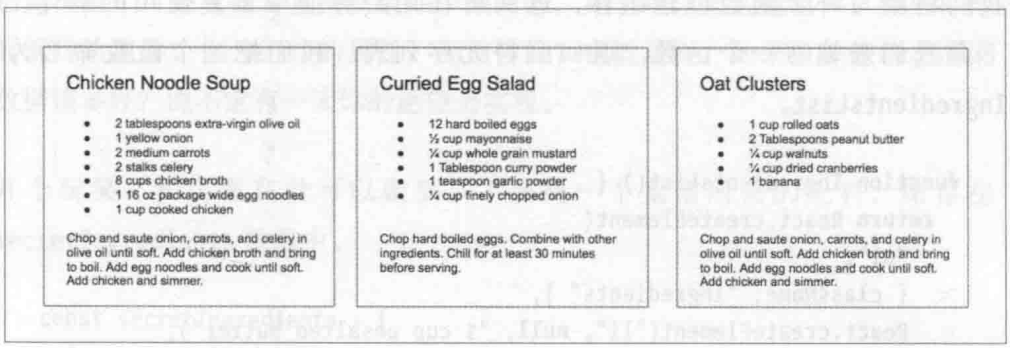

图 4-2：菜谱应用

在React中，我们把这样的部件叫作组件(component)。组件方便重用相同的结构，只需在结构中填充不同的数据即可。

使用 React 构建用户界面时，应该尽量考虑把元素分解为可重用的片段。例如，图 4-3 中的菜谱有标题、配料列表和操作说明。这些都是一个完整的菜谱或应用组件的构成部分。图中突出显示的部分都可以创建为组件，包括配料列表、操作说明等。

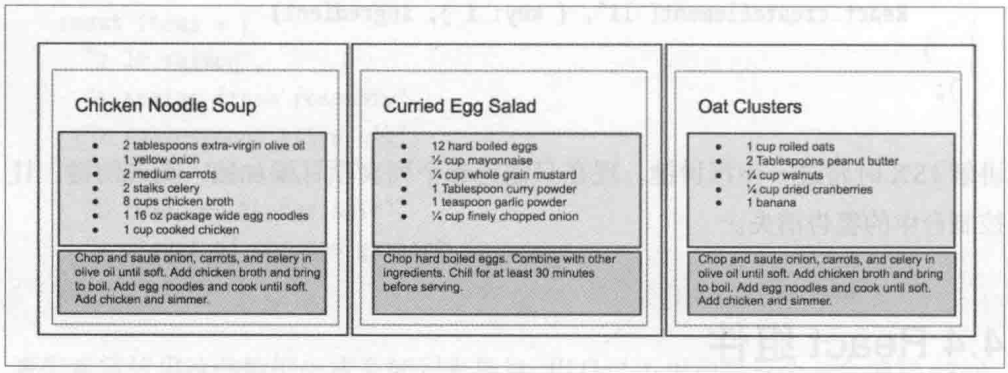

图 4-3：框出各个组件：App、IngredientsList、Instructions

你能看出这样做多么方便扩充吗？如果想显示一个菜谱，可以用这个组件。如果想显示 10000 个菜谱，只需创建 10000 个组件实例。

我们将编写一个函数创建组件，返回可在用户界面中重复使用的部件。下面我们来编写一个函数，返回配料无序列表。我们把这个函数命名为 IngredientsList。

```
function IngredientsList() {
  return React.createElement(
    "ul",
    { className: "ingredients" },
    React.createElement("li", null, "1 cup unsalted butter"),
    React.createElement("li", null, "1 cup crunchy peanut butter"),
    React.createElement("li", null, "1 cup brown sugar"),
    React.createElement("li", null, "1 cup white sugar"),
    React.createElement("li", null, "2 eggs"),
```

```
      React.createElement("li", null, "2.5 cups all purpose flour"),
      React.createElement("li", null, "1 teaspoon baking powder"),
      React.createElement("li", null, "0.5 teaspoon salt")
    );
  }
ReactDOM.render(
  React.createElement(IngredientsList, null, null),
  document.getElementById("root")
);
```

这个组件的名称为 IngredientsList。上述函数输出的元素如下所示：

```
<IngredientsList>
  <ul className="ingredients">
    <li>1 cup unsalted butter</li>
    <li>1 cup crunchy peanut butter</li>
    <li>1 cup brown sugar</li>
    <li>1 cup white sugar</li>
    <li>2 eggs</li>
    <li>2.5 cups all purpose flour</li>
    <li>1 teaspoon baking powder</li>
    <li>0.5 teaspoon salt</li>
  </ul>
</IngredientsList>
```

感觉不错，可是数据是直接在组件中编写的。如果可以先构建组件，然后再把数据传给组件，作为组件的属性，那该多好？也就是说，如果组件能动态渲染数据该多好？说不定有一天你的愿望会实现。

开个玩笑，其实现在就可以做到。下面是一个菜谱所需的配料，保存在 secretIngredients 数组中。

```
const secretIngredients = [
  "1 cup unsalted butter",
  "1 cup crunchy peanut butter",
  "1 cup brown sugar",
  "1 cup white sugar",
  "2 eggs",
```

```
    "2.5 cups all purpose flour",
    "1 teaspoon baking powder",
    "0.5 teaspoon salt"
];
```

接下来，我们要修改 IngredientsList 组件，映射 items 数组，为数组中的各个元素构建一个 li。

```
function IngredientsList() {
  return React.createElement(
    "ul",
    { className: "ingredients" },
    items.map((ingredient, i) =>
      React.createElement("li", { key: i }, ingredient)
    )
  );
}
```

然后，我们要通过 createElement 的第二个参数把 secretIngredients 传给 items 属性：

```
ReactDOM.render(
  React.createElement(IngredientsList, { items: secretIngredients }, null),
  document.getElementById("root")
);
```

现在，来看一下 DOM。数据属性 items 是一个数组，含有 8 种配料。我们使用一个循环创建各个 li 标签，因此可以把循环的索引设为唯一的键。

```
<IngredientsList items="[...]">
  <ul className="ingredients">
    <li key="0">1 cup unsalted butter</li>
    <li key="1">1 cup crunchy peanut butter</li>
    <li key="2">1 cup brown sugar</li>
    <li key="3">1 cup white sugar</li>
    <li key="4">2 eggs</li>
    <li key="5">2.5 cups all purpose flour</li>
    <li key="6">1 teaspoon baking powder</li>
```

```
      <li key="7">0.5 teaspoon salt</li>
    </ul>
  </IngredientsList>
```

像这样创建组件更为灵活。不管 items 数组中有一个元素还是一百个元素，这个组件都可以把各个元素渲染为列表项目。

我们可以调整一下，从 React 属性中引用 items 数组，不再映射全局的 items，而是从 props 对象上获取 items。我们要把 props 传给函数，然后再映射 props.items。

```
function IngredientsList(props) {
  return React.createElement(
    "ul",
    { className: "ingredients" },
    props.items.map((ingredient, i) =>
      React.createElement("li", { key: i }, ingredient)
    )
  );
}
```

我们还可以稍微清理一下代码，直接从 props 中析构 items：

```
function IngredientsList({ items }) {
  return React.createElement(
    "ul",
    { className: "ingredients" },
    items.map((ingredient, i) =>
      React.createElement("li", { key: i }, ingredient)
    )
  );
}
```

与 IngredientsList 的 UI 有关的一切现在都封装在一个组件中。我们所需的一切都在那个组件中。

React 组件历史回顾

在出现函数组件之前，创建组件有几种其他方式。本书不再着墨于这些方式，不过对 React 组件的历史还是要有一定的了解，毕竟在陈旧的代码基中仍要使用这些 API。下面我们来简单回顾一下 React API 的历史。

第一站：createClass

2013 年，React 刚开源时，创建组件使用 `createClass`。使用 `React.createClass` 创建组件的方法如下所示：

```
const IngredientsList = React.createClass({
  displayName: "IngredientsList",
  render() {
    return React.createElement(
      "ul",
      { className: "ingredients" },
      this.props.items.map((ingredient, i) =>
        React.createElement("li", { key: i }, ingredient)
      )
    );
  }
});
```

使用 `createClass` 创建的组件需要一个 `render()` 方法，指明返回及渲染什么 React 元素。组件背后的思想还是一样的，即描述一些可重用的 UI 供渲染。

在 React 15.5（2017 年 4 月）中，React 开始在使用 `createClass` 的地方发出警告。在 React 16（2017 年 9 月）中，`React.createClass` 正式被弃用，移到了单独的包 `create-react-class` 中。

第二站：类组件

ES2015 为 JavaScript 添加类句法后，React 引入了一种创建 React 组件的新方法。`React.Component` API 支持使用类句法创建组件实例：

```
class IngredientsList extends React.Component {
  render() {
    return React.createElement(
      "ul",
      { className: "ingredients" },
      this.props.items.map((ingredient, i) =>
        React.createElement("li", { key: i }, ingredient)
      )
    );
  }
}
```

现在依然可以使用类句法创建React组件，但是笔者要提前告诉你，React.Component即将被弃用。现在虽然还支持，但可以预见，它也将步React.createClass的后尘。这位老朋友虽然让我们获益不少，但是现在不常见到了，世事无常，我们自己也在变。从现在开始，本书都使用函数创建组件，偶尔指出旧的模式，仅供参考。

第5章

在React中使用JSX

前一章讲解了React的运行机制，我们知道一个React应用是由一系列小型的可重用片段即组件构成的。组件渲染元素树和其他组件。通过`createElement`函数可以清楚地看出React是如何运作的，可是作为React开发者，我们不会这么做。自己动手编写复杂、可读性却不高的JavaScript句法树可不是一件令人愉快的事。为了高效使用React，我们还需要一个工具：JSX。

JSX是JavaScript中的JS与XML中的X的综合体，是对JavaScript的扩展，使用一种基于标签的句法直接在JavaScript代码中定义React元素。JSX容易与HTML混淆，毕竟二者看起来很像。JSX也是一种创建React元素的方式，有了它你就再也不用为复杂的`createElement`调用中缺少逗号而抓狂了。

本章讨论如何使用JSX构建React应用。

5.1 使用JSX创建React元素

Facebook的React团队在发布React时也发布了JSX，目的是提供一种简洁的句法创建带有属性的复杂的DOM树，另外也希望React代码像HTML和XML一样易于阅读。在JSX中，元素的类型通过标签指定，标签的属性表示元素的属性，子元素添加在起始和结束标签之间。

子元素也可以是其他 JSX 元素。对一个无序列表来说，列表项目就可以通过 JSX 标签添加。得到的结果与 HTML 十分相似。

```
<ul>
  <li>1 lb Salmon</li>
  <li>1 cup Pine Nuts</li>
  <li>2 cups Butter Lettuce</li>
  <li>1 Yellow Squash</li>
  <li>1/2 cup Olive Oil</li>
  <li>3 Cloves of Garlic</li>
</ul>
```

JSX 还可以与组件一起使用，只需使用类名定义组件即可。在图 5-1 中，我们使用 JSX 的属性把配料数组传给 IngredientsList。

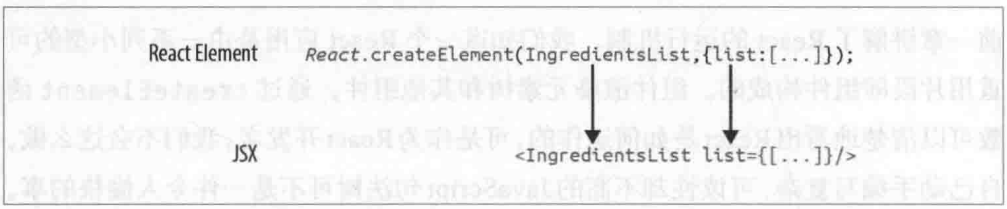

图 5-1：使用 JSX 创建 IngredientsList

把配料数组传给组件时，要把数组放在一对花括号内。这叫 JavaScript 表达式（expression），通过属性把 JavaScript 值传给组件时，必须这么做。组件的属性有两种类型：字符串或 JavaScript 表达式。JavaScript 表达式可以是数组、对象，甚至是函数。传入的表达式必须放在一对花括号内。

5.1.1 JSX 小贴士

JSX 看着眼熟，而且多数指令得到的句法都与 HTML 类似。然而，使用 JSX 时还是要注意一些事项。

嵌套组件

JSX 支持把组件添加为组件的子元素。例如，在 IngredientsList 组件中可以多次渲染另一个名为 Ingredient 的组件。

```
<IngredientsList>
  <Ingredient />
  <Ingredient />
  <Ingredient />
</IngredientsList>
```

className

由于 class 是 JavaScript 的保留字，因此定义 class 属性时要使用 className。

```
<h1 className="fancy">Baked Salmon</h1>
```

JavaScript 表达式

JavaScript 表达式放在一对花括号内，表示要对变量求值，并返回得到的值。比如说，我们想在一个元素中显示 title 属性的值，此时就可以使用一个 JavaScript 表达式插入那个值。插入的变量经求值后返回结果。

```
<h1>{title}</h1>
```

字符串以外的值都要通过 JavaScript 表达式传入：

```
<input type="checkbox" defaultChecked={false} />
```

求值

一对花括号之间的 JavaScript 代码会做求值。这意味着，拼接或相加等操作都会正确执行。这也意味着，JavaScript 表达式中的函数也会被调用。

```
<h1>{"Hello" + title}</h1>

<h1>{title.toLowerCase().replace}</h1>
```

5.1.2 使用 JSX 映射数组

JSX 是 JavaScript，可以直接在 JavaScript 函数中使用。例如，你可以把数组映射为 JSX 元素：

```
<ul>
  {props.ingredients.map((ingredient, i) => (
    <li key="{i}">{ingredient}</li>
  ))}
</ul>
```

JSX 简洁，可读性高，但是浏览器无法解释。所有 JSX 都要转换成 `createElement` 调用。幸好，有一个工具可以胜任这项工作：Babel。

5.2 Babel

很多软件开发语言需要编译源码。JavaScript 是一门解释型语言，浏览器把代码解释为文本，无须编译。然而，不是所有浏览器都支持最新的 JavaScript 句法，而且没有浏览器支持 JSX 句法。由于我们想在 JSX 中使用最新的 JavaScript 特性，那就需要找到一种方式把我们写出的源码转换成浏览器可以解释的版本。这个过程叫作编译，Babel（*https://babeljs.io*）就是做这项工作的。

这个项目的一个版本叫 6to5，于 2014 年 9 月发布。6to5 把 ES6 句法转换成 Web 浏览器支持情况更好的 ES5 句法。随着项目的发展，它的目标变成了支持 ECMAScript 所有的最新变化。而且，后来还支持把 JSX 转换成 JavaScript。2015 年 2 月，项目的名称改成了 Babel。

Facebook、Netflix、PayPal、Airbnb 等公司都在工作中使用 Babel。以前，Facebook 推出过一个 JSX 转换程序，这是当时的标准工具，不过随着 Babel 的出现，这个工具便退出舞台了。

使用 Babel 的方式有好多种，最容易上手的是直接在 HTML 中通过 CDN 引入 Babel，编译类型为“text/babel”的脚本块中的代码。Babel 在客户端运行源码

之前编译代码。这虽然不是在生产环境中使用 Babel 的最佳方式，却是开始着手使用 JSX 的好方法。

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Examples</title>
  </head>
  <body>
    <div id="root"></div>

    <!-- React 和 React DOM 库 -->
    <script
  src="https://unpkg.com/react@16.8.6/umd/react.development.js">
  </script>
    <script
  src="https://unpkg.com/react-dom@16.8.6/umd/react-dom.development.js">
  </script>
    <script
  src="https://unpkg.com/@babel/standalone/babel.min.js">
  </script>

    <script type="text/babel">
      // 在这里编写 JSX 代码，或者链接保存 JSX 的 JavaScript 文件
    </script>
  </body>
</html>
```



在浏览器内使用 Babel 时控制台发出的警告

直接在浏览器内使用转换程序，你会看到一个警告，要求我们事先编译好脚本，供生产环境使用。对这个示例和其他小型演示应用来说，不要在意这个警告。本章后文将说明如何为生产环境做好准备。

5.3 使用 JSX 编写菜谱应用

使用 JSX 可以简洁明了地表达 React 元素，我们自己在编写代码时思路清晰，而且其他开发者在阅读时也容易理解。JSX 的缺点是浏览器读不懂。浏览器在解释代码之前，要把 JSX 转换成 JavaScript。

下面的 data 数组中有两个菜谱，这表示应用的当前状态：

```
const data = [
  {
    name: "Baked Salmon",
    ingredients: [
      { name: "Salmon", amount: 1, measurement: "l lb" },
      { name: "Pine Nuts", amount: 1, measurement: "cup" },
      { name: "Butter Lettuce", amount: 2, measurement: "cups" },
      { name: "Yellow Squash", amount: 1, measurement: "med" },
      { name: "Olive Oil", amount: 0.5, measurement: "cup" },
      { name: "Garlic", amount: 3, measurement: "cloves" }
    ],
    steps: [
      "Preheat the oven to 350 degrees.",
      "Spread the olive oil around a glass baking dish.",
      "Add the yellow squash and place in the oven for 30 mins.",
      "Add the salmon, garlic, and pine nuts to the dish.",
      "Bake for 15 minutes.",
      "Remove from oven. Add the lettuce and serve."
    ]
  },
  {
    name: "Fish Tacos",
    ingredients: [
      { name: "Whitefish", amount: 1, measurement: "l lb" },
      { name: "Cheese", amount: 1, measurement: "cup" },
      { name: "Iceberg Lettuce", amount: 2, measurement: "cups" },
      { name: "Tomatoes", amount: 2, measurement: "large" },
      { name: "Tortillas", amount: 3, measurement: "med" }
    ],
    steps: [
      "Cook the fish on the grill until cooked through.",
      "Place the fish on the 3 tortillas.",
```

```
        "Top them with lettuce, tomatoes, and cheese."
      ]
    }
];
```

这份数据由两个 JavaScript 对象构成的数组表示，各对象中有菜谱的名称、所需的配料清单和制作菜肴的操作步骤。

我们可以使用两个组件为这些菜谱创建 UI：一个 Menu 组件用于列出菜谱，一个 Recipe 组件描述各个菜谱的 UI。我们要在 DOM 中渲染的是 Menu 组件。数据通过一个名为 recipes 的属性传给 Menu 组件。

```
// 数据，一个由 Recipe 对象构成的数组
const data = [ ... ];

// 一个函数组件，描述单个菜谱
function Recipe (props) {
  ...
}

// 一个函数组件，列出菜谱
function Menu (props) {
  ...
}

// 调用 ReactDOM.render，在当前的 DOM 中渲染 Menu
ReactDOM.render(
  <Menu recipes={data} title="Delicious Recipes" />,
  document.getElementById("root")
);
```

Menu 组件中的 React 元素使用 JSX 表达。所有内容都放在一个 article 元素中。Menu 的 DOM 包含一个 header 元素、一个 h1 元素和一个 div.recipes 元素。title 属性的值显示为 h1 的文本。

```
function Menu(props) {
  return (
    <article>
```

```
        <header>
          <h1>{props.title}</h1>
        </header>
        <div className="recipes" />
      </article>
    );
  }
```

然后，在 div.recipes 元素中为每个菜谱添加一个组件：

```
<div className="recipes">
  {props.recipes.map((recipe, i) => (
    <Recipe
      key={i}
      name={recipe.name}
      ingredients={recipe.ingredients}
      steps={recipe.steps}
    />
  ))}
</div>
```

为了在 div.recipes 元素中列出菜谱，我们把 JavaScript 表达式放在一对花括号中，这个表达式返回一个包含子元素的数组。我们在 props.recipes 数组上调用 map 函数为数组中的每个对象返回一个组件。前面说过，每个菜谱都有一个名称、一些配料和操作说明（步骤）。我们要通过属性把这些数据传给各个 Recipe 组件。另外，别忘了使用 key 属性为每个元素设置唯一的标识。

我们还可以使用展开句法重构这段代码。JSX 的展开运算符用起来与对象的展开运算符一样。展开运算符将把 recipe 对象的每个字段作为属性传给 Recipe 组件。下述句法将把所有属性都提供给组件。

```
{
  props.recipes.map((recipe, i) => <Recipe key={i} {...recipe} />);
}
```

注意，上述简短的句法即可以把所有属性提供给 Recipe 组件。这样做是不错，但是提供给组件的属性有时太多了。

Menu 组件还有一处句法可以改进，即接受的 props 参数。我们可以使用对象析构限定传给函数的变量。这样一来，我们便可以直接访问 title 和 recipes 变量，不用再加上 props 前缀。

```
function Menu({ title, recipes }) {
  return (
    <article>
      <header>
        <h1>{title}</h1>
      </header>
      <div className="recipes">
        {recipes.map((recipe, i) => (
          <Recipe key={i} {...recipe} />
        ))}
      </div>
    </article>
  );
}
```

下面来编写各个单独菜谱的组件：

```
function Recipe({ name, ingredients, steps }) {
  return (
    <section id={name.toLowerCase().replace(/ /g, "-")}>
      <h1>{name}</h1>
      <ul className="ingredients">
        {ingredients.map((ingredient, i) => (
          <li key={i}>{ingredient.name}</li>
        ))}
      </ul>
      <section className="instructions">
        <h2>Cooking Instructions</h2>
        {steps.map((step, i) => (
          <p key={i}>{step}</p>
        ))}
      </section>
    </section>
  );
}
```

各菜谱有一个字符串名称、一个配料对象数组和一个操作步骤字符串数组。使用对象析构可以让组件把这些字段的名称限定在本地作用域中，以便直接访问，而不用再使用 `props.name`、`props.ingredients` 或 `props.steps`。

组件中的第一个 JavaScript 表达式用于设置根元素 `section` 的 `id` 属性。这个表达式把菜谱的名称转换成小写字符串，再把空格全局替换为连字符。结果是，在 UI 中设置 `id` 属性之前，把“Baked Salmon”转换成“baked-salmon”（同样，如果菜谱的名称是“Boston Baked Beans,”，那么得到的结果是“boston-baked-beans”）。`name` 的值还作为文本节点显示在 `h1` 中。

在无序列表中，我们使用一个 JavaScript 表达式把每个配料映射为一个 `li` 元素，显示配料的名称。在操作说明部分，使用相同的方式返回段落元素，显示每一步。这里，`map` 函数返回由子元素构成的数组。

这个应用的完整代码如下所示：

```
const data = [
  {
    name: "Baked Salmon",
    ingredients: [
      { name: "Salmon", amount: 1, measurement: "l lb" },
      { name: "Pine Nuts", amount: 1, measurement: "cup" },
      { name: "Butter Lettuce", amount: 2, measurement: "cups" },
      { name: "Yellow Squash", amount: 1, measurement: "med" },
      { name: "Olive Oil", amount: 0.5, measurement: "cup" },
      { name: "Garlic", amount: 3, measurement: "cloves" }
    ],
    steps: [
      "Preheat the oven to 350 degrees.",
      "Spread the olive oil around a glass baking dish.",
      "Add the yellow squash and place in the oven for 30 mins.",
      "Add the salmon, garlic, and pine nuts to the dish.",
      "Bake for 15 minutes.",
      "Remove from oven. Add the lettuce and serve."
    ]
  },
  {
```

```
    name: "Fish Tacos",
    ingredients: [
      { name: "Whitefish", amount: 1, measurement: "1 lb" },
      { name: "Cheese", amount: 1, measurement: "cup" },
      { name: "Iceberg Lettuce", amount: 2, measurement: "cups" },
      { name: "Tomatoes", amount: 2, measurement: "large" },
      { name: "Tortillas", amount: 3, measurement: "med" }
    ],
    steps: [
      "Cook the fish on the grill until hot.",
      "Place the fish on the 3 tortillas.",
      "Top them with lettuce, tomatoes, and cheese."
    ]
  }
];

function Recipe({ name, ingredients, steps }) {
  return (
    <section id={name.toLowerCase().replace(/ /g, "-")}>
      <h1>{name}</h1>
      <ul className="ingredients">
        {ingredients.map((ingredient, i) => (
          <li key={i}>{ingredient.name}</li>
        ))}
      </ul>
      <section className="instructions">
        <h2>Cooking Instructions</h2>
        {steps.map((step, i) => (
          <p key={i}>{step}</p>
        ))}
      </section>
    </section>
  );
}

function Menu({ title, recipes }) {
  return (
    <article>
      <header>
        <h1>{title}</h1>
      </header>
```

```
        <div className="recipes">
          {recipes.map((recipe, i) => (
            <Recipe key={i} {...recipe} />
          ))}
        </div>
      </article>
    );
  }

ReactDOM.render(
  <Menu recipes={data} title="Delicious Recipes" />,
  document.getElementById("root")
);
```

在浏览器中运行这段代码，React 会使用我们指定的菜谱数据构建 UI，如图 5-2 所示。

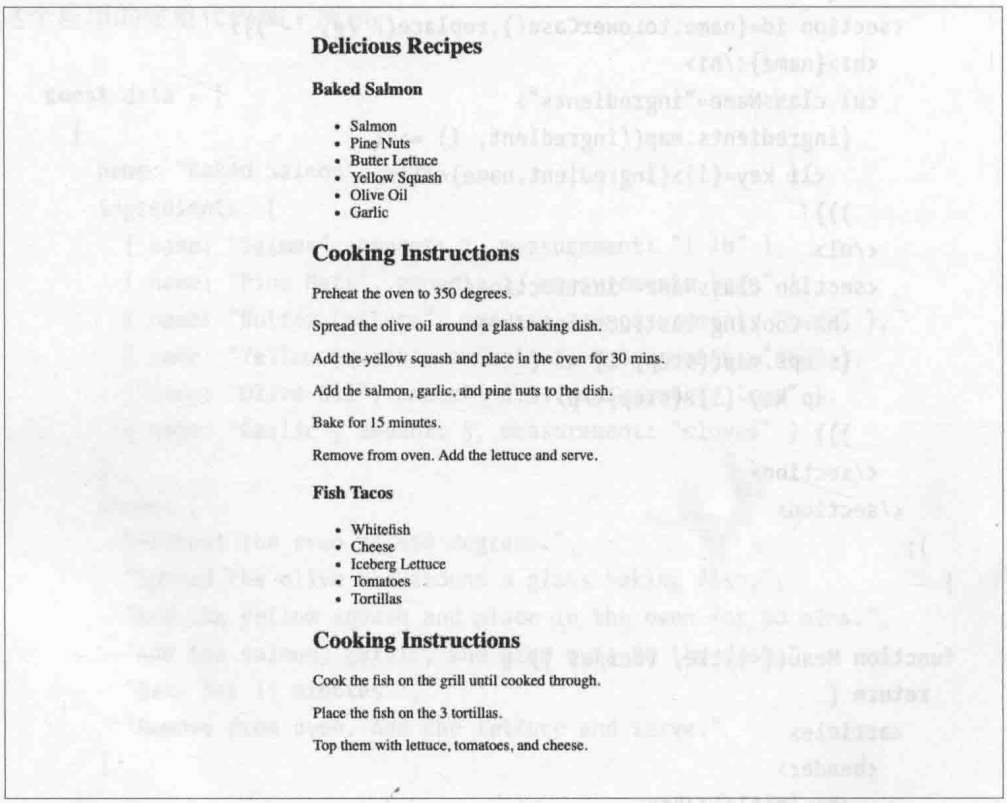

图 5-2：输出的诱人菜谱

如果你使用的是 Google Chrome，而且安装了 React 开发者工具扩展，可以看一下组件树的当前状态。查看方法是打开开发者工具，选择“Components”标签页，如图 5-3 所示。

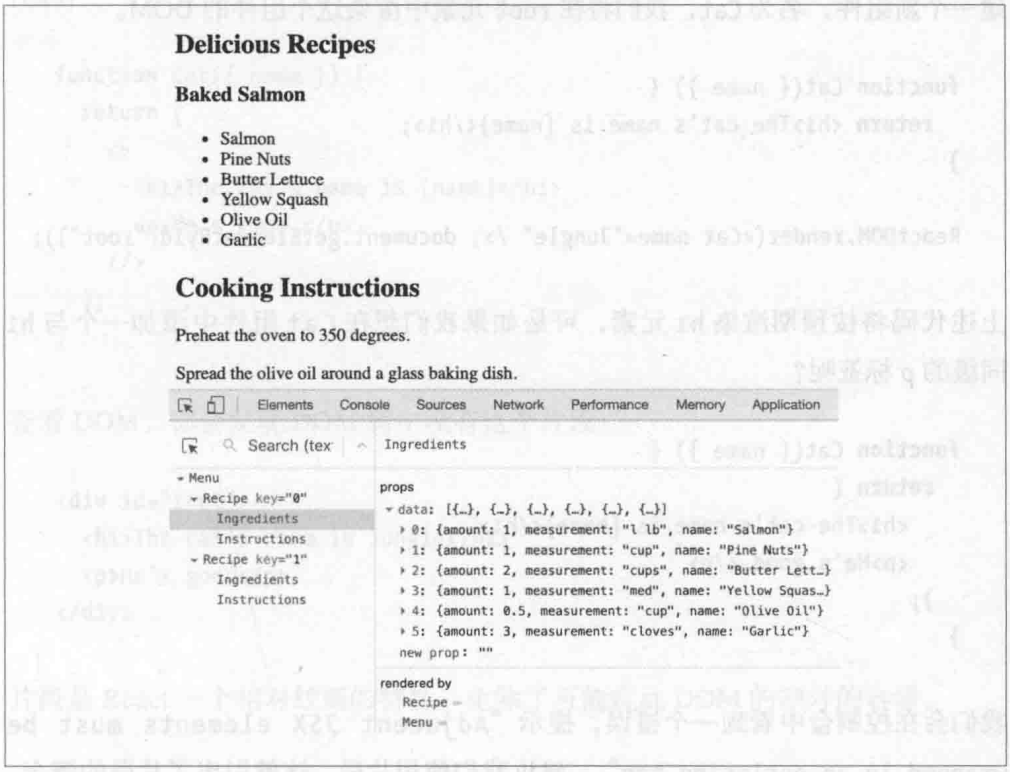

图 5-3：在 React 开发者工具中查看得到的虚拟 DOM

在这个标签页中可以看到 `Menu` 及其子元素。`data` 数组中有两个菜谱对象，UI 中有两个 `Recipe` 元素。每个 `Recipe` 元素都有菜谱名称、配料和操作步骤三个属性。配料和操作步骤通过 `data` 传给对应的组件。

菜谱组件根据通过属性传给 `Menu` 组件的数据构建。如果修改 `recipes` 数组的内容，然后重新渲染 `Menu` 组件，React 将尽可能高效地修改 DOM。

5.4 React 片段

前一节渲染的 Menu 组件是一个父级组件，它又渲染 Recipe 组件。我们来花点儿时间看一个小例子：使用一个 React 片段（fragment）渲染两个同辈组件。首先创建一个新组件，名为 Cat，我们将在 root 元素中渲染这个组件的 DOM。

```
function Cat({ name }) {
  return <h1>The cat's name is {name}</h1>;
}

ReactDOM.render(<Cat name="Jungle" />, document.getElementById("root"));
```

上述代码将按预期渲染 h1 元素，可是如果我们想在 Cat 组件中添加一个与 h1 同级的 p 标签呢？

```
function Cat({ name }) {
  return (
    <h1>The cat's name is {name}</h1>
    <p>He's good.</p>
  );
}
```

我们会在控制台中看到一个错误，提示“Adjacent JSX elements must be wrapped in an enclosing tag”，建议我们使用片段。这就引出了片段的概念。React 不在一个组件中渲染两个或以上毗邻元素或同辈元素，必须把这些元素放在一个标签内，例如一个 div。然而，这样会导致创建很多非必需的标签，产生很多没什么用的容器。使用 React 片段，我们可以模拟容器的行为，但不真正创建新标签。

先把毗邻的标签，即 h1 和 p 放在 React.Fragment 标签内：

```
function Cat({ name }) {
  return (
    <React.Fragment>
      <h1>The cat's name is {name}</h1>
      <p>He's good.</p>
    </React.Fragment>
```

```
  );
}
```

添加这个标签后，警告就没有了。我们还可以使用片段的简写句法进一步精简代码：

```
function Cat({ name }) {
  return (
    <>
      <h1>The cat's name is {name}</h1>
      <p>He's good.</p>
    </>
  );
}
```

查看 DOM，你会发现 DOM 树中没有这个片段：

```
<div id="root">
  <h1>The cat's name is Jungle</h1>
  <p>He's good</p>
</div>
```

片段是 React 一个相对较新的特性，免除了可能扰乱 DOM 的额外的容器。

5.5 webpack 简介

使用 React 开发真正的项目时，需要考虑很多问题：如何转换 JSX 和 ESNext？如何管理依赖？如何优化图像和 CSS？

为了解决这些问题，涌现出了很多工具，例如 Browserify、gulp、Grunt、Prepack 等。webpack 功能丰富，在大公司中使用广泛，很快就爬升到了构建工具的领军位置。

React 生态系统很成熟，有 create-react-app、Gatsby 和 CodeSandbox 这样的工具。然而，这些工具把很多细节隐藏起来了，你并不知道代码是如何编译的。本章余下的内容说明如何使用 webpack 构建。今时今日，我们都知道 JavaScript/

React 代码要经过 webpack 之类的工具编译，至于 JavaScript/React 代码是如何编译的就没多少人关心了。如果你想跳过这部分内容，笔者也理解。

webpack 在宣传上称自己为模块打包工具，负责把各种文件（JavaScript、LESS、CSS、JSX、ESNext 等）汇集成一个文件。打包的好处主要体现在两方面：模块化和网络性能。

模块化指把源码分解为不同的部分，即模块，这样更便于展开工作，尤其是在团队中。

网络性能的提升体现在浏览器只加载一个依赖，即构建包。每个 script 标签发起一个 HTTP 请求，而每个 HTTP 请求都有延迟损耗。把所有依赖打包到一个文件中，所有依赖通过一个 HTTP 请求就能加载，规避了额外的延迟。

除了编译代码之外，webpack 还可以处理以下事项。

代码分拆

把代码分成一块块，在需要时加载。这个过程有时也叫作汇总（rollup）或分层（layer），目的是按照不同页面或设备的需求打散代码。

代码简化

去掉空白、换行符、较长的变量名称和不必要的代码，减少文件大小。

特性标记

把代码发给一个或多个（但不是所有）环境，测试特性。

模块热替换（Hot Module Replacement，HMR）

监视源码的变化，立即更新有变化的模块。

本章前面构建的菜谱应用有些不足之处，webpack 可以帮助我们弥补。使用 webpack 这类工具静态构建客户端 JavaScript，方便团队协同开发大型 Web 应用。采用模块打包工具 webpack 还可以获得以下好处。

模块化

使用模块模式导出的模块可在应用的其他部分导入或引入，源码浅显易懂。模块化方便开发团队协同工作，各自负责编写自己的文件，在上线之前再合并成一个静态文件。

合成

借助模块，我们可以构建简单、可重用的小型React组件，再高效组合在一起，构成完整的应用。小型组件易于理解、测试和重用，需要增强应用的功能时也方便替换。

速度

把应用中的所有模块和依赖打包成一个客户端构建包能减少应用的加载时间，因为每个HTTP请求都有延迟。把一切打包进一个文件，客户端就只需发起一个请求。简化构建包中的代码也能减少加载时间。

一致性

webpack可以把JSX编译成JavaScript，现在就可以使用未来的JavaScript句法。Babel支持大量ESNext句法，我们无须担心浏览器是否支持我们写出的代码。开发者可以始终使用最新的JavaScript句法。

5.5.1 创建项目

下面请在你的电脑中新建一个文件夹，命名为recipes-app，我们将演示如何从头开始设置一个React项目。

```
mkdir recipes-app
cd recipes-app
```

对这个项目，我们将采取下述操作步骤。

1. 创建项目。
2. 把菜谱应用分解为一个个组件，保存在不同的文件中。
3. 设置webpack，与Babel协同构建。



create-react-app

有个可以自动生成预先配置好一切的工具，名为 CreateReact App。不过，这个工具隐藏了具体的步骤，在使用它之前我们要了解一下背后的机制。

1. 创建项目

接下来，使用 npm 创建这个项目和 *package.json* 文件，传入 -y 标志，一切都采用默认值。然后，安装 webpack、webpack-cli、react 和 react-dom。

```
npm init -y
npm install react react-dom serve
```

如果你使用的是 npm 5，安装各个包时无须传入 `--save` 标志。接下来，我们要创建如下的目录结构，存放各个组件。

```
recipes-app (folder)
  > node_modules (already added with npm install command)
  > package.json (already added with npm init command)
  > package-lock.json (already added with npm init command)
  > index.html
  > /src (folder)
    > index.js
    > /data (folder)
      > recipes.json
    > /components (folder)
      > Recipe.js
      > Instructions.js
      > Ingredients.js
```



文件组织方式

React 项目中文件的组织方式没有统一标准。这里给出的只是一种可能的方式。

2. 按模块划分组件

目前，Recipe组件做的事情有点多，显示了菜谱的名称、用无序列表列出了配料，还显示了操作说明，每一步都放在一个段落元素中。这个组件应该放在*Recipe.js*文件中。只要使用JSX，我们就要在文件顶部导入React。

```
// ./src/components/Recipe.js

import React from "react";

export default function Recipe({ name, ingredients, steps }) {
  return (
    <section id="baked-salmon">
      <h1>{name}</h1>
      <ul className="ingredients">
        {ingredients.map((ingredient, i) => (
          <li key={i}>{ingredient.name}</li>
        ))}
      </ul>
      <section className="instructions">
        <h2>Cooking Instructions</h2>
        {steps.map((step, i) => (
          <p key={i}>{step}</p>
        ))}
      </section>
    </section>
  );
}
```

更符合函数式编程风格的做法是把Recipe组件拆解成多个功能更专一的小型组件，然后再组合在一起。首先，我们可以把操作说明拿出来，放到单独的组件中，新建一个模块文件，方便在其他地方使用。

在新建的*Instructions.js*文件中创建如下的组件。

```
// ./src/components/Instructions.js

import React from "react";
```

```
export default function Instructions({ title, steps }) {
  return (
    <section className="instructions">
      <h2>{title}</h2>
      {steps.map((s, i) => (
        <p key={i}>{s}</p>
      ))}
    </section>
  );
}
```

这里，我们创建了一个名为 Instructions 的组件，传入操作说明的标题和步骤。如此一来，我们便可以重复使用这个组件创建“烹调说明”“烘焙说明”“备料说明”或“厨前检查表”等分步操作。

再来看配料。Recipe 组件只显示配料的名称，可是一道菜的每种配料都需要指定量和单位。为此，我们可以创建一个名为 Ingredient 的组件。

```
// ./src/components/Ingredient.js

import React from "react";

export default function Ingredient({ amount, measurement, name }) {
  return (
    <li>
      {amount} {measurement} {name}
    </li>
  );
}
```

这里，我们假定每种配料都有一定的量、单位和名称。这些数据从 props 对象中析构而来，各自显示在一个指定了类的 span 元素中。

有了 Ingredient 组件，我们可以构建一个 IngredientsList 组件，在需要显示配料清单的地方使用。

```
// ./src/components/IngredientsList.js
```

```
import React from "react";
import Ingredient from "./Ingredient";

export default function IngredientsList({ list }) {
  return (
    <ul className="ingredients">
      {list.map((ingredient, i) => (
        <Ingredient key={i} {...ingredient} />
      ))}
    </ul>
  );
}
```

在这个文件中，我们首先导入 Ingredient 组件，用于显示各个配料。配料以数组的形式，通过 list 属性传入组件。list 数组中的每个配料映射到 Ingredient 组件上。我们使用 JSX 展开运算符，通过属性把所有数据传给 Ingredient 组件。

下述展开运算符：

```
<Ingredient {...ingredient} />
```

是下面这个语句的另一种表达方式：

```
<Ingredient
  amount={ingredient.amount}
  measurement={ingredient.measurement}
  name={ingredient.name}
/>
```

对包含下述字段的配料对象来说：

```
let ingredient = {
  amount: 1,
  measurement: "cup",
  name: "sugar"
};
```

得到的结果为：

```
<Ingredient amount={1} measurement="cup" name="sugar" />
```

创建好配料和操作说明相关的组件之后，我们可以组合使用这些组件构建菜谱：

```
// ./src/components/Recipe.js

import React from "react";
import IngredientsList from "./IngredientsList";
import Instructions from "./Instructions";

function Recipe({ name, ingredients, steps }) {
  return (
    <section id={name.toLowerCase().replace(/ /g, "-")}>
      <h1>{name}</h1>
      <IngredientsList list={ingredients} />
      <Instructions title="Cooking Instructions" steps={steps} />
    </section>
  );
}

export default Recipe;
```

首先，导入要使用的组件：IngredientsList 和 Instructions。导入之后就可以用它们创建 Recipe 组件了。我们不再集中一处使用一堆复杂的代码构建整个菜谱，而是采用更符合声明式编程的风格把小型组件组合在一起。这样写出的代码简洁明了，也易于阅读。可以看出，菜谱中会显示名称、配料列表和操作说明。我们把配料和操作说明的具体显示方式隐藏到简单的小型组件中了。

在模块化方案中，Menu 组件大致相同。只不过，Menu 组件保存在单独的文件中，导入需要使用的模块，再导出自身：

```
// ./src/components/Menu.js

import React from "react";
import Recipe from "./Recipe";

function Menu({ recipes }) {
  return (
```

```
      <article>
        <header>
          <h1>Delicious Recipes</h1>
        </header>
        <div className="recipes">
          {recipes.map((recipe, i) => (
            <Recipe key={i} {...recipe} />
          ))}
        </div>
      </article>
    );
  }

  export default Menu;
```

我们还需要使用 ReactDOM 渲染 Menu 组件。这个项目的主文件是 *index.js*，负责在 DOM 中渲染组件。

创建这个文件：

```
// ./src/index.js

import React from "react";
import { render } from "react-dom";
import Menu from "./components/Menu";
import data from "./data/recipes.json";

render(<Menu recipes={data} />, document.getElementById("root"));
```

前四个语句导入这个应用所需的模块。我们没有通过 `script` 标签加载 `react` 和 `react-dom`，而是导入它们，这样 webpack 便可以把它们添加到构建包中。我们还从单独的文件中导入了 Menu 组件和示例数据数组。示例数据中有两个菜谱，Baked Salmon 和 Fish Tacos。

导入的所有变量，作用域都在 *index.js* 文件本地。渲染 Menu 组件时，通过属性把菜谱数据数组传给组件。

数据从 *recipes.json* 文件中获取。这与本章前面使用的数据是一样的，不过现在是用有效的 JSON 格式编写的。

```
// ./src/data/recipes.json

[
  {
    "name": "Baked Salmon",
    "ingredients": [
      { "name": "Salmon", "amount": 1, "measurement": "lb" },
      { "name": "Pine Nuts", "amount": 1, "measurement": "cup" },
      { "name": "Butter Lettuce", "amount": 2, "measurement": "cups" },
      { "name": "Yellow Squash", "amount": 1, "measurement": "med" },
      { "name": "Olive Oil", "amount": 0.5, "measurement": "cup" },
      { "name": "Garlic", "amount": 3, "measurement": "cloves" }
    ],
    "steps": [
      "Preheat the oven to 350 degrees.",
      "Spread the olive oil around a glass baking dish.",
      "Add the yellow squash and place in the oven for 30 mins.",
      "Add the salmon, garlic, and pine nuts to the dish.",
      "Bake for 15 minutes.",
      "Remove from oven. Add the lettuce and serve."
    ]
  },
  {
    "name": "Fish Tacos",
    "ingredients": [
      { "name": "Whitefish", "amount": 1, "measurement": "lb" },
      { "name": "Cheese", "amount": 1, "measurement": "cup" },
      { "name": "Iceberg Lettuce", "amount": 2, "measurement": "cups" },
      { "name": "Tomatoes", "amount": 2, "measurement": "large" },
      { "name": "Tortillas", "amount": 3, "measurement": "med" }
    ],
    "steps": [
      "Cook the fish on the grill until cooked through.",
      "Place the fish on the 3 tortillas.",
      "Top them with lettuce, tomatoes, and cheese."
    ]
```

```
  }
]
```

至此，我们把代码分成不同的模块和文件了，接下来要创建一个构建过程，再把一切都汇总到一个文件中。你可能心生疑惑，“慢着，我们大费周章才把一切分拆开来，现在又要使用一个工具把所有东西合为一体，有这个必要吗？”把一个项目分拆为不同的文件，主要的好处是便于管理大型项目，方便团队成员分工协作，不产生交叉。这样也便于单独测试各个文件。

3. 使用 webpack 构建

为了使用 webpack 创建静态构建过程，我们要安装一些包。我们所需的一切都可以使用 npm 安装。

```
npm install --save-dev webpack webpack-cli
```

别忘了，我们已经安装了 React 和 ReactDOM。

为了让这个模块化的菜谱应用能顺利运行，我们要告诉 webpack 如何把源码打包成一个文件。从 4.0.0 版开始，webpack 无须配置文件就可以打包项目。没有配置文件，webpack 就使用默认设置打包代码。不过，使用配置文件可以定制打包过程。而且，我们也可以窥见被 webpack 隐藏起来的部分神秘操作。webpack 默认使用的配置文件名为 *webpack.config.js*。

这个菜谱应用的入口文件是 *index.js*，在这个文件中导入了 React、ReactDOM 和 *Menu.js* 文件。在浏览器中，我们就想先运行这个文件。只要 webpack 发现 `import` 语句，就会在文件系统中寻找对应的模块，把它放到构建包中。*index.js* 导入 *Menu.js*，*Menu.js* 导入 *Recipe.js*，*Recipe.js* 导入 *Instructions.js* 和 *IngredientsList.js*，*IngredientsList.js* 又导入 *Ingredient.js*。webpack 将沿着导入树，把用到的所有模块都放入构建包中。遍历这些文件得到一个依赖图（dependency graph）。依赖指应用需要用到的什么东西，可以是一个组件文件、一个库（例如 React）或一个图像。各个文件相当于图上的一个个圆圈，而 webpack 则负责划线，把这些圆圈连起来，形成依赖图。整个依赖图就是构建包。



import 语句

我们使用的 `import` 语句，目前多数浏览器和 Node.js 都不支持。但是，我们依然可以使用，因为在最终的代码中 Babel 将把 `import` 语句转换成 `require('module/path');`。CommonJS 模块通常使用 `require` 函数。

使用 webpack 构建时，我们要指明把 JSX 转换成 React 元素。

webpack.config.js 文件其实也是一个模块，导出一个 JavaScript 字面量对象，描述 webpack 要执行的操作。这个配置文件应该保存在项目的根文件夹中，与 *index.js* 文件同级。

```
// ./webpack.config.js

var path = require("path");

module.exports = {
  entry: "./src/index.js",
  output: {
    path: path.join(__dirname, "dist", "assets"),
    filename: "bundle.js"
  }
};
```

首先，告诉 webpack，客户端的入口文件为 *./src/index.js*。webpack 将根据这个文件中的 `import` 语句自动构建依赖图。然后，指明我们想把打包后的 JavaScript 输出到 *./dist/bundle.js* 文件中。webpack 最终打包好的 JavaScript 将保存在这个文件中。

下面我们要安装所需的 Babel 依赖：`babel-loader` 和 `@babel/core`。

```
npm install babel-loader @babel/core --save-dev
```

接下来，我们要设置的 webpack 指令是一些 loader，在特定的模块上运行。这部分指令添加到配置文件中的 `module` 字段名下。

```
module.exports = {
  entry: "./src/index.js",
  output: {
    path: path.join(__dirname, "dist", "assets"),
    filename: "bundle.js"
  },
  module: {
    rules: [{ test: /\.js$/, exclude: /node_modules/, loader: "babel-loader" }]
  }
};
```

`rules` 字段的值是一个数组，因为 webpack 支持的 loader 有好多种。本例只用到了 `babel-loader`。每个 loader 在一个 JavaScript 对象中指定。`test` 字段的值是一个正则表达式，匹配模块的文件路径，执行相关的操作。这里，我们在所有导入的 JavaScript 文件上运行 `babel-loader`，但把 *node_modules* 文件夹排除在外。

现在，我们要指定运行 Babel 的预设（preset）。Babel 通过预设设置执行什么转换操作。比如说，我们可以说，“Babel，如果你遇到 ESNext 句法，把它转换成浏览器可以理解的句法。如果你遇到 JSX，也这样转换。”先来安装预设：

```
npm install @babel/preset-env @babel/preset-react --save-dev
```

然后，在项目的根文件夹中新建一个文件，命名为 *.babelrc*。

```
{
  "presets": ["@babel/preset-env", "@babel/preset-react"]
}
```

至此，我们便创建好了一个模拟真实 React 应用的项目。下面来运行 webpack，确保一切正常。

webpack 是静态运行的。通常，构建包在把应用部署到服务器之前创建。webpack 可以在命令行中使用 npx 运行：

```
npx webpack --mode development
```

webpack 在成功运行后创建一个构建包，倘若失败会报错。多数错误与导入目标失效有关。调试 webpack 错误时，请特别留意 `import` 语句中的文件名和路径。

你还可以在 *package.json* 文件中添加一个 npm 脚本，创建一个快捷命令：

```
"scripts": {
  "build": "webpack --mode production"
},
```

这样就可以运行快捷命令生成构建包了：

```
npm run build
```

5.5.2 加载构建包

现在构建包有了，接下来呢？我们把构建包导出到 *dist* 文件夹中，这个文件夹中就是要在 Web 服务器中运行的文件。*index.html* 文件应该放在 *dist* 文件夹中。这个文件中应该有一个 `div` 元素，用于挂载 React `Menu` 组件。此外，还要有一个 `script` 标签，加载打包好的 JavaScript。

```
// ./dist/index.html

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Recipes App</title>
  </head>
  <body>
    <div id="root"></div>
    <script src="bundle.js"></script>
  </body>
</html>
```

这是应用的首页，通过一个 HTTP 请求从一个文件中加载所有代码，即 *bundle.js*。你要把这些文件部署到 Web 服务器中，或者使用 Node.js 或 Ruby on Rails 构建一个 Web 服务器端应用，伺服这些文件。

5.5.3 源码映射

把代码打包进一个文件，在浏览器中调试应用可能有些不便。为了缓解这方面的问题，我们可以提供源码映射（source mapping）。源码映射把构建包映射到源文件上。使用 webpack，我们只需要在 *webpack.config.js* 文件中添加几行代码：

```
// 在 webpack.config.js 中设置源码映射

module.exports = {
  ...
  devtool: "#source-map" // 添加这个选项，开启源码映射
};
```

把 devtool 属性设为 `'#source-map'`，这样 webpack 便知道我们想使用源码映射。再次运行 webpack，你会看到 *dist* 文件夹中生成了两个文件：老面孔 *bundle.js* 和新出现的 *bundle.js.map*。

源码映射的作用是方便我们使用源码文件调试。在浏览器开发者工具的“Sources”标签页中，你会看到一个名为 *webpack://* 的文件夹。这个文件夹中是构建包里的所有源码文件，如图 5-4 所示。

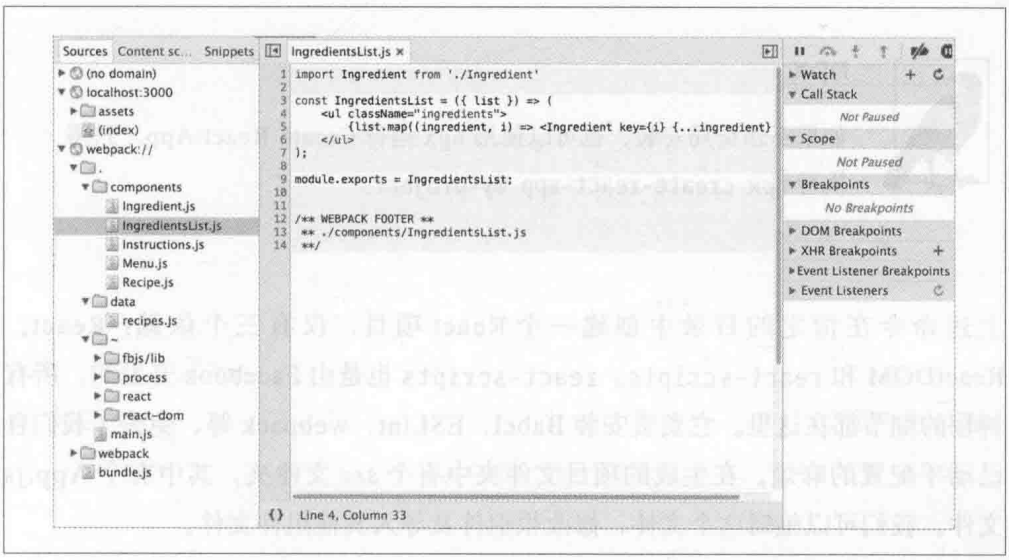

图 5-4：Chrome 开发者工具中的“Sources”面板

你可以使用浏览器中的单步调试器调试这些文件。任意点击一个行号即可添加一个断点。刷新浏览器，JavaScript 将在运行到源码文件中的断点处暂停。你可以在“Scope”面板中查看作用域中的变量，或者在“Watch”面板中添加要监视的变量。

5.5.4 Create React App

作为 React 开发者，还有一个优秀工具供我们使用，即 Create React App。这是一个命令行工具，能自动生成 React 项目。Create React App 受 Ember CLI 项目（*https://ember-cli.com*）启发，目的是帮助开发者快速创建 React 项目，不用再自己动手配置 webpack、Babel、ESLint 等相关的工具。

首先，全局安装 Create React App 包：

```
npm install -g create-react-app
```

然后，执行下述命令，指定用于存放项目的文件夹的名称：

```
create-react-app my-project
```



npx

如果不想全局安装，也可以使用 npx 运行 Create React App。只需执行 `npx create-react-app my-project`。

上述命令在指定的目录中创建一个 React 项目，仅有三个依赖：React、ReactDOM 和 `react-scripts`。`react-scripts` 也是由 Facebook 开发的，所有神秘的细节都在这里。它负责安装 Babel、ESLint、webpack 等，免去了我们自己动手配置的麻烦。在生成的项目文件夹中有个 *src* 文件夹，其中有个 App.js 文件。我们可以编辑这个文件，修改根组件及导入其他组件文件。

进入 *my-project* 文件夹，执行 `npm start`。如果愿意，也可以执行 `yarn start`。这两个命令在 3000 端口上启动应用。

你还可以执行 `npm test` 或 `yarn test` 命令运行测试。这两个命令在非交互模式下运行项目中的所有测试文件。

另外，也可以执行 `npm run build` 命令。使用 `yarn` 的话，执行 `yarn build` 命令。

这两个命令创建供生产环境使用的构建包，经过了转换和简化。

Create React App 对新手来说是个不错的工具，有经验的 React 开发者也可以使用。在发展的过程中，这个工具或许会增加更多功能，你可以到 GitHub 中关注一下变化情况（*https://github.com/facebook/create-react-app*）。学习 React 时如果不想自己定制 webpack 构建过程，还可以使用 CodeSandbox。这是一个在线 IDE，地址为 *https://codesandbox.io*。

本章学习了 JSX，React 技能更上层楼。我们创建了几个组件，又按照项目结构做了拆分，还深入学习了 Babel 和 webpack。现在可以更进一步学习组件了，是时候探讨 React Hooks 了。

第 6 章

React 状态管理

React 组件的运作离不开数据。前一章构建的菜谱用户界面，如果没有菜谱数组，没有任何用处。整个应用的价值就体现在菜谱、配料和清晰的操作说明上。用户界面是创作人员制作内容的工具。为了给内容创作人员构建出最好的工具，我们需要知道如何有效地操作和修改数据。

前一章构建了一个组件树（component tree），即具有一定层次结构的组件，数据以属性的形式在其中流动。属性只起到一半的作用，另一半靠状态（state）。React 应用的状态在数据的驱动下改变。在我们开发的菜谱应用中引入状态后，厨师便可以新建菜谱、修改现有菜谱和删除旧菜谱。

状态和属性之间有一定的关系。我们就根据这层关系把 React 应用中的不同组件紧密联系起来。当组件树的状态发生变化时，属性也随之改变。新数据沿着组件树流动，渲染特定枝叶上的组件，呈现新的内容。

本章引入状态，让应用鲜活起来。我们将学习如何创建有状态的组件、如何沿着组件树向下发送状态，以及如何沿着组件树向上反馈用户的交互。我们将学习从用户那里收集表单数据的方法，还将学习如何借助有状态的上下文供应组件，在应用中分离关注点。

6.1 构建一个星级评价组件

假如没有五星评价系统，我们肯定会吃到难吃的食物、看到难看的电影。如果你打算让用户参与到网站内容的建设中，那就需要一种方式来评定内容的优劣。这个时候，一个星级评价组件（StarRating）肯定是不可或缺的 React 组件（见图 6-1）。



图 6-1：StarRating 组件

StarRating 组件供用户为内容打分，给出心目中的星级数。较差的内容给一颗星，强烈推荐的内容获得五星。用户为内容打分时，直接点击某个星标即可。首先，我们需要一个星标。这个图标可以从 react-icons 中获取：

```
npm i react-icons
```

react-icons 是一个 npm 库，以 React 组件的形式分发，包含数百个 SVG 图标。安装这个库之后，我们便可以使用数个流行的图标库中的数百个 SVG 图标。这个库中包含的全部图标可以到它的网站中查看（*https://react-icons.netlify.com*）。我们将使用 Font Awesome 中的星标：

```
import React from "react";
import { FaStar } from "react-icons/fa";

export default function StarRating() {
  return [
    <FaStar color="red" />,
    <FaStar color="red" />,
    <FaStar color="red" />,
    <FaStar color="grey" />,
    <FaStar color="grey" />
```

```
    ];
  }
```

这里，我们创建了一个 StarRating 组件，渲染从 react-icons 中导入的五个 SVG 星标。前三个星标填充红色，后两个填充灰色。首先渲染星标，是为了让自己知道要构建的是什么。我们将把选中的星标填充为红色，而未选中的则填充为灰色。下面来创建一个组件，根据 selected 属性自动为星标填充颜色：

```
const Star = ({ selected = false }) => (
  <FaStar color={selected ? "red" : "grey"} />
);
```

Star 组件渲染单个星标，根据 selected 属性的值填充相应的颜色。如果没有把 selected 属性传给该组件，我们假定没有选中当前星标，填充默认的灰色。

五星评价系统十分流行，不过十星评价系统更为精准。在开发者把这个组件添加到自己的应用中时，我们应该让他们自行选择想使用多少颗星。为此，要在 StarRating 组件中添加一个 totalStars 属性：

```
const createArray = length => [...Array(length)];

export default function StarRating({ totalStars = 5 }) {
  return createArray(totalStars).map((n, i) => <Star key={i} />);
}
```

这里，我们添加了第 2 章定义的 createArray 函数。我们只需提供一个长度就能创建该长度的数组。我们使用这个函数和 totalStars 属性创建一个指定长度的数组，然后映射，渲染 Star 组件。默认情况下，totalStars 等于 5，即渲染 5 个灰色星标，如图 6-2 所示。



图 6-2：显示五个星标

6.2 useState 钩子

是时候把 StarRating 组件变成可点击的了，即让用户改变 rating 的值。由于 rating 是个可变化的值，我们将使用 React 状态存储和修改它的值。状态通过 React 的 Hooks（钩子）特性纳入函数组件。Hooks 是一组可重用的代码逻辑，放在组件树之外，用于接入组件的功能。React 自带多个钩子，可以直接使用。这里，我们想为 React 组件添加状态，要使用的 React 钩子是 useState。这个钩子在 react 包中，导入即可：

```
import React, { useState } from "react";
import { FaStar } from "react-icons/fa";
```

用户选中的星标数量表示评定的分值。我们将创建一个名为 selectedStars 的状态变量，存储用户评定的分值。我们直接在 StarRating 组件中使用 useState 钩子创建这个变量：

```
export default function StarRating({ totalStars = 5 }) {
  const [selectedStars] = useState(3);
  return (
    <>
      {createArray(totalStars).map((n, i) => (
        <Star key={i} selected={selectedStars > i} />
      ))}
      <p>
        {selectedStars} of {totalStars} stars
      </p>
    </>
  );
}
```

这就把组件与状态连接起来了。useState 钩子是一个函数，调用后返回一个数组。数组中的第一个值是我们想使用的状态变量。在这里，状态变量是 selectedStars，即 StarRating 组件填充红色星标的数量。既然 useState 返回一个数组，那我们就可以利用数组析构把状态变量命名为任何名称。我们传给 useState 函数的值是状态变量的默认值。这里，selectedStars 变量的初始值为 3，如图 6-3 所示。



图 6-3：选中五个星标中的三个

为了收集用户评定的分值，我们要让用户能点击各个星标。也就是说，我们要为 FaStar 组件添加一个 onClick 处理程序，把星标变成可以点击的。

```
const Star = ({ selected = false, onSelect = f => f }) => (
  <FaStar color={selected ? "red" : "grey"} onClick={onSelect} />
);
```

这里，我们为星标添加了一个 onSelect 属性。注意，这个属性是一个函数。用户点击 FaStar 组件时，我们可以调用这个函数，通知父级组件该星标被点击了。这个属性的默认值是 f => f。这是一个虚假函数，什么也不做，只是返回传入的参数。然而，倘若不设置一个默认函数，onSelect 属性就是未定义的，点击 FaStar 组件时将报错，因为 onSelect 的值必须时一个函数。虽然 f => f 什么也没做，但它仍是一个函数，调用时不会报错。不定义 onSelect 属性也没关系，React 会调用那个虚假的函数，什么也不发生。

现在，Star 组件可以点击了。我们将使用它改变 StarRating 的状态：

```
export default function StarRating({ totalStars = 5 }) {
  const [selectedStars, setSelectedStars] = useState(0);
  return (
    <>
      {createArray(totalStars).map((n, i) => (
        <Star
          key={i}
          selected={selectedStars > i}
          onSelect={() => setSelectedStars(i + 1)}
```

```
        />
      ))}
      <p>
        {selectedStars} of {totalStars} stars
      </p>
    </>
  );
}
```

为了改变 StarRating 组件的状态，我们需要一个用于修改 selectedStars 值的函数。useState 钩子返回的数组中的第二个元素是一个函数，可用于修改状态值。同样，通过析构数组，我们可以把那个函数命名为任何名称。这里，我们把它命名为 setSelectedStars，表明它的作用，即设定 selectedStars 的值。

使用钩子时有一件事一定要记住：钩子会导致所在的组件重新渲染。每次调用 setSelectedStars 函数修改 selectedStars 的值，useState 钩子都会调用 StarRating 函数组件，使用 selectedStars 的新值重新渲染。钩子的强大之处就体现在这里。钩子中的数据发生变化后，钩子会使用新数据重新渲染所在的组件。

用户每次点击 Star 组件，StarRating 组件都会重新渲染。用户点击某个 Star 组件时，调用该组件的 onSelect 属性。而调用 onSelect 属性时，又调用 setSelectedStars 函数，传入选中的星标数量。我们可以借助 map 函数的 i 变量计算数量。map 函数渲染第一个 Star 组件时，i 的值为 0。这意味着，我们要在 i 上加 1 才能得到正确的星标数量。调用 setSelectedStars 时，传给 StarRating 组件一个 selectedStars 值，如图 6-4 所示。

React 开发者工具会显示某个组件接入了哪些钩子。在浏览器中渲染 StarRating 组件时，可以在开发者工具中选中该组件，查看关于组件的调试信息。在右栏中可以看到，StarRating 组件接入了一个状态钩子，值为 2。在与应用交互的过程中，你会发现状态的值在变化，而组件树将使用选中的星标数量重新渲染。

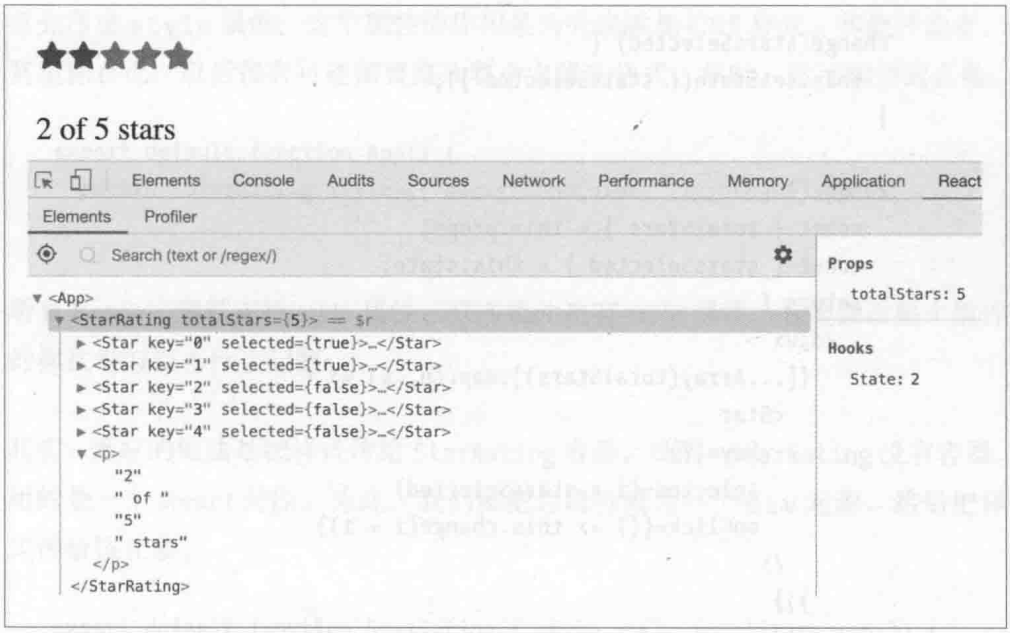

图 6-4：在 React 开发者工具中观察钩子

React 管理状态的“旧方式”

在 React 的旧版本中（v16.8.0 之前），为组件添加状态只能使用类组件。这不仅需要更多的句法，还不便于在组件之间重用功能。钩子的出现就是为了解决类组件存在的问题，提供了一种把功能接入函数组件的方案。

下述代码是一个类组件，这是本书第一版中的 StarRating 组件：

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

export default class StarRating extends Component {
  constructor(props) {
    super(props);
    this.state = {
      starsSelected: 0
    };
    this.change = this.change.bind(this);
  }
```

```
  change(starsSelected) {
    this.setState({ starsSelected });
  }

  render() {
    const { totalStars } = this.props;
    const { starsSelected } = this.state;
    return (
      <div>
        {[...Array(totalStars)].map((n, i) => (
          <Star
            key={i}
            selected={i < starsSelected}
            onClick={() => this.change(i + 1)}
          />
        ))}
        <p>
          {starsSelected} of {totalStars} stars
        </p>
      </div>
    );
  }
}
```

这个组件的作用与前文中的函数组件一样，不过代码量多了很多。另外，上述组件在使用 this 关键字和函数绑定上还容易让人摸不着头脑。

现在，这样的代码仍然可用。本书之所以没有涵盖类组件，是因为不再需要它了。函数组件和钩子是 React 的未来，我们不能开倒车。未来，类组件或许会正式退出历史舞台，这段代码也就不能继续使用了。

6.3 为提高可重用性而重构

现在，Star 组件可以放到线上使用了。只要想从用户那里收集评分，你的应用就可以使用这个组件。然而，如果你想把这个组件发布到 npm 中，供世界上的其他人使用，用于从用户那里收集评分，或许应该考虑再多处理几种情况。

首先考虑 style 属性。这个属性的作用是为元素添加 CSS 样式。其他开发者，甚至你自己，以后很有可能需要修改整个容器的样式。届时，你可能会这么做：

```
export default function App() {
  return <StarRating style={{ backgroundColor: "lightblue" }} />;
}
```

所有 React 元素都支持 style 属性，很多组件也有 style 属性。直接修改整个组件的样式看似没有什么问题。

其实，更好的做法是把样式传给 StarRating 容器。目前，StarRating 没有容器，用的是一个 React 片段。为此，我们要把片段升级为一个 div 元素，然后把样式传给该元素。

```
export default function StarRating({ style = {}, totalStars = 5 }) {
  const [selectedStars, setSelectedStars] = useState(0);
  return (
    <div style={{ padding: "5px", ...style }}>
      {createArray(totalStars).map((n, i) => (
        <Star
          key={i}
          selected={selectedStars > i}
          onSelect={() => setSelectedStars(i + 1)}
        />
      ))}
      <p>
        {selectedStars} of {totalStars} stars
      </p>
    </div>
  );
}
```

在上述代码中，我们把片段换成了一个 div 元素，再把样式应用到该元素上。我们为这个 div 元素设置了默认 5px 的内边距，然后使用展开运算符从 style 对象中获取余下的样式属性，应用到 div 的样式上。

此外，有些开发者可能会想为整个星级评价系统实现其他常用的属性：

```
export default function App() {
  return (
    <StarRating
      style={{ backgroundColor: "lightblue" }}
      onDoubleClick={e => alert("double click")}
    />
  );
}
```

在这个示例中，用户想为整个 StarRating 组件添加一个双击处理函数。如果觉得有必要，也可以把这个处理函数连同其他属性一起传给 div 容器：

```
export default function StarRating({ style = {}, totalStars = 5, ...props }) {
  const [selectedStars, setSelectedStars] = useState(0);
  return (
    <div style={{ padding: 5, ...style }} {...props}>
      ...
    </div>
  );
}
```

第一步是收集用户想传给 StarRating 组件的所有属性。我们通过展开运算符，即 ...props 收集这些属性。然后，把余下的所有属性传给 div 元素：{...props}。

这里做了两个假设。首先，假设用户只会添加这个 div 元素支持的属性。其次，假设用户无法为组件添加恶意属性。

这条规则不适用于所有组件。其实，在某些情况下最好只提供这种级别的支持。本节的要点是让你知道，应该为以后可能使用组件的用户考虑。

6.4 组件树中的状态

在每个组件中都使用状态不是个太好的主意。状态数据分散在多个组件中不便于追踪 bug 和修改应用，因为你很难确定状态值在组件树中的具体位置。如果集中在一处管理，你对应用的状态或某个功能的状态将有更好地掌控。这个思

想有很多具体做法，我们要分析的第一种做法是在状态树的根部存储状态，通过属性把状态传给子组件。

我们来编写一个小型应用，用于保存颜色列表。我们把这个应用命名为“Color Organizer”，用户在该应用中可以创建颜色列表，并为列表命名和评分。先来看示例数据集，如下所示：

```
[
  {
    "id": "0175d1f0-a8c6-41bf-8d02-df5734d829a4",
    "title": "ocean at dusk",
    "color": "#00c4e2",
    "rating": 5
  },
  {
    "id": "83c7ba2f-7392-4d7d-9e23-35adbe186046",
    "title": "lawn",
    "color": "#26ac56",
    "rating": 3
  },
  {
    "id": "a11e3995-b0bd-4d58-8c48-5e49ae7f7f23",
    "title": "bright red",
    "color": "#ff0000",
    "rating": 0
  }
]
```

这个 *color-data.json* 文件中有三个颜色数组，每个颜色都有 `id`、`title`、`color` 和 `rating`。首先，我们将创建 UI，使用几个 React 组件在浏览器中显示这些数据。然后，再实现添加新颜色、为颜色打分，以及从列表中删除颜色的功能。

6.4.1 沿组件树向下发送状态

这一步，我们把状态存储在 Color Organizer 应用的根组件（即 `App` 组件）中，然后把颜色向下传给子组件，负责渲染。在这个应用中，只有 `App` 组件持有状态。下面使用 `useState` 钩子把颜色列表添加到 `App` 组件中：

```
import React, { useState } from "react";
import colorData from "./color-data.json";
import ColorList from "./ColorList.js";

export default function App() {
  const [colors] = useState(colorData);
  return <ColorList colors={colors} />;
}
```

App 组件位于组件树的根部。在这个组件中添加 useState 钩子就接入了颜色的状态管理。在这个示例中，colorData 是前面给出的示例颜色数组。App 组件以 colorData 为 colors 的初始状态。然后，把 colors 向下传给名为 ColorList 的组件。

```
import React from "react";
import Color from "./Color";

export default function ColorList({ colors = [] }) {
  if(!colors.length) return <div>No Colors Listed.</div>;
  return (
    <div>
      {
        colors.map(color => <Color key={color.id} {...color} />)
      }
    </div>
  );
}
```

ColorList 通过属性从 App 组件那里获取颜色。如果颜色列表为空，这个组件显示一个消息，呈现给用户。颜色数组中有内容时映射数组，把各颜色的详细信息继续向下传给 Color 组件。

```
export default function Color({ title, color, rating }) {
  return (
    <section>
      <h1>{title}</h1>
      <div style={{ height: 50, backgroundColor: color }} />
      <StarRating selectedStars={rating} />
    </section>
```

```
  );
}
```

Color 组件接受三个属性：title、color 和 rating。这些属性的值在各个 color 对象中，已通过展开运算符 <Color {...color} /> 传给该组件。这样做，color 对象中的各个字段将使用与键相同的名称传给 Color 组件。Color 组件负责显示这些值。title 在一个 h1 元素中渲染。color 值显示为一个 div 元素的背景色（backgroundColor）。rating 继续沿着组件树向下传递，传给 StarRating 组件，以着色的星标突出显示。

```
export default function StarRating({ totalStars = 5, selectedStars = 0 }) {
  return (
    <>
      {createArray(totalStars).map((n, i) => (
        <Star
          key={i}
          selected={selectedStars > i}
        />
      ))}
      <p>
        {selectedStars} of {totalStars} stars
      </p>
    </>
  );
}
```

我们对 StarRating 组件做了一些修改，把它打造成了一个纯组件。纯组件指不含状态的函数组件，而且对给定的属性始终渲染得到相同的用户界面。这里之所以能把 StarRating 打造成纯组件，是因为颜色分值状态存储在组件树根部的 colors 数组中。还记得吗？这一步的目标是把状态集中存储在一个位置，而不是分散在组件树中的不同组件内。



StarRating 组件仍然可以自己持有状态，通过属性从父级组件中获取状态。如果是分发组件供社区中更多的人使用就要这么做。下一章介绍 useEffect 钩子时将演示具体做法。

至此，我们从 App 组件开始一路向下，沿着组件树把状态一直传到 Star 组件，以红色填充的视觉效果展示每个颜色的评分。如果使用前面给出的 color-data.json 文件渲染这个应用，在浏览器中将看到各个颜色，如图 6-5 所示。

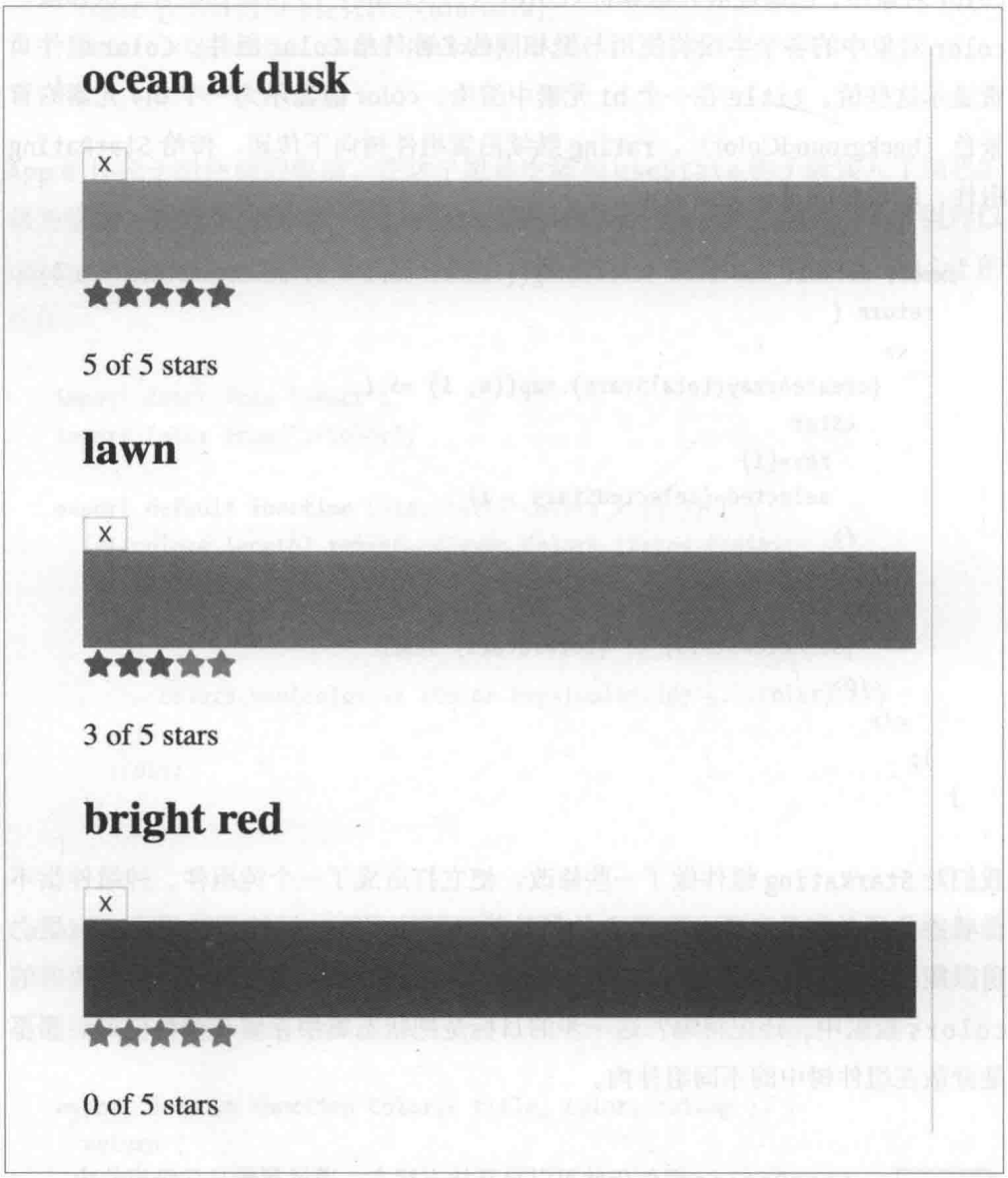

图 6-5：在浏览器中渲染 Color Organizer 应用

6.4.2 沿组件树向上发送交互

目前，我们编写了一系列 React 组件，通过属性沿着组件树从父级组件向子组件传递数据，在 UI 中渲染了 `colors` 数组。那么，如果我们想从列表中删除一个颜色，或者修改某个颜色的评分，该怎么办呢？`colors` 存储在组件树根部的状态中。我们要收集子组件的交互，沿组件树向上发送给根组件，在那里修改状态。

比如说我们想在各个颜色的标题旁边添加一个“Remove”按钮，让用户从状态中删除颜色。这个按钮应该添加到 `Color` 组件中：

```
import { FaTrash } from "react-icons/fa";

export default function Color({ id, title, color, rating, onRemove = f => f }) {
  return (
    <section>
      <h1>{title}</h1>
      <button onClick={() => onRemove(id)}>
        <FaTrash />
      </button>
      <div style={{ height: 50, backgroundColor: color }} />
      <StarRating selectedStars={rating} />
    </section>
  );
}
```

这里，我们修改 Color 组件，添加了一个按钮，供用户删除颜色。首先，从 `react-icons` 中导入一个垃圾桶图标。然后，把 `FaTrash` 图标放在一个按钮中，再为这个按钮添加一个 `onClick` 处理函数，调用随 `id` 等一起传入的属性列表中的 `onRemove` 函数。用户点击“Remove”按钮后，调用 `onRemove`，传入想删除的颜色的 id。鉴于此，我们才从 `Color` 组件的属性中获取了 `id` 值。

这个解决方法很好，因为保证了 `Color` 组件的纯粹性。`Color` 组件没有状态，可以轻易在应用中的其他部分，甚至是其他应用中重用。`Color` 组件不关心用户点击“Remove”按钮后会发生什么，它只负责通知父级组件发生了这个事件，并传递用户想删除的颜色的有关信息。这个事件的处理责任在父级组件身上：

```
export default function ColorList({ colors = [], onRemoveColor = f => f }) {
  if (!colors.length) return <div>No Colors Listed. (Add a Color)</div>;

return (
    colors.map(color => (
        <Color key={color.id} {...color} onRemove={onRemoveColor} />
      )
    }
    </div>
  );
}
```

Color 组件的父级组件是 ColorList，该组件也没有状态，它不真正删除颜色，而是把事件向上发给父级组件。为此，要在属性列表中添加 onRemoveColor 函数。如果 Color 组件调用了 onRemove 属性，ColorList 则调用 onRemoveColor 属性，把删除颜色的职责向上发给父级组件。颜色的 id 也传给 onRemoveColor 函数了。

ColorList 组件的父级组件是 App，状态就是在这个组件中接入的。因此，我们要在这里捕获颜色的 id，从状态中删除对应的颜色。

```
export default function App() {
  const [colors, setColors] = useState(colorData);
  return (
    <ColorList
      colors={colors}
      onRemoveColor={id => {
        const newColors = colors.filter(color => color.id !== id);
        setColors(newColors);
      }}
    />
  );
}
```

首先，添加一个 setColors 变量。还记得吗？ useState 返回的数组中的第二个元素是一个函数，可用于修改状态。ColorList 触发 onRemoveColor 事件后，App 组件从参数中捕获要删除的颜色的 id，通过 id 筛选颜色列表，排除用户想删除的颜色。然后，修改状态，使用 setColors 函数把颜色数组换成筛选后的新数组。

修改 colors 数组的状态后，App 组件将重新渲染新的颜色列表。这些新颜色传给 ColorList 组件，重新渲染。ColorList 组件又通过 Color 组件渲染剩下的颜色（少一个），在 UI 中体现这次改动。

如果想为 App 组件的状态中存储的 colors 评分，要通过 onRate 事件重复上述过程。首先，从被点击的各个星标上收集新的评分，把分值传给 StarRating 的父级组件：

```
export default function StarRating({
  totalStars = 5,
  selectedStars = 0,
  onRate = f => f
}) {
  return (
    <>
      {createArray(totalStars).map((n, i) => (
        <Star
          key={i}
          selected={selectedStars > i}
          onSelect={() => onRate(i + 1)}
        />
      ))}
    </>
  );
}
```

然后，从添加到 StarRating 组件中的 onRate 处理函数上获取评分，把新分值连同对应颜色的 id 一起，通过 onRate 函数属性传给 Color 组件的父组件。

```
export default function Color({
  id,
  title,
  color,
  rating,
  onRemove = f => f,
  onRate = f => f
}) {
  return (
```

```
      <section>
        <h1>{title}</h1>
        <button onClick={() => onRemove(id)}>
          <FaTrash />
        </button>
        <div style={{ height: 50, backgroundColor: color }} />
        <StarRating
          selectedStars={rating}
          onRate={rating => onRate(id, rating)}
        />
      </section>
    );
  }
```

在ColorList组件中，我们将从各个Color组件中捕获onRate事件，通过onRateColor函数属性向上传给父级组件。

```
  export default function ColorList({
    colors = [],
    onRemoveColor = f => f,
    onRateColor = f => f
  }) {
  if (!colors.length) return <div>No Colors Listed. (Add a Color)</div>;
    return (
      <div className="color-list">
        {
          colors.map(color => (
            <Color
              key={color.id}
              {...color}
              onRemove={onRemoveColor}
              onRate={onRateColor}
            />
          )
        }
      </div>
    );
  }
```

这个事件经过几个组件的传递之后，最终到达存储状态的App组件，新评分在这里保存：

```
export default function App() {
  const [colors, setColors] = useState(colorData);
  return (
    <ColorList
      colors={colors}
      onRateColor={(id, rating) => {
        const newColors = colors.map(color =>
          color.id === id ? { ...color, rating } : color
        );
        setColors(newColors);
      }}
      onRemoveColor={id => {
        const newColors = colors.filter(color => color.id !== id);
        setColors(newColors);
      }}
    />
  );
}
```

ColorList组件调用onRateColor属性，传入颜色的id和新分值时，App组件将更改颜色的评分。我们映射现有的颜色数组，修改与id属性匹配的颜色的评分，使用新值构建一个新颜色数组。然后，把newColors发给setColors函数，改变colors的状态值。此时，App组件将再次被调用，渲染colors数组中的新值。

colors数组的状态发生变化后，UI树将使用新数据重新渲染。新的评分通过红色星标呈现给用户。就像数据可以通过属性沿着组件树向下传递一样，交互可以通过函数属性连同数据一起沿着组件树向上传递。

6.5 构建表单

很多Web开发者的工作都涉及通过表单从用户手中收集大量信息。如果你也是这样，肯定会使用React构建大量表单。在DOM中可用的HTML表单元素全部都有对应的React元素，因此你或许已经知道如何使用JSX渲染表单了：

```
<form>
  <input type="text" placeholder="color title..." required />
  <input type="color" required />
  <button>ADD</button>
</form>
```

这个 form 元素有三个子元素：两个 input 元素和一个 button 元素。第一个 input 元素是一个文本输入框，用于从用户那里收集新颜色的 title 值。第二个 input 元素是一个 HTML 颜色输入框，供用户在一个色轮中拾取颜色。我们想使用基本的 HTML 表单验证，因此把两个输入框都标记为必须的（required）。“ADD”按钮用于添加新颜色。

6.5.1 使用 ref

在 React 中构建表单组件有好几种模式可用。在其中一个模式中，要使用 React 的一项特性直接访问 DOM 节点。这个特性称为 ref。在 React 中，ref 是一个对象，存储着一个组件整个生命期内的值。ref 适合在几个场景下使用。本节介绍如何使用 ref 直接访问 DOM 节点。

我们可以使用 React 提供的 useRef 钩子创建 *ref*。下面使用这个钩子构建 AddColorForm 组件。

```
import React, { useRef } from "react";

export default function AddColorForm({ onNewColor = f => f }) {
  const txtTitle = useRef();
  const hexColor = useRef();

  const submit = e => { ... }

  return (...)
}
```

首先，创建这个组件时，我们还使用 useRef 钩子创建了两个 ref。txtTitle ref 引用添加在表单中的文本输入框，用于收集颜色的标题。hexColor ref 用于访问

HTML 颜色输入框中的十六进制颜色值。这两个 ref 的值可以在 JSX 中直接使用 ref 属性设定：

```
    return (
      <form onSubmit={submit}>
        <input ref={txtTitle} type="text" placeholder="color title..." required />
        <input ref={hexColor} type="color" required />
        <button>ADD</button>
      </form>
    );
  }
```

这里，我们在 JSX 中的输入框元素上添加了 ref 属性，用以设定 txtTitle 和 hexColor 两个 ref 的值。如此，在 ref 对象上就创建了一个 current 字段，直接引用 DOM 元素。现在我们便可以访问 DOM 元素，捕获元素的值了。用户点击“ADD”按钮提交表单时，调用下述 submit 函数：

```
  const submit = e => {
    e.preventDefault();
    const title = txtTitle.current.value;
    const color = hexColor.current.value;
    onNewColor(title, color);
    txtTitle.current.value = "";
    hexColor.current.value = "";
  };
```

默认情况下，提交 HTML 表单后向当前 URL 发送 POST 请求，表单元素的值存储在请求主体中。我们不想用这种行为，因此 submit 函数的第一行代码是 e.preventDefault()，即阻止浏览器使用 POST 请求提交表单。

接下来，使用 ref 捕获各表单元素的当前值。然后，通过 onNewColor 函数属性把捕获的值传给该组件的父级组件。新颜色的标题和十六进制值都作为参数传给 onNewColor 函数。最后，重置两个输入框的 value 属性，清理数据，让表单做好收集另一个颜色的准备。

你注意到使用 ref 时微妙的范式转移（paradigm shift）了吗？我们直接修改

了 DOM 节点的 value 属性，把值设为空字符串 ""。这是命令式代码。现在，AddColorForm 变成了不受控组件（uncontrolled component），因为它使用 DOM 保存表单值。有时，使用不受控组件可以帮助我们解决棘手的问题。例如，你可能想让 React 之外的代码访问表单及其值。然而，使用受控组件（controlled component）是更好的方案。

6.5.2 受控组件

在受控组件中，表单值由 React 管理，而非 DOM。此时，无须使用 ref，不用编写命令式代码，添加可靠的表单验证等功能也要容易得多。下面来修改 AddColorForm，让该组件掌控表单的状态。

```
import React, { useState } from "react";

export default function AddColorForm({ onNewColor = f => f }) {
  const [title, setTitle] = useState("");
  const [color, setColor] = useState("#000000");

  const submit = e => { ... };

  return ( ... );
}
```

首先，我们不再使用 ref，而是通过 React 状态保存 title 和 color 的值。为此，要定义 title 和 color 变量，还要定义用于改变状态的函数：setTitle 和 setColor。

现在，组件掌控了 title 和 color 的值，接下来要设定 value 属性的值，在表单输入框元素中显示它们的值。一旦设定输入框元素的 value 属性，就不能通过表单修改了。此时，修改值的唯一方法是在用户输入每个字符时修改状态变量。具体做法如下。

```
<form onSubmit={submit}>
  <input
    value={title}
```

```
      onChange={event => setTitle(event.target.value)}
      type="text"
      placeholder="color title..."
      required
    />
    <input
      value={color}
      onChange={event => setColor(event.target.value)}
      type="color"
      required
    />
    <button>ADD</button>
  </form>
  )
```

现在，这个受控组件使用状态中的 title 和 color 设定两个 input 元素的值。只要这两个元素触发了 onChange 事件，我们就可以使用 event 参数访问新值。event.target 是对 DOM 元素的引用，因此我们可以使用 event.target.value 获取元素的当前值。title 发生变化时，调用 setTitle 修改状态中的标题值。标题值发生变化会导致该组件重新渲染，借此便可以在 input 元素中显示 title 的新值。修改颜色的过程与此完全一致。

提交表单时，只需把 title 和 color 的状态值作为参数传给 onNewColor 函数属性。把新颜色传给父级组件之后，可以使用 setTitle 和 setColor 函数重置状态值。

```
const submit = e => {
  e.preventDefault();
  onNewColor(title, color);
  setTitle("");
  setColor("");
};
```

这个组件之所以叫作受控组件，是因为 React 控制着表单的状态。值得注意的是，受控的表单组件经常重新渲染。想象一下，每在 title 字段中输入一个字符，AddColorForm 就会重新渲染。使用颜色拾取器中的色轮导致组件重新渲染的次数比 title 字段多得多，这是因为用户在色轮上拖拽鼠标时将不断改变颜色值。

这不算什么，React 禁受得住这种考验。希望知道受控组件会频繁重新渲染后，你能避免在受控组件中添加耗时、耗资源的操作。在优化 React 组件时，你至少要能合理运用这个知识。

6.5.3 自定义钩子

如果一个表单有大量 input 元素，你肯定会不断复制粘贴下面两行代码：

```
value={title}
onChange={event => setTitle(event.target.value)}
```

复制属性、粘贴到各个表单元素中，再调整变量名称，这样看似效率高，可是每次复制粘贴代码，你都会听到脑中有个声音在提醒你，复制粘贴表明是重复的，应该提取出来，定义为函数。

我们可以把创建受控表单组件的详细过程独立打包，自定义为一个钩子。我们可以自己定义一个 useInput 钩子，去除创建受控的表单输入框过程中涉及的重复。

```
import { useState } from "react";

export const useInput = initialValue => {
  const [value, setValue] = useState(initialValue);
  return [
    { value, onChange: e => setValue(e.target.value) },
    () => setValue(initialValue)
  ];
};
```

这是一个自定义钩子，实现代码并不多。在这个钩子内部，我们仍然使用 useState 钩子创建状态值。然后，返回一个数组。数组中的第一个值是一个对象，包含我们之前复制粘贴的属性：状态中的 value 和用于修改状态值的 onChange 函数属性。数组中的第二个值是一个函数，它的作用是把 value 重置为初始值。下面在 AddColorForm 中使用这个钩子。

```
import React from "react";
import { useInput } from "./hooks";

export default function AddColorForm({ onNewColor = f => f }) {
  const [titleProps, resetTitle] = useInput("");
  const [colorProps, resetColor] = useInput("#000000");

  const submit = event => { ... }

  return ( ... )
}
```

useState 钩子封装在我们自己定义的 useInput 钩子内。标题和颜色的属性可以通过析构返回数组的第一个值得到。返回数组的第二个值是一个函数，我们可以用它把 value 属性重置为初始值，即一个空字符串。下面展开 titleProps 和 colorProps，放入相应的输入框元素：

```
return (
  <form onSubmit={submit}>
    <input
      {...titleProps}
      type="text"
      placeholder="color title..."
      required
    />
    <input {...colorProps} type="color" required />
    <button>ADD</button>
  </form>
);
}
```

展开从自定义的钩子中获取的属性比直接粘贴属性有趣多了。现在，标题和颜色输入框都能接收属性的值和 onChange 事件。我们使用自定义的钩子创建了受控的表单输入框，无须担心底层实现细节。此外，还有一处需要修改，即提交表单的方式：

```
const submit = event => {
  event.preventDefault();
```

```
    onNewColor(titleProps.value, colorProps.value);
    resetTitle();
    resetColor();
  };
```

在 submit 函数中，我们要从标题和颜色的属性中获取二者的 value。最后，使用 useInput 钩子返回的函数重置值。

钩子应在 React 组件内部使用。在自定义的钩子内可以使用其他钩子，毕竟钩子最终要在组件中使用。在我们自定义的这个钩子中，如果状态发生了变化，仍会导致 AddColorForm 重新渲染，呈现 titleProps 或 colorProps 的新值。

6.5.4 把颜色添加到状态中

不管是受控的表单组件还是不受控的表单组件，title 和 color 的值都通过 onNewColor 函数传给父级组件。父级组件才不管我们使用受控组件还是不受控组件，只要能得到新颜色的值即可。

下面我们把 AddColorForm（不管使用哪种实现）添加到 App 组件中。调用 onNewColor 属性时，把新颜色保存到状态中。

```
import React, { useState } from "react";
import colorData from "./color-data.json";
import ColorList from "./ColorList.js";
import AddColorForm from "./AddColorForm";
import { v4 } from "uuid";

export default function App() {
  const [colors, setColors] = useState(colorData);
  return (
    <>
      <AddColorForm
        onNewColor={(title, color) => {
          const newColors = [
            ...colors,
            {
              id: v4(),
```

```
            rating: 0,
            title,
            color
          }
        ];
        setColors(newColors);
      }}
    />
    <ColorList .../>
  </>
 );
}
```

添加新颜色时，调用 onNewColor 属性。新颜色的 title 和十六进制值 color 作为参数传给该函数，使用这两个参数创建新颜色数组。首先，在新数组中展开状态中现有的 colors 数组。然后，根据 title 和 color 的值添加一个全新的颜色对象。我们把新颜色的 rating 设为 0，因为还没有人打分。此外，我们使用 uuid 包中的 v4 函数为新颜色生成一个唯一的 id 值。得到包含新颜色的颜色数组之后，调用 setColors 把数组保存到状态中。此时，App 组件将使用新的 colors 数组重新渲染。UI 得到更新，呈现新数组的内容。新颜色将出现在列表的底部。

经过上述修改之后，Color Organizer 应用的第一次迭代完成了。现在，用户可以把新颜色添加到列表中，可以从列表中删除颜色，还可以为列表中的颜色评分。

6.6 React 上下文

在 React 的早期版本中，把状态集中放在组件树的根部是一个重要的模式，让我们受益良多。作为 React 开发者，我们都应该掌握如何通过属性在组件树中向上和向下传递状态。然而，随着 React 的发展，以及组件树枝繁叶茂，遵守这个原则慢慢变得不切实际。对很多开发者来说，在复杂的应用中集中于组件树的根部维护状态不是一件容易的事。在众多组件之间向上和向下传递状态，操作繁琐，而且容易出错。

我们处理的 UI 元素大都复杂。组件树的根部离枝叶往往较远。这导致应用依赖的数据离使用数据的组件相距甚远。每个组件都要接收只传给子组件的属性，写出的代码臃肿不堪，UI 难以扩充。

状态数据通过属性经由一个个组件传递，一直传到需要使用状态的组件，这就好像从旧金山坐火车到华盛顿特区，途径很多州，到终点站才下车（见图 6-6）。

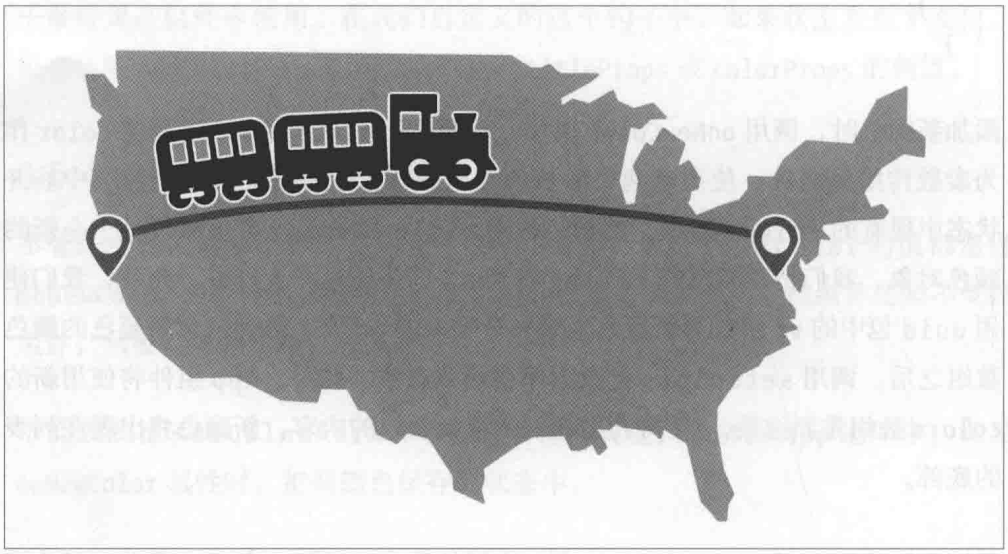

图 6-6：从旧金山坐火车到华盛顿特区

显然，从旧金山乘飞机到华盛顿特区更便捷。这样不用经停各州，而是直接飞过（见图 6-7）。

在 React 中，上下文（context）就像是数据的飞行装备。为了把数据放入 React 上下文，我们要创建上下文供应组件（context provider）。这是一种 React 组件，可以包含整个组件树，也可以包含组件树的特定部分。上下文供应组件是始发港，数据在这里登机。上下文供应组件也是枢纽，所有航班都从这里离港，飞往不同的目的地。各个目的地是上下文消费组件（context consumer），即从上下文中获取数据的 React 组件。上下文消费组件是到达港，数据在这里降落、下机，投入到工作中。

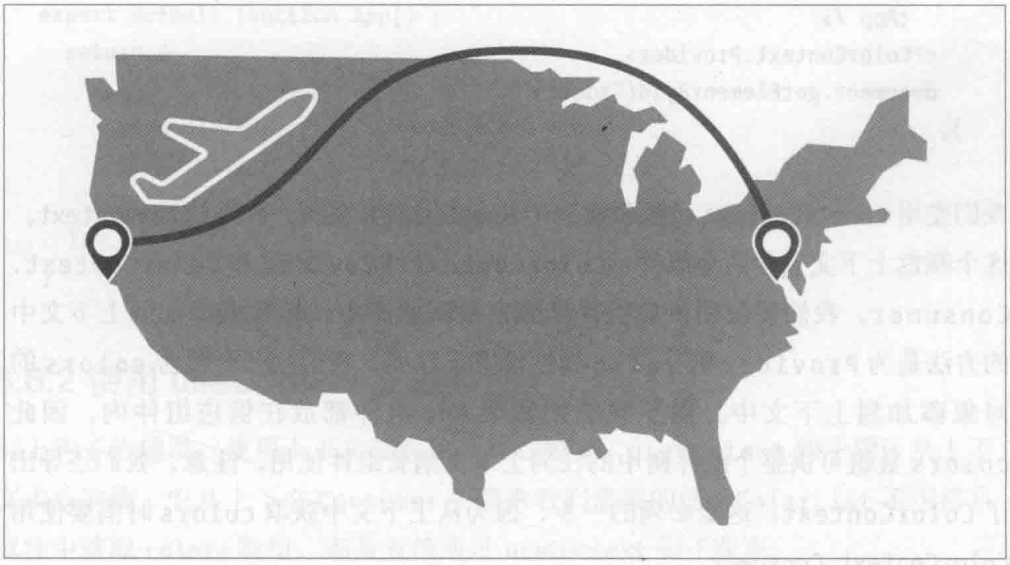

图 6-7：从旧金山乘飞机到华盛顿特区

使用上下文不妨碍我们集中在一处存储状态数据，只是不用再经过一堆用不到状态的组件传递数据了。

6.6.1 把颜色放入上下文

在 React 中使用上下文，首先要把数据放入上下文供应组件，并把供应组件添加到组件树中。我们使用 React 提供的 `createContext` 函数创建上下文对象。这个对象包含两个组件：一个上下文 `Provider` 和一个 `Consumer`。

下面我们把 *color-data.json* 文件中的默认颜色放入上下文。我们将在应用的入口文件 *index.js* 中添加上下文。

```
import React, { createContext } from "react";
import colors from "./color-data";
import { render } from "react-dom";
import App from "./App";

export const ColorContext = createContext();

render(
  <ColorContext.Provider value={{ colors }}>
```

```
      <App />
    </ColorContext.Provider>,
    document.getElementById("root")
  );
```

我们使用 createContext 函数创建一个 React 上下文实例，名为 ColorContext。这个颜色上下文包含两个组件：ColorContext.Provider 和 ColorContext.Consumer。我们要使用供应组件把颜色放到状态中。把数据添加到上下文中的方法是为 Provider 的 value 属性设值。这里，我们把一个包含 colors 的对象添加到上下文中。由于我们把整个 App 组件都放在供应组件内，因此 colors 数组可供整个组件树中的任何上下文消费组件使用。注意，我们还导出了 ColorContext。这是必须的一步，因为从上下文中获取 colors 时需要使用 ColorContext.Consumer。



上下文供应组件不是总要包含整个应用。有时也会包含部分特定的组件，这样效率更高。供应组件只为所含的子组件提供上下文值。

一个应用中可以有多个上下文供应组件。其实，你可能已经在不知情的情况下在自己的 React 应用中使用了上下文供应组件。很多支持 React 的 npm 包在背后就使用上下文。

现在，我们在上下文中提供了 colors 值，App 组件无须再持有状态，并通过属性向下传给子组件了。我们把 App 组件变成了“天桥”。Provider 是 App 组件的父级组件，在上下文中提供了 colors。ColorList 是 App 组件的子组件，可以直接获取 colors。如此一来，App 组件彻底不用接触颜色了。这很好，毕竟 App 组件本身也用不到颜色。处理颜色的责任委托给组件树中的下级组件了。

App 组件的很多行代码都可以删除。现在，该组件只需要渲染 AddColorForm 和 ColorList，不用再关心数据：

```
import React from "react";
import ColorList from "./ColorList.js";
import AddColorForm from "./AddColorForm";
```

```
export default function App() {
  return (
    <>
      <AddColorForm />
      <ColorList />
    </>
  );
}
```

6.6.2 使用 useContext 获取颜色

加上钩子的辅助，使用上下文简直让人身心愉悦。useContext 钩子用于从上下文中获取值，它从上下文 Consumer 中获取我们需要的值。ColorList 不用再从属性中获取 colors 数组，而是直接通过 useContext 钩子获取。

```
import React, { useContext } from "react";
import { ColorContext } from "./";
import Color from "./Color";

export default function ColorList() {
  const { colors } = useContext(ColorContext);
  if (!colors.length) return <div>No Colors Listed. (Add a Color)</div>;
  return (
    <div className="color-list">
      {
        colors.map(color => <Color key={color.id} {...color} />)
      }
    </div>
  );
}
```

这里，我们修改 ColorList 组件，删掉了 colors=[] 属性，因为 colors 将从上下文中获取。为了从上下文中获取值，useContext 钩子要用到上下文实例。为此，我们从创建上下文及把供应组件添加到组件树中的 *index.js* 文件里导入了 ColorContext 实例。现在，ColorList 可以根据上下文提供的数据构建用户界面了。



使用上下文消费组件

Consumer 用 useContext 钩子访问，这意味着我们不用直接处理消费组件了。在钩子出现之前，若想从上下文中获取颜色，要在上下文消费组件中使用一种称为“渲染属性”（render props）的模式。渲染属性作为参数传给一个子函数。下述代码示例展示了如何使用消费组件从上下文中获取颜色。

```
export default function ColorList() {
  return (
    <ColorContext.Consumer>
      {context => {
      if (!context.colors.length)
      return <div>No Colors Listed. (Add a Color)</div>;
        return (
          <div className="color-list">
            {
              context.colors.map(color =>
  <Color key={color.id} {...color} />)
            }
          </div>
        )
      }}
    </ColorContext.Consumer>
  )
}
```

6.6.3 有状态的上下文供应组件

上下文供应组件可以把对象放入上下文，但是它自己无法修改上下文中的值，需要父级组件的协助。具体做法是创建一个有状态的组件，让它渲染上下文供应组件。当有状态的组件的状态发生变化时，将使用新的上下文数据重新渲染上下文供应组件。上下文供应组件的所有子组件也将使用新的上下文数据重新渲染。

渲染上下文供应组件的有状态的组件是一个自定义的供应组件。我们将把 App 放在这个供应组件中。新建一个文件，创建一个 ColorProvider。

```
import React, { createContext, useState } from "react";
import colorData from "./color-data.json";

const ColorContext = createContext();

export default function ColorProvider ({ children }) {
  const [colors, setColors] = useState(colorData);
  return (
    <ColorContext.Provider value={{ colors, setColors }}>
      {children}
    </ColorContext.Provider>
  );
};
```

ColorProvider 组件负责渲染 ColorContext.Provider。在这个组件中，我们使用 useState 钩子创建了状态变量 colors。colors 的初始数据仍然从 *color-data.json* 文件中读取。接下来，通过 ColorContext.Provider 的 value 属性把状态中的 colors 添加到上下文中。ColorProvider 的所有子组件都将放入 ColorContext.Provider 中，因此可以从上下文中访问 colors 数组。

你可能注意到了，我们也把 setColors 函数添加到上下文中了。这样，上下文消费组件便可以修改颜色数组的值。只要调用 setColors，colors 数组就会发生变化，从而导致 ColorProvider 重新渲染，在 UI 中显示 colors 数组的新内容。

把 setColors 添加到上下文中或许不是个好主意。使用时，你和其他开发者都可能会犯错。修改 colors 数组的值只有三种可能的方式：用户添加颜色、删除颜色或为颜色打分。因此，更好的方法是在上下文中为每一种操作添加一个函数。这样就无须把 setColors 函数开放给消费组件，而是只开放允许用户执行修改操作的函数。

```
export default function ColorProvider ({ children }) {
  const [colors, setColors] = useState(colorData);

  const addColor = (title, color) =>
    setColors([
      ...colors,
      {
```

```
        id: v4(),
        rating: 0,
        title,
        color
      }
    ]);

  const rateColor = (id, rating) =>
    setColors(
      colors.map(color => (color.id === id ? { ...color, rating } : color))
    );

  const removeColor = id => setColors(colors.filter(color => color.id !== id));

  return (
    <ColorContext.Provider value={{ colors, addColor, removeColor, rateColor }}>
      {children}
    </ColorContext.Provider>
  );
};
```

这样做更好。我们在上下文中为允许对颜色数组执行的每一项操作都添加了一个函数。这样一来，组件树中的任何组件都可以使用这些目的明确的简单函数修改颜色数组了。

6.6.4 使用上下文自定义钩子

还有一个重要的地方需要修改。引入钩子之后，我们完全不用把上下文开放给消费组件。不得不承认，没读过本书的团队成员看到上下文会摸不着头脑的。为了减轻团队成员的负担，我们可以把上下文包装到一个自定义的钩子中。我们不再对外开放 ColorContext 实例，而是创建一个名为 useColors 的钩子，从上下文中返回颜色。

```
import React, { createContext, useState, useContext } from "react";
import colorData from "./color-data.json";
import { v4 } from "uuid";
```

```
const ColorContext = createContext();
export const useColors = () => useContext(ColorContext);
```

这个简单的改动对架构有巨大的影响。我们把渲染和处理有状态的颜色所需的全部功能都打包到一个 JavaScript 模块中，上下文隐藏在这个模块中，通过一个钩子对外开放。之所以可以这样做，是因为我们使用 `useContext` 钩子返回了上下文，让外部代码可以访问这个文件中的 `ColorContext`。现在，我们可以把这个模块重命名为 `color-hooks.js`，分发出去，供社区中更多的人使用。

通过 `ColorProvider` 和 `useColors` 钩子处理颜色是一件让人身心愉悦的事。这正是我们追求的编程体验。下面就在 Color Organizer 应用中使用这个钩子。首先，要把 App 组件放在我们自定义的 `ColorProvider` 中。这一步在 *index.js* 文件中操作。

```
import React from "react";
import { ColorProvider } from "./color-hooks.js";
import { render } from "react-dom";
import App from "./App";

render(
  <ColorProvider>
    <App />
  </ColorProvider>,
  document.getElementById("root")
);
```

显然，App 的子组件都可以通过 `useColors` 钩子获取 `colors`。`ColorList` 组件要访问 colors 数组，在屏幕上渲染颜色：

```
import React from "react";
import Color from "./Color";
import { useColors } from "./color-hooks";

export default function ColorList() {
  const { colors } = useColors();
  return ( ... );
}
```

我们删掉了这个组件中对上下文的引用。现在，所需的一切都由钩子提供。Color 组件可以使用这个钩子直接获取为颜色打分和删除颜色的函数：

```
import React from "react";
import StarRating from "./StarRating";
import { useColors } from "./color-hooks";

export default function Color({ id, title, color, rating }) {
  const { rateColor, removeColor } = useColors();
  return (
    <section>
      <h1>{title}</h1>
      <button onClick={() => removeColor(id)}>X</button>
      <div style={{ height: 50, backgroundColor: color }} />
      <StarRating
        selectedStars={rating}
        onRate={rating => rateColor(id, rating)}
      />
    </section>
  );
}
```

现在，Color 组件不用再通过函数属性把事件传给父级组件，它能访问上下文中的 rateColor 和 removeColor 函数。这两个函数通过 useColors 钩子就能轻易获取。这样做是舒畅了不少，不过还没结束。AddColorForm 也能从 useColors 钩子上获益。

```
import React from "react";
import { useInput } from "./hooks";
import { useColors } from "./color-hooks";

export default function AddColorForm() {
  const [titleProps, resetTitle] = useInput("");
  const [colorProps, resetColor] = useInput("#000000");
  const { addColor } = useColors();

  const submit = e => {
    e.preventDefault();
    addColor(titleProps.value, colorProps.value);
```

```
    resetTitle();
    resetColor();
  };

  return ( ... );
}
```

AddColorForm组件可以直接使用addColor函数添加颜色。添加、评分或删除颜色后，上下文中colors值的状态会发生变化。状态发生变化后，ColorProvider的子组件会重新渲染新的上下文数据。这一切都通过一个简单的钩子实现。

钩子为软件开发人员提供了保持动力和享受前端编程所需的精神激励。这主要是因为钩子能很好地分离关注点。现在，React组件只需关注自身如何渲染其他React组件，并保持用户界面显示最新的数据。React钩子关注的是让应用正常运行所需的逻辑。UI和钩子都可以单独开发、测试，甚至单独部署。这对React来说是特别好的消息。

本章只接触了钩子的皮毛，钩子能做的事情还有很多，下一章深入探讨。

第 7 章

使用钩子增强组件

渲染是 React 应用的中心。属性和状态发生变化后，组件树重新渲染，在用户界面中呈现最新的数据。目前，在描述如何渲染组件时，我们使用的都是 useState。其实，我们能做的事情还有很多。我们可以使用其他钩子定义规则，指明为何及何时渲染；我们还可以使用其他钩子提升渲染性能，能助我们一臂之力的钩子还有很多。

前一章介绍了 useState、useRef 和 useContext，还学习了如何使用这些钩子自定义钩子：useInput 和 useColors。除此之外，可用的钩子还有很多，React 就自带了很多钩子。本章将深入探讨 useEffect、useLayoutEffect 和 useReducer。这三个钩子在构建应用时都有重要用途。另外，本章还将介绍 useCallback 和 useMemo，这两个钩子可以优化组件的性能。

7.1 useEffect 介绍

现在，我们基本知道渲染组件是怎么回事了。组件就是负责渲染用户界面的函数。首次加载应用，以及属性和状态值发生变化之时进行渲染。那么，如果我们想在渲染之后做些事情该怎么办呢？下面来分析一下。

以下面这个简单的 Checkbox 组件为例。我们使用 useState 设定 checked 值，使用 setChecked 函数改变 checked 的值。用户可以勾选或取消勾选复选框，但要是想在勾选之后通知用户呢？下面用 alert 试一下，这是阻塞线程的一种好方式。

```
import React, { useState } from "react";

function Checkbox() {
  const [checked, setChecked] = useState(false);

  alert(`checked: ${checked.toString()}`);

  return (
    <>
      <input
        type="checkbox"
        value={checked}
        onChange={() => setChecked(checked => !checked)}
      />
      {checked ? "checked" : "not checked"}
    </>
  );
};
```

我们把 alert 添加到渲染操作之前，阻塞渲染。在用户单击弹出框上的“确定”按钮之前，这个组件不会渲染。由于 alert 阻塞了线程，在单击“确认”按钮之前，复选框的下一个状态不会重新渲染。

这可不是我们想要的效果，那么把 alert 放在 return 语句之后呢？

```
function Checkbox {
  const [checked, setChecked] = useState(false);

  return (
    <>
      <input
        type="checkbox"
        value={checked}
        onChange={() => setChecked(checked => !checked)}
```

```
        />
        {checked ? "checked" : "not checked"}
      </>
    );

    alert(`checked: ${checked.toString()}`);
  };
```

算了吧，alert 不能放在渲染操作之后，因为代码根本执行不到那一行。为了确保能正常看到弹出框，可以使用 useEffect。把 alert 放在 useEffect 函数中，渲染之后就会被调用了，这是一个副作用。

```
function Checkbox {
  const [checked, setChecked] = useState(false);

  useEffect(() => {
    alert(`checked: ${checked.toString()}`);
  });

  return (
    <>
      <input
        type="checkbox"
        value={checked}
        onChange={() => setChecked(checked => !checked)}
      />
      {checked ? "checked" : "not checked"}
    </>
  );
};
```

想让渲染产生副作用就使用 useEffect。副作用可以理解为函数在返回之外所做的其他事情。这里的函数是 Checkbox。Checkbox 函数负责渲染 UI。可是，除此之外，我们还想让这个组件做别的事情。想让组件在返回 UI 之外去做的其他事情就叫作“效应”（useEffect 中的“effect”）。

alert、console.log，或者与浏览器或原生 API 的交互都不能作为渲染操作的一部分，不能写在 return 语句中。然而，在 React 应用中，渲染会影响这些事

件的结果。useEffect 的作用是等待渲染结束，把值提供给 alert 或 console.log。

```
useEffect(() => {
  console.log(checked ? "Yes, checked" : "No, not checked");
});
```

类似地，我们可以获取渲染后 checked 的值，把它保存到 localStorage 中：

```
useEffect(() => {
  localStorage.setItem("checkbox-value", checked);
});
```

我们还可以使用 useEffect 让添加到 DOM 中的某个文本输入框获得焦点。React 先渲染输出，再调用 useEffect，让元素获得焦点：

```
useEffect(() => {
  txtInputRef.current.focus();
});
```

渲染之后，txtInputRef 就有值了，因此在这段代码中可以访问它的值，让元素获得焦点。每次渲染之后，useEffect 都能访问属性、状态、ref 等的最新值。

useEffect 相当于是在渲染之后调用的一个函数。渲染之后，我们便可以访问组件中当前的状态值，用这些值做其他事情。如果重新渲染了，一切都重来一遍。新值，新渲染，新效应。

7.1.1 依赖数组

根据设计，useEffect 应与其他有状态的钩子结合起来使用，例如 useState 和尚未提及的 useReducer（本章后文介绍）。React 在状态发生变化时重新渲染组件树。前面学过 useEffect 在渲染之后调用。

请看下面的示例，App 组件有两个独立的状态值：

```
import React, { useState, useEffect } from "react";
import "./App.css";

function App() {
  const [val, set] = useState("");
  const [phrase, setPhrase] = useState("example phrase");

  const createPhrase = () => {
    setPhrase(val);
    set("");
  };

  useEffect(() => {
    console.log(`typing "${val}"`);
  });

  useEffect(() => {
    console.log(`saved phrase: "${phrase}"`);
  });

  return (
    <>
      <label>Favorite phrase:</label>
      <input
        value={val}
        placeholder={phrase}
        onChange={e => set(e.target.value)}
      />
      <button onClick={createPhrase}>send</button>
    </>
  );
}
```

val 是一个状态变量，表示输入字段的值。输入字段的值发生变化，val 随之改变。用户每输入一个字符，都会导致组件重新渲染。用户单击“Send”按钮后，文本框的值将保存为 phrase，而 val 则被重置为 ""，即清空文本字段。

这是预期的行为，然而 useEffect 钩子被调用的次数比预期要多。每次渲染，两个 useEffect 钩子都会被调用。

```
typing ""                          // 第一次渲染
saved phrase: "example phrase"     // 第一次渲染
typing "S"                         // 第二次渲染
saved phrase: "example phrase"     // 第二次渲染
typing "Sh"                        // 第三次渲染
saved phrase: "example phrase"     // 第三次渲染
typing "Shr"                       // 第四次渲染
saved phrase: "example phrase"     // 第四次渲染
typing "Shre"                      // 第五次渲染
saved phrase: "example phrase"     // 第五次渲染
typing "Shred"                     // 第六次渲染
saved phrase: "example phrase"     // 第六次渲染
```

我们不想每次渲染都产生效应。我们要把 `useEffect` 钩子与特定的数据变化关联起来。为了解决这个问题，我们可以使用依赖数组（dependency array）。依赖数组可以控制在什么时候调用 `useEffect`。

```
useEffect(() => {
  console.log(`typing "${val}"`);
}, [val]);

useEffect(() => {
  console.log(`saved phrase: "${phrase}"`);
}, [phrase]);
```

我们在两个效应中都添加了依赖数组，控制各自在何时调用。第一个效应只在 `val` 的值发生变化时调用，第二个效应只在 `phrase` 的值发生变化时调用。现在运行应用，控制台中更新的内容更符合预期。

```
typing ""                          // 第一次渲染
saved phrase: "example phrase"     // 第一次渲染
typing "S"                         // 第二次渲染
typing "Sh"                        // 第三次渲染
typing "Shr"                       // 第四次渲染
typing "Shre"                      // 第五次渲染
typing "Shred"                     // 第六次渲染
typing ""                          // 第七次渲染
saved phrase: "Shred"              // 第七次渲染
```

在输入框中输入内容，改变的是 val 的值，只会触发第一个效应。点击按钮后才把值保存到 phrase 中，并把 val 重置为 ""。

依赖数组是一个数组，因此可以检查多个值。假如我们想在 val 或 phrase 发生变化时运行一个效应：

```
useEffect(() => {
  console.log("either val or phrase has changed");
}, [val, phrase]);
```

如果两个值中有一个发生了变化，就会调用这个效应。useEffect 函数的第二个参数可以是一个空数组，此时只在首次渲染后调用效应。

```
useEffect(() => {
  console.log("only once after initial render");
}, []);
```

由于数组中没有依赖，因此这个效应只在首次渲染时调用。没有依赖意味着没有变化，所以这个效应以后都不会再调用。只在首次渲染时调用的效应特别适合用于初始化：

```
useEffect(() => {
  welcomeChime.play();
}, []);
```

如果效应返回一个函数，该函数将在把组件从组件树上移除时调用：

```
useEffect(() => {
  welcomeChime.play();
  return () => goodbyeChime.play();
}, []);
```

这意味着，我们可以使用 useEffect 做事前设置和事后清理。我们提供的是一个空数组，因此欢迎乐只在首次渲染时播放。然后，返回一个函数，做清理工作，在从组件树上删除组件时播放送别乐。

这个模式在很多情况下都有用。或许我们想在首次渲染时订阅一个新闻源，然后通过清理函数退订。具体而言，首先创建状态值 posts 和用于修改值的 setPosts 函数。然后，定义 addPosts 函数，把最新文章添加到数组中。接下来，使用 useEffect 订阅新闻源，并播放欢迎乐。另外，我们可以返回一个清理函数，退订并播放送别乐。

```
const [posts, setPosts] = useState([]);
const addPost = post => setPosts(allPosts => [post, ...allPosts]);

useEffect(() => {
  newsFeed.subscribe(addPost);
  welcomeChime.play();
  return () => {
    newsFeed.unsubscribe(addPost);
    goodbyeChime.play();
  };
}, []);
```

可是，这个 useEffect 做的事情太多了。或许可以使用一个 useEffect 处理新闻源事件，使用另一个 useEffect 处理音乐事件：

```
useEffect(() => {
  newsFeed.subscribe(addPost);
  return () => newsFeed.unsubscribe(addPost);
}, []);

useEffect(() => {
  welcomeChime.play();
  return () => goodbyeChime.play();
}, []);
```

把功能分散到多个 useEffect 调用中通常是不错的主意。不过，我们再改进一下吧。我们想实现的功能是订阅新闻源，并在订阅、退订和有新文章时播放不同的爵士乐。每个人都喜欢音效，不是吗？那就自定义一个钩子吧。或许我们可以把这个钩子命名为 useJazzyNews：

```
const useJazzyNews = () => {
```

```
  const [posts, setPosts] = useState([]);
  const addPost = post => setPosts(allPosts => [post, ...allPosts]);

  useEffect(() => {
    newsFeed.subscribe(addPost);
    return () => newsFeed.unsubscribe(addPost);
  }, []);

  useEffect(() => {
    welcomeChime.play();
    return () => goodbyeChime.play();
  }, []);

  return posts;
};
```

我们自定义的钩子包含处理爵士乐新闻源的全部功能，可以轻易地在组件之间共享。下面在一个新的 NewsFeed 组件中使用这个自定义的钩子：

```
function NewsFeed({ url }) {
  const posts = useJazzyNews();

  return (
    <>
      <h1>{posts.length} articles</h1>
      {posts.map(post => (
        <Post key={post.id} {...post} />
      ))}
    </>
  );
}
```

7.1.2 深入检查依赖

目前，我们添加到依赖数组中的都是字符串。JavaScript 原始类型，例如字符串、布尔值、数字等都是可比较的。字符串与字符串相等是符合直觉的：

```
if ("gnar" === "gnar") {
  console.log("gnarly!!");
}
```

然而，对象、数组和函数的比较就不同了。例如，比较两个数组：

```
if ([1, 2, 3] !== [1, 2, 3]) {
  console.log("but they are the same");
}
```

这里的 [1, 2, 3] 和 [1, 2, 3] 是不相等的，尽管二者的长度和元素完全一样。这是因为它们是一个看似相同的数组的两个不同实例。如果创建一个变量存储这个数组值，然后比较，那就会看到预期的输出：

```
const array = [1, 2, 3];
if (array === array) {
  console.log("because it's the exact same instance");
}
```

在 JavaScript 中，对数组、对象和函数来说，仅当完全是同一个实例时才相等。说了这么多，与 useEffect 的依赖数组有什么关系呢？为了说明这一点，我们需要一个能随时强制渲染的组件。下面构建一个钩子，不管按什么键都渲染组件。

```
const useAnyKeyToRender = () => {
  const [, forceRender] = useState();

  useEffect(() => {
    window.addEventListener("keydown", forceRender);
    return () => window.removeEventListener("keydown", forceRender);
  }, []);
};
```

往简单了说，只需调用改变状态的函数就能强制渲染。我们不关心状态的值，有改变状态的 forceRender 函数就行（鉴于此，使用数组析构时添加了一个逗号。第 2 章讲过，还记得吗）。这个组件首次渲染时，监听按键事件。发现按键后，调用 forceRender，强制组件渲染。按照前面的做法，我们返回了一个清理函数，用于停止监听按键事件。把这个钩子添加到组件中，只需按下一个键就能强制组件重新渲染。

构建好这个自定义的钩子后，在 App 组件中使用（其实任何组件均可以，钩子不挑的）。

```
function App() {
  useAnyKeyToRender();

  useEffect(() => {
    console.log("fresh render");
  });

  return <h1>Open the console</h1>;
}
```

每次按键，App 组件都会渲染。为了表明这一点，每次渲染 App 组件都通过 useEffect 在控制台中输出“fresh render”。下面修改一下 App 中的 useEffect，让它参考 word 的值，在 word 发生变化时重新渲染：

```
const word = "gnar";
useEffect(() => {
  console.log("fresh render");
}, [word]);
```

现在不是每次按键都调用 useEffect 了，而是只在首次渲染之后和 word 的值有变化时。这个值是不变的，因此后面不会重新渲染。在依赖数组中添加原始类型或数字均是如此，这个效应只被调用一次。

如果不是一个单词，而是一个单词数组呢？

```
const words = ["sick", "powder", "day"];
useEffect(() => {
  console.log("fresh render");
}, [words]);
```

words 变量的值是一个数组。由于每次渲染都新声明一个数组，因此在 JavaScript 看来，words 是有变化的，所以每次都会调用“fresh render”效应。每一次数组都是一个新实例，这被视为可触发重新渲染的变动。

在 App 的作用域之外声明 words 就可以解决这个问题：

```
const words = ["sick", "powder", "day"];

function App() {
  useAnyKeyToRender();
  useEffect(() => {
    console.log("fresh render");
  }, [words]);

  return <h1>component</h1>;
}
```

这里的依赖数组引用的是在组件函数外部声明的同一个 words 实例。首次渲染之后，“fresh render”效应不再被调用，因为 words 实例始终不变。对这个示例来说，这是个好的方法，然而并不是所有变量都适合（或建议）在组件函数外部定义，有时传给依赖数组的值必须在作用域内定义。例如，可能需要根据一个 React 属性（下例中的 children）创建 words：

```
function WordCount({ children = "" }) {
  useAnyKeyToRender();

  const words = children.split(" ");

  useEffect(() => {
    console.log("fresh render");
  }, [words]);

  return (
    <>
      <p>{children}</p>
      <p>
        <strong>{words.length} - words</strong>
      </p>
    </>
  );
}
```

```
function App() {
  return <WordCount>You are not going to believe this but...</WordCount>;
}
```

App 组件中的词通过 children 属性传给 WordCount 组件，然后调用 .split 把 words 设为由这些词构成的数组。我们只希望在 words 发生变化时重新渲染组件，可是只要按下一个键，就会在控制台中看到“fresh render”。

不要慌张，React 团队没有坐视不管，为我们提供了避免额外渲染的方式。你可能猜到了，这个问题的解决方法是使用另一个钩子——useMemo。

useMemo 调用一个函数，计算得到一个备忘值。在计算机科学中，备忘（memoization）技术一般用于提升性能。对支持备忘的函数来说，调用函数得到的结果会被保存并缓存起来。以后如果使用相同的输入调用函数，返回的是缓存的值。在 React 中，我们使用 useMemo 比较缓存的值与当前值，判断值是不是真的变了。

使用 useMemo 时要传入一个函数，用于计算并创建备忘值。仅当有依赖发生变化时，useMemo 才重新计算值。首先，导入 useMemo 钩子：

```
import React, { useEffect, useMemo } from "react";
```

然后，使用这个函数设定 words：

```
const words = useMemo(() => {
  const words = children.split(" ");
  return words;
}, []);

useEffect(() => {
  console.log("fresh render");
}, [words]);
```

useMemo 调用传给它的函数，把 words 设为该函数的返回值。与 useEffect 一样，useMemo 根据一个依赖数组做判断。

```
const words = useMemo(() => children.split(" "));
```

如果没有为 useMemo 提供依赖数组，每次渲染都会计算 words 的值。依赖数组控制着何时调用回调函数。传给 useMemo 函数的第二个参数是依赖数组，应该包含 children 值：

```
function WordCount({ children = "" }) {
  useAnyKeyToRender();

  const words = useMemo(() => children.split(" "), [children]);

  useEffect(() => {
    console.log("fresh render");
  }, [words]);

  return (...);
}
```

words 数组依赖 children 属性。如果 children 发生了变化，应该重新计算 words 的值，体现变化。useMemo 在组件首次渲染和 children 属性发生变化时重新计算 words 的值。

创建 React 应用，一定要理解 useMemo 钩子的作用。

useCallback 的作用与 useMemo 类似，不过备忘的是函数而不是值。例如：

```
const fn = () => {
  console.log("hello");
  console.log("world");
};

useEffect(() => {
  console.log("fresh render");
  fn();
}, [fn]);
```

fn 函数先输出“hello”再输出“world”。这个函数是 useEffect 的一个依赖，

与 words 一样，在 JavaScript 看来，每次渲染 fn 都是不同的。因此，每次渲染都会触发效应，每次按键都会输出“fresh render”。这可不是我们想要的。

下面使用 useCallback 包装这个函数：

```
const fn = useCallback(() => {
  console.log("hello");
  console.log("world");
}, []);

useEffect(() => {
  console.log("fresh render");
  fn();
}, [fn]);
```

useCallback 将备忘 fn 函数的值。与 useMemo 和 useEffect 一样，useCallback 的第二个参数也是一个依赖数组。这里，这个备忘回调只创建了一次，因为依赖数组是空的。

现在，我们知道 useMemo 和 useCallback 的用法及二者之间的区别了，下面来改进前面自定义的 useJazzyNews 钩子。每次有新文章，我们要调用 newPostChime.play()。在这个钩子中，posts 是一个数组，因此要使用 useMemo 备忘它的值：

```
const useJazzyNews = () => {
  const [_posts, setPosts] = useState([]);
  const addPost = post => setPosts(allPosts => [post, ...allPosts]);

  const posts = useMemo(() => _posts, [_posts]);

  useEffect(() => {
    newPostChime.play();
  }, [posts]);

  useEffect(() => {
    newsFeed.subscribe(addPost);
    return () => newsFeed.unsubscribe(addPost);
  }, []);
```

```
    useEffect(() => {
      welcomeChime.play();
      return () => goodbyeChime.play();
    }, []);

    return posts;
  };
```

现在，每次有新文章，useJazzyNews 钩子都会播放一段音效。为此，我们对钩子做了几处修改。首先，const [posts, setPosts] 变成了 const [_posts, setPosts]。我们将在 _posts 发生变化时计算 posts 的新值。

其次，增加了一个效应，在 posts 数组发生变化时播放音效。我们监听着新闻源，一旦出现新文章，重新调用钩子，把新文章存入 _posts。接着，备忘 post 的新值，因为 _posts 发生了变化。然后，播放音效，因为这个效应依赖 posts。音效只在文章列表发生变化时播放，而文章列表只在增加了新文章时发生变化。

本章后文将探讨 React Profiler，这是一个浏览器扩展，用于测试性能和检测 React 组件的渲染情况。届时将进一步说明何时使用 useMemo 和 useCallback（剧透一下，要谨慎使用）。

7.1.3 何时使用 useLayoutEffect

我们知道，渲染始终发生在 useEffect 之前。渲染在前，然后各个效应按顺序运行，而且效应可以访问渲染后的所有值。浏览一下 React 文档你便会发现还有一种效应钩子，即 useLayoutEffect。

useLayoutEffect 在渲染循环的特定时刻调用。这一系列事件是按照下述顺序发生的：

1. 渲染。
2. 调用 useLayoutEffect。

3. 浏览器绘制，即把组件元素添加到 DOM 中。

4. 调用 useEffect。

只需添加一些控制台消息便可以看出执行顺序：

```
import React, { useEffect, useLayoutEffect } from "react";

function App() {
  useEffect(() => console.log("useEffect"));
  useLayoutEffect(() => console.log("useLayoutEffect"));
  return <div>ready</div>;
}
```

在 App 组件中，第一个钩子是 useEffect，随后是 useLayoutEffect。我们发现，useLayoutEffect 在 useEffect 之前调用：

```
useLayoutEffect
useEffect
```

useLayoutEffect 在渲染之后，浏览器绘制变化之前调用。多数情况下，你需要的是 useEffect，但是如果要实现的效果对浏览器绘制很重要（屏幕上 UI 元素的外观或位置），那就要使用 useLayoutEffect。比如说，我们想在调整窗口的大小之后获取元素的宽度和高度：

```
function useWindowSize {
  const [width, setWidth] = useState(0);
  const [height, setHeight] = useState(0);

  const resize = () => {
    setWidth(window.innerWidth);
    setHeight(window.innerHeight);
  };

  useLayoutEffect(() => {
    window.addEventListener("resize", resize);
    resize();
    return () => window.removeEventListener("resize", resize);
  }, []);
```

```
  return [width, height];
};
```

你的组件可能想在浏览器开始绘制之前知道窗口的 width 和 height 信息，因此我们使用 useLayoutEffect 在绘制之前计算窗口的 width 和 height。跟踪鼠标的位置也要使用 useLayoutEffect，例如：

```
function useMousePosition {
  const [x, setX] = useState(0);
  const [y, setY] = useState(0);

  const setPosition = ({ x, y }) => {
    setX(x);
    setY(y);
  };

  useLayoutEffect(() => {
    window.addEventListener("mousemove", setPosition);
    return () => window.removeEventListener("mousemove", setPosition);
  }, []);

  return [x, y];
};
```

在绘制屏幕时很有可能需要使用鼠标的 x 和 y 坐标位置。在绘制之前，可以使用 useLayoutEffect 精确计算这两个坐标位置。

7.1.4 钩子使用规则

使用钩子时，有一些准则要熟记于心，以免出现 bug 或不寻常的行为。

钩子只在组件的作用域中运行

钩子只能在 React 组件中调用。钩子也可以添加到自定义的钩子中，不过最终也是添加到组件中。钩子不是常规的 JavaScript 代码，而是一种 React 模式，不过其他库也开始使用了。

建议把功能分解到多个钩子中

在前面的爵士乐新闻组件示例中，我们把与订阅有关的功能放在一个效应中，把与音效有关的功能都放在另一个效应中。这样写出的代码更易于阅读，此外还有一个好处。由于钩子是按顺序调用的，因此最好让钩子保持小的体量。调用钩子后，React 在一个数组中保存钩子的值，以便跟踪值。以下面的组件为例：

```
function Counter() {
  const [count, setCount] = useState(0);
  const [checked, toggle] = useState(false);

  useEffect(() => {
    ...
  }, [checked]);

  useEffect(() => {
    ...
  }, []);

  useEffect(() => {
    ...
  }, [count]);

  return ( ... )
}
```

每一次渲染钩子的调用顺序都是一样的：

```
[count, checked, DependencyArray, DependencyArray, DependencyArray]
```

钩子只应该在顶层代码中调用

钩子只应该在 React 函数的顶层代码中使用，不能放在条件语句、循环或嵌套函数中。我们来调整一下前面的计数器：

```
function Counter() {
  const [count, setCount] = useState(0);
```

```
  if (count > 5) {
    const [checked, toggle] = useState(false);
  }

  useEffect(() => {
    ...
  });

  if (count > 5) {
    useEffect(() => {
      ...
    });
  }

  useEffect(() => {
    ...
  });

  return ( ... )
}
```

放在 `if` 语句中的 `useState`，意思是当 `count` 值大于 5 时调用钩子。这样钩子便游离在数组值之外了。有时数组是 `[count, checked, DependencyArray, 0, DependencyArray]`，有时是 `[count, DependencyArray, 1]`。效应在这个数组中的索引对 React 是很重要的，值就是按索引保存的。

那么，这是不是意味着在 React 应用中不能使用条件判断逻辑了？当然不是！我们只是需要以不同的方式组织条件语句。我们可以在钩子中嵌套 `if` 语句、循环和其他条件语句：

```
function Counter() {
  const [count, setCount] = useState(0);
  const [checked, toggle] =
    useState(
      count => (count < 5)
      ? undefined
      : !c,
      (count < 5) ? undefined
    );
```

```
  useEffect(() => {
    ...
  });

  useEffect(() => {
    if (count < 5) return;
    ...
  });

  useEffect(() => {
    ...
  });

  return ( ... )
}
```

这里，checked的值取决于count是不是大于5。如果count小于5，checked的值是undefined。像这样把条件语句嵌套在钩子中，钩子依然在顶层，而最终的效果是类似的。第二个钩子也遵守相同的规则。如果count小于5，return语句将阻止该效应继续执行。这样可以确保钩子数组的值保持不变，始终为[countValue, checkedValue, DependencyArray, DependencyArray, DependencyArray]。

与条件逻辑一样，异步操作也要嵌套到钩子中。useEffect的第一个参数是一个函数，而不是一个promise。因此，第一个参数不能是异步函数，例如useEffect(async () => {})。然而，在嵌套的函数中可以创建异步函数，如下所示：

```
useEffect(() => {
  const fn = async () => {
    await SomePromise();
  };
  fn();
});
```

我们创建变量fn处理async/await，然后在函数结尾调用这个函数。我们可以为异步函数命名，也可以使用匿名函数：

```
useEffect(() => {
  (async () => {
    await SomePromise();
  })();
});
```

遵守上述规则可以避免一些常见的 React 钩子陷阱。Create React App 包含一个名为 eslint-plugin-react-hooks 的 ESLint 插件，如果你违反了这些规则它会提醒你。

7.1.5 使用 useReducer 改进代码

回过头再来看 Checkbox 组件。这个组件持有简单的状态，是个不错的示例。复选框要么勾选了，要么未勾选。checked 是状态值，setChecked 是用于改变状态的函数。首次渲染这个组件时，checked 的值为 false。

```
function Checkbox() {
  const [checked, setChecked] = useState(false);

  return (
    <>
      <input
        type="checkbox"
        value={checked}
        onChange={() => setChecked(checked => !checked)}
      />
      {checked ? "checked" : "not checked"}
    </>
  );
}
```

这样写没什么问题，但是其中一部分可能会导致异常：

```
onChange={() => setChecked(checked => !checked)}
```

请仔细观察。乍一看是没什么，可这不是在自找苦吃吗？我们传入一个函数，该函数接收的参数为 checked 的当前值，返回值是它的相反值，即 !checked。

这太复杂了，似乎没这个必要。开发者极易发送错误的信息，酿成大祸。与其这样处理，为何不提供一个转换函数呢？

下面添加一个名为 toggle 的函数，让它执行相同的操作，即调用 setChecked，返回 checked 当前值的相反值。

```
function Checkbox() {
  const [checked, setChecked] = useState(false);

  function toggle() {
    setChecked(checked => !checked);
  }

  return (
    <>
      <input type="checkbox" value={checked} onChange={toggle} />
      {checked ? "checked" : "not checked"}
    </>
  );
}
```

这样好多了。我们把 onChange 设为一个可以想见的值——toggle 函数。不管何时何地，我们都知道该函数会执行什么操作。我们甚至还可以更进一步，让每次使用 Checkbox 组件得到的结果都可以预见。还记得我们在 toggle 函数中发给 setChecked 的函数吗？

```
setChecked(checked => !checked);
```

现在，我们要使用一个专门的名称指代 checked => !checked 函数：reducer。reducer 函数最简单的定义是，接收当前状态并返回新状态的函数。如果 checked 为 false，应该返回相反值 true。我们不再把这个行为硬编码在 onChange 事件中，而是提取到一个 reducer 函数中，始终生成相同的结果。现在，在组件中不使用 useState 了，转而使用 useReducer。

```
function Checkbox() {
  const [checked, toggle] = useReducer(checked => !checked, false);
```

```
  return (
    <>
      <input type="checkbox" value={checked} onChange={toggle} />
      {checked ? "checked" : "not checked"}
    </>
  );
}
```

useReducer 接受的参数为 reducer 函数和初始状态 false。然后，把 onChange 属性设为 toggle，调用 reducer 函数。

前面的 reducer 函数 checked => !checked 是个很有代表性的示例。如果为函数提供相同的输入，得到的输出也应该相同。这个概念源自 JavaScript 中的 Array.reduce。reduce 的作用与 reducer 函数基本相同：接收一个函数（把全部值归约为一个值）和一个初始值，返回一个值。

Array.reduce 接收一个 reducer 函数和一个初始值。numbers 数组中的各个值依次传给 reducer 函数，直到最终返回一个值。

```
const numbers = [28, 34, 67, 68];

numbers.reduce((number, nextNumber) => number + nextNumber, 0); // 197
```

这里，传给 Array.reduce 的 reducer 函数接收两个参数。此外，reducer 函数还可以接收更多参数。

```
function Numbers() {
  const [number, setNumber] = useReducer(
    (number, newNumber) => number + newNumber,
    0
  );

  return <h1 onClick={() => setNumber(30)}>{number}</h1>;
}
```

每次点击 h1，在总数上加 30。

7.1.6 使用 useReducer 处理复杂的状态

对复杂的状态来说，使用 useReducer 处理状态更新，结果更容易预料。以下面包含用户数据的对象为例：

```
const firstUser = {
  id: "0391-3233-3201",
  firstName: "Bill",
  lastName: "Wilson",
  city: "Missoula",
  state: "Montana",
  email: "bwilson@mtnwilsons.com",
  admin: false
};
```

假设有个名为 User 的组件，设定 firstUser 为初始状态，然后显示相应的数据。

```
function User() {
  const [user, setUser] = useState(firstUser);

  return (
    <div>
      <h1>
        {user.firstName} {user.lastName} - {user.admin ? "Admin" : "User"}
      </h1>
      <p>Email: {user.email}</p>
      <p>
        Location: {user.city}, {user.state}
      </p>
      <button>Make Admin</button>
    </div>
  );
}
```

管理状态时一个常见的错误是覆盖状态：

```
<button
  onClick={() => {
    setUser({ admin: true });
  }}
```

```
>
  Make Admin
</button>
```

这样做将覆盖 firstUser 状态，把状态替换为发给 setUser 函数的值，即 {admin: true}。正确的做法是展开用户对象的当前值，然后覆盖 admin 值：

```
<button
  onClick={() => {
    setUser({ ...user, admin: true });
  }}
>
  Make Admin
</button>
```

现在是在初始状态的基础上增加新的键值对 {admin: true}。每次处理 onClick 事件都要重写这个逻辑，很容易出错（明天接着开发应用可能就会忘记）。

```
function User() {
  const [user, setUser] = useReducer(
    (user, newDetails) => ({ ...user, ...newDetails }),
    firstUser
  );
  ...
}
```

我们可以把新状态值 newDetails 发给 reducer 函数，让它把新值推送进对象。

```
<button
  onClick={() => {
    setUser({ admin: true });
  }}
>
  Make Admin
</button>
```

如果状态有多个子值，或者下一个状态依赖于前一个状态，就可以使用这种模式。授之以鱼（展开）不如授之以渔（useReducer）。

以前的 setState 和 useReducer

在 React 之前的版本中，使用 setState 函数更新状态，初始状态要在构造方法中使用对象赋值。

```
class User extends React.Component {
  constructor(props) {
   super(props);
   this.state = {
     id: "0391-3233-3201",
     firstName: "Bill",
     lastName: "Wilson",
     city: "Missoula",
     state: "Montana",
     email: "bwilson@mtnwilsons.com",
     admin: false
   };
  }
}
<button onSubmit={() =>
       {this.setState({admin: true });}}
Make Admin
</button>
```

以前，setState 合并状态值。useReducer 也是如此。

```
const [state, setState] = useReducer(
       (state, newState) =>
            ({...state, ...newState}),
       initialState);

<button onSubmit={() =>
       {setState({admin: true });}}
Make Admin
</button>
</div>);
```

如果你喜欢这种模式，可以使用 npm 包 legacy-set-state 或 useReducer。

前文给出的示例简单演示了 reducer 函数的应用。下一章将深入探讨 reducer 设计模式，借以简化应用中的状态管理。

7.1.7 提升组件性能

在 React 应用中，组件渲染的次数不在少数。提升组件性能涉及两方面，一是避免不必要的渲染，二是减少渲染传播的时间。React 自身提供了一些可用来避免非必要渲染的工具：memo、useMemo 和 useCallback。本章前面已经介绍了 useMemo 和 useCallback，本节将深入分析，说明如何使用这两个钩子提升网站的性能。

memo 函数用于创建纯组件。读过第 3 章我们知道，对给定的参数，纯函数始终返回相同的结果。纯组件与此相同。在 React 中，对给定的属性，纯组件始终渲染相同的输出。

下面来创建一个名为 Cat 的组件：

```
const Cat = ({ name }) => {
  console.log(`rendering ${name}`);
  return <p>{name}</p>;
};
```

Cat 就是一个纯组件，输出始终是一个显示 name 属性的段落。如果通过属性提供的名称相同，输出也就相同。

```
function App() {
  const [cats, setCats] = useState(["Biscuit", "Jungle", "Outlaw"]);
  return (
    <>
      {cats.map((name, i) => (
        <Cat key={i} name={name} />
      ))}
      <button onClick={() => setCats([...cats, prompt("Name a cat")])}>
        Add a Cat
      </button>
```

```
      </>
    );
  }
```

这个应用使用 Cat 组件。首次渲染后，控制台中将输出：

```
rendering Biscuit
rendering Jungle
rendering Outlaw
```

单击“Add a Cat”按钮将弹出一个窗口，让用户添加一只猫。

假如我们添加了一只名为“Ripple”的猫，渲染 Cat 组件输出的结果为：

```
rendering Biscuit
rendering Jungle
rendering Outlaw
rendering Ripple
```



这段代码可以正常运行，因为 prompt 会阻塞代码执行。这里只是举个例子，在真实的应用中不要使用 prompt。

每增加一只猫的名称，Cat 组件就多渲染一次，可 Cat 是纯组件，对给定的属性来说，输出并没有变化，不是每次都要渲染。如果属性没有变化，我们可不想重新渲染纯组件。使用 memo 函数可以创建只在属性有变化时渲染的组件。首先从 React 库中导入这个函数，然后使用它包装当前的 Cat 组件：

```
import React, { useState, memo } from "react";

const Cat = ({ name }) => {
  console.log(`rendering ${name}`);
  return <p>{name}</p>;
};

const PureCat = memo(Cat);
```

这里，我们创建了一个新组件，名为 PureCat。只有属性发生变化时，PureCat 才会渲染 Cat。然后，我们可以把 App 组件中的 Cat 换成 PureCat：

```
cats.map((name, i) => <PureCat key={i} name={name} />);
```

现在，新增一只猫的名称后，例如“Pancake”，在控制台中只会看到一次渲染：

```
rendering Pancake
```

由于其他猫的名称没有变化，因此不需要渲染对应的 Cat 组件。这对 name 属性作用明显，那么要是为 Cat 组件增加一个函数属性呢？

```
const Cat = memo(({ name, meow = f => f }) => {
  console.log(`rendering ${name}`);
  return <p onClick={() => meow(name)}>{name}</p>;
});
```

每次单击一只猫的名称，我们可以使用这个属性在控制台中输出一次喵喵叫：

```
<PureCat key={i} name={name} meow={name => console.log(`${name} has meowed`)} />
```

这样改动之后，PureCat 不能像设想中那样使用了，即使 name 属性保持不变，每个 Cat 组件还是全部渲染。这都是添加 meow 属性导致的结果。让人意想不到的是，每次定义的 meow 函数属性都是一个新函数。对 React 来说，meow 属性发生了变化，因此要重新渲染组件。

我们可以使用 memo 函数定义更具体的规则，指明何时重新渲染组件：

```
const RenderCatOnce = memo(Cat, () => true);
const AlwaysRenderCat = memo(Cat, () => false);
```

传给 memo 函数的第二个参数是一个断言（predicate），即一个只返回 true 或 false 的函数。这个函数决定是否重新渲染 Cat 组件：返回 false，重新渲染 Cat 组件；返回 true，不重新渲染 Cat 组件。无论如何，Cat 组件至少会渲染一

次。鉴于此，对 RenderCatOnce 来说，组件渲染一次后就不再渲染了。通常，我们使用断言函数检查具体的值：

```
const PureCat = memo(
  Cat,
  (prevProps, nextProps) => prevProps.name === nextProps.name
);
```

我们在第二个参数中比较属性，决定是否重新渲染 Cat 组件。断言函数能接收到前一组属性和下一组属性，我们就通过这两个对象比较 name 属性。如果 name 属性变了，重新渲染组件；如果 name 属性没变，不管 meow 属性如何，都重新渲染组件。

7.1.8 shouldComponentUpdate 和 PureComponent

前面讨论的概念不是 React 新引进的。memo 函数是一个常见问题的新解决方案。在 React 之前的版本中，有个名为 shouldComponentUpdate 的方法。如果在组件中定义了这个方法，React 将据此判断在什么情况下应该更新组件。shouldComponentUpdate 定义哪些属性或状态发生变化才应该重新渲染组件。React 库引入 shouldComponentUpdate 后，立即成为众多用户的掌中宝。由于太受欢迎了，React 团队甚至推出了一种创建类组件的新方式。类组件像下面这样定义：

```
class Cat extends React.Component {
  render() {
    return (
      {name} is a good cat!
    )
  }
}
```

PureComponent 像下面这样定义：

```
class Cat extends React.PureComponent {
  render() {
    return (
```

```
        {name} is a good cat!
      )
    }
  }
```

PureComponent 的作用与 React.memo 相同，不过前者只适用类组件，而后者只适用函数组件。

useCallback 和 useMemo 可用于备忘对象和函数属性。下面在 Cat 组件中使用 useCallback：

```
const PureCat = memo(Cat);
function App() {
  const meow = useCallback(name => console.log(`${name} has meowed`, []);
  return <PureCat name="Biscuit" meow={meow} />
}
```

这里，我们没有为 memo(Cat) 提供检查属性的断言，而是使用 useCallback 确保 meow 函数没有变化。使用这些函数可以减少组件树中重新渲染的次数。

7.1.9 何时重构

我们讨论的最后两个钩子（useMemo 和 useCallback）以及 memo 函数，往往被过度使用了。在设计上，React 追求速度，为此势必要大量渲染组件。在你决定使用 React 开始就要考虑优化性能。React 本身并不慢，深层次的优化应该放在最后。

重构也有取舍。到处都使用 useCallback 和 useMemo 不见得是件好事，也有可能对性能产生负面影响。编写的代码增多了，而且花费的时间也增加了。针对性能进行重构时，一定要有目标，比如说不再让屏幕假死或闪烁。有时，你知道是一些耗时的函数在拖慢应用的运行速度。

你可以使用 React Profiler 衡量各个组件的性能。你可能已经安装了 React 开发者工具（Chrome 扩展：*https://oreil.ly/1UNct*，Firefox 扩展：*https://oreil.ly/0NYbR*），它也带有分析程序。

在重构之前，要确保应用可以正常运行，而且对代码基是满意的。过度重构，或者在应用可运行之前重构，都可能引入难以发现的奇怪 bug，而且或许并不值得投入时间和精力去做优化。

本章和前一章介绍了很多 React 自带的钩子，举例说明了各个钩子的用法，还利用这些钩子自定义了钩子。接下来，我们将在这些基本技能的基础上介绍一些其他库和高级模式。

第 8 章
处理数据

数据是应用的命脉。数据像水一样具有流动性，用值滋养着组件。我们编写的用户界面组件是数据的容器，盛放着来自互联网的数据。与此同时，我们还收集、创建，并向互联网中发送新数据。应用的价值不在组件身上，而在于流经组件的数据。

数据好比是水或食物。云是无穷无尽的源头，我们向它发送数据，也从中获取数据。这就是互联网。我们在网络、服务、系统和数据库中处理和存储 ZB 级数据。云为客户端源源不断灌输最新鲜的数据。我们则在本地处理甚至存储这些数据。然而，倘若本地数据与源头失去了同步，数据就不新鲜了，我们称之为数据腐败。

这是开发者在处理数据时要面对的挑战。我们要确保云始终为应用灌输新鲜的数据。本章介绍从源头加载并处理数据的几种不同技术。

8.1 请求数据

在“星球大战”电影中，C-3P0 是一个礼仪机器人。显而易见，他的专长是沟通。他能说六百多万种语言。当然，C-3P0 知道如何发送 HTTP 请求，毕竟超文本传送协议（Hyper Text Transfer Protocol）是在互联网中收发数据最为流行的一种协议。

HTTP 是互联网通信的支柱。我们在浏览器中加载 *http://www.google.com*，其实是请求 Google 给我们发送一个搜索表单。实现搜索功能的文件就是通过 HTTP 传送给浏览器的。我们在 Google 中搜索“cat photos”，其实是请求 Google 为我们寻找猫咪图片。作为回应，Google 通过 HTTP 把图像数据传送给浏览器。

在 JavaScript 中，最常使用 fetch 发起 HTTP 请求。如果向 GitHub 询问有关 Moon Highway 的信息，可以使用 fetch 发送一个请求。

```
fetch(`https://api.github.com/users/moonhighway`)
  .then(response => response.json())
  .then(console.log)
  .catch(console.error);
```

fetch 函数返回一个 promise。这里，我们向指定的 URL，即 *https://api.github.com/users/moonhighway* 发起一个异步请求。请求在互联网中传送，以及响应信息要花点儿时间。传送完毕后，信息通过 .then(callback) 方法传给一个回调。GitHub API 响应的数据是 JSON 格式，而且包含在 HTTP 响应的主体中，因此我们调用 response.json() 获取数据，把数据解析为 JSON 格式。获得响应信息后，把数据输出到控制台中。如果有什么地方出错了，把错误传给 console.error 方法。

针对这个请求，GitHub 返回的响应是一个 JSON 对象：

```
{
  "login": "MoonHighway",
  "id": 5952087,
  "node_id": "MDEyOk9yZ2FuaXphdGlvbjU5NTIwODc=",
  "avatar_url": "https://avatars0.githubusercontent.com/u/5952087?v=4",
  "bio": "Web Development classroom training materials.",

  ...

}
```

GitHub API 提供了关于用户账户的基本信息。你可以请求自己的信息试一试：*https://api.github.com/users/<YOUR_GITHUB_USER_NAME>*。

promise 还可以使用 async/await 处理。由于 fetch 返回一个 promise，因此我们可以在 async 函数中异步等待（await）请求：

```
async function requestGithubUser(githubLogin) {
  try {
    const response = await fetch(
      `https://api.github.com/users/${githubLogin}`
    );
    const userData = await response.json();
    console.log(userData);
  } catch (error) {
    console.error(error);
  }
}
```

这与前面串接 .then 函数处理请求的效果是完全一样的。在异步等待一个 promise 时，下一行代码直到 promise 得到解决后才执行。以这种方式在代码中处理 promise 比较简洁。这两种方式在本章后面的内容中都有用到。

8.1.1 随请求发送数据

很多请求需要随之上传数据。比如说，我们要收集有关用户的信息，用于创建账户，或者需要提供关于用户的新信息，用于更新账户。

一般来说，创建数据使用 POST 请求，修改数据使用 PUT 请求。fetch 函数的第二个参数接收一个选项对象，供 fetch 创建 HTTP 请求。

```
fetch("/create/user", {
  method: "POST",
  body: JSON.stringify({ username, password, bio })
});
```

这个请求使用 POST 方法创建一个新用户。username、password 和用户的 bio 以字符串的形式在请求的 body 中发送。

8.1.2 使用 fetch 上传文件

上传文件需要使用一种不同的 HTTP 请求：multipart-formdata 请求。这种请求告诉服务器，请求的主体中有一个或多个文件。在 JavaScript 中发起这种请求，只需在请求的主体中传送一个 FormData 对象。

```
const formData = new FormData();
formData.append("username", "moontahoe");
formData.append("fullname", "Alex Banks");
forData.append("avatar", imgFile);

fetch("/create/user", {
  method: "POST",
  body: formData
});
```

这一次创建用户，我们通过一个 formData 对象随请求一起传送 username、fullname 和 avatar 图像。这些值目前是硬编码的，不过也可以从表单中轻易收集。

8.1.3 授权请求

有时，我们要获得授权才能发起请求。通常，授权意味着获取个人或敏感数据。而且，基本上要求用户通过 POST、PUT 或 DELETE 请求在服务器上执行一定的操作。

一般情况下，我们在每个请求中添加一个唯一的令牌（token），供服务识别用户的身份。这个令牌通常在 Authorization 首部中添加。以 GitHub 为例，在请求中发送令牌才能获得个人账户信息：

```
fetch(`https://api.github.com/users/${login}`, {
  method: "GET",
  headers: {
    Authorization: `Bearer ${token}`
  }
});
```

令牌通常在用户使用用户名和密码登录服务后获取。此外，也可以使用开放标准协议 OAuth 从第三方（例如 GitHub 或 Facebook）获取。

GitHub 可为你生成一个个人用户令牌。首先登录 GitHub，依次进入 Settings → Developer Settings → Personal Access Tokens。然后创建具有特定读写规则的令牌，最后使用令牌从 GitHub API 中获取个人信息。随 fetch 请求一起发送 Personal Access Token，GitHub 将提供更多有关账户的隐私信息。

在 React 组件中获取数据要合理使用 useState 和 useEffect 钩子。前者的作用是把响应存储在状态中，后者的作用是发起 fetch 请求。假如我们想在一个组件中显示一个 GitHub 用户的信息，可以使用下述代码：

```
import React, { useState, useEffect } from "react";

function GitHubUser({ login }) {
  const [data, setData] = useState();

  useEffect(() => {
    if (!login) return;
    fetch(`https://api.github.com/users/${login}`)
      .then(response => response.json())
      .then(setData)
      .catch(console.error);
  }, [login]);

  if (data)
    return <pre>{JSON.stringify(data, null, 2)}</pre>;

  return null;
}

export default function App() {
  return <GitHubUser login="moonhighway" />;
}
```

在这段代码中，App 组件渲染 GitHubUser 组件，以 JSON 格式显示有关 moonhighway 的数据。首次渲染时，GitHubUser 使用 useState 钩子为 data 设

置一个状态变量。由于 data 的初始值是 null，所以组件返回 null。组件返回 null 的意思是告诉React，什么也不渲染。这样做没有问题，只是会看到一个黑屏。

渲染组件之后，调用 useEffect 钩子。fetch 请求就在这里发起。得到响应后，获取响应数据，解析为 JSON 格式。然后，把 JSON 对象传给 setData 函数，再次渲染组件，不过这一次有数据了。这个 useEffect 钩子不会再次被调用，除非 login 的值发生了变化。如果这个值有变化，表明我们想请求另一个 GitHub 用户的信息。

有数据时，在一个 pre 元素中渲染 JSON 字符串。JSON.stringify 方法接受三个参数：要转换成字符串的 JSON 数据、用于替换 JSON 对象中属性的函数，以及格式化数据时使用的空格数量。这里，我们把替换函数设为 null，即不做任何替换。2 表示格式化代码时使用的空格数量，即以两个空格缩进 JSON 字符串。使用 pre 元素的目的是保留空白，确保最终渲染出来的 JSON 字符串具有良好的可读性。

8.1.4 在本地保存数据

我们可以使用 Web Storage API 把数据存储在本地浏览器中。保存数据可以使用 window.localStorage 或 window.sessionStorage 对象。sessionStorage API 只把数据保存到用户会话中，关闭标签页或重启浏览器，保存的数据都将清空。而 localStorage 将无限期保存数据，除非主动删除。

JSON 数据应该以字符串的形式保存到浏览器存储空间中。这意味着，保存时要把对象转换成 JSON 对象，而在加载时则要把字符串解析为 JSON。下面是可用于保存和加载 JSON 数据的函数：

```
const loadJSON = key =>
  key && JSON.parse(localStorage.getItem(key));
const saveJSON = (key, data) =>
  localStorage.setItem(key, JSON.stringify(data));
```

loadJSON 函数从 localStorage 中加载指定 key 名下的数据。localStorage.

getItem 函数用于加载数据，如果名下有数据，解析为 JSON 再返回。如果没有数据，loadJSON 函数返回 null。

saveJSON 函数使用一个唯一的标识符 key 把数据保存到 localStorage 中。localStorage.setItem 函数的作用是把数据保存到浏览器中。在保存之前，要先把数据转换成 JSON 字符串。

从 Web Storage 中加载数据、把数据保存到 Web Storage 中、把数据转换成字符串，以及解析 JSON 字符串，统统都是异步任务。loadJSON 和 saveJSON 都是异步函数。因此请注意，如果频繁调用这两个函数处理大量数据，可能会导致性能问题。为了性能，通常最好限制这两个函数的使用频率。

我们可以把前面向 GitHub 发起请求获取的用户数据保存到本地，这样，下次查看同一个用户的信息时就可以使用保存在 localStorage 中的数据，而不用再向 GitHub 发送请求了。把下述代码添加到 GitHubUser 组件中：

```
const [data, setData] = useState(loadJSON(`user:${login}`));
useEffect(() => {
  if (!data) return;
  if (data.login === login) return;
  const { name, avatar_url, location } = data;
  saveJSON(`user:${login}`, {
    name,
    login,
    avatar_url,
    location
  });
}, [data]);
```

loadJSON 是异步函数，因此调用 useState 时可以使用它设定状态数据的初始值。如果浏览器中存有用户数据，放在 user:moonhighway 名下，那就用这里的数据设定初始数据。否则，把 data 的初始值设为 null。

如果 data 发生了变化，说明从 GitHub 加载了数据，那就调用 saveJSON 函数，保存我们需要的用户信息：name、login、avatar_url 和 location。用户对象

中不需要的信息就不用保存了。如果 data 对象为空，即 !data 为真，跳过保存操作。另外，如果当前登录用户的用户名与 data.login 相等，说明该用户的数据已经保存，此时也跳过保存操作。

使用 localStorage 把数据保存到本地浏览器的 GitHubUser 组件的完整代码如下：

```
import React, { useState, useEffect } from "react";

const loadJSON = key =>
  key && JSON.parse(localStorage.getItem(key));
const saveJSON = (key, data) =>
  localStorage.setItem(key, JSON.stringify(data));

function GitHubUser({ login }) {
  const [data, setData] = useState(
    loadJSON(`user:${login}`)
  );

  useEffect(() => {
    if (!data) return;
    if (data.login === login) return;
    const { name, avatar_url, location } = data;
    saveJSON(`user:${login}`, {
      name,
      login,
      avatar_url,
      location
    });
  }, [data]);

  useEffect(() => {
    if (!login) return;
    if (data && data.login === login) return;
    fetch(`https://api.github.com/users/${login}`)
      .then(response => response.json())
      .then(setData)
      .catch(console.error);
  }, [login]);
```

```
    if (data)
      return <pre>{JSON.stringify(data, null, 2)}</pre>;

    return null;
  }
```

注意，GitHubUser 组件现在有两个 useEffect 钩子。第一个钩子的作用是把数据保存到浏览器中，在 data 发生变化时调用。第二个钩子用于从 GitHub 中请求更多的数据。如果本地已经保存了指定用户的数据，不发送 fetch 请求。这一点由第二个钩子中的第二个 if 语句判断，即 if (data && data.login === login) return;。如果 data 有内容，而且内容中的 login 与 login 属性一致，那就没必要再向 GitHub 发送请求，使用本地数据即可。

首次运行这个应用，如果把 login 设为 moonhighway，页面中将渲染如下的对象：

```
{
  "login": "MoonHighway",
  "id": 5952087,
  "node_id": "MDEyOk9yZ2FuaXphdGlvbjU5NTIwODc=",
  "avatar_url": "https://avatars0.githubusercontent.com/u/5952087?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/MoonHighway",
  "html_url": "https://github.com/MoonHighway",

  ...

}
```

这是 GitHub 返回的响应，因为对象中有很多我们不需要的用户信息。首次运行看到的是这个冗长的响应。然而，第二次运行，响应就简短多了：

```
{
  "name": "Moon Highway",
  "login": "moonhighway",
  "avatar_url": "https://avatars0.githubusercontent.com/u/5952087?v=4",
  "location": "Tahoe City, CA"
}
```

这一次，浏览器渲染的是在本地保存的 moonhighway 的信息。由于我们只需要四个字段的数据，因此只保存了那四个字段。在清空存储空间之前，看到的始终是这个简短的离线字段。

```
localStorage.clear();
```

sessionStorage 和 localStorage 均是 Web 开发者的得力武器。离线时，我们可以使用本地数据，减少网络请求数，从而提升应用的性能。然而，一定要合理使用。实现离线存储增加了应用的复杂度，而且在开发的过程中很难处理。另外，请勿使用 Web Storage 缓存数据。如果发现性能有波动，可以试着让 HTTP 处理缓存。添加 Cache-Control: max-age=<EXP_DATE> 首部后，浏览器将自动缓存内容。EXP_DATE 定义内容的失效日期。

8.1.5 处理 promise 状态

HTTP 请求和 promise 均有三种状态：待定、成功（完成）和失败（被拒）。发送请求后等待响应期间，请求处于待定状态。响应只有两种可能，成功或失败。成功的响应表示顺利连接上服务器，收到了数据。对 promise 来说，成功的响应表示 promise 得到解决了。倘若在请求的过程中有什么地方出错了，那就可以说 HTTP 请求失败，或者 promise 被拒了。在这两种情况下，我们将收到一个 error 对象，说明具体情况。

发起 HTTP 请求时，这三种状态都要处理。我们要修改 GitHubUser 组件，在渲染成功响应之外再增加一些内容。请求处理待定状态时，我们可以显示一个“loading...”消息；如果出错了，那就渲染 error 对象中的详情。

```
function GitHubUser({ login }) {
  const [data, setData] = useState();
  const [error, setError] = useState();
  const [loading, setLoading] = useState(false);

  useEffect(() => {
    if (!login) return;
    setLoading(true);
```

```
    fetch(`https://api.github.com/users/${login}`)
      .then(data => data.json())
      .then(setData)
      .then(() => setLoading(false))
      .catch(setError);
  }, [login]);

  if (loading) return <h1>loading...</h1>;
  if (error)
    return <pre>{JSON.stringify(error, null, 2)}</pre>;
  if (!data) return null;

  return (
    <div className="githubUser">
      <img
        src={data.avatar_url}
        alt={data.login}
        style={{ width: 200 }}
      />
      <div>
        <h1>{data.login}</h1>
        {data.name && <p>{data.name}</p>}
        {data.location && <p>{data.location}</p>}
      </div>
    </div>
  );
}
```

如果请求是成功的，屏幕上将渲染 Moon Highway 的信息，供用户查看，如图 8-1 所示。

倘若出了问题，那就以 JSON 字符串的形式显示 `error` 对象。在生产环境中，我们可以进一步处理错误，比如说把跟踪到的错误记录到日志中，或者尝试再次发起请求。在开发环境中，只渲染错误详情没什么问题，反而能让开发者立即得到反馈。

最后，当请求处理待定状态时，我们在一个 `h1` 元素中显示“loading...”消息。



图 8-1：示例输出

有时，HTTP 请求虽然是成功的，但是仍有错误。如果请求成功，成功连上了服务器，也收到了响应，但是响应主体中有错误消息，就是这种情况。有些服务器会在成功的响应中发送额外的失败消息。

处理请求的三种状态增加了不少代码，可是每个请求都需要这么做。请求会耗费一定的时间，而且时常出错。由于所有请求（以及 promise）都有这三种状态，因此我们可以使用一个可重用的钩子、一个组件，甚至是 React 的 Suspense 功能处理所有 HTTP 请求。我们将一一讲解这三种方法，不过首先要介绍一下渲染属性的概念。

8.2 渲染属性

“渲染组件”就是字面意思，即把组件渲染出来。比如说，在满足特定的条件时渲染以属性传入的组件，或者渲染返回组件的函数属性。在第二种情况中，我们可以通过参数传递数据，渲染函数返回的组件。

在异步组件中，为了最大程度提升可重用性，经常利用渲染属性模式。采用这种模式创建组件，可以把开发应用所需的复杂机制或单调的样板代码抽离出来。

以显示一个列表为例：

```
import React from "react";

const tahoe_peaks = [
  { name: "Freel Peak", elevation: 10891 },
  { name: "Monument Peak", elevation: 10067 },
  { name: "Pyramid Peak", elevation: 9983 },
  { name: "Mt. Tallac", elevation: 9735 }
];

export default function App() {
  return (
    <ul>
      {tahoe_peaks.map((peak, i) => (
        <li key={i}>
          {peak.name} - {peak.elevation.toLocaleString()}ft
        </li>
      ))}
    </ul>
  );
}
```

这个示例在一个无序列表中渲染塔霍湖景区的四大高峰。这段代码没什么大问题，不过映射数组后分别渲染各个元素却有点复杂。而且，映射数组也是一件十分常见的任务，在开发中经常用到。我们可以创建一个 `List` 组件，在需要渲染无序列表时使用。

在 JavaScript 中，数组可以包含一些值，也可以为空。如果列表是空的，我们要显示一个消息，让用户知晓。然而，这个消息在不同的场合下可能有所不同。不用担心，列表为空时我们可以传入一个组件供渲染：

```
function List({ data = [], renderEmpty }) {
  if (!data.length) return renderEmpty;
```

```
  return <p>{data.length} items</p>;
}

export default function App() {
  return <List renderEmpty={<p>This list is empty</p>} />;
}
```

List 组件接受两个属性：data 和 renderEmpty。第一个属性 data 表示一组要映射的元素，默认值为一个空数组。第二个属性 renderEmpty 是列表为空时要渲染的组件。当 data.length 等于 0 时，List 组件返回 renderEmpty 属性，渲染通过这个属性传入的组件。

这里，用户看到的消息为 This list is empty。

renderEmpty 就是一个渲染属性，它的值为一个在满足特定条件时渲染的组件。这里，条件是列表为空或未提供 data 属性。

下面通过 data 把一个具体的数组传给该组件：

```
export default function App() {
  return (
    <List
      data={tahoe_peaks}
      renderEmpty={<p>This list is empty</p>}
    />
  );
}
```

此时，这个组件渲染数组中的 4 个元素。

我们还可以指明如何渲染数组中的各个元素。例如，我们可以传入 renderItem 属性：

```
export default function App() {
  return (
    <List
      data={tahoe_peaks}
```

```
        renderEmpty={<p>This list is empty</p>}
        renderItem={item => (
          <>
            {item.name} - {item.elevation.toLocaleString()}ft
          </>
        )}
      />
    );
  }
```

这一次，渲染属性是一个函数。数据（item）通过参数传给这个函数，以指定的方式渲染塔霍湖景区的高峰。这里，我们渲染一个 React 片段，显示元素的 name 和 elevation。对 tahoe_peaks 数组来说，renderItem 属性将被调用四次，数组中的一座高峰调用一次。

这样做可以把映射数组机制抽取出来。现在，映射操作交由 List 组件处理，我们只需告诉它要渲染什么。

```
function List({ data = [], renderItem, renderEmpty }) {
  return !data.length ? (
    renderEmpty
  ) : (
    <ul>
      {data.map((item, i) => (
        <li key={i}>{renderItem(item)}</li>
      ))}
    </ul>
  );
}
```

如果 data 数组不为空，List 组件渲染一个无序列表，即 。List 组件使用 .map 方法映射数组中的各个元素，在列表项目 <li> 中渲染数组中的各个元素。List 数组将确保每个列表项目都有一个唯一的键。在各个 <li> 元素中，我们调用 renderItem 属性，通过参数传入数组元素。最终得到的无序列表显示着塔霍湖景区四大高峰的名称和海拔。

好消息是，我们得到了一个可重用的 `List` 组件，需要渲染无序列表时都可以拿来使用。坏消息是，这个组件的功能略显单薄。其实，处理这项任务有更合适的组件可用。

8.3 虚拟化列表

如果想自己动手开发一个可重用的组件，用于渲染列表，要考虑的情况不少。其中最为重要的一点是要考虑特大型列表。我们在生产环境接触的很多数据点（data point）往往给人一种无穷无尽的感觉，比如说 Google 搜索结果一页接着一页，在 Airbnb 中搜索塔霍湖景区的住处，住宅和公寓列表也仿佛没有尽头。生产环境中的应用通常要渲染大量数据，但是又不能一次性全部渲染。

浏览器的渲染能力是有限的。渲染要耗费时间、占据处理能力及消耗内存，这三项都是有限制的。因此，在开发可重用的列表组件时要考虑到这一点。如果 `data` 数组非常大，我们要怎样应对呢？

尽管搜索住处得到的结果数以千计，但是我们不可能一次性全部看完，任何屏幕空间也不足以显示所有图像、店名和价格。我们一次大概只能看五个结果。如果想看更多的结果，我们可以滚动界面，可是数千个结果滚动的距离也不短。在一个可滚动的图层中渲染一千个结果对手机的要求很高。

我们不渲染一千个结果，而是只渲染 11 个呢？由于用户在屏幕上一次差不多只能查看五个结果，那就渲染用户能看到的五个项目，在窗口可见区域上方和下方再渲染六个项目。在窗口可见区域上方和下方都渲染，方便用户在两个方向上滚动界面，如图 8-2 所示。

在用户滚动界面的过程中，我们要消除用户已经看过的结果，并渲染位于屏幕范围以外的新结果，随时准备展示。这种解决方案一次只渲染 11 个元素，其他数据排队等候，后面再渲染。这种技术称为虚拟化（virtualization，也叫 windowing）。使用这种技术滚动特大型列表，数据量无穷尽时，浏览器不会崩溃。



图 8-2：屏幕窗口外部的内容

构建虚拟化列表组件要考虑很多事情。幸好，我们不用从头开始动手，社区已经开发了很多虚拟化列表组件，直接拿来使用即可。对浏览器渲染来说，最受欢迎的是 `react-window` 和 `react-virtualized`。虚拟化列表十分重要，React Native 甚至自带了一个这样的组件，即 `FlatList`。多数时候，我们无须自己动手构建虚拟化列表组件，知道如何使用就可以了。

为了实现虚拟化列表，我们需要大量数据。这里指的是大量虚拟数据。

```
npm i faker
```

安装 faker 之后，我们可以创建一组大型的虚拟数据。对这个示例来说，我们需要一些虚拟的用户。下面随机创建五千个虚拟用户：

```
import faker from "faker";

const bigList = [...Array(5000)].map(() => ({
  name: faker.name.findName(),
  email: faker.internet.email(),
  avatar: faker.internet.avatar()
}));
```

我们映射一个有五千个空值的数组，把空值替换成虚拟用户的信息，创建 bigList 变量。各用户的 name、email 和 avatar 使用 faker 提供的函数随机生成。

如果使用前一节创建的 List 组件，五千个用户将一次性全部渲染出来：

```
export default function App() {
  const renderItem = item => (
    <div style={{ display: "flex" }}>
      <img src={item.avatar} alt={item.name} width={50} />
      <p>
        {item.name} - {item.email}
      </p>
    </div>
  );

  return <List data={bigList} renderItem={renderItem} />;
}
```

上述代码为每个用户创建一个 div 元素，在这个元素中使用 img 元素渲染用户的头像，在一个段落中渲染用户的 name 和 email，如图 8-3 所示。

React 和现代浏览器是十分优秀的组合，能顺利渲染五千个用户，只是时间稍长一点。具体而言，这个示例耗时 52 ms。随着列表中用户数量的增加，耗时也会增长，最终到达临界点。

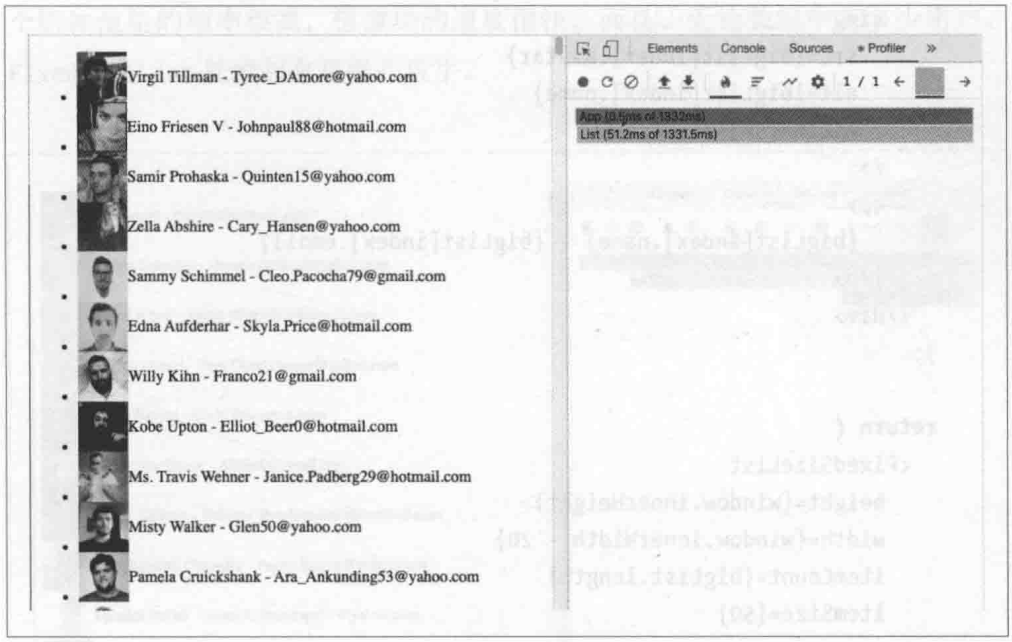

图 8-3：性能结果

下面使用 react-window 渲染这个虚拟用户列表：

```
npm i react-window
```

react-window 库提供了多个用于渲染虚拟列表的组件。在这个示例中，我们使用 react-window 提供的 FixedSizeList 组件：

```
import React from "react";
import { FixedSizeList } from "react-window";
import faker from "faker";

const bigList = [...Array(5000)].map(() => ({
  name: faker.name.findName(),
  email: faker.internet.email(),
  avatar: faker.internet.avatar()
}));

export default function App() {
  const renderRow = ({ index, style }) => (
    <div style={{ ...style, ...{ display: "flex" } }}>
```

```
        <img
          src={bigList[index].avatar}
          alt={bigList[index].name}
          width={50}
        />
        <p>
          {bigList[index].name} - {bigList[index].email}
        </p>
      </div>
    );

    return (
      <FixedSizeList
        height={window.innerHeight}
        width={window.innerWidth - 20}
        itemCount={bigList.length}
        itemSize={50}
      >
        {renderRow}
      </FixedSizeList>
    );
  }
```

FixedSizeList 与我们自己创建的 List 组件稍有不同，我们要提供列表项目的总数，还要通过 itemSize 属性指定每一行占据的像素数。句法上另外一个重要的不同之处是，渲染属性通过 children 属性传给 FixedSizeList。这种渲染属性模式时常用到。

我们来看一下使用 FixedSizeList 组件渲染五千个虚拟用户的结果如何（见图 8-4）。

这一次，没有一次性渲染所有用户，只渲染了用户能看到或者稍微滚动就能看到的用户。注意，首次渲染只用了 2.6 ms。

如果向下滚动，FixedSizeList 会开足马力渲染屏幕范围之外的用户，并把滚动到屏幕外部的用户删掉。这个组件自身就能处理两个方向上的滚动。尽管这

个组件渲染的频率很高，但渲染的速度很快。而且，无论数组中有多少用户，FixedSizeList 处理起来都得心应手。

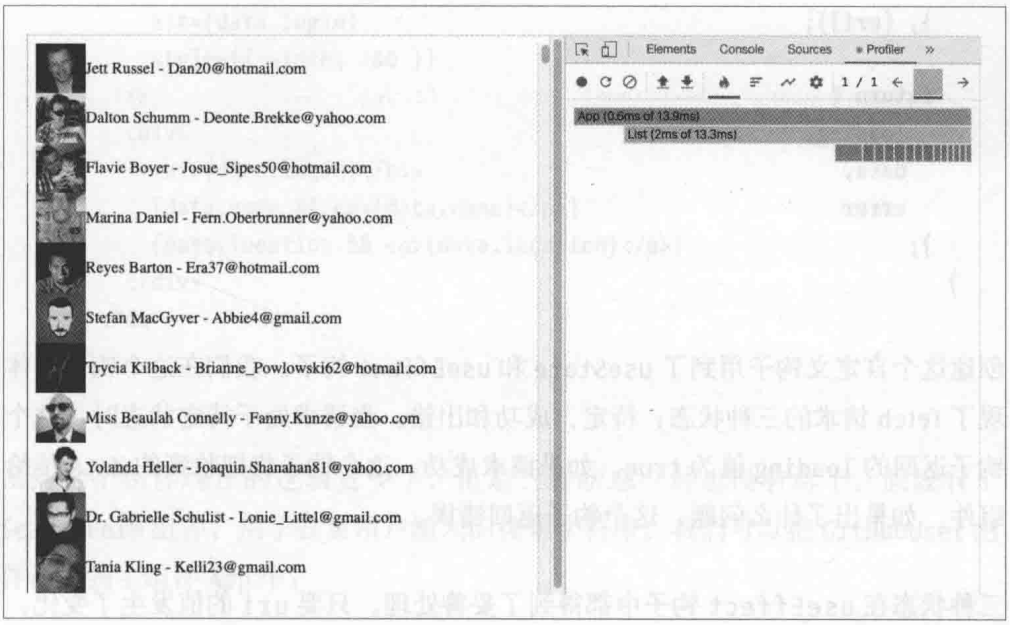

图 8-4：这次渲染耗时 2.6 ms

8.3.1 创建 useFetch 钩子

我们知道，一个请求有三种状态：待定、成功或失败。为了重用 fetch 请求的这个逻辑，我们可以自定义一个钩子，将其命名为 useFetch。在整个应用中，只要想发起 fetch 请求，就可以在组件中使用这个钩子。

```
import React, { useState, useEffect } from "react";

export function useFetch(uri) {
  const [data, setData] = useState();
  const [error, setError] = useState();
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    if (!uri) return;
    fetch(uri)
```

```
      .then(data => data.json())
      .then(setData)
      .then(() => setLoading(false))
      .catch(setError);
  }, [uri]);

  return {
    loading,
    data,
    error
  };
}
```

创建这个自定义钩子用到了 useState 和 useEffect 钩子。我们在这个钩子中体现了 fetch 请求的三种状态：待定、成功和出错。当请求处于待定状态时，这个钩子返回的 loading 值为 true。如果请求成功，这个钩子将把收到的 data 传给组件。如果出了什么问题，这个钩子返回错误。

三种状态在 useEffect 钩子中都得到了妥善处理。只要 uri 的值发生了变化，这个钩子就会被调用。倘若没有 uri，那就不发起 fetch 请求。传入 uri，fetch 请求便开始。如果请求成功，把得到的 JSON 传给 setData 函数，修改 data 的状态值。然后，把 loading 的状态值改为 false，因为请求成功了（即不再是待定状态）。最后，如果出了什么问题，捕获问题，传给 setError，修改 error 的状态值。

现在，我们可以在组件中使用这个钩子发起 fetch 请求了。只要 loading、data 或 error 的值发生了变化，GitHubUser 组件就会使用新值重新渲染。

```
function GitHubUser({ login }) {
  const { loading, data, error } = useFetch(
    `https://api.github.com/users/${login}`
  );

  if (loading) return <h1>loading...</h1>;
  if (error)
    return <pre>{JSON.stringify(error, null, 2)}</pre>;
```

```
  return (
    <div className="githubUser">
      <img
        src={data.avatar_url}
        alt={data.login}
        style={{ width: 200 }}
      />
      <div>
        <h1>{data.login}</h1>
        {data.name && <p>{data.name}</p>}
        {data.location && <p>{data.location}</p>}
      </div>
    </div>
  );
}
```

虽然这个组件现在的逻辑更少了，但是三种状态一种也没有落下。假设有个 SearchForm 组件，用于收集用户输入的搜索字符串，我们可以把 GitHubUser 组件添加到主组件 App 中：

```
import React, { useState } from "react";
import GitHubUser from "./GitHubUser";
import SearchForm from "./SearchForm";

export default function App() {
  const [login, setLogin] = useState("moontahoe");

  return (
    <>
      <SearchForm value={login} onSearch={setLogin} />
      <GitHubUser login={login} />
    </>
  );
}
```

主组件 App 在状态中存储 GitHub 用户的用户名。若想修改这个值，只能使用搜索表单搜索一个新用户。只要 login 的值发生了变化，发给 useFetch 的值随之改变，因为 URL 是由 login 属性构成的：*https://api.github.com/users/${login}*。

钩子中的 uri 发生了变化就会触发 fetch 请求，获取新用户的信息。我们自定义了一个钩子，使用它成功构建了一个小型应用，查找并显示 GitHub 用户的详情。在迭代这个应用的过程中，我们将继续使用这个钩子。

8.3.2 创建 Fetch 组件

钩子最大的作用是跨组件重用功能。有时，在不同的组件中需要重复渲染同样的内容。比如说，选定一种加载旋转框之后，在整个应用中，当 fetch 请求处于待定状态时，我们都希望渲染这个旋转框。一个应用中，fetch 请求的错误处理方式或许也应该保持一致。

与其在多个组件中重复编写相同的代码，不如创建一个组件，在整个域名下都使用相同的方式渲染加载旋转框，并采用一致的方式处理错误。下面来创建一个 Fetch 组件。

```
function Fetch({
  uri,
  renderSuccess,
  loadingFallback = <p>loading...</p>,
  renderError = error => (
    <pre>{JSON.stringify(error, null, 2)}</pre>
  )
}) {
  const { loading, data, error } = useFetch(uri);
  if (loading) return loadingFallback;
  if (error) return renderError(error);
  if (data) return renderSuccess({ data });
}
```

自定义钩子 useFetch 是一层抽象，抽离了发起 fetch 请求的机制。Fetch 组件又是一层抽象，抽离了处理渲染什么的机制。请求正在加载时，Fetch 组件渲染传给可选属性 loadingFallback 的组件。如果请求成功，把 JSON 响应数据传给 renderSuccess 属性。如果有错误，使用可选属性 renderError 渲染。loadingFallback 和 renderError 属性是可选的，提供了一种自定义机制。如果没有提供这两个属性，则回落到默认值。

有了 Fetch 组件，我们可以极大地简化 GitHubUser 组件的逻辑：

```
import React from "react";
import Fetch from "./Fetch";

export default function GitHubUser({ login }) {
  return (
    <Fetch
      uri={`https://api.github.com/users/${login}`}
      renderSuccess={UserDetails}
    />
  );
}

function UserDetails({ data }) {
  return (
    <div className="githubUser">
      <img
        src={data.avatar_url}
        alt={data.login}
        style={{ width: 200 }}
      />
      <div>
        <h1>{data.login}</h1>
        {data.name && <p>{data.name}</p>}
        {data.location && <p>{data.location}</p>}
      </div>
    </div>
  );
}
```

GitHubUser 组件接收的 login 是要在 GitHub 中查找的用户，传给 Fetch 组件的 uri 属性由这个 login 构成。如果请求成功，渲染 UserDetails 组件。Fetch 组件正在加载中时，显示默认的“loading...”消息。倘若出了什么问题，则显示错误详情。

我们也可以为这两个属性提供自定义的值。下述示例体现了这个组件的灵活性：

```
<Fetch
  uri={`https://api.github.com/users/${login}`}
  loadingFallback={<LoadingSpinner />}
  renderError={error => {
    // handle error
    return <p>Something went wrong... {error.message}</p>;
  }}
  renderSuccess={({ data }) => (
    <>
      <h1>Todo: Render UI for data</h1>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </>
  )}
/>
```

这一次，Fetch 组件将渲染我们指定的加载旋转框。如果什么地方出错了，我们隐藏错误详情。请求成功时，渲染原始数据，并显示一个 TODO 消息，提醒自己。

注意，额外的抽象，不管是钩子还是组件，都可能增加代码的复杂度。作为开发人员，我们要尽量减少复杂度。然而，在这个例子中，我们把可重用的逻辑抽取到组件和钩子中，其实降低了复杂度。

8.3.3 处理多个请求

一旦开始从互联网中请求数据，我们会乐此不疲。通常，我们要发起多个 HTTP 请求才能得到应用所需的全部数据。比如说，目前我们让 GitHub 提供了用户的账户信息。除此之外，我们可能还需要获取用户的仓库信息。这两个数据点要通过不同的 HTTP 请求获取。

GitHub 用户一般都有多个仓库，返回的仓库信息是一个对象数组。下面创建一个名为 useIterator 的自定义钩子，迭代任意类型的对象数组：

```
export const useIterator = (
  items = [],
  initialIndex = 0
) => {
  const [i, setIndex] = useState(initialIndex);
```

```
  const prev = () => {
    if (i === 0) return setIndex(items.length - 1);
    setIndex(i - 1);
  };

  const next = () => {
    if (i === items.length - 1) return setIndex(0);
    setIndex(i + 1);
  };

  return [items[i], prev, next];
};
```

使用这个钩子可以遍历任何数组。由于这个钩子返回数组中的元素，因此我们可以利用数组析构为值提供有意义的名称：

```
const [letter, previous, next] = useIterator([
  "a",
  "b",
  "c"
]);
```

这里，letter 的初始值是“a”。如果调用 next，组件将重新渲染，letter 的值变成“b”。再调用两次 next，letter 的值又变成“a”，因为这个迭代器不会让 index 超出界限，绕了一圈又回到了数组的第一个元素。

useIterator 钩子接受 items 数组和初始索引。这个迭代器钩子的关键值是由 useState 钩子创建的索引 i。我们使用 i 标识数组中的当前元素。这个钩子返回当前元素 item[i]，以及迭代数组的两个函数：prev 和 next。这两个函数调用 setIndex，分别递减和递增 i 的值，导致钩子使用新的 index 值重新渲染。

8.3.4 备忘值

useIterator 钩子实现的效果很棒。不过，我们可以做得更好，备忘 item 的值及 prev 和 next 两个函数。

```
import React, { useCallback, useMemo } from "react";
```

```
export const useIterator = (
  items = [],
  initialValue = 0
) => {
  const [i, setIndex] = useState(initialValue);

  const prev = useCallback(() => {
    if (i === 0) return setIndex(items.length - 1);
    setIndex(i - 1);
  }, [i]);

  const next = useCallback(() => {
    if (i === items.length - 1) return setIndex(0);
    setIndex(i + 1);
  }, [i]);

  const item = useMemo(() => items[i], [i]);

  return [item || items[0], prev, next];
};
```

这里，prev 和 next 都使用 useCallback 钩子创建。这样可以确保 prev 函数始终保持不变，除非 i 的值发生变化。同样，item 的值也始终指向同一个元素对象，除非 i 的值发生变化。

备忘这些值不会大幅提升性能，至少不足以让我们下决心增加代码的复杂度。然而，使用 useIterator 钩子时，备忘的值始终指向相同的对象和函数，这样方便比较值或者在依赖数组中使用。

现在，我们来创建一个仓库目录组件。在这个组件中，我们将使用 useIterator 钩子，让用户遍览仓库列表：

```
< learning-react >
```

用户单击“>”按钮，会看到下一个仓库的名称。类似的，单击“<”按钮，会看到前一个仓库的名称。下面是实现这个功能的 RepoMenu 组件。

```
import React from "react";
import { useIterator } from "../hooks";

export function RepoMenu({
  repositories,
  onSelect = f => f
}) {
  const [{ name }, previous, next] = useIterator(
    repositories
  );

  useEffect(() => {
    if (!name) return;
    onSelect(name);
  }, [name]);

  return (
    <div style={{ display: "flex" }}>
      <button onClick={previous}>&lt;</button>
      <p>{name}</p>
      <button onClick={next}>&gt;</button>
    </div>
  );
}
```

RepoMenu 组件通过 repositories 属性接收仓库列表，使用 useIterator 钩子从中析构出当前仓库对象的 name 及 previous 和 next 函数。< 是“小于号”的实体，显示一个小于号“<”。类似地，> 是大于号。这两个符号表示前一个和下一个，用户点击后，组件将使用新的仓库名称重新渲染。如果 name 变了，说明用户选择了不同的仓库，那就调用 onSelect 函数，通过参数传入新仓库的名称。

注意，使用数组析构可以为数组元素任意命名。即使在钩子中我们把函数命名为 prev 和 next，但是在使用钩子时，我们可以修改名称，改为 previous 和 next。

现在可以创建 UserRepositories 组件了。这个组件先请求 GitHub 用户的仓库列表，然后把列表传给 RepoMenu 组件：

```
import React from "react";
import Fetch from "./Fetch";
import RepoMenu from "./RepoMenu";

export default function UserRepositories({
  login,
  selectedRepo,
  onSelect = f => f
}) {
  return (
    <Fetch
      uri={`https://api.github.com/users/${login}/repos`}
      renderSuccess={({ data }) => (
        <RepoMenu
          repositories={data}
          selectedRepo={selectedRepo}
          onSelect={onSelect}
        />
      )}
    />
  );
}
```

我们要为 UserRepositories 组件提供 login 属性，指明请求哪个用户的仓库列表。传给 Fetch 组件的 URI 也由这个 login 构成。成功处理 fetch 请求后，通过 data 把 Fetch 组件返回的仓库列表传给 RepoMenu 组件渲染结果。当用户选择其他仓库时，我们把新仓库的名称传给父级对象。

```
function UserDetails({ data }) {
  return (
    <div className="githubUser">
      <img src={data.avatar_url} alt={data.login} style={{ width: 200 }} />
      <div>
        <h1>{data.login}</h1>
        {data.name && <p>{data.name}</p>}
        {data.location && <p>{data.location}</p>}
      </div>
      <UserRepositories
        login={data.login}
```

```
        onSelect={repoName => console.log(`${repoName} selected`)}
      />
    </div>
  );
}
```

现在，我们要把新组件添加到 `UserDetails` 组件中。渲染 `UserDetails` 组件时会同时渲染用户的仓库列表。假设 `login` 的值为 `eveporcello`，上述组件的渲染结果如图 8-5 所示。

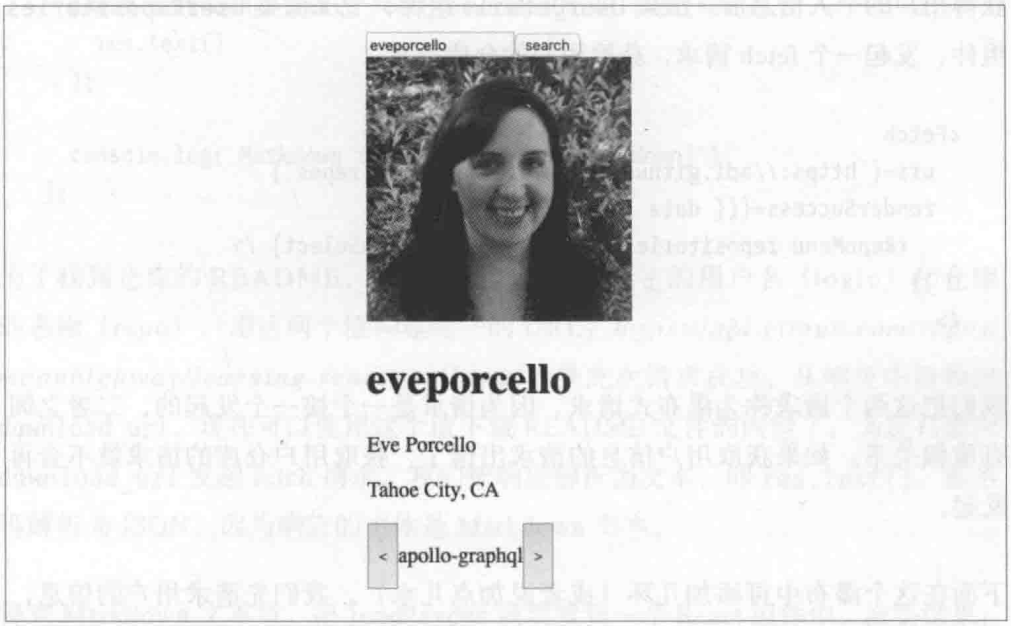

图 8-5：输出仓库名称

为了同时获得 Eve 账户的信息和仓库列表，我们要发送两个独立的 HTTP 请求。作为 React 开发者，我们的生活基本上是这样度过的：发起多个请求获取信息，把收到的信息合理地放到应用精美的用户界面中。发起两个请求获取信息只是一个开始。下一节将向 GitHub 发起更多请求，显示选中仓库的 README.md。

8.3.5 瀑布式请求

前一节发起了两个 HTTP 请求，第一个请求用于获取用户的个人信息，获得之

后又发起第二个请求，获取用户的仓库信息。这两个请求一次发起一个，一个接着一个。

第一个请求在首次获取用户的个人信息时发起：

```
<Fetch
  uri={`https://api.github.com/users/${login}`}
  renderSuccess={UserDetails}
/>
```

获得用户的个人信息后，渲染 UserDetails 组件，它又渲染 UserRepositories 组件，发起一个 fetch 请求，获取用户的仓库信息：

```
<Fetch
  uri={`https://api.github.com/users/${login}/repos`}
  renderSuccess={({ data }) => (
    <RepoMenu repositories={data} onSelect={onSelect} />
  )}
/>
```

我们把这两个请求称为瀑布式请求，因为请求是一个接一个发起的，二者之间有依赖关系。如果获取用户信息的请求出错了，获取用户仓库的请求就不会再发起。

下面在这个瀑布中再添加几环（或者说加点儿水）。我们先请求用户的信息，然后请求用户的仓库列表，获得仓库列表之后，再请求第一个仓库的 README.md 文件。在用户遍览仓库列表的过程中，我们将再次发起请求，获取相应仓库的 README 文件。

仓库的 README 文件使用 Markdown 编写，这是一种文本格式，可使用 ReactMarkdown 组件渲染为 HTML。先来安装 react-markdown 包：

```
npm i react-markdown
```

请求仓库的 README 文件内容也涉及瀑布式请求。首先，要向仓库 README 文件的路由（*https://api.github.com/repos/${login}/${repo}/*）发起数据请求。

GitHub对这个路由的响应是关于仓库README文件的详情，而不是文件的内容。不过，响应中有个 download_url，我们可以通过它请求 README 文件的内容。可见，为了获取 Markdown 内容，要多发起一次请求。这两个请求可以在同一个异步函数中发起：

```
const loadReadme = async (login, repo) => {
  const uri = `https://api.github.com/repos/${login}/${repo}/readme`;
  const { download_url } = await fetch(uri).then(res =>
    res.json()
  );
  const markdown = await fetch(download_url).then(res =>
    res.text()
  );

  console.log(`Markdown for ${repo}\n\n${markdown}`);
};
```

为了找到仓库的 README，我们需要知道仓库属主的用户名（login）和仓库的名称（repo），用这两个值构建唯一的 URL：*https://api.github.com/repos/moonhighway/learning-react/readme*。如果此次请求成功，从响应中析构出 download_url。现在可以使用这个值下载 README 文件的内容了，为此只需向 download_url 发起 fetch 请求。我们把响应解析为文本，即 res.text()，而不再解析为 JSON，因为响应的主体是 Markdown 文本。

得到 Markdown 文本后，把 loadReadme 函数放到一个 React 组件中，渲染结果：

```
import React, {
  useState,
  useEffect,
  useCallback
} from "react";
import ReactMarkdown from "react-markdown";

export default function RepositoryReadme({ repo, login }) {
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState();
  const [markdown, setMarkdown] = useState("");
```

```
  const loadReadme = useCallback(async (login, repo) => {
    setLoading(true);
    const uri = `https://api.github.com/repos/${login}/${repo}/readme`;
    const { download_url } = await fetch(uri).then(res =>
      res.json()
    );
    const markdown = await fetch(download_url).then(res =>
      res.text()
    );
    setMarkdown(markdown);
    setLoading(false);
  }, []);

  useEffect(() => {
    if (!repo || !login) return;
    loadReadme(login, repo).catch(setError);
  }, [repo]);

  if (error)
    return <pre>{JSON.stringify(error, null, 2)}</pre>;
  if (loading) return <p>Loading...</p>;

  return <ReactMarkdown source={markdown} />;
}
```

首先，使用 useCallback 钩子把 loadReadme 函数添加到组件中，在首次渲染组件时备忘该函数。这个函数在发起 fetch 请求前把 loading 状态设为 true，请求结束后又设回 false。收到 Markdown 文本后，使用 setMarkdown 函数把文本保存到状态中。

接下来要调用 loadReadme，因此我们添加了一个 useEffect 钩子，在首次渲染组件后加载 README 文件。倘若出于什么原因，repo 和 login 属性没有值，那就不加载 README 文件。这个钩子的依赖数组是 [repo]。这表示，repo 值发生变化后我们想加载另一个 README 文件。在加载 README 文件的过程中如果出错了，捕获错误，发给 setError 函数。

注意，我们要处理每个 fetch 请求都有的三种状态：待定、成功和失败。最后，成功获得响应后，使用 ReactMarkdown 组件渲染 Markdown 文本。

现在，剩下的工作是在 RepoMenu 组件中渲染 RepositoryReadme 组件。用户使用 RepoMenu 组件遍览仓库时，同时还会加载并显示各仓库的 README 文件。

```
export function RepoMenu({ repositories, login }) {
  const [{ name }, previous, next] = useIterator(
    repositories
  );
  return (
    <>
      <div style={{ display: "flex" }}>
        <button onClick={previous}>&lt;</button>
        <p>{name}</p>
        <button onClick={next}>&gt;</button>
      </div>
      <RepositoryReadme login={login} repo={name} />
    </>
  );
}
```

现在，这个应用真的发起多个请求了。一开始，发起四个请求：一个请求用户的个人信息，一个请求用户的仓库列表，再有一个请求选中的仓库的 README 文件的信息，最后一个请求 README 文件的文本内容。这四个是瀑布式请求，一个接一个发起。

此外，用户与应用交互的过程中还会发起更多请求。用户每次改变当前仓库都会发起两个瀑布式请求，获取对应的 README 文件。每次改变要搜索的 GitHub 用户，最初的四个瀑布式请求都会重新发起。

8.3.6 限制网络速度

所有请求都可在开发者工具的“Network”标签页中查看。在这个标签页中，你可以查看每一个请求，还可以限制网络速度，考察请求在较慢的网速下表现如何。

如果你想查看瀑布式请求是如何一个接一个发出的，可以降低网络速度，查看每个请求渲染的加载消息。

多数主流浏览器的开发者工具都有“Network”标签页。在 Google Chrome 中如果想限制网络速度，单击“Online”旁边的箭头，如图 8-6 所示。

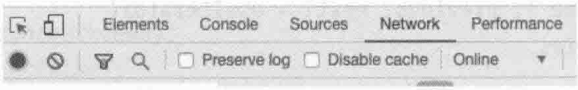

图 8-6：修改网络请求的速度

在打开的菜单中，你可以选择不同的速度，如图 8-7 所示。

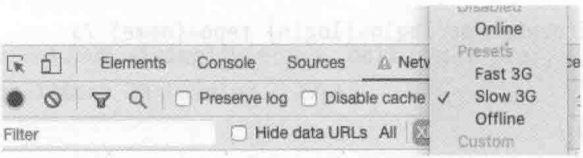

图 8-7：选择网络请求的速度

选择“Fast 3G”或“Slow 3G”对网络请求的速度限制较大。

另外，“Network”标签页还显示有全部 HTTP 请求的时间线。你可以筛选时间线，只查看“XHR”请求，即只显示 fetch 发出的请求（见图 8-8）。



图 8-8：瀑布式请求

可以看到，四个请求是一个接一个发起的。注意，加载图的标题是“Waterfall”，这表明后一个请求是在前一个请求结束后发起的。

8.3.7 并行请求

有时候，可以一次发送全部请求，提升应用的速度。这种情况下，不再像瀑布式那样一个接一个发起请求，而是并行（或同时）发送所有请求。

这个应用现在之所以发送瀑布式请求，是因为组件是一个套一个渲染的，GitHubUser 最终渲染 UserRepositories，而 UserRepositories 最终又渲染 RepositoryReadme。前一个组件未渲染之前不能发起下一个请求。

若想并行发起请求，要修改编程方案。首先，要从 RepoMenu 组件的渲染函数中删除 <RepositoryReadme /> 。这是一次不错的改动。RepoMenu 组件应该专注于实现创建仓库目录的逻辑，供用户遍览。RepositoryReadme 组件应该放到其他组件中渲染。

接下来，要从 UserRepositories 组件的 renderSuccess 属性中删除 <RepoMenu />。同样，要从 UserDetails 组件中删除 <UserRepositories />。

我们要把层层嵌套的组件放在同一级，都放在 App 组件中：

```
import React, { useState } from "react";
import SearchForm from "./SearchForm";
import GitHubUser from "./GitHubUser";
import UserRepositories from "./UserRepositories";
import RepositoryReadme from "./RepositoryReadme";

export default function App() {
  const [login, setLogin] = useState("moonhighway");
  const [repo, setRepo] = useState("learning-react");
  return (
    <>
      <SearchForm value={login} onSearch={setLogin} />
      <GitHubUser login={login} />
      <UserRepositories
```

```
          login={login}
          repo={repo}
          onSelect={setRepo}
        />
        <RepositoryReadme login={login} repo={repo} />
      </>
    );
  }
```

GitHubUser、UserRepositories 和 RepositoryReadme 组件都向 GitHub 发送 HTTP 请求，获取数据。在同一级渲染这三个组件，所有请求同时并行发起。

为了发起请求，每一个组件都需要特定的信息。获取 GitHub 用户信息需要 login，获取用户的仓库列表也需要 login。RepositoryReadme 要有 login 和 repo 才能正常运行。为了保证每个组件都有发起请求所需的信息，初始化应用时指定显示“moonhighway”用户和“learning-react”仓库的信息。

如果用户在 SearchForm 中搜索另一个 GitHub 用户，login 的值将发生变化，从而触发组件中的 useEffect 钩子，致使组件再次发起请求，获取相应的数据。用户在遍览仓库列表的过程中将调用 UserRepositories 组件的 onSelect 属性，导致 repo 值发生变化。repo 值发生变化将触发 RepositoryReadme 组件中的 useEffect 钩子，请求一个新的 README 文件。

无论如何，RepoMenu 组件始终从第一个仓库开始。我们要判断 selectedRepo 属性是否存在，如果存在，通过它查找最初要显示的仓库的索引：

```
export function RepoMenu({ repositories, selected, onSelect = f => f }) {
  const [{ name }, previous, next] = useIterator(
    repositories,
    selected ? repositories.findIndex(repo => repo.name === selected) : null
  );
  ...
}
```

useIterator 钩子的第二个参数是起始索引。如果有 selected 属性，就通过

name 查找被选中仓库的索引。这么做是为了确保仓库目录一开始能显示正确的仓库名称。UserRepositories 组件也要把 selected 属性传给这个组件：

```
<Fetch
  uri={`https://api.github.com/users/${login}/repos`}
  renderSuccess={({ data }) => (
    <RepoMenu
      repositories={data}
      selected={repo}
      onSelect={onSelect}
    />
  )}
/>
```

现在，我们把 repo 属性向下传给了 RepoMenu 组件，仓库目录应该选中起始仓库，即我们指定的“learning-react”。

打开“Network”标签页，你会看到三个请求是并行的，如图 8-9 所示。

Name	Status	Type	Initiator	Size	Time	Waterfall
moonhighway	200	fetch	hooks.js:11	1.8 KB	2.22 s	
repos	200	fetch	hooks.js:11	14.7 KB	2.57 s	
readme	200	fetch	RepositoryRe...	3.4 KB	2.22 s	
README.md	200	fetch	RepositoryRe...	2.0 KB	2.11 s	

图 8-9：并行发起请求

如此一来，各个组件将同时发起请求。不过，RepoReadme 组件依然通过瀑布式请求获取 README 文件的内容。这没什么问题，毕竟首次渲染应用时很难同时发起全部请求。并行请求和瀑布式请求可以并存。

8.3.8 等待值出现

目前，我们把 login 和 repo 的初始值设为“moonhighway”和“learning-react”。可是，我们并不能始终猜测出一开始要渲染什么数据。遇到这种情况，我们干脆不渲染组件，等到获得所需的数据再渲染。

```
export default function App() {
  const [login, setLogin] = useState();
  const [repo, setRepo] = useState();
  return (
    <>
      <SearchForm value={login} onSearch={setLogin} />
      {login && <GitHubUser login={login} />}
      {login && (
        <UserRepositories
          login={login}
          repo={repo}
          onSelect={setRepo}
        />
      )}
      {login && repo && (
        <RepositoryReadme login={login} repo={repo} />
      )}
    </>
  );
}
```

这里，在所需的属性获得值之前，不渲染对应的组件。一开始，只渲染SearchForm组件。搜索某个用户后，login的值发生变化，渲染UserRepositories组件。这个组件查找仓库时，选中列表中的第一个仓库，从而调用setRepo。最后，我们获得了login和repo，此时渲染RepositoryReadme组件。

8.3.9 取消请求

深入思考一番，你会意识到用户或许会清空搜索字段，不搜索任何用户。这种情况下，我们要确保repo的值也是空的。下面添加一个handleSearch方法，确保login没有值时repo值随之变化。

```
export default function App() {
  const [login, setLogin] = useState("moonhighway");
  const [repo, setRepo] = useState("learning-react");

  const handleSearch = login => {
    if (login) return setLogin(login);
```

```
    setLogin("");
    setRepo("");
  };

  if (!login)
    return (
      <SearchForm value={login} onSearch={handleSearch} />
    );

  return (
    <>
      <SearchForm value={login} onSearch={handleSearch} />
      <GitHubUser login={login} />
      <UserRepositories
        login={login}
        repo={repo}
        onSelect={setRepo}
      />
      <RepositoryReadme login={login} repo={repo} />
    </>
  );
}
```

我们添加了 handleSearch 方法。现在，用户清空搜索字段后，即搜索一个空字符串时，我们也把 repo 的值设为一个空字符串。如果出于什么原因，login 没有值，我们只渲染一个组件，即 SearchForm。当 login 有值时，再渲染全部四个组件。

严格来说，现在我们的应用有两个界面。一个界面只显示一个搜索表单，另一个界面只在搜索表单中有值时显示全部四个组件。这是根据用户的操作选择挂载或卸载组件。假设我们正在查看“moonhighway”的信息。如果用户清空了搜索字段，那么 GitHubUser、UserRepositories 和 RepositoryReadme 组件将被卸载，不再显示。可是，如果卸载时这几个组件正在加载数据呢？

你可以试一下：

1. 把网速限制为“Slow 3G”，留出足够的时间让问题发生。

2. 把搜索字段中的“moonhighway”值改为“eveporcello”。

3. 在加载数据的过程中，搜索的是空字符串 ""。

上述步骤导致 GitHubUser、UserRepositories 和 RepositoryReadme 在发起 fetch 请求的中途被卸载。结果是，当 fetch 请求返回响应时，这三个组件已经卸载完毕了。试图修改已被卸载的组件的状态值会导致如图 8-10 所示的错误。

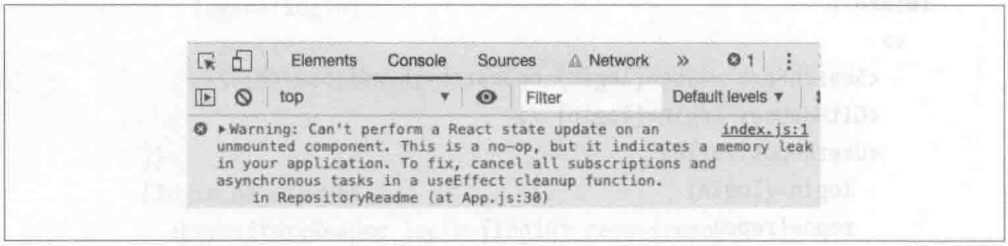

图 8-10：挂载错误

只要用户在较慢的网速下加载数据，就有可能遇到这个错误。其实，我们可以做些防护措施。首先，我们可以创建一个钩子，获知当前的组件有没有挂载：

```
export function useMountedRef() {
  const mounted = useRef(false);
  useEffect(() => {
    mounted.current = true;
    return () => (mounted.current = false);
  });
  return mounted;
}
```

useMountedRef 用到了一个 ref。如果组件被卸载了，状态虽被清除了，但是 ref 还在。上面的 useEffect 没有依赖数组，每一次渲染组件都会调用，确保 ref 的值为 true。只要卸载了组件，调用 useEffect 导致函数返回，ref 的值变成 false。

下面在 RepoReadme 组件中使用这个钩子，确保更新状态之前组件是挂载的。

```
const mounted = useMountedRef();
```

```
const loadReadme = useCallback(async (login, repo) => {
  setLoading(true);
  const uri = `https://api.github.com/repos/${login}/${repo}/readme`;
  const { download_url } = await fetch(uri).then(res =>
    res.json()
  );
  const markdown = await fetch(download_url).then(res =>
    res.text()
  );
  if (mounted.current) {
    setMarkdown(markdown);
    setLoading(false);
  }
}, []);
```

现在，我们可以通过一个 ref 判断组件有没有挂载。这里的两个请求都要花点儿时间才能结束。结束后，在调用 setMarkdown 或 setLoading 之前，我们要检查一下，确保组件还是挂载的。

下面在 useFetch 钩子中添加同样的逻辑：

```
const mounted = useMountedRef();

useEffect(() => {
  if (!uri) return;
  if (!mounted.current) return;
  setLoading(true);
  fetch(uri)
    .then(data => {
      if (!mounted.current) throw new Error("component is not mounted");
      return data;
    })
    .then(data => data.json())
    .then(setData)
    .then(() => setLoading(false))
    .catch(error => {
      if (!mounted.current) return;
      setError(error);
    });
```

useFetch 钩子用于发起应用中其余的 fetch 请求。在这个钩子中，我们通过串接 .then() 函数发起 fetch 请求，而没有使用 async/await。fetch 请求结束后，我们在第一个 .then 回调中检查组件有没有挂载。如果组件是挂载的，返回 data，接着调用余下的 .then 函数。如果组件没有挂载，第一个 .then 函数抛出错误，阻止执行后续的 .then 函数，转而调用 .catch 函数，把新错误传给函数。.catch 函数在调用 setError 之前会检查组件有没有挂载。

这样就取消了请求。我们没有直接阻止发起 HTTP 请求，却保全了请求处理完毕后修改状态的函数调用。在较慢的网速下做测试是有必要的，可以发现并消除此类 bug。

8.4 GraphQL 简介

与 React 一样，GraphQL 也是由 Facebook 设计的。我们知道，React 为构建用户界面提供了一种声明式方案，类似地，GraphQL 为与 API 通信提供一种声明式方案。并行发起数据请求时，我们希望能同时获得所需的全部数据。GraphQL 就是为此而设计的。

为了从 GraphQL API 获取数据，我们仍然要向指定的 URI 发起 HTTP 请求。不过，还要随请求一起发送查询（query）。GraphQL 查询是对想要请求的数据的一种声明式描述。服务负责解析这个描述，把请求的所有数据打包进一个响应中。

8.4.1 GitHub GraphQL API

若想在 React 应用中使用 GraphQL，与之通信的后端服务要按照 GraphQL 规范构建。幸好，GitHub 也开放了 GraphQL API。多数 GraphQL 服务都提供有探索 GraphQL API 的方式。GitHub 提供的是 GraphQL Explorer（*https://developer.github.com/v4/explorer*）。若想使用 GraphQL Explorer，必须使用 GitHub 账户登录。

我们在 GraphQL Explorer 的左侧面板中草拟 GraphQL 查询。在这个面板中，我们可以添加一个查询，获取某个 GitHub 用户的信息：

```
query {
  user(login: "moontahoe") {
    id
    login
    name
    location
    avatarUrl
  }
}
```

这是一个 GraphQL 查询，获取 GitHub 用户“moontahoe”的信息。我们没有获取 moontahoe 的全部公开信息，而是只获取指定的数据：`id`、`login`、`avatarUrl`、`name` 和 `location`。单击页面中的播放按钮后，这个查询将通过一个 HTTP POST 请求发给 *https://api.github.com/graphql*。所有 GitHub GraphQL 查询都发往这个 URI。GitHub 将解析查询，只返回我们请求的数据：

```
{
  "data": {
    "user": {
      "id": "MDQ6VXNlcjU5NTIwODI=",
      "login": "MoonTahoe",
      "name": "Alex Banks",
      "location": "Tahoe City, CA",
      "avatarUrl": "https://github.com/moontahoe.png"
    }
  }
}
```

根据这个 GraphQL 查询，我们可以编写一个可重用的操作，命名为 `findRepos`。只要想查找用户及其仓库有关的信息，我们就可以使用这个查询，传入 `login` 变量。

```
query findRepos($login: String!) {
  user(login: $login) {
    login
    name
    location
    avatar_url: avatarUrl
```

```
      repositories(first: 100) {
        totalCount
        nodes {
          name
        }
      }
    }
  }
```

我们定义了一个可重用的 `findRepos` 查询，只需传入 `$login` 变量的值。这个变量的值可在 Query Variables 面板中设置，如图 8-11 所示。

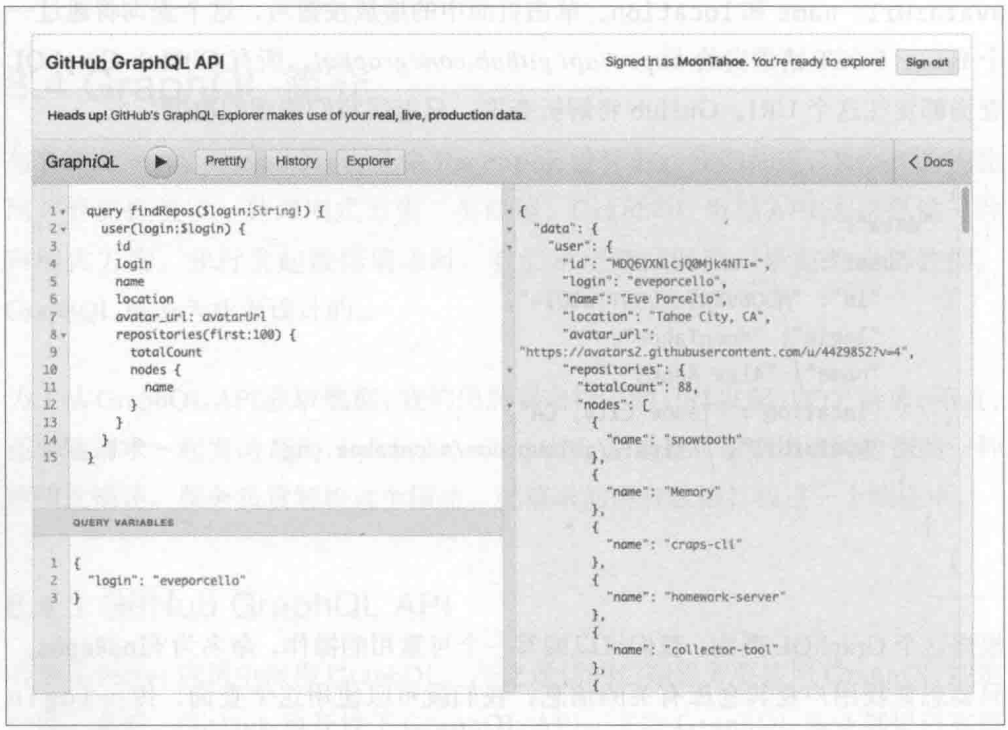

图 8-11：GitHub GraphQL Explorer

除了获取用户有关的信息之外，我们还请求了用户的前 100 个仓库。我们不仅获取了查询返回的仓库数量，还获取了每个仓库的名称（`name`）。GraphQL 只返回我们请求的数据。这里，我们仅获取各仓库的名称，其他信息都不要。

此外，我们还对查询做了一处变动：为 avatarUrl 指定别名。获取用户头像的 GraphQL 字段是 avatarUrl，但我们希望变量的名称是 avatar_url。指定别名后，GitHub 返回的响应数据将重命名对应的字段。

GraphQL 相关的话题还有很多，笔者专门写了一本书，名为《Learning GraphQL》。本节只介绍了一点皮毛知识，GraphQL 发展势头良好，是开发人员必备的一项技能。在 21 世纪，若想在开发上取得成就，势必要掌握 GraphQL 的基本用法。

8.4.2 发起一个 GraphQL 请求

GraphQL 请求是在请求主体中包含查询的 HTTP 请求。我们可以使用 fetch 发起 GraphQL 请求。此外，也有一些专门的库和框架能辅助我们处理这类请求各方面的细节。本节说明如何使用 graphql-request 库为应用输送 GraphQL 数据。



GraphQL 不限于 HTTP。GraphQL 是一个规范，规定如何通过网络发起数据请求。理论上，GraphQL 适用于任何网络协议。而且，GraphQL 不限定必须使用某种编程语言。

先来安装 graphql-request 库：

```
npm i graphql-request
```

GitHub 的 GraphQL API 要求客户端应用在发送请求时验证身份。为了完成下文的示例，你必须从 GitHub 获取一个个人访问令牌，而且必须在每个请求中发送这个令牌。

若想获取供 GraphQL 请求使用的个人访问令牌，依次单击“Settings → Developer Settings → Personal Access Tokens”。在表单中，我们可以创建一个具有指定权限的访问令牌。为了发起 GraphQL 查询，令牌必须具有下述读权限：

- user

- public_repo
- repo
- repo_deployment
- repo:status
- read:repo_hook
- read:org
- read:public_key
- read:gpg_key

然后，在 JavaScript 应用中使用 graphql-request 库发起 GraphQL 请求：

```
import { GraphQLClient } from "graphql-request";

const query = `
  query findRepos($login:String!) {
    user(login:$login) {
      login
      name
      location
      avatar_url: avatarUrl
      repositories(first:100) {
        totalCount
        nodes {
          name
        }
      }
    }
  }
`;

const client = new GraphQLClient(
  "https://api.github.com/graphql",
  {
    headers: {
      Authorization: `Bearer <PERSONAL_ACCESS_TOKEN>`
```

```
    }
  }
);

client
  .request(query, { login: "moontahoe" })
  .then(results => JSON.stringify(results, null, 2))
  .then(console.log)
  .catch(console.error);
```

我们使用 graphql-request 库中的 GraphQLClient 构造函数发送这个请求。创建这个客户端时，我们指定使用 GitHub GraphQL API 的 URI，即 *https://api.github.com/graphql*。此外，我们还发送了一个首部，指定个人访问令牌。这个令牌用于识别我们的身份，根据要求，使用 GitHub GraphQL API 必须提供这个令牌。现在可以使用 client 发起 GraphQL 请求了。

发起 GraphQL 请求需要提供 query，即前面定义的 GraphQL 查询。我们把 query 及查询所需的变量传给 request 函数。这里，query 需要一个名为 $login 的变量，因此我们传入一个对象，在 login 字段中指定 $login 的值。

这里，我们只是把得到的 JSON 转换成字符串，输出到控制台中：

```
{
  "user": {
    "id": "MDQ6VXNlcjU5NTIwODI=",
    "login": "MoonTahoe",
    "name": "Alex Banks",
    "location": "Tahoe City, CA",
    "avatar_url": "https://avatars0.githubusercontent.com/u/5952082?v=4",
    "repositories": {
      "totalCount": 52,
      "nodes": [
        {
          "name": "snowtooth"
        },
        {
          "name": "Memory"
        },
```

```
        {
          "name": "snowtooth-status"
        },

        ...

      ]
    }
  }
}
```

与 fetch 一样，client.request 也返回一个 promise。在 React 组件中获取这个数据与使用 fetch 从路由上获取数据的过程十分类似：

```
export default function App() {
  const [login, setLogin] = useState("moontahoe");
  const [userData, setUserData] = useState();
  useEffect(() => {
    client
      .request(query, { login })
      .then(({ user }) => user)
      .then(setUserData)
      .catch(console.error);
  }, [client, query, login]);

  if (!userData) return <p>loading...</p>;

  return (
    <>
      <SearchForm value={login} onSearch={setLogin} />
      <UserDetails {...userData} />
      <p>{userData.repositories.totalCount} - repos</p>
      <List
        data={userData.repositories.nodes}
        renderItem={repo => <span>{repo.name}</span>}
      />
    </>
  );
}
```

我们在一个 `useEffect` 钩子中调用 `client.request` 发起请求。如果 `client`、`query` 或者 `login` 发生了变化，`useEffect` 钩子将发起一个新请求。然后，让 React 渲染得到的 JSON，如图 8-12 所示。

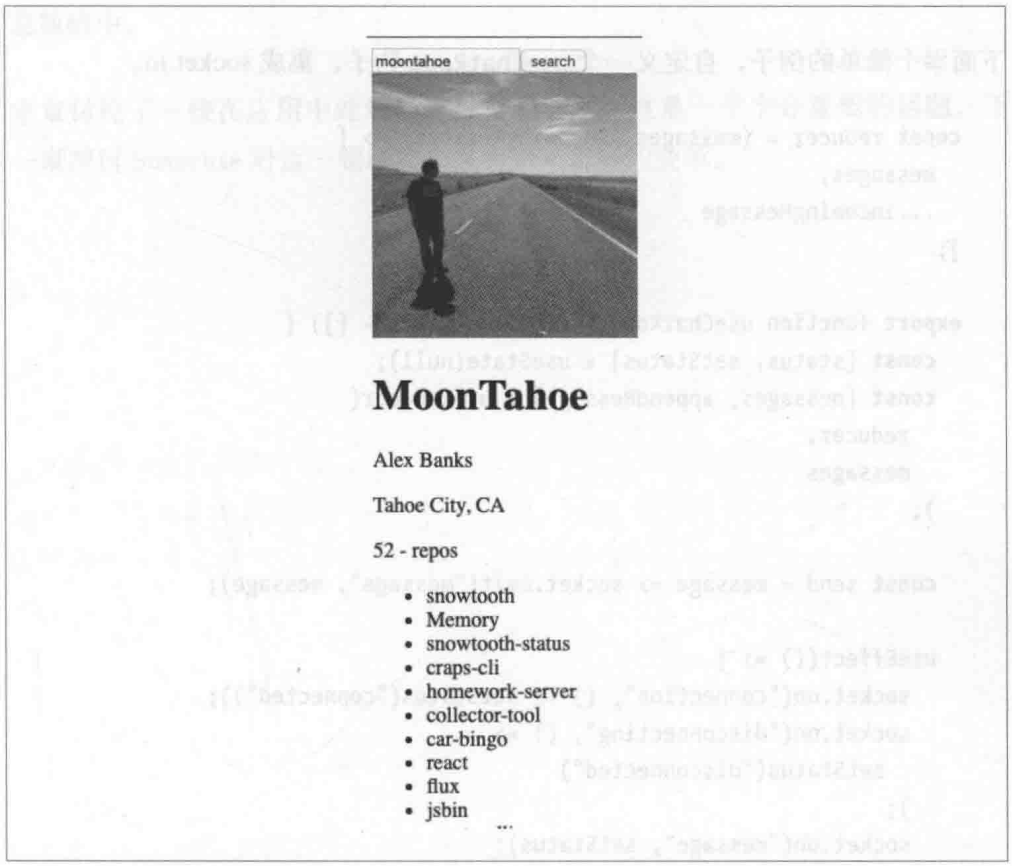

图 8-12：GraphQL 应用

这个示例没有处理 `loading` 和 `error` 状态，不过本章所学的知识都可以应用到 GraphQL 上。React 并不关心你是如何获得数据的，只要知道在组件中怎样处理 promise 等异步对象即可。

从互联网中加载数据是一项异步任务。获取数据的请求要耗费一定时间传送，说不定什么地方会出错。我们要掌握如何在 React 组件中使用 `useEffect` 钩子处理 promise 的 `pending`、`success` 和 `fail` 三个状态。

本章用了大量篇幅讲解 promise、fetch 和 HTTP，这是因为 HTTP 依然是从互联网中请求数据最为流行的方式，而 promise 又特别适合用来处理 HTTP 请求。有时，你可能需要使用其他协议，比如说 WebSockets。不用担心，有状态的钩子和 useEffect 也能处理。

下面举个简单的例子，自定义一个 useChatRoom 钩子，集成 socket.io。

```
const reducer = (messages, incomingMessage) => [
  messages,
  ...incomingMessage
];

export function useChatRoom(socket, messages = []) {
  const [status, setStatus] = useState(null);
  const [messages, appendMessage] = useReducer(
    reducer,
    messages
  );

  const send = message => socket.emit("message", message);

  useEffect(() => {
    socket.on("connection", () => setStatus("connected"));
    socket.on("disconnecting", () =>
      setStatus("disconnected")
    );
    socket.on("message", setStatus);
    return () => {
      socket.removeAllListeners("connect");
      socket.removeAllListeners("disconnect");
      socket.removeAllListeners("message");
    };
  }, []);

  return {
    status,
    messages,
    send
  };
}
```

这个钩子提供一个聊天消息数组、websocket 连接状态，以及一个把新消息广播给套接字（socket）的函数。这些值均受该 `useEffect` 钩子中定义的监听程序（listener）的影响。当套接字发出 `connection` 或 `disconnecting` 事件时，`status` 的值将发生变化。收到新消息时，新消息通过 `useReducer` 钩子追加到消息数组中。

本章讨论了一些在应用中处理异步数据的技术，这是一个十分重要的话题。下一章探讨 Suspense 对这一领域的未来将带来怎样的变革。

第9章

Suspense

这是本书最不重要的一章，至少React团队是这样跟笔者说的。他们没有直接挑明，“这是最不重要的一章，不要写了。”React团队只是发了一系列推文，提醒教育工作者和布道者，他们在这一领域所做的工作很快就要过时。一切都将发生变化。

可以说，React团队在Fiber、Suspense和并发模式方面所做的工作代表了Web开发的未来。这项工作可能会改变浏览器解释JavaScript的方式。听起来这是一项十分重要的工作。之所以说这是本书最不重要的一章，是因为社区对Suspense的炒作热情太高，笔者必须点醒你，适当降低你的期待。Suspense涵盖的API和模式并不是唯一的主导性理论，不能定义世间万物的运作机制。

Suspense只是一个特性，你可能永远也用不到。设计Suspense的目的是解决Facebook在弹性伸缩中遇到的特定问题。不是所有人都有和Facebook一样的问题，所以在寻求使用这些工具解决我们的问题之前一定要三思，说不定会把简单的问题复杂化。另外，一切都在改变。并发模式是一个实验性功能，React团队已经发出严厉警告，奉劝大家不要在生产环境中使用。事实上，Suspense相关的概念大都涉及使用钩子。如果你在开发中不是频繁自定义钩子，可能永远不需要了解这些特性。Suspense涉及的多数机制都可以通过钩子抽象出来。

前三段虽然说了一些泄气的话，但是本章要讲的概念还是很令人兴奋的。如果使用得当，有朝一日这些概念可以帮助我们创造更好的用户体验。如果你拥有或维护着 React 钩子和（或）组件库，或许你会发现这些概念的价值所在。我们可以借助这些概念细致调整自定义钩子，改善反馈和优先顺序。

本章将再构建一个小型应用，演示 Suspense 的部分特性。其实，我们是要重新构建第 8 章那个应用，不过这一次将增加一些结构。例如，我们将使用一个 SiteLayout 组件：

```
export default function SiteLayout({
  children,
  menu = c => null
}) {
  return (
    <div className="site-container">
      <div>{menu}</div>
      <div>{children}</div>
    </div>
  );
}
```

SiteLayout 在 App 组件中渲染，帮助我们构建 UI：

```
export default function App() {
  return (
    <SiteLayout menu={<p>Menu</p>}>
      <>
        <Callout>Callout</Callout>
        <h1>Contents</h1>
        <p>This is the main part of the example layout</p>
      </>
    </SiteLayout>
  );
}
```

这个组件为布局提供一些样式，如图 9-1 所示。

具体而言，借助这个组件可以清楚地看出各组件在何处及何时渲染。

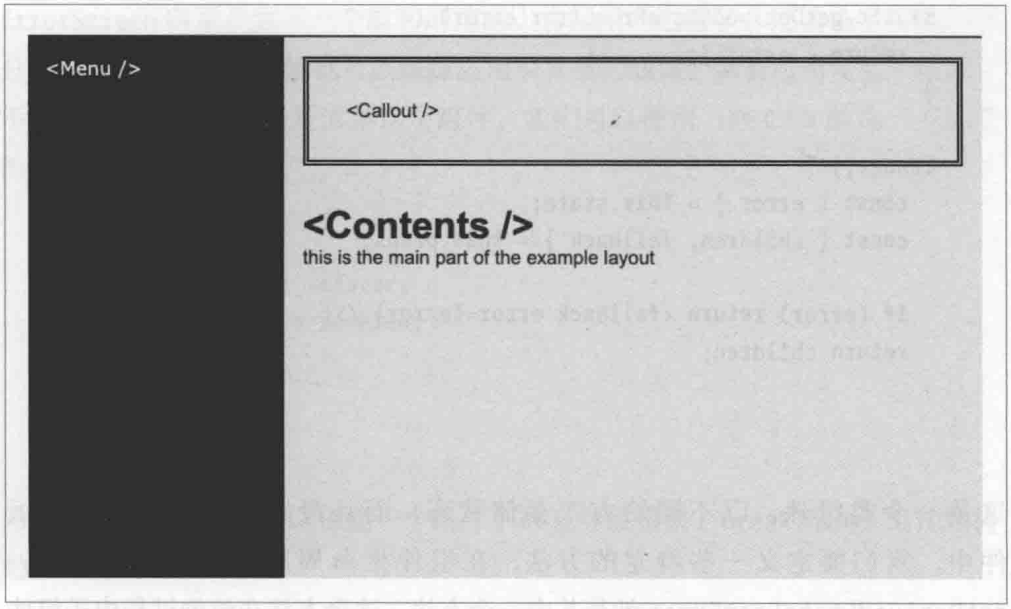

图 9-1：示例布局

9.1 错误边界组件

目前，错误处理没有做到尽善尽美，不管组件树中的什么地方抛出了错误，整个应用都无法运行了。组件树越庞大，项目越复杂，越难调试。有时，我们很难确定错误发生在何处，如果组件不是我们自己写的，情况更糟。

错误边界组件可以防止整个应用崩溃。另外，借此还可以在生产环境中渲染合理的错误消息。由于错误集中在一个组件中处理，我们可以跟踪应用中的错误，把错误发给问题管理系统。

当下，创建错误边界组件的唯一方法是使用类组件。与本章讨论的多数话题一样，这种情况最终也会发生变化。未来，错误边界组件或许能够通过钩子或者其他不用类的方案创建。现在，我们可以像下面这样定义 ErrorBoundary 组件：

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

export default class ErrorBoundary extends Component {
  state = { error: null };
```

```
  static getDerivedStateFromError(error) {
    return { error };
  }

  render() {
    const { error } = this.state;
    const { children, fallback } = this.props;

    if (error) return <fallback error={error} />;
    return children;
  }
}
```

这是一个类组件，以不同的方式存储状态，而且没有使用钩子。在类组件中，我们要定义一些特定的方法，在组件生命周期的不同时刻调用。getDerivedStateFromError 就是其中一个方法。这个方法在渲染过程中子组件发生错误时调用。一旦发生错误，将为 state.error 设值。只要有错误，就通过属性把错误传给 fallback 组件，渲染该组件。

现在，我们可以使用这个组件捕获组件树中的错误，倘若发现错误，渲染 fallback 组件。例如，我们可以把整个应用放在一个错误边界组件中：

```
function ErrorScreen({ error }) {
  //
  // 可以在渲染消息之前处理或跟踪错误
  //
  return (
    <div className="error">
      <h3>We are sorry... something went wrong</h3>
      <p>We cannot process your request at this moment.</p>
      <p>ERROR: {error.message}</p>
    </div>
  );
}

<ErrorBoundary fallback={ErrorScreen}>
  <App />
</ErrorBoundary>;
```

ErrorScreen 向用户显示一个温和的消息，告知用户出错了，还渲染了错误的部分详情。此外，这个函数也可以跟踪应用中发生的错误。倘若应用发生了错误，不再显示一个黑屏，而是渲染这个组件。我们可以使用一些 CSS 装饰一下这个组件：

```
.error {
  background-color: #efacac;
  border: double 4px darkred;
  color: darkred;
  padding: 1em;
}
```

为了测试效果，我们可以创建一个组件，故意导致错误。BreakThings 组件始终抛出错误：

```
const BreakThings = () => {
  throw new Error("We intentionally broke something");
};
```

错误边界组件可以与其他组件交织使用。我们知道，App 组件可以放在 ErrorBoundary 组件中，此外，也可以把 App 中的各个组件分别放到 ErrorBoundary 组件中：

```
  return (
    <SiteLayout
      menu={
        <ErrorBoundary fallback={ErrorScreen}>
          <p>Site Layout Menu</p>
          <BreakThings />
        </ErrorBoundary>
      }
    >
      <ErrorBoundary fallback={ErrorScreen}>
        <Callout>Callout<BreakThings /></Callout>
      </ErrorBoundary>
      <ErrorBoundary fallback={ErrorScreen}>
        <h1>Contents</h1>
        <p>this is the main part of the example layout</p>
```

```
        </ErrorBoundary>
      </SiteLayout>
```

只要 ErrorBoundary 组件的子组件发生了错误，ErrorBoundary 就渲染 fallback 组件。这里，我们把 BreakThings 组件放到菜单和 Callout 中，因此 ErrorScreen 将渲染两次，如图 9-2 所示。

可以看到，ErrorBoundary 组件在恰当的位置渲染了。注意，两个错误发生在各自的区域中。错误边界组件就像一道墙一样，能防止错误侵袭应用中的其他区域。尽管我们故意抛出了两个错误，区域中的内容依然能正常渲染。

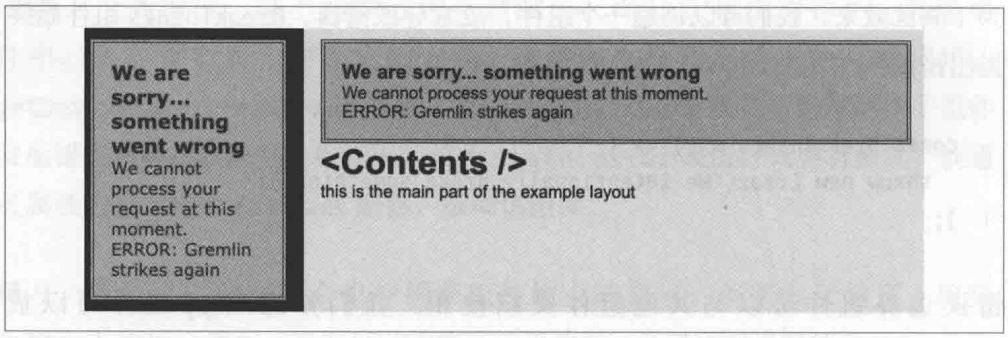

图 9-2：错误边界组件的效果

只把 BreakThings 组件放到内容区的效果如图 9-3 所示。

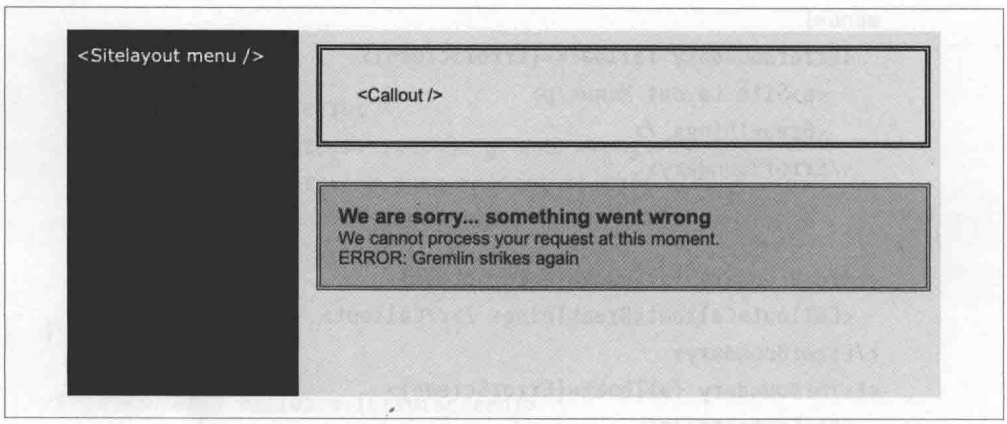

图 9-3：出错啦

可以看到，菜单和 Callout 可以正常渲染，不过内容区中渲染了一个错误消息，告知用户发生了错误。

在 ErrorBoundary 类组件的 render 方法中，我们可以调整一下，把 fallback 属性改成可选的。如果没有提供这个属性，直接使用 ErrorScreen 组件。

```
render() {
  const { error } = this.state;
  const { children } = this.props;

  if (error && !fallback) return <ErrorScreen error={error} />;
  if (error) return <fallback error={error} />;

  return children;
}
```

在整个应用中以统一的方式处理错误是个好方法。现在，我们只需把组件树中的特定部分放到 ErrorBoundary 组件中，把余下的事情都交给它处理即可：

```
<ErrorBoundary>
  <h1>&lt;Contents /&gt;</h1>
  <p>this is the main part of the example layout</p>
  <BreakThings />
</ErrorBoundary>
```

错误边界组件不是空想出来的一个好主意，这个概念对保有线上用户十分重要，而且可以避免相对不太重要的组件发生的小 bug 导致整个应用崩溃。

9.2 代码分拆

如果目前你的应用体量较小，情况可能并不尽然。随着时间的积累，很多应用的代码基可能包含上百甚至上千个组件。多数用户在访问你的应用时会使用网速较慢的手机，他们可不想等到应用的整个代码基都下载完毕后才能看到 React 初次渲染的结果。

代码分拆技术可以把代码基分解为便于管理的小块，然后按需加载各小块。为了展示代码分拆的效果，下面举个例子，在应用中添加一个用户协议界面：

```
export default function Agreement({ onAgree = f => f }) {
  return (
    <div>
      <p>Terms...</p>
      <p>These are the terms and stuff. Do you agree?</p>
      <button onClick={onAgree}>I agree</button>
    </div>
  );
}
```

然后，把代码基中余下的代码从App组件移到Main组件中，保存在单独的文件中：

```
import React from "react";
import ErrorBoundary from "./ErrorBoundary";

const SiteLayout = ({ children, menu = c => null }) => {
  return (
    <div className="site-container">
      <div>{menu}</div>
      <div>{children}</div>
    </div>
  );
};

const Menu = () => (
  <ErrorBoundary>
    <p style={{ color: "white" }}>TODO: Build Menu</p>
  </ErrorBoundary>
);

const Callout = ({ children }) => (
  <ErrorBoundary>
    <div className="callout">{children}</div>
  </ErrorBoundary>
);
```

```
export default function Main() {
  return (
    <SiteLayout menu={<Menu />}>
      <Callout>Welcome to the site</Callout>
      <ErrorBoundary>
        <h1>TODO: Home Page</h1>
        <p>Complete the main contents for this home page</p>
      </ErrorBoundary>
    </SiteLayout>
  );
}
```

现在，网站布局在Main组件中渲染。接下来，修改App组件，渲染Agreement，等待用户同意。用户同意后，卸载Agreement组件，再渲染Main组件。

```
import React, { useState } from "react";
import Agreement from "./Agreement";
import Main from "./Main";
import "./SiteLayout.css";

export default function App() {
  const [agree, setAgree] = useState(false);

  if (!agree)
    return <Agreement onAgree={() => setAgree(true)} />;

  return <Main />;
}
```

一开始，唯一渲染的组件是Agreement。用户同意后，agree的值变成true，此时渲染Main组件。现在的问题是，Main组件及其所有子组件都打包在一个JavaScript文件中。这意味着，用户要等待代码基下载完毕后才能开始渲染Agreement组件。

为了在渲染时才加载Main组件，我们可以使用React.lazy声明，而不是直接导入：

```
const Main = React.lazy(() => import("./Main"));
```

这行代码让 React 在首次渲染 Main 组件时再加载对应的代码基，即在渲染时再使用 import 函数导入 Main 组件。

在运行时导入代码与从互联网中加载内容是一样的。首先，对 JavaScript 代码的请求处于等待状态，真正加载时要么成功，返回一个 JavaScript 文件，要么失败，抛出错误。加载数据时我们要让用户知道数据正在加载中，加载代码也一样，要让用户知道代码正在加载中。

9.2.1 Suspense 组件介绍

你会发现，我们又一次进入管理异步请求的情况之中。这一次，我们有 Suspense 组件的协助。Suspense 组件的用法与 ErrorBoundary 组件类似，即把组件树中特定的组件嵌套其中。只不过，ErrorBoundary 组件在出现错误时回落到一个错误消息，而 Suspense 组件在惰性加载组件时渲染一个消息，提示正在加载中。

我们来修改一下惰性加载 Main 组件的代码，如下所示：

```
import React, { useState, Suspense, lazy } from "react";
import Agreement from "./Agreement";
import ClimbingBoxLoader from "react-spinners/ClimbingBoxLoader";

const Main = lazy(() => import("./Main"));

export default function App() {
  const [agree, setAgree] = useState(false);

  if (!agree)
    return <Agreement onAgree={() => setAgree(true)} />;

  return (
    <Suspense fallback={<ClimbingBoxLoader />}>
      <Main />
    </Suspense>
  );
}
```

现在，应用最初只加载 React、Agreement 组件和 ClimbingBoxLoader，等用户同意协议后再加载 Main 组件。

Main 组件嵌套在 Suspense 组件中。一旦用户同意协议，立即开始加载 Main 组件的代码基。由于对这部分代码基的请求目前处于等待状态，因此 Suspense 组件将在 Main 组件的位置渲染 ClimbingBoxLoader，直至成功加载 Main 组件。加载成功后，Suspense 组件将卸载 ClimbingBoxLoader，转而渲染 Main 组件。



React Spinners 是一个动画加载框库，指明正在加载什么东西或应用正在处理中。本章后文将举例说明这个库中不同加载框组件的用法。使用之前请安装：npm i react-spinners。

如果在加载 Main 组件之前网络连接中断了呢？这是一个错误，要做处理。我们可以把 Suspense 组件嵌套在 ErrorBoundary 组件中：

```
<ErrorBoundary fallback={ErrorScreen}>
  <Suspense fallback={<ClimbingBoxLoader />}>
    <Main />
  </Suspense>
</ErrorBoundary>
```

这三个组件组合在一起可以处理多数异步请求。我们可以处理等待状态：Suspense 组件在请求源码的过程中渲染一个加载框。我们可以处理失败状态：在加载 Main 组件的过程中如果出错了，由 ErrorBoundary 组件捕获并处理。我们还可以处理成功状态：请求成功后渲染 Main 组件。

9.2.2 使用 Suspense 加载数据

前一章构建了一个 useFetch 钩子和一个 Fetch 组件，通过它们处理 GitHub 请求涉及的三个状态，即待定、成功和失败。当时采用这种方法觉得挺好，然而前一节我们又换了一种更优雅的方法，通过 ErrorBoundary 和 Suspense 组件处理三个状态。虽然前一节是为了惰性加载 JavaScript 源码，不过这个模式也可用于加载数据。

假设有一个 Status 组件，用于渲染状态消息：

```
import React from "react";

const loadStatus = () => "success - ready";

function Status() {
  const status = loadStatus();
  return <h1>status: {status}</h1>;
}
```

这个组件调用 loadStatus 函数，获取当前的状态消息。我们可以在 App 组件中渲染 Status 组件：

```
export default function App() {
  return (
    <ErrorBoundary>
      <Status />
    </ErrorBoundary>
  );
}
```

直接运行这段代码，我们会看到成功状态消息，如图 9-4 所示。

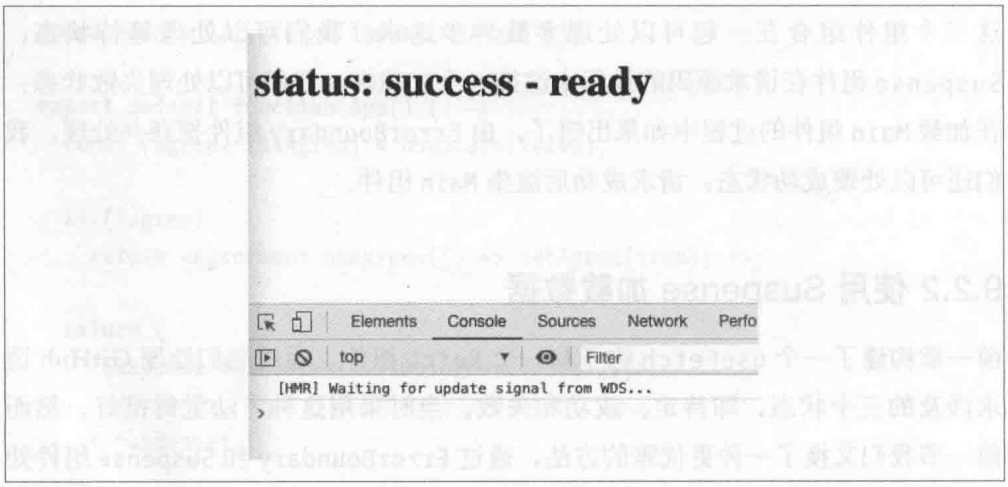

图 9-4：成功：一切正常

像这样在 App 组件中渲染 Status 组件，即把 Status 组件放在一个错误边界组件中，说明你是一个不错的 React 开发者。现在，在加载状态消息的过程中如果出错了，ErrorBoundary 将回落到默认的错误界面。为了演示效果，我们在 loadStatus 函数中抛出一个错误：

```
const loadStatus = () => {
  throw new Error("something went wrong");
};
```

然后运行应用，我们会看到预期的输出。ErrorBoundary 捕获到错误，渲染一个消息，呈现给用户（见图 9-5）。

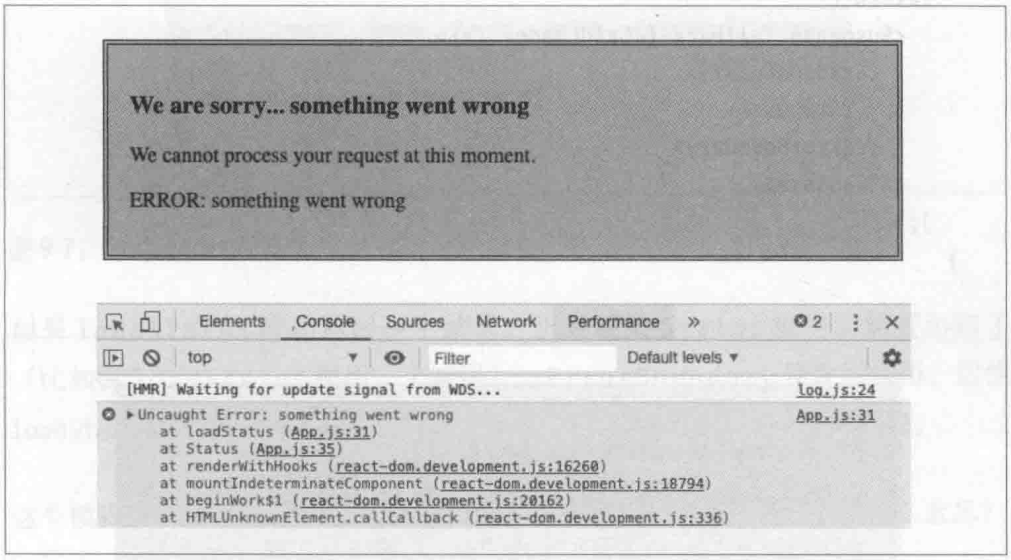

图 9-5：失败：触发错误边界

目前，一切都在我们的预料之中。我们把 Status 组件放在 ErrorBoundary 组件中，综合二者处理 promise 三个状态中的两个，即成功和被拒。“被拒”是 promise 失败或出错状态的正式术语。

我们涵盖了三个状态中的两个，那么第三个状态，即待定状态怎么办呢？抛出一个 promise 可以触发这个状态：

```
const loadStatus = () => {
  throw new Promise(resolves => null);
};
```

在 loadStatus 函数中抛出一个 promise，在浏览器中将看到一个特殊的错误（见图 9-6）。

错误消息指出我们触发了待定状态，但是组件树中更高层级上没有配置 Suspense 组件。只要在 React 应用中抛出一个 promise，就需要一个 Suspense 组件负责渲染回落组件：

```
export default function App() {
  return (
    <Suspense fallback={<GridLoader />}>
      <ErrorBoundary>
        <Status />
      </ErrorBoundary>
    </Suspense>
  );
}
```

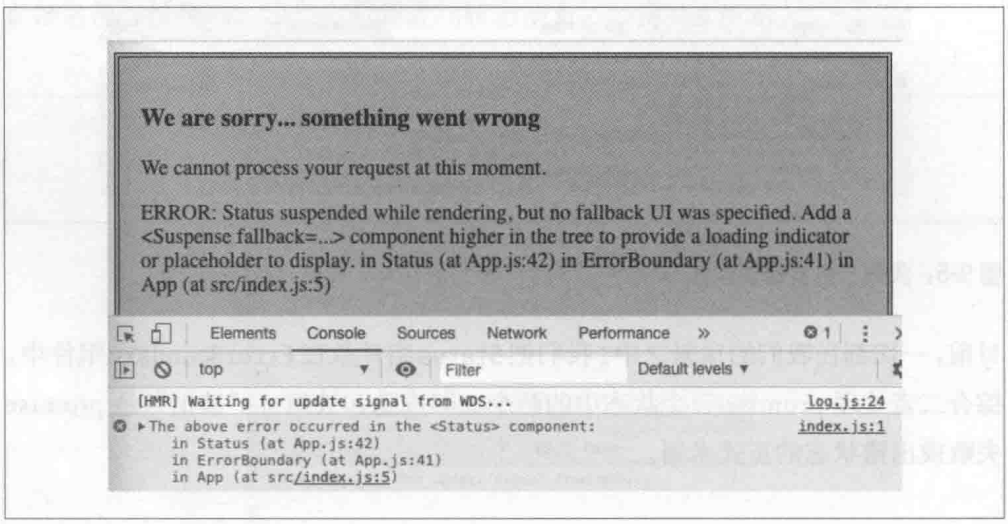

图 9-6：抛出 promise

现在，我们通过组合不同的组件正确处理了全部三种状态。虽然 loadStatus 函数抛出一个 promise，但是组件树中更高层级上有一个 Suspense 组件可以处理。抛出 promise，React 就知道现在处于待定状态，从而渲染回落的 GridLoader 组件（见图 9-7）。

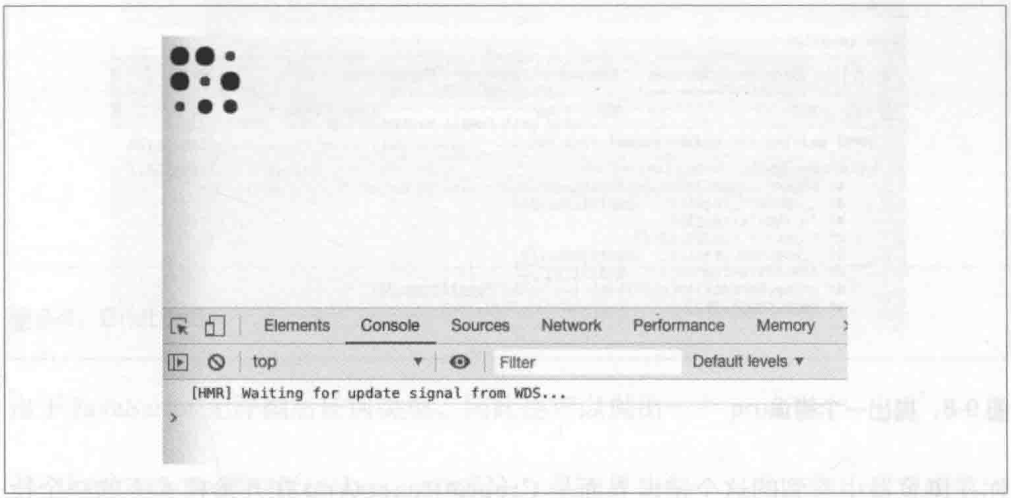

图 9-7：GridLoader 加载框

如果 loadStatus 成功返回一个结果，正常渲染 Status 组件。倘若出错了（比如说 loadStatus 抛出一个错误），ErrorBoundary 便发挥作用。假使 loadStatus 抛出一个 promise，触发待定状态，则由 Suspense 组件处理。

这个模式非常棒，但是等一下，本节所说的“抛出一个 promise”是什么意思？

9.2.3 抛出 promise

严格来说，JavaScript 中的 throw 关键字用于抛出错误。你自己写代码时或许已经用过很多次了：

```
throw new Error("inspecting errors");
```

这行代码触发一个错误。倘若不对错误做任何处理，将导致整个应用崩溃，如图 9-8 所示。

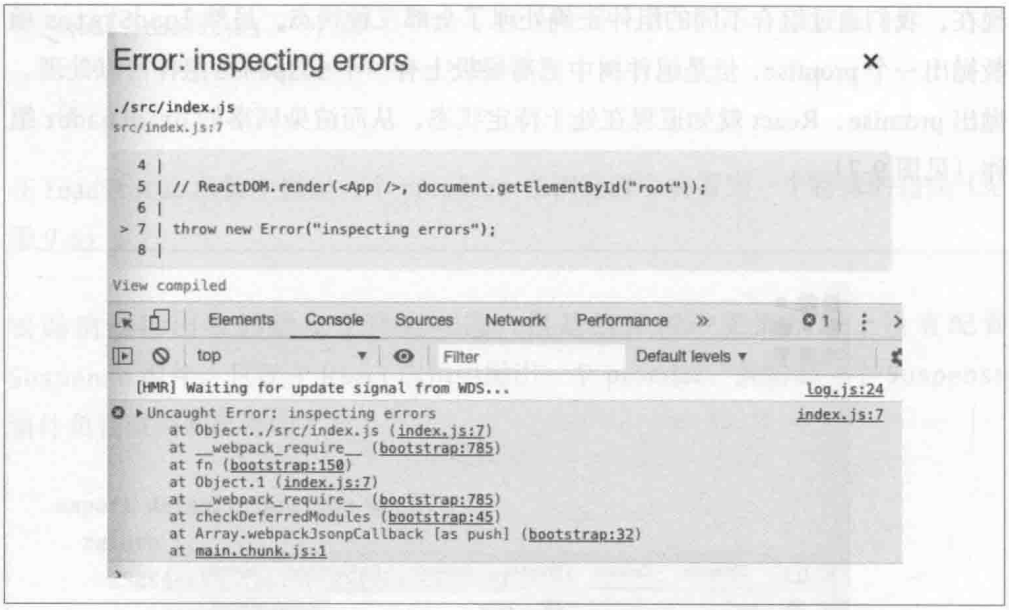

图 9-8：抛出一个错误

你在浏览器中看到的这个错误界面是 Create React App 在开发模式下的一个特性。在开发模式中，未处理的错误被捕获后都直接显示在屏幕上。如果单击右上角的“X”关掉这个界面，你看到的界面与生产环境中出错时用户看到的界面一样，即一个空白界面。

未处理的错误在控制台中始终可见。在控制台中看到的所有红色文本都是有关错误的信息。

JavaScript 是一门非常自由的语言，很多在传统的类型语言中做不到的事情，在 JavaScript 中都可以做到。比如说，在 JavaScript 中可以抛出任何类型：

```
throw "inspecting errors";
```

这里抛出一个字符串。浏览器会提醒我们有什么东西未捕获，但不是错误（见图 9-9）。

这一次，我们抛出的是一个字符串，浏览器没有渲染 Create React App 的错误界面。React 知道错误和字符串是有区别的。

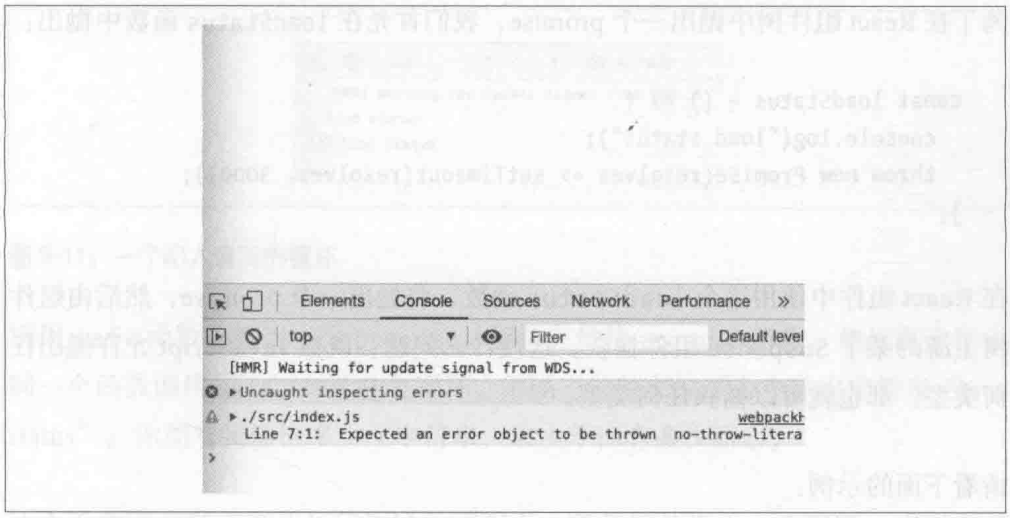

图 9-9：GridLoader

由于 JavaScript 允许抛出任何类型，因此也可以抛出一个 promise：

```
throw new Promise(resolves => null);
```

现在，浏览器会告诉我们有什么东西未捕获，但不是错误，而是一个 promise，如图 9-10 所示。

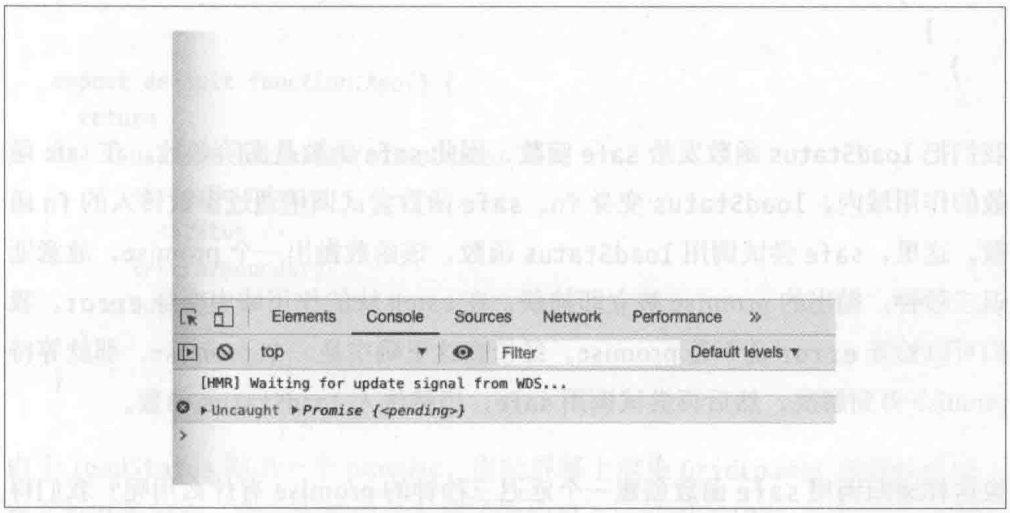

图 9-10：抛出一个 promise

为了在 React 组件树中抛出一个 promise，我们首先在 loadStatus 函数中抛出：

```
const loadStatus = () => {
  console.log("load status");
  throw new Promise(resolves => setTimeout(resolves, 3000));
};
```

在 React 组件中使用这个 loadStatus 函数，将抛出一个 promise，然后由组件树上游的某个 Suspense 组件捕获。这没什么问题，既然 JavaScript 允许抛出任何类型，那也就可以捕获任何类型。

请看下面的示例：

```
safe(loadStatus);

function safe(fn) {
  try {
    fn();
  } catch (error) {
    if (error instanceof Promise) {
      error.then(() => safe(fn));
    } else {
      throw error;
    }
  }
}
```

我们把 loadStatus 函数发给 safe 函数，因此 safe 函数是高阶函数。在 safe 函数的作用域内，loadStatus 变身 fn。safe 函数尝试调用通过参数传入的 fn 函数。这里，safe 尝试调用 loadStatus 函数，该函数抛出一个 promise，故意延迟三秒钟。抛出的 promise 被立即捕获，在 catch 块的作用域中变身 error。我们可以检查 error 是不是 promise。经检查这里确实是一个 promise，那就等待 promise 得到解决，然后再尝试调用 safe，仍然传入 loadStatus 函数。

像这样递归调用 safe 函数创建一个延迟三秒钟的 promise 有什么用呢？我们得到的是一个延迟的循环，如图 9-11 所示。

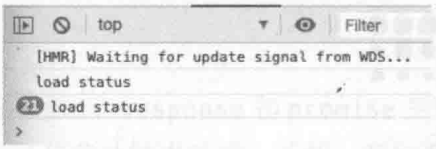

图 9-11：一个陷入窘境的循环

调用 safe 函数，抛出一个 promise，等待三秒待 promise 解决，然后再次传入同一个函数调用 safe，如此往复循环，每隔三秒钟在控制台中输出字符串“load status”。你能看到输出多少次字符串，取决于你有多少耐心。

这个无穷递归循环不是为了测试你的耐心，而是借此指出一个问题。下面来看与前面的 Status 组件一起搭配使用这个新 loadStatus 函数有什么效果：

```
const loadStatus = () => {
  console.log("load status");
  throw new Promise(resolves => setTimeout(resolves, 3000));
};

function Status() {
  const status = loadStatus();
  return <h1>status: {status}</h1>;
}

export default function App() {
  return (
    <Suspense fallback={<GridLoader />}>
      <ErrorBoundary>
        <Status />
      </ErrorBoundary>
    </Suspense>
  );
}
```

由于 loadStatus 抛出一个 promise，因此屏幕上渲染 GridLoader 加载框动画。现在打开控制台，你会发现它又在测试你的耐心（见图 9-12）。

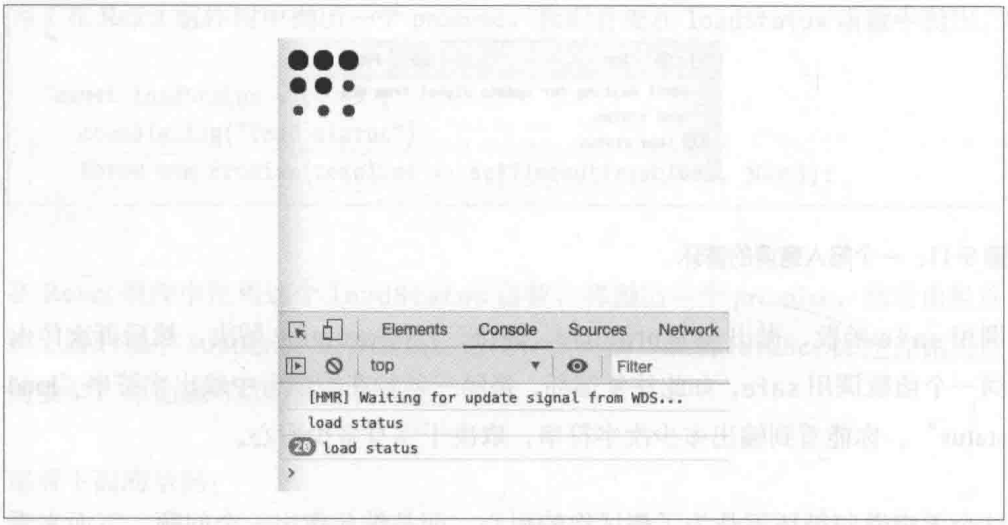

图 9-12：递归渲染 Suspense 组件

这与调用 safe 函数的情况类似。Suspense 组件知道我们抛出了一个 promise，遂渲染 fallback 组件。然后，与 safe 函数一样，Suspense 组件等待抛出的 promise 得到解决。解决后，Suspense 组件再次渲染 Status 组件，而 Status 又调用 loadStatus 函数，整个过程再重复一次。结果是，控制台每隔三秒输出一个“load status”字符串，无穷无尽。

通常，无穷循环并不是我们想要的效果，在 React 中也是如此。我们要知道的是，抛出的 promise 会被 Suspense 组件捕获，随之进入待定状态，直至 promise 得到解决。

9.2.4 构建兼容 Suspense 的数据源

兼容 Suspense 的数据源要提供一个函数处理加载数据相关的全部状态，即待定、成功和出错。loadStatus 函数现在一次只能返回或抛出一个类型。我们需要 loadStatus 函数在加载数据时抛出一个 promise，成功加载数据后返回一个响应，出错时抛出一个错误。

```
function loadStatus() {
  if (error) throw error;
  if (response) return response;
```

```
    throw promise;
  }
```

我们要在某个地方声明 error、response 和 promise 变量，并确保这些变量位于恰当的作用域内，以免与其他请求冲突。为此，我们要使用闭包（closure）定义 loadStatus 函数：

```
const loadStatus = (function() {
  let error, promise, response;

  return function() {
    if (error) throw error;
    if (response) return response;
    throw promise;
  };
})();
```

这是一个闭包。error、promise 和 response 的作用域在 loadStatus 函数的定义体外部将关闭。我们声明的 loadStatus 是一个匿名函数，而且立即调用：fn() 的作用与 (fn)() 相同。loadStatus 函数的返回值是内部返回的那个函数。loadStatus 函数可以访问 error、promise 和 response 变量，但是其余的 JavaScript 代码不可以。

现在，我们只需处理 error、response 和 promise 的值。promise 等待三秒后成功解决，解决后，把 response 的值设为“success”。如果出错，或者 promise 被拒，将其设为 error 的值。

```
const loadStatus = (function() {
  let error, response;
  const promise = new Promise(resolves =>
    setTimeout(resolves, 3000)
  )
    .then(() => (response = "success"))
    .catch(e => (error = e));
  return function() {
    if (error) throw error;
    if (response) return response;
```

```
      throw pending;
    };
  })();
```

我们创建了一个等待三秒钟的 promise。如果在等待期间调用 loadStatus 函数，抛出该 promise。三秒过后，promise 成功得到解决，为 response 赋一个值。如果现在调用 loadStatus 函数，返回值是“success”响应。如果什么地方出错了，loadStatus 函数返回 error。

这个 loadStatus 函数就是兼容 Suspense 的数据源，能与 Suspense 架构互通状态。目前，loadStatus 函数是硬编码的，始终能解决延迟三秒的 promise。不过，对 error、response 和 promise 的处理机制是可以套用的。我们可以使用这种技术包装任何 promise，得到兼容 Suspense 的数据源。

创建兼容 Suspense 的数据源只需要一个 promise，因此我们可以定义一个函数，接受一个 promise 为参数，返回一个兼容 Suspense 的数据源。在下面的示例中，我们把这个函数命名为 createResource：

```
const resource = createResource(promise);
const result = resource.read();
```

这段代码假定 createResource(promise) 能成功创建一个 resource 对象。该对象有个 read 函数，可以调用任意多次。promise 得到解决后，read 返回最终的数据。在 promise 处于待定状态时，read 抛出 promise。如果有什么地方出错了，read 将抛出错误。这个数据源兼容 Suspense。

createResource 函数与前面定义的那个匿名函数很像：

```
function createResource(pending) {
  let error, response;
  pending.then(r => (response = r)).catch(e => (error = e));
  return {
    read() {
      if (error) throw error;
      if (response) return response;
```

```
        throw pending;
      }
    };
  }
```

这个函数依然隔离 error 和 response 的值，不过允许调用方传入一个名为 pending 的 promise 参数。待定的 promise 得到解决后，通过 .then 函数捕获结果。如果 promise 被拒，捕获错误，为 error 变量赋值。

createResource 函数返回一个资源对象，该对象有个名为 read 的函数。如果 promise 仍然处于待定状态，error 和 response 是未定义的，read 抛出 promise。如果在 error 有值时调用 read，将抛出 error。最后，有响应时调用 read 将产出解决 promise 后得到的数据。不管我们调用多少次 read，它始终能准确报告 promise 的状态。

为了在组件中测试这个函数的作用，我们需要一个 promise，名称最好听起来像 20 世纪八十年代的滑雪电影：

```
const threeSecondsToGnar = new Promise(resolves =>
  setTimeout(() => resolves({ gnar: "gnarly!" }), 3000)
);
```

threeSecondsToGnar 这个 promise 等待三秒后解析出一个对象，该对象中有个 gnar 字段，并为其设定了值。下面使用这个 promise 创建一个兼容 Suspense 的数据源，然后在一个小型 React 应用中使用该数据源：

```
const resource = createResource(threeSecondsToGnar);

function Gnar() {
  const result = resource.read();
  return <h1>Gnar: {result.gnar}</h1>;
}

export default function App() {
  return (
    <Suspense fallback={<GridLoader />}>
```

```
        <ErrorBoundary>
          <Gnar />
        </ErrorBoundary>
      </Suspense>
    );
  }
```

React 组件的渲染能力强大。Gnar 组件将被渲染多次，最后才返回一个响应。每次渲染 Gnar 组件，都调用 resource.read()。第一次渲染 Gnar 组件，抛出一个 promise。这个 promise 由 Suspense 组件处理，渲染指定的 fallback 组件。

promise 得到解决后，Suspense 组件再次渲染 Gnar 组件。Gnar 组件再次调用 resource.read()，不过这一次假设一切顺利，resource.read() 成功返回 Gnar，在 h1 元素中渲染 Gnar 的状态。如果有什么地方出错了，resource.read() 将抛出一个错误，由 ErrorBoundary 处理。

可以看到，createResource 函数相当完备。我们的资源可以处理错误，假如遇到网络错误，可以等待几秒，然后再尝试加载数据。我们的资源可以与其他资源通信，例如把所有资源的性能统计数据记录到日志中。只有想不到，没有做不到。只要有这么一个函数可以读取资源的当前状态，我们就可以对资源做任何操作。

以上是目前为止 Suspense 的运作方式，任何类型的异步资源都可以像前文所述的那样使用 Suspense 组件处理。具体方式有可能会变，我们也希望改变。然而，无论怎么变，Suspense 最终的 API 肯定仍然能够处理三个状态：待定、成功和失败。

我们对 Suspense API 的介绍有些简略，这是有意为之的，毕竟这是一项实验特性，未来会发生变化。本章的主旨是让你知道，React 一直在努力提升 React 的运行速度。

这个特性背后涉及很多 React 自身的运行机制，尤其是调和算法 Fiber。

9.2.5 Fiber

本书贯穿始终强调一点，React组件是函数，返回的数据用于渲染UI。每有数据（属性、状态、远程数据，等待）发生变化，React都将重新渲染组件。单击星标为颜色评分后，我们希望UI发生变化，而且速度越快越好。我们相信React可以尽职尽责。可是，React具体是如何做到的呢？为了理解React是如何高效更新DOM的，我们要深入介绍一下React的运作机制。

假设你撰写了一篇文章，准备发表到公司的博客中。在发布之前，你把文章发给同事，想得到一些反馈。同事建议做几处小改动，你准备采纳。你新建一个文档，从头开始输入整篇文章，然后做了改动。

这些额外工作可能会让你抱怨，但这就是很多库以前的运作方式。为了做一次更新，我们要推倒一切，从头重新构建DOM。

现在，你又写了一篇博客文章，发给同事检查。这一次，你采用了现代化的文章撰写流程，用上了GitHub。你的同事检出一个GitHub分支，做些改动，然后合并到分支中。速度更快，效率更高。

这个过程与React的运作方式类似。发生变化时，React以JavaScript对象的形式创建一份组件树副本，找出组件树中需要改动的部分，只改动那些部分。修改完毕后，使用副本（称为work-in-progress tree）替换现有组件树。强调一点，React将继续使用组件树中现存的部分。比如说我们要把列表中的`red`项改为`green`：

```
<ul>
  <li>blue</li>
  <li>purple</li>
  <li>red</li>
</ul>
```

React不会删掉第三个`li`，而是把子节点（`red`文本）替换为`green`文本。这样更新效率高，自问世以来React就是这样更新DOM的。然而，这里隐藏一个问

题。更新 DOM 是一项耗时的任务，因为这个操作是同步的。我们要等到所有更新都到位并渲染出来之后，在主线程中才能执行其他任务。也就是说，我们要等到所有更新都结束，这会影响用户体验，好似没有反应。

为了解决这个问题，React 开发团队彻底重写了 React 的调和算法，名为 Fiber。Fiber 在 16.0 版中发布，重写了 DOM 的更新机制，采用一种异步方式。16.0 版引入的第一处改动是分离渲染器（renderer）和调和器（reconciler）。渲染器是 React 库中处理渲染的部分，调和器是 React 库中管理更新的部分。

把渲染器和调和器分开意义重大。调和算法保留在 React Core 中（要使用 React 就安装这个包），而各个渲染目标则负责具体渲染。也就是说，ReactDOM、React Native、React 360 等负责处理渲染逻辑，可以接入 React 的核心调和算法。

React Fiber 的另一项重要改动是调整了调和算法。还记得耗时的 DOM 更新操作会阻塞主线程吗？长时间阻塞的更新操作叫作“work”，React Fiber 把 work 分解为较小的单元，叫作“fiber”。一个 fiber 是一个 JavaScript 对象，负责记录正在调和什么，及其在更新循环中的什么位置。

当一个 fiber（work 单元）结束后，React 检查主线程中有没有什么重要的事情要做。如果有重要的工作，React 将交出主线程的控制权。当重要的工作做完之后，React 再继续更新。如果主线程接下来没有重要的任务，React 转到下一个 work 单元，然后在 DOM 中渲染改动。

以前文的 GitHub 示例为例，每个 fiber 表示一个分支中的一个提交，把分支合并到主分支中后，又表示更新后的 DOM 树。Fiber 把一项工作分拆成多个小块，这样优先级高的任务就可以插队，让主线程先行处理。这样实现的用户体验给人的感觉更流畅。

Fiber 能做到这一点就已经成功了，而它另有更多的潜力。除了把工作分解为小型单元得到了性能提升之外，此次重写还为将来埋下了伏笔，让人振奋。Fiber 为按优先级更新奠定了基础。长远来看，说不定开发者甚至可以调整默认的优先次序，自行决定哪种任务的优先级最高。指定 work 单元优先级的过程叫作调

度，这个概念是实验特性并发模式的底层基础，终有一天我们可以并行执行各work单元。

在生产环境中使用React不要求掌握Fiber，不过了解对调和算法的重写可以深入认识React的运作机制，并了解React的贡献者对未来的思考。

第 10 章

React 测试

为了不落后于竞争对手，我们必须行动迅速，同时确保质量。保证质量的一个重要工具是单元测试（unit testing）。单元测试可以确认应用的每一个功能或单元能按预期工作。[注 1]

函数式编程技术的一个优势是写出的代码易于测试。纯函数生而便于测试。不可变性易于测试。使用专注特定任务的小型函数构织应用，实现的功能或代码单元易于测试。

本章介绍几种对 React 应用做单元测试的技术。本章不仅涵盖测试，还涉及可帮助我们评估及改善代码和测试的工具。

10.1 ESLint

对多数编程语言来说，代码要先编译才能运行。这些编程语言对代码风格有十分严格的规则要求，若不合理格式化，代码无法编译。JavaScript 没有如此严格的规则，而且没有编译器。写好代码后，我们十指交叉，然后在浏览器中运行，

注 1：Martin Fowler 写的文章“Unit Testing”（*http://martinfowler.com/bliki/UnitTest.html*）对单元测试做了简单介绍。

确认能否运行。幸好有些工具可以帮助我们分析代码，确保代码遵守一定的格式化指导方针。

分析 JavaScript 代码的过程称为 *hinting* 或 *linting*。最早用来分析 JavaScript 代码的工具是 JSHint 和 JSLint，二者可以给我们提供格式方面的反馈。ESLint（*http://eslint.org*）是一个新近代码 lint 工具，支持最新的 JavaScript 句法。而且，ESLint 支持插件。我们可以创建和分享自己的插件，为 ESLint 增加配置项目，扩展 ESLint 的功能。

Create React App 原生支持 ESLint，我们在控制台中已经见过它发出的 lint 提醒和错误。

我们将使用一个名为 `eslint-plugin-react`（*https://oreil.ly/3yeXO*）的插件。这个插件除了分析 JavaScript 代码之外，还能分析 JSX 和 React 句法。

下面安装 `eslint`，作为一个开发依赖。`eslint` 可使用 npm 安装：

```
npm install eslint --save-dev

# 或者

yarn add eslint --dev
```

使用 ESLint 之前，我们要配置一些应遵守的规则。这些规则在项目根目录中的一个配置文件里定义。这个文件可以使用 JSON 格式，也可以使用 YAML 格式。YAML 是一种数据序列化格式，与 JSON 类似，不过句法更少，便于人类阅读。

ESLint 自带一个创建配置的工具。我们可以在一些公司创建的 ESLint 配置文件的基础上修改，也可以自己动手创建。

下面运行 `eslint --init` 命令，回答一些有关代码风格的问题，创建一个 ESLint 配置文件：

```
npx eslint --init
```

```
How would you like to configure ESLint?
To check syntax and find problems

What type of modules does your project use?
JavaScript modules (import/export)

Which framework does your project use?
React

Does your project use TypeScript?
N

Where does your code run? (Press space to select, a to toggle all,
i to invert selection)
Browser

What format do you want your config file to be in?
JSON

Would you like to install them now with npm?
Y
```

npx eslint --init 命令做了三件事：

1. 在项目的 *./node_modules* 文件夹中安装 eslint-plugin-react。
2. 自动把相关依赖添加到 *package.json* 文件中。
3. 在项目的根目录中创建配置文件 *.eslintrc.json*。

打开 *.eslintrc.json*，里面有个设置对象：

```
{
  "env": {
    "browser": true,
    "es6": true
  },
  "extends": [
    "eslint:recommended",
    "plugin:react/recommended"
  ],
```

```
  "globals": {
    "Atomics": "readonly",
    "SharedArrayBuffer": "readonly"
  },
  "parserOptions": {
    "ecmaFeatures": {
      "jsx": true
    },
    "ecmaVersion": 2018,
    "sourceType": "module"
  },
  "plugins": ["react"],
  "rules": {}
}
```

注意 extends 键，你会发现 --init 命令默认对 eslint 和 react 做了初始化。这意味着我们不用自己动手配置所有规则，相关规则已经给出了。

下面创建一个 *sample.js* 文件，测试一下 ESLint 配置和规则：

```
const gnar = "gnarly";

const info = ({
  file = __filename,
  dir = __dirname
}) => (
  <p>
    {dir}: {file}
  </p>
);

switch (gnar) {
  default:
    console.log("gnarly");
    break;
}
```

这个文件中隐含一些问题，但在浏览器中运行还不至于报错。严格来说，这段

代码可以正常运行。下面运行 ESLint 检查这个文件，看看我们制定的规则会提供什么反馈：

```
npx eslint sample.js

3:7 error 'info' is assigned a value but never used no-unused-vars
4:3 error 'file' is missing in props validation react/prop-types
4:10 error 'filename' is not defined no-undef
5:3 error 'dir' is missing in props validation react/prop-types
5:9 error 'dirname' is not defined no-undef
7:3 error 'React' must be in scope when using JSX react/react-in-jsx-
scope

✖ 6 problems (6 errors, 0 warnings)
```

ESLint 对代码做了静态分析，根据配置报告了一些问题。ESLint 指出了属性验证上的错误，还提到了 `__filename` 和 `__dirname`，因为 ESLint 不会自动引入 Node.js 全局变量。最后，ESLint 默认的 React 提醒告诉我们，使用 JSX 时 React 必须在作用域内。

`eslint .` 命令对整个目录执行 lint 操作。此时，我们往往希望 ESLint 忽略一些 JavaScript 文件。请把想忽略的文件或目录添加到 *.eslintignore* 文件中。

```
dist/assets/
sample.js
```

这个 *.eslintignore* 文件让 ESLint 忽略前面创建的 *sample.js* 文件，以及 *dist/assets* 文件夹中的所有文件。如果不忽略 *assets* 文件夹，ESLint 将分析供客户端使用的 *bundle.js* 文件，有可能会发现大量问题。

下面在 *package.json* 文件中添加一个脚本，用于运行 ESLint：

```
{
  "scripts": {
    "lint": "eslint ."
  }
}
```

现在，执行 npm run lint 命令即可运行 ESLint，分析项目中除了被忽略以外的所有文件。

ESLint 插件

ESLint 插件众多，我们可以把插件添加到 ESLint 配置中，辅助我们编写代码。对 React 项目来说，绝对应该安装 eslint-plugin-react-hooks 插件（*https://reactjs.org/docs/hooks-rules.html*），实施 React Hooks 相关的规则。这个包由 React 团队发布，目的是帮助我们修正钩子相关的 bug。

先来安装：

```
npm install eslint-plugin-react-hooks --save-dev

# 或者

yarn add eslint-plugin-react-hooks --dev
```

然后，打开 *.eslintrc.json* 文件，添加下述内容：

```
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    "react-hooks/rules-of-hooks": "error",
    "react-hooks/exhaustive-deps": "warn"
  }
}
```

这个插件检查名称以“use”开头的函数（假定为钩子），确保遵守钩子相关的规则。

添加上述内容后，我们来编写一些示例代码，测试一下这个插件。参照下述代码修改 *sample.js* 文件。这段代码无法运行，不过我们是为了测试这个插件能不能正常发挥作用。

```
function gnar() {
  const [nickname, setNickname] = useState(
    "dude"
  );
  return <h1>gnarly</h1>;
}
```

这段代码有好几个错误，不过最为重要的一个错误是，我们在一个既非组件也不是钩子的函数中调用了 useState：

```
4:35 error React Hook "useState" is called in function "gnar" that is neither
a React function component nor a custom React Hook function
react-hooks/rules-of-hooks
```

这样的提醒在我们学习使用钩子的过程中可以帮助我们了解各方面的细节。

建议你在项目中使用的另一个 ESLint 插件是 eslint-pluginjsx-a11y。a11y 是一个带数字的缩略词，表示在 accessibility 这个单词中，字母“a”和“y”之间有 11 个字母。可访问性指在构建工具、网站或技术时要考虑残障人士能不能使用。

这个插件分析代码，确保没有违背任何可访问性规则。我们每个人都应该关注可访问性，这个插件可以帮助我们提升 React 应用的可访问性。

这个插件也可以使用 npm 或 yarn 安装：

```
npm install eslint-plugin-jsx-a11y

// 或者

yarn add eslint-plugin-jsx-a11y
```

然后，添加到 *.eslintrc.json* 配置文件中：

```
{
  "extends": [
    // ...
```

```
    "plugin:jsx-a11y/recommended"
  ],
  "plugins": [
    // ...
    "jsx-a11y"
  ]
}
```

现在来测试一下。我们在 *sample.js* 文件中添加一个没有 `alt` 属性的 `image` 标签。要想通过 lint 检查，图像必须有 `alt` 属性，如果图像不影响用户对内容的理解，`alt` 属性的值可以为空字符串。

```
function Image() {
  return <img src="/img.png" />;
}
```

再次执行 `npm run lint` 命令，我们会看到一个由 `jsx/a11y` 插件发出的新错误：

```
5:10 error img elements must have an alt prop, either with meaningful
text, or an empty string for decorative images
```

可用于静态分析代码的 ESLint 插件还有很多，我们通常要花上几周时间才能把 ESLint 配置调整到最佳状态。如果你想进一步完善自己的配置，可以参考 Awesome ESLint 仓库（*https://github.com/dustinspecker/awesome-eslint*）中众多的优秀资源。

10.2 Prettier

Prettier 是一个代码格式化工具，适用于各种不同的项目，可以说是强迫症患者的福音。自发布以来，Prettier 就对 Web 开发人员的日常工作产生了令人难以置信的影响。据史料记载，JavaScript 开发人员平均一天有 87% 的时间在争论句法，有了 Prettier 之后，我们可以为各个项目定义代码句法规则，让 Prettier 处理代码格式化工作。这样节省的时间是非常可观的。另外，如果你使用 Prettier 格式化过 Markdown 表格，干净利落的格式化肯定让你惊叹不已。

以前，很多项目使用 ESLint 格式化代码，不过现在职责划分得很清晰，ESLint 负责保证代码质量，而 Prettier 负责处理代码格式化。

为了使 Prettier 能与 ESLint 协同工作，我们要稍微调整一下项目的配置。首先，全局安装 Prettier：

```
sudo npm install -g prettier
```

现在，不管项目保存在何处，都可以使用 Prettier。

10.2.1 在项目中配置 Prettier

为了在项目中配置 Prettier，要创建一个 *.prettierrc* 文件。我们在这个文件中设置项目的默认配置：

```
{
  "semi": true,
  "trailingComma": none,
  "singleQuote": false,
  "printWidth": 80
}
```

这是笔者选用的默认配置，当然，你要根据自己的需求选择合适的配置。Prettier 支持的格式化选项参见文档（*https://prettier.io/docs/en/options.html*）。

下面我们把 *sample.js* 文件中的内容替换为需要格式化的代码：

```
console.log("Prettier Test")
```

然后，在终端或命令提示符中运行 Prettier CLI：

```
prettier --check "sample.js"
```

Prettier 运行要测试的代码后显示如下消息：`"Code style issues found in the above file(s). Forgot to run Prettier?"`。为了让 CLI 运行 Prettier，要传入 `write` 标志：

```
prettier --write "sample.js"
```

这一次，我们看到的输出是 Prettier 格式化文件所用的毫秒数。打开 *sample.js* 文件，你会发现文件的内容根据 *.prettierrc* 文件中的默认配置做了更改。你可能觉得这个过程有点费事，应该可以简化。没错。下面就来实现自动化。

首先，我们要安装一个配置工具和一个插件，把 ESLint 和 Prettier 集成在一起：

```
npm install eslint-config-prettier eslint-plugin-prettier --save-dev
```

我们安装的配置工具（`eslint-config-prettier`）禁用与 Prettier 有冲突的 ESLint 规则，安装的插件（`eslint-plugin-prettier`）则把 Prettier 规则集成到 ESLint 规则中。如此一来，运行 `lint` 脚本时也将运行 Prettier。

然后，在 *.eslintrc.json* 文件中设置这些工具：

```
{
  "extends": [
    // ...
    "plugin:prettier/recommended"
  ],
  "plugins": [
    //,
  "prettier"],
  "rules": {
    // ...
    "prettier/prettier": "error"
  }
}
```

我们可以故意违背一些代码格式化规则，确认 Prettier 能发挥作用。比如说，在 *sample.js* 文件中写入下述内容：

```
console.log("Prettier Test");
```

执行 `npm run lint` 命令会看到如下输出：

```
1:13 error Replace `'Prettier·Test')` with `"Prettier·Test");` prettier/prettier
```

没有漏网之鱼。现在，运行 Prettier 的写命令，扫除一个文件中的格式化问题：

```
prettier --write "sample.js"
```

或者，针对指定文件夹中的所有 JavaScript 文件：

```
prettier --write "src/*.js"
```

10.2.2 在 VSCode 中使用 Prettier

如果你使用 VSCode，强烈建议在编辑器中设置 Prettier。配置十分简单，节省下来的编程时间却不少。

首先要安装 Prettier 的 VSCode 插件。打开链接 *https://oreil.ly/-7Zgz*，单击“Install”即可。安装之后，在 Mac 中按 Control + Command + P 组合键，在 PC 上按 Ctrl + Shift + P 组合键即可手动格式化一个文件或选中的部分代码。更好的做法是在保存时自动格式化代码。为此，要添加一些 VSCode 设置。

依次选择菜单“Code-Preferences-Settings”（急性子也可以按组合键，在 Mac 中按 Command + ,，在 PC 上按 Ctrl + ,）。然后，点击右上角那个小纸片图标，以 JSON 格式打开 VSCode 设置。在设置中添加下述键值对：

```
{
  "editor.formatOnSave": true
}
```

现在，每次保存文件，Prettier 都会根据 `.prettierrc` 文件中的默认配置格式化代码。真是方便！如果希望在项目中没有 *.prettierrc* 配置文件时也格式化代码，可以把 Prettier 的配置添加到编辑器的设置中。

使用其他编辑器的话，Prettier 可能也提供了相应的支持。其他代码编辑器的具体设置说明参阅文档中的“Editor Integration”部分（*https://prettier.io/docs/en/editors.html*）。

10.3 对 React 应用做类型检查

开发大型应用时，你可能想引入类型检查，借此找出特定的 bug。对 React 应用做类型检查主要有三种方案：prop-types 库、Flow 和 TypeScript。接下来的几节详细说明如何借助这些工具提升代码质量。

10.3.1 PropTypes

写作本书第一版时，PropTypes 在 React 核心库中，是对应用做类型检查的推荐方式。如今，随着其他方案的出现，例如 Flow 和 TypeScript，为了减小 React 库的体积，这个功能从核心库中移除，变成了独立的库。不过，PropTypes 依然使用广泛。

若想在应用中使用 PropTypes，要安装 prop-types 库：

```
npm install prop-types --save-dev
```

为了测试 PropTypes 的功能，下面创建一个极简单的 App 组件，渲染一个库的名称：

```
import React from "react";
import ReactDOM from "react-dom";

function App({ name }) {
  return (
    <div>
      <h1>{name}</h1>
    </div>
  );
}

ReactDOM.render(
  <App name="React" />,
  document.getElementById("root")
);
```

然后，导入 prop-types 库，使用 App.propTypes 定义各属性应该为什么类型：

```
import PropTypes from "prop-types";

function App({ name }) {
  return (
    <div>
      <h1>{name}</h1>
    </div>
  );
}

App.propTypes = {
  name: PropTypes.string
};
```

App 组件有一个 name 属性，其值始终应为字符串。倘若传入的类型不对，将抛出错误。比如说，使用一个布尔值：

```
ReactDOM.render(
  <App name="React" />,
  document.getElementById("root")
);
```

在控制台中会看到下述错误：

```
Warning: Failed prop type: Invalid prop name of type boolean supplied to
App, expected string. in App
```

为属性提供错误类型的值导致的提醒只在开发模式下显示，在生产环境中不显示，也不会导致渲染中断。

当然，对属性也可以做其他类型验证。例如，我们可以添加一个布尔值属性，指明一项技术在公司中有没有应用：

```
function App({ name, using }) {
  return (
    <div>
      <h1>{name}</h1>
      <p>
```

```
          {using ? "used here" : "not used here"}
        </p>
      </div>
    );
  }

  App.propTypes = {
    name: PropTypes.string,
    using: PropTypes.bool
  };

  ReactDOM.render(
    <App name="React" using={true} />,
    document.getElementById("root")
  );
```

PropTypes 支持的类型检查包括：

- PropTypes.array
- PropTypes.object
- PropTypes.bool
- PropTypes.func
- PropTypes.number
- PropTypes.string
- PropTypes.symbol

另外，如果想确认有没有提供值，可以在上述检查的后面串上 .isRequired。比如说，要求必须提供字符串，可以这样检查：

```
App.propTypes = {
  name: PropTypes.string.isRequired
};

ReactDOM.render(
```

```
  <App />,
  document.getElementById("root")
);
```

这里，如果没有为该字段提供值，控制台中将出现下述提醒：

```
index.js:1 Warning: Failed prop type: The prop name is marked as required
in App, but its value is undefined.
```

还有些情况，我们不关心提供的是什么值，只要提供一个值就可以。此时，使用 any。例如：

```
App.propTypes = {
  name: PropTypes.any.isRequired
};
```

这意味着，我们可以提供布尔值、字符串、数字等类型的值，只要 name 不是 undefined，类型检查就能通过。

除了基本的类型检查之外，还有一些其他实用方法可以帮助我们解决很多实际问题。以下述组件为例，status 的值有两个选项：Open 和 Closed。

```
function App({ status }) {
  return (
    <div>
      <h1>
        We're {status === "Open" ? "Open!" : "Closed!"}
      </h1>
    </div>
  );
}

ReactDOM.render(
  <App status="Open" />,
  document.getElementById("root")
);
```

状态是一个字符串，因此我们可能会像这样做字符串检查：

```
App.propTypes = {
  status: PropTypes.string.isRequired
};
```

这个检查本身没什么问题，但是如果传入了 `Open` 和 `Closed` 之外的字符串值，属性将是无效的。其实，我们应该实施的是枚举检查。枚举类型是一个受限的选项列表，相应的字段或属性只能使用其中的值。我们要像下面这样调整 `propTypes` 对象：

```
App.propTypes = {
  status: PropTypes.oneOf(["Open", "Closed"])
};
```

在 React 应用中，PropTypes 可用的全部配置选项参见文档（*https://oreil.ly/pO2Js*）。

10.3.2 Flow

Flow 是一个类型检查库，由 Facebook Open Source 小组开发和维护。这个工具通过静态类型注解检查错误。也就是说，我们在创建变量时指定其为特定的类型，Flow 就按照这一点检查使用的值是不是正确类型。

下面使用 Create React App 创建一个项目：

```
npx create-react-app in-the-flow
```

然后，要把 Flow 添加到项目中。Create React App 不假定你想使用 Flow，因此默认没有提供，不过自己安装也不难：

```
npm install --save flow-bin
```

安装好之后，添加一个 `npm` 脚本，通过 `npm run flow` 命令运行 Flow。打开 *package.json* 文件，在 `scripts` 中添加这个键：

```
{
```

```
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject",
    "flow": "flow"
  }
}
```

现在，执行 `flow` 命令即可对项目中的文件做类型检查。不过，在此之前还要创建一个 *.flowconfig* 文件。为此，运行：

```
npm run flow init
```

这个命令创建一个配置文件骨架，如下所示：

```
[ignore]

[include]

[libs]

[lints]

[options]

[strict]
```

多数情况下，各部分留空，使用 Flow 的默认配置即可。如果你想进一步配置 Flow，可以阅读文档（*https://flow.org/en/docs/config/*），深入了解各选项。

Flow 的一大特色是可以采用渐进的方式逐步应用 Flow。为整个项目添加类型检查功能是一个大工程。而使用 Flow，无须一步到位。我们只需在想做类型检查的文件顶部加上 `//@flow`，这样 Flow 便会自动检查有这一行内容的文件。

另外一种选择是安装 Flow 的 VSCode 扩展，使用代码补全和参数提示功能。如

果你安装了 Prettier 或 lint 工具，这个扩展可以让 VSCode 支持 Flow 非同一般的句法。这个扩展在市场中的地址为 *https://oreil.ly/zdaPv*。

简单起见，我们把所有代码都写在同一个文件中。打开 *index.js* 文件，在顶部添加 `//@flow`：

```
//@flow

import React from "react";
import ReactDOM from "react-dom";

function App(props) {
  return (
    <div>
      <h1>{props.item}</h1>
    </div>
  );
}

ReactDOM.render(
  <App item="jacket" />,
  document.getElementById("root")
);
```

接下来，定义属性的类型：

```
type Props = {
  item: string
};

function App(props: Props) {
  //...
}
```

然后，执行 `npm run flow` 命令，运行 Flow。在 Flow 的某些版本中，你可能会看到这个提醒：

```
Cannot call ReactDOM.render with root bound to container because null [1]
is incompatible with Element [2]
```

Flow 之所以发出这个提醒，是因为 document.getElementById("root") 返回 null 将导致应用崩溃。为了防止应用崩溃（清除错误），我们有两个选择。第一，使用 if 语句检查 root 是不是 null：

```
const root = document.getElementById("root");

if (root !== null) {
  ReactDOM.render(<App item="jacket" />, root);
}
```

另一种选择是使用 Flow 句法对 root 常量做类型检查：

```
const root = document.getElementById("root");

ReactDOM.render(<App item="jacket" />, root);
```

这两种方法都能清除这个错误，你会发现代码没有错误了。

一个错误也没有！

我们当然要信任 Flow，不过搞个破坏也不失为一个好主意。下面传给应用一个其他类型的属性：

```
ReactDOM.render(<App item={3} />, root);
```

搞破坏就是这么简单！我们看到的错误消息如下所示：

```
Cannot create App element because number [1] is incompatible with string [2]
in property item.
```

现在，把属性改回去，然后添加一个数字属性，同时调整组件和属性定义：

```
type Props = {
  item: string,
  cost: number
};
```

```
function App(props: Props) {
  return (
    <div>
      <h1>{props.item}</h1>
      <p>Cost: {props.cost}</p>
    </div>
  );
}

ReactDOM.render(
  <App item="jacket" cost={249} />,
  root
);
```

这段代码可以正常运行，但是如果去掉 cost 值呢？

```
ReactDOM.render(<App item="jacket" />, root);
```

无疑，我们会看到一个错误：

```
Cannot create App element because property cost is missing in props [1]
but exists in Props [2].
```

如果 cost 确实不是必要的值，在属性定义中可以在属性名称后面加上问号（cost?），把属性标记为可选的：

```
type Props = {
  item: string,
  cost?: number
};
```

再次运行上述代码，没有错误了。

以上只是 Flow 众多功能的冰山一角。如果想进一步学习、紧跟变化，请访问文档网站（*https://flow.org/en/docs/getting-started/*）。

10.3.3 TypeScript

在 React 应用中做类型检查，TypeScript 也是一个使用广泛的工具。开源的 TypeScript 是 JavaScript 的超集，为 JavaScript 语言添加了一些额外的特性。TypeScript 由 Microsoft 开发，旨在帮助开发者找出大型应用的 bug，并快速迭代项目。

TypeScript 的铁杆支持者不断增加，生态系统中的工具也不断完善。我们熟知的 Create React App 工具就提供有 TypeScript 模板。下面我们来设置一下，实现与 PropTypes 和 Flow 类似的基本类型检查功能，借此感受一下如何在应用中使用 TypeScript。

首先，使用 Create React App 再生成一个应用。这一次使用不同的标志：

```
npx create-react-app my-type --template typescript
```

我们来看一下生成的项目骨架。注意 *src* 目录中的文件，现在扩展名是 *.ts* 或 *.tsx*。此外，还有一个 *.tsconfig.json* 文件，存储所有 TypeScript 设置。后文细说。

另外，打开 *package.json* 文件，你会发现列出了一些与 TypeScript 有关的新依赖，包括 TypeScript 库本身和 Jest、React、ReactDOM 等的类型定义。以 `@types/` 开头的依赖就是某个库的类型定义。有了这些依赖，库中的函数和方法就有了类型，我们不用再自己动手设定类型。



如果你生成的项目不含 TypeScript 功能，可能是使用的 Create React App 版本较旧。解决方法是执行 `npm uninstall -g create-react-app` 命令。

接下来，把 Flow 节创建的组件添加到这个项目中。把下述内容添加到 *index.ts* 文件中：

```
import React from "react";
import ReactDOM from "react-dom";
```

```
function App(props) {
  return (
    <div>
      <h1>{props.item}</h1>
    </div>
  );
}

ReactDOM.render(
  <App item="jacket" />,
  document.getElementById("root")
);
```

现在执行 npm start 命令，运行这个项目，我们会看到第一个 TypeScript 错误。目前，这个错误在预料之中：

```
Parameter 'props' implicitly has an 'any' type.
```

这表明，我们要为 App 组件添加类型规则。我们可以按照之前使用 Flow 那样定义类型。item 是一个字符串，那就把这个信息添加到 AppProps 类型中：

```
type AppProps = {
  item: string;
};

ReactDOM.render(
  <App item="jacket" />,
  document.getElementById("root")
);
```


然后，在组件中引用 AppProps：

```
function App(props: AppProps) {
  return (
    <div>
      <h1>{props.item}</h1>
    </div>
  );
}
```

现在，这个组件可以正常渲染，没有与 TypeScript 有关的问题了。如果愿意，还可以析构 props：

```
function App({ item }: AppProps) {
  return (
    <div>
      <h1>{item}</h1>
    </div>
  );
}
```

若想破坏这个规则，可以把其他类型的值传给 item 属性：

```
ReactDOM.render(
  <App item={1} />,
  document.getElementById("root")
);
```

随即，我们将看到一个错误：

```
Type 'number' is not assignable to type 'string'.
```

错误消息还会指出问题具体出现在哪一行。这对调试有极大的帮助。

当然，TypeScript 的作用不止验证属性。我们可以使用 TypeScript 的类型推导（type inference）对钩子值做类型检查。

假设在一个组件中，fabricColor 状态的初始值为 purple，如下所示：

```
type AppProps = {
  item: string;
};

function App({ item }: AppProps) {
  const [fabricColor, setFabricColor] = useState(
    "purple"
  );
  return (
```

```
      <div>
        <h1>
          {fabricColor} {item}
        </h1>
        <button
          onClick={() => setFabricColor("blue")}
        >
          Make the Jacket Blue
        </button>
      </div>
    );
  }
```

注意，我们没有在类型定义对象中添加任何内容，经 TypeScript 推导，`fabricColor` 的类型应该与初始状态的类型一致。倘若试图把 `fabricColor` 的值设为一个数字，而不是 `blue` 之外的其他字符串，立刻就会看到一个错误：

```
<button onClick={() => setFabricColor(3)}>
```

抛出的错误是这样的：

```
Argument of type '3' is not assignable to parameter of type string.
```

TypeScript 对这个值所做的类型检查节省了相当多的人力。当然，我们可以进一步定制，参照本节的介绍逐步为应用添加类型检查功能。

关于 TypeScript 的更多信息，参阅官方文档（*https://oreil.ly/97_Px*）和 GitHub 中的 React+TypeScript Cheatsheets（*https://oreil.ly/vmran*）。

10.4 测试驱动开发

测试驱动开发（Test-driven development，TDD）是一种实践方式，而不是一项工程技术。TDD 也不仅仅是为应用编写测试，而是由测试驱动开发工作的实践过程。践行 TDD 要遵守下述步骤。

先编写测试

这是最重要的步骤。首先通过测试声明要构建的功能，指明功能应该怎样运作。测试的步骤是遇红、变绿和镀金。

运行测试，看着测试失败（遇红）

在开始编写代码之前运行测试，看着测试失败。

编写最少量的代码，让测试通过（变绿）

一次关注一个测试，让测试通过。不要添加超出测试覆盖范畴的功能。

重构代码和测试（镀金）

测试通过后再重新审视代码和测试，尽量使用简洁优雅的方式编写代码。[注2]

TDD 特别适合用于开发 React 应用，尤其方便测试钩子。在真正动手编写代码之前想好钩子的作用往往能达到事半功倍的效果。践行 TDD 可以让我们脱离 UI 去思考如何构建和验证一项功能或整个应用的数据结构。

边学习边践行 TDD

刚接触 TDD 或要测试的语言，在编写代码之前先写测试可能有点困难。这是正常现象，在熟练掌握之前，可以先写代码再写测试。步子不要迈得太大，写点儿代码，写点儿测试，如此反复。熟悉如何编写测试之后，你就不会觉得先编写测试有什么难度了。

本章余下的内容为现有的代码编写测试，严格来说，这不是 TDD。不过，在下一节，我们假装代码不存在，来感受一下 TDD 流程。

10.5 使用 Jest

着手编写测试之前，我们要选择一个测试框架。为 React 编写测试可以使用任何 JavaScript 测试框架，不过 React 官方文档推荐使用 Jest。这是一个 JavaScript

注 2：关于这个开发模式的更多信息，请阅读 Jeff McWherter 和 James Bender 写的文章“Red, Green, Refactor”（*https://oreil.ly/Hr6Me*）。

测试运行程序，可以通过 JSDOM 访问 DOM。不要小看访问 DOM 这个功能，因为我们要检查 React 渲染的内容，确保应用运作正常。

Create React App 和测试

使用 Create React App 生成的项目已经安装了 `jest` 包。我们可以使用 Create React App 再创建一个项目，或者使用现有的项目：

```
npx create-react-app testing
```

下面通过一个简单的例子说明如何测试。在 *src* 文件夹中新建两个文件：*functions.js* 和 *functions.test.js*。注意，Create React App 已经安装和配置好了 Jest，我们什么都不用做，直接编写测试即可。下面在 *functions.test.js* 文件中拟定测试，即表达出我们希望函数应该具有什么功能。

我们希望这个函数接受一个值，返回该值的两倍。测试就要把这个要求表达出来。`test` 是 Jest 提供的函数，用于测试某个具体功能。

functions.test.js

```
test("Multiplies by two", () => {
  expect();
});
```

第一个参数（`Multiplies by two`）是测试的名称。第二个参数是一个函数，提供要测试的内容。第三个参数可选，指定超时时间。默认的超时时间为 5 秒。

接下来，我们要初步定义这个函数，求一个数的两倍是多少。下文把这个函数称为被测系统（system under test，SUT）。在 *functions.js* 文件中定义这个函数：

```
export default function timesTwo() {...}
```

为了在测试中使用这个被测系统，要把它导出。在测试文件中，我们要导入这个函数，然后使用 `expect` 编写一个断言（assertion）。在断言中，我们表达的意思是，如果把 4 传给 `timesTwo` 函数，预期返回 8：

```
import { timesTwo } from "./functions";

test("Multiplies by two", () => {
  expect(timesTwo(4)).toBe(8);
});
```

在 Jest 中，expect 函数返回一个“匹配器”（matcher），用于验证结果。测试这个函数，使用的是 .toBe 匹配器，确认得到的对象与发给 .toBe 的参数是一致的。

现在，执行 npm test 或 npm run test 命令，运行测试，看着测试失败。Jest 为每个失败测试提供了详细信息，包括堆栈跟踪：

```
FAIL  src/functions.test.js
  × Multiplies by two (5ms)

  ● Multiplies by two

    expect(received).toBe(expected) // Object.is equality

    Expected: 8
    Received: undefined

      2 |
      3 | test("Multiplies by two", () => {
    > 4 |   expect(timesTwo(4)).toBe(8);
        |                       ^
      5 | });
      6 |

      at Object.<anonymous> (src/functions.test.js:4:23)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        1.048s
Ran all test suites related to changed files.
```

花时间编写测试，然后运行测试，看着测试失败，这个过程表明测试真正起作

用了。失败反馈就是我们的待办事项清单。我们要编写最少量的代码，让测试通过。

现在，打开*functions.js*文件，添加适当的功能，让测试通过：

```
export function timesTwo(a) {
  return a * 2;
}
```

.toBe匹配器测试的是与单个值是否相等。如果想测试一个对象或一个数组，要使用.toEqual。下面再走一遍测试循环，测试与一个对象数组是否相等。

假设我们手上有拉斯维加斯Guy Fieri餐馆的菜单。我们要构建一个对象，包含顾客点的菜品，计算要付多少钱。首先草拟测试：

```
test("Build an order object", () => {
  expect();
});
```

然后，初步定义函数：

```
export function order(items) {
  // ...
}
```

接下来，在测试文件中使用order函数。我们将使用一个假想订单，对这些数据做转换：

```
import { timesTwo, order } from "./functions";

const menuItems = [
  {
    id: "1",
    name: "Tatted Up Turkey Burger",
    price: 19.5
  },
  {
```

```
    id: "2",
    name: "Lobster Lollipops",
    price: 16.5
  },
  {
    id: "3",
    name: "Motley Que Pulled Pork Sandwich",
    price: 21.5
  },
  {
    id: "4",
    name: "Trash Can Nachos",
    price: 19.5
  }
];

test("Build an order object", () => {
  expect(order(menuItems));
});
```

注意，我们要使用 toEqual，因为这里检查的是对象的值，而不是数组的值。我们希望结果等于什么呢？我们想创建一个如下的对象：

```
const result = {
  orderItems: menuItems,
  total: 77
};
```

那就把这个对象添加到测试中，然后在断言中使用：

```
test("Build an order object", () => {
  const result = {
    orderItems: menuItems,
    total: 77
  };
  expect(order(menuItems)).toEqual(result);
});
```

接下来，在 *functions.js* 文件中完善函数的定义：

```
export function order(items) {
  const total = items.reduce(
    (price, item) => price + item.price,
    0
  );
  return {
    orderItems: items,
    total
  };
}
```

在终端里运行测试，你会发现所有测试都能通过。或许你会觉得这个例子没什么大不了的，但是在获取数据时，我们有可能就要测试数组和对象的结构。

另外一个常用的 Jest 函数是 `describe()`。如果你用过其他测试库，可能见过类似的函数。这个函数通常用于组织多个相关的测试。比如说，我们可以使用 describe 语句把几个针对类似函数的测试组织在一起：

```
describe("Math functions", () => {
  test("Multiplies by two", () => {
    expect(timesTwo(4)).toBe(8);
  });
  test("Adds two numbers", () => {
    expect(sum(4, 2)).toBe(6);
  });
  test("Subtracts two numbers", () => {
    expect(subtract(4, 2)).toBe(2);
  });
});
```

使用 describe 语句把测试组织在一起之后，测试运行程序会把这些测试归在一起，终端输出的测试报告更有条理，更方便阅读：

```
Math functions
    ✓ Multiplies by two
    ✓ Adds two numbers
    ✓ Subtracts two numbers (1ms)
```

测试越写越多，使用 describe 块分组是一项不错的增强措施。

以上过程就是典型的TDD循环。我们先编写测试，然后编写让测试通过的代码。测试通过之后，我们可以进一步审视代码，找出可以重构的地方，简化代码、提升性能。对JavaScript来说，TDD是一个行之有效的开发方式（其实对任何语言来说都是如此）。

10.6 测试React组件

现在，我们对测试的过程有了一些基本的认识，下面运用这些知识来测试React组件。

React组件是对React所下的指令，用于创建和更新DOM。因此，对组件的测试首先要渲染组件，然后检查得到的DOM。

组件测试不在浏览器中运行，而是在终端里使用Node.js运行。Node.js没有提供与各浏览器兼容的DOM API，不过Jest集成的jsdom包可以在Node.js中模拟浏览器环境，这是测试React组件必不可少的。

组件测试基本上都要把React组件树渲染成DOM元素。下面以Star.js文件中的Star组件为例说明整个流程。

```
import { FaStar } from "react-icons/fa";

export default function Star({ selected = false }) {
  return (
    <FaStar color={selected ? "red" : "grey"} id="star" />
  );
}
```

然后，在*index.js*文件中导入并渲染星标：

```
import Star from "./Star";

ReactDOM.render(
  <Star />,
  document.getElementById("root")
);
```

接下来编写测试。星标的代码已经编写好了，因此这里采用的不是 TDD 技术。为现有应用增加测试，过程就是这样的。新建 *Star.test.js* 文件，导入 React、ReactDOM 和 Star 组件：

```
import React from "react";
import ReactDOM from "react-dom";
import Star from "./Star";

test("renders a star", () => {
  const div = document.createElement("div");
  ReactDOM.render(<Star />, div);
});
```

我们还编写了一个测试。注意，提供给 `test` 函数的第一个参数是测试的名称。然后，我们做些准备工作，创建一个 `div` 元素，供 `ReactDOM.render` 渲染星标。创建好元素之后，编写断言：

```
test("renders a star", () => {
  const div = document.createElement("div");
  ReactDOM.render(<Star />, div);
  expect(div.querySelector("svg")).toBeTruthy();
});
```

我们预计在这个 `div` 元素中可以找到一个 `svg` 元素。运行测试，你会发现这个测试可以通过。为了确保断言不在不该通过时通过，我们可以把选择的目标修改为一个不存在的元素，看测试是否失败：

```
expect(
  div.querySelector("notrealthing")
).toBeTruthy();
```

文档（*https://oreil.ly/ah7ZU*）对各个自定义匹配器做了详细说明，请仔细阅读，了解在测试中有哪些匹配器可用。

生成 React 项目时你可能注意到了，除了基本的 React 和 ReactDOM 之外，还安装了一些 `@testing-library` 名下的包。React Testing Library 是 Kent C.

Dodds 发起的一个项目，旨在帮助我们实施好的测试实践，增强 React 生态环境中有关测试工具的功能。Testing Library 是一个综合项目，不仅对 React，还为 Vue、Svelte、Reason、Angular 等库提供了测试增强包。

React Testing Library 吸引人的因素之一是测试失败消息更详尽。对下述测试断言来说：

```
expect(
  div.querySelector("notrealthing")
).toBeTruthy();
```

目前的错误消息是：

```
expect(received).toBeTruthy()

Received: null
```

下面添加 React Testing Library，增强功能。这个库在使用 Create React App 生成项目时已经安装了。首先，从 `@testing-library/jest-dom` 中导入 `toHaveAttribute` 函数：

```
import { toHaveAttribute } from "@testing-library/jest-dom";
```

然后，增强 `expect` 的功能，包含这个函数：

```
expect.extend({ toHaveAttribute });
```

现在，我们不再使用错误消息含糊其词的 `toBeTruthy`，转而使用 `toHaveAttribute`：

```
test("renders a star", () => {
  const div = document.createElement("div");
  ReactDOM.render(<Star />, div);
  expect(
    div.querySelector("svg")
  ).toHaveAttribute("id", "hotdog");
});
```

然后运行测试，你会发现错误消息指出了问题的真正所在：

```
    expect(element).toHaveAttribute("id", "hotdog")
    // element.getAttribute("id") === "hotdog"

    Expected the element to have attribute:
      id="hotdog"
    Received:
      id="star"
```

我们也就十分明确该怎么修正：

```
expect(div.querySelector("svg")).toHaveAttribute(
  "id",
  "star"
);
```

如果想使用其他自定义匹配器，过程是一样的，即导入、扩展和使用：

```
import {
  toHaveAttribute,
  toHaveClass
} from "@testing-library/jest-dom";

expect.extend({ toHaveAttribute, toHaveClass });

expect(you).toHaveClass("evenALittle");
```

不过，还有个简便的方法。如果导入的匹配器太多，不想一一列出或记住，可以导入 extend-expect 库：

```
import "@testing-library/jest-dom/extend-expect";

// 删除这一行 --> expect.extend({ toHaveAttribute, toHaveClass });
```

断言依然可以按预期运行。其实，Create React App 在它生成的 *setupTests.js* 文件中有一行代码，已经导入了 extend-expect 库提供的自定义匹配器。打开 src 文件夹中的 *setupTests.js* 文件，你会看到如下内容：

```
// jest-dom adds custom jest matchers for asserting on DOM nodes.
// allows you to do things like:
// expect(element).toHaveTextContent(/react/i)
// learn more: https://github.com/testing-library/jest-dom
import "@testing-library/jest-dom/extend-expect";
```

因此，使用 Create React App 的话甚至不用在测试文件中导入这个库。

10.6.1 查询

React Testing Library 的另一个功能是查询（query），即根据条件匹配。为了演示查询的用法，下面稍微调整一下 `Star` 组件，加入标题。然后，我们便可以编写一种常见的测试——根据文本匹配。

```
export default function Star({ selected = false }) {
  return (
    <>
      <h1>Great Star</h1>
      <FaStar
        id="star"
        color={selected ? "red" : "grey"}
      />
    </>
  );
}
```

暂停，我们先想一下要测试的是什么。我们要渲染组件，然后测试 `h1` 中是否包含正确的文本。React Testing Library 提供的 `render` 函数正好能为我们提供帮助。我们将使用 `render` 替代 `ReactDOM.render()`，因此测试有些许差别。先从 React Testing Library 中导入 `render`：

```
import { render } from "@testing-library/react";
```

`render` 接受一个参数，即要渲染的组件或元素。该函数返回一个查询对象，我们就通过这个对象检查组件或元素中的值。我们要使用的查询是 `getByText`，通

过它找到匹配查询的第一个节点；如果没有元素匹配，抛出一个错误。如果想返回匹配的所有结点，使用返回一个数组的 getAllBy。

```
test("renders an h1", () => {
  const { getByText } = render(<Star />);
  const h1 = getByText(/Great Star/);
  expect(h1).toHaveTextContent("Great Star");
});
```

getByText 通过传入的正则表达式查找 h1 元素。然后，使用 Jest 匹配器 toHaveTextContent 指明 h1 中应该有什么文本。

现在运行测试，都能通过。倘若修改传给 toHaveTextContent() 函数的文本，这个测试将失败。

10.6.2 测试事件

测试往往还涉及组件中的事件。以第 7 章创建的 Checkbox 组件为例：

```
export function Checkbox() {
  const [checked, setChecked] = useReducer(
    checked => !checked,
    false
  );

  return (
    <>
      <label>
        {checked ? "checked" : "not checked"}
        <input
          type="checkbox"
          value={checked}
          onChange={setChecked}
        />
      </label>
    </>
  );
}
```

这个组件使用 useReducer 改变复选框的勾选状态。这里，我们想通过一个自动化测试点击该复选框，把 checked 的值由默认的 false 改为 true。编写检查这个复选框的测试也能触发 useReducer，从而测试该钩子。

下面来草拟测试：

```
import React from "react";

test("Selecting the checkbox should change the value of checked to true", () => {
  // .. 编写一个测试
});
```

我们要做的第一件事是选择想触发事件的元素。也就是说，指明让自动化测试点击哪个元素。我们将使用 Testing Library 提供的一个查询查找元素。由于目标输入框有一个标注（label），因此我们可以使用 getByLabelText()。

```
import { render } from "@testing-library/react";
import { Checkbox } from "./Checkbox";

test("Selecting the checkbox should change the value of checked to true", () => {
  const { getByLabelText } = render(<Checkbox />);
});
```

这个组件首次渲染时，标注文本是 not checked，因此我们可以通过正则表达式查找匹配的字符串：

```
test("Selecting the checkbox should change the value of checked to true", () => {
  const { getByLabelText } = render(<Checkbox />);
  const checkbox = getByLabelText(/not checked/);
});
```

这里使用的正则表达式区分大小写，如果不需要区分，可以在正则表达式末尾加上 i。要不要这么做取决于查询的宽松程度。

```
const checkbox = getByLabelText(/not checked/i);
```

现在，目标复选框找到了。接下来，我们只需触发事件，编写一个断言，确认单击该复选框后 checked 属性的值变成了 true。

```
mport { render, fireEvent } from "@testing-library/react"

test("Selecting the checkbox should change the value of checked to true", () => {
  const { getByLabelText } = render(<Checkbox />);
  const checkbox = getByLabelText(/not checked/i);
  fireEvent.click(checkbox);
  expect(checkbox.checked).toEqual(true);
});
```

我们还可以反过来测试，检查再次触发事件后 checked 属性的值有没有被设为 false。我们修改了测试的名称，更准确地表明测试的意图。

```
test("Selecting the checkbox should toggle its value", () => {
  const { getByLabelText } = render(<Checkbox />);
  const checkbox = getByLabelText(/not checked/i);
  fireEvent.click(checkbox);
  expect(checkbox.checked).toEqual(true);
  fireEvent.click(checkbox);
  expect(checkbox.checked).toEqual(false);
});
```

这里，选择目标复选框十分容易，我们可以通过标注找到想勾选的输入框。在不太容易访问 DOM 元素的情况下，我们可以使用 Testing Library 提供的另一个函数搜索 DOM 元素。首先，为要选择的元素添加一个属性：

```
<input
  type="checkbox"
  value={checked}
  onChange={setChecked}
  data-testid="checkbox" // 添加 data-testid= 属性
/>
```

然后使用 getByTestId 查询：

```
test("Selecting the checkbox should change the value of checked to true", () => {
  const { getByTestId } = render(<Checkbox />);
```

```
    const checkbox = getByTestId("checkbox");
    fireEvent.click(checkbox);
    expect(checkbox.checked).toEqual(true);
});
```

这样做的效果一样，但是在不太方便访问 DOM 元素时特别有用。

测试过 Checkbox 组件之后，我们便可以放心在应用中反复使用它了。

10.6.3 代码覆盖度

代码覆盖度（code coverage）指具体对多少行代码做了测试的统计，通过这个指标可以判断有没有编写足够的测试。

Jest 提供的 Istanbul 是一个 JavaScript 工具，用于审查测试，生成一份报告，指明覆盖了多少语句、分支、函数和代码行。

若想让 Jest 检查代码覆盖度，在运行 test 命令时加上 coverage 标志即可：

```
npm test -- --coverage
```

生成的报告将指出在测试过程中各个文件（包括导入测试的所有文件）中的代码执行了多少。

Jest 还会生成一份可在浏览器中查看的报告，进一步说明测试覆盖了哪些代码。Jest 生成覆盖度报告后，根目录中会出现一个 *coverage* 文件夹。在浏览器中打开 */coverage/lcov-report/index.html* 文件，这是一份交互式报告，具体说明代码覆盖度。

这份报告指出总体覆盖了多少代码，以及各子文件夹的覆盖情况。我们可以只关注某个子文件夹，查看其中各文件的覆盖报告。假如你选择 *components/ui* 文件夹，那么看到的就是用户界面组件的测试覆盖度。

单击文件名还可以查看各文件中有哪些代码行被覆盖了。

代码覆盖度可以衡量测试的范围。这个指标有助于我们判断有没有为代码编写足够的测试。一般来说，不是每个项目都要求具有 100% 的代码覆盖度，超过 85% 就算不错了。[注 3]

人们往往认为测试是多此一举，可是如今 React 相关的测试工具佳作频出，不做测试的借口似乎又少了一个。即使不为所有代码编写测试，仅以测试思维去思考问题也能节省时间和金钱，为应用顺利上线助一臂之力。

注 3：参见 Martin Fowler 写的“Test-Coverage”一文（*https://oreil.ly/Hbb-D*）。

第 11 章

React Router

Web 出现之时，多数网站是由一系列单独的页面构成的，用户若想查看其他页面，要请求打开另外的文件。当前文件或资源的地址显示在浏览器的地址栏中。浏览器的前进和后退按钮能按预期工作。如果用户想保存一份对特定文件的引用，可以把网站中特定的页面加入收藏夹，在需要查看时再次打开。对这样由不同页面构成，或者说由服务器渲染的网站来说，浏览器的导航按钮和历史记录都能按预期工作。

而在单页应用中，这些功能都将出现问题。别忘了，单页应用的一切内容都在同一个页面中，由 JavaScript 负责加载信息和变换 UI。如果没有路由方案，浏览器的很多功能，例如历史记录、收藏夹及前进和后退按钮都无法使用。路由为客户端请求定义端点，[注 1] 这些端点兼容浏览器的地址和历史记录对象。端点用于标识请求的内容，以便 JavaScript 加载和渲染相应的用户界面。

与 Angular、Ember 或 Backbone 不同，React 没有自带路由器。认识到路由方案的重要性之后，Michael Jackson 和 Ryan Florence 两位工程师开发出了 React

注 1：参见 Express.js 文档中的“Basic routing”一文（*https://oreil.ly/jD1HC*）。

Router。随后，React Router 在社区中广泛使用，成为 React 应用一个流行的路由方案。[注2] Uber、Zendesk、PayPal 和 Vimeo 等公司也在使用。[注3]

本章介绍 React Router，说明如何利用该库提供的功能在客户端处理路由。

11.1 使用 React Router

为了演示 React Router 的功能，我们将构建一个典型的简易网站，包括“About”“Events”“Products”和“Contact Us”部分。这个网站看似包含多个页面，但其实只有一个页面，这是一个单页应用（single-page application，SPA），如图 11-1 所示。

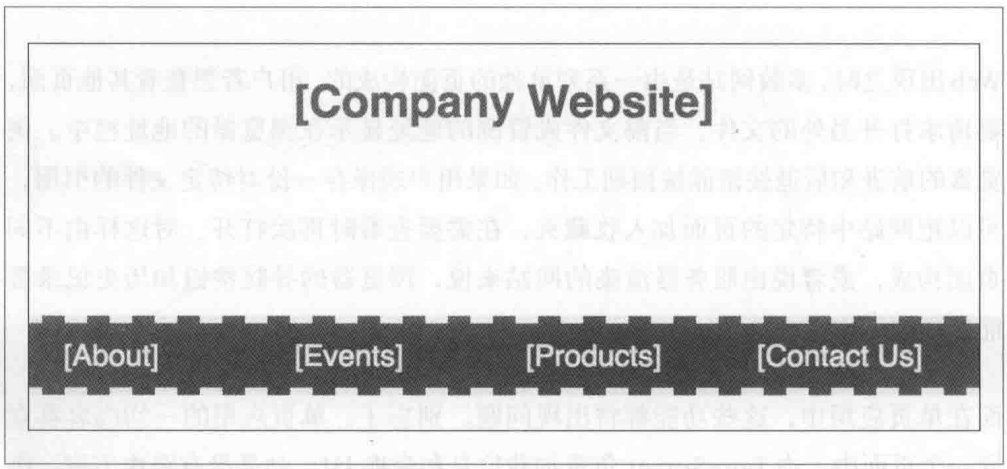

图 11-1：一个简易网站中的链接导航

这个网站的站点地图包含一个首页，首页中有各个部分，另外还有一个错误页面，用于处理 404 Not Found 错误（见图 11-2）。

路由器的作用是为网站中的各部分设置路由。一个路由就是一个端点，即可在浏览器的地址栏中输入的地址。请求某个路由，应用将渲染相应的内容。

注 2：这个项目在 GitHub 中的星标数超过 20000（*https://oreil.ly/TNG9*）。

注 3：参见“Sites Using React Router”（*https://oreil.ly/staEF*）。

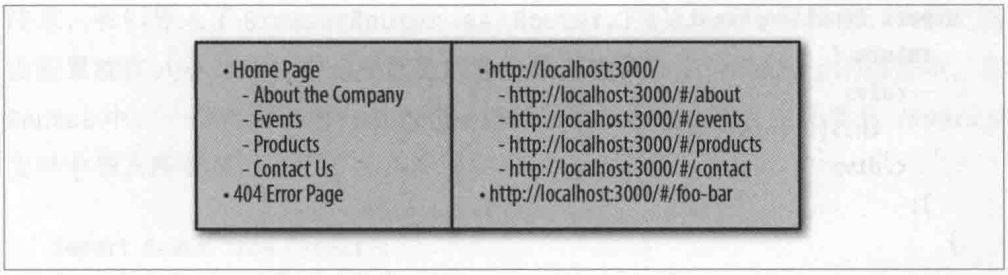

图 11-2：网站地图（本地地址）

首先，我们要安装 React Router 和 React Router DOM。React Router DOM 供使用 DOM 的常规 React 应用使用。如果你编写的是 React Native 应用，要使用 `react-router-native`。我们将安装这两个包的实验版本，因为本书付梓之时 React Router 6 还没有正式发布。等到正式发布后，你就不用再使用实验版本了。

```
npm install react-router@experimental react-router-dom@experimental
```

接下来要为站点地图中的各部分或页面创建几个占位组件。这些组件可以集中在一个名为 *pages.js* 的文件中导出。

```
import React from "react";

export function Home() {
  return (
    <div>
      <h1>[Company Website]</h1>
    </div>
  );
}

export function About() {
  return (
    <div>
      <h1>[About]</h1>
    </div>
  );
}
```

```
export function Events() {
  return (
    <div>
      <h1>[Events]</h1>
    </div>
  );
}

export function Products() {
  return (
    <div>
      <h1>[Products]</h1>
    </div>
  );
}

export function Contact() {
  return (
    <div>
      <h1>[Contact]</h1>
    </div>
  );
}
```

草拟好这些页面之后，我们要调整 *index.js* 文件。现在，我们不再渲染 App 组件，而是渲染 Router 组件。Router 组件把当前地址的有关信息传给内部嵌套的子组件。Router 组件只能使用一次，而且应该放在组件树中靠近根的位置上。

```
import React from "react";
import { render } from "react-dom";
import App from "./App";

import { BrowserRouter as Router } from "react-router-dom";

render(
  <Router>
    <App />
  </Router>,
  document.getElementById("root")
);
```

注意，我们导入了 BrowserRouter as Router。接下来要配置路由。我们把路由配置放在 *App.js* 文件中。要渲染的各个路由嵌套在名为 Routes 的组件中。在 Routes 中，一个页面使用一个 Route 组件设置。在这个文件中还要从 *./pages.js* 文件中导入所有页面。

```
import React from "react";
import { Routes, Route } from "react-router-dom";
import {
  Home,
  About,
  Events,
  Products,
  Contact
} from "./pages";

function App() {
  return (
    <div>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route
          path="/about"
          element={<About />}
        />
        <Route
          path="/events"
          element={<Events />}
        />
        <Route
          path="/products"
          element={<Products />}
        />
        <Route
          path="/contact"
          element={<Contact />}
        />
      </Routes>
    </div>
  );
}
```

这些路由告诉 React Router 在地址发生变化后渲染什么组件。每个 Route 组件都有 path 和 element 属性。当浏览器中的地址匹配某个 path 时，显示对应的 element。如果地址是 /，路由器渲染 Home 组件；如果地址是 /products，路由器渲染 Products 组件。

现在，我们可以运行应用，然后自己动手在浏览器的地址栏中输入路由，你会发现显示的内容会相应发生变化。例如，在地址栏中输入 *http://localhost:3000/about*，渲染的是 About 组件。

让用户自己动手在地址栏中输入路由或许不太现实。我们可以使用 react-router-dom 提供的 Link 组件创建链接。

下面来修改首页，加入指向各个路由的导航链接目录：

```
import { Link } from "react-router-dom";

export function Home() {
  return (
    <div>
      <h1>[Company Website]</h1>
      <nav>
        <Link to="about">About</Link>
        <Link to="events">Events</Link>
        <Link to="products">Products</Link>
        <Link to="contact">Contact Us</Link>
      </nav>
    </div>
  );
}
```

现在，用户在首页中单击链接就可以查看各个内部页面，而且单击浏览器的后退按钮能把用户带回首页。

11.2 React Router 属性

React Router 把属性传给它渲染的组件。比如说，可以通过属性获取当前地址。下面使用当前地址创建一个 404 Not Found 组件。首先，创建这个组件：

```
export function Whoops404() {
  return (
    <div>
      <h1>Resource not found</h1>
    </div>
  );
}
```

然后，把这个组件添加到 *App.js* 文件里的路由配置中。倘若用户访问一个不存在的路由，例如 `highway`，应用将显示 `Whoops404` 组件。这个 `Route` 设置的 `path` 值为 `*`，`element` 值为 `Whoops404` 组件。

```
function App() {
  return (
    <div>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route
          path="/about"
          element={<About />}
        />
        <Route
          path="/events"
          element={<Events />}
        />
        <Route
          path="/products"
          element={<Products />}
        />
        <Route
          path="/contact"
          element={<Contact />}
        />
        <Route path="*" element={<Whoops404 />} />
```

```
        </Routes>
      </div>
    );
  }
```

现在访问 *localhost:3000/highway*，你会看到渲染的是这个 404 页面组件。此外，我们可以使用当前地址值显示用户访问的路由。我们生活在一个有 React Hooks 的世界中，恰好有一个钩子可以做到这一点。在 `Whoops404` 组件中创建一个名为 `location` 的变量，返回当前的地址值（即有关当前访问页面的属性）。然后，使用 `location.pathname` 的值显示用户访问的路由。

```
export function Whoops404() {
  let location = useLocation();
  console.log(location);
  return (
    <div>
      <h1>
        Resource not found at {location.pathname}
      </h1>
    </div>
  );
}
```

在控制台中输出 `location` 可以进一步探索这个对象。

本节介绍了 React Router 的基本实现和使用方法。`Router` 组件只能使用一次，要路由的组件都嵌套在这个组件中。所有 `Route` 组件都嵌套在 `Routes` 组件中，根据浏览器中的当前地址选择渲染哪个组件。我们可以使用 `Link` 组件实现导航链接。这些基本知识就够用了，但只是 React Router 众多功能中的一小部分。

嵌套路由

`Route` 组件只在匹配特定的 URL 时显示对应的内容。利用这个功能可以按照严格的层次结构组织 Web 应用中的不同页面，促进内容重用。

有时候，用户在应用中浏览时，我们想固定 UI。过去，Web 开发人员可以使用页面模板和主页面重用 UI 元素。

以前面那个简易的网站为例。我们可能想为“About”页面创建一些子页面，显示更多的内容。用户访问“About”页面时，默认显示该部分下的“Company”页面。页面大纲如下所示：

- Home Page
  - **About the Company**
    - **Company（默认）**
    - **History**
    - **Services**
    - **Location**
  - Events
  - Products
  - Contact Us
- 404 Error Page

对应的，我们要创建如下的路由层次结构：

- *http://localhost:3000/*
  - *http://localhost:3000/about*
    - *http://localhost:3000/about*
    - *http://localhost:3000/about/history*
    - *http://localhost:3000/about/services*
    - *http://localhost:3000/about/location*
  - *http://localhost:3000/events*
  - *http://localhost:3000/products*
  - *http://localhost:3000/contact*

- *http://localhost:3000/hot-potato*

另外，还要为这些新部分草拟占位组件，包括：Company、Services、History和Location。例如，下面是Services组件的文本内容，在编写其他组件时可以重复使用：

```
export function Services() {
  <section>
    <h2>Our Services</h2>
    <p>
      Lorem ipsum dolor sit amet, consectetur
      adipiscing elit. Integer nec odio. Praesent
      libero. Sed cursus ante dapibus diam. Sed
      nisi. Nulla quis sem at nibh elementum
      imperdiet. Duis sagittis ipsum. Praesent
      mauris. Fusce nec tellus sed augue semper
      porta. Mauris massa. Vestibulum lacinia arcu
      eget nulla. Class aptent taciti sociosqu ad
      litora torquent per conubia nostra, per
      inceptos himenaeos. Curabitur sodales ligula
      in libero.
    </p>
  </section>;
}
```

创建好这些组件之后，打开*App.js*文件，配置路由器。创建具有层次结构的页面路由，只需在Route组件中嵌套Route组件：

```
import {
  Home,
  About,
  Events,
  Products,
  Contact,
  Whoops404,
  Services,
  History,
  Location
} from "./pages";
```

```
function App() {
  return (
    <div>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="about" element={<About />}>
          <Route
            path="services"
            element={<Services />}
          />
          <Route
            path="history"
            element={<History />}
          />
          <Route
            path="location"
            element={<Location />}
          />
        </Route>
        <Route
          path="events"
          element={<Events />}
        />
        <Route
          path="products"
          element={<Products />}
        />
        <Route
          path="contact"
          element={<Contact />}
        />
        <Route path="*" element={<Whoops404 />} />
      </Routes>
    </div>
  );
}
```

把相关路由嵌套到“About”页面的 `Route` 组件中之后，我们就可以访问这些页面了。现在打开 *http://localhost:3000/about/history*，看到的是“About”页面

的内容，但是没有显示History组件。若想显示那个组件，我们要使用React Router DOM的另一个特性：Outlet组件。嵌套组件就使用Outlet渲染。想渲染子页面内容使用这个组件即可。

打开*pages.js*文件，在About组件中的<h1>下面添加这个组件：

```
import {
  Link,
  useLocation,
  Outlet
} from "react-router-dom";

export function About() {
  return (
    <div>
      <h1>[About]</h1>
      <Outlet />
    </div>
  );
}
```

现在，整个“About”页面都将重用About组件，分别显示各个嵌套的组件。应用通过地址判断该显示哪个子部分。例如，访问*http://localhost:3000/about/history*地址，渲染的是About组件内部的History组件。

11.3 重定向

有时，我们想把用户从一个路由重定向到另一个路由。比如说，我们可能希望把访问*http://localhost:3000/services*的用户重定向到正确的路由*http://localhost:3000/about/services*上。

下面来修改应用，添加一个重定向设置，确保用户能访问到正确的内容：

```
import {
  Routes,
  Route,
```

```
  Redirect
} from "react-router-dom";

function App() {
  return (
    <div>
      <Routes>
        <Route path="/" element={<Home />} />
        // 其他路由
        <Redirect
          from="services"
          to="about/services"
        />
      </Routes>
    </div>
  );
}
```

Redirect 组件的作用是把用户重定向到指定的路由。

如果线上应用的路由发生变化，用户可能会通过旧路由访问原有内容。比如说，用户收藏了某个地址就有可能发生这种情况。Redirect 组件可以确保加载正确的内容，呈现给用户，即便用户是通过收藏夹中的旧地址访问的。

本章始终使用 Route 组件配置路由。如果你喜欢这种结构，请忽略下一节。不过也可以看一下，了解配置路由的其他方式。此外，也可以使用 useRoutes 钩子配置应用的路由。

如果你想使用 useRoutes 重构应用，要调整一下 App 组件（或者路由所在的其他组件）。下面来重构：

```
import { useRoutes } from "react-router-dom";

function App() {
  let element = useRoutes([
    { path: "/", element: <Home /> },
    {
      path: "about",
```

```
      element: <About />,
      children: [
        {
          path: "services",
          element: <Services />
        },
        { path: "history", element: <History /> },
        {
          path: "location",
          element: <Location />
        }
      ]
    },
    { path: "events", element: <Events /> },
    { path: "products", element: <Products /> },
    { path: "contact", element: <Contact /> },
    { path: "*", element: <Whoops404 /> },
    {
      path: "services",
      redirectTo: "about/services"
    }
  ]);
  return element;
}
```

官方文档把路由配置命名为 element，不过你也可以使用其他你喜欢的名称。另外，你也不一定非要使用这个句法。Route 是对 useRoutes 的包装，其实使用哪种句法都可以。请根据自己的喜好选择句法和风格。

路由参数

React Router 的另一个有用特性是为路由设置参数。路由参数（routing parameter）是一种变量，作用是从 URL 中获取相应的值。路由参数在数据驱动型 Web 应用中特别有用，可用于筛选内容或管理显示偏好设置。

下面我们以前面的颜色管理应用为例，通过 React Router 添加选择和显示单个颜色的功能。用户点击选择一个颜色后，应用渲染该颜色，并显示颜色的 title 和 hex 值。

有了路由器，我们可以从 URL 中获取颜色的 ID。例如，我们可以通过下面的 URL 访问“草绿色”，因为 URL 中有这个颜色的 ID：

```
http://localhost:3000/58d9caee-6ea6-4d7b-9984-65b145031979
```

首先，打开 *index.js* 文件，设置路由器。我们要导入 Router，嵌套 App 组件：

```
import { BrowserRouter as Router } from "react-router-dom";

render(
  <Router>
    <App />
  </Router>,
  document.getElementById("root")
);
```

嵌套 App 组件之后，路由的所有属性都将传给它及它的子组件。路由就在 App 组件中配置。我们不使用 useRoutes 钩子，而是使用 Routes 和 Route 组件，当然如果你想使用那种句法也不是不可以。首先导入 Routes 和 Route：

```
import { Routes, Route } from "react-router-dom";
```

然后，在 App 中添加路由。这个应用有两个路由：ColorList 和 ColorDetails。我们还没有编写 ColorDetails 组件，先导入：

```
import { ColorDetails } from "./ColorDetails";

export default function App() {
  return (
    <ColorProvider>
      <AddColorForm />
      <Routes>
        <Route
          path="/"
          element={<ColorList />}
        />
        <Route
          path=":id"
```

```
            element={<ColorDetails />}
          />
        </Routes>
      </ColorProvider>
    );
  }
```

ColorDetails组件根据颜色的id动态显示内容。新建一个文件，命名为*ColorDetails.js*，在这个文件中定义ColorDetails组件。先编写一些占位内容：

```
import React from "react";

export function ColorDetails() {
  return (
    <div>
      <h1>Details</h1>
    </div>
  );
}
```

我们怎么知道这样做是可行的呢？最简单的方法是打开React开发者工具，找到渲染出来的某个颜色的id。如果应用中还没有颜色，那就添加一个，然后再查看id。获得id之后，把ID值添加到localhost:3000 URL后面，例如localhost:3000/00fdb4c5-c5bd-4087-a48f-4ff7a9d90af8。

现在，你会看到ColorDetails页面。这说明路由器和路由设置是正确的，不过我们想要的是动态内容。我们希望在ColorDetails页面显示URL中id值对应的颜色。为此，要使用useParams钩子：

```
import { useParams } from "react-router-dom";

export function ColorDetails() {
  let params = useParams();
  console.log(params);
  return (
    <div>
      <h1>Details</h1>
    </div>
```

```
  );
}
```

通过输出到控制台的 params 可以看到，这个对象包含路由器传入的所有参数。我们要析构这个对象，从中获取 id，然后通过 id 在 colors 数组中找到对应的颜色。下面使用 useColors 钩子实现：

```
import { useColors } from "./";

export function ColorDetails() {
  let { id } = useParams(); // 析构 id

  let { colors } = useColors();

  let foundColor = colors.find(
    color => color.id === id
  );
  console.log(foundColor);

  return (
    <div>
      <h1>Details</h1>
    </div>
  );
}
```

输出到控制台的 foundColor 表明我们找到了正确的颜色。现在，我们要做的是在 ColorDetails 组件中显示这个颜色的相关数据：

```
export function ColorDetails() {
  let { id } = useParams();
  let { colors } = useColors();

  let foundColor = colors.find(
    color => color.id === id
  );

  return (
    <div>
```

```
        <div
          style={{
            backgroundColor: foundColor.color,
            height: 100,
            width: 100
          }}
        ></div>
        <h1>{foundColor.title}</h1>
        <h1>{foundColor.color}</h1>
      </div>
    );
  }
```

我们想为这个颜色管理应用添加的另一个功能是，点击颜色列表中的某个颜色后打开 ColorDetails 页面。下面在 Color 组件中添加这个功能。为此，我们要使用另一个路由器钩子 useNavigate，在用户点击该组件时打开详情页面。先从 react-router-dom 中导入这个钩子：

```
import { useNavigate } from "react-router-dom";
```

然后，调用 useNavigate 钩子，返回一个函数，导航到另一个页面：

```
let navigate = useNavigate();
```

接下来，为 section 元素添加一个 onClick 处理函数，根据颜色的 id 导航到正确的路由：

```
let navigate = useNavigate();

return (
  <section
    className="color"
    onClick={() => navigate(`/${id}`)}
  >
    // Color 组件
  </section>
);
```

现在，单击这个 section 元素就可以转到正确的页面了。

路由参数是获取影响用户界面呈现内容所需数据的理想工具。然而，我们应该只在希望用户从 URL 中捕获这些信息时才使用。例如，对这个颜色管理应用来说，一个用户可以把某个颜色的 URL，或者按某个字段排序的颜色列表 URL 发给另一个用户。用户也可以把返回特定数据的链接放入收藏夹。

本章介绍了 React Router 的基本用法。下一章学习如何在服务器端使用路由。

第 12 章

React 服务器端渲染

目前，我们使用 React 构建的小型应用都完全在浏览器中运行，在浏览器中收集数据，把数据保存到浏览器的存储器中。这也说得通，毕竟 React 负责的是视图层，功用是渲染 UI。然而，多数应用都至少需要有某种后端的存在，因此我们要知道如何在考虑服务器的情况下构建应用。

即使客户端应用完全依赖云服务为后端，还是避免不了要从云服务中获取数据，以及把数据发给云服务。这些业务要在特定的地方处理，而且有些库可以帮助我们处理 HTTP 请求导致的延迟。

此外，我们可以采用同构方式渲染 React，以便支持浏览器以外的平台。这意味着，我们可以在服务器端渲染 UI，然后再发给浏览器。借助服务器端渲染，可以提升性能、增进可移植性、提高安全性。

本章首先说明同构和普适之间的区别，以及这两个概念与 React 的联系。然后，介绍如何全局使用 JavaScript 构建同构应用。最后，改进颜色管理应用，添加服务器，先在服务器端渲染 UI。

12.1 同构与普适

同构（isomorphic）和普适（universal）这两个术语通常都指在客户端和服务器端均可以运行的应用。虽然二者往往可以互换，但还是有点儿区别的。同构应用指可在多个平台上渲染的应用，而普适代码指同一套代码可在多个环境中运行。[注1]

有了 Node.js，我们可以在其他应用，例如服务器端应用、CLI 应用，甚至是原生应用中重用为浏览器编写的代码。下面来看一段普适 JavaScript 代码：

```
const userDetails = response => {
  const login = response.login;
  console.log(login);
};
```

这里的 `userDetails` 函数就是普适的，既可以在浏览器中调用，也能在服务器端调用，改都不用改。这意味着，使用 Node.js 构建的服务器端应用，说不定就可以在两个环境之间重用代码。普适 JavaScript 代码既可以在服务器端运行，也能在浏览器中运行，完全不会出错（见图 12-1）。

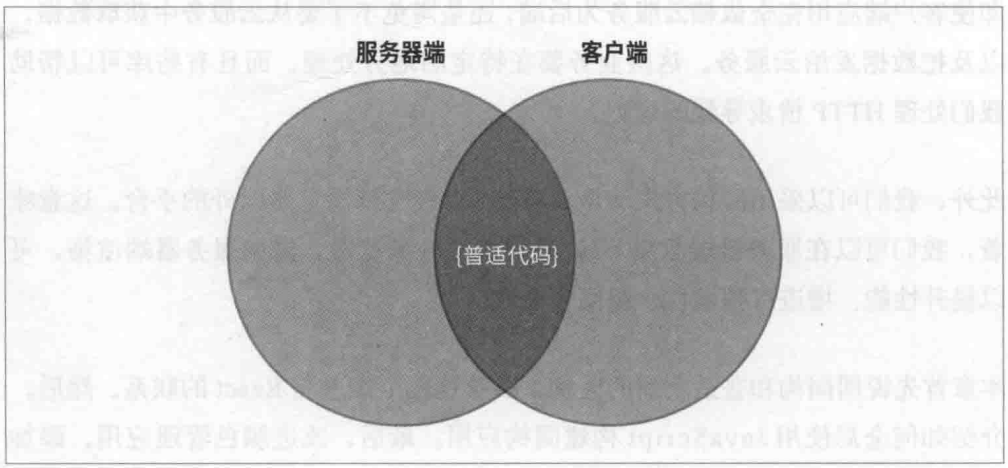

图 12-1：客户端和服务器端范畴

注 1：　Gert Hengeveld，“Isomorphism vs Universal JavaScript” (*https://oreil.ly/i70W2*)，Medium.

客户端和服务器端范畴

服务器端和客户端是完全不同的范畴，不是所有 JavaScript 代码不经一点改动就能在两端运行。下面是在浏览器中创建 Ajax 请求的方法：

```
fetch("https://api.github.com/users/moonhighway")
  .then(res => res.json())
  .then(console.log);
```

这里，我们向 GitHub API 发起一个 fetch 请求，把响应转换成 JSON，然后调用一个函数解析得到的 JSON。

然而，如果使用 Node.js 运行这段代码，会报错：

```
fetch("https://api.github.com/users/moonhighway")
^

ReferenceError: fetch is not defined
at Object.<anonymous> (/Users/eveporcello/Desktop/index.js:7:1)
at Module.\_compile (internal/modules/cjs/loader.js:1063:30)
at Object.Module.\_extensions..js (internal/modules/cjs/loader.js:1103:10)
at Module.load (internal/modules/cjs/loader.js:914:32)
at Function.Module.\_load (internal/modules/cjs/loader.js:822:14)
at Function.Module.runMain (internal/modules/cjs/loader.js:1143:12)
at internal/main/run_main_module.js:16:11
```

出现这个错误的原因是，Node.js 与浏览器不同，没有内置 `fetch` 函数。在 Node.js 中，我们可以使用 npm 中的 `isomorphic-fetch`，也可以使用内置的 `https` 模块。既然我们已经使用了 `fetch` 句法，那就用 `isomorphic-fetch`：

```
npm install isomorphic-fetch
```

然后，只需导入 `isomorphic-fetch`，其他代码无须改动：

```
const fetch = require("isomorphic-fetch");

const userDetails = response => {
  const login = response.login;
```

```
  console.log(login);
};

fetch("https://api.github.com/users/moonhighway")
  .then(res => res.json())
  .then(userDetails);
```

使用 Node.js 从 API 中加载数据要用到核心模块，写出的代码是不同的。在上述示例中，userDetails 函数是普适的，因此在两个环境中均可正常运行。

现在，这个 JavaScript 文件就是同构的，文件中的代码是普适的。不是说所有代码都是普适的，但这个文件在两个环境中都可以使用，既可以在 Node.js 中运行，也可以在浏览器中使用 <script> 标签引入。

下面来看 Star 组件。这个组件是普适的吗？

```
function Star({
  selected = false,
  onClick = f => f
}) {
  return (
    <div
      className={
        selected ? "star selected" : "star"
      }
      onClick={onClick}
    ></div>
  );
}
```

肯定是。别忘了，JSX 会编译成 JavaScript。Star 组件就是一个函数而已。

```
function Star({
  selected = false,
  onClick = f => f
}) {
  return React.createElement("div", {
    className: selected
```

```
        ? "star selected"
        : "star",
      onClick: onClick
    });
  }
```

这个组件可以直接在浏览器中渲染，也可以捕获输出的 HTML 字符串，在其他环境中渲染。ReactDOM 提供的 renderToString 方法可以把 UI 渲染为 HTML 字符串：

```
// 在浏览器中直接渲染 HTML
ReactDOM.render(<Star />);

// 把 HTML 渲染为字符串
let html = ReactDOM.renderToString(<Star />);
```

我们可以构建同构应用，在不同的平台中渲染组件，还可以通过一定的架构方式，跨多个平台全局重用 JavaScript 代码。另外，使用其他编程语言，例如 Go 或 Python 也可以构建同构应用，不限于只能使用 Node.js。

12.2 服务器端渲染 React

使用 ReactDOM.renderToString 方法可以在服务器端渲染 UI。服务器功能强大，能访问浏览器无法访问的各种资源。服务器安全性高，可以访问机密数据。我们可以充分利用这些额外的优势，在服务器端渲染初始内容。

我们将在服务器端渲染第 5 章构建的菜谱应用。你可以使用 Create React App 生成一个应用，把下述代码写入 *index.js* 文件：

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import { Menu } from "./Menu";

const data = [
  {
```

```
  name: "Baked Salmon",
  ingredients: [
    {
      name: "Salmon",
      amount: 1,
      measurement: "lb"
    },
    {
      name: "Pine Nuts",
      amount: 1,
      measurement: "cup"
    },
    {
      name: "Butter Lettuce",
      amount: 2,
      measurement: "cups"
    },
    {
      name: "Yellow Squash",
      amount: 1,
      measurement: "med"
    },
    {
      name: "Olive Oil",
      amount: 0.5,
      measurement: "cup"
    },
    {
      name: "Garlic",
      amount: 3,
      measurement: "cloves"
    }
  ],
  steps: [
    "Preheat the oven to 350 degrees.",
    "Spread the olive oil around a glass baking dish.",
    "Add the yellow squash and place in the oven for 30 mins.",
    "Add the salmon, garlic, and pine nuts to the dish.",
    "Bake for 15 minutes.",
    "Remove from oven. Add the lettuce and serve."
  ]
```

```
  },
  {
    name: "Fish Tacos",
    ingredients: [
      {
        name: "Whitefish",
        amount: 1,
        measurement: "1 lb"
      },
      {
        name: "Cheese",
        amount: 1,
        measurement: "cup"
      },
      {
        name: "Iceberg Lettuce",
        amount: 2,
        measurement: "cups"
      },
      {
        name: "Tomatoes",
        amount: 2,
        measurement: "large"
      },
      {
        name: "Tortillas",
        amount: 3,
        measurement: "med"
      }
    ],
    steps: [
      "Cook the fish on the grill until hot.",
      "Place the fish on the 3 tortillas.",
      "Top them with lettuce, tomatoes, and cheese."
    ]
  }
];

ReactDOM.render(
  <Menu
    recipes={data}
```

```
    title="Delicious Recipes"
  />,
  document.getElementById("root")
);
```

相关组件放在单独的 *Menu.js* 文件中：

```
function Recipe({ name, ingredients, steps }) {
  return (
    <section
      id={name.toLowerCase().replace(/ /g, "-")}
    >
      <h1>{name}</h1>
      <ul className="ingredients">
        {ingredients.map((ingredient, i) => (
          <li key={i}>{ingredient.name}</li>
        ))}
      </ul>
      <section className="instructions">
        <h2>Cooking Instructions</h2>
        {steps.map((step, i) => (
          <p key={i}>{step}</p>
        ))}
      </section>
    </section>
  );
}

export function Menu({ title, recipes }) {
  return (
    <article>
      <header>
        <h1>{title}</h1>
      </header>
      <div className="recipes">
        {recipes.map((recipe, i) => (
          <Recipe key={i} {...recipe} />
        ))}
      </div>
    </article>
```

```
    );
  }
```

截至目前，本书一直在客户端渲染组件。构建应用时，客户端渲染通常是首选方案。我们伺服 Create React App 生成的 *build* 文件夹，在浏览器中运行 HTML，通过 *script.js* 文件加载所需的 JavaScript 代码。

但是这个方案有点儿耗时。用户可能要等一会儿才能看到内容加载出来，具体等待时间取决于网络速度。Create React App 搭配 Express 服务器可以实现一种混合客户端和服务器端渲染的体验。

这个应用渲染 `Menu` 组件，通过它渲染各个菜谱。我们首先要做的改动是，把 `ReactDOM.render` 换成 `ReactDOM.hydrate`。

这两个函数的作用相同，只不过 `hydrate` 把内容添加到一个由 `ReactDOMServer` 渲染的容器中。操作顺序是这样的：

1. 渲染应用的静态版本，让用户知道应用在做事，“加载了”页面。
2. 请求动态 JavaScript 代码。
3. 把静态内容替换为动态内容。
4. 用户点击，交互正常。

这其实就是在服务器端渲染之后把内容再输送给应用。再输送是指先加载静态 HTML 内容，然后加载 JavaScript。这样做能让用户在感官上体验到性能的顺畅。用户发现页面中有内容就会更愿意停留。

接下来要设置项目的服务器。我们将使用轻量级 Node 服务器 Express。先安装：

```
npm install express
```

然后，创建一个服务器文件夹，命名为 *server*，再新建一个文件 *index.js*，保存在该文件夹中。我们在这个文件中构建一个服务器，让它伺服 *build* 文件夹，另外预先加载静态 HTML 内容。

```
import express from "express";
const app = express();

app.use(express.static("./build"));
```

这段代码导入服务器，静态伺服 *build* 文件夹。接着，使用 ReactDOM 提供的 renderToString 函数，以静态 HTML 字符串的形式渲染应用：

```
import React from "react";
import ReactDOMServer from "react-dom/server";
import { Menu } from "../src/Menu.js";

const PORT = process.env.PORT || 4000;

app.get("/*", (req, res) => {
  const app = ReactDOMServer.renderToString(
    <Menu />
  );
});

app.listen(PORT, () =>
  console.log(
    `Server is listening on port ${PORT}`
  )
);
```

我们要把 Menu 组件传给这个函数，因为这是我们想静态渲染的组件。然后，我们要从构建好的客户端应用中读取静态的 *index.html* 文件，把应用的内容写入 div 元素，并把这部分内容作为请求的响应。

```
app.get("/*", (req, res) => {
  const app = ReactDOMServer.renderToString(
    <Menu />
  );

  const indexFile = path.resolve(
    "./build/index.html"
  );
```

```
    fs.readFile(indexFile, "utf8", (err, data) => {
      return res.send(
        data.replace(
          '<div id="root"></div>',
          `<div id="root">${app}</div>`
        )
      );
    });
  });
```

完成以上操作后，我们要配置 webpack 和 Babel。虽然 Create React App 自身就能编译和构建，但是我们要为服务器端设置和实施一些不同的规则。

首先安装几个依赖（好吧，是大量依赖）：

```
npm install @babel/core @babel/preset-env babel-loader nodemon npm-run-all
webpack webpack-cli webpack-node-externals
```

安装好 Babel 之后，创建 `.babelrc` 文件，写入下述预设规则：

```
{
  "presets": ["@babel/preset-env", "react-app"]
}
```

别忘了添加 `react-app`，毕竟这个项目使用的是 Create React App（已经安装）。

接下来，为服务器增加一个 webpack 配置文件，名为 *webpack.server.js*：

```
const path = require("path");
const nodeExternals = require("webpack-node-externals");

module.exports = {
  entry: "./server/index.js",
  target: "node",
  externals: [nodeExternals()],
  output: {
    path: path.resolve("build-server"),
    filename: "index.js"
  },
```

```
  module: {
    rules: [
      {
        test: /\.js$/,
        use: "babel-loader"
      }
    ]
  }
};
```

babel-loader 的作用是转换 JavaScript 文件，而 nodeExternals 则负责扫描 *node_modules* 文件夹，找到所有 node_modules 名称，构建一个外部函数，告诉 webpack 不要打包这些模块及其子模块。

另外，你可能会遇到一个有关 webpack 的错误，这是由 Create React App 安装的版本与我们自己安装的版本发生冲突导致的。这个冲突的解决方法是，在项目根目录中创建 *.env* 文件，写入下述内容：

```
SKIP_PREFLIGHT_CHECK=true
```

最后，可以添加几个 npm 脚本，运行开发相关的命令：

```
{
  "scripts": {
    //...
    "dev:build-server": "NODE_ENV=development webpack --config webpack.server.js
    --mode=development -w",
    "dev:start": "nodemon ./server-build/index.js",
    "dev": "npm-run-all --parallel build dev:*"
  }
}
```

1. dev:build-server：设置环境变量 development，使用新服务器配置运行 webpack。
2. dev:start：使用 nodemon 运行服务器文件，监听变动。
3. dev：并行运行前两个进程。

现在，执行`npm run dev`命令，两个进程都会运行。你会看到应用运行在`localhost:4000`地址上。这个应用将按顺序加载内容，先预渲染HTML，再渲染JavaScript构建包。

使用这样的技术可以缩减加载时间，提升感知性能。用户期望页面加载时间在2秒以内，如能适当提升性能，说不定可以留住用户，不流往竞争对手网站。

12.3 使用Next.js做服务器端渲染

在服务器端渲染生态环境中，还有一个使用广泛的强大工具，即Next.js。Next.js是Zeit发布的一项开源技术，旨在帮助工程人员更轻松地编写服务器端渲染的应用。这个工具提供的功能有直观的路由、静态优化、自动分拆代码等。本节详细说明如何使用Next.js在服务器端渲染应用。

首先，执行下述命令，创建一个全新的项目：

```
mkdir project-next
cd project-next
npm init -y
npm install --save react react-dom next
mkdir pages
```

然后，创建几个npm脚本，方便运行常用的命令：

```
{
  //...
  "scripts": {
    "dev": "next",
    "build": "next build",
    "start": "next start"
  }
}
```

在*pages*文件夹中新建*index.js*文件。我们在这个文件中编写组件，但是不自己导入React和ReactDOM，只编写一个组件：

```
export default function Index() {
  return (
    <div>
      <p>Hello everyone!</p>
    </div>
  );
}
```

编写好这个组件之后，我们可以执行 `npm run dev` 命令，在 `localhost:3000` 地址上运行这个页面。你会看到预期的内容。

此外，你会发现界面右下方有个小闪电球图标。把鼠标悬停在这个图标上，显示一个按钮，内容是“Prerendered Page”。单击这个按钮将打开“Static Optimization Indicator”文档。这表明当前页面符合自动静态优化的条件，也就是说可以预渲染，没有获取数据的操作阻塞页面渲染。自动静态优化后的页面加载速度更快，因为不需要服务器端参与。我们可以把这样的页面放到 CDN 中，极大地提升用户体验。这项性能增强是自动的，不用人为参与。

那么，如果是需要获取数据的页面呢？如果页面不能预渲染该怎么办？在进一步探讨之前，我们要充实一下应用，构建一个组件，从远程 API 中获取数据。新建文件 *Pets.js*，写入下述内容：

```
export default function Pets() {
  return <h1>Pets!</h1>;
}
```

现在，我们只渲染一个 `h1` 元素。访问 `localhost:3000/pets`，页面可以正常加载。这是个不错的开始，接下来添加一些链接，再编写一个布局组件，显示各个页面的内容。我们可以创建一个页头，显示链接，供两个页面使用：

```
import Link from "next/link";

export default function Header() {
  return (
    <div>
      <Link href="/">
```

```
          <a>Home</a>
        </Link>
        <Link href="/pets">
          <a>Pets</a>
        </Link>
      </div>
    );
  }
```

链接放在 Link 组件中。这与使用 React Router 创建的链接相似。我们还可以为各个 <a> 标签添加样式：

```
const linkStyle = {
  marginRight: 15,
  color: "salmon"
};

export default function Header() {
  return (
    <div>
      <Link href="/">
        <a style={linkStyle}>Home</a>
      </Link>
      <Link href="/pets">
        <a style={linkStyle}>Pets</a>
      </Link>
    </div>
  );
}
```

接下来，新建文件 *Layout.js*，在其中使用 Header 组件。这个文件根据路由动态显示组件：

```
import Header from "./Header";

export function Layout(props) {
  return (
    <div>
      <Header />
```

```
      {props.children}
    </div>
  );
}
```

Layout 组件通过传入的属性参数在 Header 组件下方显示额外的内容。这样，我们就可以在各个页面中编写内容，传给 Layout 组件渲染。比如说，*index.js* 文件现在的内容如下：

```
import Layout from "./Layout";

export default function Index() {
  return (
    <Layout>
      <div>
        <h1>Hello everyone!</h1>
      </div>
    </Layout>
  );
}
```

再为 *Pets.js* 文件添加一些内容：

```
import Layout from "./Layout";

export default function Pets() {
  return (
    <Layout>
      <div>
        <h1>Hey pets!</h1>
      </div>
    </Layout>
  );
}
```

现在访问首页，你会看到页头，单击“Pets”链接，又显示“Pets”页面。

单击右下角的闪电球按钮，你会发现这两个页面仍是预渲染的。这也不奇怪，

因为我们渲染的全是静态内容。下面在“Pets”页面中加载一些数据，看看情况如何。

首先，安装本章前文用过的 isomorphic-unfetch：

```
npm install isomorphic-unfetch
```

我们将使用这个库向 Pet Library API 发起 fetch 请求。先在 *Pages.js* 文件中导入：

```
import fetch from "isomorphic-unfetch";
```

然后，定义一个名为 getInitialProps 的函数，负责获取和加载数据：

```
Pets.getInitialProps = async function() {
  const res = await fetch(
    `http://pet-library.moonhighway.com/api/pets`
  );
  const data = await res.json();
  return {
    pets: data
  };
};
```

我们把返回的数据赋给 pets 键，随后便可以在组件中映射这些数据。

修改 Pets 组件，映射 pets 属性：

```
export default function Pets(props) {
  return (
    <Layout>
      <div>
        <h1>Pets!</h1>
        <ul>
          {props.pets.map(pet => (
            <li key={pet.id}>{pet.name}</li>
          ))}
        </ul>
      </div>
```

```
      </Layout>
    );
  }
```

由于组件中存在 `getInitialProps` 函数，每次请求 Next.js 都会渲染页面。这意味着，该页面将在服务器端渲染，而不再静态预渲染，从而保证每次从 API 中请求的数据都是最新的。

对应用的功能满意之后，我们可以执行 `npm run build` 命令。Next.js 十分关注性能，给出了每个文件的千字节数。这样方便抽查不寻常的大文件。

每个文件旁边都有一个图标，指明是运行时在服务器端渲染的（λ）、是自动渲染为 HTML 的（○），还是自动生成为 HTML 和 JSON 的（●）。

构建好应用之后就可以部署了。Next.js 是云托管服务商 Zeit 推出的一个开源产品，因此部署到 Zeit 中是最简单的。不过，你也可以使用其他众多托管服务商部署自己的应用。

下面总结一下构建应用方面一些重要的术语。

CSR（*client-side rendering*，客户端渲染）

在浏览器中渲染应用，一般使用 DOM。如未调整，Create React App 就是这么做的。

SSR（*server-side rendering*，服务器端渲染）

在服务器端渲染客户端应用或普适应用。

再输送

在客户端加载 JavaScript 视图，重用服务器端渲染的 HTML DOM 树和数据。

预渲染

在构建时运行客户端应用，把初始状态生成为静态 HTML。

12.4 Gatsby

基于 React 的另一个受欢迎的网站生成工具是 Gatsby。Gatsby 为创建内容驱动型网站提供了一种简单直观的方式。Gatsby 的默认设置足够智能，性能、可访问性、图像处理等都不用我们自己操心。正在阅读这本书的你，说不定以后也会使用 Gatsby 创建项目。

Gatsby 可以创建的项目类型很多，不过通常用于构建内容驱动型网站。也就是说，对博客或静态内容网站而言，Gatsby 是个不错的选择，尤其是对 React 熟悉的人。此外，Gatsby 也可以处理动态内容，例如从 API 中加载数据、与框架集成等。

本节构建一个简单的 Gatsby 网站，说明它的用法。其实，我们是把前文那个 Next.js 应用构建成 Gatsby 应用。

```
npm install -g gatsby-cli
gatsby new pets
```

如果你全局安装了 yarn，CLI 会询问你想使用 yarn 还是 npm。二者都可以。然后，切换目录，进入 *pets* 文件夹：

```
cd pets
```

现在，执行 `gatsby develop` 命令，启动项目。访问 `localhost:8000`，你会看到 Gatsby 新生成的这个网站。下面来看一下文件结构。

打开项目中的 *src* 文件夹，里面有三个子文件夹：components、images 和 pages。

pages 文件夹中有个 *404.js* 错误页面、一个 *index.js* 文件（访问 `localhost:8000` 时渲染的页面），还有一个 *page-2.js* 文件，渲染第二个页面的内容。

components 文件夹是 Gatsby 施展魔法的地方。还记得我们使用 Next.js 编写的 `Header` 和 `Layout` 组件吗？这两个组件已经创建，作为模板保存在 *components* 文件夹中。

下面两个文件着重说一下。

layout.js

这个文件中是 Layout 组件。使用 useStaticQuery 钩子，通过 GraphQL 查询网站所需的数据。

seo.js

这个组件用于访问页面的元数据，做搜索引擎优化。

在 *pages* 文件夹中添加页面，就是为网站添加页面。下面在 *pages* 文件夹中新建 *page-3.js* 文件，在该文件中写入下述代码，创建一个简单的页面：

```
import React from "react";
import { Link } from "gatsby";

import Layout from "../components/layout";
import SEO from "../components/seo";

const ThirdPage = () => (
  <Layout>
    <SEO title="Page three" />
    <h1>Hi from the third page</h1>
    <Link to="/">Go back to the homepage</Link>
  </Layout>
);

export default ThirdPage;
```

内容是套在 Layout 组件中的，因此内容作为 children 显示。现在，不仅 Layout 组件能显示动态内容，而且页面也将自动生成。

Gatsby 的功能还有很多，本节只是蜻蜓点水。下面提供一些信息，帮你了解其他功能：

静态内容

可以把网站构建成静态文件，无须服务器即可部署。

CDN 支持

可以在遍布世界各地的 CDN 中缓存网站，提升性能和可用性。

响应式和渐进式图像

Gatsby 先加载模糊的占位图像，而后慢慢显现完整版本。由 Medium 带动起来的这个策略，能让用户在看到完整的资源之前知道图像正在渲染。

预获取链接的页面

单击下一个页面的链接之前在后台加载内容。

本节提到的功能，以及没有涉及的功能，都是为了确保用户体验的顺畅。Gatsby 为你做了很多决定，有些是好的，有些不那么好，不过目的都是为了让你把注意力集中在内容上。

12.5 React 未来展望

虽然 Angular、Ember 和 Vue 在 JavaScript 生态系统中占有的市场份额一直很大，但 React 是目前使用最广泛、影响最大的 JavaScript 应用构建库，这是毋庸置疑的事实。React 库的流行，也带动了 JavaScript 社区中更多的工具选择 React 作为开发工具，Next.js 和 Gatsby 是其中的佼佼者。

那么，读完本书后，接下来呢？笔者建议你使用从书中学到的技能，自己构建项目。如果你想构建移动应用，可以使用 React Native；如果你想使用声明式句法获取数据，可以使用 GraphQL；如果你想构建内容类网站，可以进一步探索 Next.js 和 Gatsby。

条条大路通罗马，不过你在本书中学到的 React 技能在构建应用上将给你提供极大的帮助。笔者希望本书能为选用 React 的你夯实基础，同时也作为你开发过程中的参考书。几乎可以肯定的是，React 及相关的库在发展过程中势必发生变化，但是这些工具的整体架构已经定型，你可以放心使用。使用函数式、声明式 JavaScript 句法构建 React 应用乐趣无穷，笔者希望能看到你的优秀作品。

O'REILLY®

React学习手册

如果你想学习如何构建高效运行的React应用，这本书正适合你。如果你是Web开发者或软件工程师，知道JavaScript、CSS和HTML在浏览器中的运行方式，本书特别适合你。这一版全面更新，道出了编写现代化React代码的最佳实践和最佳模式。阅读本书无须事先了解React，也不用知道函数式JavaScript。

Alex Banks和Eve Porcello编制的学习路线将教会你如何创建UI，在大型数据驱动型网站中灵巧地显示内容变动，但不重新加载页面。此外，你还将学习函数式编程和ECMAScript最新的特性。通过这本实用的指南学会如何构建React组件之后，你便会知晓React能为团队带来多大的帮助。

- **理解JavaScript函数式编程的关键概念。**
- **深入了解React在浏览器中的运行机制。**
- **使用React组件创建应用的表现层。**
- **管理数据，减少花在调试应用上的时间。**
- **利用React钩子管理状态及获取数据。**
- **在单页应用中使用路由方案。**
- **学习站在服务器的角度上规划React应用。**

> **“前端社区正缺少这样一本React书籍。Alex和Eve不仅深入探讨了React，还介绍了JavaScript基本概念、测试、状态管理等。而且两位作者的语言浅显易懂，不失幽默。我力荐这本书。”**
>
> **—Emma Bostian**
> Spotify 软件工程师

Alex Banks和Eve Porcello是软件工程师、作者和培训讲师。他们合办的公司Moon Highway向全世界的工程开发人员教授JavaScript、Node.js、React和GraphQL课程。他们还为O'Reilly、LinkedIn Learning 和egghead.io录制视频及编写培训材料。

PROGRAMMING LANGUAGES / JAVASCRIPT

ISBN 978-7-5198-5654-0

O'Reilly Media, Inc.授权中国电力出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of the People's Republic of China (excluding Hong Kong, Macao and Taiwan)

定价：98.00元

[General Information]
书名=REACT学习手册 第2版
页数=361
SS号=14982779