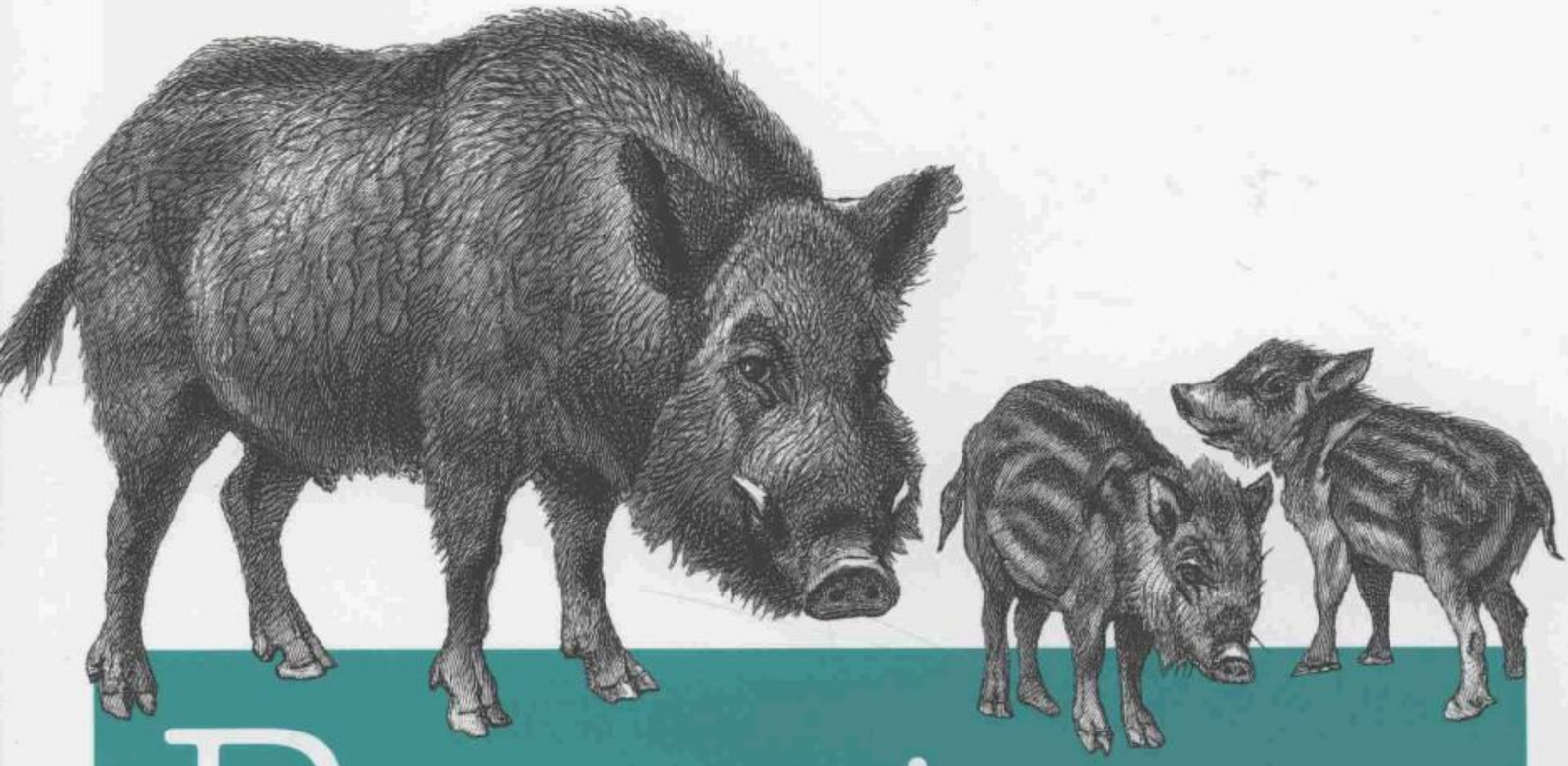
O'REILLY®



React 学习手册

Learning React

Alex Banks, Eve Porcello 著
邓世超 译

中国电力出版社

图书在版编目（CIP）数据

React学习手册 /（美） 亚历克斯·班克斯（Alex Banks），（美）伊夫·波尔切洛（Eve Porcello）著；邓世超译. — 北京：中国电力出版社，2017.12

书名原文：Learning React

ISBN 978-7-5198-1423-6

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

中国版本图书馆CIP数据核字(2017)第291672号

北京市版权局著作权合同登记 图字：01-2017-7619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：李 楠
装帧设计：Karen Montgomery，张 健
责任印制：蔺义舟

印　刷：三河市百盛印装有限公司
版　次：2017年12月第一版
印　次：2017年12月北京第一次印刷
开　本：787毫米 × 1092毫米　16开本
印　张：20.5
字　数：393千字
印　数：0001—3000册
定　价：78.00元

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法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、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’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

前言

本书是专门为希望学习React脚本库，同时也希望学习当前JavaScript语言最新特性的开发人员而编写的。成为一名JavaScript开发人员时是激动人心的。这个生态系统中的新工具、语法和最佳实践如雨后春笋般层出不穷，从而确保用户可以解决开发过程中的大部分问题。本书的目标是将这些技术有机地组织起来，以便用户可以通过正确的方式使用React。我们还将介绍Redux、React Router，以及构建工具，因此我们将承诺不会只介绍一些基本知识后，就让你投入实际开发工作。

本书并不要求你事先了解任何与React有关的知识。我们会从头开始介绍React的基础知识。同样，我们也不会假设你已经了解ES6或者其他最新的JavaScript语法。这些内容将会在第2章详细介绍，并作为后续章节的基础。

如果你对HTML、CSS和JavaScript已经非常熟悉，那么实际上就已经为阅读本书做好了准备。在深入了解JavaScript脚本库之前，这三大技术几乎是最关键和最适合事先了解的。

在阅读本书过程中，可以不时地查看GitHub版本库（*http://github.com/moonhighway/learning-react*）。书中的所有示例代码都存放在上述版本库中，你可以方便地动手实践这些示例。

排版约定

本书使用如下排版约定：

斜体（Italic）

表示新的术语、URL、email地址、文件名、文件扩展名。

等宽字体（`Constant width`）

程序片段，比如引用程序的一些变量名、函数名、数据库、数据类型、环境变量、语句和关键字等。

等宽粗体（**`Constant width bold`**）

命令行程序或者其他需要用户输入的内容。

等宽斜体（*`Constant width italic`*）

表示应该由用户输入的值或根据上下文确定的值。

这个图标表示提示或建议。

这个图标表示一般的注释说明。

这个图标表示警告或提醒。

使用代码示例

补充资料（代码示例、练习等）可以到以下地址下载：*https://github.com/moonhighway/learning-react*。

本书中的代码可以帮助你完成工作。一般而言，本书中的源代码，不需要得到我们的许可就可以应用到你自己的程序或文档中，除非你希望重新发布这些代码的副本。举例来说，一个程序中用到若干本书中的代码片段不需要授权许可。销售或发布包含O'Reilly图书中的示例代码的CD-Rom则需要授权许可。回答一个问题引用本书中的

内容或代码不需要授权许可。在你的产品文档中加入大量的本书示例代码需要授权许可。

我们建议但非强制要求标明署名。署名通常包括标题，作者，发行商和ISBN。例如：Learning React by Alex Banks and Eve Porcello (O'Reilly). Copyright 2017 Alex Banks, Eve Porcello, 978-1-491-95462-1。

如果你觉得使用代码示例超出了上述许可范围，请通过permissions@oreilly.com与我们取得联系。

O'Reilly Safari

Safari（以前的Safari Books Online）是一个面向企业、政府、教育机构和个人，并且基于会员的培训和参考平台。

会员可以访问数千种图书、培训视频、学习路径、互动教程，以及超过250家出版社提供的播放列表。其中包括O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press, Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett，以及大量的技术课程等。

欲知详情，可以前往*http://oreilly.com/safari*。

联系我们

请通过以下地址与本书发行商取得联系:

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

本书有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。这个网页的地址是：*http://bit.ly/learning-react-2e*。

如果你对本书有一些评论或技术上的建议，请发送电子邮件到*bookquestions@oreilly.com*。

有关我们的书籍、课程、会议和新闻的更多信息，请访问我们的网站*http://www.oreilly.com*。

致谢

如果没有一些老朋友的帮助，我们的React之旅将无法启程。当我们在Yahoo做内部培训时，是采用YUI库为全栈JavaScript程序课程编写的培训材料。到了2014年8月，YUI库的开发终止了。我们不得不修改所有的课程文件，但是应该采用什么库呢？现在前端应用该使用哪一种库呢？答案是：React。我们并没有马上喜欢上React，它花费了几小时来吸引我们的注意。看上去React会改变一切。我们能够早点接触它真的很幸运。

如果没有Ally MacDonald的支持，本书就不可能面世，他帮助我们迈出了一大步，并且非常耐心地帮助我们更新若干脚本库。我们还要感谢Melanie Yarbrough、Colleen Toporek和Rachel Head对本书细节的认真严谨。感谢Sarah Ronau在本书面世之前的仔细审读，感谢Bonnie Eisenman为提高本书阅读体验的良好建议。还要感谢Stoyan Stefanov，即使他正忙于在Facebook开发一些非常酷的东西，仍然抽出时间完成了本书的技术审阅。

如果没有Sharon Adams和Marilyn Messineo的帮助，那么本书可能就不会存在。他们共同为Alex购买了第一台计算机，一款Tandy TRS 80彩色电脑。没有Jim和Lorri Porcello，以及Mike和Sharon Adams的爱、支持和鼓励，本书也不能如期完成。

我们还要感谢加利福尼亚州塔霍市的Connexion咖啡，它为我们提供了编写本书过程中所需的咖啡，以及它的老板Robin给我们的金玉良言："一本关于编程的书？听起来就很无聊！"

目录

第1章

初识React

React是一款用于创建用户界面的流行库。它是Facebook为了辅助数据驱动的大型网站解决某些问题而研发的。2013年React刚发布时，该项目受到了外界的一些质疑，因为React的技术规范非常独特。

为了不让新用户望而却步，React核心团队编写了名为“Why React”的文章建议读者“给它（即React）5分钟”。他们鼓励人们摒弃先入为主的观念，先了解一下React再对它评判也不迟。

的确，React是一款小型脚本库，并不能满足用户即插即用构建自己应用程序的所有需求。那么请给它5分钟。

的确，在React中，开发工作就像在JavaScript中编写HTML代码。当然，这些标签需要经过预处理才能在浏览器中工作。用户可能还需要类似webpack这类构建工具处理这些事情。那么请给它5分钟。

如果你像我们一样读到了这篇文章，你也许会被层出不穷的JavaScript库允诺的功能弄得眼花缭乱，即一个能够解决所有和DOM相关问题的脚本库；该库可以大幅度减轻我们的工作量，并且还不会让我们陷入沮丧。

然后问题接踵而至：如何转换JSX格式？如何载入数据？在哪里存放CSS文件？什么是声明式编程？每种问题又会引出更多问题，比如用户如何将该库集成到日常的工作中。每种规范又引入新术语、新技术和更多问题。

障碍和绊脚石

花几分钟时间学习了React组件的相关知识之后，读者应该会了解到与以往不同的一种Web开发方式。不过在用户开始使用React编写程序之前还有不少障碍需要克服。

React 是一个脚本库

首先，React是一款小型脚本库，并且只能完成部分工作。它并没有像一般的JavaScript框架那样附带用户需要的所有工具。开发生态环境中所需的大部分工具需要用户自主选择。此外，新工具的不断涌现，旧的工具就会被取而代之。随着越来越多的库名称出现在团队讨论中，真有点让人应接不暇。

新的 ECMAScript语法

React相对于JavaScript历史来说，正处于一个非常重要而又混乱的时代。ECMA过去发布相关规范的频率非常低。有时需要长达10年才发布一版新规范。这意味着开发者不需要花太多时间学习新的语法。

自2015年以来，每年都会有新的语言特性和语法发布。最近一年多（ECMAScript 2016,ECMAScript 2017）甚至更新了数个版本系统（ECMAScript3, ECMAScript 5）。随着该语言的发展，React社区的早期使用者倾向于使用新的语法。这通常意味着开发文档会假定用户是了解最新的ECMAScript语法的。如果用户不熟悉最新的语法规范，那么浏览React代码时就会晕头转向。

JavaScript函数式编程的兴起

除了语言层面上出现的变化之外，JavaScript函数式编程也后劲十足。JavaScript不一定必须是一门函数式编程语言，不过可以在JavaScript代码中使用函数式编程范式。React鼓励用户使用函数式编程胜过面向对象编程。这种思维转变可以在程序的可测试性和执行性能方面获得更多好处。不过大量的React材料采用了上述编程范式时，学习曲线也变得异常陡峭。

JavaScript工具审美疲劳

JavaScript工具审美疲劳（*http://bit.ly/2pSiuE4*）已经是老生常谈了，不过归根结底这个问题是由构建过程产生的。在过去，用户只需将JavaScript文件添加到自己的页面即可。而现在JavaScript文件通常都必须经过编译，通常还伴随着一个自动化持续分

发过程。新兴的语法必须经过转换以便可以在大部分浏览器上运行。JSX格式的文件必须转换成JavaScript文件。SCSS文件有可能还需要进行预处理。这些组件也需要通过测试。用户也许会爱上React，但是现在必须先成为一名使用webpack的专家，处理代码分割、压缩和测试等工作。

React易于上手的原因

本书的目标是通过将相关材料有机地组织起来，避免读者在学习过程中产生混乱，同时也为后续的学习打下坚实的基础。我们将从学习新的语法开始，让读者了解JavaScript的新特性，特别是React中经常用到的。然后我们将会简要介绍一下JavaScript的函数式编程，使得用户能够马上应用这些技术，并理解上述范式对React产生的重要影响。

然后我们将通过向读者介绍第一个React组件来讲解相关的基础知识，同时还会阐述如何转换代码和必须这么做的原因。在了解相关基础知识之后，我们将会向读者介绍一个可以保存和调整颜色的新应用示例。该应用将会使用React构建，并使用若干高级React技术进行优化，采用Redux作为客户端数据容器，通过Jest测试和React路由完成该应用的构建。最后一章，我们将会介绍通用的同构代码，并通过在服务器端渲染来提高颜色管理器应用的性能。

我们希望以这种方式加速读者对React生态系统的深入了解，而非浅尝辄止。同时可以让读者掌握在实际工作中构建React应用必备的技能和工具。

React技术展望

React仍然很年轻。它的核心功能已经非常稳定，不过还是有改善空间的。React的未来版本将会加入纤程，以及旨在提高渲染速度的核心算法的重新实现。这些特性对于开发者会产生重大影响的观点还为时过早，不过这将大幅度提高应用App渲染和更新的效率是毋庸置疑的。

许多必要的改进都和目标设备有关。本书介绍的技术主要是通过React开发单页Web应用的，不过我们不应该假定Web浏览就是React应用App可以运行的唯一场景。2015年诞生的React Native项目允许用户将React应用转换成原生的iOS和Android应用App。React VR是一个构建交互式虚拟现实应用的框架，开发者可以使用React和JavaScript为用户提供360°的全景体验。React库的一条命令就可以帮助开发者构建跨屏幕和设备的快速开发环境。

我们希望能够为用户提供一个坚实的平台，这个平台能够适应不断变化的开发生态环境，并且构建的应用可运行的平台不仅仅局限于Web浏览器。

拥抱变化

React和相关工具的发展可谓日新月异，有时甚至是一些革命性的变化。事实上，这些工具的某些未来版本和本书目前介绍的内容也可能大相径庭。用户仍然可以放心地运行本书的代码示例。我们将会在*package.json*文件中提供精确的软件版本信息，因此读者在安装相关版本时大可放心。

除了本书之外，读者仍然可以通过React官方博客(*https://facebook.github.io/react/blog/*)了解该项目的最新动态。当React有新版本发布时，核心团队将会撰写博文介绍该版本的细节和新的功能特性。

还有不少流行的React技术峰会可供读者追踪它的最新动态。如果用户不能亲临会场，React技术峰会还会将会议内容录制成视频放在YouTube视频网站上供用户浏览。这些峰会主要包括：

React Conf（http://conf.reactjs.com/）
: Facebook赞助的湾区会议。

React Rally（http://www.reactrally.com/）
: 盐湖城的社区会议。

ReactiveConf（https://reactiveconf.com/）
: 斯洛伐克首都布拉迪斯拉发的社区会议。

React Amsterdam（http://react.amsterdam/）
: 阿姆斯特丹的社区会议。

文件资源

本节将会介绍如何使用本书相关的资源文件，以及如何安装若干有用的React工具。

文件版本库

与本书有关的GitHub版本库（*https://github.com/moonhighway/learning-react*）提供的代码文件都是以章节为单位进行组织的。该版本库混合了代码文件和JSBin示例程

序。如果你以前从没有用过JSBin，那么它其实是类似CodePen和JSFiddle的在线代码编辑器。JSBin的主要特性之一就是用户只需单击相关链接，马上就可以修复该文件。当用户创建或者开始编辑一个JSBin时，它会为用户的示例代码创建一个唯一的URL地址，如图1-1所示。

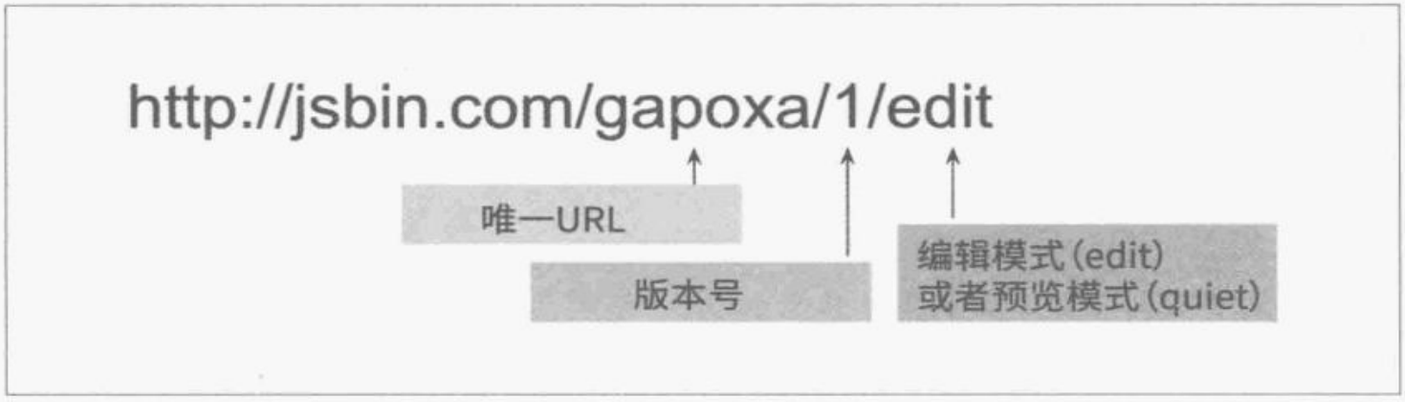


图 1-1：JSBin的URL地址

出现在*jsbin.com*后面的字母表示唯一的URL地址。接下来的斜杠后面是版本号。URL最后一部分有两个选项可选：一个是表示编辑模式的*edit*，另一个是表示预览模式的*quiet*。

React开发者工具

有不少开发者工具可以通过浏览器扩展或插件进行安装，你会发现其中很多都是非常有用的：

react-detector(http://bit.ly/2mwcoXR)

react-detector是一款Chrome浏览器扩展，它可以帮助用户检测某个网站是否采用了React技术。

show-me-the-react

这是一款工具，同时支持Firefox和Chrome浏览器，它可以帮助用户检测浏览器访问的网站是否使用了React技术。

React Developer Tools (http://bit.ly/2nvFKar)(见图1-2)

这是一款可以扩展浏览器开发者工具的插件。它可以在开发者工具中新建一个Tab页，方便用户查看React元素。

如果用户喜欢使用Chrome浏览器，那么还可以将它作为扩展（*http://bit.*

ly/1O5DTlX）进行安装，同时用户可以将它作为插件安装到Firefox(*https://mzl.la/2mMVgi5*)浏览器上。

图 1-2：使用开发者工具浏览相关元素

如果用户发现react-detector或者show-me-the-react处于激活状态，那么可以打开开发者工具了解该网站采用React技术的详细情况。

安装Node.js

Node.js是JavaScript独立于浏览器的运行环境。该运行环境可以用于构建全栈式的JavaScript应用。Node是一个开源项目，可以安装在Windows、macOS、Linux，以及其他平台。我们将会在第12章搭建Express服务器时用到Node。

用户不需要通过Node来使用React库。不过当使用React进行开发时，用户需要使用Node包管理器（即npm）来安装项目依赖。它是安装Node过程中一起自动安装的。

如果你不能确定自己的电脑上是否安装了Node.js，那么可以打开终端或者命令行窗口，然后输入如下命令：

```
$ node -v

Output: v7.3.0
```

理想情况下，用户应该会得到一个高于4的Node版本号。如果用户输入上述命令之后获得了“Command not found,”这样的信息提示，那么说明该用户机器上还没有安装Node.js。解决的办法也非常简单，用户直接用浏览器打开Node.js官方网站（*http://*

nodejs.org）。然后在该网站下载Node.js的自动化安装程序。安装完毕之后，在命令行再次输入node -v，就会看到相关的软件版本号了。

使用 Yarn进行依赖管理

Yarn是npm的一个备选替代性方案。它是由Facebook于2016年推出的，它最初的目的是为了方便和Exponent、Google和Tilde等公司协作。该项目帮助Facebook和其他公司更可靠地管理它们的依赖项。同时在使用它安装软件包时，你会发现它的速度更快。你可以在Yarn官方网站(*https://yarnpkg.com/en/compare*)上比较npm和Yarn之间的性能差异。

如果你对npm工作流非常熟悉，那么Yarn的使用也就驾轻就熟了。首先，使用npm对Yarn进行全局安装：

```
npm install -g yarn
```

然后用户就可以安装软件包了。当从package.json中安装相关依赖时，可以使用yarn替代npm。

在安装特定软件包时，为了替代npm install --save [package-name]命令，用户可以执行如下命令：

```
yarn add [package-name]
```

为了移除一个依赖，相关命令也是类似的：

```
yarn remove [package-name]
```

Facebook公司在实际开发工作中采用的就是Yarn，并且它还被包含到React、React Native和create-react-app中。如果读者发现某个项目中包含*yarn.lock*文件，那么该项目就是采用yarn作为包管理器的。和npm安装命令类似，用户可以输入**yarn install**或者**yarn**命令安装项目的所有依赖项。

现在你已经搭建好React开发环境了，我们已经做好在学习之路上披荆斩棘的准备。第2章将会介绍ECMA，快速地回顾一下React项目代码中常见的JavaScript新特性。

第2章

JavaScript新特性

JavaScript自1995年问世以来，它的经历非常丰富。首先，它使得给Web页面添加交互元素变得更简单。然后它让DHTML和AJAX技术更健壮。现在，通过Node.js,JavaScript成为了一门可以构建全栈应用的语言。负责对JavaScript语言规范进行维护的是欧洲计算机制造协会（ECMA）。

语言规范的修改是通过社区驱动的。它们来自社区成员提交的建议(*https://tc39.github.io/process-document/*)。任何人都可以向ECMA协会提交提案。ECMA协会的职责是管理和区分这些提案的优先级，然后决定哪些内容可以添加到规范中。提案的审核是由明确的若干阶段组成的。从代表最新提案的阶段0开始，直到代表提案审核完成的第4阶段结束。

该规范最近的重大更新是2015年6月审核通过的，[注1]并且该提案还有很多名称：ECMAScript 6、ES6、 ES2015和ES6Harmony。根据当前的计划，新规范将会以年为周期发布。2016年发布的规范相对来说变化不大，不过目前看2017版的规范会包含很多非常有用的新特性。我们将会在本书中用到这些新特性，并尽可能使用最新版本的JavaScript规范。

目前不少流行的浏览器都为这些新特性提供了支持。我们还会介绍如何将代码从支持新特性的JavaScript语法转换为目前主流浏览器都支持的ES5语法。kangax兼容性表格(*http://kangax.github.io/compat-table/esnext/*)能够很好地帮助用户了解主流浏览器对最新的JavaScript特性的兼容程度。

注1： Abel Avram, "ECMAScript 2015 Has Been Approved" (*http://bit.ly/2nvMJjJ*), 2015年6月17日 InfoQ技术峰会。

本章将会向读者介绍本书用到的所有JavaScript新特性。如果你还没有选择使用最新的语法，那么现在应该是一个良好的契机。如果你已经非常熟悉ES.Next语言规范中的特性，那么完全可以略过本章，直接阅读下一章的内容。

ES6中的变量声明

在ES6之前，声明变量的唯一方式就是通过var关键字。现在我们有了一些不同的选项对这一功能进行改进。

关键字const

常量是一个不能被修改的变量。和之前其他语言类似，JavaScript在ES6中引入了常量的概念。

在常量之前，我们使用的都是变量，所有变量都可以被重写：

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

我们无法重置一个常量的值，并且当我们尝试重写常量时，系统会生成一个控制台错误信息（见图2-1）：

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

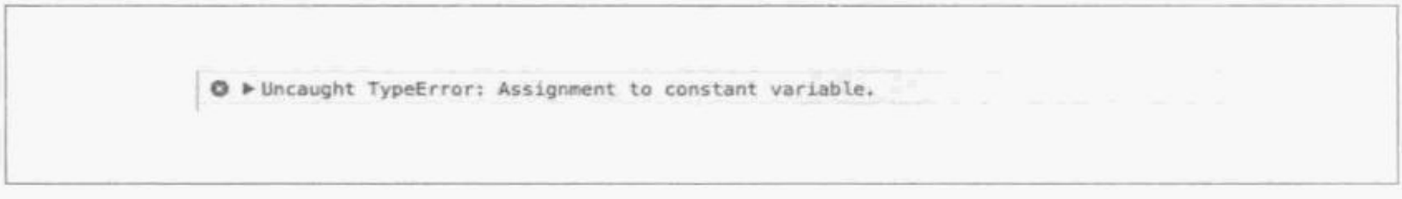


图 2-1：尝试重写一个常量后系统给出的错误提示

关键字let

JavaScript目前已经拥有文法层面的变量作用域了。在JavaScript中，我们使用一对花括号表示代码块，这些花括号限定了变量的作用域。换句话说，类似`if/else`语句这样的情况。如果你以前学习过其他编程语言，可能也会假定这些代码块限定了变量作用域，不过并非如此。

如果变量是在if/else语句块中创建的，那么变量的作用域并不会受到该代码块的限制。

```
var topic = "JavaScript"

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

console.log('global', topic)          // global React
```

if语句块中的变量topic重置了全局变量topic的值。

通过使用let关键字，我们可以将一个变量的作用域限定在任意代码块中。使用let关键字可以确保全局变量的值不受干扰：

```
var topic = "JavaScript"

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

console.log('global', topic)          // JavaScript
```

topic的值没有重置语句块之外的变量。

花括号不能限制变量作用域范围的另外一个地方是在循环体中：

```
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，我们在容器中创建了5个div元素。每个div都关联了一个单击事件来弹出一个警示对话框显示索引。在for循环中声明i时，创建一个全局变量i，然后进行循环直到它的值变成5。当用户单击任意一个div时，警示对话框显示i的值都会是5，因为当前全局变量i的值就是5（见图2-2）。

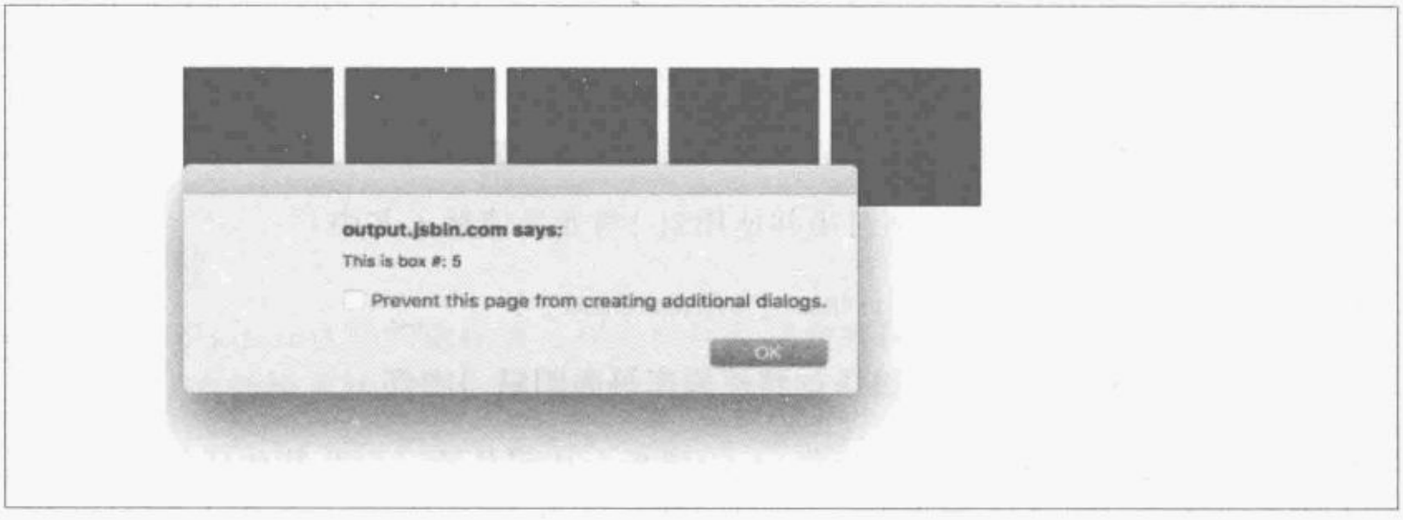


图 2-2：对于每个box容器来说，i的值都是5

使用关键字let替代var定义循环计数器i，从而限定它的作用域。现在单击任意一个box后，将会显示i的作用域被限定在循环过程内部的值（见图2-3）：

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

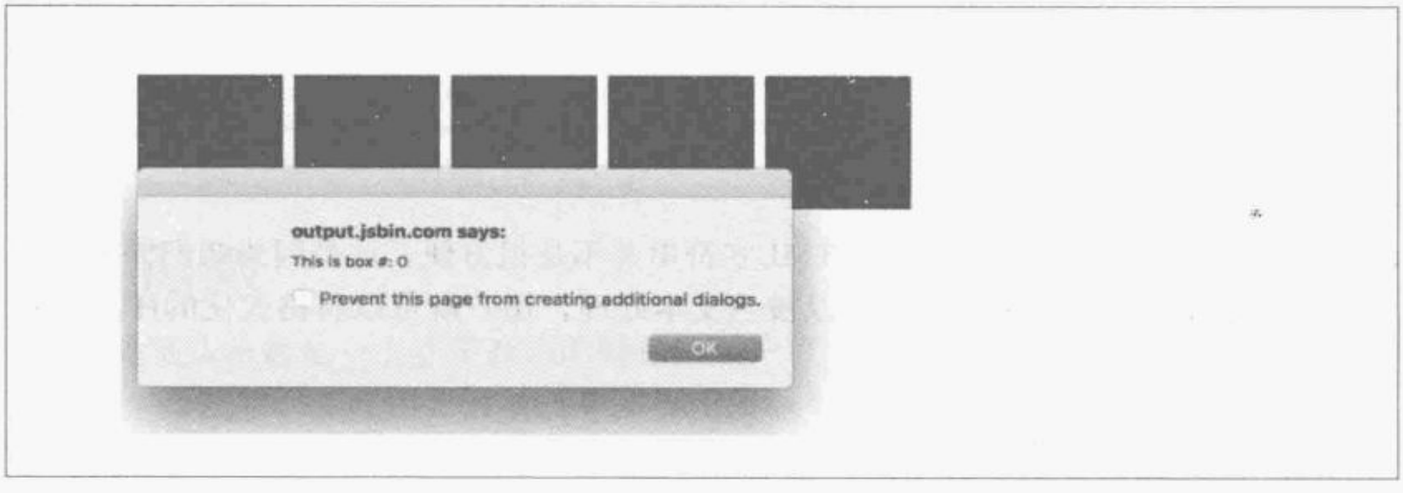


图2-3：关键字let限定了i的作用域

模板字符串

模板字符串为用户提供了一种连接字符串的替代性方案。它还允许用户在字符串中插入变量。

普通的字符串拼接是通过加号（+）和逗号（，）将变量和若干字符串拼接在一起后得到一个新字符串的：

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

通过模板，我们可以创建一个字符串并使用${ }将变量值插入其中：

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

JavaScript的任意返回值都可以被添加到模板字符串的${ }内部。

模板字符串还支持空格，这使得起草email模板、代码片段，以及其他任何包含空格的内容更容易了。现在用户可以在不破坏代码片段的情况下构造一个跨越多行的字符串。示例2-1演示了Tab制表符、换行符、空格和变量名在email模板中的具体用法。

示例2-1：模板字符串对空格的支持

```
`

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 at will call 30 minutes before the show.

Thanks,

${ticketAgent}

`
```

以前在JavaScript代码中直接使用HTML字符串并不是很方便，这是因为我们需要将它们挤在一行才能运行。现在空格可以被当文本处理，用户就可以将格式化的HTML插入其中，并且也更易于理解：

```
document.body.innerHTML = `
<section>
  <header>
      <h1>The HTML5 Blog</h1>
  </header>
  <article>
      <h2>${article.title}</h2>
      ${article.body}
  </article>
  <footer>
      <p>copyright ${new Date().getYear()} | The HTML5 Blog</p>
```

```
    </footer>
  </section>
`
```

注意，我们还可以将页面的标题和其中的内容文本作为变量插入模板字符串中。

默认参数

包括C++和Python这样的编程语言都允许开发者为函数参数声明默认值。ES6中也添加了对默认参数的支持，因此如果事件调用过程没有提供参数，那么系统将会使用默认参数值。

比如我们可以构造一个默认字符串：

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

如果在调用函数favoriteActivity时没有给它提供参数，那么它可以使用默认参数值正确地执行。默认参数并不局限于字符串，可以是任意类型：

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

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

箭头函数

ES6中的箭头函数是一个非常有用的特性，用户可以在不使用function关键字的情况下创建一个函数，并且用户通常还不需要使用return关键字。示例2-2是一个使用普通函数语法的例子。

示例2-2：一个普通函数

```
var lordify = function(firstname) {
  return `${firstname} of Canterbury`
}

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

使用箭头函数，我们可以大幅度简化函数语法声明，如示例2-3所示。

示例2-3：一个箭头函数

```
var lordify = firstname => `${firstname} of Canterbury`
```

本书中的分号

分号在JavaScript代码中是可选的，所以为什么要将非必需的分号添加到代码中呢？本书遵循极简风格，排除了不必要的语法。

通过使用箭头，我们现在只需一行代码就获得了一个函数实体。关键字function被移除了。我们还可以移除关键字return，因为箭头指向的内容将会自动返回。另外一个优点是如果函数只包含一个参数，我们还可以移除参数两边的括号。

包含一个以上的参数时，函数两边的圆括号是必不可少的。

```
// 旧方案
var lordify = function(firstName, land) {
  return `${firstName} of ${land}`
}

// 新方案
var lordify = (firstName, land) => `${firstName} of ${land}`

console.log( lordify("Dale", "Maryland") )      // Dale of Maryland
console.log( lordify("Daryle", "Culpeper") )    // Daryle of Culpeper
```

我们可以将它当作单行函数使用，因为只有一行语句需要返回。

出现多行语句声明时需要使用一对花括号将它们括起来：

```
// 旧方案
var lordify = function(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}`
}

// 新方案
var _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
```

这些if/else语句中虽然使用了花括号，不过它们仍然可以利用箭头函数简短语法的特性。

箭头函数并不会拘泥于此。比如在setTimeout回调函数中的关键字this可以表示tahoe对象之外的其他对象：

```
var tahoe = {
  resorts: ["Kirkwood","Squaw","Alpine","Heavenly","Northstar"],
  print: function(delay=1000) {

    setTimeout(function() {
      console.log(this.resorts.join(","))
    }, delay)

  }
}

tahoe.print() // Cannot read property 'join' of undefined
```

上述代码抛出异常信息是因为它尝试在this指代的对象上调用.join方法。在这种情况下，this指代是window对象。此外，我们可以使用箭头函数的语法限定this的作用域：

```
var tahoe = {
  resorts: ["Kirkwood","Squaw","Alpine","Heavenly","Northstar"],
  print: function(delay=1000) {

    setTimeout(() => {
      console.log(this.resorts.join(","))
    }, delay)

  }
}

tahoe.print() // Kirkwood, Squaw, Alpine, Heavenly, Northstar
```

这一次代码能够正常工作，并且我们可以调用.join方法对resorts中的元素用逗号分隔后打印输出。不过需要注意的是，用户心里一定要时刻保持对作用域的关注。箭头函数不会限定下列代码中关键字this的作用域：

```
var tahoe = {
  resorts: ["Kirkwood","Squaw","Alpine","Heavenly","Northstar"],
  print: (delay=1000) => {

    setTimeout(() => {
      console.log(this.resorts.join(","))
    }, delay)

  }
}

tahoe.print() // Cannot read property resorts of undefined
```

将print函数声明改为一个箭头函数的形式意味着其中关键字this指代的对象就是window。

为此，我们可以修改控制台的信息输出，来验证一下其中的关键字this指代的是否就是window：

```
var tahoe = {
  resorts: ["Kirkwood","Squaw","Alpine","Heavenly","Northstar"],
  print: (delay=1000)=> {

    setTimeout(() => {
      console.log(this === window)
    }, delay)

  }
}

tahoe.print()
```

上述代码的输出结果是true。为了解决这个问题，我们可以使用普通的函数声明方式：

```
var tahoe = {
  resorts: ["Kirkwood","Squaw","Alpine","Heavenly","Northstar"],
  print: function(delay=1000) {

    setTimeout(() => {
      console.log(this === window)
    }, delay)

  }
}
```

```
tahoe.print() // false
```

ES6转译

并不是所有浏览器都支持ES6,其中有些甚至根本不支持该规范。确保用户根据ES6规范编写的代码能够正常工作的唯一办法是，在浏览器中运行这些代码之前将它们转换成符合ES5规范的代码。这个过程被称为转译。Babel(*http://www.babeljs.io/*)是当前最流行的转译工具之一。

在过去，使用JavaScript最新特性的唯一办法是等待数周、数月，甚至数年，直到大部分主流的浏览器都支持它们。现在，转译机制使得马上使用JavaScript最新特性的想法变成了现实。转译的步骤和其他语言类似。转译不是编译：我们的代码并没有被编译成二进制形式。相反，它们被转译成能够被绝大多数浏览器识别的语法。当然，JavaScript目前还有自己的源代码，这意味着属于用户项目的某些源代码文件并不是直接在浏览器中运行的。

示例2-4展示了一些ES6代码。其中包括一个箭头函数，并且其中已经包含了某些参数名为x和y的默认参数。

示例2-4：Babel转译之前的ES6代码

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

对上述代码进行转译之后，结果如下列代码所示：

```
"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);
};
```

转译器在代码前面添加了“use strict”声明，这表示上述代码会以严格模式（strict mode）运行。变量x和y默认会使用参数数组，该技术也许读者已经耳熟能详了。转译后的JavaScript代码将会被绝大多数浏览器兼容。

用户还可以使用内联式Babel转译器在浏览器中直接对JavaScript代码进行转译。用户只需添加*browser.js*文件，并将任意脚本的类型指定为`type="text/babel"`即可（虽然当前Babel的版本是Babel 6，不过调用Babel 5版本的CDN服务仍然可以正常工作）。

```
<script
    src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.js">
</script>
<script src="script.js" type="text/babel">
</script>
```

浏览器中的转译

这种方法意味着浏览器会在运行时执行转译工作。这对于上线的产品来说并非良策，因为它会大大拖慢页面应用的响应速度。第5章将会向读者详细介绍如何在已经上线的产品中解决这个问题。现在，CDN服务将会允许客户端发现和使用ES6的特性。

你心里也许会这么想："太棒了！当所有浏览器都支持ES6时，我们就不需要再使用Babel了！"不过，就目前来看，如果我们希望使用下一版本的特性还离不开Babel。除非发生革命性的变化，在可预见的将来我们可能还需要使用Babel转译代码。

ES6的对象和数组

ES6规范为我们提供了一种使用对象和数组的新方法，并且可以在这些数据集中限定变量作用域的范围。这些特性包括解构，对象语义增强和扩展运算符。

解构赋值

解构赋值允许用户将某个对象内的字段的作用域本地化，并且可以声明哪些值是将要使用的。

以对象sandwich为例。它包含四个字段，但是我们只想使用其中两个字段的值。那么我们可以将其中的bread和meat属性的作用域本地化：

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

var {bread, meat} = sandwich

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

上述代码将bread和meat从对象中提取出来，然后为它们创建了相应的局部变量。当然，变量bread和meat的值还可以被修改：

```
var {bread, meat} = sandwich

bread = "garlic"
meat = "turkey"

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

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

我们还可以解构传入的函数参数。比如有一个主要用于记录人名的函数：

```
var lordify = regularPerson => {
  console.log(`${regularPerson.firstname} of Canterbury`)
}

var regularPerson = {
  firstname: "Bill",
  lastname: "Wilson"
}

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

除了使用点符号语法访问对象内部属性之外，我们还可以在regularPerson对象之外解构我们需要使用的值：

```
var lordify = ({firstname}) => {
  console.log(`${firstname} of Canterbury`)
}

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

解构还可以更具声明性，这意味着我们尝试完成的代码内容可以更具描述性。通过解构firstname，声明将会只使用变量firstname。关于声明式编程的细节将会在下一章深入展开。

数组中的值也可以被解构。假定我们希望使用数组中的第一个值作为变量名：

```
var [firstResort] = ["Kirkwood", "Squaw", "Alpine"]

console.log(firstResort) // Kirkwood
```

我们还可以使用逗号进行列表匹配，继而跳过不必要的值。当用逗号取代应该被跳过的元素时会触发列表匹配操作。对于同一个数组，我们可以通过逗号替换其中的前两个值来达到访问最后一个值的目的:

```
var [,,thirdResort] = ["Kirkwood", "Squaw", "Alpine"]

console.log(thirdResort) // Alpine
```

本章后续的内容中，将会结合数组解构和扩展运算符进一步拓展这个示例。

对象语义增强

对象语义增强和解构恰恰相反。它是重组或者回炉再造的过程。通过对象语义增强，我们可以从全局作用域中获得变量并将它们转换成一个对象：

```
var name = "Tallac"
var elevation = 9738

var funHike = {name,elevation}

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

name和elevation现在变成funHike对象的属性键了。

我们还可以通过对象语义增强或重组构造对象的方法：

```
var name = "Tallac"
var elevation = 9738
var print = function() {
  console.log(`Mt. ${this.name} is ${this.elevation} feet tall`)
}

var funHike = {name,elevation,print}

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

注意，这里我们是使用关键字this访问对象属性的。

当定义了对象方法后，就不需要再使用关键字function了（参见示例2-5）。

示例2-5：新旧对象语法比较

```
// 旧方案
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)
    }
}

// 新方案
const skier={
    name,
```

```
    sound,
    powderYell() {
        let yell = this.sound.toUpperCase()
        console.log(`${yell} ${yell} ${yell}!!!`)
    },
    speed(mph) {
        this.speed = mph
        console.log('speed:', mph)
    }
}
```

对象语义增强允许我们将全局变量变成对象，并且因为省去了不必要的关键字function，从而减少了代码的输入工作。

扩展运算符

扩展运算符是通过三个点号来完成多种工作任务的。首先，扩展运算符允许用户联合数组中的内容。比如我们有两个数组，可以通过将二者合二为一的方式构造第三个数组：

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

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

数组peaks和canyons中的所有元素都被移动到名为tahoe的新数组中了。

让我们来看看扩展运算符是如何帮助用户解决问题的。还是以上面示例中的peaks数组为例，假设我们希望获取该数组中的最后一个元素。可以使用Array.reverse方法对数组元素进行数组解构：

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

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

能看出来发生了什么吗？reverse函数实际上已经修改或者转变了数组的结构。如果采用了扩展运算符，我们不需要转变数组的原生结构，我们可以创建一个该数组的副本，然后对它进行翻转：

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

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

因为我们使用了扩展运算符拷贝数组，所以peaks数组仍然是完好无损的，后续操作中还能够以原生形式继续使用它。

扩展运算符至少还可以用来获取数组中的其余元素：

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

var [first, ...rest] = lakes

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

我们还可以使用扩展运算符将函数参数收集到一个数组中。这里将使用扩展运算符构建一个包含n个参数的函数，然后使用这些参数输出一些信息到控制台：

```
function directions(...args) {
  var [start, ...remaining] = args
  var [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"
)
```

函数directions采用扩展运算符接收参数。第一个参数被分配给了变量start。最后一个参数使用Array.reverse方法被分配给了变量finish。然后将使用参数数组的长度表示我们经历过的城镇数目。停靠站点的数目是参数数组的长度减去终点的差值。这提供了令人难以置信的灵活性，因为我们可以使用directions函数处理任意数目的停靠站点。

扩展运算符的特性还适用于对象。[注2]对象上使用扩展运算符的方法和使用数组类似。在这个示例中，我们将使用相同的方式将两个数组合二为一，构造一个新的数组，不过区别在于将使用对象进行处理，而不是数组：

```
var morning = {
  breakfast: "oatmeal",
```

注2：Rest/Spread Properties（*https://github.com/tc39/proposals*）。

```
  lunch: "peanut butter and jelly"
}

var dinner = "mac and cheese"

var backpackingMeals = {
  ...morning,
  dinner
}

console.log(backpackingMeals) // {breakfast: "oatmeal",
                                 lunch: "peanut butter and jelly",
                                 dinner: "mac and cheese"}
```

Promise对象

*Promise*对象为我们提供了一种更合理的方式处理异步行为。当创建一个异步请求时，会产生下列两种结果之一：一切都符合预期地进行或者存在一个错误。可能存在若干种成功的或不成功的请求。比如我们可以尝试多种方式获取数据来达成目标。我们也会接收到多种错误提示。Promise对象为我们提供了一种简单的方式，将结果简化为通过或者失败。

接下来创建一个异步Promise对象，用于从*randomuser.me*的API加载数据。该API包含诸如email地址、姓名、电话号码，地址等伪成员信息，非常适合生成虚拟数据。

`getFakeMembers`函数会返回一个新的Promise对象，它会向上述API发送一个请求。如果Promise对象请求成功，那么将会加载相关数据，否则将会报错：

```
const getFakeMembers = 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()
})
```

从上述代码可以看到，Promise对象已经被创建了，但是还未被调用。我们可以通过调用函数`getFakeMembers`，并将应该载入的成员数目作为参数传入来使用Promise对象。一旦Promise对象成功加载数据后，`then`函数可以通过链式调用执行某些任务。这就是被称为合成（composition）的机制。我们还可以使用另外一个回调函数处理异常：

```
getFakeMembers(5).then(
  members => console.log(members),
  err => console.error(
      new Error("cannot load members from randomuser.me"))
)
```

Promise对象处理异步请求的方法更容易，效果也更好，因为我们在JavaScript中必须处理大量的异步数据。读者还会发现在Node.js中Promise对象的身影随处可见，因此深刻理解Promise对象对于当前的JavaScript工程师来说至关重要。

类

在以前的JavaScript中，官方规范中并没有类的概念。类型定义是通过函数完成的。我们创建了一个函数，然后通过原型（prototype）在函数对象上定义方法：

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

Vacation.prototype.print = function() {
    console.log(this.destination + " | " + this.length + " days")
}

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

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

如果读者以前使用的是经典的面向对象程序设计方法进行开发的，那么这一点可能会让你倍感沮丧。

ES6引入了声明类的语法，但是JavaScript的工作机制仍然和以前是一样的。

函数即对象，继承是通过原型机制实现的，不过这些新增的语法对于以前是经典的面向对象程序开发者来说更易于接受：

```
class Vacation {

  constructor(destination, length) {
    this.destination = destination
    this.length = length
  }

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

}
```

大小写约定

首字母大写规则是指所有类都应该以大写字母开头。因此，我们将会对所有类名应用首字母大写规则。

创建好类对象之后，用户就可以使用关键字new创建一个该类的新实例了，并且用户还可以在类上调用自定义方法：

```
const trip = new Vacation("Santiago, Chile", 7);
console.log(trip.print()); // Chile will take 7 days
```

现在已经创建了一个类对象，用户可以通过创建一个新的Vacation实例来使用它，还可以对类进行扩展。当一个类被扩展之后，子类会继承父类的属性和方法。这里还可以对属性和方法进行精确控制，不过默认情况下，所有信息都将会被继承。

用户还可以将Vacation类当作一个抽象类，继而创建不同于Vacation类的多种类对象。比如，扩展自Vacation类的Expedition类可以包含一个新的gear参数：

```
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 ")}`)
  }
}
```

上述代码中的继承非常简单：子类继承了父类的属性。 不仅可以调用Vacation类的print方法，而且我们还可以在Expedition类中的print方法中附加一些新的内容。创建一个新实例的方法几乎差不多，使用new关键字定义一个变量：

```
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
```

类和原型继承

使用类之后仍然意味着用户使用的是JavaScript的原型继承机制。细究Vacation.prototype的内部细节，你会发现它的原型中包含了构造函数和print方法。

本书的内容将会涉及到类的概念，不过我们的重点会聚焦于函数式编程范式上。类还有其他特性，比如getters、setters和静态方法，不过本书对函数式编程技术的推崇要甚于面向对象技术。我们介绍上述内容是因为后续章节中创建React组件时会用到它们。

ES6模块

JavaScript的一个模块表示可以被轻松地集成到其他JavaScript文件中的一段可复用代码。直到最近，使用JavaScript模块的唯一方法是通过集成一个代码库来处理模块的导入和导出。[注3]

现在，随着ES6规范的出现，JavaScript自身也为模块提供了支持。

JavaScript模块可以存储在独立的文件中，一个文件对应一个模块。创建和导出一个模块中的数据有两种方式：用户可以从一个独立模块中导出多个JavaScript对象，或者从每个模块中导出一个JavaScript对象。

在示例2-6中，模块文件*text-helpers.js*中的两个函数将会被导出。

示例2-6：模块文件./text-helpers.js

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

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

关键字`export`还可以用来导出将会在其他模块中用到的任意JavaScript类型数据。在本示例中，`print`和`log`函数被导出了。任何在模块文件*text-helpers.js*中声明的其他变量将会以局部变量的形式存在。

有时用户也许只希望从某个模块中导出一个变量，那么这种情况下，用户可以使用`export default`来实现（参见示例2-7）。

注3：Mozilla Developer Network, JavaScript Code Modules（*https://mzl.la/2nvHwIR*）。

示例2-7：模块文件./mt-freel.js

```
const freel = new Expedition("Mt. Freel", 2, ["water", "snack"])

export default freel
```

当希望导出的数据只包含一种类型时，用户可以使用export default替换export。当然，export和export default都适用于任意JavaScript数据类型：基元、对象、数组和函数。[注4]

还可以使用import语句在其他JavaScript文件中调用模块。包含多个exoprt语句的模块还可以进行对象解构。使用epxort default的模块还可以被导入到单个变量中：

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

print('printing a message')
log('logging a message')

freel.print()
```

用户甚至可以通过别名实现模块变量的本地化：

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

p('printing a message')
l('logging a message')
```

用户还可以通过星号*将所有信息导入单个变量中：

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

目前并非所有主流的浏览器都为ES6的模块提供了支持。Babel支持ES6的模块特性，所以本书将会使用它。

CommonJS

CommonJS是所有版本的Node.js都支持的模块模式。[注5]用户可以通过Babel和webpack使用这些模块。在CommonJS中是通过module.exports语句导出JavaScript对象的，参见示例2-8。

注4：Mozilla Developer Network，“Using JavaScript Code Modules”。（*https://mzl.la/2nvBS9r*）。

注5：Node.js 文档，"Modules”。（*https://nodejs.org/docs/latest/api/modules.html*）。

示例 2-8：./txt-helpers.js

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

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

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

CommonJS并不支持import语句。相反，模块导入是通过require函数实现的：

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

JavaScript的发展真可谓日新月异，并且满足了工程师对语言特性日益增长的需求。目前主流的浏览器也快速地实现了对ES6标准和其他特性的兼容，因此毫不犹豫地使用这些特性绝对是明智之举。[注6]ES6标准中的很多新特性并不陌生，因为它们都支持函数式编程。在JavaScript函数式编程方面，我们可以将程序代码看作一组函数的集合，而且可以将它们组合起来构造相关的应用程序。下一章，我们将进一步了解函数式编程的细节，并讨论用户可能希望使用它们的原因。

注6：希望了解最新的兼容性资讯，可以参考 ES6 compatibility table。（*http://kangax.github.io/compat-table/es6/*）。

第3章

JavaScript函数式编程

当踏上React编程之旅时，你会发现函数式编程的概念随处可见。越来越多的JavaScript项目采用了函数式编程技术。

也许你已经在不假思索的情况下编写过JavaScript的函数式代码。如果曾经在一个数组上调用过map或reduce方法，那么你已经在成为一名函数式程序员的道路上披荆斩棘了。React、Flux和Redux都适用JavaScript的函数式编程范式。了解函数式编程的基本概念将会有助于提高读者构建React应用的能力。

如果读者希望对函数式编程风格追根溯源，那么答案是20世纪30年代lambda演算或λ演算的发明。[注1]函数自17世纪诞生以来，一直是作为微积分的一部分而存在的。函数可以作为函数的参数进行传递，还可以作为函数的执行结果被返回。更复杂的函数被称为高阶函数，它可以精确地控制函数，既可以将函数当作参数传递，也可以将函数作为执行结果返回，或者二者兼而有之。在20世纪30年代，Alonzo Church在普林斯顿大学用高阶函数做实验时发明了lambda演算。

20世纪50年代初，John McCarthy借鉴了λ-演算的概念，并将它应用到一门新的名为Lisp的编程语言上。Lisp实现了高阶函数的概念，并将函数作为第一类成员或者第一类公民。一个函数被当作第一类成员时，它不仅可以被声明为一个变量，而且可以被当作函数参数传递。这些函数甚至可以作为函数的执行结果被返回。

注1：Data S. Scott，“λ-Calculus: Then & Now”（*http://turing100.acm.org/lambda_calculus_timeline.pdf*）。

在本章中，我们将会介绍一些基本的函数式编程概念，同时还会阐述如何在JavaScript中使用函数式编程技术。

什么是函数式编程

JavaScript可以进行函数式编程，因为JavaScript中的函数就是第一类公民。这意味着变量可以做的事情函数同样也可以。ES6标准中还添加了不少语言特性，可以帮助用户更充分地使用函数式编程技术，其中包括箭头函数、Promise对象和扩展运算符等（详情可以参考第2章）。

在JavaScript中，函数可以表示应用程序中的数据。细心的读者应该已经发现，可以使用关键字var像声明字符串、数字或者其他任意变量那样声明函数：

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

log("In JavaScript functions are variables")

// In JavaScript, functions are variables
```

在ES6规范下，我们可以使用箭头函数编写同样的函数。函数式程序员会编写大量的小型函数，使用箭头函数会方便很多：

```
const log = message => console.log(message)
```

因为函数就是变量，我们可以将它们添加到对象中：

```
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
```

这些语句的效果殊途同归：它们都将一个函数存储到了一个名为log的变量中。此外，关键字const被用来声明第二个函数，主要的目的是防止该函数被重写。

在JavaScript中，我们还可以将函数添加到数组中：

```
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
```

函数可以像其他变量那样，作为其他函数的参数进行传递：

```
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
```

函数还可以像变量那样，作为其他函数的执行结果被返回：

```
var 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!!!
```

最后两个示例是和高阶函数有关的，这种函数既可以接收作为参数的函数，也可以将其他函数作为返回值。采用了ES6语法之后，我们可以使用箭头表示同样的高阶函数createScream：

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

从现在开始，我们需要注意函数声明过程中使用箭头的数目。一个以上的箭头表示我们声明的是高阶函数。

我们可以说JavaScript就是函数式编程语言，因为它的函数是第一类成员。这意味着函数就是数据。它们可以像变量那样被保存、检索或者在应用程序内部传递。

命令式和声明式

函数式编程还是更广义编程范式的一部分：声明式编程。声明式编程是一种编程风格，采用该风格的应用程序代码有一个比较突出的特点，那就是对执行结果的描述远胜于执行过程。

为了加深对声明式编程的理解，我们将会把它和命令式编程进行对比，该编程风格的特点是,其代码重点关注的是达成目标的具体过程。接下来以一个比较常见的任务为例：让字符串兼容URL格式。一般来说，这可以通过连字符替换字符串中的所有空格实现，因为空格对URL地址的兼容性不佳。首先，我们使用命令式编程风格完成此任务：

```
var string = "This is the midday show with Cheryl Waters";
var urlFriendly = "";

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

console.log(urlFriendly);
```

在本示例中，我们循环遍历了字符串中的每个字符，并对其中遇到的空格进行了替换。从该程序的结构来看，它只关注如何完成这样一个任务。我们使用了for循环和if语句，并使用同等的运算符进行赋值。单独看这些代码并不能告诉我们更多信息。命令式编程风格需要辅以大量注释说明帮助用户理解它的具体用途。

现在我们使用声明式编程风格来解决同样的问题：

```
const string = "This is the mid day show with Cheryl Waters"
const urlFriendly = string.replace(/ /g, "-")

console.log(urlFriendly)
```

这里我们使用了string.replace方法和一个正则表达式将字符串中的空格替换成连字符。使用string.replace方法是一种说明可能会发生什么的方式：字符串中的空格将会被替换。如何处理空格的细节被抽象封装到了replace函数内部。在一个声明式程序中，语法本身描述了将会发生什么，相关的执行细节被隐藏了。

声明式程序易于解释具体用途，因为其代码本身就描述了将会发生什么。比如阅读如下示例代码，它描述了自API载入成员后将会发生什么：

```
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);
```

声明式方法更易读，因此也更方便解释具体用途。每个这类函数的具体实现细节都被封装起来。这些小型函数命名规范，并且被有机地组合在一起用于描述成员数据从被载入、保存到在地图上显示的过程，并且这方面不需要太多的注释信息。本质上来说，使用声明式编程编写的应用程序更容易解释具体用途，当一个应用易于解释具体用途时，该应用也更易于进行功能扩展。[注2]

现在，我们来考虑构建文档对象模型或者DOM(*https://www.w3.org/DOM/*)这样一个任务。命令式编程方法会把重点放在如何构建DOM上：

```
var target = document.getElementById('target');
var wrapper = document.createElement('div');
var headline = document.createElement('h1');

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

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

上述代码将主要精力聚焦于创建元素、设置元素，以及将它们添加到文档中。该命令式风格构建的DOM程序代码，将会非常难于修改、添加特性或者扩展到10000行以上。

现在让我们来看看如何使用React组件以声明式风格构造一个DOM：

```
const { render } = ReactDOM

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

注2：希望了解更多声明式编程风格的详情可以参考声明式编程wiki页面(http://c2.com/cgi/wiki?DeclarativeProgramming)。

```
        </div>
    )

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

React采用了声明式编程风格。这里组件Welcome描述了将会被渲染的DOM。render函数调用了组件中声明的指令来构建DOM,它封装了DOM被渲染的具体细节。我们可以清楚地知道，用户希望在ID为'target'的元素中渲染Welcome组件。

函数式编程基本概念

现在读者已经了解了函数式编程，以及“函数”和“声明式”的意义，接下来我们将继续介绍函数式编程的核心概念：不可变性、纯函数、数据转换、高阶函数和递归。

不可变性

不可变性就是指不可改变。在函数式编程中，数据是不可变的，它们永远无法修改。如果用户需要对外公开自己的出生证明信息，但是希望修改或者删除自己的私人信息，那么有两个选择：用户可以拿一支大号荧光笔将自己的私人信息划掉，或者找一台复印机。找到一台复印机后，复印一份用户的出生证明文件，然后用大号荧光笔在该复印件上将自己的私人信息划掉可能是一个更好的办法。这样一来用户就拥有了一个经过编辑的出生证明文件方便和其他人共享，而且出生证明的原件仍然完好无损。

这也是不可变数据在应用程序中的工作机制。在不修改原生数据结构的前提下，我们在这些数据结构的拷贝上进行编辑，并使用它们取代原生的数据。

为了了解不可变性的工作机制，让我们看看它是如何修改数据的。现在来考察一个表示颜色的对象：

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

我们可以构造一个为颜色评分的函数，并使用它来修改颜色对象的颜色评分：

```
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，函数参数会被指向实际的数据。像这样设置颜色的评分是一种比较糟糕的做法，因为它修改异化了原来的颜色对象（想象一下，如果你和其他人合作时，需要给对方共享自己的出生证明文件，对方将出生证明文件原件还给你时，其中重要的信息都被黑色标记覆盖了）。我们可以重写颜色评分函数从而达到不破坏原生物（颜色对象）的目的：

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

console.log(rateColor(color_lawn, 5).rating)    // 5
console.log(color_lawn.rating)                  // 4
```

这里我们使用Object.assign方法修改颜色评分。Object.assign方法是一种拷贝机制，它会提供一个空白对象，将颜色对象拷贝到该对象上，然后在该拷贝上重写颜色评分值。现在我们可以在不修改原生对象的情况下，获得一个包含新评分值的颜色对象了。

我们可以使用ES6规范下的箭头函数和ES7规范下的对象扩展运算符编写同样的函数。rateColor函数使用扩展运算符将颜色对象拷贝到一个新对象中，然后重写它的评分：

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

采用了JavaScript新版本语法特性的rateColor函数几乎和上一个函数一样。它将颜色对象视为一个不可变对象，这样一来用到的语法更少，而且可看起来更简洁一些。注意，我们使用圆括号将返回的对象包裹了起来。在箭头函数中，这是一个必要的步骤，因为箭头不能指向一个对象的花括号。

接下来考察一个包含若干颜色名称的数组：

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

我们可以构造一个函数，使用Array.push方法将颜色添加到该数组中：

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

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

不过Array.push方法并不是一个不可变函数。addColor函数会通过向它添加其他字段的形式改变原来的数组。为了确保颜色数组的不可变性，我们必须使用Array.concat方法取而代之：

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

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

Array.concat方法会将数组串联起来。这种情况下，它会生成一个包含新的颜色标题的对象，并将它添加到原生数组的副本上。

用户还可以使用ES6的扩展运算符串联数组，同时该操作符可以使用同样的机制拷贝对象。这里使用了JavaScript的新语法，其效果和前面的addColor函数是等价的：

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

该函数拷贝了原生的列表到一个新的数组中，然后将一个包含颜色标题的新对象添加到上述拷贝中。它是不可变的。

纯函数

纯函数是一个返回结果只依赖于输入参数的函数。纯函数至少需要接收一个参数并且总是返回一个值或者其他函数。它们不会产生副作用、不修改全局变量，或者任何应用程序的State。它们将输入的参数当作不可变数据。

为了理解纯函数，接下来看一个非纯函数：

```
var 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}
```

函数selfEducate不是一个纯函数。它并没有接收任何参数，并且也没有返回一个值或者函数。它还修改了其作用域之外的变量：Frederick。一旦selfEducate函数被执行后，“世界”就发生了变化。它产生了副作用：

```
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}
```

纯函数的可测试性

纯函数天生是可测试的。它们不会改变执行环境或者“世界”中的任何东西，因此不需要装配或者卸载复杂的测试环境。纯函数需要访问的任意数据都是通过参数进行传递的。当测试一个纯函数时，用户控制着参数，因此也可以预估执行结果。和测试有关的详情可以参考第10章。

函数selfEducate也是一个非纯函数：它会产生副作用。调用该函数之后，它会修改传递给它的对象。如果我们可以将传递给该函数的参数当作不可变数据，那么我们就可以得到一个纯函数。

让我们来看这样一个函数：

```
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}
```

最后，这个版本的selfEducate成了一个纯函数。它的返回值是根据传递给它的参数生成的：person。它在不改变传递给它的参数的情况下，返回了一个新的person对象，因此不会产生副作用。

现在让我们考察一个非纯函数修改DOM的示例：

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

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

Header函数创建了一个包含特定文本的h1标签元素，并将它添加到了DOM中。该函数是非纯函数。它并没有返回一个值或者函数，并且产生了副作用：对DOM做了修改。

在React中，UI是用纯函数表示的。在接下来的示例中，Header函数将会是一个纯函数，并且可以像上述示例中那样创建h1标签元素。不过该函数本身并不会产生副作用，因为它没有修改DOM。该函数将会创建一个h1标签元素，并且被应用程序其他部分调用继而修改DOM：

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

纯函数是函数式编程中的另外一个核心概念。它会使你的开发工作更容易，因为它们不会影响应用程序的状态。当编写函数时，请务必遵循以下三条原则：

1. 函数应该至少接收一个参数。
2. 函数应该返回一个值或者其他函数。
3. 函数不应该修改或者影响任何传给它的参数。

数据转换

如果数据是不可变的，那么应用程序内部如何进行状态转换呢？ 函数式编程的做法是将一种数据转换成另外一种数据。我们使用函数生成转换后的副本。这些函数使得命令式的代码更少，并且大大降低了复杂度。

用户不需要通过一个特定的框架来了解如何从一种数据集转换到另外一种。JavaScript语言已经内置了完成该任务所需的工具。为了充分利用JavaScript的函数式特性，下面两个核心函数是用户必须熟练掌握的：Array.map和Array.reduce。

在本章中，我们将对这些以及其他核心函数如何将数据从一种类型转换到另一种类型做一个简要介绍。

比如包含中学名称的这样一个数组：

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

我们可以通过Array.join方法将这些或者其他字符串转换成逗号分隔的列表:

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

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

join是JavaScript数组内置的方法，我们可以使用它将数组中的元素转换成包含分隔符的字符串，同时原有的数组仍然完好无损，join只是为用户提供了另外一种对原有数组元素的观察视角。上述字符串构造的细节对程序员来说已经被隐藏了。

如果我们想创建一个函数来创建中学名以字母“W”开头的数组，那么我们可以使用Array.filter方法：

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

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

Array.filter是JavaScript内置的函数，它根据源数组创建了一个新的数组。该函数会使用一个谓词作为它的唯一参数。一个谓词就是一个永远都会返回布尔值的函数：true或false。Array.filter方法会在数组中的每个元素上调用该谓词。上述元素会作为谓词函数的参数进行传递，返回值将会用来决定该元素是否可以被添加到新的数

组中。 在这种情况下，Array.filter会检查每个学校的名称是否是以字母“W”开头的。

当需要从一个数组中移除某个元素时，我们倾向于使用Array.filter方法替代Array.pop或Array.splice方法，因为Array.filter方法是不可变的。在接下来的示例中，函数cutSchool会返回一个过滤了特定学校名称的新数组：

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

console.log(cutSchool("Washington & Lee", schools).join(" * "))

// "Yorktown * Wakefield"

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

// Yorktown
// Washington & Lee
// Wakefield
```

在这种情况下，函数cutSchool用来返回一个不包含“Washington & Lee”的新数组。然后在返回的新数组上使用join方法，在剩下的两个学校名称上构造一个用星号做分隔符的字符串。cutSchool是一个纯函数。它获得了学校名称列表和需要被移除的元素名称等参数，然后返回了一个不包含特定学校名称的新数组。

函数式编程中另外一个必不可少的数组函数是Array.map。和使用谓词作为参数相反，Array.map方法会接收一个函数作为它的参数。该函数会在访问数组中的每个元素时执行，无论该元素是否会被添加到新的数组中：

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

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

// Yorktown High School
// Washington & Lee High School
// Wakefield High School

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

// Yorktown
// Washington & Lee
// Wakefield
```

这种情况下，map函数被用来将“High School”追加到每个学校名的后面。原来的学校名称数组仍然完好无损。

在最后一个示例中，我们从一个字符串数组构造了一个新的字符串数组。map函数可以构造任意对象、数值、数组、函数，以及其他JavaScript类型的数组。这里的示例中map函数会为每所学校返回一个对象：

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

console.log( highSchools )

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

一个包含对象的数组是根据一个包含字符串的数组生成的。

如果用户需要创建一个纯函数来修改对象数组中的某个对象，map函数也能胜任这项工作。在接下来的示例中，我们将会在不改变数组schools的情况下，将其中的“Stratford”改为“HB Woodlawn”：

```
let schools = [
    { name: "Yorktown"},
    { name: "Stratford" },
    { name: "Washington & Lee"},
    { name: "Wakefield"}
]

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

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

数组schools是一个对象数组。变量updatedSchools调用函数editName，然后我们将希望更新的学校名、新的学校名和schools数组作为参数传递给该函数。这些变更是在新的数组上进行的，并不会对原有的数组产生影响：

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

在editName函数内部，map函数根据原数组创建一个新的对象数组。Array.map会将每个元素的索引作为第二个参数注入回调函数中，即变量i。当i和目标元素的索引不相等时，我们只需将相同元素插入新数组即可。当i和目标元素的索引相等时，我们会使用一个新对象替换新数组中相同索引位置的对象。

函数editName完全可以写成一行的形式。这里是一个使用if/else语句简写形式构造的相同函数：

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

如果用户需要将一个数组转换成一个对象，那么用户可以通过Array.map搭配Object.keys一起使用来达到上述目的。Object.keys方法可以用来获得某个对象中属性键的数组。

比如我们可以将schools对象转换成schools的数组：

```
const schools = {
  "Yorktown": 10,
  "Washington & Lee": 2,
  "Wakefield": 5
}

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

console.log(schoolArray)

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

在这个示例中，Object.keys会返回一个包含学校名称的数组，然后我们可以使用map方法根据上述数组构造一个同样长度的新数组。新对象的名称将会使用属性键设置，同时还会使用对应的数值设置wins。

目前为止我们已经学习了使用Array.map和Array.filter方法转换数组。还学习了通过Object.keys搭配Array.map方法将数组转换成对象。函数式编程武器库中的终极武器必须具备将数组转换成基元或者其他对象类型的能力。

reduce和reduceRight函数可以用来将数组转换成任意值，比如数字、字符串、布尔值、对象，甚至是函数。

接下来的示例将会演示如何在一个数字数组中找出最大值。我们需要将一个数组转换成数字，为此，可以使用reduce方法：

```
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
```

数组ages已经简化为一个单个值：最大年龄64。reduce方法会接收两个参数：回调函数和原生值。在这种情况下，原生值为0，它会将初始的最大值设为0。在访问数组中每个元素时会执行回调函数。回调函数第一次被调用时，年龄是数组中的第一个值21，最大值是初始值0。回调函数会返回两个数值中较大的那个，即21，然后它将会作为下一次迭代中的最大值。每次迭代都会将年龄和最大值比较，然后返回二者较大的那个值。最后，比较完数组中的最后一个数之后，比较结果将作为上一次回调函数的返回值返回。

如果我们将上述函数中的console.log语句移除，并使用if/else语句的简写形式替换相关代码，我们就可以使用下列代码计算出任意数组中的最大值：

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

Array.reduceRight

Array.reduceRight和Array.reduce的工作原理类似，主要差别在于，它是从数组的末尾开始处理元素的，而不是数组的起始位置。

有时我们需要将一个数组转成一个对象。下面的示例将使用reduce函数将一个包含颜色的数组转换成哈希对象：

```
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
//   }
// }
```

在这个例子中，传递给reduce函数的第二个参数是一个空对象。这也是我们给哈希对象设置的初始值。在每次迭代期间，回调函数使用括号标记为哈希对象添加了一个新的属性键，同时使用数组中id字段的值作为上述属性键的值。Array.reduce方法可以通过这种方式将一个数组简化为单一的值，这种情况下是指一个对象。

我们甚至还可以使用reduce方法将数组转换成完全不同的其他数组。比如将一个包含多个相同值的数组简化为一个不包含重复值的数组。reduce方法完全可以用来完成这个任务：

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

const distinctColors = colors.reduce(
    (distinct, color) =>
        (distinct.indexOf(color) !== -1) ?
            distinct :
            [...distinct, color],
    []
)

console.log(distinctColors)

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

在这个示例中，数组colors被转化成一个不包含重复值的数组。 传递给reduce函数的第二个参数是一个空数组。这是distinct数组的初始值。当distinct数组中不包含某个特定的颜色时，该颜色将会被添加到其中。否则就会忽略该颜色，当前的distinct数组会被返回。

map和reduce是任何函数式程序员的主要武器，对于JavaScript也不例外。如果你希望成为一名熟练的JavaScript工程师，那么最好掌握这些函数。从其他数据源创建一个新的数据集的能力是程序员的必备技能，并且对于任何编程范式都是非常有用的。

高阶函数

高阶函数的使用对于函数式编程也是必不可少的。前文已经提到过高阶函数，本章也用到不少。高阶函数是可以操作其他函数的函数。它们可以将函数当作参数传递，也可以返回一个函数，或者二者兼而有之。

第一类高阶函数是将其他函数当作参数传递的函数。Array.map、Array.filter和Array.reduce都可以将函数当作参数进行传递，所以它们都是高阶函数。[注3]

接下来看看如何实现一个高阶函数。在下列示例中，我们将创建一个名为invokeIf的回调函数，当条件经过测试为true时将会调用一个回调函数；当条件经过测试为false时，将会调用另外一个回调函数：

```
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预计会包含两个函数：一个是为了处理通过条件的情况，另外一个是为了不符合条件值为false的情况。这一过程是通过将showWelcome和 showUnauthorized函数传递给invokeIf函数来演示的。当条件为true时，会执行showWelcome函数；当条件为false时，将会执行showUnauthorized函数。

返回其他函数的高阶函数还可以帮助我们处理JavaScript中复杂的异步操作。它们可以帮助用户方便地创建需要使用或者复用的函数。

柯里化（Currying）是一种采用了高阶函数的函数式编程技巧。柯里化实际上是一种将某个操作中已经完成的结果保留，直到其余部分后续也完成后可以一并提供的机制。这是通过在一个函数中返回另外一个函数实现的，即柯里函数。

注3：希望进一步了解高阶函数的详情，可以参考第5章。

下面是一个柯里化的例子。函数userLogs会保存一些信息（username），当其余的信息（message）可用时返回一个函数方便其他函数调用或者复用。在本示例中，会为所有相关的username预置日志信息。注意，其中的getFakeMembers函数会返回一个第2章介绍过的Promise对象：

```
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是一个高阶函数。log函数是由userLogs函数返回的，并且每次都会调用log函数，“grandpa23”是为相关信息预置的。

递归

递归是用户创建的函数调用自身的一种技术。一般来说，在解决实际问题涉及到循环时，递归函数可以提供一种替代性的方案。考虑这样一个问题，从10开始倒数计数。我们可以创建一个for循环解决这个问题，也可以创建一个递归函数解决它。在本示例中，倒计时是通过递归函数实现的：

```
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
```

函数countdown需要接收一个数字和一个函数作为参数。在本示例中，它是通过数字10和一个回调函数执行调用的。当执行countdown函数时，回调函数也同时执行了，其主要的用途是记录当前的值。然后countdown函数会检查接收到的数值是否大于0，如果条件为真，那么它会使用上述数值自动减1之后的值作为参数调用自身。最终，该数值将会变成0，countdown函数会顺次返回这些值以便记录调用堆栈。

浏览器堆栈调用的不足之处

应该尽可能地使用递归解决循环有关的问题，不过并非所有JavaScript引擎都对大量的递归调用做了性能优化。过多的递归调用会导致JavaScript报错。可以通过一些高级技术清理调用堆栈并停止递归调用来避免这些错误。未来的JavaScript引擎预计会完全解决调用堆栈的不足。

递归是另外一个可以很好地处理异步过程的函数式编程技术。函数调用自身是也能实现延迟调用。

countdown函数可以经过修改实现延迟计数的功能。修改版的countdown函数可以用来创建一个倒计时时钟：

```
const countdown = (value, fn, delay=1000) => {
    fn(value)
    return (value > 0) ?
        setTimeout(() => countdown(value-1, fn), delay) :
        value
}

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

在这个例子中，我们创建了一个10秒的倒计时时钟，起初会在函数中调用countdown函数一次，并将数值10作为参数传递给它，以便为countdown函数添加日志信息。和马上再次调用自身相反，countdown函数会等待1秒之后再调用自身，这样一来就创建了一个时钟。

递归还是一种搜索数据结构的好方法。用户可以使用递归来遍历文件夹，直到找到文件夹中符合条件的文件。还可以通过递归遍历HTML的DOM，直到找到某个不包含任何子节点的元素。在下一个示例中，我们将使用递归深入某个对象查找符合条件的嵌套值：

```
var 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"
```

函数deepPick可以用来访问Dan的类别，并且可以将相关信息立刻存储到第一级对象中，也可以继续深入嵌套对象查找Dan的名字。发送一个包含点符号的字符串，我们可以声明深入查找对象内部嵌套结构的具体位置：

```
const deepPick = (fields, object={}) => {
   const [first, ...remaining] = fields.split(".")
   return (remaining.length) ?
      deepPick(remaining.join("."), object[first]) :
      object[first]
}
```

函数deepPick不仅可以返回一个值，而且可以调用自身，直到它最终返回一个值。首先，该函数会将接收到的字符串根据点符号进行字段分割，然后将结果作为一个数组返回，接着使用数组解构将其中的第一个值和其余的值分开。如果数组中仍然有元素，那么deepPick会调用自身并记录这些数据之间细微的差别，使得它可以进一步深入挖掘。

该函数会持续调用自身，直到字符串中不包含点符号标记，这意味着其中没有字段存在了。在这个例子中，用户可以看到数组中第一个元素、其余的元素和object[first]的值在deepPick函数遍历迭代过程中发生的变化：

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

// 第一次迭代
//  first = "data"
//  remaining.join(".") = "info.fullname.first"
//  object[first] = { gender: "male", {info} }

// 第二次迭代
//  first = "info"
//  remaining.join(".") = "fullname.first"
//  object[first] = {id: 22, {fullname}}
```

```
// 第三次迭代
//  first = "fullname"
//  remaining.join("." = "first"
//  object[first] = {first: "Dan", last: "Deacon" }

// 最终结果
//  first = "first"
//  remaining.length = 0
//  object[first] = "Deacon"
```

递归是一种非常强大的函数式编程技巧，并且易于实现。在实际开发过程中处理循环操作时应该尽量使用递归。

合成

函数式编程会将具体的业务逻辑拆分成小型的纯函数，以便能够将精力聚焦于特定任务。最终，用户将会需要把这些小型函数整合到一起。具体来说，用户可能需要合成它们，以串联或者并联的方式对它们进行调用，或者将它们合成为一个更大的函数，直到构造出一个应用程序为止。

对于合成来说，与之有关的实现、模式和技术真可谓五花八门。读者可能比较熟悉的一种方式就是链式调用。在JavaScript中，函数可以使用点符号连接在一起，其作用是获得上一个函数的返回值。

字符串有一个replace方法，replace方法返回的模板字符串也包含一个replace方法。因此我们可以在转换一个字符时使用点符号将replace方法串联起来实现链式调用。

```
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"
```

在这个例子中，template是一个字符串。通过将replace方法和末尾的template字符串链式连接，我们可以将字符串中的小时、分钟、秒和日期替换成新的值。template本身并没有发生变化，并且可以继续复用以便创建更多的时钟显示。

链式调用只是合成技术之一，还有其他方法可供用户选择。合成的目标是“通过整合若干简单函数构造一个更高阶的函数”。[注4]

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

both函数充当了两个函数之间传递数据的管道。函数civilian的输出成了函数appendAMPM的输入，并且我们可以使用这些合二为一的函数修改日期。不过这种语法难于理解，因此维护和扩展也比较困难。当我们传递某个值需要穿过20个不同的函数时，会发生什么呢？

一个更优雅的方法是创建一个高阶函数，以便用户可以将这些函数合成一个更大的函数。

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

both(new Date())
```

该方法看上去效果更好。它还能够方便地进行扩展，因为我们后续可以添加更多函数。它还可以方便地修改被合成函数的顺序。

compose函数是一个高阶函数。它将函数作为参数，然后返回单一的值。

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

函数compose接收函数作为参数，并且会返回一个独立的函数。在上述代码中，扩展运算符主要用于将这些函数参数添加到名为fns的数组中。随后将会返回可以接收一个参数的函数，即arg。当该函数被调用时，数组fns将会作为在函数之间传递数据的起始管道。这些参数会作为composed的初始值，然后它会累积reduce回调函数的返回值。注意，回调函数会接收两个参数:一个是composed，另外一个是代表函数的f。每个函数被调用后的结果累积就是上一函数的输出结果。总之，最后一个函数被调用并返回了最终结果。

注4： Functional.js Composition。

这是一个非常简单的合成函数示例，主要用于演示合成技术。当需要处理一个以上的参数或者非函数的参数时，函数会变得更复杂。合成技术的其他实现也许会用到reduceRight方法，[注5]其中合成的函数是逆序调用的。

综合应用

现在我们已经介绍过函数式编程的核心概念，接下来让我将这些概念应用到实际工作中，构建一个小型的JavaScript应用。

因为JavaScript可以让用户从函数式编程范式中解脱出来，并且用户也不必循规蹈矩，关注重点即可。遵循下列三个简单的规则将有助于帮助用户达成目标：

1. 保持数据的不可变性。
2. 确保尽量使用纯函数，只接收一个参数，返回数据或者其他函数。
3. 尽量使用递归处理循环（如果有可能的话）。

我们的目标是构建一个滴答作响的时钟。时钟需要显示小时、分钟、秒，以及当地时间的日期。

每个字段必须保证是双位数字，这意味在类似1或者2这样的单位数字时，需要在它们前面加上0来补足位置。时钟必须发出嘀嗒声并且显示每秒的时间变化。

首先，让我们看看使用命令式风格编写的显示时钟的方案。

```
//记录每秒的时钟状态变化
setInterval(logClockTime, 1000);

function logClockTime() {

  //获取本地时间格式的时钟时间字符串
  var time = getClockTime();

  //清空控制台并记录时间
  console.clear();
  console.log(time);
}

function getClockTime() {

  //获取当前时间
```

注5： 合成方法基于Redux的另外一个实现。

```
  var date = new Date();
  var time = "";

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

//转换成当地时间
if (time.hours == 12) {
  time.ampm = "PM";
} else if (time.hours > 12) {
  time.ampm = "PM";
  time.hours -= 12;
}

//为小时位置上预置0，以便构造双位数字
if (time.hours < 10) {
  time.hours = "0" + time.hours;
}

//为分钟位置上预置0，以便构造双位数字
if (time.minutes < 10) {
  time.minutes = "0" + time.minutes;
}

//为秒位置上预置0，以便构造双位数字
if (time.seconds < 10) {
  time.seconds = "0" + time.seconds;
}

//将时钟时间格式化为一个字符串"hh:mm:ss tt"
return time.hours + ":"
    + time.minutes + ":"
    + time.seconds + " "
    + time.ampm;

}
```

这个解决方案非常简单。它可以工作，其中的代码注释可以帮助我们了解具体发生了什么。不过这些函数大而复杂。每个函数做的事情太多。它们很难理解，需要注释辅助用户阅读，并且也不好维护。接下来让我们看看如何使用函数式编程方法构建一个扩展性更强的应用。

我们的目标是将该应用的业务逻辑分解成更小的部分或者函数。每个函数将聚焦于单个任务，然后我们会将它们合成更大的函数，以便我们可以创建一个时钟程序。

首先，我们将创建为程序提供输入数值和管理控制台的函数。我们需要一个函数提供1秒的数据，一个函数提供当前时间，以及一对用于在控制台上显示信息和清空信息的函数。在函数式编程中，应该尽量使用函数代替变量赋值。我们将会根据需要调用相关函数获取输入数据。

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

接下来我们还需要一些用于转换数据的函数。这三个函数将会用来将日期对象转换成方便时钟程序调用的对象：

serializeClockTime

接收一个Date对象，为时钟构造一个包含时、分、秒的对象。

civilianHours

接收一个时钟对象，返回一个小时被转换成本地时间的对象。比如：将1300转换成时钟上的1点。

appendAMPM

接收时钟对象，然后在该对象中追加日期，AM（上午）或者PM（下午）标记。

```
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"
    })
```

这三个函数用于在不改变原有数据的情况下转换数据。它们将接收到的参数当作不可变对象。

下面我们将会用到一些高阶函数：

display

获取目标函数，返回的函数将会把时间发送到目标。在本示例中目标是console.log。

formatClock

获得一个模板字符串，然后使用它对时钟时间进行相应的格式化。在本示例中，模板是“hh:mm:ss tt”。因此，formatClock将会使用时、分、秒和当天的日期替换其中的占位符。

prependZero

获取某个对象的属性键作为参数，并预先将0赋值给该对象的属性键。它会将属性键的值添加到特定字段，如果其中的数字是小于10的，那么就会在该数字前面用0补足双数。

```
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]
  })
```

将会调用这些高阶函数来创建函数，以便每次嘀嗒声响起时格式化时钟时间。函数formatClock和prependZero都会被调用一次，用于初始化必需的模板和属性键。

它们返回的内部函数将会每隔一秒调用一次，以便格式化显示屏上的时间。

现在我们已经准备好构造一个滴答作响的时钟所需的全部函数了，接下来将会对它们进行合成。我们将使用上一小节介绍过的compose函数对这些函数进行合成：

convertToCivilianTime

一个独立函数，将会获取时钟时间作为参数，并通过本地时间规范将时钟时间转换成本地时间。

doubleDigits

一个独立函数，将会获取本地时间，并确保时、分、秒是以双位数格式显示的，在必要的情况下会在单位数时间数字前补0。

startTicking

通过设置时间间隔启动时钟程序，将会每隔一秒执行一次回调程序。回调函数是由上述所有子程序合成的。每隔一秒控制台都会被清空、获取当前时间、转换格式、本地化、格式化，然后显示。

```
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()
```

声明式版本的时钟程序和命令式版本的输出结果是一样的。不过，这种方法有诸多好处。首先，所有函数易于测试和复用。它们还可以在将来的时钟程序或者其他数字显示程序中复用，而且这个程序非常容易进行功能扩展。它没有副作用。函数外部不存在全局变量。它们可能仍然存在不少问题，不过诊断这些问题也是非常容易的。

在本章中，我们介绍了函数式编程的基本概念。本书在讨论React和Flux的最佳实践时，还会继续演示这些类库是如何充分发掘函数式编程的潜力的。下一章，我们将会正式介绍React，深入了解它的开发理念，指导用户进行实际开发。

第4章

React进阶

为了了解React在浏览器中的运行机制，本章将会单独介绍React。下一章将会介绍JSX，即JavaScript下的XML格式。也许读者已经和React打过交道，但是从未看过将JSX格式的代码转译成React源码。用户完全可以在不了解React源码的情况下放心使用React。不过，如果你希望花时间了解应用场景背后的具体细节，那么将会有助于提高开发效率，特别是在查找代码Bug时。这也是本章的目标：了解React内部的运行机制。

建立页面

为了在浏览器中使用React，我们需要引用两种类库：React和ReactDOM。React库是用来创建视图的。ReactDOM库是用来在浏览器中渲染UI的。

ReactDOM

在0.14版时，React和ReactDOM被一分为二独立打包了。发行说明指出："React的美丽和精髓和浏览器、DOM的关联甚少，这个(一分为二)方式，为编写能够在React和React Native应用之间的Web版本实现组件共享铺平了道路"。[注1] React将不仅限于在浏览器中进行元素渲染，未来的版本将会支持多平台的元素渲染。

注1： Ben Alpert, "React v0.14" (*http://bit.ly/2nvPHEQ*), React blog, October 7, 2015。

我们还需要一个HTML元素，以便ReactDOM可以用来渲染UI。读者可以通过示例4-1了解添加脚本和HTML元素的具体过程。引用的所有类库都是通过访问Facebook的CDN服务实现的。

示例4-1：为React应用配置HTML文档

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Pure React Samples</title>
</head>
<body>

  <!-- Target container -->
  <div class="react-container"></div>

  <!-- React library & ReactDOM-->
  <script src="https://unpkg.com/react@15.4.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.4.2/dist/react-dom.js"></script>

  <script>

      // 此处可以添加React应用的JavaScript代码

  </script>

</body>
</html>
```

这些是在浏览器中使用React的最低要求。用户可以将自己的JavaScript代码放在某个独立文件中，不过在React脚本库加载完毕之后，必须在页面的某处载入上述代码。

虚拟DOM

HTML只是浏览器构造文档对象模型（DOM）时执行的一组简单指令。当浏览器载入HTML并渲染用户界面时，构成HTML文档的元素就是DOM元素。

假设用户需要构造一个HTML层级式的菜谱。处理这类任务的一个方案可能会和下列内容类似：

```
<section id="baked-salmon">
  <h1>Baked Salmon</h1>
  <ul class="ingredients">
      <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>
    <section class="instructions">
        <h2>Cooking Instructions</h2>
        <p>Preheat the oven to 350 degrees.</p>
        <p>Spread the olive oil around a glass baking dish.</p>
        <p>Add the salmon, garlic, and pine nuts to the dish.</p>
        <p>Bake for 15 minutes.</p>
        <p>Add the yellow squash and put back in the oven for 30 mins.</p>
        <p>Remove from oven and let cool for 15 minutes.
        Add the lettuce and serve.</p>
     </section>
</section>
```

在HTML中，层级中元素之间的关系和家族树类似。我们可以说根元素有三个子节点：一个标题（h1），一个无序的成分列表（ul），以及一个包含操作指南的段落（section）。

一般来说，网站是由独立的页面组成的。当用户导航到这些页面时，浏览器会向服务器请求载入不同的HTML文档。 AJAX技术的兴起也导致了单页应用（SPA）的出现。因为浏览器可以使用AJAX技术请求和载入一小部分数据，所以整个Web应用程序可以只使用一个页面，并且依靠JavaScript更新用户界面即可。

在一个单页应用中，浏览器初始化时会载入一个HTML文档。虽然用户也可以进行网站页面导航访问，但是他们实际上仍然停留在相同的页面上。JavaScript会根据用户的交互操作销毁或者创建一个新的用户界面。这使得用户感觉自己从一个页面跳转到了另外一个页面，但是实际上用户仍然在相同的HTML页面，JavaScript承担了大量繁重的工作。

DOM API（*https://mzl.la/2m1oQDJ*）是一组对象集合，JavaScript可以通过它们和浏览器交互并修改DOM。如果读者曾经用过`document.createElement`或者`document.appendChild`，那么对DOM API应该并不陌生。在JavaScript中更新或者修改已经渲染过的DOM相对来说容易一些。[注2]不过插入新元素的过程非常低效。[注3]这意味着如果Web开发人员可以认真地了解如何修改UI的细节，那么他们可以进一步优化应用程序的性能。

使用JavaScript高效地管理DOM元素变更可能会变得复杂而又耗时。从编码角度来看，将特定元素的子节点清空，然后重新构造它们，比将保留这些子元素并尝试高

注2： Lindsey Simon, "Minimizing Browser Reflow" （*http://bit.ly/2m1pa58*）。

注3： Steven Luscher, "Building User Interfaces with Facebook's React", Super VanJS 2013（*http://bit.ly/2m1pEs3*）。

效地更新它们要容易一些。[注4]问题在于，我们可能并没有充足的时间或者丰富的JavaScript知识，在每次构建新应用时可以高效地使用DOM API。这个问题的解决方案是React。

React脚本库的设计初衷就是帮助用户更新浏览器的DOM。我们不再需要关心和构建高性能单页应用相关的复杂性，因为这些都可以由React为用户代劳。采用React之后，我们不需要直接和DOM API打交道。取而代之的是和一个虚拟DOM交互，或者一组指令，让React构造UI或者和浏览器交互。[注5]

虚拟DOM是由React元素组成的，概念上和HTML元素类似，不过它们实际上是JavaScript对象。直接访问JavaScript对象要比访问DOM API高效的多。我们可以修改JavaScript对象，即虚拟DOM，然后React通过DOM API为用户尽可能高效地渲染这些变更。

React元素

浏览器的DOM是由DOM元素构成的。同样，React的DOM是由React元素构成的。DOM元素和React的元素看上去是一样的，不过实际上它们存在很大差别。一个React元素是对实际DOM元素应该如何表示的具体描述。换句话说，React元素表示应该如何创建浏览器DOM的一组指令。

我们使用`React.createElement`创建一个React元素来表示`h1`标题元素：

```
React.createElement("h1", null, "Baked Salmon")
```

第1个参数定义了我们希望创建的元素类型。在这种情况下，我们创建了一个`h1`标题元素。第3个参数表示元素的子节点，任意节点都可以被插入到开闭标签之间。第2个参数表示元素的特性。这个`h1`标签目前不包含任何属性。

在渲染过程中，React将会把这个元素转换成实际的DOM元素：

```
<h1>Baked Salmon</h1>
```

当元素包含属性时，它们可以通过特性进行描述。下面是一个HTML的`h1`标签的示例，它包含`id`和`data-type`两个属性：

注4：Mark Wilton-Jones, "Efficient JavaScript", Dev.Opera, November 2, 2006. (*http://opr.as/2m1f5Fr*)。

注5：React Docs, "Refs and the DOM" (*http://bit.ly/2m1faJf*)。

```
React.createElement("h1",
    {id: "recipe-0", 'data-type': "title"},
    "Baked Salmon"
)

<h1 data-reactroot id="recipe-0" data-type="title">Baked Salmon</h1>
```

属性同样可以应用到新的DOM元素上：特性可以被当作属性添加到标签中，并且子节点文本可以被当作文本嵌入元素。 用户还需要留意data-reactroot，它是用来标记React组件的根元素的（见图4-1）。

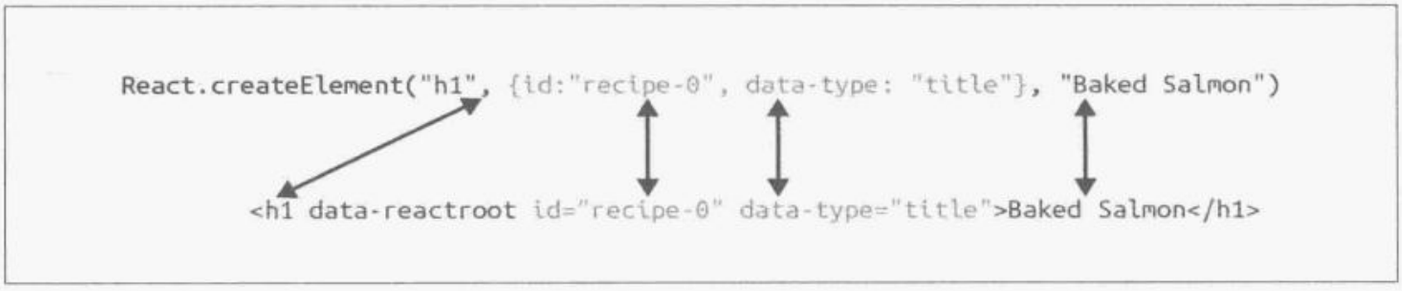


图4-1：createElement和DOM元素之间的关系

data-reactroot

data-reactroot将会一直作为React组件的根元素属性出现。在版本15之前，React的ID会被添加到每个节点中，并将之作为组件的一部分。这有助于渲染和跟踪那些需要更新的元素。现在，只有一个属性添加到了根节点中，渲染时跟踪元素的方法是基于元素的层级的。

因此，一个React元素只是一个JavaScript语法，用来告知React如何构造DOM元素。示例4-2展示了createElement实际创建的内容。

示例4-2：创建title元素时，实际创建的内容

```
{
    $$typeof: Symbol(React.element),
    "type": "h1",
    "key": null,
    "ref": null,
    "props": {"children": "Baked Salmon"},
    "_owner": null,
    "_store": {}
}
```

这是一个React元素。React用到的字段包含以下几个：_owner、_store和$$typeof。字段key和ref对于React元素来说非常重要，不过我们将会在第5章介绍它们。现在，让我们把重点放在示例4-2中的type和props属性上。

React元素的type属性是用来告知React需要创建的元素是HTML还是SVG元素。pros属性表示构建一个DOM元素所需的数据和子元素。children属性是用来将其他嵌套元素当作文本显示的。

创建元素时的注意事项

我们仔细地查看React.createElement返回的对象后，会发现其中的内容和手动编码创建元素完全不同。用户必须一直使用React.createElement或者工厂类（factory）创建React元素，工厂类的详情将会在本章结尾部分深入介绍。

ReactDOM

ReactDOM中包含在浏览器中渲染React元素必需的工具。ReactDOM中还可以找到诸如renderToString和renderToStaticMarkup方法，它们主要用于服务器端渲染。上述内容将会在第12章详细介绍。所有根据虚拟DOM生成HTML必需的工具都可以在该脚本库中找到。

我们在使用ReactDOM.render渲染一个React元素时，还能将其子节点一同渲染到DOM上。我们希望渲染的元素是通过第一个参数传递的，第二个参数代表目标节点，即渲染元素的具体位置：

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

ReactDOM.render(dish, document.getElementById('react-container'))
```

将标题元素渲染到DOM中时，将会把h1标签元素添加到id为react-container的div标签中，这个div我们已经在HTML中预先定义了。在示例4-3中，我们将在body标签内部构造这样一个div。

示例4-3：React将h1标签添加到了目标中：react-container

```
<body>
    <div id="react-container">
        <h1>Baked Salmon</h1>
    </div>
</body>
```

React中所有和DOM渲染有关的功能都移动到了ReactDOM中，因为我们也可以使用React构造原生的移动应用。浏览器只是React的运行环境之一。

这就是需要用户完成的全部工作。创建一个元素，然后将它渲染到DOM上。下一节中，我们将会详细介绍props.children的具体使用。

子节点

ReactDOM允许用户将一个独立的元素渲染到DOM上。[注6]React会使用data-reactroot对它进行标记。所有其他的React元素都可以通过嵌套的方式合成到上述独立元素中。

React是采用props.children渲染子元素的。在上一节中，我们已经将一个文本元素作为h1标签的子元素进行渲染了，并且props.children的值被设置为了“Baked Salmon”。我们还可以将其他React元素渲染为子元素，继而创建一个元素树。这也是我们使用组件树这个术语的原因。这棵树包含一个根组件，同时衍生出很多分支。

接下来看看包含若干食材成分的无序列表，如示例4-4所示。

示例 4-4：食材成分列表

```
<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>
```

在这个示例中，无序列表是根元素，它包含6个子节点。我们可以使用React.createElement来表示这个ul和它的子节点（参见示例4-5）。

示例4-5：React元素表示的无序列表

```
React.createElement(
    "ul",
    null,
    React.createElement("li", null, "1 lb Salmon"),
    React.createElement("li", null, "1 cup Pine Nuts"),
    React.createElement("li", null, "2 cups Butter Lettuce"),
    React.createElement("li", null, "1 Yellow Squash"),
    React.createElement("li", null, "1/2 cup Olive Oil"),
    React.createElement("li", null, "3 cloves of Garlic")
)
```

每个额外传递给createElement方法的参数都代表另外一个子元素。React会为这些子元素创建一个数组，然后将props.children的值添加到数组中。

如果我们仔细查看一下创建React元素后的结果，将会发现一个React元素代表一个列表项，并且将一个数组添加到了props.children中（参见示例4-6）。

注6： Rendering Elements (http://bit.ly/2nvR2vf)。

示例4-6：创建React元素之后的结果

```
{
  "type": "ul",
  "props": {
    "children": [
      { "type": "li", "props": { "children": "1 lb Salmon" } ··· },
      { "type": "li", "props": { "children": "1 cup Pine Nuts"} ··· },
      { "type": "li", "props": { "children": "2 cups Butter Lettuce" } ··· },
      { "type": "li", "props": { "children": "1 Yellow Squash"} ··· },
      { "type": "li", "props": { "children": "1/2 cup Olive Oil"} ··· },
      { "type": "li", "props": { "children": "3 cloves of Garlic"} ··· }
    ]
    ...
  }
}
```

从上述代码可知，每个列表项就是一个子节点。如前所述，我们介绍过在HTML中一个植根于section元素的完整菜谱。为了使用React创建这个菜谱，我们将会执行一系列的createElement调用，如示例4-7所示。

示例4-7：React元素树

```
React.createElement("section", {id: "baked-salmon"},
    React.createElement("h1", null, "Baked Salmon"),
    React.createElement("ul", {"className": "ingredients"},
        React.createElement("li", null, "1 lb Salmon"),
        React.createElement("li", null, "1 cup Pine Nuts"),
        React.createElement("li", null, "2 cups Butter Lettuce"),
        React.createElement("li", null, "1 Yellow Squash"),
        React.createElement("li", null, "1/2 cup Olive Oil"),
        React.createElement("li", null, "3 cloves of Garlic")
    ),
    React.createElement("section", {"className": "instructions"},
        React.createElement("h2", null, "Cooking Instructions"),
        React.createElement("p", null, "Preheat the oven to 350 degrees."),
        React.createElement("p", null,
        "Spread the olive oil around a glass baking dish."),
        React.createElement("p", null, "Add the salmon, garlic, and pine..."),
        React.createElement("p", null, "Bake for 15 minutes."),
        React.createElement("p", null, "Add the yellow squash and put..."),
        React.createElement("p", null, "Remove from oven and let cool for 15 ....")
    )
)
```

React中的className

任何在HTML中的元素都包含一个class属性，不过是使用className代替class表示该属性。因为class是JavaScript中保留关键字，所以我们必须使用className来定义HTML元素的class属性。

这个示例演示了单纯使用React时的样子。React代码最终会在浏览器中运行。虚拟DOM是一个包含单个根元素的React元素树。React元素是一组操作指令，React将根据该指令在浏览器中构建UI界面。

使用数据构造元素

使用React的主要优点是它可以将数据和UI元素有效隔离。因为React只包含JavaScript，我们可以添加JavaScript的业务逻辑来帮助我们构建React组件树。比如，食材成分可以存储在一个数组中，然后我们可以将这个数组映射到React元素上。

接下来我们将会在示例4-8中重新回顾一下无序列表的应用。

示例4-8：无序列表

```
React.createElement("ul", {"className": "ingredients"},
    React.createElement("li", null, "1 lb Salmon"),
    React.createElement("li", null, "1 cup Pine Nuts"),
    React.createElement("li", null, "2 cups Butter Lettuce"),
    React.createElement("li", null, "1 Yellow Squash"),
    React.createElement("li", null, "1/2 cup Olive Oil"),
    React.createElement("li", null, "3 cloves of Garlic")
);
```

该食材成分列表中用到的数据可以很容易地使用JavaScript的数组进行表示（参见示例4-9）。

示例4-9：列表项数组

```
var items = [
    "1 lb Salmon",
    "1 cup Pine Nuts",
    "2 cups Butter Lettuce",
    "1 Yellow Squash",
    "1/2 cup Olive Oil",
    "3 cloves of Garlic"
]
```

我们可以使用Array.map根据上述数据构造一个虚拟DOM，如示例4-10所示。

示例4-10：将一个数组映射为li元素

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

上述代码为数组中的每种成分创建了一个React元素。每个字符串将会作为列表元素子节点中的文本予以显示。每种成分的值将会作为列表项显示。

当运行这些代码时，你将会在控制台发现一个错误提示，如图4-2所示。

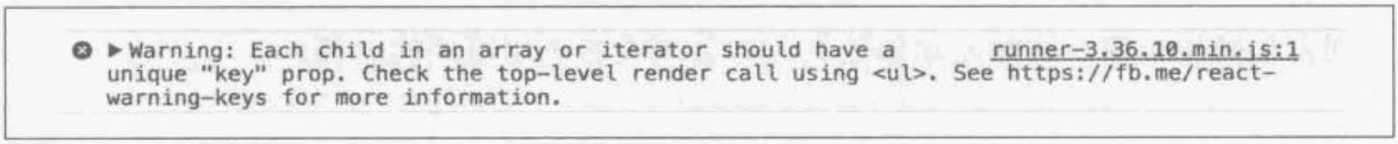


图4-2：控制台错误提示

当我们通过遍历数组构造了一个子元素列表时，React希望其中的每个元素都包含一个key属性。key属性主要是用来辅助React高效地更新DOM的。我们将会在第5章深入讨论key，以及必须这么做的原因，不过现在可以通过为每个列表项元素添加一个唯一key属性，来避免系统出现这个警告信息（参见示例4-11）。我们可以使用数组的索引作为每种成分的key属性的唯一值。

示例4-11：添加一个key属性

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

React组件

每个用户界面都是由若干零部件构成的。这里用到的菜谱示例中包含若干菜谱，每个菜谱又由若干零部件构成（见图4-3）。

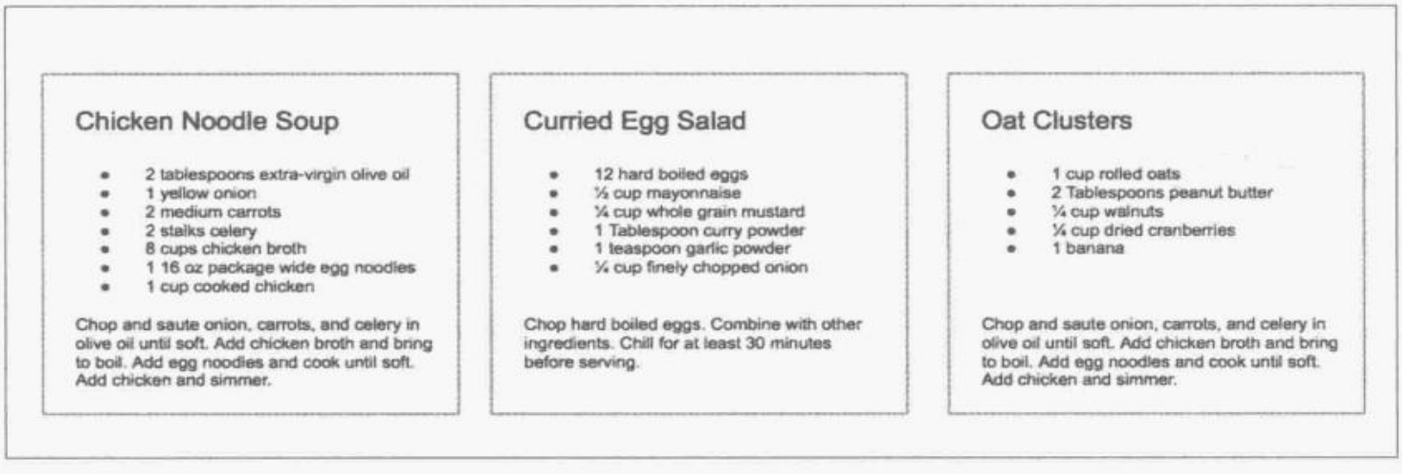


图4-3：菜谱应用

在React中，我们将这些零部件称为组件。组件允许用户对不同菜谱或者数据集复用同一DOM结构。

当考虑使用React构建一个用户界面时，可以在适当的时机将这些元素通用的部分提取出来，封装成可复用的组件。比如，图4-4中的菜谱列表元素都包含一个标题，成分列表和操作步骤。这些元素都是更大型菜谱或者应用组件的一部分。

我们可以为每个高亮部分创建一个组件：成分列表，操作步骤等。

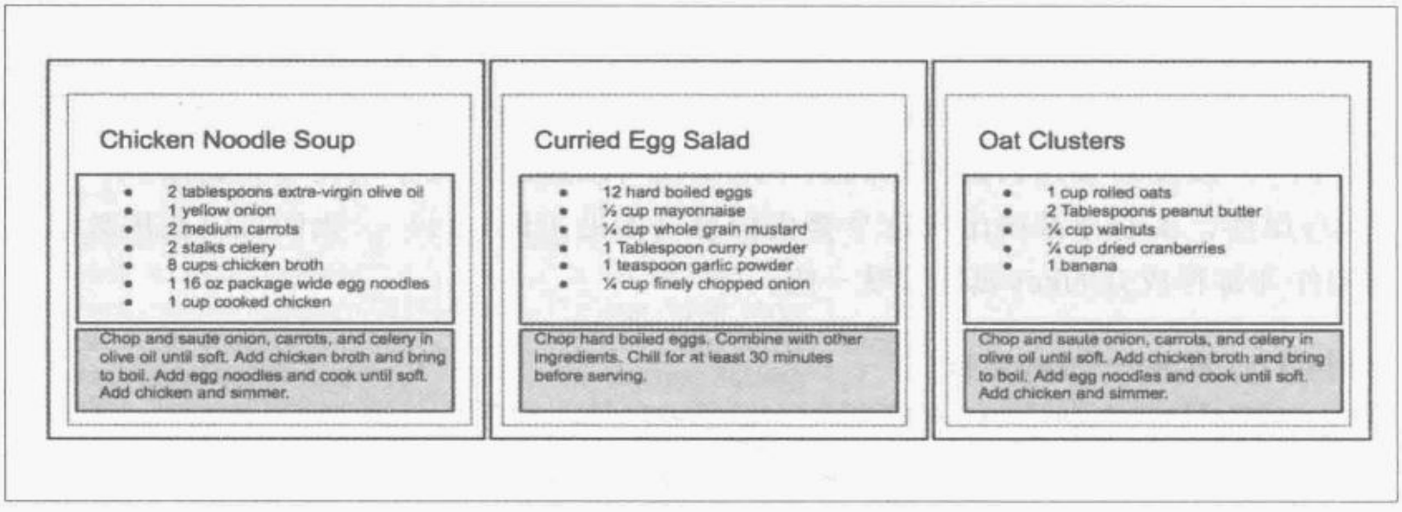


图4-4：每个组件的大致轮廓：App应用，成分列表和操作步骤

读者可以思考一下如何对上述代码进行扩展。如果希望显示一个菜谱，我们的组件结构可以满足需要。如果希望显示10000个菜谱，那么我们只需要创建该组件的若干实例即可。

接下来将会介绍创建组件的三种不同方法：createClass、ES6的类和无状态函数式组件。

React.createClass

2013年，React诞生之初，创建组件的方法只有一种，即createClass函数。

虽然后来又引入了不少创建组件的新方法，不过createClass函数在React项目中仍然非常流行。React研发团队已经表示，未来可能会弃用createClass函数。

接下来让我看看包含到每个菜谱中的食材成分列表。如示例4-12所示，我们可以使用React.createClass创建一个React组件，它返回的单个无序列表包含了一个和数组中每种成分对应的子元素列表。

示例4-12：React组件构造的食材成分列表

```
const IngredientsList = React.createClass({
  displayName: "IngredientsList",
  render() {
    return React.createElement("ul", {"className": "ingredients"},
        React.createElement("li", null, "1 lb Salmon"),
        React.createElement("li", null, "1 cup Pine Nuts"),
        React.createElement("li", null, "2 cups Butter Lettuce"),
        React.createElement("li", null, "1 Yellow Squash"),
        React.createElement("li", null, "1/2 cup Olive Oil"),
        React.createElement("li", null, "3 cloves of Garlic")
    )
  }
})

const list = React.createElement(IngredientsList, null, null)

ReactDOM.render(
  list,
  document.getElementById('react-container')
)
```

组件还允许用户使用数据构造一个可复用的UI。在render函数中，我们可以使用this关键字访问组件实例，可以通过this.props访问该实例的属性。

现在，我们使用上述组件创建了一个元素，并将其命名为IngredientsList：

```
<IngredientsList>
    <ul className="ingredients">
        <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>
</IngredientsList>
```

数据还可以作为属性传递给React组件。我们可以通过数组的形式将数据传递到列表，继而创建一个可复用的成分列表：

```
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)
        )
    )
  }
})
```

```
const items = [
    "1 lb Salmon",
    "1 cup Pine Nuts",
    "2 cups Butter Lettuce",
    "1 Yellow Squash",
    "1/2 cup Olive Oil",
    "3 cloves of Garlic"
]

ReactDOM.render(
  React.createElement(IngredientsList, {items}, null),
  document.getElementById('react-container')
)
```

现在，轮到ReactDOM出场了。数据属性items是一个包含6种成分的数组。因为我们是循环构造li标签的，还可以使用循环的索引作为它们唯一的key值：

```
<IngredientsList items=[...]>
    <ul className="ingredients">
        <li key="0">1 lb Salmon</li>
        <li key="1">1 cup Pine Nuts</li>
        <li key="2">2 cups Butter Lettuce</li>
        <li key="3">1 Yellow Squash</li>
        <li key="4">1/2 cup Olive Oil</li>
        <li key="5">3 cloves of Garlic</li>
    </ul>
</IngredientsList>
```

组件即对象。它们可以像类一样封装代码。我们可以创建一个方法，用于渲染单个列表元素，继而构造所有列表元素（参见示例4-13）。

示例4-13：使用自定义方法

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

这也是MVC架构下视图的基本理念。IngredientsList和UI有关的内容都会被封装到一个组件中，让程序的各部分各司其职。

现在可以使用我们的组件创建一个React元素了，同时可以将这些元素列表作为一个属性进行传递。注意，目前元素的类型是字符串，它就是一个组件类。

组件类即类型

当渲染一个HTML或者SVG元素时，我们使用的是字符串。当创建组件的元素时，我们使用的是组件类。这也是IngredientsList没有使用引号括起来的原因，我们可以将该类传递给createElement，因为它就是一个组件。React会创建一个该类的实例，并为用户管理它。

使用组件IngredientsList和这些数据将会把下列无序列表渲染到DOM上：

```
<ul data-react-root class="ingredients">
    <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>
```

React.Component

如第2章所述，React.Component是一个主要特性，并且兼容ES6规范，它是一个抽象类，用户可以通过它来构建新的React组件。还可以使用ES6规范中的语法创建支持继承的自定义组件类，可以使用相同的语法创建IngredientsList（参见示例4-14）。

示例4-14：使用ES6规范中的class创建IngredientsList

```
class IngredientsList extends React.Component {

  renderListItem(ingredient, i) {
    return React.createElement("li", { key: i }, ingredient)
  }

  render() {
    return React.createElement("ul", {className: "ingredients"},
        this.props.items.map(this.renderListItem)
    )
  }

}
```

无状态函数式组件

无状态函数式组件是函数而非对象，因此，它们没有“this”作用域。因为它们是简单的纯函数，所以我们应该在项目开发中尽可能地使用它们。可能有些人认为无状态函数式组件不够健壮，我们应该退而求其次使用类或者createClass方法，但是一般而言，它们对用户来说是多多益善的。

无状态函数式组件是由函数组成的，它们可以接收属性然后返回一个DOM元素。无状态函数式组件是实践函数式编程范式的好方法。用户应该努力将每个无状态函数式组件打造成一个纯函数。它们可以在不产生副作用的情况下接收属性然后返回一个DOM元素。这样可以编写出简洁的代码，并且代码库也非常容易测试。

无状态函数式组件将会确保用户的应用架构尽量简单，而且React开发团队承诺使用它们之后能够在某些方面提高性能。如果用户需要封装函数或者获得一个“this”作用域，那么你将无法使用它们。

在示例4-15中，我们将会把函数renderListItem和render合二为一，构造一个独立的函数。

示例4-15：创建一个无状态函数式组件

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

我们将使用ReactDOM.render方法渲染这个组件，其方式和渲染我们使用createClass或者ES6的类创建的组件差不多。这只是一个函数。该函数通过参数props收集数据，然后根据获得的数据为每个元素返回一个无序列表。

一种可以改进这个无状态函数式组件的方法是解构属性参数（参见示例4-16）。使用ES6规范中的解构语法，我们可以将列表属性作用域直接限定在函数内部，从而减少了点标记符号的使用。现在我们可以使用IngredientsList以同样方式渲染组件类。

示例4-16：属性参数解构

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

关键字const 与无状态函数式组件

在创建组件时，使用关键字const代替了关键字var。这是一个比较通用的做法，但不是必须的。关键字const声明函数时，会把该函数当作一个常量，同时还可以阻止用户重新定义该变量。

除了稍微简洁的语法之外，Facebook暗示将来无状态函数式组件将会比createClass或者ES6的class语法效率更高。

DOM渲染

因为我们可以通过属性参数将数据传递给组件，所以我们可以将用于创建UI的应用程序数据和业务逻辑隔离。这使得用户获得了一组独立的数据集，它们比处理文档对象模型更容易一些。当我们在这个独立数据集中做任何改动时，应用程序的State也随之发生变化。

假定将应用程序中的所有数据都存放在单个JavaScript对象中。每次用户修改这个对象后，可以将相关的变更数据作为参数传递给组件，继而对UI进行重绘。这意味着ReactDOM.render将会负担很大的工作量。

为了让React能够高效地运作，ReactDOM.render必须心灵手巧的工作。和清空或者重构整个DOM相反，ReactDOM.render会保留当前的DOM，只对该DOM做最低限度的修改。

假定我们有这样一个应用程序，使用微笑或者皱眉等表情来表示团队成员的心情。我们可以在单个JavaScript数组中表示5个成员各自的心情。

```
["smile", "smile", "frown", "smile", "frown"];
```

使用这个数组构造的UI可能和下列内容类似：

如果有急事，团队成员必须一直工作到周末，我们可以通过修改数组中的数据来反映团队成员的心情，最后的结果如下图所示：

```
["frown", "frown", "frown", "frown", "frown"];
```

我们必须修改第一个数组几个元素才能和第二个数组中全是皱眉的表情一致呢？

```
["smile", "smile", "frown", "smile", "frown"];
```

```
["frown", "frown", "frown", "frown", "frown"];
```

我们必须将第一个、第二个和第4个元素中的笑脸改成皱眉。因此，我们需要修改第一个数组中的3个元素才能和第二个数组相匹配。

现在考虑如何更新DOM来表示这些变化。一个低效的解决方案是为了将这些变化展示在UI上，移除整个DOM，然后重新构建它，如示例4-17所示。

示例4-17：从当前列表开始

```
<ul>
    <li class="smile">smile</li>
    <li class="smile">smile</li>
    <li class="frown">frown</li>
    <li class="smile">smile</li>
    <li class="frown">frown</li>
</ul>
```

这需要执行如下步骤：

1. 清空当前数据：

```
<ul>

</ul>
```

2. 开始循环访问数据，然后构造第一个列表元素：

```
<ul>
  <li class="frown">frown</li>
</ul>
```

3. 构建和添加第二个列表元素：

```
<ul>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
</ul>
```

4. 构建和追加第3个列表元素：

```
<ul>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
</ul>
```

5. 构建和追加第4个列表元素：

```
<ul>
```

```
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
</ul>
```

6. 构建和追加第5个列表元素：

```
<ul>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
</ul>
```

如果我们通过擦除和重建DOM的方式修改UI，那么我们需要插入5个新的DOM元素。向DOM中插入一个元素是代价高昂的DOM API操作之一，因为它太慢了。相比之下，更新已存在的DOM元素比插入新元素快得多。

ReactDOM.render通过保留当前DOM，适时更新需要更新的DOM元素，来达到更新UI的目的。在我们的示例中，只有3个元素发生了变化，因此ReactDOM.render只需更新这3个DOM元素即可（见图4-5）。

```
<ul>
  <li class="smile">smile</li>
  <li class="smile">smile</li>
  <li class="frown">frown</li>
  <li class="smile">smile</li>
  <li class="frown">frown</li>
</ul>
```

```
<ul>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
  <li class="frown">frown</li>
</ul>
```

图4-5：被更新的3个DOM元素

如果需要插入新的DOM元素，ReactDOM将会插入它们，不过会尝试花最小的代价完成这些插入操作（最耗资源的操作）。

这种智能化的DOM渲染对React的高效运作是非常有必要的，因为我们的应用程序的State经常会发生变动。每次我们修改了它的State之后，都必须依赖ReactDOM.render高效地完成UI重绘的工作。

工厂类

目前为止，我们创建元素的唯一方式是通过React.createElement。另外一种创建

React元素的方式是使用工厂类（Factory）。工厂类就是一种特殊的对象，它可以将实例化对象的细节封装起来。在React中，我们可以借助工厂类创建React元素的实例。

React内置的工厂类为所有HTML和SVG元素提供了支持，用户可以使用React.createFactory方法针对特定组件创建自定义工厂类。

比如，对于本章前面提到的h1元素：

```
    <h1>Baked Salmon</h1>
```

除了使用createElement方法，我们可以使用内置的工厂类创建一个React元素（参见示例4-18）。

示例4-18：使用createFactory方法创建一个h1元素

```
React.DOM.h1(null, "Baked Salmon")
```

在这种情况下，第一个参数表示元素属性，第二个参数表示子节点。我们还可以使用DOM工厂类构建一个无序列表，如示例4-19所示。

示例4-19：使用DOM工厂类构建一个无序列表

```
React.DOM.ul({"className": "ingredients"},
    React.DOM.li(null, "1 lb Salmon"),
    React.DOM.li(null, "1 cup Pine Nuts"),
    React.DOM.li(null, "2 cups Butter Lettuce"),
    React.DOM.li(null, "1 Yellow Squash"),
    React.DOM.li(null, "1/2 cup Olive Oil"),
    React.DOM.li(null, "3 cloves of Garlic")
)
```

在这种情况下，第一个参数用于传递属性，这里我们定义了className。其他的参数表示将会被添加到无序列表的子节点元素。我们还可以分离食材成分数据，使用工厂类改进之前的定义（参见示例4-20）。

示例4-20：在工厂类中使用map方法

```
var items = [
    "1 lb Salmon",
    "1 cup Pine Nuts",
    "2 cups Butter Lettuce",
    "1 Yellow Squash",
    "1/2 cup Olive Oil",
    "3 cloves of Garlic"
]

var list = React.DOM.ul(
    { className: "ingredients" },
```

```
    items.map((ingredient, key) =>
        React.DOM.li({key}, ingredient)
    )
)

ReactDOM.render(
    list,
    document.getElementById('react-container')
)
```

使用工厂类的组件

如果用户希望像函数调用那样使用组件来达到简化代码的目的，那么需要显式创建一个工厂类（参见示例4-21）。

示例4-21：为IngredientsList创建一个工厂类

```
const { render } = ReactDOM;

const IngredientsList = ({ list }) =>
    React.createElement('ul', null,
        list.map((ingredient, i) =>
            React.createElement('li', {key: i}, ingredient)
        )
    )

const Ingredients = React.createFactory(IngredientsList)

const list = [
    "1 lb Salmon",
    "1 cup Pine Nuts",
    "2 cups Butter Lettuce",
    "1 Yellow Squash",
    "1/2 cup Olive Oil",
    "3 cloves of Garlic"
]

render(
    Ingredients({list}),
    document.getElementById('react-container')
)
```

在这个示例中，我们可以使用工厂类Ingredients便捷地渲染React元素。Ingredients是一个函数，可以像DOM工厂类那样接收属性和子元素作为参数。

如果读者以前从未接触过JSX，也许会觉得使用工厂类比多次调用React.createElement方法效果更好。不过，定义React元素最简单和常用的方法是使用JSX标签。如果你曾经在React中使用过JSX，那么也许永远也不会使用工厂类。

在本章中，我们介绍了使用createElement和createFactory方法构建组件。在第5章中，我们将介绍如何使用JSX简化创建组件的过程。

第5章

React与JSX

在上一章中，我们已经知道虚拟DOM是React创建和更新用户界面时遵循的一组指令。这些指令是由JavaScript对象构成的，同时也被称为React元素。到目前为止，我们已经学习了两种方法创建React元素：使用`React.createElement`方法和使用工厂类。

为了避免冗长的输入，`React.createElement`方法又被简称为JSX，一种JavaScript的扩展，允许用户使用类似HTML的语法定义React元素。在本章中，我们将会讨论如何使用JSX构造一个包含React元素的虚拟DOM。

React元素和JSX

Facebook的React团队在发布React的同时也发布了JSX，它提供了一种简洁的语法，方便用户使用属性创建复杂的DOM树。他们还希望React项目可以像HTML和XML一样易于理解。

在JSX中，一个元素的类型是由一个标签声明的。该标签的属性就表示元素的属性。

元素的子节点可以被添加到开闭标签之间。

用户还可以添加其他JSX元素作为子节点。假定有一个无序列表，用户可以使用JSX标签为该列表添加子列表元素(参见示例5-1)。

示例5-1：JSX构造的一个无序列表

```
<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还可以和组件一起使用。只需使用类名定义组件即可。在图5-1中，我们使用JSX将数组ingredients传递给了IngredientsList的属性。

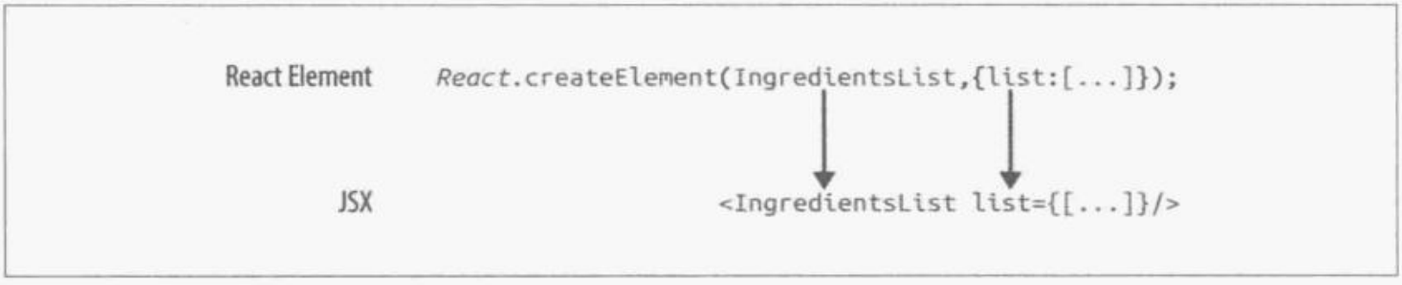


图5-1：使用JXS创建IngredientsList

当我们将数组ingredients传给这个组件时，需要使用花括号将它括起来。它也被称为JavaScript表达式，在我们将JavaScript的值传给组件参数时，必须采用这种形式。组件可以接受两种类型：一个字符串或者一个JavaScript表达式。JavaScript表达式可以包含数组、对象，甚至函数。为了引用它们，用户必须使用花括号将它们括起来。

JSX小技巧

JSX语法看上去并不陌生，大部分语法规则和HTML类似。不过，在使用JSX时，有一些注意事项是读者应该知道的。

组件嵌套

JSX允许用户将组件作为其他组件的子元素添加。比如，在IngredientsList内部，我们可以渲染名为Ingredient的其他组件多次（参见示例5-2）。

示例5-2：IngredientsList内部嵌套了三个Ingredient组件

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

className

因为class是JavaScript中保留关键字，所以使用className替代class来定义class的属性：

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

JavaScript表达式

JavaScript表达式是用花括号括起来的，其中声明的变量会进行求值并返回计算结果。比如，如果我们希望显示某个元素中title属性的值，那么可以使用JavaScript表达式插入该值。变量将会被求值并返回计算结果：

```
<h1>{this.props.title}</h1>
```

除了字符串以外的值也可以用JavaScript表达式表示：

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

求值计算

JavaScript中被添加到花括号内部的内容将会进行求值。这意味着会发生字符串连接或者数值相加这类操作。这同时也意味着出现在JavaScript表达式中的函数将会被调用。

```
<h1>{"Hello" + this.props.title}</h1>

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

function appendTitle({this.props.title}) {
    console.log(`${this.props.title} is great!`)
}
```

JSX数组映射

JSX是JavaScript代码，因此用户可以直接将JSX集成到JavaScript函数内部。比如，可以将一个数组映射为JSX元素（参见示例5-3）。

示例5-3：在JSX中使用Array.map()方法

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

JSX看起来简洁易读，不过它无法被浏览器解析。所有JSX必须转换成createElement调用或者工厂类。幸运的是，有一款堪称完美的工具可以完成这个任务：Babel。

Babel

大部分软件编程语言都允许用户编译源码。JavaScript是一门解释性的语言：浏览器会将JavaScript代码解析为文本，因此不需要对JavaScript代码进行编译。不过，并非所有浏览器都支持最新的ES6和ES7规范，而且没有浏览器支持JSX格式的语法。

因为我们希望使用采用了JavaScript新特性的JSX，所以需要一种机制将我们偏爱的源码转换成浏览器可以解析的格式。这个过程被称为转译，它也是Babel(*https://babeljs.io/*)的主要功能。

它的第一版程序名叫6to5，发布于2014年9月。6to5是一款可以将ES6语法转换成ES5规范的工具，几乎支持所有主流的浏览器。随着该项目的发展，其旨在成为支持所有ECMAScript最新特性的平台。同时它还支持将JSX转译成纯粹的React源码。该项目于2015年2月正式更名为Babel。

Facebook、Netflix、PayPal和Airbnb等知名科技公司在生产环境中都采用了Babel。此前Facebook曾经研发了一个JSX转译器作为他们的规范，不过不久之后他们就投入了Babel的怀抱。

Babel的用法有多种。最简单的应用方式是直接在HTML文件中引用一个babel-core转译器的链接，这将会转译任意类型为“text/babel”的脚本代码块。Babel会在客户端浏览器执行这些源代码之前对它们进行转译。不过这对于生产环境也许并不是最佳的解决方案，这对于JSX新手是一种很好的方式（参见示例5-4）。

示例5-4：引用babel-core

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

    <!-- React Library & React DOM -->
    <script src="https://unpkg.com/react@15.4.2/dist/react.js"></script>
    <script src="https://unpkg.com/react-dom@15.4.2/dist/react-dom.js"></script>
    <script
      src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.29/browser.js">
```

```
    </script>

    <script type="text/babel">

        //在这里编写JSX代码，或者链接其他包含JSX代码的独立JavaScript文件
    </script>

  </body>
</html>
```

Babel v5.8 是必需的

为了在浏览器中转译代码，最好使用Babel v.5.8。Babel 6.0+不能作为浏览器内部的转译器使用。

本章后续的章节中，将会介绍Babel和webpack一起搭配使用，对JavaScript文件进行静态转译。现在，使用浏览器内部的转译器已经能够满足需要了。

菜谱与JSX

React日渐风靡的原因之一是它允许用户使用优雅的代码编写Web应用。创建优雅的代码模块非常有用，它可以清楚地说明应用程序的功能。JSX为用户提供了一种漂亮、简洁的方式表示代码中的React元素，这对我们是非常有意义的，并且能够被团队的其他开发人员快速地阅读理解。JSX的不足之处在于，它不能被浏览器识别。在我们的代码可以被浏览器解析之前，需要将JSX格式转译成纯粹的React源码。

示例5-5中的数组包含两个菜谱，它们表示我们应用程序的当前State。

示例5-5：菜谱数组

```
var 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 salmon, garlic, and pine nuts to the dish.",
      "Bake for 15 minutes.",
      "Add the yellow squash and put back in the oven for 30 mins.",
```

```
      "Remove from oven and let cool for 15 minutes. 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 hot.",
      "Place the fish on the 3 tortillas.",
      "Top them with lettuce, tomatoes, and cheese."
    ]
  }
];
```

这些数据是用数组中的两个JavaScript对象表示的。每个对象包含菜谱名称、所需食材的成分列表和烹饪这道菜的具体步骤。

我们可以使用两个组件为这些菜谱创建UI：一个用于列出这些菜谱的Menu组件，以及一个用于描述每个菜谱UI的Recipe组件。我们将会把Menu组件渲染到DOM上。将会通过一个名为recipes的属性将数据传递给Menu组件（参见示例5-6）。

示例5-6：菜谱应用代码架构

```
// 菜谱对象数组
var data = [ ... ];

//单个菜谱的无状态函数式组件
const Recipe = (props) => (
  ...
)

//菜谱菜单的无状态函数式组件
const Menu = (props) => (
  ...
)

// 调用ReactDOM.render将菜单渲染到当前DOM上
ReactDOM.render(
    <Menu recipes={data} title="Delicious Recipes" />,
    document.getElementById("react-container")
)
```

ES6 支持

我们还将在这个文件中使用ES6语法。当将我们的代码从JSX转译成纯粹的React源码时，Babel将会把ES6规范的代码转换成符合ES5规范的代码，以便所有浏览器都可以解析。所有采用的ES6新特性已经在第2章详细介绍过。

Menu组件中的React元素是用JSX表示的（参见示例5-7）。一切都包含在了一个article元素中。其中包括一个标题元素，一个h1元素，以及一个用于为菜单描述DOM的div.recipes元素。title的属性值将会以文本的形式显示在h1元素中。

示例5-7：Menu组件结构

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

在div.recipes元素中，我们为每个菜谱添加了一个组件（参见示例5-8）。

示例5-8：菜谱数据映射

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

为了显示div.recipes元素中的菜谱，我们可在花括号中添加一个JavaScript表达式来返回子元素的数组。可以在数组props.recipes上调用map函数，继而返回数组中每个对象的组件。如前所述，每个菜谱包含名称，食材成分和烹饪指令（步骤）。我们将需要把这些数据作为属性传递给每个菜谱。当然还应该使用key属性对每个元素进行唯一性标识。

使用JSX扩展运算符可以对我们的代码进行改进。JSX扩展运算符的工作机制和第2章介绍的对象扩展运算符类似。它会把recipe对象的每个字段当作Recipe组件的属性进行添加。示例5-9中的代码语法可以获得相同的结果。

示例5-9：功能优化：JSX扩展运算符

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

另外一个可以使用ES6规范改进我们的Menu组件的地方是props参数。我们可以使用对象解构将变量的作用域限定在该函数内部。这使得用户可以直接访问变量title和recipes，无需通过在上述变量前加上props前缀来访问它们（参见示例5-10）。

示例5-10：Menu组件重构

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

现在我们将会为单个菜谱编写组件（参见示例5-11）。

示例5-11：完成Recipe组件

```
const Recipe = ({ name, ingredients, steps }) =>
    <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>
```

这个组件也是一个无状态函数式组件。每个菜谱都包含一个用字符串表示的名称，一个食材成分的对象数组，以及一个烹饪步骤的字符串数组。使用ES6规范中的对象解构，我们可以让该组件通过名称将这些字段的作用域本地化，以便用户可以直接访问它们，而不是必须通过props.name、props.ingredients和props.steps这样的形式才可以访问。

我们可以看到第一个JavaScript表达式被用来给根节点section元素设置id属性。它会把菜谱的名称转换成全是小写字母的字符串，然后使用连字符（-）替换其中的所有空格。结果是字符串“Baked Salmon”在被用作我们的UI元素id之前，被转换成了“baked-salmon”（当前，如果我们的菜谱名称是“Boston Baked Beans”，那么它将被转换成“boston-baked-beans”）。name的值还会在h1标签中作为文本节点显示。

在无序列表内部，一个JavaScript表达式将每种食材成分和一个li元素关联，以便显示食材的名称。在烹饪步骤部分，我们可以看到同样的模式，用于返回显示每个步骤的段落元素。这些map函数会返回一个子元素的数组。该应用的完整代码可以参见示例5-12。

示例5-12：菜谱应用的完整代码

```
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 salmon, garlic, and pine nuts to the dish.",
      "Bake for 15 minutes.",
      "Add the yellow squash and put back in the oven for 30 mins.",
      "Remove from oven and let cool for 15 minutes. 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 hot.",
      "Place the fish on the 3 tortillas.",
      "Top them with lettuce, tomatoes, and cheese."
    ]
  }
]
```

```
const Recipe = ({ name, ingredients, steps }) =>
    <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>

const Menu = ({ title, recipes }) =>
    <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("react-container")
)
```

当在浏览器中执行上述代码后，React将会使用图5-2所示的菜谱数据和相应的操作指令构造一个显示菜谱的界面。

如果你使用的是Google Chrome浏览器，并且安装了React 开发者工具扩展，那么可以查看一下虚拟DOM的当前状态。为此，请在上述浏览器中打开开发者工具，选择React项（见图5-3）。

Delicious Recipes

Baked Salmon

- Salmon
- Pine Nuts
- Butter Lettuce
- Yellow Squash
- Olive Oil
- Garlic

Cooking Instructions

Preheat the oven to 350 degrees.

Spread the olive oil around a glass baking dish.

Add the salmon, garlic, and pine nuts to the dish.

Bake for 15 minutes.

Add the yellow squash and put back in the oven for 30 mins.

Remove from oven and let cool for 15 minutes. Add the lettuce and serve.

Fish Tacos

- Whitefish
- Cheese
- Iceberg Lettuce
- Tomatoes
- Tortillas

Cooking Instructions

Cook the fish on the grill until hot.

Place the fish on the 3 tortillas.

Top them with lettuce, tomatoes, and cheese.

图5-2：精美的菜谱页面

图5-3：React开发者工具中虚拟DOM的状态

这里可以看到我们的Menu组件及其子元素。数据数组中包含两个菜谱对象，我们有两个Recipe元素。每个Recipe元素的属性包括菜谱名称、食材成分和烹饪步骤。

虚拟DOM是基于应用程序的State数据构造的，该数据会作为属性传递给Menu组件。如果我们希望修改数组recipes并重绘Menu组件，React将会以尽可能高效的方式修改这个DOM。

Babel的 presets模块

Babel 6中的presets模块中的转译机制有重大变化。它需要工程师显式声明转译过程中采用哪些模块。其目的是为了让程序更加模块化，同时允许开发人员决定哪些语法应该被转译。插件可以分为几个类别，所有这些都可以根据应用程序的需要进行选择。用户根据偏好选择即可：

babel-preset-es2015

将ES2015或者ES6转译成ES5。

babel-preset-es2016

将ES2016转译成ES2015。

babel-preset-es2017

将属于ES2017规范的特性转译成ES2017。

`babel-preset-env`

对属于ES2015、ES201和ES2017规范的内容进行转译。包含了前面三个预设模块的功能。

`babel-preset-react`

将JSX格式转译成React.createElement调用。

当某个新特性被纳入ECMAScript规范时，它会经历阶段0，即Strawman（新提出的，带实验性质的）到第4阶段的成功（被接受为标准的一部分）。Babel为这4个阶段也提供了presets模块，因此用户可以根据需要选择相应的阶段presets模块：

- `babel-preset-stage-0`：实验性提议。
- `babel-preset-stage-1`：建议。
- `babel-preset-stage-2`：草稿。
- `babel-preset-stage-3`：候选。

Webpack简介

一旦我们在实际工作中采用了React，那么就会有很多问题需要考虑：如何处理JSX和ES6+的代码转译？如何管理依赖项？如果优化我们的图片和CSS文件？

为了解决这些问题诞生了各式各样的工具，其中包括Browserify、Gulp和Grunt。基于其自身特性，并且被很多大公司所采用，webpack已经成为了打包CommonJS模块的主流工具之一（CommonJS的详情可以参考第2章）。

webpack是一个模块绑定器。一个模块绑定器可以兼容多种不同类型的文件（JavaScript, LESS, CSS, JSX, ES6等），并且可以将它们转换成单个文件。模块化绑定的两个主要优点是模块化和网络性能。

模块化使得用户可以将源代码分解成易于使用的部分或模块，特别是对于团队开发来说非常有用。

网络性能的优点在于在浏览器中只需载入依赖即可，即bundle。每个脚本标签都会创建HTTP请求，并且每个HTTP请求都会有延迟。将所有依赖项打包成单个文件后，只需发送一次HTTP请求，就能加载所有必须的资源，从而避免了额外的网络延迟。

除了代码转译，webpack还可以处理如下任务：

代码拆分

将代码拆分成不同的片段，使得用户可以按需加载它们。有时这种操作也被称为汇总（rollup）或分层，其目标是为不同页面或者设备的需要对代码进行拆分打包。

代码压缩

删除代码中的空格，换行符，冗长的变量名，以及不必要的代码，来达到减小文件尺寸的目的。

特性标记

将代码发送到一个或者多个系统，测试代码的兼容性。

热替换（HMR）

监控源代码的变更，实时更新发生变更的模块。

webpack加载器

加载器是一个函数，它主要的作用是帮助用户在构建过程中处理代码转换的问题。如果我们的应用程序采用了ES6规范，JSX、CoffeeScript，以及其他不能被浏览器自动解析的语言，那么我们将在文件*webpack.config.js*中声明必要的加载器来完成代码的语法转换工作，从而达到让代码可以被浏览器自动解析的目的。

webpack包含大量的加载器，并且可以分为几类。最常用的加载器是将某个语言的一种方言转换成另外一种。比如ES6规范和React代码是通过其内置的加载器`babel-loader`进行转译的。我们声明Babel将要处理的文件类型，然后其他工作就交给webpack完成即可。

另外一种比较常用的加载器是和样式表有关的。`css-loader`会查找扩展名为*.scss*的文件，然后将它们转译成CSS。`css-loader`还可以用来在用户的bundle中引用CSS模块。所有的CSS代码将会打包成JavaScript，当打包文件中引用了JavaScript文件时，系统会自动添加CSS代码。这样就不需要使用link元素引用样式表了。

如果读者对加载器的不同选项感兴趣，可以查阅与之相关的完整列表（*https://webpack.js.org/concepts/loaders/*）。

使用webpack构建菜谱应用

本章前面构建的菜谱应用存在一些不足之处，webpack可以帮助我们弥补这些不足。

使用webpack这类工具可以静态构建客户端的JavaScript代码，这使得我们可以方便地进行团队协作，共同开发大型的Web应用。通过集成webpack模块打包工具，我们可以获得以下好处：

模块化

为了导出模块可以采用CommonJS模块模式，这样可以方便应用程序其他部分导入或者引用模块，从而使得我们的源代码更易于使用。它允许开发团队通过创建各自独立的文件进行协同工作，在交付产品之前可以将这些独立的文件整合成单个文件。

合成

通过模块，我们可以构建小型、简单、可复用的React组件，然后可以高效地将它们合成到应用程序中。较小的组件也更容易理解、测试和复用。在需要对应用程序进行功能扩展时，它们也更容易替换。

效率

将应用程序的所有模块和依赖项打包成单个客户端软件包将会减少应用程序的载入时间，因为每个HTTP请求都存在网络延迟。将所有文件打包成单个文件意味着客户端只需发送一次请求。打包文件中经过压缩的代码也能减少加载时间。

一致性

因为webpack将会把JSX转译成React源码，将ES6甚至ES7规范的代码转换成普通的JavaScript代码，所以我们今天就可以使用JavaScript将来才会发布的新特性。Babel支持的未来ES规范的范围很广，这意味用户不需要为浏览器是否能够解析代码而操心。它使得开发人员能够与时俱进，总是可以使用最前沿的JavaScript语法特性。

将组件拆分成模块

菜谱应用采用了webpack和Babel等工具之后，用户可以将代码拆分成可以使用ES6规范的模块。接下来让我们看看菜谱应用的无状态函数式组件（参见示例5-13）。

示例5-13：当前的Recipe组件

```
const Recipe = ({ name, ingredients, steps }) =>
    <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>
```

这个组件功能更强一些。我们可以显示菜谱的名称，构造一个食材成分的无序列表，并使用专属于每个步骤的段落元素显示烹饪步骤。

让Recipe组件更偏重于函数式编程的方法是将它拆分成更小、高内聚的无状态函数式组件并将它们合成到一起。我们可以从构造烹饪步骤自己的无状态函数式组件入手，在一个独立文件中创建一个模块，供其他任意烹饪步骤调用（参见示例5-14）。

示例5-14：Instructions（烹饪步骤）组件

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

export default Instructions
```

这里我们创建了一个新的组件叫Instructions。我们将会把烹饪步骤的名称和步骤编号传递给该组件。这使得我们可以在“Cooking Instructions”“Baking Instructions”“Prep Instructions”和“Pre-cook Checklist”等任意包含编号的烹饪过程复用这个组件。

现在考虑一下食材成分。在Recipe组件中，我们只显示了食材名称，但是数据中的每种食材成分还包含数量和尺寸值。我们可以创建一个无状态的函数式组件表示单个成分（参见示例5-15）。

示例5-15：Ingredient（食材成分）组件

```
const Ingredient = ({ amount, measurement, name }) =>
    <li>
        <span className="amount">{amount}</span>
        <span className="measurement">{measurement}</span>
        <span className="name">{name}</span>
    </li>
export default Ingredient
```

这里我们假定每种食材成分只包含一种数量、尺寸和名称。我们可以从props对象中将每种成分的数据解析出来，然后在单个span元素中独立显示它们。

使用Ingredient组件后，我们随时可以构造一个显示成分列表的IngredientsList组件（参见示例5-16）。

示例5-16：使用Ingredient组件构造IngredientsList组件

```
import Ingredient from './Ingredient'

const IngredientsList = ({ list }) =>
    <ul className="ingredients">
        {list.map((ingredient, i) =>
            <Ingredient key={i} {...ingredient} />
        )}
    </ul>

export default IngredientsList
```

在这个文件中，我们首先导入了Ingredient组件，因为我们会在显示每种食材成分时用到它。食材成分数据是通过一个名为list的数组传递给该组件的。list数组中每种成分将会映射为一个Ingredient组件。JSX扩展运算符还可以作为参数给Ingredient组件传递数据。

使用扩展运算符之后：

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

这是另外一种表示方法：

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

所以，使用相应字段设定一种食材成分：

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

我们可以得到：

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

现在，我们已经有了食材成分和逐步烹饪的组件，然后我们就可以使用这些组件合成菜谱组件（参见示例5-17）。

示例5-17：重构后的Recipe组件

```
import IngredientsList from './IngredientsList'
import Instructions from './Instructions'

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

export default Recipe
```

首先，我们导入了将会用到的组件IngredientsList和Instructions。现在我们就可以使用它们创建Recipe组件了。和将一堆复杂的代码聚集在一处构建Recipe实体相反，我们通过更具声明式编程风格的方式，通过将小型组件合成来构造Recipe组件。其代码不仅优雅简单，读起来也赏心悦目。它告知我们，菜谱应该显示菜谱名称、食材成分列表，以及一些烹饪步骤。我们已经将显示食材成分和烹饪步骤的工程封装成了更小、更简单的组件。

采用CommonJS规范的模块化方法中，Menu组件看起来非常简单。主要差异在于它拥有专属于自己的文件，需要导入必须的模块，同时还要导出其自身（参见示例5-18）。

示例5-18：优化后的Menu组件

```
import Recipe from './Recipe'

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

export default Menu
```

我们仍然需要使用ReactDOM渲染该Menu组件。我们还需要一个*index.js*文件，不过它看起来与众不同（参见示例5-19）。

示例5-19：完成后的index.js文件

```
import React from 'react'
import { render } from 'react-dom'
import Menu from './components/Menu'
import data from './data/recipes'

window.React = React

render(
    <Menu recipes={data} />,
    document.getElementById("react-container")
)
```

前四行语句导入的模块是确保我们的应用能够正常运作所必需的。和使用script标签导入react和react-dom相反，我们是使用import语句将它们导入的，这样做的目的是方便webpack将它们添加到bundle打包文件中。我们还需要Menu组件，以及一个已经被移动到独立模块中的样本数据数组。 它仍然包含两个菜谱：Baked Salmon和Fish Tacos。

所有导入的变量对于index.js文件来说都是本地变量。将React赋值给window.React的含义是让React脚本库在浏览器中可以全局访问。这样一来所有对React.createElement方法的调用都是可以正常工作的。

当渲染Menu组件时，我们可以通过参数属性将菜谱数据的数组传递给该组件。这个单一的ReactDOM.render方法调用将会执行并渲染我们的Menu组件。

现在我们已经将代码拆分成独立的模块和文件了，接下来我们将使用webpack构建一个静态构建过程，最终将会把所有文件整合打包成单个文件。

安装webpack依赖项

为了使用webpack创建一个静态构建过程，我们将需要安装一些组件。所有与之相关的依赖项都可以通过npm安装。首先，我们可能需要对webpack进行全局安装，以便我们可以随时随地执行webpack命令：

```
sudo npm install -g webpack
```

webpack可以和Babel搭配使用，将我们的代码从JSX和ES6规范转换成浏览器可以解析的JavaScript代码。我们将会预置几个加载器来完成这个任务：

```
npm install babel-core babel-loader babel-preset-env babel-preset-react
babel-preset-stage-0 --save-dev
```

我们的应用程序采用了React和ReactDOM脚本库。之前我们已经使用script标签引入了这些脚本。现在我们为了让webpakc可以将它们添加到bundle打包文件中，需要在本地安装React和ReactDOM的依赖项：

```
npm install react react-dom --save
```

上述操作后，react和react-dom必需的脚本会被添加到*./node_modules*文件夹下。现在我们已经做好了配置webpack静态构建过程之前的所有准备工作。

配置webpack

为了让模块化的菜谱应用程序可以正常运行，需要告知webpack如何将我们的源代码打包成单个文件。我们可以通过配置文件达到该目的，webpack的默认配置文件始终是*webpack.config.js*。

菜谱应用的启动文件是*index.js*。 它导入了React、ReactDOM和*Menu.js*文件。这也是我们希望在浏览器中首先执行的内容。一旦webpack找到了一个import语句，它将会在文件系统中查找与之相关的模块，然后将它添加到bundle打包文件中。*Index.js*导入了*Menu.js*,*Menu.js*导入了*Recipe.js*,*Recipe.js*导入了*Instructions.js*和*IngredientsList.js*，*IngredientsList.js*导入了*Ingredient.js*。webpack将会依次导入这些必需的模块到我们的bundle打包文件中。

ES6的import语句

我们使用了ES6规范下的import语句，但是目前大部分浏览器或者Node.js都没有对它提供支持。ES6的import语句可以正常工作的原因是，Babel会将它们转换成require('module/path')的形式，即我们的最终代码。require函数式是CommonJS规范中比较常见的加载模块方式。

因为webpack会将项目代码打包成bundle文件，我们需要告知webpack将JSX格式转换成纯粹的React元素。我们还需要将任何ES6规范的代码转换成符合ES5规范的代码。我们的构建过程最初包含三个步骤（见图5-4）。

*webpack.config.js*文件只是另外一个导出JavaScript对象的模块，它描述了webpack将会执行的动作。这个文件（参见示例5-20）将会保存在项目根目录下，和*index.js*文件并列。

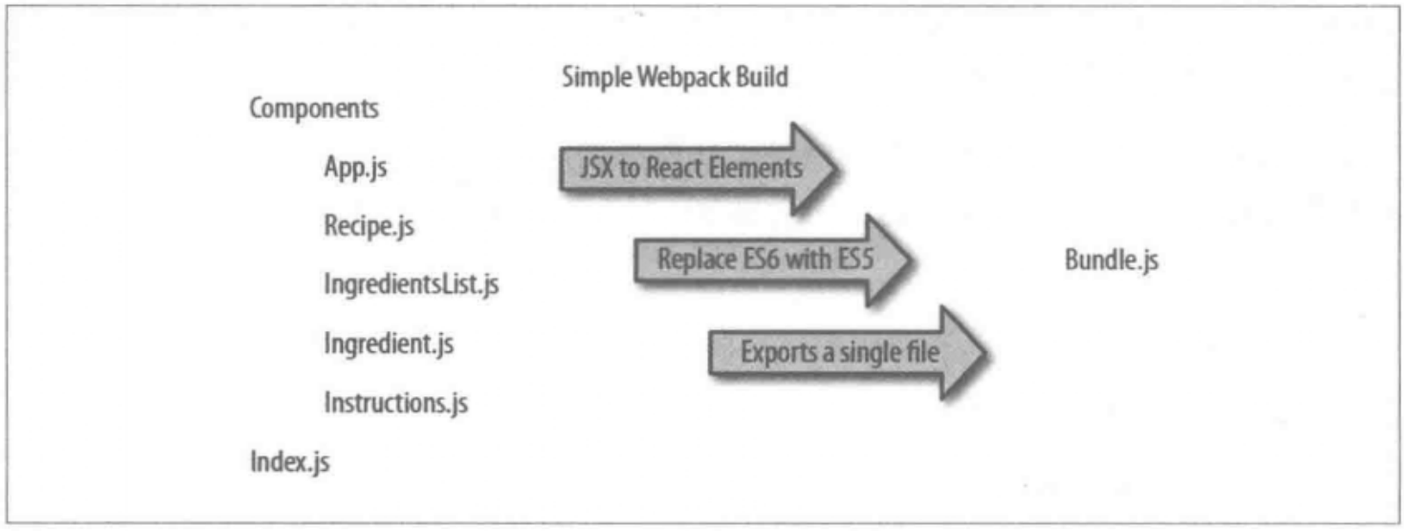


图5-4：菜谱应用的构建过程

示例5-20：webpack.config.js

```
module.exports = {
  entry: "./src/index.js",
   output: {
       path: "dist/assets",
       filename: "bundle.js"
   },
   module: {
       rules: [
           {
              test: /\.js$/,
              exclude: /(node_modules)/,
              loader: ['babel-loader'],
              query: {
                  presets: ['env', 'stage-0', 'react']
              }
           }
        ]
    }
}
```

首先，我们告知webpack客户端实体文件是*./src/index.js*。它会根据该文件中`import`语句的声明依次自动构建相关的依赖项。然后，我们声明了希望打包的JavaScript文件的目标是*./dist/assets/bundle.js*。这也是webpack存放最终打包JavaScript文件的位置。

webpack接下来的一组指令包括执行特定模块的加载器列表。`rules`字段是一个数组，因为用户可以使用webpack集成多种加载器。在本示例中，我们只集成了`Babel`。

每个加载器就是一个JavaScript对象。`test`字段是一个正则表达式，用于匹配加载器将要访问的每个模块的路径。在这种情况下，我们正在所有导入的JavaScript文件上执行加载器`babel-loader`，希望可以在`node_modules`文件夹中找到它们。当运行

babel-loader时，它会使用ES2015 (ES6)和React的预定设置，将任何ES6或者JSX语法转换成大部分浏览器都可以识别的JavaScript代码。

webpack是静态执行的。通常在应用程序被部署到服务器之前已经对所有资源文件进行打包处理了。因为用户在全局范围安装了webpack，所以可以在命令行中执行如下命令：

```
$ webpack
   Time: 1727ms
   Asset     Size  Chunks             Chunk Names
   bundle.js  693 kB       0  [emitted]  main
     + 169 hidden modules
```

webpack既可能成功地创建一个bundle文件，也可能因为某些原因无法创建该bundle文件，并向用户报告错误信息。大部分错误都和导入引用被破坏有关。当调试webpack错误时，需要认真地检查import语句中文件名和文件路径是否正确。

加载bundle

现在已经有了一个bundle文件，接下来该怎么做呢？我们需要把bundle文件导入dist文件夹。这个文件夹中包含所有用户希望在Web服务器上运行的文件。*dist*文件夹是放置首页文件*index.html*（参见示例5-21）的地方。此文件需要包含一个目标div元素，用来承载React菜单组件。它还需要一个单独的script标签，用于加载经过打包的bundle文件。

示例5-21：index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>React Recipes App</title>
</head>
<body>
  <div id="react-container"></div>
  <script src="assets/bundle.js"></script>
</body>
</html>
```

这是用户应用的主页。它会发送一个HTTP请求载入单个文件加载所必需的资源：*bundle.js*。用户需要部署这些文件到Web服务器上，或者构建一个Web服务应用，比如Node.js或者Ruby on Rails，为公众提供上述文件的访问服务。

源代码映射

将源代码打包成一个单独的文件后，可能会对开发人员在浏览器中调试应用程序带来一些麻烦。我们可以通过提供源代码映射来解决这个问题。源代码映射文件就是一个描述已打包的bundle文件和原来的源代码文件一一对应关系的文件。通过webpack，需要我们做的所有工作就是在*webpack.config.js*文件中添加几行代码（参见示例5-22）。

示例5-22：webpack.config.js中的源代码映射

```
module.exports = {
    entry: "./src/index.js",
    output: {
        path: "dist/assets",
        filename: "bundle.js",
        sourceMapFilename: 'bundle.map'
    },
    devtool: '#source-map',
    module: {
        rules: [
            {
                test: /\.js$/,
                exclude: /(node_modules)/,
                loader: ['babel-loader'],
                query: {
                    presets: ['env', 'stage-0', 'react']
                }
            }
        ]
    }
}
```

将devtool属性值设置为'#source-map'是告知webpack，用户希望使用源代码映射功能。sourceMapFilename属性是必需的。在目标依赖项后面为用户的源代码映射文件命名是一个非常好的习惯。webpack将会在导出过程中将bundle文件和源代码文件映射关联。

当用户下次执行webpack时，将会发现系统最终生成了两个文件，并将它们添加到了assets文件夹下，即原生的*bundle.js*文件，以及*bundle.map*文件。

源代码映射文件将会使得开发者可以使用原来的源代码文件调试应用程序。在浏览器的开发者工具中的源代码选项中，用户将会发现一个名为*webpack://*的文件夹。在这个文件夹中，将会找到bundle文件中所有的源代码文件（见图5-5）。

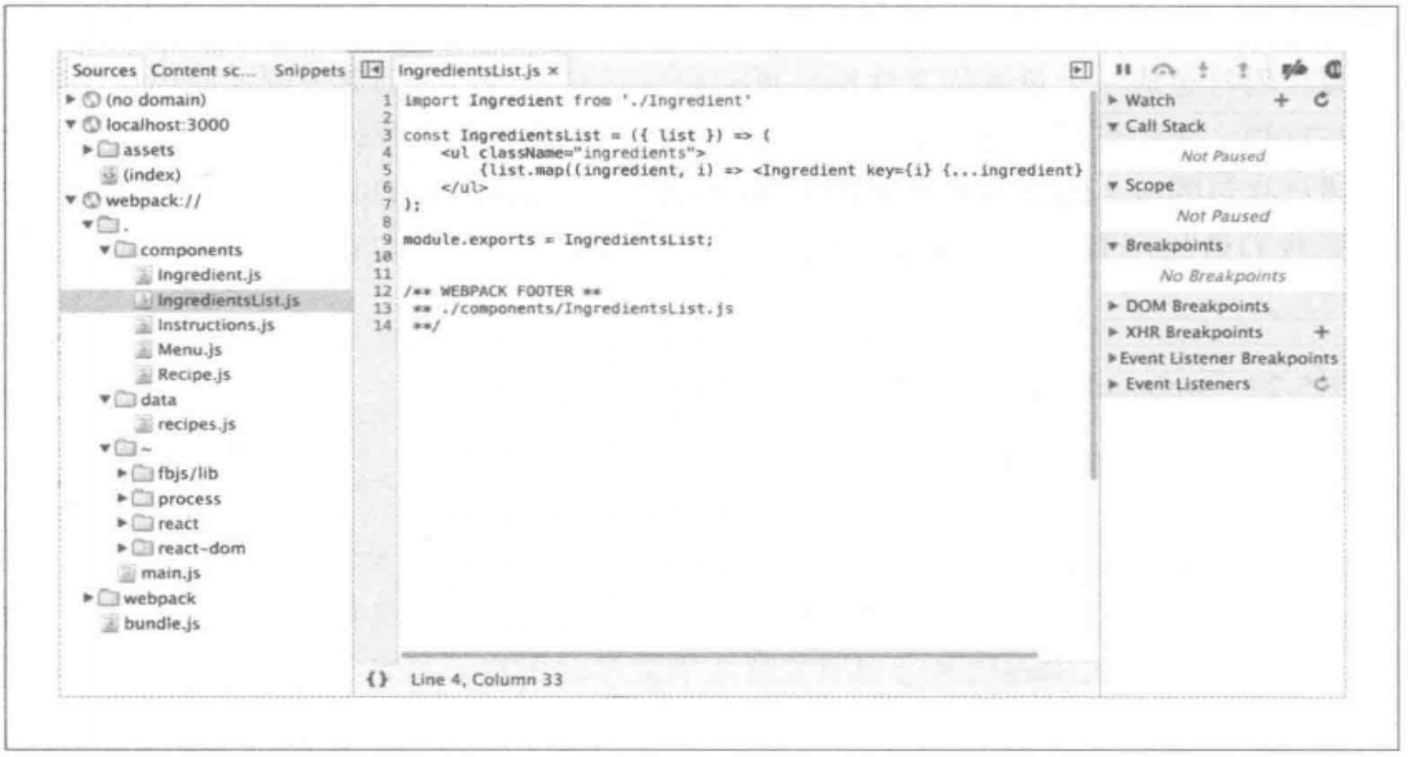


图5-5：Chrome开发者工具中的源码查看面板

用户可以使用浏览器的单步调试器对这些文件进行调试。单击其中任何一行会添加一个断点。刷新浏览器后，当遇到源代码中的任意断点时，将会暂停执行JavaScript代码。用户可以在Scope面板中查看本地变量，或者在Watch面板中添加对某个变量的监视。

优化bundle文件

输出的bundle文件仍然是一个简单的文本文件，因此我们可以通过缩减文本的数量来减小文件的尺寸，继而使得它能够以更快的速度响应HTTP请求。有助于减小文件尺寸的措施包括移除所有空格、简化过长的变量名，以及删除解析器永远不会解析的代码行。使用这些技巧减小JavaScript代码的尺寸通常被称为代码压缩或者代码混淆。

webpack内置了一款可以用来压缩bundle文件的工具，为了使用它，用户必须先在本地安装webpack：

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

我们可以使用webpack插件为构建过程添加额外的步骤。在本示例中，我们将会为构建过程添加一个步骤，来对我们的bundle文件进行压缩，这一步骤会明显减少文件的尺寸（参见示例5-23）。

示例5-23：在webpack.config.js中使用Uglify插件

```
var webpack = require("webpack");

module.exports = {
    entry: "./src/index.js",
    output: {
        path: "dist/assets", filename: "bundle.js",
        sourceMapFilename: 'bundle.map'
    },
    devtool: '#source-map',
    module: {
        rules: [
            {
                test: /\.js$/,
                exclude: /(node_modules)/,
                loader: ['babel-loader'],
                query: {
                    presets: ['env', 'stage-0', 'react']
                }
            }
        ]
    },
    plugins: [
        new webpack.optimize.UglifyJsPlugin({
            sourceMap: true,
            warnings: false,
            mangle: true
        })
    ]
}
```

为了使用Uglify插件，我们需要用到webpack，这也是我们需要在本地安装webpack的原因。

UglifyJsPlugin是一个函数，它会从传递给它的参数中获得操作指令。我们的源代码一经压缩，它们将会变得无法识别。我们将会需要一个源代码映射文件，这也是我们将sourceMap属性设置为true的原因。将warnings的属性值设置为false将会移除导出的bundle文件中的任意控制台警告信息。将代码混淆意味着我们将会把比较长的变量名，比如recipes或者ingredients用单个字母表示。

下次用户运行webpack时，将会发现打包输出的bundle文件的尺寸明显减少了，并且也无法识别了。引用一个源代码映射文件后，即使打包后的bundle文件已经压缩过了，用户仍然可以根据原来的源代码文件对程序进行调试。

打包CSS

webpack另外一个非常有用的特性是，它可以将CSS打包成与打包过的JavaScript文件类似的文件。这使得用户可以下载包含应用程序所需的所有CSS和JavaScript代码的单个文件。

CSS可以使用import语句引入bundle打包文件。这些语句告知webpack使用JavaScript模块将相关的CSS文件打包：

```
import Recipe from './Recipe'
import '../../stylesheets/Menu.css'

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

export default Menu
```

为了在webpack配置中实现对CSS文件的打包，用户需要安装相关的加载器：

```
npm install style-loader css-loader postcss-loader --save-dev
```

最后，用户还必须在webpack配置中声明这些加载器：

```
rules: [
  {
    test: /\.js$/,
    exclude: /(node_modules)/,
    loader: ['babel-loader'],
    query: {
      presets: ['env', 'stage-0', 'react']
    }
  },
  {
              test: /\.css$/,
              use: ['style-loader','css-loader', {
                  loader: 'postcss-loader',
                  options: {
                    plugins: () => [require('autoprefixer')]
                  }}]
  }
]
```

使用webpack打包CSS文件会导致你的网站加载更快，减少浏览器为所需资源创建的请求数。

create-react-app

如Facebook开发团队在他们的博文中所述，“React生态系统已经和工具的爆炸性增长息息相关了”。[注1]为此，React团队推出了一个可以自动生成React项目的命令行工具，名为`create-react-app`（*http://bit.ly/2mtQwNC*）。它的诞生灵感来自Ember CLI项目（*https://ember-cli.com/*），并且它使得开发人员在不必手工配置webpack、Babel、ESLint，以及其他相关工具的情况下，快速启动React项目。

为了使用`create-react-app`，我们首先需要全局安装与之相关的软件包：

```
npm install -g create-react-app
```

然后使用该命令，并指定新建应用程序的目标文件夹名称：

```
create-react-app my-react-project
```

这将在上述目录中创建一个React项目，并包含三个依赖项：React、ReactDOM和`react-scripts`。`react-scripts`也是由Facebook创建的，并且也是奇迹发生的地方。其中安装了Babel、ESLint、webpack等工具，因此用户就无需手动配置它们。在生成的项目文件夹中，还会找到一个包含*App.js*文件的*src*文件夹。在这里，用户可以编辑root组件，以及导入其他组件文件。

在*my-react-project*文件夹中，用户可以执行`npm start`命令，如果有必要的话，还可以执行`yarn start`命令。

用户可以使用`npm test`或者`yarn test`执行测试。这会以交互模式执行项目中的所有测试文件。

这还会在端口3000启动用户的应用程序。用户还可以执行`npm run build`命令，如果使用的是yarn，可以执行`yarn build`命令。

这将会创建一个准备上线发布的bundle文件，并且其中的代码已经经过转译和压缩。

`create-react-app`是一款对初学者和资深React开发者都非常好的工具。随着该工具的不断发展，它的功能会越来越丰富。因此用户务必留意它在GitHub上的最新资讯。

注1：Dan Abramov, “Create Apps with No Configuration”, React Blog, July 22, 2016（*http://bit.ly/2ndUXzR*）。

第6章

Props、State和组件树

在上一章中，我们探讨了如何创建组件。其重点是如何通过合成React组件构建一个用户界面。本章介绍的技术将有助于读者更好地管理数据，以及减少调试应用程序的时间。

组件树内部数据处理是React的主要优点之一。在React组件中处理数据时，有不少技术可供用户选择，长期来看，这可以大幅度减轻用户的开发工作量。如果能够从单一数据源管理数据，并且基于上述数据构建UI,那么我们的应用程序将会更容易理解和扩展。

属性验证

JavaScript是一种弱类型的语言，这意味着可以修改变量值的数据类型。比如，用户可以定义某个JavaScript变量为字符串类型，然后将它的值改为数组，JavaScript并不会对此操作有什么异议。低效地管理我们的变量类型可能会导致花费大量时间在调试应用程序上。

React组件提供了一种声明和验证属性类型的方法。使用这一特性将会大幅度减少调试应用程序的时间。提供不正确的属性类型将触发警告，这有助于我们查找那些不易察觉的Bug。

React为变量类型内置了自动属性验证机制(*http://bit.ly/2okjSzJ*)，见表6-1。

表6-1：React属性验证

类型	验证器
数组	React.PropTypes.array
布尔值	React.PropTypes.bool
函数	React.PropTypes.func
数字	React.PropTypes.number
对象	React.PropTypes.object
字符串	React.PropTypes.string

在这一节中，我们将为菜谱应用创建一个Summary组件。这个Summary组件将会显示菜谱的标题，以及食材成分和烹饪步骤的数目（见图6-1）。

图6-1：Baked Salmon使用Summary组件描述的输出结果

为了显示这些数据，我们必须为Summary组件提供以下三种属性：标题，食材成分数组和烹饪步骤数组。现在希望对这些属性进行验证，从而确保第一个属性是字符串类型，其余的是数组类型，并且在它们无法通过属性验证时，提供默认参数。如何实现属性验证功能取决于组件是如何创建的。无状态函数式组件和ES6的类有多种实现属性验证的方法。

首先，让我们来看看用户应使用属性验证的原因，以及如何在React.createClass创建的组件中实现它。

使用createClass验证Props

我们需要理解验证组件属性为何如此重要的原因。考虑下面这个Summary组件的代码实现：

```
const Summary = createClass({
    displayName: "Summary",
    render() {
        const {ingredients, steps, title} = this.props
```

```
        return (
            <div className="summary">
                <h1>{title}</h1>
                <p>
                     <span>{ingredients.length} Ingredients</span> |
                    <span>{steps.length} Steps</span>
                </p>
             </div>
        )
    }
})
```

Summary组件从属性对象中解析出了食材成分列表、烹饪步骤和标题，然后构造了一个UI来显示这些数据。因为我们希望食材成分和烹饪步骤都采用数组来表示，这样就可以使用Array.length统计数组的元素个数。

如果我们因为失误而使用字符串渲染这个Summary组件又该怎么办呢？

```
render(
  <Summary title="Peanut Butter and Jelly"
           ingredients="peanut butter, jelly, bread"
           steps="spread peanut butter and jelly between bread" />,
  document.getElementById('react-container')
)
```

JavaScript将不会有任何怨言，不过统计数量的信息将会是根据每个字符串的字符数而得出的（见图6-2）。

图6-2：Summary组件为Peanut Butter and Jelly显示的摘要信息

上述代码的输出结果中食材成分的数量是奇数。不管你的花生酱和果冻如何美味，都不可能包含27种食材成分和44个烹饪步骤。和看到正确的成分种类和烹饪步骤相反，我们看到的数字是每个字符串的长度。这样的Bug很容易被忽视。如果我们在创建Summary组件时能够对属性类型进行验证，那么React将会捕获此类错误：

```
const Summary = createClass({
    displayName: "Summary",
    propTypes: {
        ingredients: PropTypes.array,
```

```
        steps: PropTypes.array,
        title: PropTypes.string
    },
    render() {
        const {ingredients, steps, title} = this.props
        return (
            <div className="summary">
                <h1>{title}</h1>
                <p>
                    <span>{ingredients.length} Ingredients | </span>
                    <span>{steps.length} Steps</span>
                </p>
            </div>
        )
    }
})
```

通过React内置的属性类型验证机制，我们可以确保食材成分种类和烹饪步骤都是数组。

此外，我们还可以确保标题是字符串类型。现在，当我们传递了不正确的属性类型时，将会看到一个错误提示信息（见图6-3）。

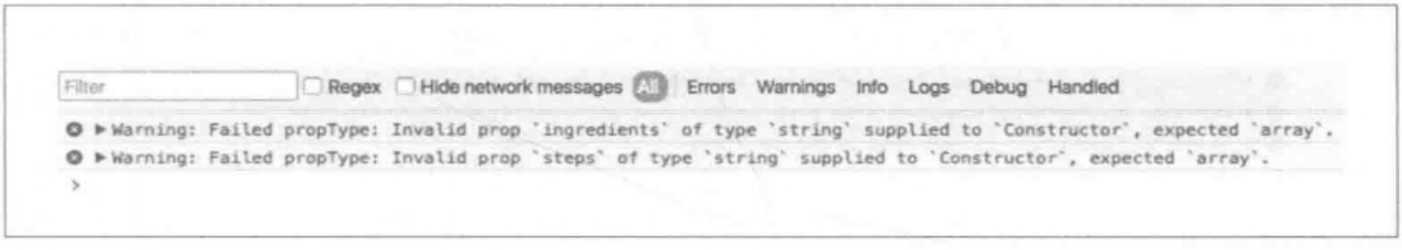


图6-3：属性类型验证的警告信息

当我们渲染Summary组件时，如果我们不发送任何属性数据又会怎么样呢？

```
render(
  <Summary />,
  document.getElementById('react-container')
)
```

在没有提供任何属性数据的情况下渲染Summary组件会导致JavaScript报错，从而拖垮整个Web应用（见图6-4）。

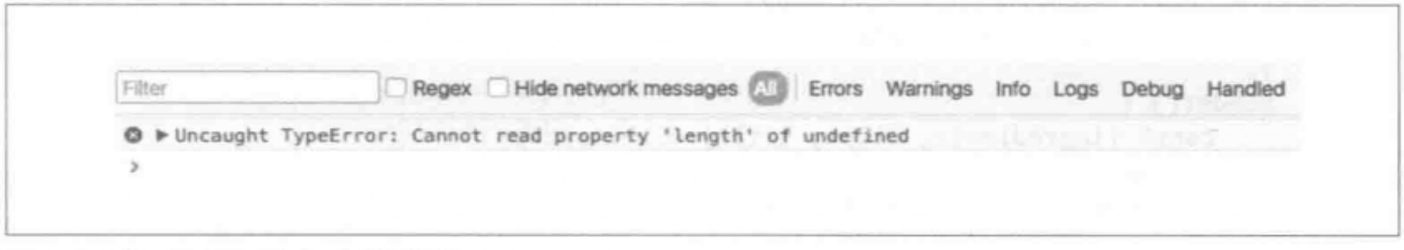


图6-4：缺乏数据时产生的错误

导致这个错误是因为属性ingredients的类型是未定义的，并且该未定义属性不是诸如数组或者字符串这样包含length属性的对象。React包含一种声明所需属性的方法。当这些属性和预期不一致时，React将会在控制台输出一个警告信息：

```
const Summary = createClass({
    displayName: "Summary",
    propTypes: {
        ingredients: PropTypes.array.isRequired,
        steps: PropTypes.array.isRequired,
        title: PropTypes.string
    },
    render() {
        ...
    }
})
```

现在当我们渲染Summary组件不提供相应的数据时，React将会在触发上述错误之前就在控制台向用户发送警告信息。这使得我们更容易了解产生问题的具体原因（见图6-5）。

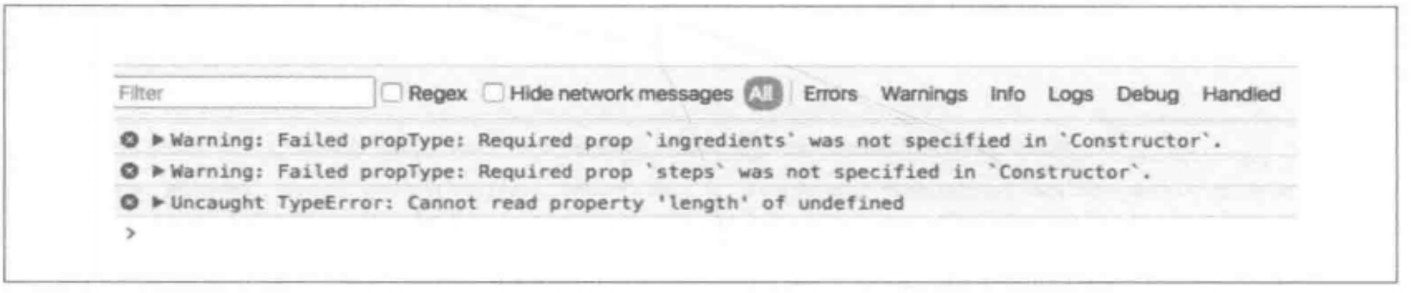


图6-5：React会警告用户缺少相关属性

Summary组件预期会接收一个食材成分数组和一个烹饪步骤数组，不过它只用到了每个数组的length属性。这个组件的主要用途是显示这些属性值的统计数据。重构组件可能会更有意义，因为该组件实际上并不需要使用数组：

```
import { createClass, PropTypes } from 'react'

export const Summary = createClass({
    displayName: "Summary",
    propTypes: {
        ingredients: PropTypes.number.isRequired,
        steps: PropTypes.number.isRequired,
        title: PropTypes.string
    },
    render() {
        const {ingredients, steps, title} = this.props
        return (
            <div className="summary">
                <h1>{title}</h1>
```

```
                <p>
                    <span>{ingredients} Ingredients</span> |
                    <span>{steps} Steps</span>
                </p>
            </div>
        )
    }
})
```

在组件中使用数字是一个更灵活的方法。现在Summary组件只需显示UI即可，它将实际的统计食材成分种类和烹饪步骤工作进一步沿着组件树向上传递给了它的父节点或者祖先节点。

默认Props

另外一种改进组件质量的方法是为属性设置默认值。[注1]验证行为和用户预期的一致：如果没有提供其他属性值，那么系统将会采用默认的属性值。

现在假定我们希望Summary组件即使在没有提供属性数据的情况也可以正常工作：

```
import { render } from 'react-dom'

render(<Summary />, document.getElementById('react-container'))
```

在createClass方法中，我们可以添加一个名为getDefaultProps的方法，以便可以在用户没有提供属性值的情况下返回默认的属性值：

```
const Summary = createClass({
    displayName: "Summary",
    propTypes: {
        ingredients: PropTypes.number,
        steps: PropTypes.number,
        title: PropTypes.string
    },
    getDefaultProps() {
        return {
            ingredients: 0,
            steps: 0,
            title: "[recipe]"
        }
    },
    render() {
        const {ingredients, steps, title} = this.props
        return (
            <div className="summary">
                <h1>{title}</h1>
                <p>
```

注1：React Docs，“Default Prop Values”（*http://bit.ly/2oYLr4r*）。

```
                    <span>{ingredients} Ingredients | </span>
                    <span>{steps} Steps</span>
                </p>
            </div>
        )
    }
}
```

现在，当我们尝试在不提供属性数据的情况下渲染该组件，会看到系统采用了默认属性值完成相应操作，如图6-6所示。

图6-6：采用了默认属性值的Summary组件

采用了默认属性后可以提高组件的灵活性，并且可以防止用户未明确声明每个必须的属性时发生错误。

自定义属性验证

React内置的验证机制非常适合确保用户所需变量类型的正确性。不过有些情况下需要更强大的验证功能才能满足需要。比如用户需要确保某个数字是属于特定区间的，或者某个参数值包含特定字符串。针对这种情况，React提供了一种方法允许用户构建自己的验证机制。

React中的自定义验证机制是通过一个函数来实现的。该函数既可以在不满足特定验证条件时返回一个错误，也可以在属性通过验证后返回一个null值。

对于基本的属性类型验证，我们只能基于一种条件验证一个属性。不过幸运的是，自定义属性验证允许用户以多种方式对属性进行验证。在自定义函数中，我们首先会验证该属性值是否是字符串类型。然后将其长度限制在20个字符以内（参见示例6-1）。

示例6-1：自定义属性验证

```
propTypes: {
    ingredients: PropTypes.number,
    steps: PropTypes.number,
    title: (props, propName) =>
        (typeof props[propName] !== 'string') ?
```

```
            new Error("A title must be a string") :
            (props[propName].length > 20) ?
                new Error(`title is over 20 characters`) :
                null
}
```

所有属性类型验证器都是函数。为了实现自定义属性验证函数，我们将会对propTypes对象下的title属性和一个回调函数关联。当渲染该组件时，React将会把属性对象和当前属性作为参数传递给该函数。我们可以使用这些参数对特定属性的属性值进行校验。

在这种情况下，我们首先需要确保title是字符串类型。如果title不是字符串，那么验证器将会返回一个包含错误提示的信息："A title must be a string"。如果title是字符串类型，那么就会检查它的值是否超过了20个字符。如果title的值在20个字符以内，验证器函数将会返回null值。如果title的值大于20个字符，那么验证器函数将会返回一个错误。React将会捕获该错误信息，然后以警告的形式在控制台上输出。

自定义验证器允许用户实现特定的验证规则。一个自定义验证器可以执行多重验证，只在不符合某些特定规则时返回错误。在使用或者复用用户组件时，采用自定义验证器是一种是防止产生错误的好办法。

ES6类和无状态函数式组件

在前面章节中，我们已经知道属性验证功能，以及使用React.createClass创建自定义组件时，可以添加默认的属性值。

这种类型检查也适用于ES6规范中的类和无状态函数式组件，不过语法稍有不同。

在使用ES6类时，propTypes和defaultProps是在类的实体之外定义的。一旦定义了类，我们就可以设置propTypes和defaultProps对象属性了（参见示例6-2）。

示例6-2：ES6类

```
class Summary extends React.Component {
  render() {
   const {ingredients, steps, title} = this.props
      return (
          <div className="summary">
              <h1>{title}</h1>
              <p>
                  <span>{ingredients} Ingredients | </span>
                  <span>{steps} Steps</span>
              </p>
          </div>
      )
```

```
  }
}

Summary.propTypes = {
  ingredients: PropTypes.number,
  steps: PropTypes.number,
  title: (props, propName) =>
    (typeof props[propName] !== 'string') ?
        new Error("A title must be a string") :
        (props[propName].length > 20) ?
            new Error(`title is over 20 characters`) :
            null
}

Summary.defaultProps = {
    ingredients: 0,
    steps: 0,
    title: "[recipe]"
}
```

对象属性propTypes和defaultProps还可以添加到无状态函数式组件中（参见示例6-3）。

示例6-3：无状态函数式组件

```
const Summary = ({ ingredients, steps, title }) => {
  return <div>
    <h1>{title}</h1>
    <p>{ingredients} Ingredients | {steps} Steps</p>
  </div>
}

Summary.propTypes = {
  ingredients: React.PropTypes.number.isRequired,
  steps: React.PropTypes.number.isRequired
}

Summary.defaultProps = {
  ingredients: 1,
  steps: 1
}
```

在无状态函数式组件中，用户还可以直接在函数参数中设置默认属性。在解析属性对象时，我们还可以在函数参数中为食材成分、烹饪步骤和标题等属性设置默认值：

```
    const Summary = ({ ingredients=0, steps=0, title='[recipe]' }) => {
      return <div>
        <h1>{title}</h1>
        <p>{ingredients} Ingredients | {steps} Steps</p>
      </div>
    }
```

类的静态属性

在上一章中，我们已经学习了如何在类的外部定义属性defaultProps和propTypes。最新的ECMAScript规范提案中提供了一种替代性的解决方案：*Class Fields & Static Properties*。

类的静态属性允许用户在类的内部声明中封装propTypes和defaultProps属性。属性构造器也提供了封装功能和更简洁的语法：

```
class Summary extends React.Component {

  static propTypes = {
    ingredients: PropTypes.number,
    steps: PropTypes.number,
    title: (props, propName) =>
      (typeof props[propName] !== 'string') ?
          new Error("A title must be a string") :
          (props[propName].length > 20) ?
              new Error(`title is over 20 characters`) :
              null
  }

  static defaultProps = {
    ingredients: 0,
    steps: 0,
    title: "[recipe]"
  }

  render() {
   const {ingredients, steps, title} = this.props
      return (
          <div className="summary">
              <h1>{title}</h1>
              <p>
                  <span>{ingredients} Ingredients | </span>
                  <span>{steps} Steps</span>
              </p>
          </div>
      )
    }
  }
```

最好在每个组件都实现属性验证、自定义属性验证，以及设置默认属性值等功能。这样一来，复用组件就更容易了，因为任何与组件属性有关的问题都会在控制台中输出警告信息。

引用

引用（ref）这个特性允许React组件能够和其子元素交互。引用最常见的用途是和UI元素交互，以便收集用户的输入信息。比如HTML中form表单元素。这些元素已经初始化渲染完毕，不过用户可以与之交互。当用户执行上述操作时，组件需要响应用户的操作。

本章接下来将会构建一个应用，帮助用户管理和保存特定的十六进制颜色值。这个颜色管理器程序允许用户将颜色添加到一个列表中。某个颜色被添加到列表中后，用户就可以对它评级或者删除。我们将需要一个表单来收集用户提供的新颜色信息。用户可以在相应字段中输入颜色名称和它的十六进制值。AddColorForm组件将会使用一个文本输入框和一个用于选择十六进制颜色值的按钮来渲染HTML元素（参见示例6-4）。

示例6-4：AddColorForm组件

```
import { Component } from 'react'

class AddColorForm extends Component {
  render() {
      return (
        <form onSubmit={e=>e.preventDefault()}>
            <input type="text"
                  placeholder="color title..." required/>
            <input type="color" required/>
            <button>ADD</button>
        </form>
      )
  }
}
```

AddColorForm组件渲染的HTML表单包含三个元素：一个用于输入颜色标题的文本框，一个用于选择十六进制颜色值的按钮，以及一个表单提交按钮。当表单被提交后，将会触发一个句柄函数用来忽略默认的表单事件。这样可以防止表单提交请求之后再次尝试发送GET请求。

表单渲染完毕之后，我们需要提供一种方法能够与之交互。具体来说，当第一次提交表单请求后，我们需要收集新的颜色信息，然后重置表单中的字段，以便用户可以添加更多颜色。通过引用，我们可以访问标题和颜色元素，并和它们交互（参见示例6-5）。

示例6-5：包含submit方法的AddColorForm组件

```
import { Component } from 'react'

class AddColorForm extends Component {
  constructor(props) {
    super(props)
    this.submit = this.submit.bind(this)
  }
  submit(e) {
    const { _title, _color } = this.refs
    e.preventDefault();
    alert(`New Color: ${_title.value} ${_color.value}`)
    _title.value = '';
    _color.value = '#000000';
    _title.focus();
  }
  render() {
      return (
          <form  onSubmit={this.submit}>
              <input ref="_title"
                     type="text"
                     placeholder="color title..." required/>
              <input ref="_color"
                     type="color" required/>
              <button>ADD</button>
          </form>
      )
  }
}
```

需要为这个ES6类添加一个构造器，因为我们将submit移动到了它自己的函数中。对于ES6组件类来说，我们需要使用this关键字将组件的作用域和任何希望访问该组件作用域的方法绑定。

然后，在render方法中，我们将表单的onSubmit句柄指向了组件的submit方法。同时还将相关的引用字段添加到了组件中方便访问。一个引用就是一个标识符，React通过标识符访问DOM元素。为每个输入框创建引用属性_title和_color，意味着我们可以通过this.refs_title或者this.refs_color访问这些元素。

当用户添加了一个新的标题，选择了新的颜色值，然后提交了该表单，组件的submit方法将会被触发来处理该事件。当我们阻止了表单的默认提交行为之后，我们向用户发送了一个警告提示对话框，回传通过引用收集到的数据。当用户关闭警告对话框之后，再次调用引用重置表单，并将输入焦点定位于标题输入框上。

绑定'this'作用域

当用户使用React.createClass创建组件时，不需要将this作用域和用户的组件方法绑定。React.createClass会自动为用户完成这些工作。

翻转数据流

在警告对话框中回传表单中输入的数据是一个不错的想法，但是这样一款产品并不能带来实际的经济价值。我们需要从用户那里收集数据，然后将这些数据发送到某处进行加工。这意味着任何收集到数据最终都会被发送到服务端，与之有关的细节我们将会在第12章深入展开。

首先，我们需要从表单组件中收集数据，并且可以传递它们。

一个常用的从React组件中收集数据的方案是翻转数据流。[注2]有时人们也称这种方法为双向数据绑定。它通过将一个回调函数作为属性传递给组件，使得组件能够通过属性访问回传数据。之所以称为翻转数据流是因为我们将组件函数当作一个属性进行传递的，并且组件将回传数据当作了函数参数。

假如我们采用了颜色表单，但是目前用户提交的新颜色，即我们希望收集的信息要求在控制台中输出。

我们可以创建一个名为logColor的函数，并将标题和颜色作为函数参数。这些参数的值可以被输出到控制台上。当我们使用AddColorForm时，我们可以简单地添加一个名为onNewColor的属性，并将它与logColor函数关联。当用户添加新颜色时，logColor被触发执行了，然后我们可以将该函数作为属性传递：

```
const logColor = (title, color) =>
    console.log(`New Color: ${title} | ${value}`)

<AddColorForm onNewColor={logColor} />
```

为了确保数据遵循了正确的格式，我们将会使用相应的数据对onNewColor属性进行初始化。

```
submit() {
    const {_title, _color} = this.refs
    this.props.onNewColor(_title.value, _color.value)
    _title.value = ''
    _color.value = '#000000'
```

注2： Pete Hunt, "Thinking in React" (*http://bit.ly/2nvMwgl*)。

```
        _title.focus()
    }
```

在我们的组件中，这意味着通过调用this.props.onNewColor替换了对弹出警示框的调用，并且将我们通过引用获得的新标题和颜色值传递了出去。

AddColorForm组件的作用是收集和传递数据。它并不关心具体的数据是什么。现在我们可以使用这个表单收集用户提交的颜色数据，然后将它们发送给其它组件或者方法来处理收集到的数据：

```
<AddColorForm onNewColor={(title, color) => {
    console.log(`TODO: add new ${title} and ${color} to the list`)
    console.log(`TODO: render UI with new Color`)
}} />
```

一切准备就绪后，我们就可以从该组件中收集数据，然后将新的颜色添加到颜色列表中。

可选的函数属性

为了使用数据双向绑定选项，用户必须首先在调用它们之前确认该函数属性是否存在。在上一个示例中，如果未提供函数属性onNewColor将会导致JavaScript报错，因为该组件尝试调用一个未定义的值。

这可以通过首先检查函数属性的存在性来避免：

```
if (this.props.onNewColor) {
    this.props.onNewColor(_title.value,  _color.value)
}
```

更好的解决解决方案是在组件的propTypes和defaultProps中定义该函数属性：

```
AddColorForm.propTypes = {
    onNewColor: PropTypes.func
}

AddColorForm.defaultProps = {
    onNewColor: f=>f
}
```

现在，当提供的属性类型是函数以外的其他类型时，React将会报错。如果未提供onNewColor属性，那么系统会默认调用简单函数f=>f。这是一个简单的占位符函数，目的是返回发送给它的第一个参数。虽然这个占位符函数没有做任何事情，但是它可以被JavaScript正确调用。

无状态函数式组件中的引用

引用也适用于无状态函数式组件。这些组件中没有'this'，因此不能通过this.refs这样的方式使用引用。除了使用字符串属性，我们将通过引用调用函数。该函数会将输入实例通过参数传递给我们。我们可以获得该实例，然后将它保存为本地变量。

接下来让我们对AddColorForm重构，使之成为一个无状态函数式组件：

```
const AddColorForm = ({onNewColor=f=>f}) => {
    let _title, _color
    const submit = e => {
        e.preventDefault()
        onNewColor(_title.value, _color.value)
        _title.value = ''
        _color.value = '#000000'
        _title.focus()
    }
    return (
        <form onSubmit={submit}>
            <input ref={input => _title = input}
                   type="text"
                   placeholder="color title..." required/>
            <input ref={input => _color = input}
                   type="color" required/>
            <button>ADD</button>
        </form>
    )
}
```

在这个无状态函数式组件中，ref引用指向了一个回调函数而不是一个字符串。回调函数会将元素实例作为参数进行传递。可以捕获该实例，并将它保存为类似_title和_color这样的本地变量。一旦我们将引用保存为本地变量，提交表单时就很容易访问它们了。

React的State管理

目前为止，我们只介绍了在React组件中使用属性处理数据。属性是不可变。一旦渲染完毕，一个组件的属性就不能更改了。为了处理UI变化的情况，我们需要其它机制来使用新的属性重新渲染组件树。React的State是一种内置的数据管理机制，它可以修改组件内部的状态。当应用程序状态变化时，UI会被重新渲染来反映这些变化。

用户会与应用程序交互。他们会进行导航、搜索、过滤、查询、添加、更新和删除等操作。当用户和一个应用程序交互时，应用程序的State也随之变化，同时这些变化又以UI的变化反馈给用户。屏幕和菜单的出现和消失、可见内容的变化、表示强调的光线明暗变化等。在React中，是用UI来反映应用程序State变化的。

React组件中的State可以用单个JavaScript对象表示。当某个组件的State发生变化时，该组件会渲染一个新的UI反映这些变化。那有没有更符合函数式风格的方法呢？答案当然是肯定的。给出一些数据，React组件将会在UI上反映这些数据变化。修改这些数据后，React会尽可能高效地更新UI来反映这些变化。

接下来让我们看看如何在React组件集成State管理。

组件State简介

State反映的数据表示我们希望对组件内部做出的更改。为了演示这一过程，我们将会以StarRating组件为例进行说明（见图6-7）。

图6-7：StarRating组件

StarRating组件所需的关键数据有两种：要显示的星标符号总数，以及需要高亮显示的评级星标数目。

我们将会需要一个包含selected属性并且支持点击的Star组件。一个无状态函数式组件对于每个星标都是适用的：

```
const Star = ({ selected=false, onClick=f=>f }) =>
    <div className={(selected) ? "star selected" : "star"}
         onClick={onClick}>
    </div>

Star.propTypes = {
    selected: PropTypes.bool,
    onClick: PropTypes.func
}
```

每个Star元素将由一个包含'star'类的div构成。如果星标被选中了，那么还会额外添加'selected'类。该组件还包含一个可选的onClick属性。当用户单击任意一个星标div时，onClick属性将会被触发。这也将告知父组件StarRating，某个星标被单击了。

Star是一个无状态函数式组件。顾名思义：你无法在一个无状态函数式组件中使用State。无状态函数式组件通常意味着其子元素是更复杂、State多样的组件。尽可能尝试编写无状态的组件是一个非常好的习惯。

由CSS构造的Star

我们的StarRating组件是使用CSS构造和显示星标的。具体来说就是使用*clip-path*属性，我们可以对div的某些区域进行剪裁来构造一个星标图形。剪裁路径的点集构造多边形：

```
.star {
    cursor: pointer;
    height: 25px;
    width: 25px;
    margin: 2px;
    float: left;
    background-color: grey;
    clip-path: polygon(
        50% 0%,
        63% 38%,
        100% 38%,
        69% 59%,
        82% 100%,
        50% 75%,
        18% 100%,
        31% 59%,
        0% 38%,
        37% 38%
    );
}

.star.selected {
  background-color: red;
}
```

一个普通星标的背景色是灰的，不过被选中星标的背景色是红的。

现在我们有了一个Star组件，接下来就可以使用它构造StarRating组件了。StarRating组件会从组件属性中读取希望显示的星标数目。用户可以修改评分值，该数值会被存放在State中。

首先，让我们来看看如何将State集成到使用createClass定义的组件中：

```
const StarRating = createClass({
    displayName: 'StarRating',
    propTypes: {
        totalStars: PropTypes.number
    },
    getDefaultProps() {
```

```
        return {
            totalStars: 5
        }
    },
    getInitialState() {
        return {
            starsSelected: 0
        }
    },
    change(starsSelected) {
        this.setState({starsSelected})
    },
    render() {
        const {totalStars} = this.props
        const {starsSelected} = this.state
        return (
            <div className="star-rating">
                {[...Array(totalStars)].map((n, i) =>
                    <Star key={i}
                          selected={i<starsSelected}
                          onClick={() => this.change(i+1)}
                    />
                )}
            <p>{starsSelected} of {totalStars} stars</p>
        </div>
    )
  }
})
```

在使用createClass时，State可以通过在组件配置中添加getInitialState方法进行初始化，返回的JavaScript对象可以将State变量starsSelected的值设置为0。

在渲染组件时，totalStars是从组件属性中读取的，并可以用来渲染特定数目的Star元素。具体来说，将扩展运算符和Array构造函数一起使用，从而初始化一个特定长度的新数组，用于映射Star元素。

当渲染组件时，State变量starsSelected是从this.state中解析的。它可以用于在段落元素中以文本的形式显示评级结果。它还可以用于计算被选中星标数目。每个Star元素是通过将其自身索引和被选中星标数目进行比较而获得selected属性值的。如果有三个星标被选中了，那么前三个Star元素的selected属性值将会是true，其余的元素的selected属性值将会是false。

最后，当用户单击单个星标时，特定Star元素的索引将会自动加1，然后发送给change函数。该索引值增加的原因是因为假定第一个星标被选中了，其索引值是0，但是评分值应该是1。

在ES6组件类中初始化State和使用createClass方法稍有不同。在这些类中，State可以在构造函数中初始化：

```
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 className="star-rating">
                {[...Array(totalStars)].map((n, i) =>
                    <Star key={i}
                          selected={i<starsSelected}
                          onClick={() => this.change(i+1)}
                    />
                )}
                <p>{starsSelected} of {totalStars} stars</p>
             </div>
        )
    }

}

StarRating.propTypes = {
    totalStars: PropTypes.number
}

StarRating.defaultProps = {
    totalStars: 5
}
```

加载一个ES6组件后，它会使用注入的属性作为第一个参数调用构造函数。这些属性通过调用super方法最终被传递给了其父类。在这种情况下，其父类就是React.Component。调用super方法将会初始化组件实例，React.Component将通过附加State管理功能的方式对上述实例进行修饰。super方法执行完毕后，我们可以对组件的State变量进行初始化。

Stae被初始化之后，它的操作就和createClass组件中类似了。State只能通过调用

this.setState来修改。每次调用setState之后，render函数也会被调用，渲染新的UI反映State的变化。

根据属性初始化State

我们可以使用传入的属性初始化State值。这种模式只适用于极个别必要的情况。这种情况最常见的场景是我们创建了一个可复用的组件时，希望在不同组件树之间跨应用调用该组件。

在使用createClass时，基于输入的属性初始化State变量的好办法是添加一个名为componentWillMount的方法。当加载组件时，该方法会被调用一次，并且用户可以在该方法中调用this.setState()。它还可以访问this.props，因此用户可以读取this.props中的数据辅助执行初始化State的操作：

```
const StarRating = createClass({
    displayName: 'StarRating',
    propTypes: {
        totalStars: PropTypes.number
    },
    getDefaultProps() {
        return {
            totalStars: 5
        }
    },
    getInitialState() {
        return {
            starsSelected: 0
        }
    },
    componentWillMount() {
        const { starsSelected } = this.props
        if (starsSelected) {
          this.setState({starsSelected})
        }
    },
    change(starsSelected) {
        this.setState({starsSelected})
    },
    render() {
        const {totalStars} = this.props
        const {starsSelected} = this.state
        return (
            <div className="star-rating">
                {[...Array(totalStars)].map((n, i) =>
                    <Star key={i}
                        selected={i<starsSelected}
                        onClick={() => this.change(i+1)}
                    />
                )}
```

```
                <p>{starsSelected} of {totalStars} stars</p>
            </div>
        )
    }
})

render(
    <StarRating totalStars={7} starsSelected={3} />,
    document.getElementById('react-container')
)
```

componentWillMount是组件生命周期的一部分。它可以用于帮助用户在由createClass创建的组件或者ES6类组件中根据属性值初始化State。我们将会在下一章深入介绍组件的生命周期。

在ES6类组件内部，有一种更简单的方法初始化State。构造函数通过参数接收属性数据，因此用户可以简单地调用传给构造函数的props参数：

```
constructor(props) {
    super(props)
    this.state = {
        starsSelected: props.starsSelected || 0
    }
    this.change = this.change.bind(this)
}
```

大部分情况下，用户将会希望避免根据属性初始化State变量。只有在绝对必要的情况下才可以这么做。用户将会发现这个目标很容易达成，因为在使用React组件时，用户希望限制包含State的组件数量。[注3]

更新组件属性

根据组件属性初始化State变量时，当父组件改变了这些属性，用户也许需要重新初始化组件State。组件生命周期方法componentWillRecieveProps可以用来解决这个问题。第7章将会深入介绍这个问题，并介绍相关的组件生命周期方法。

组件树的内部State

所有的React组件都包含自身的State，但是应该怎么管理它们呢？使用React的乐趣并不在于对应用程序中的Stae变量寻根究底。其乐趣在于构建易于理解的可扩展应用程序。让应用程序易于理解的关键之一是尽可能减少使用State的组件数量。

注3： React Docs, “Lifting State Up” (*http://bit.ly/2o6ob0z*)。

在大部分React应用中，在根组件中对所有Stae数据进行分组是完全可能的。State数据可以通过属性在组件树上传递，并且数据可以通过双向数据绑定回传到树根。最终的结果是整个应用程序用到的State数据都存放于一处。这通常可以确保“真实单一数据源”。[注4]

接下来，我们将介绍如何构建表现层，将所有State存储在一个地方，即根节点组件。

颜色管理器程序概述

颜色管理器允许用户在他们的自定义列表中对颜色进行添加、命名、评分和移除操作。颜色管理器实际的State可以用单个数组表示：

```
{
    colors: [
        {
            "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
        }
    ]
}
```

这个数组告诉我们需要显示三种颜色：海洋蓝、草坪绿和鲜红色（见图6-8）。它给我们提供了颜色的十六进制值，并展示了每种颜色的当前评分。它还提供了唯一表示每种颜色的方法。

这些State数据将会驱动我们的应用程序。它还会用于每次对象发生变更之后构造UI界面。当用户添加或者移除一种颜色时，它们也会相应的从数组中添加或者移除。当用户对颜色评分时，数组中的评分值也会发生相应的变化 。

注4：Paul Hudson, “State and the Single Source of Truth”, 《Hacking with React》第12章（*http://bit.ly/2ne6BdY*）。

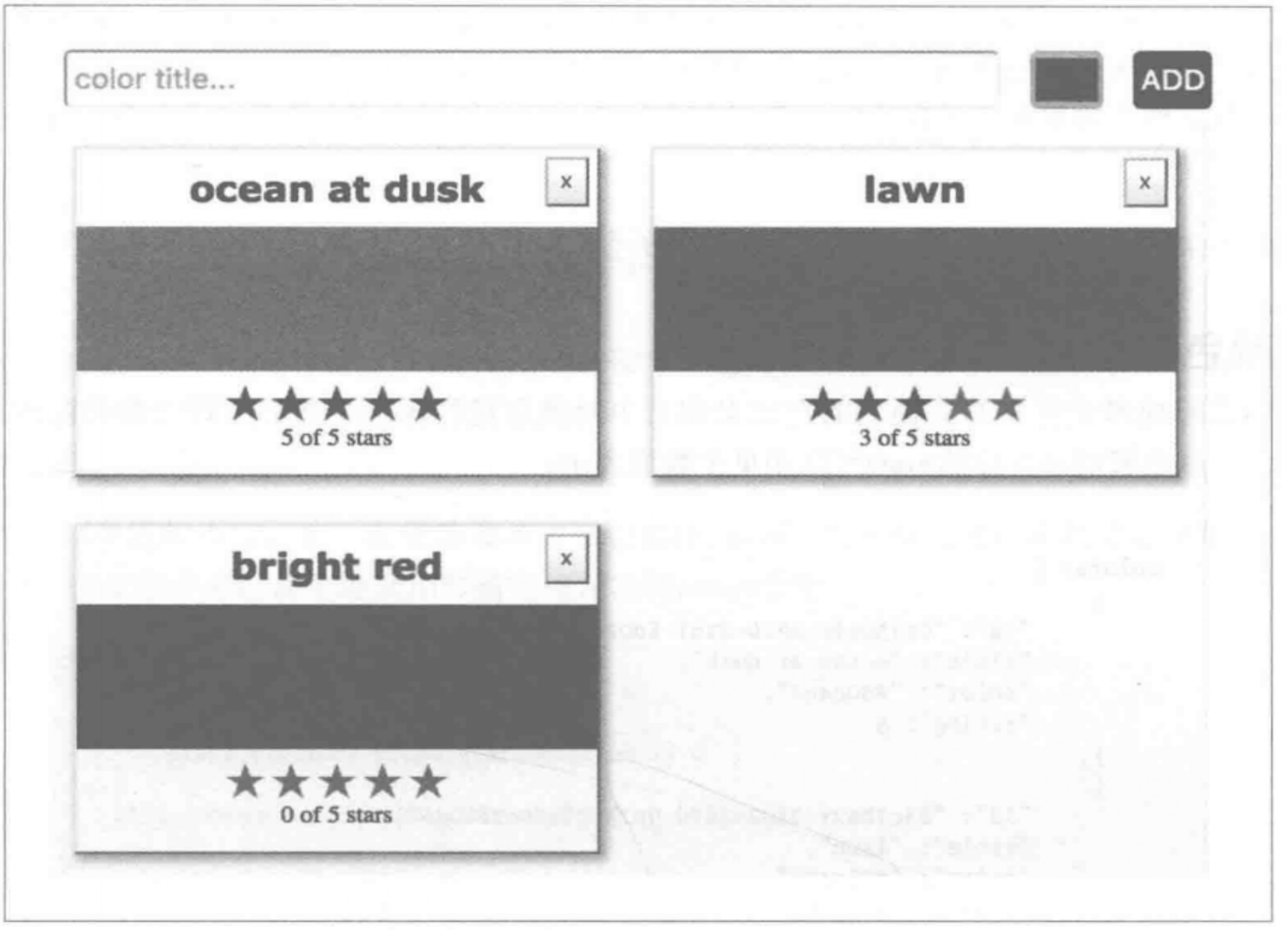


图6-8：包含三种颜色State的颜色管理器

在组件树上向下传递属性

本章前面介绍过将属性在组件树上向下传递，我们创建了一个StarRating组件用于保存State中的评分数据。在颜色管理器中，评分数据是存储在每个颜色对象中的。将StarRating组件当作表现层组件更有意义，[注5]并且将它声明为一个无状态函数式组件。

表现层组件只关心应用程序的外观。它们只渲染DOM元素或者其他表现层组件。传递给这些组件的所有数据都是通过属性完成的，输出的数据是通过回调函数传递的。

为了让StarRating组件变成纯粹的表现层组件，我们需要将其中的State数据移除。表现层组件只使用参数属性。因为我们已经从该组件中移除了State数据，当用户修改评分时，这些数据将会通过一个回调函数从组件中回传输出：

注5：Dan Abramov, "Presentational and Container Components", Medium, March 23, 2015 (*http://bit.ly/2ndQ9u0*)。

```
const StarRating = ({starsSelected=0, totalStars=5, onRate=f=>f}) =>
    <div className="star-rating">
        {[...Array(totalStars)].map((n, i) =>
            <Star key={i}
                  selected={i<starsSelected}
                  onClick={() => onRate(i+1)}/>
        )}
        <p>{starsSelected} of {totalStars} stars</p>
    </div>
```

首先，starsSelected不再是一个State变量，它现在是一个属性。其次，一个onRate回调函数被添加到组件中。与之前当用户修改评分后调用setState相反，该组件现在会调用onRate回调函数，并将评分数据作为参数进行传递。

可复用组件的State

用户也许需要创建一个包含多种State的UI组件，以便在若干不同应用程序中分发和复用。将组件中的所有State变量移除，使之只用于表现外观的做法并不是绝对的。这是一条非常好的规则，不过有时在表现层组件中保留State可能效果更好。

将State数据限制在单一位置，即根组件，意味着所有数据都必须作为属性向下传递给子组件（见图6-9）。

在颜色管理器中，State数据是以一个颜色数组在App应用组件中进行声明的。这些颜色会作为属性向下传递给ColorList组件：

```
class App extends Component {

    constructor(props) {
        super(props)
        this.state = {
            colors: []
        }
    }

    render() {
        const { colors } = this.state
        return (
            <div className="app">
                <AddColorForm />
                <ColorList colors={colors} />
            </div>
        )
    }

}
```

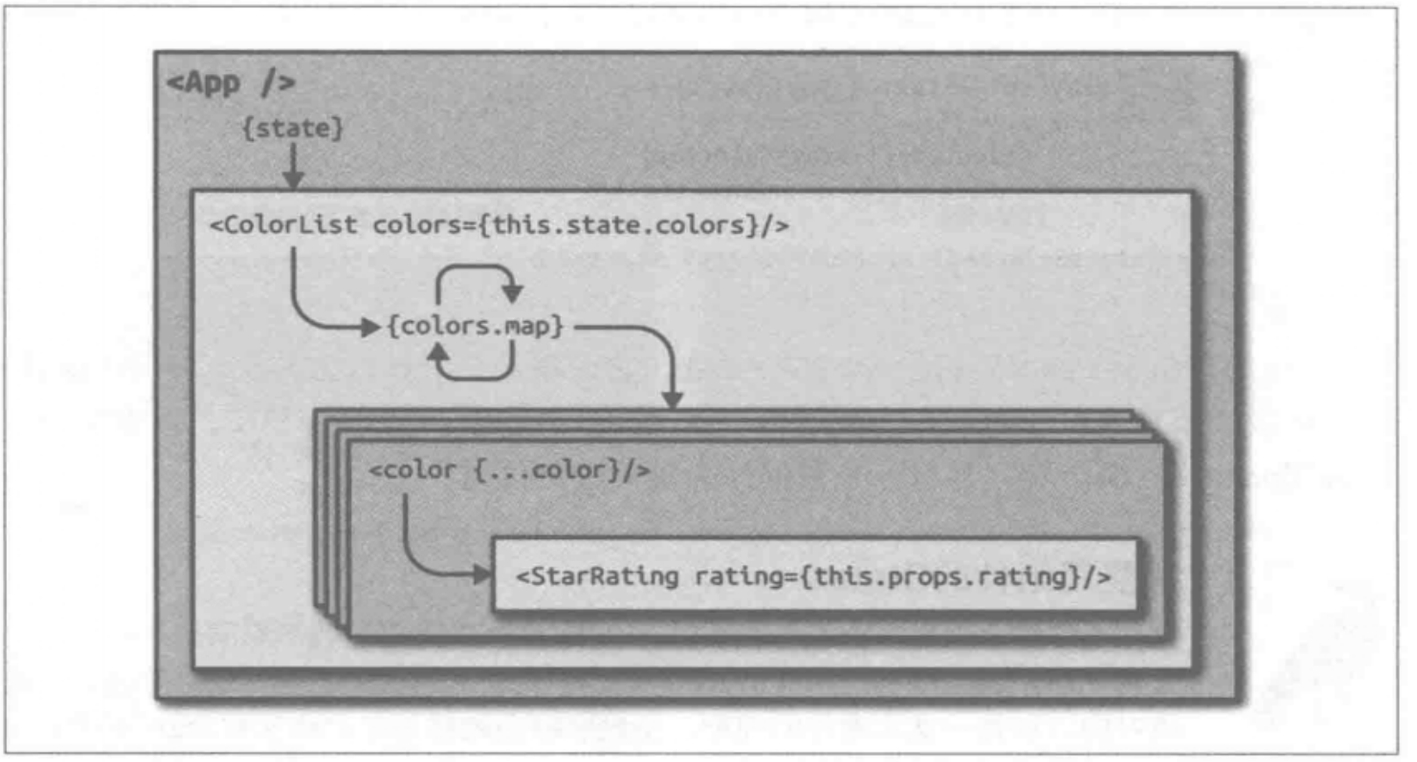


图6-9：App应用中的State数据是通过属性向下传递给子组件的

颜色数组初始化时是空的，因此ColorList组件将会显示一段文本来替代颜色。当数组中包含颜色时，每种独立的颜色数据将作为属性传递给Color组件：

```
const ColorList = ({ colors=[] }) =>
    <div className="color-list">
        {(colors.length === 0) ?
            <p>No Colors Listed. (Add a Color)</p> :
            colors.map(color =>
                <Color key={color.id} {...color} />
            )
        }
    </div>
```

现在Color组件可以显示颜色的标题和十六进制值，并将颜色评分作为属性向下传递给StarRating组件：

```
const Color = ({ title, color, rating=0 }) =>
    <section className="color">
        <h1>{title}</h1>
        <div className="color"
            style={{ backgroundColor: color }}>
        </div>
        <div>
            <StarRating starsSelected={rating} />
        </div>
    </section>
```

在星标评分中，starsSelected的数量来自每种颜色的评分。每种颜色的所有State数据在组件树中都是通过属性向下传递给子组件的。当根组件中的State数据发生变化后，React将会尽可能高效地修改UI来反映这些新的State。

回传数据到组件树

颜色管理器中的State只能通过调用App组件中的setState方法更新。如果用户初始化读取了任意来自UI的变更，它们的输入将会需要在组件树中向上回传到App组件，以便更新State数据（见图6-10）。这可以通过回调函数的属性来完成。

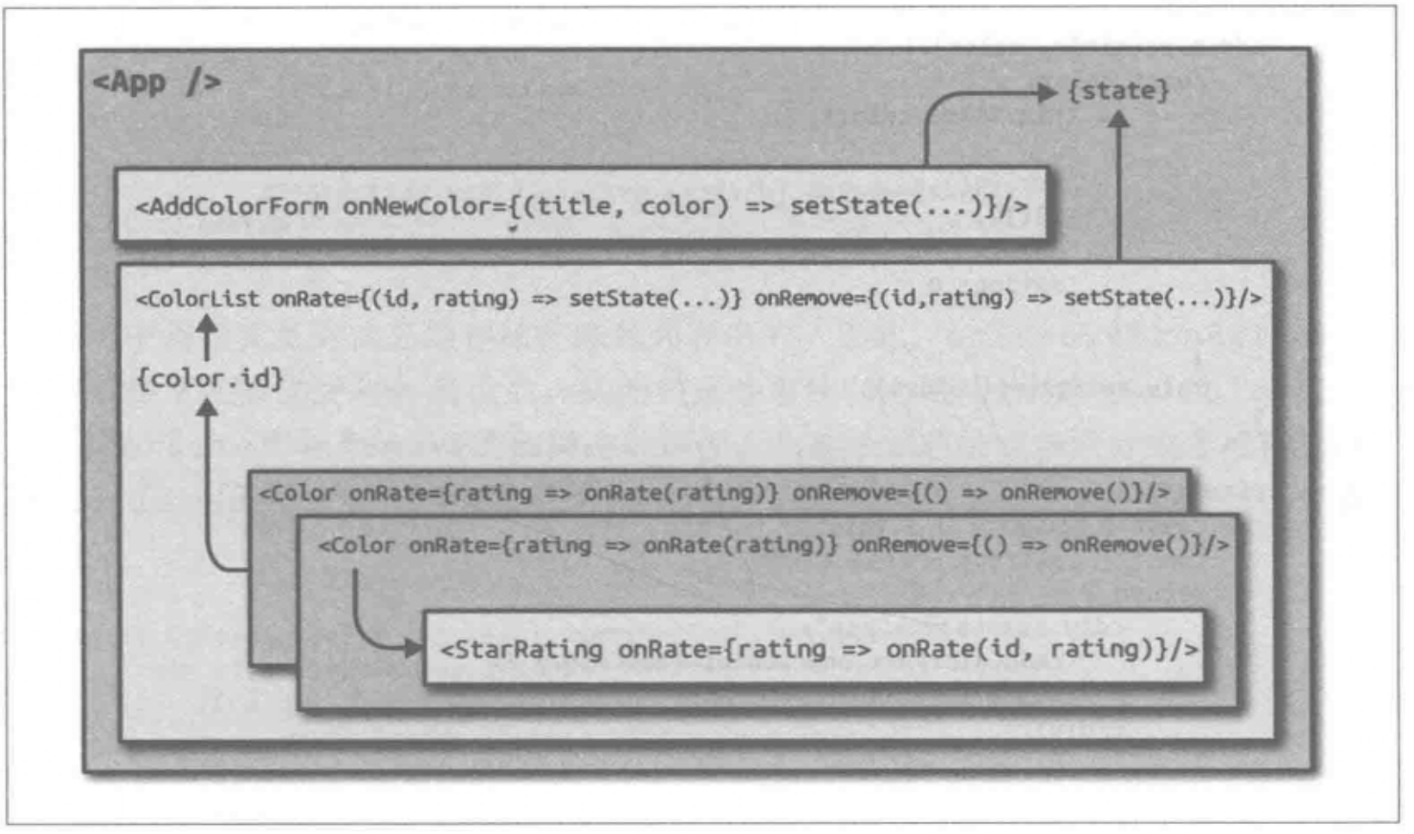


图6-10：当存在UI事件时，向上回传数据到根组件

为了添加新的颜色，我们需要一种唯一标识每种颜色的方法。这个标识符将会用于定位State数组中的颜色。我们可以使用uuid库来创建绝对唯一的ID：

```
npm install uuid --save
```

所有从我们在本章前面的"引用”中构造的AddColorForm组件收集到的新颜色将会被添加到颜色管理器中。该组件包含一个可选的回调函数属性，名为onNewColor。当用户添加了一种新的颜色，然后提交表单，onNewColor回调函数会根据从用户那里获取的新标题和十六进制颜色值触发执行：

```
import { Component } from 'react'
import { v4 } from 'uuid'
import AddColorForm from './AddColorForm'
import ColorList from './ColorList'

export class App extends Component {

    constructor(props) {
        super(props)
        this.state = {
            colors: []
        }
        this.addColor = this.addColor.bind(this)
    }

    addColor(title, color) {
        const colors = [
            ...this.state.colors,
            {
                id: v4(),
                title,
                color,
                rating: 0
            }
        ]
        this.setState({colors})
    }

    render() {
        const { addColor } = this
        const { colors } = this.state
        return (
            <div className="app">
                <AddColorForm onNewColor={addColor} />
                <ColorList colors={colors} />
            </div>
        )
    }

}
```

App组件中的addColor方法获得的所有新颜色都可以被添加。该函数是在构造函数中和组件绑定的，这意味着它拥有访问this.state和this.setState的能力。

新的颜色通过在当前颜色数组中添加一个新的颜色对象的方式被添加了。新颜色对象的ID是通过调用uuid库的v4函数设置的。这会为每种颜色设定唯一标识符。标题和颜色是通过AddColorForm组件传递给addColor方法的。最后，每种颜色的初始评分数据将会是0。

当用户使用AddColorForm组件添加了一种颜色，addColor方法将会使用新的颜色列表更新组件State。一旦State数据被更新了，App组件将会使用新的颜色列表重新渲染组

件树。 render方法会在每次调用setState方法之后触发。新的数据将会以属性的形式在组件树中向下传递，然后用于构造UI。

如果用户希望对颜色评分或者移除某个颜色，我们需要收集和该颜色有关的信息。每种颜色将会包含一个移除按钮：如果用户单击了移除按钮，我们将会知道用户希望移除该颜色。当然，如果用户通过StarRating组件修改某个颜色的评分数据，我们将希望修改该颜色的评分：

```
const Color = ({title,color,rating=0,onRemove=f=>f,onRate=f=>f}) =>
    <section className="color">
        <h1>{title}</h1>
        <button onClick={onRemove}>X</button>
        <div className="color"
             style={{ backgroundColor: color }}>
        </div>
        <div>
            <StarRating starsSelected={rating} onRate={onRate} />
        </div>
    </section>
```

该应用中将要变更的信息是存储在颜色列表中的。因此，onRemove和onRate回调属性将不得不被添加到每种颜色中，从而将这些事件向上回传给组件树。Color组件也将获得onRate和onRemove的回调函数属性。当某个颜色被重新评分或者被移除时，ColorList组件将会通知它的父组件，App组件，告知该组件应该对上述颜色评分或者移除：

```
const ColorList = ({ colors=[], onRate=f=>f, onRemove=f=>f }) =>
    <div className="color-list">
        {(colors.length === 0) ?
             <p>No Colors Listed. (Add a Color)</p> :
             colors.map(color =>
                 <Color key={color.id}
                 {...color}
                 onRate={(rating) => onRate(color.id, rating)}
                 onRemove={() => onRemove(color.id)} />
        )
    }
  </div>
```

如果任意颜色被评分或者移除，ColorList组件将会调用onRate或者onRemove方法。该组件通过将一组颜色映射成独立的颜色组件来管理它们。当某个独立颜色被评分或者移除后，ColorList会识别这些变化，并通过回调函数属性将这些信息回传给其父组件。

ColorList的父组件是App。在App组件中， rateColor方法和removeColor方法可以在

其构造函数中添加并和该组件实例绑定。当有颜色被评分或者移除时，这些方法将会更新State。它们被当作回调函数的属性添加到了ColorList组件中：

```
class App extends Component {

    constructor(props) {
        super(props)
        this.state = {
            colors: []
        }
        this.addColor = this.addColor.bind(this)
        this.rateColor = this.rateColor.bind(this)
        this.removeColor = this.removeColor.bind(this)
    }

    addColor(title, color) {
        const colors = [
            ...this.state.colors,
            {
                id: v4(),
                title,
                color,
                rating: 0
            }
        ]
        this.setState({colors})
    }

    rateColor(id, rating) {
        const colors = this.state.colors.map(color =>
             (color.id !== id) ?
                 color :
                 {
                      ...color,
                      rating
                 }
        )
        this.setState({colors})
    }

    removeColor(id) {
        const colors = this.state.colors.filter(
             color => color.id !== id
        )
        this.setState({colors})
    }

    render() {
        const { addColor, rateColor, removeColor } = this
        const { colors } = this.state
        return (
             <div className="app">
                 <AddColorForm onNewColor={addColor} />
                 <ColorList colors={colors}
```

```
                        onRate={rateColor}
                        onRemove={removeColor} />
                </div>
            )
        }

    }
```

rateColor方法和removeColor方法都希望获得被评分或者移除的颜色ID。这个ID被ColorList组件获取后，将会以参数的形式传递给rateColor或者removeColor。rateColor找到被评分的颜色，然后对State数据进行相应修改。removeColor方法将会使用Array.filter方法，在不移除颜色的情况下，创建一个新的State数组。

一旦setState方法被调用，UI就会根据新的State数据重新被渲染。这个应用程序中的所有数据变更都是通过单个组件进行管理的，即App。这种方法使得用户更容易理解应用程序使用了哪些数据创建State，以及数据是如何变化的。

React组件非常健壮。它为用户提供了一种简洁的方式管理和验证属性、与子元素交互，以及管理组件内部的State数据。这些特性使得用户能够构造一个非常精美、可扩展的表现层。

我们已经强调过很多遍，State是数据的变化。用户还可以使用State在应用程序中缓存数据。比如，如果你有一组可供客户搜索的记录列表，这个列表可以暂时存放在State数据中，直到它们被搜索调用为止。

将State限制在根组件通常是一种比较推荐的做法。读者将会在大量的React应用碰到这种情形。一旦用户的应用程序增长到一定规模，双向数据绑定和显式传递属性将会变得非常麻烦。

Flux设计模式和类似的Flux库Redux可以用于管理State，并且减少这些情况下的陈规陋习。

React库是一个相对比较小的脚本库，到目前为止，我们已经介绍了它的大部分功能。我们还未讨论的React组件主要特性，包括组件生命周期和高阶组件，上述内容将会在下一章详细介绍。

第7章

组件扩展

目前为止，我们已经学习了如何使用React挂载和合成组件来创建应用表现层。只使用React组件的render方法构建应用程序的可能性微乎其微。不过，JavaScript的世界是纷繁复杂的。到处都存在异步应用。在载入数据时需要处理延迟问题。创建动画时还需要延迟执行相关操作。极有可能读者已经有偏爱的JavaScript库来帮助你在复杂而真实的JavaScript世界乘风破浪了。

在我们能够使用第三方JavaScript库或者后端数据请求扩展我们的应用程序之前，我们必须首先充分了解如何使用组件生命周期，即每次我们挂载或更新组件过程中一系列可以调用的方法。

本章的内容将会以探索组件生命周期开始。在我们介绍了生命周期之后，将会重新审视如何使用它载入数据、集成第三方JavaScript库，以及改进用户的组件性能。然后，我们将介绍如何使用高阶组件实现应用程序之间的功能复用。最后，我们将会以介绍使用其他应用程序架构管理React之外的State作为本章结尾。

组件生命周期

组件生命周期包含挂载或者更新组件过程中被触发的一系列方法。这些方法可以在组件渲染UI之前或者之后被触发。事实上，`render`方法本身就是组件生命周期的一部分。有两种主要的生命周期：挂载生命周期和更新生命周期。

挂载生命周期

挂载生命周期是指当某个组件被挂载或者下载时被触发的一系列方法。换句话说，这些方法允许用户初始化State、创建API调用请求、启动或者停止计时器，操作已渲染的DOM、初始化第三方脚本库等。这些方法允许用户使用JavaScript辅助完成初始化或销毁某个组件的操作。

挂载生命周期根据用户使用的是ES6类语法还是React.createClass方法创建组件而稍有不同。当用户使用createClass时，getDefaultProps方法会被触发首先获得组件的属性。然后，getInitialState方法将会被触发，用于初始化State。

ES6类并不包含上述方法。取而代之的是，默认的props将会被获取，然后将它作为参数传递给构造函数。构造函数也是State被初始化的地方。ES6类构造函数和getInitialState都拥有访问属性的能力，如果有必要的话，可以使用它们辅助定义初始的State。

表7-1列出了组件挂载生命周期中的所有方法。

表7-1：组件挂载生命周期

ES6 类	React.createClass()
	getDefaultProps()
constructor(props)	getInitialState()
componentWillMount()	componentWillMount()
render()	render()
componentDidMount()	componentDidMount()
componentWillUnmount()	componentWillUnmount()

类构造函数

技术上来说，构造函数并不是一个生命周期方法。这里我们将它包含进来是因为它被用于组件初始化（这也是State被初始化的地方）。当然，当挂载某个组件时，构造函数永远都是第一个被触发的。

一旦属性被获取并且也初始化了State，componentWillMount方法将会被触发。该方法是在DOM被渲染之前触发的，并且可以用来初始化第三方脚本库、启动动画、请求数据，以及其他可能需要在组件被渲染之前执行的额外步骤。还可以在该方法中触发setState方法，在组件被初次渲染之前修改组件的State。

让我们使用componentWillMount方法为某些成员初始化一个请求。当成功获得响应结果，我们将会更新State。还记得在第2章创建的Promise对象getFakeMembers么？我们将会使用它从*randomuser.me*获得一个随机的成员列表：

```
const getFakeMembers = 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()
})
```

我们将会在MemberList组件中的componentWillMount方法中使用该Promise对象。这个组件将会使用一个Member组件显示每个用户的照片、姓名、电子邮件和位置：

```
const Member = ({ email, picture, name, location }) =>
    <div className="member">
        <img src={picture.thumbnail} alt="" />
        <h1>{name.first} {name.last}</h1>
        <p><a href={"mailto:" + email}>{email}</a></p>
        <p>{location.city}, {location.state}</p>
    </div>

class MemberList extends Component {

    constructor() {
        super()
        this.state = {
            members: [],
            loading: false,
            error: null
        }
    }

    componentWillMount() {
        this.setState({loading: true})
        getFakeMembers(this.props.count).then(
            members => {
                this.setState({members, loading: false})
            },
            error => {
                this.setState({error, loading: false})
            }
        )
    }

    componentWillUpdate() {
        console.log('updating lifecycle')
    }
```

```
    render() {
        const { members, loading, error } = this.state
        return (
        <div className="member-list">
             {(loading) ?
                 <span>Loading Members</span> :
                 (members.length) ?
                     members.map((user, i) =>
                          <Member key={i} {...user} />
                      ) :
                      <span>0 members loaded...</span>
                }
                {(error) ? <p>Error Loading Members: error</p> : ""}
      </div>
    )
  }
}
```

起初，当挂载该组件后，MemberList包含一个空的members数组，并且loading的值是false。在componentWillMount方法中，State被修改了，表示实际上创建了用于载入某些用户的请求。然后，当等待请求响应完成过程中，组件被渲染了。因为loading的值不是true，系统将会显示一个信息提示框告知用户存在延迟。当Promise对象执行完毕，loading的状态将会返回，其结果既可以是载入了某些成员，也可以是一个错误提示。此时调用setState方法将会载入某些成员或者使用一个错误提示来重新渲染UI。

在componentWillMount方法中使用setState

在组件被渲染完毕之前调用setState方法将不会启动更新生命周期。在组件渲染完毕之后调用setState方法就会启动更新生命周期。如果用户在componentWillMount方法中定义的回调函数内调用了setState方法，那么它将会在组件被渲染完毕之后被触发，并且会启动更新生命周期。

组件挂载生命周期的其他方法包括componentDidMount和componentWillUnmount。componentDidMount方法只会在组件渲染完毕之后触发。componentWillUnmount方法只会在组件被卸载之前触发。

componentDidMount方法是另外一个创建API请求的好地方。该方法会在组件渲染完毕之后触发，该方法中的任意setState方法调用都将启动更新生命周期，并且重新渲染组件。

componentDidMount也是初始化任何需要用到DOM的第三方JavaScript的好地方。比如，用户也许希望集成一个拖曳脚本库或者一个用于处理触摸屏事件的库。一般来说，这些库在初始化之前都需要用到DOM。

componentDidMount方法的另外一个优点是可以用于启动诸如intervals或者timers这样的后台进程。任何在componentDidMount或者componentWillMount方法中启动的进程都可以在componentWillUnmount方法中被清除。当不需要这些后台进程时，用户可能也不希望它们在后台继续执行。

当组件的父组件将它们移除或者被react-dom中的unmountComponentAtNode函数卸载时，它们将会被卸载。该方法还可以用于卸载根组件。当一个根组件被卸载时，它的子组件将会首先被卸载。

让我们看看时钟示例。当Clock组件被挂载后，将会启动一个timer计时器。当用户单击“close”按钮后，时钟程序将会被unmountComponentAtNode函数卸载，timer计时器也终止运行了：

```
import React from 'react'
import { render, unmountComponentAtNode } from 'react-dom'
import { getClockTime } from './lib'
const { Component } = React
const target = document.getElementById('react-container')

class Clock extends Component {

   constructor() {
       super()
       this.state = getClockTime()
    }

    componentDidMount() {
       console.log("Starting Clock")
       this.ticking = setInterval(() =>
                this.setState(getClockTime())
            , 1000)
    }

    componentWillUnmount() {
       clearInterval(this.ticking)
       console.log("Stopping Clock")
    }

    render() {
       const { hours, minutes, seconds, timeOfDay } = this.state
       return (
            <div className="clock">
                <span>{hours}</span>
                <span>:</span>
                <span>{minutes}</span>
                <span>:</span>
                <span>{seconds}</span>
                <span>{timeOfDay}</span>
                <button  onClick={this.props.onClose}>x</button>
```

```
                </div>
            )
        }

}

render(
    <Clock onClose={() => unmountComponentAtNode(target) }/>,
    target
)
```

在第3章中，我们创建了一个serializeTime函数，它从数据对象使用首位补零的方式封装了获取本地时间的方法，返回的当前时间位于一个对象中，包含时、分、秒，以及表示上午（AM）或者下午（PM）的标记。起初，我们调用serializeTime方法获取时钟的初始State。

当挂载该组件后，我们启动了一个名为ticking的时间间隔函数。它每隔一秒触发一次，使用获得的新时间设置State。时钟UI会修改它的数值，每隔一秒更新一次时间。

当“close”按钮被单击后，Clock组件将会被卸载。在时钟从DOM中移除之前，ticking间隔函数将被清除，防止它在后台继续运行。

更新生命周期

更新生命周期是当组件State发生变化或者从父组件接收到新的属性时触发的一系列方法。该生命周期可以用来在更新组件之前或者之后与DOM交互时集成JavaScript。此外，它还可以用于改进应用程序的性能，因为它为用户提供了取消不必要更新操作的能力。

更新生命周期会在每次调用setState方法后启动。调用setState方法过程内部的更新生命周期将会引发无限递归循环，从而导致堆栈溢出错误。因此，只能在componentWillReceiveProps方法内部调用setState方法，它允许组件属性被更新后更新State。

更新生命周期包括以下几个方法：

componentWillReceiveProps(nextProps)

仅当新的属性被传递给了组件后才会调用。这也是唯一可以调用setState方法的地方。

shouldComponentUpdate(nextProps, nextState)

更新生命周期的门卫，一个可以取消更新操作的谓词。该方法可以通过只允许执行必须的更新来改进性能。

componentWillUpdate(nextProps, nextState)

只在组件更新之前触发。和componentWillMount类似，只是它会在每次更新操作之前被触发。

componentDidUpdate(prevProps, prevState)

只在更新操作发生后，调用render方法之后触发。类似componentDidMount方法，不过它会在每次更新之后触发。

让我们修改一下上一章创建的颜色管理器应用程序。具体来说，我们将会添加一些更新周期函数到Color组件中，使得读者可以了解更新生命周期的工作机制。假定我们在state数组中已经拥有了4种颜色：深蓝色（Ocean Blue）、番茄红（Tomato）、浅绿色（Lawn）和粉红色（Party Pink）。首先，我们将通过componentWillMount方法使用一种样式初始化color对象，并且将4种Color元素的背景设置为灰色：

```
import { Star, StarRating } from '../components'

export class Color extends Component {

    componentWillMount() {
        this.style = { backgroundColor: "#CCC" }
    }

    render() {
        const { title, rating, color, onRate } = this.props
        return
            <section className="color" style={this.style}>
                <h1 ref="title">{title}</h1>
                <div className="color"
                     style={{ backgroundColor: color }}>
                </div>
                <StarRating starsSelected={rating}
                            onRate={onRate} />
            </section>
    }

}

Color.propTypes = {
  title: PropTypes.string,
  rating: PropTypes.number,
  color: PropTypes.string,
  onRate: PropTypes.func
}
```

```
Color.defaultProps = {
  title: undefined,
  rating: 0,
  color: "#000000",
  onRate: f=>f
}
```

当颜色列表最初挂载时，每种颜色的背景是灰色（见图7-1）。

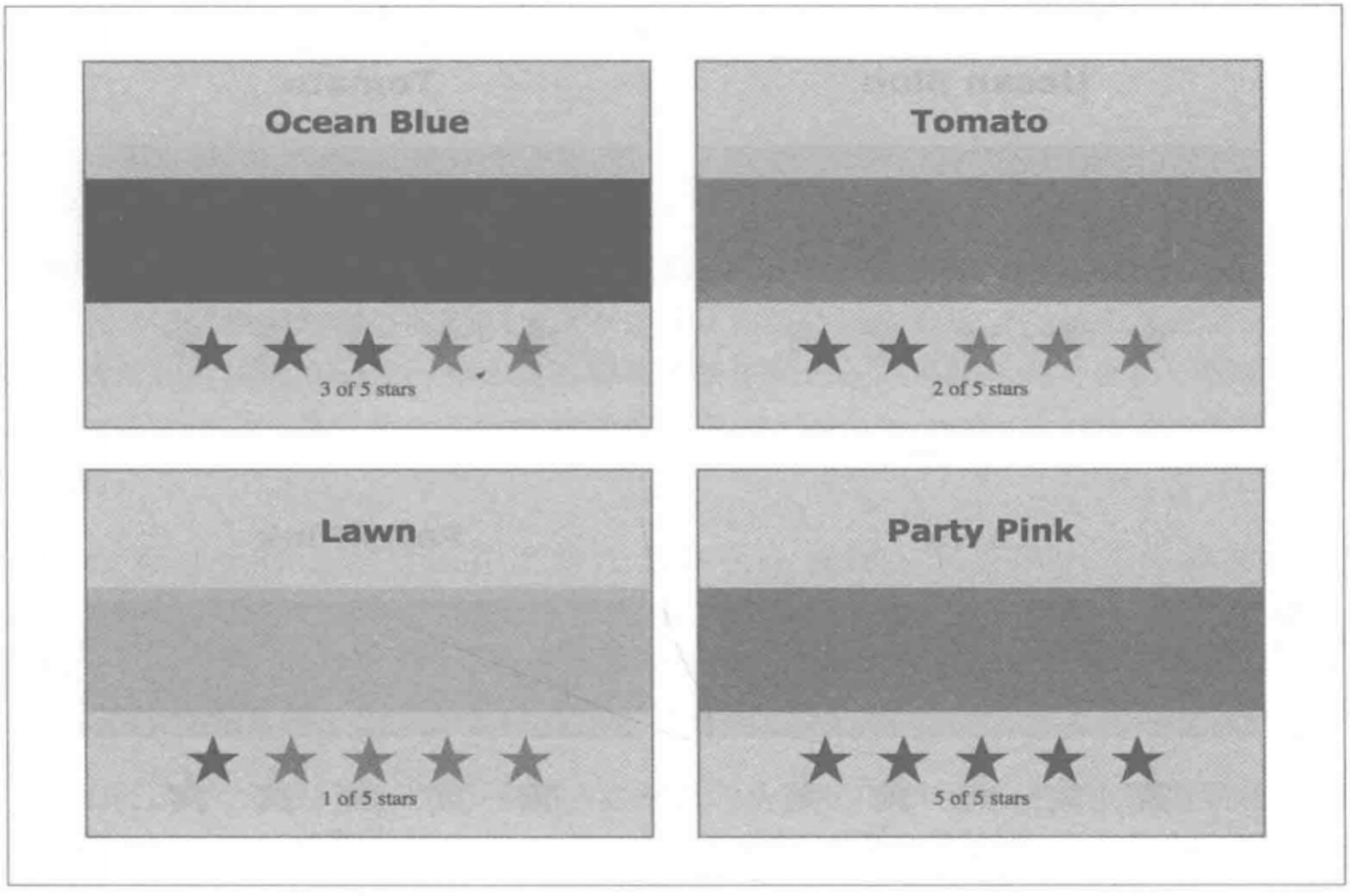


图7-1：使用灰色背景挂载的颜色列表

我们可以将componentWillUpdate添加到Color组件中，以便可以只在更新颜色之前移除每种颜色的灰色背景：

```
componentWillMount() {
  this.style = { backgroundColor: "#CCC" }
}

componentWillUpdate() {
    this.style = null
}
```

添加这些生命周期函数使得用户能够知道组件是何时挂载的，以及何时更新了组件。

起初，挂载组件将会拥有一个灰色背景。一旦某种颜色被更新了，其背景色将会恢复成白色。

如果用户运行这些代码，并对任意颜色进行评分，将会发现所有4种颜色的状态都被更新了，即使只对单个颜色进行了评分（见图7-2）。

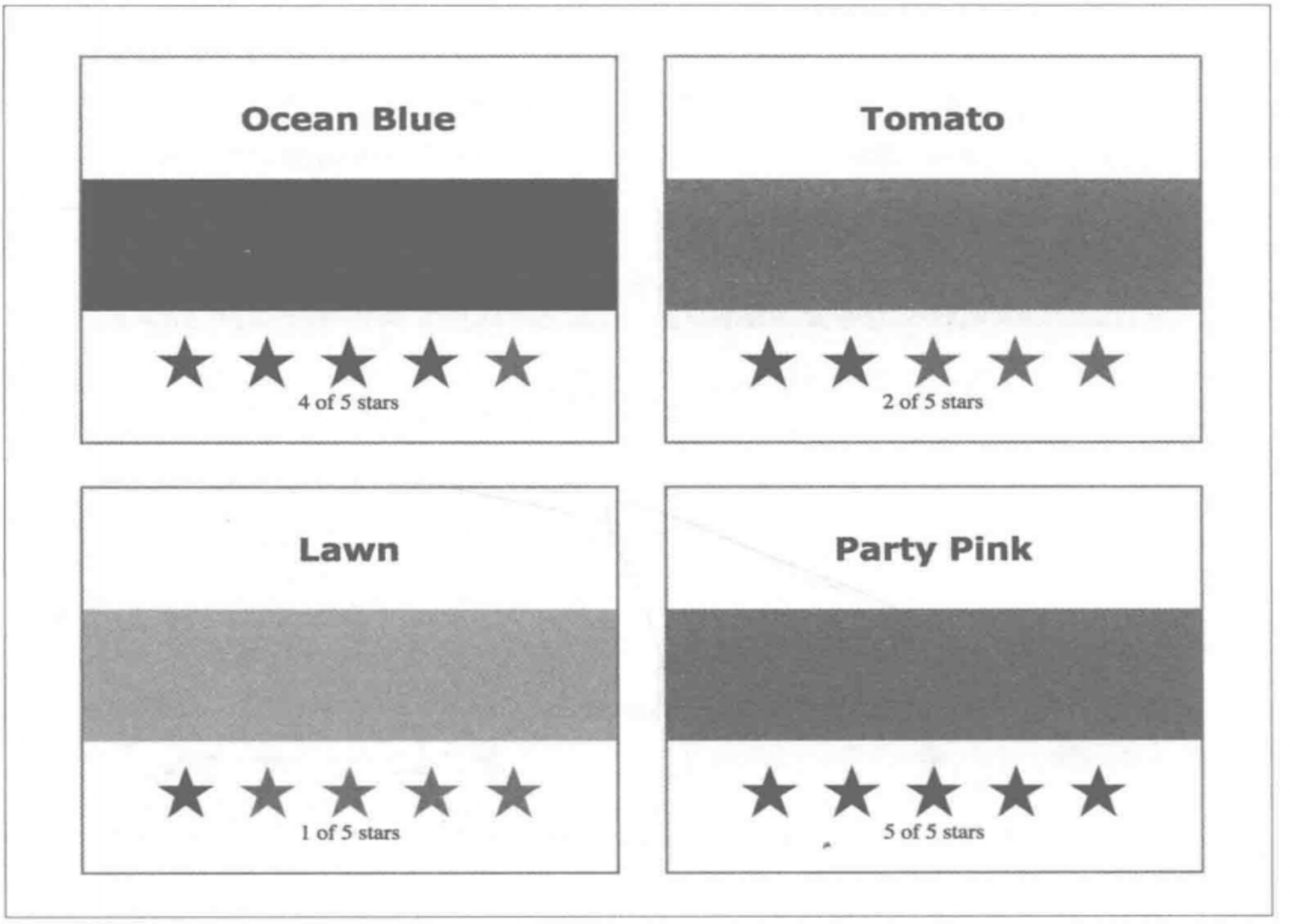


图7-2：对深蓝色进行评分后触发了更新操作，并且4个组件都更新了

这里，将深蓝色评分数据由三颗星改为4颗星后导致所有颜色元素都被更新了，这是因为其父组件ColorList更新了State，它重新渲染了每个Color组件。重写渲染组件并不是重新挂载；如果它们已经存在那里，更新将会替代挂载。

当一个组件被更新了，它的所有子元素也被更新了。当单个颜色被评分了，所有4种颜色也被更新了，所有4个StarRating组件被更新后，所有在每个组件上的5颗星标也被更新了。

我们可以通过当其属性值没有发生变化时阻止程序更新这些颜色，继而达到改进应用程序性能的目的。在相应的事件中添加生命周期函数shouldComponentUpdate来阻止

不必要的更新。该方法既可以返回true，也可以返回false（当组件应该更新时返回true，当应该跳过更新时返回false）：

```
componentWillMount() {
  this.style = { backgroundColor: "#CCC" }
}

shouldComponentUpdate(nextProps) {
    const { rating } = this.props
    return rating !== nextProps.rating
}

componentWillUpdate() {
    this.style = null
}
```

shouldComponentUpdate方法可以比较新旧属性之间的差异。新属性会作为参数传给该方法，旧属性仍然在当前的props中，并且该组件还未被更新。如果当前属性中的评分数据和新属性值一致，那么就不需要更新该颜色。如果颜色没有更新，那么它的所有子元素也不需要更新。当评分数据没有变化，每个Color组件下的组件树也不需要更新。

这可以通过运行上述代码并更新任意颜色来证明。只有该组件将要被更新时，componentWillUpdate方法才会被调用。在生命周期中，它是紧跟在shouldComponentUpdate方法之后执行的。背景会一直保持灰色，直到Color组件通过修改评分操作被更新为止（见图7-3）。

如果shouldComponentUpdate方法返回的是true，其余的更新生命周期将会知道这一点。其余的生命周期函数也会通过参数接收到新的props和新的State（componentDidUpdate方法会接收到上一个props和上一个State，因为一旦该方法被触发了，更新操作已经执行，props也发生了变化）。

让我们在组件更新后记录一条信息。在componentDidUpdate函数中，我们将会比较当前属性和旧属性之间的差别，看看评分数据是变得更好了还是更糟了：

```
componentWillMount() {
    this.style = { backgroundColor: "#CCC" }
}

shouldComponentUpdate(nextProps) {
    const { rating } = this.props
    return rating !== nextProps.rating
}

componentWillUpdate() {
    this.style = null
```

```
}

componentDidUpdate(prevProps) {
    const { title, rating } = this.props
    const status = (rating > prevProps.rating) ? 'better' : 'worse'
    console.log(`${title} is getting ${status}`)
}
```

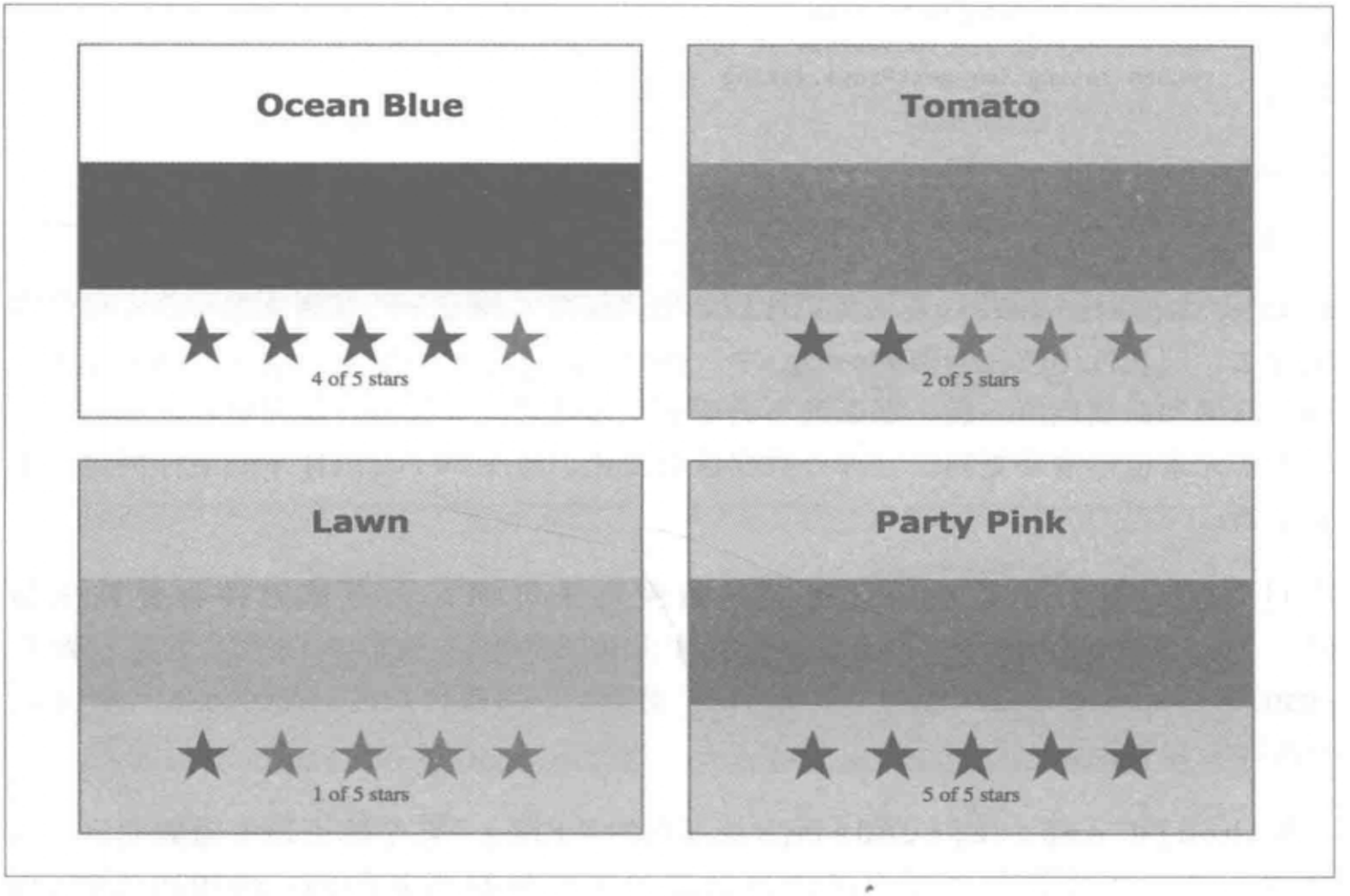


图7-3：使用shouldComponentUpdate方法每次只更新一个组件

更新生命周期方法componentWillUpdate和componentDidUpdate是更新之前或者之后与DOM交互的好地方。在接下来的示例中，更新过程将会被componentWillUpdate方法中的一个警告提示框暂停：

```
componentWillMount() {
    this.style = { backgroundColor: "#CCC" }
}

shouldComponentUpdate(nextProps) {
    return this.props.rating !== nextProps.rating
}

componentWillUpdate(nextProps) {
    const { title, rating } = this.props
    this.style = null
```

```
        this.refs.title.style.backgroundColor = "red"
        this.refs.title.style.color = "white"
        alert(`${title}: rating ${rating} -> ${nextProps.rating}`)
    }

    componentDidUpdate(prevProps) {
        const { title, rating } = this.props
        const status = (rating > prevProps.rating) ? 'better' : 'worse'
        this.refs.title.style.backgroundColor = ""
        this.refs.title.style.color = "black"
    }
```

如果将番茄红的评分从2颗星修改成4颗星，更新过程将会被一个警告提示框暂停（见图7-4）。颜色的当前DOM元素的背景和文本被设置成了不同的颜色。

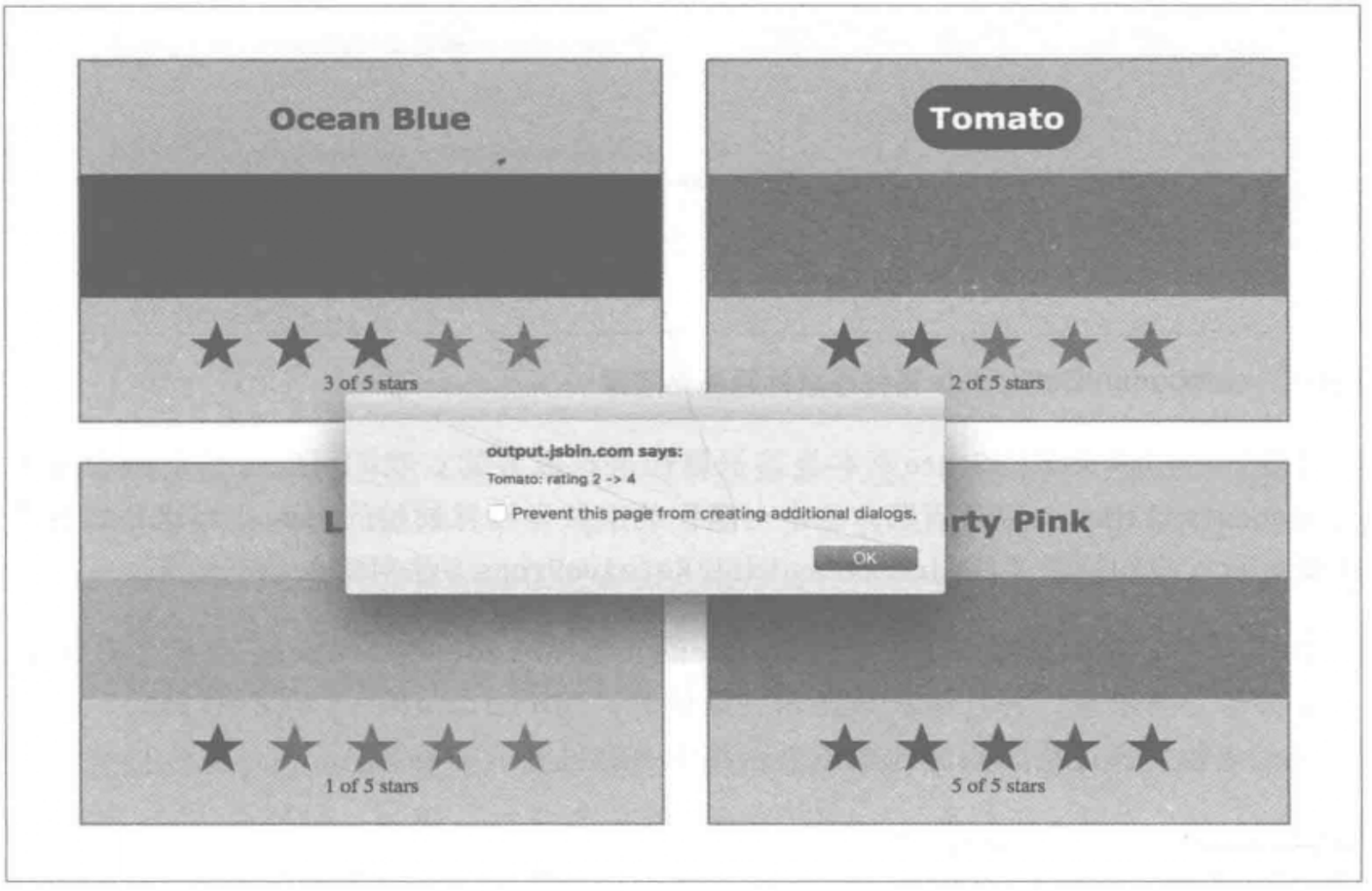


图7-4：被警告提示框暂停的更新过程

一旦我们清除了警告提示框，组件会立即更新，然后componentDidUpdate将会被触发，清除标题的背景色（见图7-5）。

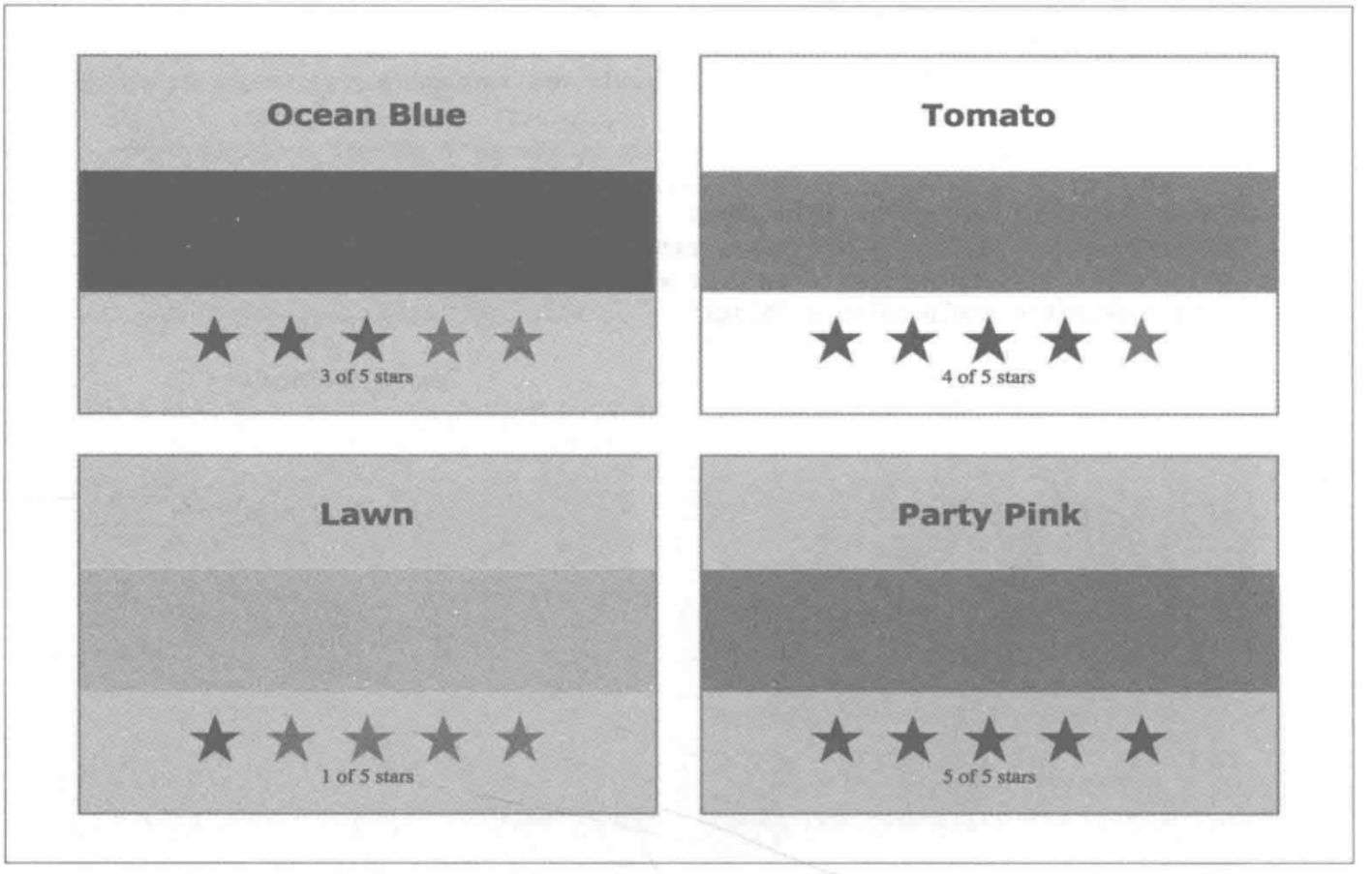


图7-5：componentDidUpdate将会移除标题高亮效果

有时我们的组件保留的State原本是基于属性进行设置的。我们可以在构造函数或者componentWillMount生命周期方法中对我们的组件类设置初始的State。当这些属性发生变更时，我们将需要使用componentWillReceiveProps方法对State进行更新。

在示例7-1中，我们已经有了一个保留State的父组件HiddenMessages。该组件在State中保留了三条信息，并且每次只显示一条信息。当HiddenMessages挂载时，一个interval函数会通过信息添加到循环中，并且一次只显示一条信息。

示例7-1：HiddenMessages组件

```
class HiddenMessages extends Component {

    constructor(props) {
        super(props)
        this.state = {
            messages: [
                "The crow crows after midnight",
                "Bring a watch and dark clothes to the spot",
                "Jericho Jericho Go"
            ],
            showing: -1
        }
    }
```

```
    componentWillMount() {
        this.interval = setInterval(() => {
            let { showing, messages } = this.state
            showing = (++showing >= messages.length) ?
                -1 :
                showing
            this.setState({showing})
        }, 1000)
    }

    componentWillUnmount() {
        clearInterval(this.interval)
    }

    render() {
        const { messages, showing } = this.state
        return (
            <div className="hidden-messages">
                {messages.map((message, i) =>
                    <HiddenMessage key={i}
                                   hide={(i!==showing)}>
                        {message}
                    </HiddenMessage>
                )}
            </div>
        )
    }
}
```

HiddenMessages组件会循环遍历state数组中的每条信息，并且每次只显示一条信息。其逻辑是在componentWillMount方法中设置的。当挂载该组件后，会添加一个interval间隔函数，继而更新将要显示的信息索引。组件会使用HiddenMessage组件渲染所有信息，并且在每个周期中只将其中某条信息的hide属性设置为true。其余信息的属性将会设置为false，隐藏的信息每秒变化一次。

仔细观察HiddenMessage组件，每个信息都会用到这个组件（参见示例7-2）。当该组件原本已经挂载完毕，hide属性会被用来决定它的State。不过当父组件更新了这个组件的属性后，不会发生任何变化。因为该组件将无法获知这一点。

示例7-2：HiddenMessage组件

```
class HiddenMessage extends Component {

    constructor(props) {
        super(props)
        this.state = {
            hidden: (props.hide) ? props.hide : true
        }
    }
```

```
    render() {
        const { children } = this.props
        const { hidden } = this.state
        return (
            <p>
                {(hidden) ?
                    children.replace(/[a-zA-Z0-9]/g, "x") :
                    children
                }
            </p>
        )
    }

}
```

产生该问题的时机是当父组件修改hide属性时。这一变更并不会自动导致HiddenMessage的State发生变化。

componentWillReceiveProps生命周期方法就是为了处理这类场景而存在的。它会在属性被父组件修改时被触发。并且这些变更的属性可以用来修改内部的State：

```
class HiddenMessage extends Component {

    constructor(props) {
        super(props)
        this.state = {
            hidden: (props.hide) ? props.hide : true
        }
    }

    componentWillReceiveProps(nextProps) {
        this.setState({hidden:  nextProps.hide})
    }

    render() {
        const { children } = this.props
        const { hidden } = this.state
        return (
            <p>
                {(hidden) ?

                    children.replace(/[a-zA-Z0-9]/g, "x") :
                    children
                }
            </p>
        )
    }

}
```

当父组件HiddenMessages修改了hide属性，componentWillReceiveProps方法允许用户对相应的State进行更新。

根据属性设置State

上述代码示例已经省去了componentWillReceiveProps用法的演示。如果这是我们使用HiddenMessage的初衷，那么应该使用一个无状态函数式组件取而代之。目前我们还将State添加到子组件的唯一原因是，希望该组件自身可以修改相关的State。

例如，如果组件需要一个setState调用，则使用componentWillReceiveProps来修改State是有必要的：

```
hide() {
    const hidden = true
    this.setState({hidden})
}

show() {
    const hidden = false
    this.setState({hidden})
}

return
    <p onMouseEnter={this.show}
       onMouseLeave={this.hide}>
        {(hidden) ?
            children.replace(/[a-zA-Z0-9]/g, "x") :
            children
        }
</p>
```

在这种情况下，最好将相应的State存放在HiddenMessage组件中。如果该组件不能修改自身的某些属性，那么最好让它保持无状态并只通过父组件管理State。

组件生命周期方法给用户提供了更精细的控制能力，来决定如何渲染或者更新某个组件。它为用户提供的钩子允许我们在挂载和更新发生之前或者之后都可以添加相应的功能。接下来，我们将会介绍如何使用这些生命周期方法集成第三方JavaScript库。首先，我们将简要介绍一下“React.Children” API。

React.Children

React.Children提供了一种可以访问特定组件子元素的方法。它允许用户统计、映射、循环，或者将props.children转换成一个数组。它还允许用户使用React.Children.only对正在显示的单个子节点进行验证：

```
import { Children, PropTypes } from 'react'
import { render } from 'react-dom'

const Display = ({ ifTruthy=true, children }) =>
    (ifTruthy) ?
        Children.only(children) :
        null

const age = 22

render(
    <Display ifTruthy={age >= 21}>
        <h1>You can enter</h1>
    </Display>,
    document.getElementById('react-container')
)
```

在这个示例中，Display组件将会只显示单个子元素，即h1标签元素。如果Display组件包含多个子节点，React将会抛出一个错误："onlyChild must be passed a children with exactly one child"。

我们还可以使用React.Children将子元素属性转化成一个数组。接下来的示例将会对Display组件进行扩展，以便可以额外处理其他情况：

```
const { Children, PropTypes } = React
const { render } = ReactDOM

const findChild = (children, child) =>
  Children.toArray(children)
          .filter(c => c.type === child )[0]

const WhenTruthy = ({children}) =>
    Children.only(children)

const WhenFalsy = ({children}) =>
    Children.only(children)

const Display = ({ ifTruthy=true, children }) =>
    (ifTruthy) ?
        findChild(children, WhenTruthy) :
        findChild(children, WhenFalsy)

const age = 19

render(
    <Display ifTruthy={age >= 21}>
        <WhenTruthy>
          <h1>You can Enter</h1>
        </WhenTruthy>
        <WhenFalsy>
          <h1>Beat it Kid</h1>
        </WhenFalsy>
```

```
        </Display>,
        document.getElementById('react-container')
    )
```

如果条件为true时，Display组件将会显示单个子节点，当条件为false时，Display组件将会显示其他内容。为了实现这一点，我们创建了WhenTruthy和WhenFalsy组件，并将它们作为Display组件的子元素。findChild函数会使用React.Children将子节点转换成一个数组。可以对该数组进行过滤和定位，并根据组件类型返回一个唯一的子节点元素。

集成JavaScript脚本库

Angular和jQuery这类框架库内置的工具包括数据访问、渲染UI、模型化State、路由处理等。换句话说，React只是一个简单的创建视图脚本库，因此我们需要将它和其他JavaScript脚本库搭配使用。如果我们理解了生命周期函数操作流程，那么我们可以让React库和其他JavaScript脚本库良好地协同工作。

React和jQuery

将React和jQuery一起搭配使用，往往会遭到社区成员的一致反对。将jQuery和React集成到一起协作是完全可行的，并且集成过程也是学习React或者将遗留代码迁移到React的好机会。不过如果我们没有选择大型的框架库，而是将更小的脚本库和React一起集成后，构造的应用程序性能会更好。此外，使用jQuery绕过虚拟DOM而直接操作DOM后，可能会导致一些奇怪的错误。

在本节中，我们将会集成一对不同的JavaScript脚本库到React组件中。具体来说，我们将会借助其他JavaScript脚本库，研究创建API调用和数据可视化的方法。

使用Fetch发送请求

Fetch是一款polyfill的脚本库（有些方法你不支持，polyfill使用你支持的方法帮你实现了这些你不支持的方法），由WHATWG小组研发，它允许用户使用Promise对象方便地创建API调用。本节将会介绍isomorphic-fetch库，它是一个可以和React良好协作的Fetch版本。让我先安装isomorphic-fetch库：

```
npm install isomorphic-fetch --save
```

组件生命周期函数为用户提供了一个地方专门集成JavaScript脚本库。在这种情况下，这个地方也是我们将会发起API调用的地方。组件发起API调用时必须处理延迟

问题，等待响应的同时还会让用户体验下降。我们可以通过引用若干变量的方式在我们的State中对这些问题进行定位，并告知组件某个请求是否被挂起了。

在下列示例中，CountryList组件创建了一个国家名称的有序列表。挂载完毕后，该组件会发起一个API调用，修改State来表示它正在载入数据。loading的值会一直保持为true，直到API请求得到响应结果：

```
import { Component } from 'react'
import { render } from 'react-dom'
import fetch from 'isomorphic-fetch'

class CountryList extends Component {

    constructor(props) {
        super(props)
        this.state = {
             countryNames: [],
              loading: false
        }
    }

    componentDidMount() {
        this.setState({loading: true})
        fetch('https://restcountries.eu/rest/v1/all')
            .then(response => response.json())
            .then(json => json.map(country => country.name))
            .then(countryNames =>
              this.setState({countryNames, loading: false})
            )
    }

    render() {
        const { countryNames, loading } = this.state,
        return (loading) ?
            <div>Loading Country Names...</div> :
            (!countryNames.length) ?
                <div>No country Names</div> :
                <ul>
                    {countryNames.map(
                        (x,i) => <li key={i}>{x}</li>
                    )}
                </ul>
    }

}

render(
    <CountryList />,
    document.getElementById('react-container')
)
```

当挂载组件时，只有在fetch请求之前，我们可以将loading的值设置为true。这会告

知我们的组件和最终用户，我们正在请求数据。当fetch调用获得响应时，我们会获得一个JSON对象，然后将它映射成一个包含国家名称的数组。最后，国家名称会被添加到State中，并且DOM也被更新了。

集成D3时间轴

数据驱动文档（D3）是一个JavaScript框架，可以用来在浏览器中构建数据可视化应用。D3提供了丰富的工具，允许用户缩放和插入数据。

此外，D3采用了函数式编程范式。用户可以通过链式函数调用合成D3应用程序，从而根据一个数组构造一个可视化DOM。

时间轴是一个数据可视化的例子。一个时间轴会将事件日期作为数据，然后以图形的形式直观地展示这些信息。以前发生的历史事件将会排列在稍后发生的事件的左侧。时间轴上每个事件之间的空格（以像素为单位）表示事件之间的间隔时间（见图7-6）。

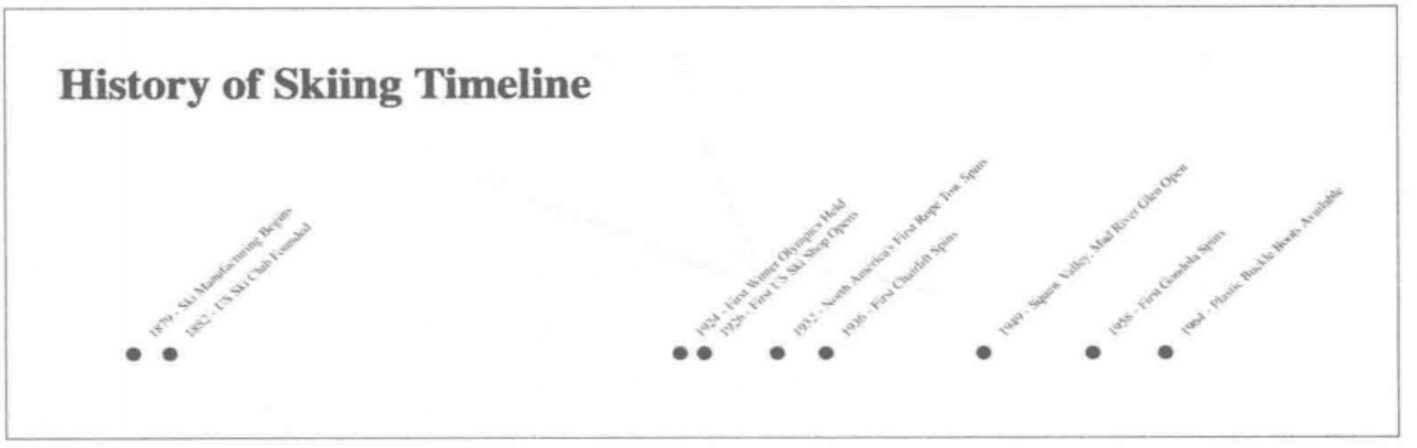


图7-6：时间轴的数据可视化效果

这个可视化时间轴只在500个像素内就表示了几乎100年间的重大事件。将年代值转换成像素值的过程被称作插值。D3提供了将一种数据区间插值到另外一种的所有必备工具。接下来让我们看看如何将D3和React集成到一起构造这个时间轴。首先，我们需要安装D3脚本库：

```
npm install d3 --save
```

D3处理的数据，一般是对象数组，开发可视化应用也是基于这些数据。先看看滑雪运动历史大事年表的数组。这是我们时间轴的数据：

```
const historicDatesForSkiing = [
    {
```

```
        year: 1879,
        event: "Ski Manufacturing Begins"
    },
    {
        year: 1882,
        event: "US Ski Club Founded"
    },
    {
        year: 1924,
        event: "First Winter Olympics Held"
    },
    {
        year: 1926,
        event: "First US Ski Shop Opens"
    },
    {
        year: 1932,
        event: "North America's First Rope Tow Spins"
    },
    {
        year: 1936,
        event: "First Chairlift Spins"
    },
    {
        year: 1949,
        event: "Squaw Valley, Mad River Glen Open"
    },
    {
        year: 1958,
        event: "First Gondola Spins"
    },
    {
        year: 1964,
        event: "Plastic Buckle Boots Available"
    }
]
```

将D3集成到一个React组件中最简单的办法是让React渲染UI，然后用D3创建和添加可视化元素。在下列示例中，D3被集成到了一个React组件中。一旦该组件渲染完毕，D3会构造可视化元素，并将它们添加到DOM中：

```
import d3 from 'd3'
import { Component } from 'react'
import { render } from 'react-dom'
class Timeline extends Component {

    constructor({data=[]}) {
        const times = d3.extent(data.map(d => d.year))
        const range = [50, 450]
        super({data})
        this.state = {data, times, range}
    }
```

```
    componentDidMount() {
        let group
        const { data, times, range } = this.state
        const { target } = this.refs
        const scale = d3.time.scale().domain(times).range(range)

        d3.select(target)
            .append('svg')
            .attr('height', 200)
            .attr('width', 500)

        group = d3.select(target.children[0])
            .selectAll('g')
            .data(data)
            .enter()
            .append('g')
            .attr(
                'transform',
                (d, i) => 'translate(' + scale(d.year) + ', 0)'
            )

        group.append('circle')
            .attr('cy', 160)
            .attr('r', 5)
            .style('fill', 'blue')

        group.append('text')
            .text(d => d.year + " - " + d.event)
            .style('font-size', 10)
            .attr('y', 115)
            .attr('x', -95)
            .attr('transform', 'rotate(-45)')
    }

    render() {
        return (
            <div className="timeline">
                <h1>{this.props.name}  Timeline</h1>
                <div ref="target"></div>
            </div>
        )
    }

}

render(
    <Timeline name="History of Skiing"
              data={historicDatesForSkiing} />,
    document.getElementById('react-container')
)
```

在这个示例中，D3的某些配置是在构造函数中设置的，不过D3负责的大部分工作是在componentDidMount函数中完成的。一旦DOM渲染完毕，D3会使用可伸缩矢量图形

格式（SVG）构建可视化效果。该方法可以奏效，并且也是一种快速集成现有D3可视化效果到React组件中的好办法。

然而，我们可以通过让React管理DOM，让D3处理数学计算来对它们进一步整合。来看看下面这三行代码：

```
const times = d3.extent(data.map(d => d.year))
const range = [50, 450]

const scale = d3.time.scale().domain(times).range(range)
```

times和range都是在构造函数中配置的，并且被添加到了组件的State中。times表示我们的时间段。它包含最早和最晚的年份日期。可以使用D3的extent函数在一个数值区间中找到最大值和最小值。区间表示时间轴上以像素为单位的区间。第一个日期1879，将会放在x标尺的0像素位置，最后一个日期1964，将会放在x标尺的450像素位置。

接下来的这行代码是创建标尺的，它是一个函数，可以用来为我们的时间标尺上的任意年份插入像素值。标尺是根据传递的时间段和像素区间到D3的time.scale函数而创建的。scale函数是用来在可视化应用中为每个位于1879年和1964年之间的大事日期获取x坐标位置的。

为了代替在componentDidMount方法中创建标尺，我们可以在获取时间段和像素区间之后，在构造函数中将它添加到组件中。现在可以使用this.scale(year)在组件中的任意位置访问该标尺了：

```
constructor({data=[]}) {
    const times = d3.extent(data.map(d => d.year))
    const range = [50, 450]
    super({data})
    this.scale = d3.time.scale().domain(times).range(range)
    this.state = {data, times, range}
}
```

在componentDidMount方法内部，D3首先创建了一个SVG元素，然后将它添加到了目标引用中：

```
d3.select(target)
    .append('svg')
    .attr('height', 200)
    .attr('width', 500)
```

构造UI是React的任务。为了代替D3完成这个任务，让我创建一个Canvas组件并返回一个SVG元素：

```
const Canvas = ({children}) =>
    <svg height="200" width="500">
        {children}
    </svg>
```

接下来，D3将会选择SVG元素，即目标下的第一个子元素，并为我们的时间轴中的每个数据点添加一个group元素。添加完毕后，将会使用scale函数将x坐标值进行转换，继而对group元素进行定位：

```
group = d3.select(target.children[0])
    .selectAll('g')
    .data(data)
    .enter()
    .append('g')
    .attr(
        'transform',
        (d, i) => 'translate(' + scale(d.year) + ', 0)'
    )
```

group元素是一个DOM元素，因此我们也可以让React处理这个任务。这里有一个名为TimelineDot的组件可以用来配置group元素，可以将它们沿着x轴进行定位：

```
const TimelineDot = ({position}) =>
    <g transform={`translate(${position},0)`}></g>
```

接下来，D3将会给group添加一个圆环和一些“样式”。文本元素通过将事件年份和事件标题连接在一起显示相应的数据。然后我们可以围绕蓝色的圆环定位和旋转上述文本：

```
group.append('circle')
    .attr('cy', 160)
    .attr('r', 5)
    .style('fill', 'blue')

group.append('text')
    .text(d => d.year + " - " + d.event)
    .style('font-size', 10)
    .attr('y', 115)
    .attr('x', -95)
    .attr('transform', 'rotate(-45)')
```

需要用户完成的所有工作就是修改TimelineDot组件，引用一个圆环元素和一个从属性中接收的文本元素：

```
const TimelineDot = ({position, txt}) =>
    <g transform={`translate(${position},0)`}>

        <circle cy={160}
                r={5}
```

```
                style={{fill: 'blue'}} />

        <text y={115}
                x={-95}
                transform="rotate(-45)"
                style={{fontSize: '10px'}}>{txt}</text>

    </g>
```

React现在可以使用虚拟DOM管理UI了。D3的角色被削弱了，不过它仍然提供了一些React不具备的基本功能。它帮助我们创建了时间段和像素区间，构造了一个scale函数，以便我们可以根据年份进行像素插值。这是我们重构Timeline组件后可能达到的效果：

```
class Timeline extends Component {

    constructor({data=[]}) {
        const times = d3.extent(data.map(d => d.year))
        const range = [50, 450]
        super({data})
        this.scale = d3.time.scale().domain(times).range(range)
        this.state = {data, times, range}
    }

    render() {
        const { data } = this.state
        const { scale } = this
        return (
            <div className="timeline">
                <h1>{this.props.name}  Timeline</h1>
                <Canvas>
                    {data.map((d, i) =>
                        <TimelineDot position={scale(d.year)}
                                     txt={`${d.year} - ${d.event}`}
                        />
                    )}
                </Canvas>
            </div>
        )
    }
}
```

我们可以将任意JavaScript库与React集成。生命周期函数是其他JavaScript库进入，React库离开的地方。不过我们应该竭力避免添加管理UI的脚本库，这是React的工作。

高阶组件

高阶组件（HOC）是一个简单函数，它会接收一个React组件作为参数，然后返回另

外一个React组件。通常，HOC会使用一个能够维护State或者包含若干功能的类来包装输入的组件。高阶组件是组件之间功能复用的最佳方式。

不兼容Mixin

直到React v0.13版本，给React组件集成新功能的最佳方式都是采用Mixin技术。Mixin是通过创建一个可以被当作可配置属性的类来直接给组件添加功能的。用户仍然可以通过React.createClass使用Mixin技术，不过ES6的类和无状态函数式组件并没有为该技术提供支持。将来版本的React也不会再支持该技术。

一个HOC允许我们使用其他组件包装一个组件。父组件可以保留State或者将若干功能作为属性向下传递给合成的组件。合成的组件并不需要知道任何与HOC代码实现有关的信息，除了它提供的属性和方法的名称。

看一下PeopleList这个组件。它从一个API中载入若干随机用户，然后渲染了一个成员名称的列表。当载入用户时，会显示一个正在载入的提示信息。一旦用户载入完毕，他们将会被显示在DOM上：

```
import { Component } from 'react'
import { render } from 'react-dom'
import fetch from 'isomorphic-fetch'

class PeopleList extends Component {

    constructor(props) {
        super(props)
        this.state = {
            data: [],
            loaded: false,
            loading: false
        }
    }

    componentWillMount() {
      this.setState({loading:true})
      fetch('https://randomuser.me/api/?results=10')
          .then(response => response.json())
          .then(obj => obj.results)
          .then(data => this.setState({
                loaded: true,
                loading: false,
                data
           }))
    }

    render() {
```

```
        const { data, loading, loaded } = this.state
        return (loading) ?
            <div>Loading...</div> :
            <ol className="people-list">
                {data.map((person, i) => {
                    const {first, last} = person.name
                    return <li key={i}>{first} {last}</li>
                })}
            </ol>
    }
}

render(
    <PeopleList />,
    document.getElementById('react-container')
)
```

PeopleList集成了一个来自jQuery的getJSON调用，以便能够从JSON API获取人员信息。当该组件被渲染后，它会显示一个正在载入数据的提示信息，或者根据loading的值是否为true显示一组姓名列表。

如果我们可以利用这种加载功能，那么就可以在组件之间复用它。我们可以创建一个高阶组件DataComponent，用来在创建React组件时加载数据。为了使用DataComponent组件，我们剥离了PeopleList的State，创建了一个通过props接收数据的无状态函数式组件：

```
import { render } from 'react-dom'

const PeopleList = ({data}) =>
    <ol className="people-list">
        {data.results.map((person, i) => {
            const {first, last} = person.name
            return <li key={i}>{first} {last}</li>
        })}
    </ol>

const RandomMeUsers = DataComponent(
                    PeopleList,
                    "https://randomuser.me/api/"
                )

render(
    <RandomMeUsers count={10} />,
    document.getElementById('react-container')
)
```

现在我们可以创建一个RandomMeUsers组件，它总是能够从同一数据源*randomuser.me*载入和显示用户数据。用户必须提供的内容仅仅是希望载入的用户数量。数据处理的工作被迁移到了HOC中，处理UI的工作是由PeopleList组件完成的。HOC提供了数

据载入State，以及载入数据和修改器自身State的机制。当载入数据时，HOC会向用户显示一个正在载入数据的信息提示。一旦数据载入完毕，HOC会挂载PeopleList组件，并将人员数目作为数据属性传递给它：

```
const DataComponent = (ComposedComponent, url) =>
    class DataComponent extends Component {
        constructor(props) {
            super(props)
            this.state = {
                data: [],
                loading: false,
                loaded: false
            }
        }

        componentWillMount() {
            this.setState({loading:true})
            fetch(url)
                .then(response => response.json())
                .then(data => this.setState({
                    loaded: true,
                    loading: false,
                    data
                }))
        }

        render() {
            return (
                <div className="data-component">
                    {(this.state.loading) ?
                        <div>Loading...</div> :
                        <ComposedComponent {...this.state} />}
                </div>
            )
        }
    }
```

注意，DataComponent实际上是一个函数。所有高阶组件都是函数。ComposedComponent是我们希望包装的组件。返回的类DataComponent，主要用于存储和管理State。当State发生变化并且数据载入完毕。ComposedComponent会被渲染，并且上述数据会作为属性传递给它。

HOC可以用来创建任意类型的数据组件。让我们看看如何复用DataComponent，将之用于创建CountryDropDown组件，该组件会从*restcountries.eu* API获得的全球数据，根据每个国家的人口数量生成相应的国家名称：

```
import { render } from 'react-dom'

const CountryNames = ({data, selected=""}) =>
    <select className="people-list" defaultValue={selected}>
```

```
        {data.map(({name}, i) =>
            <option key={i} value={name}>{name}</option>
        )}
    </select>

const CountryDropDown =
    DataComponent(
        CountryNames,
        "https://restcountries.eu/rest/v1/all"
    )

render(
    <CountryDropDown selected="United States" />,
    document.getElementById('react-container')
)
```

CountryNames组件根据props获得了国家名称。DataComponent主要负责加载数据和传递每个国家的信息。

注意，CountryNames组件还包含一个selected属性。这个属性将会导致该组件将“United States”作为默认选项。不过，现在来看，它还无法正常工作。我们还没有从HOC组件中将属性传递给合成组件。

接下来我们将修改HOC，使得它可以向下传递任意属性给合成组件：

```
render() {
    return (
        <div className="data-component">
            {(this.state.loading) ?
                <div>Loading...</div> :
                <ComposedComponent {...this.state}
                                   {...this.props} />
            }
        </div>
    )
}
```

现在HOC将State和props向下传递给合成组件了。如果我们运行这些代码，会发现CountryDropDown预置了“United States”作为默认项。

让我看看另外一个HOC。本章前面的章节曾经介绍过HiddenMessage组件。显示或者隐藏内容的能力也有值得复用的地方。接下来的示例中，我们介绍的可扩展HOC在功能方面和HiddenMessage类似。用户可以根据布尔属性的切换来显示或者隐藏内容。该HOC还提供了一种切换已折叠属性值的机制（参见示例7-3）。

示例 7-3：./components/hoc/Expandable.js

```
import { Component } from 'react'
```

```
const Expandable = ComposedComponent =>
    class Expandable extends Component {

        constructor(props) {
            super(props)
            const collapsed =
                (props.hidden && props.hidden === true) ?
                    true :
                    false
            this.state = {collapsed}
            this.expandCollapse = this.expandCollapse.bind(this)
        }

        expandCollapse() {
            let collapsed = !this.state.collapsed
            this.setState({collapsed})
        }

        render() {
            return <ComposedComponent
                        expandCollapse={this.expandCollapse}
                        {...this.state}
                        {...this.props} />
        }
    }
```

可扩展HOC会接收一个ComposedComponent作为参数，然后使用State和一些功能对它进行包装，使得它可以显示或者隐藏内容。其实，折叠的State是使用输入的属性或者默认参数值false进行设置的。折叠后的State将作为属性向下传递给ComposedComponent。

该组件还有一个名为expandCollapse的方法用于切换已折叠State。该方法还可以向下传递给ComposedComponent。它一旦被触发，将会修改已折叠的State，然后使用新的State更新ComposedComponent。

如果DataComponent的属性被某个父组件修改了，该组件将会更新已折叠State，然后将新的State作为属性向下传递给ComposedComponent。

最后，所有State和props将会被向下传递给ComposedComponent。现在我们可以使用这个HOC创建若干新的组件。首先，让我使用它创建一个隐藏信息的组件，本章前面的章节已经对该组件做了详细介绍：

```
const ShowHideMessage = ({children, collapsed, expandCollapse}) =>
    <p onClick={expandCollapse}>
        {(collapsed) ?
            children.replace(/[a-zA-Z0-9]/g, "x") :
            children}
    </p>
```

```
const HiddenMessage = Expandable(ShowHideMessage)
```

这里我们创建了一个HiddenMessage组件，当collapsed属性值为true时，将会使用“x”替换字符串中的每个字母或数字。当collapsed的属性值为false时，将会显示这些消息。读者可以尝试在本章前面章节介绍的HiddenMessages组件中使用这个HiddenMessage组件。

接下来将使用相同的HOC创建一个按钮，用于显示和隐藏某个div中的内容。在接下来的示例中，MenuButton可以用于创建PopUpButton，它是一个可以控制内容显示和隐藏的组件：

```
class MenuButton extends Component {

    componentWillReceiveProps(nextProps) {
        const collapsed =
            (nextProps.collapsed && nextProps.collapsed === true) ?
                true :
                false
        this.setState({collapsed})
    }

    render() {
        const {children, collapsed, txt, expandCollapse} = this.props
        return (
            <div className="pop-button">
                <button onClick={expandCollapse}>{txt}</button>
                {(!collapsed) ?
                    <div className="pop-up">
                        {children}
                    </div> :
                    ""
                }
            </div>
        )
    }
}

const PopUpButton = Expandable(MenuButton)

render(
    <PopUpButton hidden={true} txt="toggle popup">
        <h1>Hidden Content</h1>
        <p>This content will start off hidden</p>
    </PopUpButton>,
    document.getElementById('react-container')
    )
```

PopUpButton是由MenuButton组件创建的。它将通过已折叠State和函数一起作为属性修改MenuButton的State。当用户单击按钮时，这将会触发expandCollapse并切换已折

叠State。当State被折叠了，我们只能看到一个按钮。当它被展开时，我们可以看到一个按钮和一个包含隐藏内容的div。

高阶组件是一种极佳的功能复用方式，并且能够将组件State和生命周期管理的细节封装起来。它允许用户构建更多无状态函数式组件，以便可以一心一意地管理UI。

在React之外管理State

State管理在React中是一种非常好的机制。我们可以使用React内置的State管理系统构建大量的应用程序。不过，随着应用程序规模的不断扩张，State管理工作开始让我们焦头烂额了。坚持让State存放在组件树的根节点的某个位置将会有助于减轻用户的工作负担，不过即使如此，用户的应用程序膨胀到一定规模时，最明智的做法是将State数据和自身的层级隔离，独立于UI。

在React外部管理State的优点之一就是它将减少大量类组件的使用。如果没有使用State，那么很容易就可以确保用户的大部分组件是无状态的。当需要使用生命周期函数时，用户只需要创建一个类即可，甚至可以将类的功能和HOC隔离，从而确保组件只包含无状态的UI。无状态函数式组件更容易理解和测试。它们是纯函数，因此非常适合构造严格意义上的函数式应用程序。

在React之外管理State意味着很多不同的事情。用户可以把React和Backbone模型搭配使用，或者其他任意MVC库的模型State。用户可以创建专属于自己的State管理系统。甚至可以使用全局变量、本地存储和JavaScript纯文本管理State。在React之外管理State，简单的理解就是不在应用程序中使用React的State或者setState方法。

渲染一个时钟

在第3章中，我们创建了一个遵循函数式编程范式的时钟程序。整个应用程序是由函数和高阶函数组成，它们又经过进一步合成组成了startTicking函数。它会启动时钟，并在控制台上显示时间：

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

```
        ),
        oneSecond()
    )

startTicking()
```

但是不在控制台显示时钟信息，如果我们在浏览器中显示它又该怎么办呢？我们可以构造一个React组件，将时钟的时间信息显示在一个div中：

```
const AlarmClockDisplay = ({hours, minutes, seconds, ampm}) =>
    <div className="clock">
        <span>{hours}</span>
        <span>:</span>
        <span>{minutes}</span>
        <span>:</span>
        <span>{seconds}</span>
        <span>{ampm}</span>
    </div>
```

该组件会接收小时、分钟、秒和当天日期等属性参数。然后它会创建一个可以显示这些属性的DOM。

我们可以使用render方法替换log方法，然后使用我们的组件渲染本地时间，如果时间值小于10，会在前面用0补足双位：

```
const startTicking = () =>
    setInterval(
        compose(
            getCurrentTime,
            abstractClockTime,
            convertToCivilianTime,
            doubleDigits,
            render(AlarmClockDisplay)
        ),
        oneSecond()
    )

startTicking()
```

render方法将会需要成为一个高阶函数。当startTicking函数被合成并添加到它的内部时，它将需要接收AlarmClockDisplay函数作为参数初始化属性。最终，它将需要使用上述组件渲染每秒格式化后的时间：

```
const render = Component => civilianTime =>
    ReactDOM.render(
        <Component {...civilianTime} />,
        document.getElementById('react-container')
    )
```

render方法的高阶函数每秒都会触发一次ReactDOM.render方法，同时更新DOM。该方法吸收了React快速渲染DOM的优势，但是不需要用到一个带State的组件类。

该应用程序的State是在React之外管理的。React允许我们通过提供自己的高阶函数使用ReactDOM.render方法渲染一个组件，继而达到保留原有的函数式架构的目的。在React之外管理State并不是必须的，只是让用户多了一个选择。React是一个脚本库，由用户决定如何以最佳的方式在自己的应用程序中使用它。

接下来，我们将介绍Flux，它是一种设计模式，被视为在React中管理State的一种替代性方案。

Flux

Flux是Facebook开发的一种设计模式，旨在保持数据单向流动。在Flux诞生之前，Web开发架构由多种MVC设计模式的变体所主导。Flux是MVC的替代品，是一种完全不同的设计模式，并且与函数式编程范式相辅相成。

React或者Flux和函数式JavaScript又有什么关系呢？对于初学者来说，一个无状态函数式组件就是一个纯函数。它通过属性获取操作指令并返回UI元素。一个React类使用State或者props作为输入，同时也会构造UI元素。若干React组件可以进一步合成单个组件。不可变数据提供组件作为输入，然后返回的UI元素作为输出：

```
const Countdown = ({count}) => <h1>{count}</h1>
```

Flux为我们提供一种Web应用架构，可以把它视为React的有益补充。具体来说，Flux提供了一种方法可以为React创建UI将要用的数据提供支持。

在Flux中，应用程序的State数据是存放在React组件外部的Store进行管理的。Store保留或者修改数据，是唯一可以更新Flux视图的办法。如果某个用户准备和一个Web页面交互（单击一个按钮或者提交一个表单），然后一个Action会被创建用于表示用户的请求。一个Action会提供一组操作指令和需要变更的目标数据。Action是使用一个名为Dispatcher的中央控制组件分发的。Dispatcher的主要用途是将我们的Action排队，然后将它们分发到相应的Store中。一旦Store接收到了一个Action，会使用它作为操作指令修改State和更新View。数据流是单向的：Action到Dispatcher，然后是Store，最后到达View（见图7-7）。

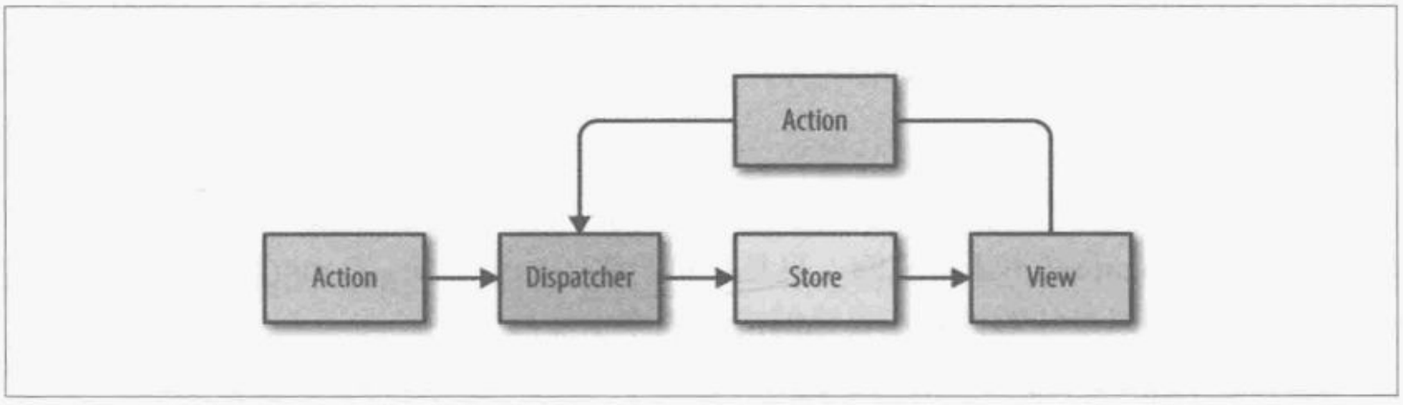


图7-7：Facebook的Flux设计模式

在Flux中，Action和State都是不可变数据。Action可以来自某个View，或者来自其他源地址，一般是一个Web服务器。

每次变更都需要一个Action。每个Action提供制造变更的一组指令集。Action也可以被当作菜谱，告诉我们变更的内容是什么，创建变更时用到了哪些数据，以及Action的源头。用户唯一可以变更的就是Store。Store更新了数据，View在UI上渲染这些更新，而Action会告知我们发生变更的过程和原因。

将用户的应用程序的数据流限制在这种设计模式下，将会使应用程序更易于修复和扩展。来看看图7-8中的应用程序。我们会发现每个被分发的Action都被记录到控制台上。这些Action告知我们当前UI显示大号数字3的具体流程。

如你所见，导致最后一次State变更的是一个TICK。它将之前的计数值修改为现在的计数值3，看上去之前的计数值应该是4。这些Action告诉我们State的变化过程。我们可以按图索骥，向上追溯到源头发现首次变更的计数值是9，因此该程序大概是一个从10开始的倒计时程序。

来看看这个倒计时应用程序是如何使用Flux设计模式构建的。我们将会介绍设计模式的每个部分，并讨论它们对倒计时应用中单向数据流的贡献。

View

让我们先从View开始，即一个React无状态组件。Flux将会管理应用程序的State，因此除非用户需要使用一个生命周期函数，否则将不会用到类组件。

倒计时应用的View会将计数作为属性获取。它还会接收一对函数：`tick`和`reset`。

图7-8：采用Flux架构的倒计时程序

```
const Countdown = ({count, tick, reset}) => {

  if (count) {
    setTimeout(() => tick(), 1000)
  }

  return (count) ?
      <h1>{count}</h1> :
      <div onClick={() => reset(10)}>
          <span>CELEBRATE!!!</span>
          <span>(click to start over)</span>
      </div>

}
```

当这个View渲染它之后会显示计数值，除非该计数值为0，否则将会显示一条信息引导用户单击“CELEBRATE”按钮。如果计数值不是0，那么该View将会设定一个超时时间，等待一秒后，调用TICK函数。

当计数值为0时，该View将不会被任何其他Action生成器触发，直到某个用户单击了主div触发了一个重置操作。该重置会将计数值设置为10，再次启动整个倒计时过程。

组件中的State

使用Flux并不意味着用户不能在自己的View组件中管理State。这意味着应用程序State并不是由你的View组件管理的。比如，Flux可以管理日期和时间来构造

时间轴应用。使用包含内部State的时间轴组件来可视化用户应用程序的时间轴并不存在什么限制。

应该谨慎地使用State，只在必要的情况下。通过可复用组件内部管理它们自身的State。应用程序其余的部分并不需要“了解”子组件的State。

Action和Action生成器

Action提供的指令和数据主要的用途是Store用来修改State的。Action生成器就是函数，主要用来封装构造某个Action的具体细节。Action本身是由若干对象构成，并且至少包含一个类型字段。Action类型一般是通过一个由大写字母组成的字符串进行描述的。此外，Action也可能打包了任何Store所需的数据。比如：

```
const countdownActions = dispatcher =>
    ({
        tick(currentCount)  {
            dispatcher.handleAction({ type: 'TICK' })
        },
        reset(count) {
            dispatcher.handleAction({
                type: 'RESET',
                count
            })
        }
    })
```

当倒计时Action生成器被载入时，dispatcher会作为一个参数传递给它。每次某个TICK或者RESET函数被调用时，dispatcher的`handleAction`方法也会被调用，以便“调度”Action对象。

Dispatcher

永远只有一个Dispatcher，它表示设计模式中的空中交通管理中心。Dispatcher接收到Action，将与之有关的某些生成源信息一并打包，然后将它发送到相应的Store或者一系列Store，以便处理这个Action。

虽然Flux不是一个框架，Facebook研发了一个开源的`Dispatcher`类供用户使用。不同Dispatcher的具体实现一般都是严格遵循规范的，因此使用Facebook的Dispatcher要好过用户自己动手编写的类似实现：

```
import Dispatcher from 'flux'

class CountdownDispatcher extends Dispatcher {
```

```
    handleAction(action) {
        console.log('dispatching action:', action)
        this.dispatch({
            source: 'VIEW_ACTION',
            action
        })
    }

}
```

当handleAction被某个Action调用时，它会和该Action起始位置的某些数据一起被分发。当某个Store被创建后，它会被Dispatcher登记注册并开始监听相关的Action。当某个Action被分发后，它会按照一定的次序被处理接收，然后发送到相应的Store中。

Store

Store主要用于存放应用程序逻辑和State数据的若干对象。Store和MVC模式中的模型类似，不过Store并没有受限于在单个对象中管理数据。可能在构建Flux应用程序时，存在使用单个Store管理多种不同数据类型的情况。

当前的State数据可以通过访问Store的属性获取。某个Store需要修改State数据的所有操作指令都是由Action提供的。Store将会按照类别处理Action，并修改相关的数据。一旦数据被修改了，该Store将会发出一个事件通知任何订阅了该Store的View，它们的数据发生了变化。让我们来看一个示例：

```
import { EventEmitter } from 'events'

class CountdownStore extends EventEmitter {

    constructor(count=5, dispatcher) {
        super()
        this._count = count
        this.dispatcherIndex = dispatcher.register(
            this.dispatch.bind(this)
        )
    }
    get count() {
        return this._count
    }

    dispatch(payload) {
        const { type, count } = payload.action
        switch(type) {

            case "TICK":
                this._count = this._count - 1
                this.emit("TICK", this._count)
                return true
```

```
            case "RESET":
                this._count = count
                this.emit("RESET", this._count)
                return true

        }
    }

}
```

Store会保存倒计时应用程序的State，即计数值。计数值可以通过一个只读属性访问。当Action被分发后，Store会使用它们来修改计数值。一个TICK Action会减少计数值。一个REST Action会重置整个计数值，以及该Action引用的所有数据。

一旦State发生了变化，Store会发起一个事件到任何正在监听的View。

综合应用

现在读者应该了解了数据流在Flux应用程序的每个部分是如何运作的，接下来看看如何将这些部分连接到一起：

```
const appDispatcher = new CountdownDispatcher()
const actions = countdownActions(appDispatcher)
const store = new CountdownStore(10, appDispatcher)

const render = count => ReactDOM.render(
    <Countdown count={count} {...actions} />,
    document.getElementById('react-container')
)

store.on("TICK", () => render(store.count))
store.on("RESET", () => render(store.count))
render(store.count)
```

首先，我们创建了appDispatcher，然后我们使用appDispatcher生成我们的Action生成器。最后，appDispatcher被注册到了Store中，并且Store将初始的计数值设为10。

render方用于渲染包含计数值的View，该计数值是通过参数进行传递的。同时还有Action生成器也作为属性被传递给了该View。

最后，某些监听器被添加到了Store中，从而完成整个循环流程。当Store发起了一个TICK或者RESET，它产生了一个新的计数，因此需要马上在View中进行渲染。然后，初始View会根据Store中的计数值进行渲染。每次View发起一个TICK或者RESET时，该Action将会沿着循环节点发送，最终作为准备重新渲染的数据返回该View。

Flux的实现

实现Flux模式的方法种类繁多。一些库基于这种设计模式的特定实现已经开源了。下面是一些值得关注的Flux模式的具体实现：

Flux(https://facebook.github.io/flux/)

Facebook的Flux是我们前面已经介绍过的设计模式。该Flux库包含一个Dispatcher的实现。

Reflux(https://github.com/reflux/refluxjs)

单向数据流的简化版实现，主要聚焦于Action、Store和View。

Flummox(http://acdlite.github.io/flummox)

一个Flux模式的具体实现，允许用户通过扩展JavaScript类来构建Flux模块。

Fluxible(http://fluxible.io)

一个由Yahoo创建的Flux框架，用于同构Flux应用。同构应用的细节将会在第12章详细介绍。

Redux(http://redux.js.org)

一个类Flux库，用函数取代对象来实现模块化。

MobX(https://mobx.js.org/getting-started.html)

一个State管理库，使用观察检测来响应State中的变化。

所有这些实现都拥有Store、Action和一种分发机制，并且将React组件作为视图层。它们都是Flux设计模式的衍生物，其核心都是和单向数据流有关的。

Redux很快就成了最受欢迎的Flux框架之一。下一章将会介绍如何使用Redux为客户端应用构建功能数据架构。

第8章

Redux

Redux（*http://redux.js.org*）已经毋庸置疑地成为了众多Flux或类Flux库中的翘楚之一。Redux是基于Flux的，旨在处理应用程序中数据流变化的问题。Redux是由Dan Abramov（*https://github.com/gaearon*）和Andrew Clark一起开发的。因为创建了Redux的缘故，他们都被邀请加入了Facebook的React研发团队。

当Andrew Clark开始帮助Dan研发Redux时，他同时还在研发第4版的Flummox，它是另外一款基于Flux的脚本库。Flummox的npm页面（*https://www.npmjs.com/package/flummox*）这样说过：

> 4.x应该是最后一个主版本，但是目前来看永远不可能了。如果希望使用最新的特性，请改用Redux，它真的很棒。[注1]

Redux库的尺寸小得惊人，只有99行代码(*http://bit.ly/2nawjzD*)。

如前所述，Redux即类Flux的脚本库，但是它不完全是Flux。它包含Action、Action生成器、Store，以及用于修改State的Action对象。

Redux通过移除Dispatcher,对Flux的概念进行了一些简化，并使用单个不可变对象表示应用程序的State。Redux还引入了Reducer，它并不是Flux模式中的内容。Reducer是纯函数，它会根据当前的State和Action返回一个新的State：`(state, action) => newState`。

注1： Flummox documentation(https://github.com/acdlite/flummox)。

State

将State存放于一处统一管理的想法并不疯狂。事实上，我们在上一章中就是这么做的。我们将它存放于应用程序的根组件上。在纯React或者Flux应用中，比较推荐的做法是将State尽量存放在少数几个对象中。在Redux中，这是一条规则。[注2]

当你听到必须将State存放在一个地方时，可能会觉得那是个不合理的需求，特别是当你需要管理多种数据时。接下来看看如何在一个包含多种数据的应用程序中实现这个目标。我们将会考察一个社交媒体应用程序，其State分布于不同组件中（见图8-1）。该应用程序本身包含用户的State。所有的短信都存储在上述State之中。每条短信包含自己的State，所有帖子都存放在posts组件中。

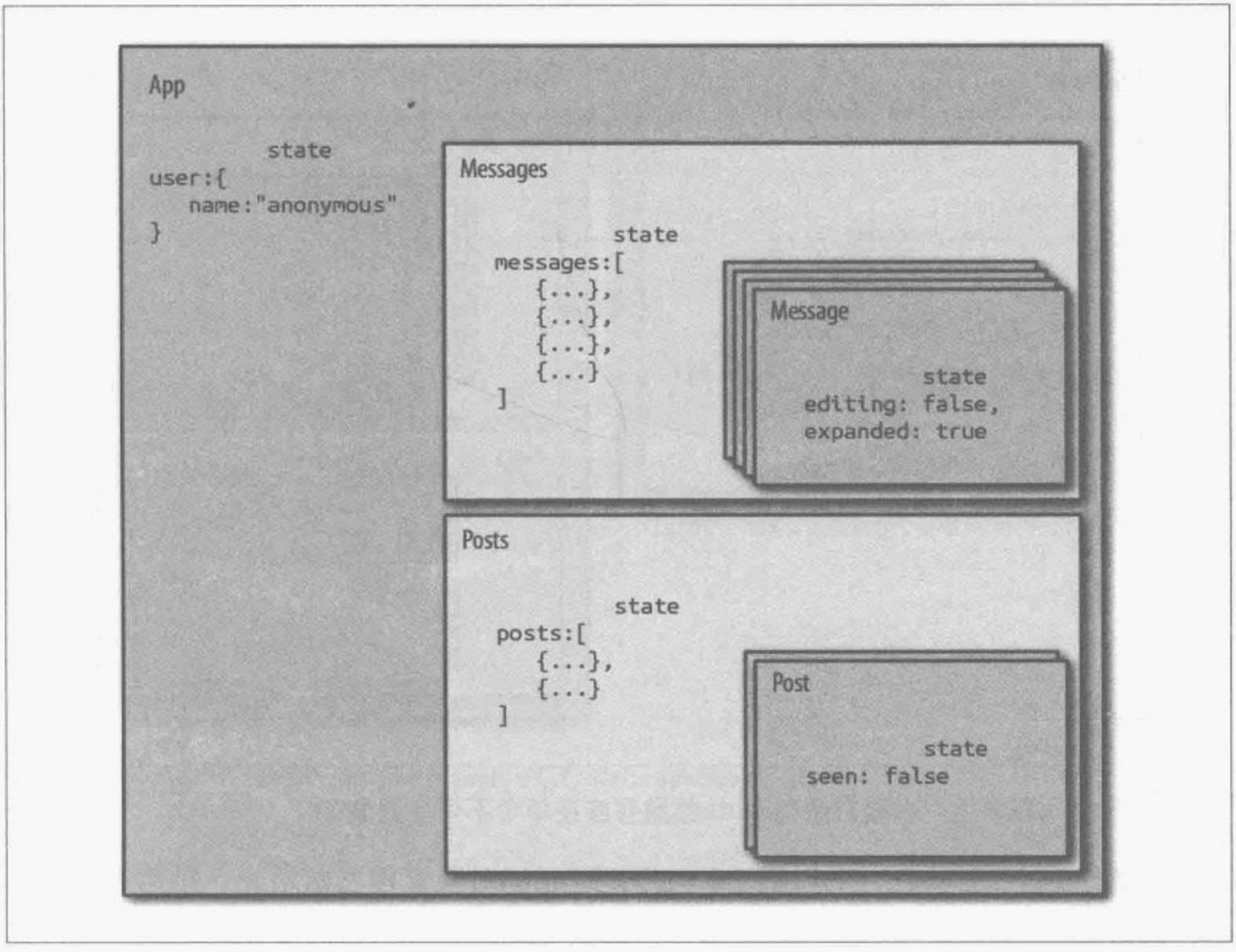


图8-1：组件包含自己的State的React应用程序

注2：Redux Docs，“Three Principles”（http://bit.ly/2mJ0U4Y）。

类似上述结构的应用程序可能会运行良好，不过随着程序规模的增长，管理应用程序的整体State可能会变得愈加困难。考虑到每个组件都将使用其内部的setState方法来改变自身的State，可能了解更新命令源也会变得很麻烦。

哪些短信被转发了？哪些帖子被阅读了？为了了解这些细节，我们必须深入组件树，向下追踪独立组件中的State。

Redux通过要求用户将所有State数据存储在单个对象，简化了我们在应用程序中查看State的方式。我们需要了解的应用程序信息都在一个地方：真实单一数据源。我们可以使用Redux通过将所有State数据移动到单个位置来构造相同的应用程序（见图8-2）。

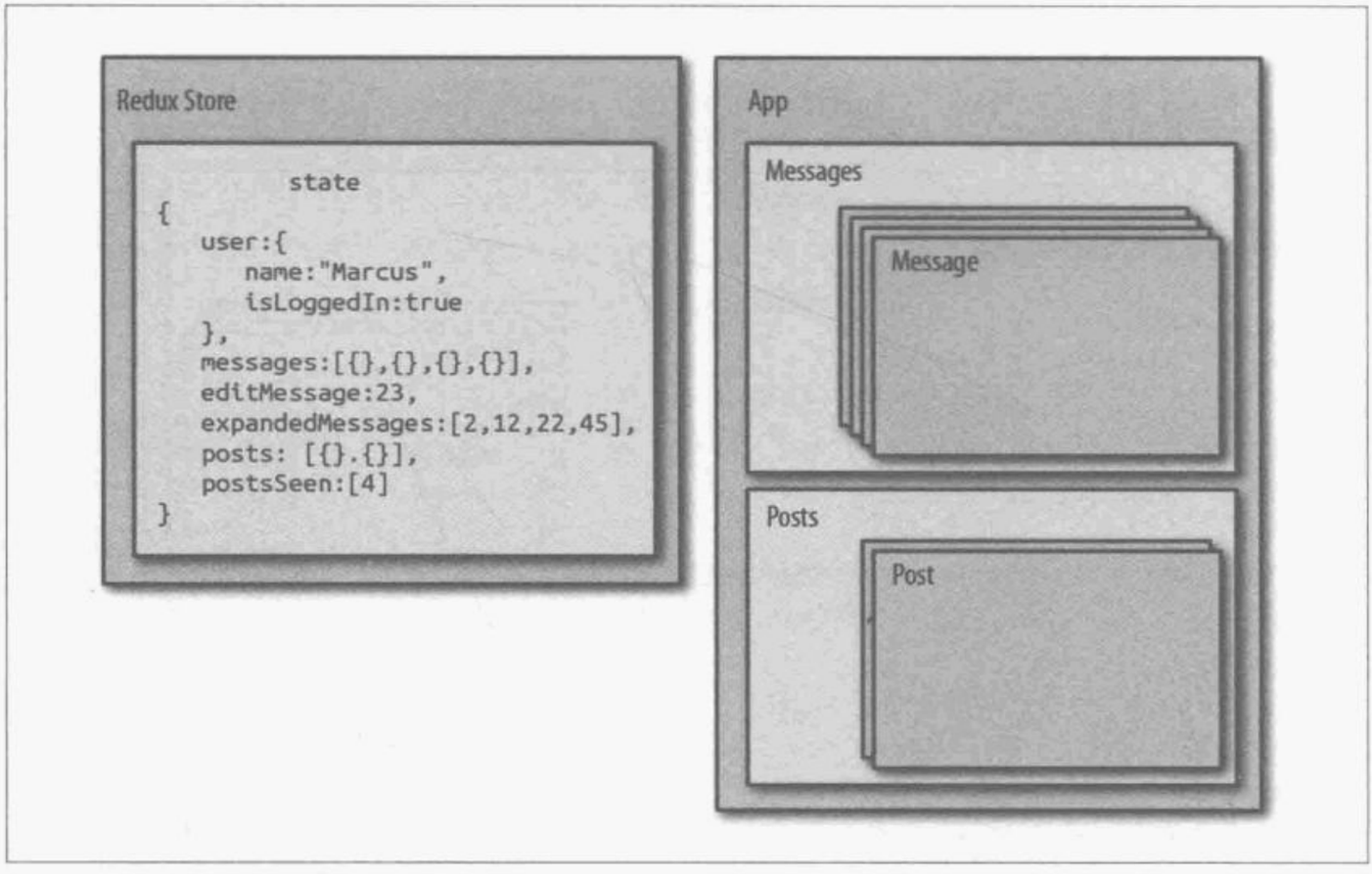


图8-2：Redux要求用户必须把所有State数据存放在单个不可变对象中

在社交媒体应用程序中，可以看到我们是通过相同对象管理当前用户、短信和帖子的State的。这个对象存储的信息甚至包括哪些短信正在被编辑，哪些短信被转发了，以及哪些帖子被阅读了。这些信息是从包含ID和引用特定记录的数组中获取的。所有短信和帖子都被缓存到了State对象中，因此数据就在那里。

通过Redux，我们将State管理和React完全剥离，Redux将会管理这些State。

在图8-3中，我们可以看到社交媒体的State树。从图中可知，我们有若干短信在一个数组中，类似的情况还包括主题帖子。我们需要的所有数据都植根于一个对象：State树。单个对象中的每个键表示State树的一个分支。

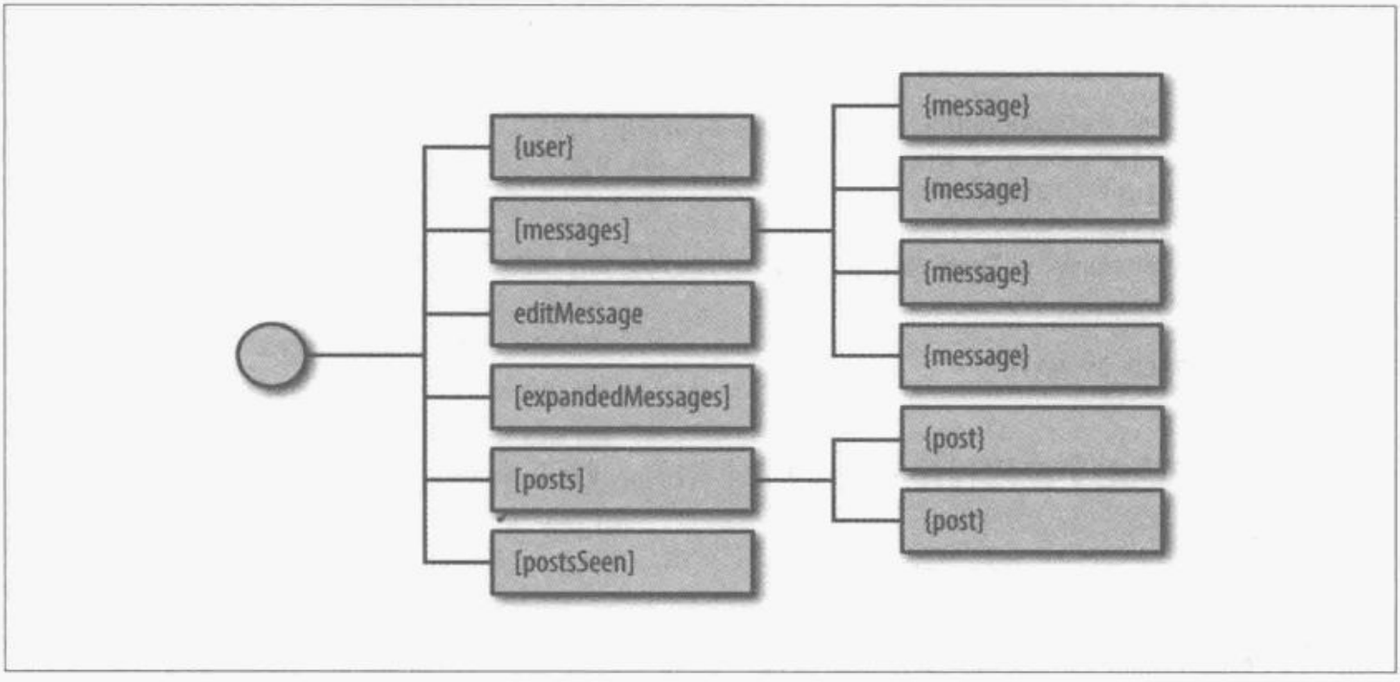


图8-3：State树示例

在构建Redux应用时，用户首先需要考虑的事情就是State树。尝试在单个对象中定义State。使用一些占位符数据草拟一个State树的JSON示例是一个非常好的习惯。

让我们再次回顾一下颜色管理器程序。在这个应用程序中，我们将会获取每种颜色存储在数组中的信息，以及这些颜色应该如何存储的信息。我们的State数据的示例将会和示例8-1类似。

示例8-1：颜色管理器应用程序State示例

```
{
    colors: [
        {
            "id": "8658c1d0-9eda-4a90-95e1-8001e8eb6036",
            "title": "Ocean Blue",
            "color": "#0070ff",
            "rating": 3,
            "timestamp": "Sat Mar 12 2016 16:12:09 GMT-0800 (PST)"
        },
        {
            "id": "f9005b4e-975e-433d-a646-79df172e1dbb",
            "title": "Tomato",
            "color": "#d10012",
            "rating": 2,
            "timestamp": "Fri Mar 11 2016 12:00:00 GMT-0800 (PST)"
```

```
        },
        {
            "id": "58d9caee-6ea6-4d7b-9984-65b145031979",
            "title": "Lawn",
            "color": "#67bf4f",
            "rating": 1,
            "timestamp": "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
        },
        {
            "id": "a5685c39-6bdc-4727-9188-6c9a00bf7f95",
            "title": "Party Pink",
            "color": "#ff00f7",
            "rating": 5,
            "timestamp": "Wed Mar 9 2016 03:26:00 GMT-0800 (PST)"
        }
    ],
    sort: "SORTED_BY_DATE"
}
```

现在我们已经确定了应用程序State的基本结构，接下来看看如何通过Action更新和修改这些State。

Action

在上一节中，我们介绍了一个非常重要的Redux规则：应用程序的State应该存放在单个不可变对象中。不可变意味着这个State对象无法变更。最终我们将会通过整体替换的方式更新这个State对象。为此，我们需要获得具体的变更指令。这也是Action提供的内容：需要变更应用程序State的具体指令，以及执行这些变更必需的数据。[注3]

Action是更新Redux应用程序State的唯一方式。Action为我们提供了应该变更哪些内容的指令，但是我们还可以将它们看作菜谱，其内容是随着时间变化的历史操作记录。如果用户准备移除3种颜色，添加4种颜色，然后对5种颜色进行评分，那么这将如图8-4所示留下一系列线索信息。

通常来说，当我们准备构造一个面向对象应用程序时，首先要确定对象，它们的属性，以及如何让它们一起协作。在我们的脑海里，这种情况是名词式的。当构建一个Redux应用程序时，我们希望将思维方式转换成动词式的。Action将会如何影响State数据？一旦确定了Action，就可以在一个名为*constants.js*的文件中将它们一一列出（参见示例8-2）。

注3： Redux Docs, “Actions” (http://bit.ly/2m09uit)。

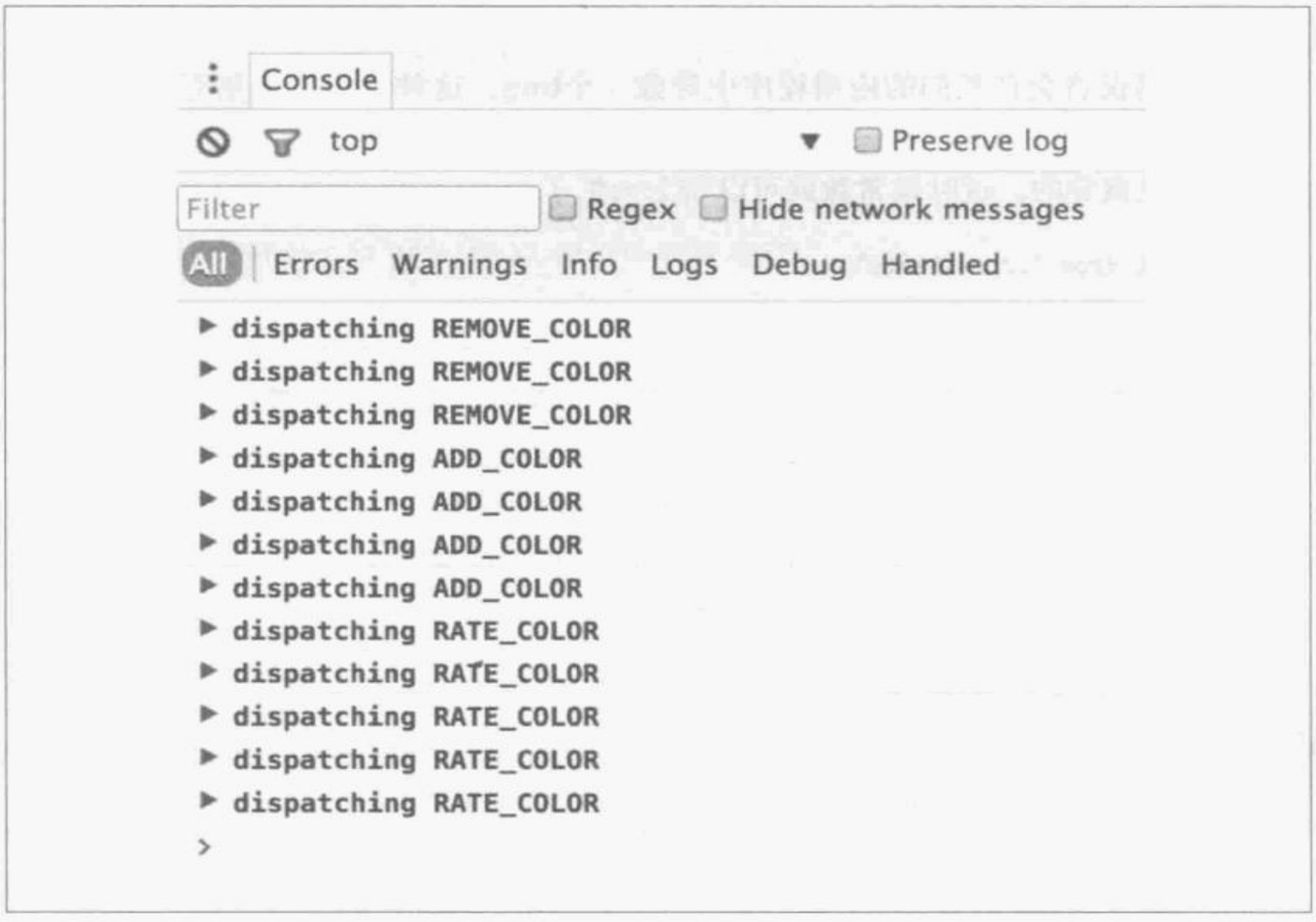


图8-4：Action的分发记录被输出到控制台

示例8-2：/constants.js中常量列表

```
const constants = {
    SORT_COLORS: "SORT_COLORS",
    ADD_COLOR: "ADD_COLOR",
    RATE_COLOR: "RATE_COLOR",
    REMOVE_COLOR: "REMOVE_COLOR"
}
export default constants
```

在这种情况下的颜色管理器中，用户将能够添加一种颜色，对一种颜色评分、移除一种颜色，以及给颜色列表排序。这里我们为每种Action类型都定义了一个字符串值。一个Action就是一个JavaScript对象，它至少包含一个类型字段：

```
{ type: "ADD_COLOR" }
```

Action类型是一个字符串，它定义了将会发生什么操作。ADD_COLOR表示该Action将会在应用程序State中，添加一种新颜色到我们的颜色列表中。在使用这些字符串创建Action时非常容易产生拼写错误：

```
{ type: "ADD_COOLOR" }
```

上述拼写错误将会在我们的应用程序中导致一个bug。这种错误通常并不会触发任何警告信息，用户只是不会看到对State数据的预期变更。如果用户犯了类似错误，那么它们是很难察觉的。这时候常数就可以帮上大忙了：

```
import C from "./constants"

{ type: C.ADD_COLOR }
```

这种方法声明了相同的Action，不过使用了一个JavaScript常量来代替字符串。JavaScript变量中的拼写错误将会导致浏览器抛出异常。将Action定义为常量也使得用户能够充分利用IDE工具的智能提示和代码自动补全功能。当输入常量的首个或者前两个字母时，IDE将会为用户自动补全其余的部分。常量的使用并不是必需的，但是采用这种编码习惯也不失为一个好的主意。

Action类型命名规范

Action类型，比如ADD_COLOR或者RATE_COLOR，只是字符串，因此从技术的角度来说，用户可以随意地对一个Action进行命名。通常来说，Action类型是由大写字母组成的，并使用下划线代替空格。同时用户还需要明确说明该Action的目的。

Action的有效载荷数据

Action是以JavaScript语法的形式提供变更某个State所需的一系列指令的。大部分State变更还需要一些数据。哪些记录应该被移除？某个新记录中应该添加哪些新的信息？

我们将此类数据称为Action的有效载荷。比如，当我们分发了一个类似RATE_COLOR的Action后，将需要知道被评分的颜色和如何给该颜色评分。这些信息可以直接以相同的JavaScript语法进行传递（参见示例8-3）。

示例8-3：RATE_COLOR Action

```
{
    type: "RATE_COLOR",
    id: "a5685c39-6bdc-4727-9188-6c9a00bf7f95",
    rating: 4
}
```

示例8-3包含一个Action类型RATE_COLOR，以及将特定颜色评分必需的数据4。

当我们添加新颜色时，将会需要知道被添加颜色的详细信息（参见示例8-4）。

示例8-4：ADD_COLOR Action

```
{
    type: "ADD_COLOR",
    color: "#FFFFFF",
    title: "Bright White",
    rating: 0,
    id: "b5685c39-3bdc-4727-9188-6c9a33df7f52",
    timestamp: "Sat Mar 12 2016 16:12:09 GMT-0800 (PST)"
}
```

这个Action告诉Redux添加一个名为Bright White的新颜色到State中。这个新颜色的所有信息都已经包含到了Action中。Action是小而美的数据包，能够方便地告知Redux如何修改State。它们还包括任何与Redux创建变更有关的数据。

Reducer

我们整个State树都是存储在单个对象中的。一个潜在的问题可能是它模块化的程度不够，比如可能用户打算用模块化的方式描述对象。Redux是通过函数进行模块化的。函数被用来更新部分State树中的内容。这些函数被称为Reducer。[注4]

Reducer就是一类函数，将当前的State和Action作为参数传入其中，通过这些参数创建和返回一个新的State。Reducer主要是用来更新State树的特定部分的，其中既包括叶子节点，也包括分支。然后我们可以将多个Reducer合成一个Reducer，来处理应用程序中任意给定Action的所有State更新。

颜色管理器程序在单个树对象中存储了所有State数据（参见示例8-5）。如果我们希望在这个应用程序中使用Redux，可以为State树中某个特定目标的叶子和分支节点创建若干Reducer。

示例8-5：颜色管理器应用中的State示例

```
{
    colors: [
        {
            "id": "8658c1d0-9eda-4a90-95e1-8001e8eb6036",
            "title": "Ocean Blue",
            "color": "#0070ff",
            "rating": 3,
            "timestamp": "Sat Mar 12 2016 16:12:09 GMT-0800 (PST)"
        },
        {
            "id": "f9005b4e-975e-433d-a646-79df172e1dbb",
            "title": "Tomato",
            "color": "#d10012",
```

注4：Redux Docs, "Reducers"(http://redux.js.org/docs/basics/Reducers.html)。

```
            "rating": 2,
            "timestamp": "Fri Mar 11 2016 12:00:00 GMT-0800 (PST)"
        },
        {
            "id": "58d9caee-6ea6-4d7b-9984-65b145031979",
            "title": "Lawn",
            "color": "#67bf4f",
            "rating": 1,
            "timestamp": "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
        },
        {
            "id": "a5685c39-6bdc-4727-9188-6c9a00bf7f95",
            "title": "Party Pink",
            "color": "#ff00f7",
            "rating": 5,
            "timestamp": "Wed Mar 9 2016 03:26:00 GMT-0800 (PST)"
        }
    ],
    sort: "SORTED_BY_DATE"
}
```

上述State数据中包含两个主要分支：colors和sort。sort分支是叶子节点。因为它不包含任何子节点。colors分支存储了多种颜色。每个颜色对象表示一个叶子节点（见图8-5）。

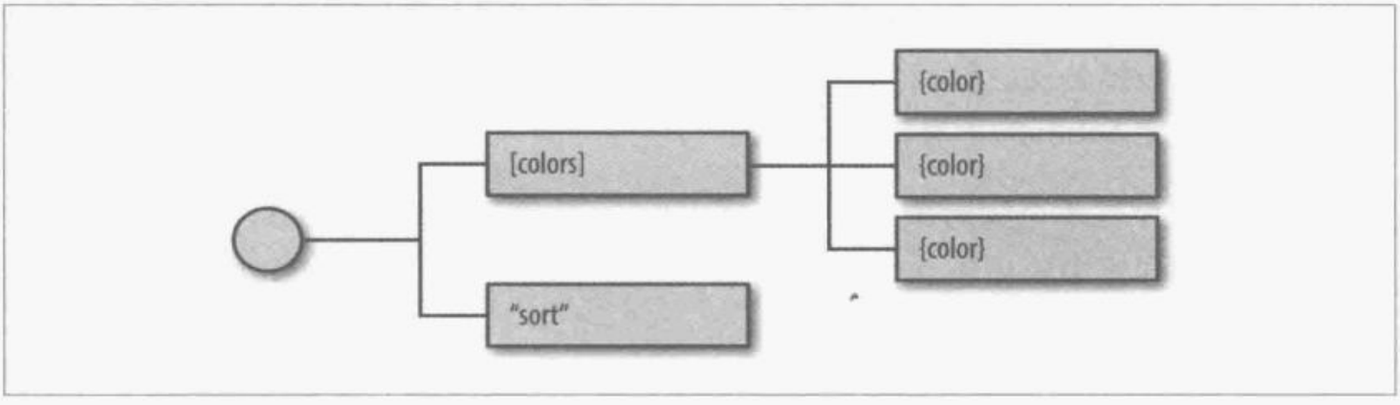


图8-5：颜色管理器的State树

将使用单独的Reducer处理这个State树的每个部分。每个Reducer就是一个简单的函数，因此我们可以使用示例8-6中的代码将它们一次性导出。

示例8-6：颜色管理器中导出所有Reducer

```
import C from '../constants'

export const color = (state={}, action) => {
    return {}
}

export const colors = (state=[], action) => {
    return []
```

```
}

export const sort = (state="SORTED_BY_DATE", action) => {
    return ""
}
```

注意，上述代码中color的Reducer希望接收的参数应该是一个对象，同时返回的也是一个对象。colors的Reducer希望接受的参数是一个数组，然后返回一个数组。sort的Reducer希望接受的参数是一个字符串，然后返回一个字符串。每个函数都聚焦于State树的特定部分。每个函数的返回值和初始State都和他们在State树中的数据类型一一对应。Colors表示一个数组，其中每种颜色是一个对象。属性sort是一个字符串。

每个Reducer只用来处理State树上Action必须更新的部分。color的Reducer将会只处理需要新增或者变更color对象的Action：ADD_COLOR和RATE_COLOR。colors的Reducer将会聚焦于管理colors数组的Action上：ADD_COLOR， REMOVE_COLOR,和RATE_COLOR。最后，sort的Reducer将会处理的Action是SORT_COLORS。

每个Reducer都可以被合成或者组合成单个Reducer函数，以便在Store中使用。colors的Reducer是由color的Reducer合成的，用于管理数组中的若干颜色对象。sort的Reducer可以和colors的Reducer组合，创建单个Reducer函数。这样一来就可以更新整个State树和处理任何接收到的Action了（见图8-6）。

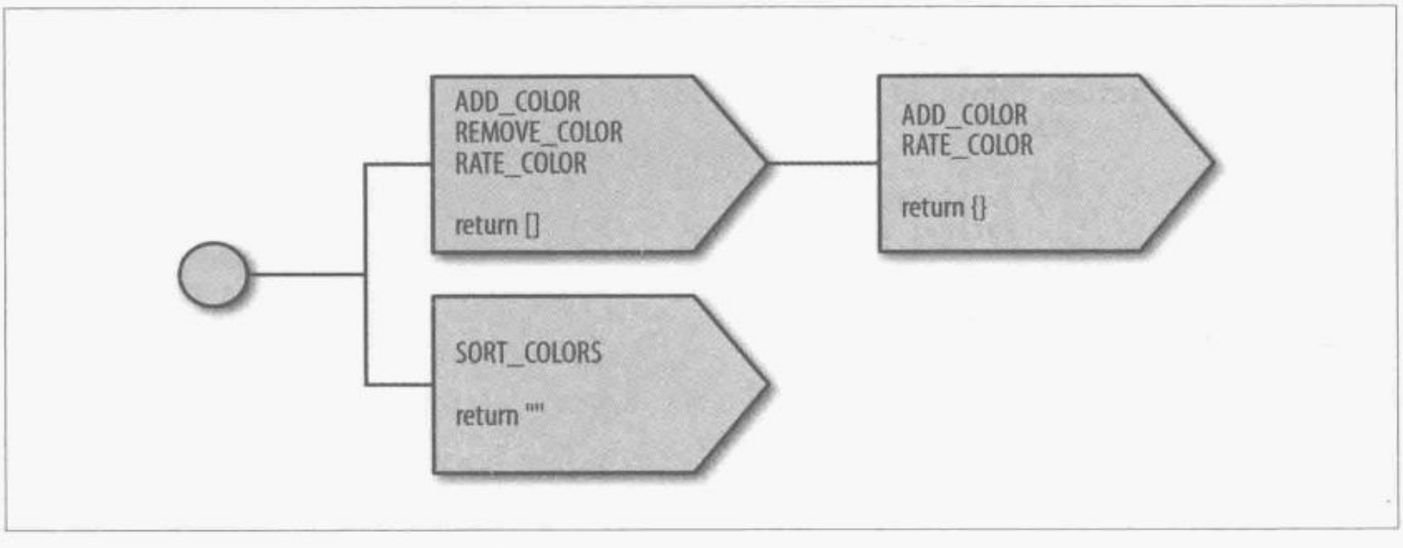


图8-6：颜色管理器的Reducer树

colors和color的Reducer都将处理ADD_COLOR和RATE_COLOR。不过需要注意的是，每个Reducer只聚焦于State树的特定部分。RATE_COLOR在color的Reducer中将会处理变更单个颜色的评分值的任务；RATE_COLOR在colors的Reducer中将会用于对数组中被评分颜

色的查找定位。ADD_COLOR在colors的Reducer中将会用于返回一个包含额外颜色对象的数组。把它们放在一起是为了方便协作。每个Reducer会聚焦于分支在State树中特定Action的具体含义。

Reducer合成不是必需的，只是一种推荐做法

Redux并不需要用户创建短小精干的若干Reducer，然后将它们合成单个Reducer。我们可以创建一个Reducer函数来处理应用程序中的每个Action。为此，我们将会失去模块化和函数式编程带来的好处。

Color Reducer

Reducer的代码实现方式有很多种。switch语句是一种比较常见的选择，因为Reducer必须可以处理的多种Action。color的Reducer会在switch语句中测试action.type，然后根据情况处理每种Action：

```
export const color = (state = {}, action) => {
    switch (action.type) {
        case C.ADD_COLOR:
            return {
                id: action.id,
                title: action.title,
                color: action.color,
                timestamp: action.timestamp,
                rating: 0
            }
        case C.RATE_COLOR:
            return (state.id !== action.id) ?
                state :
                {
                    ...state,
                    rating: action.rating
                }
        default :
            return state
    }
}
```

这里是color的Reducer将会处理的若干Action：

ADD_COLOR

根据Action有效载荷数据构造并返回一个新的color对象。

RATE_COLOR

返回一个包含预期评分的新颜色对象。ES7对象扩展运算符允许用户对一个新对象声明当前State的值。

Reducer总是会返回一些东西。如果由于某些原因，Reducer是被一个无法识别的Action调用的，那么将会返回当前的State，即默认情况。

现在我们已经有了一个color的Reducer，我们可以使用它返回一个新的colors对象或者对现有colors对象中的颜色进行评分。比如：

```
// 添加一种新的颜色

const action = {
    type: "ADD_COLOR",
    id: "4243e1p0-9abl-4e90-95p4-8001l8yf3036",
    color: "#0000FF",
    title: "Big Blue",
    timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
}

console.log( color({}, action) )

//  控制台输出

// {
//  id:  "4243e1p0-9abl-4e90-95p4-8001l8yf3036",
//  color: "#0000FF",
//  title: "Big Blue",
//  timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)",
//  rating: "0"
// }
```

返回的新color对象包含所有字段，其中包括默认的评分值0。为了修改现有的color对象，我们可以发送包含ID和新的评分值的Action，即RATE_COLOR：

```
const existingColor = {
    id: "128e1p5-3abl-0e52-33p0-8401l8yf3036",
    title: "Big Blue",
    color: "#0000FF",
    timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)",
    rating: 0
}

const action =  {
    type: "RATE_COLOR",
    id: "4243e1p0-9abl-4e90-95p4-8001l8yf3036",
    rating: 4
}
console.log( color(existingColor, action) )

// 控制台输出

// {
//  id:  "4243e1p0-9abl-4e90-95p4-8001l8yf3036",
//  title: "Big Blue",
//  color: "#0000FF",
```

```
//  timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)",
//  rating: 4
// }
```

color的Reducer是一个函数，它会创建一个新对象或者对某个现有对象评分。读者应该已经注意到color的Reducer并没有用到RATE_COLOR传递的ID。这是因为该Action的ID主要是用来在不同Reducer之间对该颜色对象进行定位的。一个Action对象可能会影响若干个Reducer。

Colors Reducer

color的Reducer主要是用来管理State树colors分支上的叶节点的。colors的Reducer将会用于管理整个colors分支：

```
export const colors = (state = [], action) => {
    switch (action.type) {
        case C.ADD_COLOR :
            return [
                ...state,
                color({}, action)
            ]
        case C.RATE_COLOR :
            return state.map(
                c => color(c, action)
            )
        case C.REMOVE_COLOR :
            return state.filter(
                c => c.id !== action.id
            )
        default:
            return state
    }
}
```

colors的Reducer将会处理任何添加、评分和移除colors的Action请求。

ADD_COLOR

使用一个新的颜色对象和现有State数组中的值串联，继而返回一个新的数组。被创建的新颜色将会传递给一个空的State对象，以及该颜色对应Action的Reducer。

RATE_COLOR

返回一个包含预期颜色评分的新colors数组。colors的Reducer会在当前State数组中对被评分的颜色进行定位查找。然后使用color的Reducer获取新的评分颜色对象，并使用它替换数组中的原有对象。

REMOVE_COLOR

将预期的颜色对象移除，然后创建一个新的数组。

colors的Reduer只关心colors数组。它使用color的Reducer处理单个颜色对象。

将State当作不可变对象

对于上述这些Reducer，我们需要将State看作一个不可变对象。即可以临时性地使用State.push({})或者State[index].rating，但是我们仍然应该尽量避免这么做。

现在可以使用下列纯函数从colors数组中添加、评分和移除某种颜色了：

```
const currentColors = [
  {
    id: "9813e2p4-3abl-2e44-95p4-8001l8yf3036",
    title: "Berry Blue",
    color: "#000066",
    rating: 0,
    timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
  }
]

const action = {
    type: "ADD_COLOR",
    id: "5523e7p8-3ab2-1e35-95p4-8001l8yf3036",
    title: "Party Pink",
    color: "#F142FF",
    timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
}

console.log( colors(currentColors, action) )


// 控制台输出

// [{
//  id:  "9813e2p4-3abl-2e44-95p4-8001l8yf3036",
//  title: "Berry Blue",
//  color: "#000066",
//  timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)",
//  rating: 0
// },

// {
//  id:  "5523e7p8-3ab2-1e35-95p4-8001l8yf3036",
//  title: "Party Pink",
//  color: "#F142FF",
//  timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)",
//  rating: 0
// }]
```

消除Reducer的副作用

Reducer应该是可预见的。它们只是用来管理State数据的。在上一个示例中需要注意，时间戳和ID是在将Action发送给Redcuer之前生成的。生成随机数据，调用API，以及其他异步过程应该在Reducer之外处理。为了避免State变异和产生副作用，所以才推荐用户这么做。

我们还可以从State中移除单个颜色，或者通过将相应的Action发送给colors的Reducer来对单个颜色进行评分。

Sort Reducer

sort的Reducer是一个完整的函数，用于管理State中某个字符串变量：

```
export const sort = (state = "SORTED_BY_DATE", action) => {
    switch (action.type) {
        case C.SORT_COLORS:
            return action.sortBy
        default :
            return state
    }
}
```

sort的Reducer是用来修改sort函数中的State参数的值。它会将State参数值设置为Action的sortBy字段中的值（如果没有提供State参数，它将返回SOR TED_BY_DATE）：

```
const state = "SORTED_BY_DATE"

const action = {
    type: C.SORT_COLORS,
    sortBy: "SORTED_BY_TITLE"
}

console.log( sort(state, action) )     // "SORTED_BY_TITLE"
```

总之，State的更新是通过Reducer处理的。Reducer是纯函数，接收State作为第一个参数，Action作为第二个参数。Reducer并没有产生副作用，并且应该将它们的参数作为不可变数据。Redux中，模块化是通过Reducer实现的。若干Reducer经过组合构成了单个Reducer，一个更新整个State树的函数。

在本节中，我们可以了解到Reducer是如何被合成的。我们已经了解了colors的Reducer是如何借助color的Reducer进行颜色管理的。在下一节中，我们将会介绍如何将colors的Reducer和sort的Reducer组合在一起，用于对State进行更新。

Store

在Redux中，Store就是保存应用程序State数据和处理所有State更新的地方。[注5]同时Flux设计模式支持多个Store共存，其中每个Store只专注于特定的数据集。Redux只有一个Store。该Store通过将当前的State和Action传递给单个Reducer来进行State更新。我们将会通过组合和合成若干Reducer创建一个这样的单个Reducer。

如果我们使用colors的Reducder创建了一个Store，那么我们的State对象将会是一个数组，即colors数组。Store的`getState`方法将会返回最近的应用程序State。在示例8-7中，我们将会创建一个包含color的Reducer的Store，这样一来用户就可以使用任意Reducer创建某个Store了。

示例8-7：包含color的Reducer的Store

```
import { createStore } from 'redux'
import { color } from './reducers'

const store = createStore(color)

console.log( store.getState() )   // {}
```

为了创建一个和图8-6类似的单个Reducer树，我们必须将colors和sort的Reducer组合起来。Redux有一个专门执行此任务的函数，即`combineReducers`，它会将所有Reducer组合到一起，构造单个Reducer。这些Reducer是用来构造用户的State树的。字段名和传递的Reducer名称是相匹配的。

一个Store还可以使用初始化数据创建。在不提供State参数的情况下调用colors的Reducer时，它将会返回一个空数组：

```
import { createStore, combineReducers } from 'redux'
import { colors, sort } from './reducers'

const store = createStore(
    combineReducers({ colors, sort })
)

console.log( store.getState() )

// 控制台输出

//{
//  colors: [],
//  sort: "SORTED_BY_DATE"
//}
```

注5：Redux Docs，“Store”。

在示例8-8中，Store是根据三种颜色和sort的值SORTED_BY_TITLE创建的。

示例8-8：初始化State数据

```
import { createStore, combineReducers } from 'redux'
import { colors, sort } from './reducers'

const initialState = {
    colors: [
        {
            id: "3315e1p5-3abl-0p523-30e4-8001l8yf3036",
            title: "Rad Red",
            color: "#FF0000",
            rating: 3,
            timestamp: "Sat Mar 12 2016 16:12:09 GMT-0800 (PST)"
        },
        {
            id: "3315e1p5-3abl-0p523-30e4-8001l8yf4457",
            title: "Crazy Green",
            color: "#00FF00",
            rating: 0,
            timestamp: "Fri Mar 11 2016 12:00:00 GMT-0800 (PST)"
        },
        {
            id: "3315e1p5-3abl-0p523-30e4-8001l8yf2412",
            title: "Big Blue",
            color: "#0000FF",
            rating: 5,
            timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
        }
    ],
    sort: "SORTED_BY_TITLE"
}

const store = createStore(
    combineReducers({ colors, sort }),
    initialState
)
console.log( store.getState().colors.length )      // 3
console.log( store.getState().sort )               // "SORTED_BY_TITLE"
```

修改用户应用程序State的唯一方式是通过Store分发Action。Store包含一个dispatch方法，可以接收Action作为参数。当用户通过Store分发某个Action时，该Action将会被分配到与之相关的Reducer上，然后State被更新了：

```
    console.log(
        "Length of colors array before ADD_COLOR",
        store.getState().colors.length
    )

    // 在 ADD_COLOR之前，颜色数组的长度是 3

    store.dispatch({
       type: "ADD_COLOR",
```

```
  id:  "2222e1p5-3abl-0p523-30e4-8001l8yf2222",
  title: "Party Pink",
  color: "#F142FF",
  timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
})

console.log(
    "Length of colors array after ADD_COLOR",
    store.getState().colors.length
)

// 在ADD_COLOR之后，颜色数组的长度是 4

console.log(
    "Color rating before RATE_COLOR",
    store.getState().colors[3].rating
)

// 在RATE_COLOR之前，颜色评分值是 0

store.dispatch({
  type: "RATE_COLOR",
  id:  "2222e1p5-3abl-0p523-30e4-8001l8yf2222",
  rating: 5
})

console.log(
    "Color rating after RATE_COLOR",
    store.getState().colors[3].rating
)

// 在RATE_COLOR之后，颜色评分值是 5
```

这里我们创建了一个Store，并分发了一个添加新颜色的Action，随后的Action是用来修改颜色评分的。控制台输出信息向我们展示了分发Action过程中State的变化。

原本在数组中我们有三种颜色。添加了一种颜色后，现在就有4种颜色了。新增的颜色原先的评分值是0。分发了一个Action将它的值改为5。修改数据的唯一方式是分发Action到Store中。

订阅Store

Store允许用户订阅每次分发完毕一个Action之后的被触发的句柄函数。在下列示例中，我们将会记录State中颜色的数目：

```
store.subscribe(() =>
    console.log('color count:', store.getState().colors.length)
)

store.dispatch({
```

```
    type: "ADD_COLOR",
    id: "2222e1p5-3abl-0p523-30e4-8001l8yf2222",
    title: "Party Pink",
    color: "#F142FF",
    timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
})

store.dispatch({
    type: "ADD_COLOR",
    id: "3315e1p5-3abl-0p523-30e4-8001l8yf2412",
    title: "Big Blue",
    color: "#0000FF",
    timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
})

store.dispatch({
    type: "RATE_COLOR",
    id: "2222e1p5-3abl-0p523-30e4-8001l8yf2222",
    rating: 5
})

store.dispatch({
    type: "REMOVE_COLOR",
    id: "3315e1p5-3abl-0p523-30e4-8001l8yf2412"
})

// 控制台输出

// color count: 1
// color count: 2
// color count: 2
// color count: 1
```

订阅这个针对Store的监听器之后，将会在我们每次提交Action时，将颜色数目记录显示在控制台上。在上述示例中，我们可以看到4条日志记录：前两条是和ADD_COLOR有关的，第3条是和RATE_COLOR有关的，第4条是和REMOVE_COLOR有关的。

Store的subscribe方法返回了一个函数，用户可以使用该函数退订上述监听器：

```
const logState = () => console.log('next state', store.getState())

const unsubscribeLogger = store.subscribe(logState)

// 当准备好退订监听器时触发执行
unsubscribeLogger()
```

保存到localStorage

在使用Store的subscribe函数时，我们将监听State的变化，并将这些变化存储到localStorage的'redux-Store'属性键下面。当我们创建Store时，可以事先检查一下

该属性键下面是否包含数据，如果包含数据，那么可以用该数据初始化State。只需几行代码，我们就可以在浏览器中拥有持久化的State数据：

```
const store = createStore(
    combineReducers({ colors, sort }),
    (localStorage['redux-store']) ?
        JSON.parse(localStorage['redux-store'])  :
        {}
)

store.subscribe(() => {
    localStorage['redux-store'] = JSON.stringify(store.getState())
})

console.log('current color count', store.getState().colors.length)
console.log('current state', store.getState())

store.dispatch({
    type: "ADD_COLOR",
    id: uuid.v4(),
    title: "Party Pink",
    color: "#F142FF",
    timestamp: new Date().toString()
})
```

每次我们刷新这些代码后，我们的colors列表都会因为新增一种颜色而变得更大。首先，在createStore调用过程中，我们会检查redux-Store这个属性键是否存在。如果存在，将会解析其中的JSON数据。如果不存在，我们将会返回一个空对象。其次，我们订阅了Store的一个监听器，每次分发一个Action之后，它都会保存Store的State数据。刷新该页面后将会一直添加同一种颜色到colors列表中。

总之，Store是Redux应用程序中保存和管理State数据的地方，也是通过Store分发Action的形式，唯一的修改State数据的方式。Store会将应用程序的State作为单个对象进行保存。State的变更是通过Reducer来完成的。Store的创建是通过提供一个Reducer和可选的初始State数据完成的。当然，我们还可以订阅Store的监听器（稍后还可以退订），它们会在每次Store分发完毕某个Action被触发执行。

Action生成器

Action对象是通过简单的JavaScript语法表示的。Action生成器就是返回这类语法格式的函数。接下来让我们看看几个Action的示例：

```
{
  type: "REMOVE_COLOR",
  id: "3315e1p5-3abl-0p523-30e4-8001l8yf2412"
}
```

```
{
  type: "RATE_COLOR",
  id: "441e0p2-9ab4-0p523-30e4-8001l8yf2412",
  rating: 5
}
```

我们可以通过为每种Action类型添加一个Action生成器来简化生成Action的逻辑：

```
import C from './constants'

export const removeColor = id =>
    ({
        type: C.REMOVE_COLOR,
        id
    })

export const rateColor = (id, rating) =>
    ({
        type: C.RATE_COLOR,
        id,
        rating
    })
```

现在无论我希望添加一个RATE_COLOR还是REMOVE_COLOR，都可以使用这个Action生成器，并且只需要将必需的数据作为函数参数传递即可：

```
store.dispatch( removeColor("3315e1p5-3abl-0p523-30e4-8001l8yf2412") )
store.dispatch( rateColor("441e0p2-9ab4-0p523-30e4-8001l8yf2412", 5) )
```

Action生成器简化了分发Action的工作，我们只需调用函数并发送必要的数据即可。Action生成器可以封装生成Action的细节，它可以大幅度简化创建Action的过程。比如，我们可以创建一个名为sortBy的Action，它可以决定分发相应的Action：

```
import C from './constants'

export const sortColors = sortedBy =>
    (sortedBy === "rating") ?
        ({
            type: C.SORT_COLORS,
            sortBy: "SORTED_BY_RATING"
        }) :
        (sortedBy === "title") ?
            ({
                type: C.SORT_COLORS,
                sortBy: "SORTED_BY_TITLE"
            }) :
            ({
                type: C.SORT_COLORS,
                sortBy: "SORTED_BY_DATE"
            })
```

Action生成器sortColors会根据“rating”、“title”和默认这三种情况检查sortBy属性。现在用户只需很少的手动输入工作就可以分发一个sortColors的Action了：

```
store.dispatch( sortColors("title") )
```

Action生成器也可以包含逻辑。当用户创建一个Action时，它还可以帮助我们封装一些不必要的细节。比如下面这个用于添加颜色的Action：

```
{
    type: "ADD_COLOR",
    id: uuid.v4(),
    title: "Party Pink",
    color: "#F142FF",
    timestamp: new Date().toString()
}
```

目前为止，Action被分发后还生成了相应的ID和时间戳。将这些逻辑转移到Action生成器中将会把分发Action过程的细节封装起来：

```
import C from './constants'
import { v4 } from 'uuid'

export const addColor = (title, color) =>
    ({
        type: C.ADD_COLOR,
        id: v4(),
        title,
        color,
        timestamp: new Date().toString()
    })
```

Action生成器addColor将会生成一个唯一ID并提供一个时间戳。现在创建一种新颜色将会更容易，我们通过创建一个能够自增的变量来生成唯一ID，时间戳是根据客户端当前时间自动生成的：

```
store.dispatch( addColor("#F142FF", "Party Pink") )
```

Action生成器独具特色的地方在于，它们提供了一个可以封装成功创建一个Action必需的所有逻辑的地方。Action生成器addColor完成了和添加新颜色有关的所有任务，其中包括提供唯一ID和Action的时间戳。它们都被聚集于一处，这使得应用程序调试更简单。

Action生成器应该是存放所有和后端API交互逻辑相关内容的地方。通过一个Action生成器，我们可以执行异步逻辑，比如请求数据或者发送一个API请求。

当我们在第12章介绍服务端应用时将会详细介绍这些内容。

compose函数

Redux还配备了一个compose函数，用户可以通过它将若干函数合成单个函数。它和我们在第3章介绍的compose函数类似，不过功能更健壮。此外，它是按照从右到左的顺序合成函数的，而不是从左到右。

如果我们只想获得一个逗号分隔的颜色列表，我们使用一行代码就能达到此目的：

```
console.log(store.getState().colors.map(c=>c.title).join(" , "))
```

一个更具函数式风格的方法是将上述内容分解成若干小型的函数，然后将它们合成单个函数：

```
import { compose } from 'redux'

const print = compose(
    list => console.log(list),
    titles => titles.join(" , "),
    map => map(c=>c.title),
    colors => colors.map.bind(colors),
    state => state.colors
)

print(store.getState())
```

这个compose函数会接收若干函数作为参数，然后从最右端开始顺次调用这些函数。首先，它会从State中获得若干颜色，然后返回一个bound映射函数，随后是一个颜色标题的数组，并使用逗号分隔列表的形式将它们连接起来，最后在控制台上输出。

中间件

如果读者曾经使用过服务端框架，比如Express、Sinatra、Django、KOA或者ASP.NET，那么应该对中间件这一概念并不陌生（如果没有，中间件可以作为两个不同层次或者不同软件之间的黏合剂）。

Redux也有中间件。它扮演了Store的分发管道的角色。在Redux中，中间件是在分发某个Action过程中一系列顺序执行的若干函数构成，如图8-7所示。

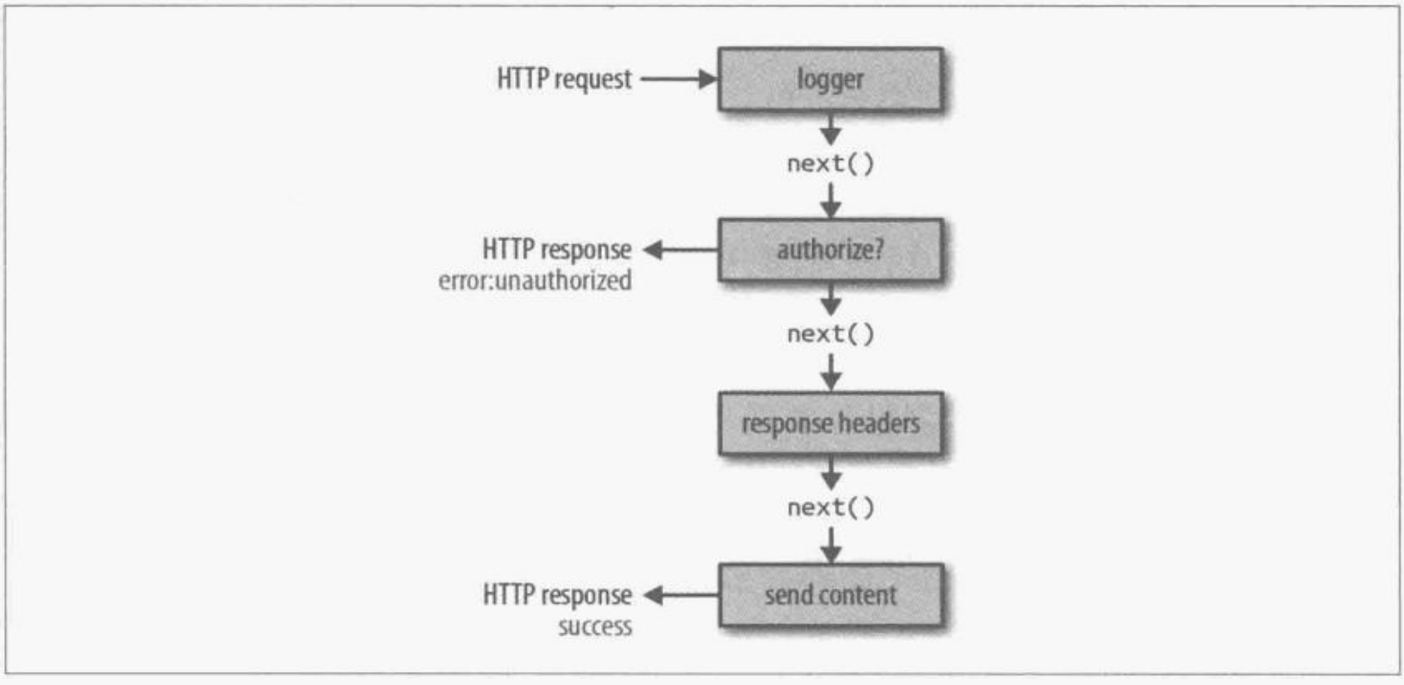


图8-7：HTTP请求中间件管道

这些高阶函数允许用户在某个Action被分发之前或之后，以及State被更新后，插入某些功能。每个中间件函数都是顺序执行的（见图8-8）。

每个中间件都是一个函数，它拥有访问Action、dispatch函数，以及下一个next函数的权限。next函数将会触发更新操作。在next函数被调用之前，State将会被更新。

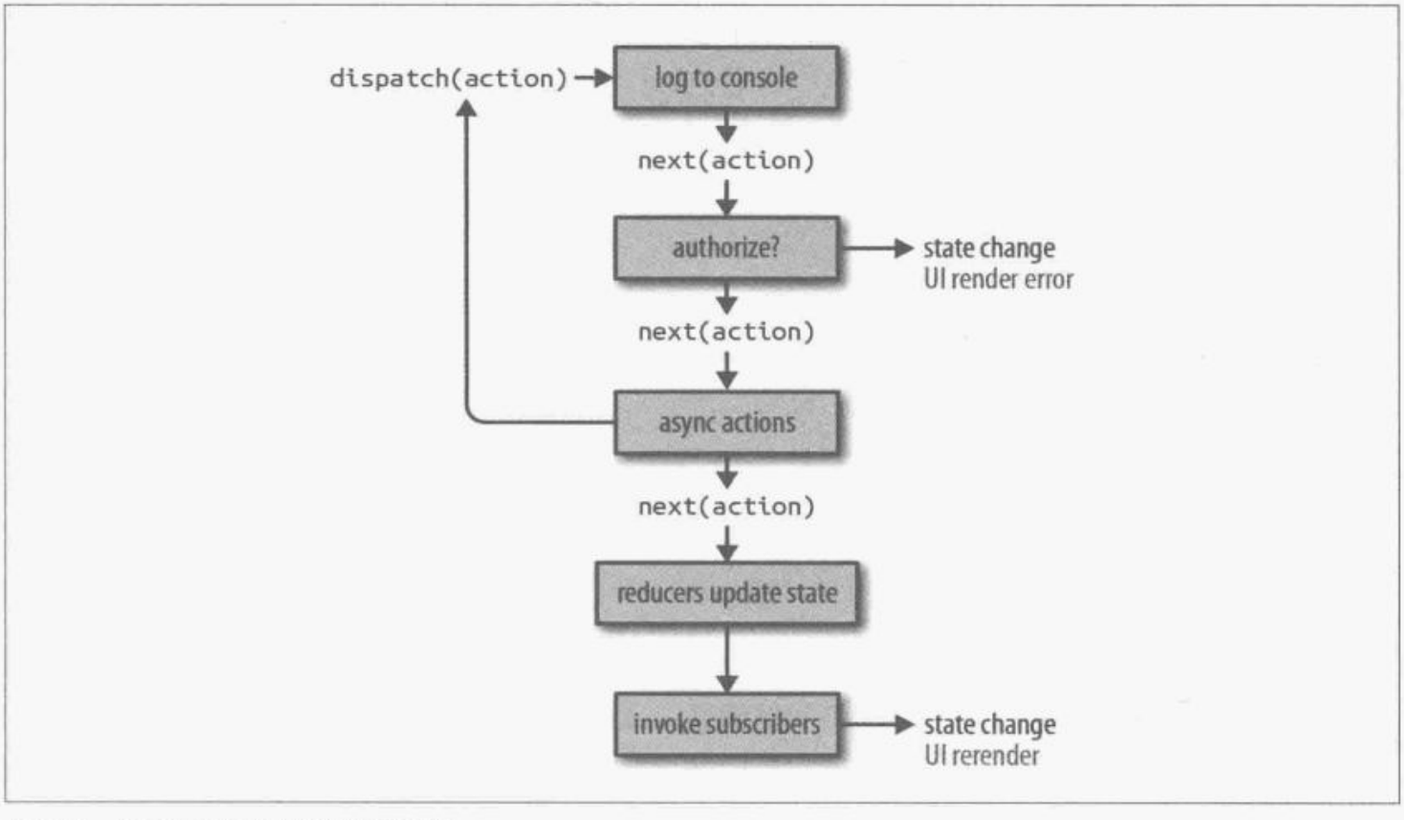


图8-8：顺次执行的中间件函数

在Store中使用中间件

在本节中，我们将会创建一个storeFactory。一个工厂类就是一个管理Store创建过程的函数。在这种情况下，该工厂类将会创建一个包含日志记录和保存数据功能的中间件的Store。storeFactory将会是一个文件，它包含一个函数，该函数专门用于对创建Store所需数据进行分组。一旦我们需要用到Store时，就可以调用该函数：

```
const store = storeFactory(initialData)
```

当我们创建了这个Store，同时也创建了两个中间件：*logger*和*saver*（参见示例8-9）。中间件将会把数据保存到localStorage中的，继而取代了Store的保存机制。

示例 8-9：storeFactory: ./store/index.js

```
import { createStore,
         combineReducers,
         applyMiddleware } from 'redux'
import { colors, sort } from './reducers'
import stateData from './initialState'

const logger = store => next => action => {
    let result
    console.groupCollapsed("dispatching", action.type)
    console.log('prev state', store.getState())
    console.log('action', action)
    result = next(action)
    console.log('next state', store.getState())
    console.groupEnd()
}

const saver = store => next => action => {
    let result = next(action)
    localStorage['redux-store'] = JSON.stringify(store.getState())
    return result
}

const storeFactory = (initialState=stateData) =>
    applyMiddleware(logger, saver)(createStore)(
        combineReducers({colors, sort}),
        (localStorage['redux-store']) ?
            JSON.parse(localStorage['redux-store']) :
            stateData
    )

export default storeFactory
```

logger和saver都是中间件函数。在Redux中，中间件会被当作一个高阶函数：一个函数，其返回的函数同时又返回了另一个函数。最后返回的函数会在每个Action被分发后触发执行。当这个函数被触发后，用户就可以访问相关的Action、Store，以及将该Action发送给下一个中间件函数。

和直接导出Store相反，我们导出了一个函数，即用来创建Store的类工厂。如果该类工厂被触发了，那么它将创建和返回一个集成了日志记录及数据保存功能的Store。

在logger中，在该Action被分发之前，我们打开了一个新的控制台分组，记录了当前的State和Action。触发next管道上的Action顺序执行中间件函数，最终到达Reducer。

此时的State已经被更新了，因此我们记录被修改过的State，最后在控制台上分组显示。

在saver中，我们触发了包含相关Action的next函数，这将会修改State的数据。然后我们将新的State保存到localStorage中并返回结果，如示例8-9所示。

在示例8-10中，我们是使用storeFactory创建Store实例的。因为我们并没有传递任何参数给该Store，因此初始的State是由State数据决定的。

示例8-10：使用类工厂创建一个Store

```
import storeFactory from "./store"

const store = storeFactory(true)
store.dispatch( addColor("#FFFFFF","Bright White") )
store.dispatch( addColor("#00FF00","Lawn") )
store.dispatch( addColor("#0000FF","Big Blue") )
```

每个从这个Store分发的Action将会在控制台上输出一个新的日志分组信息，并且新的State将会保存到localStorage中。

在本章中，我们学习了Redux的所有关键特性：State、Action、Store、Action生成器，以及中间件。我们使用Redux处理了应用程序的所有State，现在我们可以将它和用户界面连接到一起。

下一章中，我们将会学习react-redux框架，它是一款能够高效地将Redux的Store和React的UI连接到一起的工具。

第9章

React Redux

在第6章，我们已经学习了如何构造React组件。我们使用React的State管理系统构建了一个颜色管理器应用。在上一章，我们学习了如何使用Redux管理应用程序的State数据。为了方便颜色管理器程序分发Action，我们为它构建了一个功能完备的Store。在本章中，我们将会把第6章创建的UI和上一章构建的Store整合到一起。

我们第6章开发的应用程序是将State存储在单个对象中并且位置也是单一的，即App组件。

```
export default class App extends Component {

    constructor(props) {
        super(props)
        this.state = {
            colors: [
                {
                    "id":  "8658c1d0-9eda-4a90-95e1-8001e8eb6036",
                    "title": "Ocean Blue",
                    "color": "#0070ff",
                    "rating": 3
                },
                {
                    "id": "f9005b4e-975e-433d-a646-79df172e1dbb",
                    "title": "Tomato",
                    "color": "#d10012",
                    "rating": 2
                },
                {
                    "id":  "58d9caee-6ea6-4d7b-9984-65b145031979",
                    "title": "Lawn",
                    "color": "#67bf4f",
                    "rating": 1
                },
```

```
                {
                    "id":  "a5685c39-6bdc-4727-9188-6c9a00bf7f95",
                    "title": "Party Pink",
                    "color": "#ff00f7",
                    "rating": 5
                }
            ]
        }
        this.addColor = this.addColor.bind(this)
        this.rateColor = this.rateColor.bind(this)
        this.removeColor = this.removeColor.bind(this)
    }

    addColor(title, color) {
        ...
    }

    rateColor(id, rating) {
        ...
    }

    removeColor(id) {
        ...
    }

    render() {
        const { addColor, rateColor, removeColor } = this
        const { colors } = this.state
        return (
            <div className="app">
                <AddColorForm onNewColor={addColor} />
                <ColorList colors={colors}
                           onRate={rateColor}
                           onRemove={removeColor} />
            </div>
        )
    }

}
```

App组件是保存State的组件。State作为属性向下传递给子组件。具体来说，colors作为属性从App组件的State传递给了`ColorList`组件。当触发事件时，数据通过回调函数的属性沿着组件树向上回传给了App组件（见图9-1）。

数据在组件树中向上和向下传递的过程增加了程序的复杂性，类似Redux这样的库就是为了缓解这一问题而诞生的。为了替代通过双向函数绑定实现组件树上的数据传递，我们可以直接从子组件分发Action来达到更新应用程序State的目的。

在本章中，我们将会介绍多种集成Redux的Store的方法。首先我们会介绍在不使用任何额外框架的情况下如何使用Store。然后，我们将会介绍`react-redux`，它是一款可以将Redux的Store和React组件集成到一起的框架。

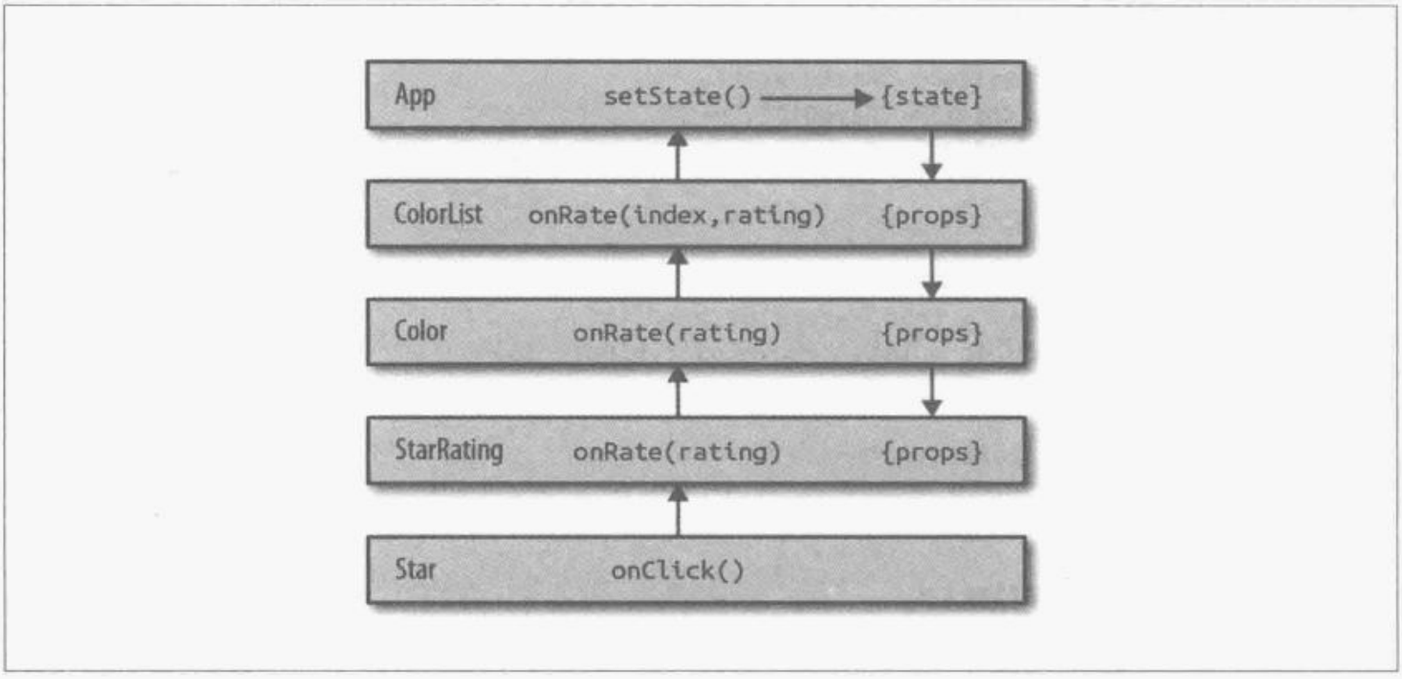


图9-1：组件树中的数据流传递

显式传递Store

首先，将Store集成到UI中最合乎逻辑的做法是显式地将它作为属性在组件树中向下传递。该方法很简单，对于只包含少量嵌套组件的小型应用程序效果非常好。

接下来看看如何将Store集成到颜色管理器程序中。在*./index.js*文件中，我们将会渲染一个App组件并传递Store：

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './components/App'
import storeFactory from './store'

const store = storeFactory()

const render = () =>
    ReactDOM.render(
        <App store={store}/>,
        document.getElementById('react-container')
    )

store.subscribe(render)
render()
```

这就是*./index.js*文件中的内容。在这个文件中，我们使用`storeFactory`创建了一个Store，并将App组件渲染到了DOM文档中。当App组件渲染完毕后，Store会作为属性

传递给它。每次Store发生变化后，render函数都会被触发执行，这样就可以使用新的State数据高效地更新UI了。

现在已经将Store传递给了App组件，我们必须继续将它向下传递给需要用到它的子组件中：

```
import AddColorForm from './AddColorForm'
import SortMenu from './SortMenu'
import ColorList from './ColorList'

const App = ({ store }) =>
    <div className="app">
        <SortMenu store={store} />
        <AddColorForm store={store} />
        <ColorList store={store} />
    </div>

export default App
```

App组件是我们的根组件。它通过属性获取了Store，然后将它显式向下传递给了它的子组件。Store作为属性传递给了SortMenu、AddColorForm和ColorList等组件。

我们已经将Store从App中传递了出去，现在可以在子组件中使用它了。务必记得我们可以使用store.getState从Store中读取State，并且可以使用store.dispatch将Action分发到Store。

通过AddColorForm组件，我们可以使用Store分发ADD_COLOR。当用户提交表单后，系统会从引用中收集颜色和标题等信息，然后使用这些数据创建和分发一个新的Action，即ADD_COLOR：

```
import { PropTypes, Component } from 'react'
import { addColor } from '../actions'

const AddColorForm = ({store}) => {

    let _title, _color

    const submit = e => {
        e.preventDefault()
        store.dispatch( addColor(_title.value, _color.value) )
        _title.value = ''
        _color.value = '#000000'
        _title.focus()
    }

    return (
        <form className="add-color" onSubmit={submit}>
            <input ref={input => _title = input}
                type="text"
```

```
                placeholder="color title..." required/>
            <input ref={input => _color = input}
                type="color" required/>
            <button>ADD</button>
        </form>
    )

}

AddColorForm.propTypes = {
    store: PropTypes.object
}

export default AddColorForm
```

从这个组件中，我们可以导入必需的Action生成器，即addColor。当用户提交表单后，我们将会使用这个Action生成器直接分发一个新的ADD_COLOR到Store。

ColorList组件可以使用Store的getState方法获取原来的颜色，并对它们进行适当的排序。它还可以直接分发RATE_COLOR和REMOVE_COLOR这两个Action，当它们被触发时：

```
import { PropTypes } from 'react'
import Color from './Color'
import { rateColor, removeColor } from '../actions'
import { sortFunction } from '../lib/array-helpers'

const ColorList = ({ store }) => {
    const { colors, sort } = store.getState()
    const sortedColors = [...colors].sort(sortFunction(sort))
    return (
        <div className="color-list">
            {(colors.length === 0) ?
                <p>No Colors Listed. (Add a Color)</p> :
                sortedColors.map(color =>
                    <Color key={color.id}
                        {...color}
                        onRate={(rating) =>
                            store.dispatch(
                                rateColor(color.id, rating)
                            )
                        }
                        onRemove={() =>
                            store.dispatch(
                            removeColor(color.id)
                    } />
                )
            }
        </div>
    )
}
ColorList.propTypes = {
```

```
    store: PropTypes.object
}

export default ColorList
```

Store已经完全通过组件树向下传递给了ColorList组件。该组件可以直接和Store交互。当对颜色进行评分或者删除操作时，这些Action将会被分发到Store。

Store还可以用来获取原始的颜色信息。这些颜色是重复的，会根据Store的sort属性排序并另存为sortedColors。sortedColors将会用于创建UI。

如果用户的组件树非常小，那么这种方法将会非常有用，比如颜色管理器程序。这种方法的缺点是用户必须显式将Store传递给子组件，这意味着需要编写更多的代码，比其他方法更繁琐一些。此外，SortMenu、AddColorForm和ColorList组件也需要用到这个特殊的Store。它可能在其他应用程序中难于复用。

在接下来的两节内容中，我们将会介绍组件获取Store的其他方式。

通过上下文传递Store

在上一节中，我们创建了一个Store，然后将它沿着组件树从App组件向下传递给了ColorList组件。该方法要求我们必须经过App和ColorList之间的每个组件才能达成目标。

假定我们有一些货物需要从华盛顿运送到旧金山，我们可以使用火车，但是这需要我们铺设横跨至少9个州的火车轨道才能抵达加利福尼亚。这和显式地沿着组件树从根节点传递到叶节点的情况类似。用户必须“铺设轨道”经过源地址和目标之间的每个组件。如果使用火车和显式通过属性传递Store类似，那么显式通过上下文传递Store就和使用喷气式客机类似。当一架喷气式飞机从华盛顿飞往旧金山时，并不需要铁轨。

类似地，我们可以利用被称为上下文的React特性，它允许用户在组件树中不显式向下传递属性的情况下将变量传递给组件。[注1]任意子组件都可以访问这些上下文变量。

如果我们希望在颜色管理器应用中使用上下文传递Store的话，需要做的第一步就是重构App组件来保存上下文。App组件还需要监听Store，以便每次State发生变化后它可以触发UI更新：

注1：Dan Abramov, “Redux: Extracting Container Components”, Egghead.io (*http://bit.ly/2mJaTr9*)。

```
import { PropTypes, Component } from 'react'
import SortMenu from './SortMenu'
import ColorList from './ColorList'
import AddColorForm from './AddColorForm'
import { sortFunction } from '../lib/array-helpers'

class App extends Component {

    getChildContext() {
        return {
            store: this.props.store
        }
    }

    componentWillMount() {
        this.unsubscribe = store.subscribe(
            () => this.forceUpdate()
        )
    }

    componentWillUnmount() {
        this.unsubscribe()
    }

    render() {
        const { colors, sort } = store.getState()
        const sortedColors = [...colors].sort(sortFunction(sort))
        return (
            <div className="app">
                <SortMenu />
                <AddColorForm />
                <ColorList colors={sortedColors} />
            </div>
        )
    }
}

App.propTypes = {
    store: PropTypes.object.isRequired
}

App.childContextTypes = {
    store: PropTypes.object.isRequired
}

export default App
```

首先，将上下文添加到组件中需要用户使用getChildContext生命周期函数。它会返回定义上下文的对象。在这种情况下，我们可以将Store添加到上下文对象中，以便可以通过属性访问它。

接下来，用户需要在组件实例上声明childContextTypes和定义上下文对象。这个过

程与添加propTypes或者defaultProps到组件实例上类似。不过，为了让上下文正常运作，用户必须执行这一步骤。

现在，任何App组件的任意子组件都可以通过上下文访问Store了。它们可以直接调用store.getState和store.dispatch方法。最后一步是订阅Store，每次Store更新State后更新组件树。这可以通过挂载生命周期函数来实现。在componentWillMount中，我们可以订阅Store，并使用this.forceUpdate启动更新生命周期，它会重新渲染UI。在componentWillUnmount中，我们可以触发退订函数并停止监听Store。由于App组件自身会触发UI更新，因此也就不需要从*./index.js*文件实体中订阅Store了。我们可以从添加Store到上下文的同一组件监听Store的变化，即App。

接下来将会重构AddColorForm组件，以便可以获取访问Store和直接分发ADD_COLOR这一Action：

```
const AddColorForm = (props, { store }) => {

    let _title, _color

    const submit = e => {
        e.preventDefault()
        store.dispatch(addColor(_title.value,   _color.value))
        _title.value = ''
        _color.value = '#000000'
        _title.focus()
    }

    return (
        <form className="add-color" onSubmit={submit}>
            <input ref={input => _title = input}
                   type="text"
                   placeholder="color title..." required/>
            <input ref={input => _color = input}
                   type="color" required/>
            <button>ADD</button>
        </form>
    )

}

AddColorForm.contextTypes = {
    store: PropTypes.object
}
```

上下文对象会排在props后面，作为第二个参数传递给无状态函数式组件。我们可以通过对象解构直接从这个参数中的对象获取Store。为了使用Store，我们必须在AddColorForm实例上定义contextTypes。这是为了告知React该组件将会使用的上下文变量类型。这是一个必需的步骤。如果没有它，用户将无法从上下文中读取Store。

接下来看看如何在一个组件类中使用上下文。Color组件可以访问Store，并且可以直接分发RATE_COLOR和REMOVE_COLOR这两个Action：

```
import { PropTypes, Component } from 'react'
import StarRating from './StarRating'
import TimeAgo from './TimeAgo'
import FaTrash from 'react-icons/lib/fa/trash-o'
import { rateColor, removeColor } from '../actions'

class Color extends Component {

    render() {
        const { id, title, color, rating, timestamp } = this.props
        const { store } = this.context
        return (
            <section className="color" style={this.style}>
                <h1 ref="title">{title}</h1>
                <button onClick={() =>
                    store.dispatch(
                        removeColor(id)
                    )
                }>
                    <FaTrash />
                </button>
                <div className="color"
                    style={{ backgroundColor: color }}>
                </div>
                <TimeAgo timestamp={timestamp} />
                <div>
                    <StarRating starsSelected={rating}
                                onRate={rating =>
                                    store.dispatch(
                                        rateColor(id, rating)
                                    )
                                } />
                </div>
            </section>
        )
    }
}

Color.contextTypes = {
    store: PropTypes.object
}

Color.propTypes = {
    id: PropTypes.string.isRequired,
    title: PropTypes.string.isRequired,
    color: PropTypes.string.isRequired,
    rating: PropTypes.number
}

Color.defaultProps = {
    rating: 0
```

```
}

export default Color
```

ColorList现在是一个组件类，并且可以通过this.context访问上下文。Colors可以通过store.getState直接从Store中读取。相同的规则也适用于无状态函数式组件。必须在实例上定义contextTypes。

从上下文中访问Store是一种减少引用的好办法，不过它并不是每个应用程序必不可少的部分。Redux的创始人Dan Abramov甚至声称可以不必遵循这些模式：

> 将容器和表现层组件分离通常是个好主意，但是用户也不必拘泥于此。只有在它可以切实降低项目代码的复杂度时才有必要这么做。[注2]

表现层和容器组件

在上一个示例中，Color组件是通过上下文访问Store的，并且使用它可以直接分发RATE_COLOR和REMOVE_COLOR这两个Action。在此之前，ColorList组件通过上下文访问了Store，从State中读取了当前的颜色列表信息。在这两个示例中，这些组件是直接和Redux的Store交互来渲染UI元素的。我们可以通过将Store和渲染UI的组件脱钩来优化应用程序架构。[注3]

表现层组件就是指渲染UI元素的组件。[注4]它们并没有和任何数据架构紧密地耦合在一起。相反，它们通过属性接收数据，并且通过回调函数属性将数据回传给它们的父组件。它们只是纯粹地聚焦于UI，并且可以在包含不同数据的应用程序之间复用。除了App组件之外，我们第6章创建的组件都是表现层组件。

容器组件就是将表现层组件和数据相连的组件。具体到我们的示例中，容器组件将会通过上下文访问Store，并且使用Store管理任意交互。它们通过映射属性到State和将回调函数属性传递给Store的dispatch方法，来达到渲染表现层组件的目的。容器组件关注的重点并不在UI元素，它们只是用来将表现层组件连接到数据。

这种架构有很多优点。表现层组件是可复用的。它们容易替换，测试也很方便。它们

注2： React Docs, "Context"。

注3： Redux Docs, "Presentational and Container Components"(http://bit.ly/2mJ92Co)。

注4： Dan Abramov, “Presentational and Container Components”, Medium, March 23, 2015(http://bit.ly/2mJfLw4)。

可以经过一系列组合来创建UI。表现层组件可以在使用了不同数据库的浏览器应用之间进行复用。

容器组件一点也不会关注UI。它们关注的重点是将表现层连接到数据的架构。容器组件可以在不同设备平台之间复用，以便达到将原生的表现层组件连接到数据的目的。

第6章我们创建的AddColorForm、ColorList、Color、StarRating和Star等组件都是表现层组件的例子。它们通过属性接收数据，当有事件触发时，它们会调用回调函数属性。我们已经对表现层组件非常熟悉了，接下来看看如何使用它们创建容器组件。

App组件大致会保持原样。它仍然会在上下文对象中定义Store，以便其子组件可以访问它。与渲染SortMenu、AddColorForm和ColorList这些组件相反，它会渲染这些元素的容器。Menu容器将会连接SortMenu组件，NewColor将会连接AddColorForm组件，Colors将会连接ColorList组件：

```
render() {
    return (
        <div className="app">
            <Menu />
            <NewColor />
            <Colors />
        </div>
    )
}
```

当用户希望将某个表现层组件和一些数据相连时，可以将该组件包装到一个容器中，以便能够控制属性并将它们和数据相连。NewColor容器、Menu容器和Colors容器都可以在同一文件中定义：

```
import { PropTypes } from 'react'
import AddColorForm from './ui/AddColorForm'
import SortMenu from './ui/SortMenu'
import ColorList from './ui/ColorList'
import { addColor,
         sortColors,
         rateColor,
         removeColor } from '../actions'
import { sortFunction } from '../lib/array-helpers'

export const NewColor = (props, { store }) =>
    <AddColorForm onNewColor={(title, color) =>
                        store.dispatch(addColor(title,color))
                    } />

NewColor.contextTypes = {
    store: PropTypes.object
}
```

```
export const Menu = (props, { store }) =>
    <SortMenu sort={store.getState().sort}
              onSelect={sortBy =>
                  store.dispatch(sortColors(sortBy))
              } />

Menu.contextTypes = {
    store: PropTypes.object
}

export const Colors = (props, { store }) => {
    const { colors, sort } = store.getState()
    const sortedColors = [...colors].sort(sortFunction(sort))
    return (
        <ColorList colors={sortedColors}
                   onRemove={id =>
                       store.dispatch( removeColor(id) )
                   }
                   onRate={(id, rating) =>
                       store.dispatch( rateColor(id, rating) )
                   }/>
    )
}

Colors.contextTypes = {
    store: PropTypes.object
}
```

NewColor容器并不会渲染UI。相反，它会渲染AddColorForm组件并处理来自该组件的onNewColor事件。该容器组件会从上下文对象中访问Store，然后通过它分发ADD_COLOR这一Action。它包含了AddColorForm组件，并且将它和Redux的Store相连。

Menu容器会渲染SortMenu组件。它会访问Store的State传递当前的sort属性，并且当用户选择了一个不同的菜单元素后，会分发sort这一Action。

Colors容器会通过上下文对象访问Store，然后根据Store的当前State渲染ColorList组件。它还会处理来自ColorList组件的onRate和onRemove事件。当这些事件触发后，Colors容器会分发相应的Action。

Redux的所有功能都被连接到了这个文件中。需要注意的是，被导入和调用的Action生成器都被聚集于一处。这是唯一可以调用store.getState或者store.dispatch方法的文件。

通过将UI组件和容器分离的方式将它们和数据连接是一种非常棒的方法。不过，这对于小型的项目、概念验证性或者原型项目来说可能有点大材小用了。

在下一节中，我们将会介绍一个新的脚本库，即React Redux。该脚本库可以用于快速地将Redux的Store添加到上下文并创建容器组件。

React Redux的Provider

React Redux是一个脚本库，它包含的一些工具可以显著降低显式通过上下文传递Store的复杂性。该库的作者也是Dan Abramov，即Redux的作者。Redux并不会强制要求用户使用该库。不过采用React Redux之后可以大幅度降低项目代码的复杂度，并且可以提高用户构建应用程序的工作效率。

为了使用React Redux，我们首先必须安装它。它可以通过npm（*https://www.npmjs.com/package/react-redux*）进行安装：

```
npm install react-redux --save
```

react-redux为用户提供了一个组件，可以通过它在上下文中配置用户自己的Store，即*Provider*。我们可以使用Provider包装任意React元素，该元素的子元素将能够通过上下文访问Store。

与在App组件中将Store配置成一个上下文变量相反，我们可以确保App组件是无状态的：

```
import { Menu, NewColor, Colors } from './containers'

const App = () =>
    <div className="app">
        <Menu />
        <NewColor />
        <Colors />
    </div>

export default App
```

当相关Action被分发后，Provider会将Store添加到上下文对象并更新App组件。Provider会接收单个子组件作为参数：

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import App from './components/App'
import storeFactory from './store'

const store = storeFactory()

render(
```

```
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById('react-container')
)
```

Provider要求用户将Store作为一个属性进行传递。它会将Store添加到上下文，以便App组件的任意子组件能够访问它。采用Provider之后可以节省我们的时间，并且还能简化代码。

一旦我们集成了Provider，用户就可以在子容器组件中通过上下文访问Store了。不过React Redux为用户提供了另外一种配合Provider快速创建容器组件的方法，即connect函数。

React Redux的connect函数

如果我们希望确保UI组件是纯粹的表现层，可以借助React Redux创建容器组件。React Redux可以通过将Redux的Store中当前State映射为表现层组件的属性来创建容器组件。它还可以将Store的dispatch函数映射成回调属性。这一切都是通过一个名为connect的高阶函数完成的。

接下来将使用connect函数创建Colors容器组件。Colors容器会将ColorList组件连接到Store：

```
import ColorList from './ColorList'

const mapStateToProps = state =>
    ({
        colors: [...state.colors].sort(sortFunction(state.sort))
    })

const mapDispatchToProps = dispatch =>
    ({
        onRemove(id) {
            dispatch(removeColor(id))
        },
        onRate(id, rating) {
            dispatch(rateColor(id, rating))
        }
    })

export const Colors = connect(
    mapStateToProps,
    mapDispatchToProps
)(ColorList)
```

connect是一个高阶函数，返回的函数返回了一个组件。请读者不要对此产生误解，以为上述文字是印刷错误或者语法问题：这就是函数式JavaScript的特性。connect函数会接收两个参数：mapStateToProps和mapDispatchToProps。它们都是函数。它返回的函数将是一个表现层组件，并且该组件被一个容器包装，可以通过属性发送数据。

第一个函数mapStateToProps，会将State作为参数注入，返回的对象将后者映射成属性。我们可以从State中将ColorList组件的颜色属性设置为一个有序的颜色数组。

第二个函数mapDispatchToProps，会将Store的dispatch函数作为参数注入，以便ColorList组件触发回调函数属性时调用。当ColorList组件触发了onRate或者onRemove事件后，被评分或者移除的颜色数据会被获取和分发。

connect函数与Provider一起协同工作。Provider将Store添加到上下文对象中，connect函数创建组件访问Store。当使用connect函数时，用户无需过多关注上下文对象。

我们使用的所有容器组件，都可以使用React Redux的connect函数在单个文件中创建：

```
import { connect } from 'react-redux'
import AddColorForm from './ui/AddColorForm'
import SortMenu from './ui/SortMenu'
import ColorList from './ui/ColorList'
import { addColor,
         sortColors,
         rateColor,
         removeColor } from '../actions'
import { sortFunction } from '../lib/array-helpers'

export const NewColor = connect(
    null,
    dispatch =>
        ({
            onNewColor(title, color) {
               dispatch(addColor(title,color))
            }
        })
)(AddColorForm)

export const Menu = connect(
    state =>
        ({
            sort: state.sort
        }),
    dispatch =>
        ({
            onSelect(sortBy) {
```

```
                dispatch(sortColors(sortBy))
            }
        })
)(SortMenu)

export const Colors = connect(
    state =>
        ({
            colors:   [...state.colors].sort(sortFunction(state.sort))
        }),
    dispatch =>
        ({
            onRemove(id) {
                dispatch(removeColor(id))
            },
            onRate(id, rating) {
                dispatch(rateColor(id, rating))
            }
        })
)(ColorList)
```

在这个示例中，我们使用的每个容器组件都是通过React Redux的connect函数定义的。connect函数会将Redux单纯地与表现层组件连接在一起。第一个参数是某个函数映射State变量而成的属性。第二个参数是当触发事件时，用于分发Action的函数。如果用户只想映射回调函数属性方便分发相关的Action，那么可以为第一个参数提供一个null值来作为占位符，因为我们在NewColor容器中已经对该函数做了定义。

在本章中，我们学习了将Redux连接到React多种方法。我们显式地将Store当作属性沿着组件树向下传递给了子组件。我们通过上下文对象将Store直接传递给了需要访问它的组件。通过使用容器组件，将Store的功能从表现层组件中剥离了出来。最后，我们使用react-redux库通过上下文对象和容器组件帮助用户快速地将Store和表现层连接到了一起。

目前来看，我们已经拥有了一个使用React和Redux技术，并且可以正常运行的应用程序。下一章，我们将会介绍如何为本应用程序的各个部分编写单元测试。

第10章

测试

为了和我们的竞争对手齐头并进，我们必须在确保产品质量的情况下勇往直前。一个能够确保用户可以达成这个目标的重要工具是单元测试。单元测试使得用户可以验证应用程序引入的每个部件或者单元能够按照预期的结果运行。

采用函数式编程技术的好处之一就是它们编写的代码都是可测试的。纯函数天生就是可测试的。不可变性也非常容易测试。为解决特定任务而设计的小型函数组合而成的应用程序也产生了可测试的函数或者代码单元。[注1]

在本节中，我们将会演示可以用于React Redux应用程序进行单元测试的技术。本章不仅会涉及测试，还会介绍辅助用户评估、改进用户代码和测试的工具。

ESLint

在大部分编程语言中，用户可以运行程序之前，代码必须经过编译。很多编程语言对代码风格有着非常严格的规定，只有代码经过相应的格式化并且格式完全正确之后才能进行编译。JavaScript并没有这些规则，并且也不需要使用编译器。我们手工编写代码，为了确认它们是否可以正常运行，在浏览器中直接运行它们即可。好消息是，有大量的工具可以帮助用户分析代码，并要求用户遵循特定的格式规范。

分析JavaScript代码的过程被称为代码检查（*hinting*或者*linting*）。JSHint和JSLint是用于分析JavaScript代码的原生工具，并且可以为用户提供代码格式化的反馈意见。

注1：希望了解单元测试的基本概念，可以参考Martin Fowler的文章“Unit Testing”（*http://martinfowler.com/bliki/UnitTest.html*）。

ESLint(*http://eslint.org/*)是一款最新的代码检查工具，并且支持最新的JavaScript语法特性。此外，ESLint支持插件机制。这意味用户可以创建和共享能够被添加到ESLint配置中的插件，继而扩展它的功能。

我们将会用到一个名为`eslint-plugin-react`(*http://bit.ly/2kuEylV*)的插件。该插件会在分析JavaScript代码的同时，对我们的JSX和React语法进行分析。

接下来将全局安装`eslint`。用户可以使用npm安装`eslint`：

```
sudo npm install -g eslint
```

在使用ESLint之前，我们将会需要定义一些用户愿意遵循的配置规则。定义这些规则的配置文件将会位于我们的项目根目录下。该文件还可以格式化成JSON或者YAML格式。YAML(*http://yaml.org/*)是一种和JSON类似的数据序列化格式，不过语法更少，用户更容易阅读理解。

ESLint自带的一个工具可以帮助用户建立配置文件。有不少公司创建的ESLint配置文件可以作为用户的入门配置，或者我们也可以创建自己的配置文件。

我们可以通过运行`eslint --init`命令创建一个ESLint配置文件，然后回答下列和代码风格有关的问题：

```
$ eslint --init

? How would you like to configure ESLint?
回答和代码风格有关的问题

? Are you using ECMAScript 6 features? Yes
? Are you using ES6 modules? Yes
? Where will your code run? Browser
? Do you use CommonJS? Yes
? Do you use JSX? Yes
? Do you use React? Yes
? What style of indentation do you use? Spaces
? What quotes do you use for strings? Single
? What line endings do you use? Unix
? Do you require semicolons? No
? What format do you want your config file to be in? YAML

Local ESLint installation not found.
Installing eslint, eslint-plugin-react
```

执行`eslint --init`命令之后，会发生以下三件事：

1. ESLint和`eslint-plugin-react`会被安装到本地的*./node_modules*文件下。

2. 这些引用依赖会自动添加到*package.json*文件中。

3. 一个以*.eslintrc.yml*为文件后缀名的配置文件被创建并添加到项目的根目录下。

接下来将会创建一个*sample.js*文件对我们的ESLint配置进行测试：

```
const gnar ="gnarly";

const info = ({file= filename, dir= dirname}) =>
    <p>{dir}: {file}</p>

switch(gnar) {
    default :
        console.log('gnarley')
        break
}
```

这个文件存在一些问题，不过对浏览器来说无伤大雅。从技术上来说，这段代码能够正常运行。让我们在这个文件上运行ESLint，看看基于我们的自定义规则会得到什么样的反馈结果：

```
$ ./node_modules/.bin/eslint sample.js

/Users/alexbanks/Desktop/eslint-learn/sample.js
  1:20  error  Strings must use singlequote             quotes
  1:28  error  Extra semicolon                          semi
  3:7   error  'info' is defined but never used         no-unused-vars
  3:28  error  '__filename' is not defined              no-undef
  3:44  error  '__dirname' is not defined               no-undef
  7:5   error  Expected indentation of 0 space ch...    indent
  8:9   error  Expected indentation of 4 space ch...    indent
  8:9   error  Unexpected console statement             no-console
  9:9   error  Expected indentation of 4 space ch...    indent

✖9 problems (9 errors, 0 warnings)
```

ESLint分析了我们的*sample.js*文件，并根据前面的配置选项报告了一些问题。这里我们可以看到ESLint指出文件中第一行代码中使用了双引号和分号，因为我们在*.eslintrc.yml*配置文件中仅指定了单引号，并且不使用分号。然后，ESLint指出虽然定义了info函数，但是它从没有被使用过，ESLint不喜欢存在这样的代码。ESLint还对文件和目录名颇有微词，因为它并没有自动引用全局的Node.js。最后，ESLint不喜欢switch语句中的缩进，以及console语句的使用。

我们可以修改ESLint的*.eslintrc.yml*配置，让它不至于那么严格：

```
env:
  browser: true
  commonjs: true
  es6: true
```

```
extends: 'eslint:recommended'
parserOptions:
  ecmaFeatures:
    experimentalObjectRestSpread: true
    jsx: true
    sourceType: module
  plugins:
    - react
  rules:
    indent:
      - error
      - 4
      - SwitchCase: 1
  quotes:
    - error
    - single
  semi:
    - error
    - never
  linebreak-style:
    - error
    - unix
  no-console: 0
globals:
    __filename: true
    __dirname: true
```

打开*.eslintrc.yml*文件后，用户将会发现该文件非常容易阅读，这也是YAML文件的目标。这里我们修改了代码缩进规则，以便在switch语句中能够使用缩进。然后我们添加了一条no-console规则，这会阻止ESLint对用户使用console.log语句时发出的警告信息。最后，我们添加了一对全局变量，要求ESlint忽略它们。

我们仍然需要对上述文件做出适当修改，以便遵循我们的代码风格规范:

```
const gnar = 'gnarly'

export const info = ({file= filename, dir= dirname}) =>
   <p>{dir}: {file}</p>

switch(gnar) {
    default :
        console.log('gnarly')
        break
}
```

我们将第一行的分号和双引号删除了。此外，导出info函数的信息意味着ESLint将不会向用户发送该函数未使用的警告信息了。在修改ESLint配置和编辑我的代码的过程中，我们就拥有了一个通过代码格式化测试的文件。

命令`eslint .`将会检查我们的整个目录。为此，用户很有可能需要ESLint忽略某些JavaScript文件。*.eslintignore*文件是用户可以添加需要ESLint忽略的文件或者目录名称的地方：

```
dist/assets/
sample.js
```

*.eslintignore*文件会告知ESLint忽略对新的*sample.js*文件的检查。这就和*dist/assets*文件夹下的其他任意文件一样。ESLint将会分析客户端*bundle.js*文件，并且会在该文件中报告大量警告信息。

让我们在*package.json*中添加一个脚本，以便可以运行代码检查命令：

```
"scripts": {
    "lint": "./node_modules/.bin/eslint ."
}
```

现在随时都可以使用`npm run lint`命令调用ESLint进行代码格式检查工作了，它会分析除了已经忽略的文件之外的所有项目文件。

测试Redux

测试对于Redux至关重要，因为它只和数据打交道，它并不包含UI。Redux天生就是可测试的，因为它的Reducer是纯函数，并且它可以方便地将State注入Store。编写一个Reducer的测试，首先使得用户更容易理解该Reducer的工作机制。为Store和Action生成器编写测试代码，可以提高用户对客户端数据层能够按照预期运行的信心。

在本节中，我们将会为颜色管理器的Redux组件编写一些单元测试。

测试驱动开发

测试驱动开发（Test-Driven Development，TDD）是一种实践而非技术。这并不意味着用户只需让自己的应用程序拥有测试代码即可。相反，它是一种测试驱动开发过程的实践。为了践行TDD，读者必须遵循以下步骤：

先写测试

这是最关键的一步。用户在一个测试中声明将要研发的内容和它的工作原理。

运行测试并观察它们的失误（红色）

在正式编码之前，运行测试用例并观察它们无法通过测试的情况。

编写最少量的必需代码，让它们通过测试（绿色）

现在，用户必须做的事情就是跑通测试用例。重点放在让每个测试通过上面，并且不要添加任何超过测试范围的功能。

重构上述程序代码和测试用例（金色）

一旦测试通过，那么就是该认真检查项目代码和测试用例的时候了，尽量将项目代码变得简洁优雅。[注2]

TDD是处理Redux应用程序绝佳方式。在编写实际的Reducer之前，通常更容易理解该Reducer的工作机制。践行TDD将允许用户独立于UI之外，构建和验证某个功能特性或者应用程序的整个数据结构。

TDD新手进阶技巧

如果你是一名TDD新手，或者对于正在测试的语言比较陌生，可能会发现在编写代码之前就编写测试用例非常具有挑战性。这是可以预见的，在你成为TDD老手之前，在测试之前编写项目代码并没有什么问题。先尝试一些小批量的测试：一小段代码，几个测试用例等。一旦可以熟练地编写测试用例之后，那么先写测试用例将会更容易一些。

本章需要提醒读者的是，我们将为已存在的代码编写测试。从技术上来说，我们并不是在践行TDD。不过，在下一节中我们将会假定项目代码不存在，以便让我们感受一下TDD工作流程。

测试Reducer

Reducer是根据输入参数进行一系列计算并返回结果的纯函数。在一个测试中，我们可以控制输入、当前State和Action。给定一个当前的State和Action，我们将能预计一个Reducer的输出结果。

在我们开始编写测试代码之前，将需要安装一个测试框架。用户可以使用任意JavaScript测试框架为React和Redux编写测试代码。我们将会使用Jest，它是为React量身定制的JavaScript测试框架：

```
sudo npm install -g jest
```

上述命令会在全局安装Jest和Jest CLI。用户可以在任意文件夹下执行jest命令对代码

注2：希望了解这种开发模式的详情，可以参考 Jeff McWherter和James Bender的文章，"Red, Green, Refactor"（*http://bit.ly/2kXvDN3*）。

进行测试了。因为我们使用的React库采用了若干JavaScript新特性，所以我们需要将项目代码和测试代码进行转译之后才能运行它们。只需安装babel-jest包即可：

```
npm install --save-dev babel-jest
```

babel-jest安装完毕之后，在运行测试之前，用户的所有项目代码和测试代码都会被Babel转译。这项工作必须用到一个*.babelrc*文件，不过我们的项目根目录下已经有一个这样的文件了。

create-react-app

使用create-react-app初始化的项目也同时安装了jest和babel-jest。同时也会在项目根目录下创建一个名为*tests*的目录。

Jest有两个重要的函数来设置测试：describe和it。describe可以用来创建一组测试用例，并且它对每个测试都适用。这两个函数都希望测试名称以及回调函数名称中包含相关的测试或者测试用例。

接下来将会创建一个测试文件，以便存放我们的测试代码。创建了一个名为*./__tests__/store/reducers*的文件夹，然后在其中创建一个名为*color.test.js*的JavaScript文件：

```
describe("color Reducer", () => {

    it("ADD_COLOR success")

    it("RATE_COLOR success")
})
```

在这个示例中，我们为color的Reducer创建了一个测试用例，对每个会对Reducer产生影响的Action做了测试。每个测试是用it函数定义的。用户可以通过只发送单个参数到it函数中来配置一个挂起测试。

使用jest命令运行该测试。jest将会执行，并报告跳过了2个挂起测试：

```
$ jest

Test Suites: 1 skipped, 0 of 1 total
Tests:       2 skipped, 2 total
Snapshots:   0 total
Time:        0.863s

Ran all test suites.
```

测试文件

jest将会运行tests文件中的任何测试，以及用户项目文件中以*.test.js*后缀结尾的任何JavaScript文件。有些开发人员偏向于将tests文件夹和它们正在测试的文件放在一起，有些则喜欢在单个文件夹下对测试分组。

现在是时候编写测试代码了。因为我们正在测试color的Reducer，将专门导入该函数。color的Reducer函数将会作为我们的被测系统（system under test，SUT）。我们将导入这个函数，给它发送一个Action，然后验证结果。

jest的“匹配器”通过expect函数返回，并可以用来验证返回结果。为了对color的Reducer进行测试，将会使用.toEqual匹配器。这可以验证结果对象和传递给.toEqual的参数是否匹配：

```
import C from '../../../src/constants'
import { color } from '../../../src/store/reducers'

describe("color Reducer", () => {

    it("ADD_COLOR success", () => {
        const state = {}
        const action = {
            type: C.ADD_COLOR,
            id: 0,
            title: 'Test Teal',
            color: '#90C3D4',
            timestamp: new Date().toString()
        }
        const results = color(state, action)
        expect(results)
            .toEqual({
                id: 0,
                title: 'Test Teal',
                color: '#90C3D4',
                timestamp: action.timestamp,
                rating: 0
            })
    })

    it("RATE_COLOR success", () => {
        const state = {
            id: 0,
            title: 'Test Teal',
            color: '#90C3D4',
            timestamp: 'Sat Mar 12 2016 16:12:09 GMT-0800 (PST)',
            rating: undefined
        }
        const action = {
            type: C.RATE_COLOR,
            id: 0,
```

```
            rating: 3
        }
        const results = color(state, action)
        expect(results)
            .toEqual({
                id: 0,
                title: 'Test Teal',
                color: '#90C3D4',
                timestamp: 'Sat Mar 12 2016 16:12:09 GMT-0800 (PST)',
                rating: 3
            })
    })

})
```

为了测试一个Reducer，我们需要一个State和样本Action。可以通过调用SUT来获得结果，即color函数和一些样本对象。最后，我们会使用.toEqual匹配器对执行结果进行验证，以便确保返回了相应的State。

为了测试ADD_COLOR，初始的State并不重要。不过当我们给color的Reducer发送了ADD_COLOR这样一个Action时，它应该返回一个新的color对象。

为了测试RATE_COLOR，我们将假定初始State会提供一个评分值为0的color对象。将这个State对象和RATE_COLOR一起发送之后应该会返回一个包含新评分值的color对象。

现在我们已经编写了测试代码，如果我们假装还没有编写color的Reducer代码，那么我们需要保留该函数。可以通过在*/src/Store/reducers.js*文件中添加一个名为color的函数来达到保留color的Reducer的目的。这使得我们的测试程序可以找到空的Reducer函数并导入它：

```
import C from '../constants'

export const color = (state={}, action=) => {
    return state
}
```

为什么首先要保留Reducer?

如果SUT不存在，我们会在测试中收到如下错误提示：

```
TypeError: (0 , _reducers.color) is not a function
```

该错误会在我们测试该函数时发生，color函数还未定义。添加被测试函数的定义将会获得更多有用的失败反馈信息。

让我们运行该测试，看看它为何报错。jest将会为每条错误提示提供具体的细节，包括堆栈跟踪：

```
$ jest

 FAIL __tests__/store/reducers/color.test.js
  .color Reducer › ADD_COLOR success

    expect(received).toEqual(expected)
    Expected value to equal:
      {"color": "#90C3D4", "id": 0, "rating": 0, "timestamp":
      "Mon Mar 13 2017 12:29:12 GMT-0700 (PDT)", "title": "Test Teal"}

    Received:
      {}

    Difference:

    - Expected
    + Received

    @@ -1,7 +1,1 @@
    -Object {
    -  "color": "#90C3D4",
    -  "id": 0,
    -  "rating": 0,
    -  "timestamp": "Mon Mar 13 2017 12:29:12 GMT-0700 (PDT)",
    -  "title": "Test Teal",
    -}

    +Object {}

        at Object.<anonymous> ( tests  /store/reducers/color.test.js:19:9)
        at process._tickCallback (internal/process/next_tick.js:103:7)

     ·color Reducer › RATE_COLOR success

      expect(received).toEqual(expected)

      Expected value to equal:
        {"color": "#90C3D4", "id": 0, "rating": 3, "timestamp":
        "Sat Mar 12 2016 16:12:09 GMT-0800 (PST)", "title": "Test Teal"}

      Received:
        {"color": "#90C3D4", "id": 0, "rating": undefined, "timestamp":
        "Sat Mar 12 2016 16:12:09 GMT-0800 (PST)", "title": "Test Teal"}

      Difference:

      - Expected
      + Received

      @@ -1,7 +1,7 @@
        Object {
          "color": "#90C3D4",
          "id": 0,
          "rating": 3,
        +  "rating": undefined,
```

```
      "timestamp": "Sat Mar 12 2016 16:12:09 GMT-0800 (PST)",
      "title": "Test Teal",
    }
    at Object.<anonymous> ( tests  /store/reducers/color.test.js:44:9)
    at process._tickCallback (internal/process/next_tick.js:103:7)

  color Reducer

    X ADD_COLOR success (8ms)
    X RATE_COLOR success (1ms)

Test Suites: 1 failed, 1 total
Tests:       2 failed, 2 total
Snapshots:   0 total
Time:        0.861s, estimated 1s
Ran all test suites.
```

投入时间编写测试和运行它们，以便查看它们显示的错误提示信息，这说明我们的测试是按照预期工作的。这些错误反馈就是我们的计划工作列表。用户的工作就是让程序代码通过这两个测试程序。

现在我们需要打开*/src/Store/reducers.js*文件，然后编写能够通过上述测试程序所需的基本代码：

```
import C from '../constants'

export const color = (state={}, action=) => {
    switch (action.type) {
        case C.ADD_COLOR:
            return {
                id: action.id,
                title: action.title,
                color: action.color,
                timestamp: action.timestamp,
                rating: 0
            }
        case C.RATE_COLOR:
            state.rating = action.rating
            return state
        default :
            return state
    }
}
```

下次我们执行jest命令后，项目代码将会通过测试程序的验证：

```
$ jest

PASS __tests__/store/reducers/color.test.js
  color Reducer

    ✓ ADD_COLOR success (4ms)
```

```
    ✓ RATE_COLOR success

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.513s, estimated 1s

Ran all test suites.
```

测试通过了，但是我们的工作还没有结束。现在应该对我们的项目代码和测试程序进行重构。来看看Reducer中的RATE_COLOR这类情况：

```
case 'RATE_COLOR':
  state.rating = action.rating
  return state
```

如果用户仔细观察，会发现这段代码有点问题。State应该是不可变对象，这里我们却为了对State对象中的颜色评分，从而修改了State的值。项目代码仍然可以通过测试的原因是，我们并没有确保State对象是不可变的。

deep-freeze(*https://github.com/substack/deep-freeze*)通过防止修改相关对象的方式，可以帮助用户确保State和Action对象的不可变性：

```
npm install deep-freeze --save-dev
```

当调用color的Reducer时，将会深度冻结State和Action对象。这两个对象将会是不可变的，深度冻结它们后，如果有任何代码要修改这些对象，将会导致程序抛出异常：

```
import C from '../../../src/constants'
import { color } from '../../../src/store/reducers'
import deepFreeze from 'deep-freeze'

describe("color Reducer", () => {

    it("ADD_COLOR success", () => {
        const state = {}
        const action = {
            type: C.ADD_COLOR,
            id: 0,
            title: 'Test Teal',
            color: '#90C3D4',
            timestamp: new Date().toString()
        }
        deepFreeze(state)
        deepFreeze(action)
        expect(color(state, action))
            .toEqual({
                id: 0,
                title: 'Test Teal',
                color: '#90C3D4',
```

```
                timestamp: action.timestamp,
                rating: 0
            })
    })

    it("RATE_COLOR success", () => {
        const state = { id: 0,
            title: 'Test Teal',
            color: '#90C3D4',
            timestamp: 'Sat Mar 12 2016 16:12:09 GMT-0800 (PST)',
            rating: undefined
        }
        const action = {
            type: C.RATE_COLOR,
            id: 0,
            rating: 3
        }
        deepFreeze(state)
        deepFreeze(action)
        expect(color(state, action))
            .toEqual({
                id: 0,
                title: 'Test Teal',
                color: '#90C3D4',
                timestamp: 'Sat Mar 12 2016 16:12:09 GMT-0800 (PST)',
                rating: 3
            })
    })

})
```

现在我们可以在当前的color的Reducer函数上运行修改过的测试代码，然后看看它报什么错，因为对颜色评分修改了传入的State对象：

```
$ jest

FAIL __tests__/store/reducers/color.test.js
 • color Reducer › RATE_COLOR success

   TypeError: Cannot assign to read only property 'rating' of object '#<Object>'
    at color (src/store/reducers.js:14:26)
    at Object.<anonymous> ( tests  /store/reducers/color.test.js:43:36)
    at process._tickCallback (internal/process/next_tick.js:103:7)

  color Reducer

    ✓ ADD_COLOR success (3ms)
    X RATE_COLOR success (3ms)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
Time:        0.513s, estimated 1s

Ran all test suites.
```

让我们对color的Reducer函数进行适当修改，以便可以通过测试程序的验证。我们将会使用扩展运算符在覆盖评分记录之前创建一个State对象的拷贝：

```
case 'RATE_COLOR':
    return {
        ...state,
         rating: action.rating
}
```

现在我们没有修改State对象，两个测试都应该可以通过：

```
$ jest

PASS __tests__/store/reducers/color.test.js

 color Reducer

   ✓ ADD_COLOR success (3ms)
   ✓ RATE_COLOR success

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.782s, estimated 1s

Ran all test suites.
```

这个过程代表了一个典型的TDD周期。我们首先编写测试代码，然后编写可以通过测试的项目代码，最后对项目代码和测试代码进行重构。在使用JavaScript工作时，该方法特别高效，特别对于Redux来说。

测试Store

如果Store能够正常运作，那么用户的应用程序也很有可能可以正常工作。测试Store的过程会创建一个和Reducer有关的Store，注入假想的State，分发Action，最后验证结果。

在测试Store时，用户可以集成Action生成器，将Store和Action生成器放在一起测试，达到一石二鸟的效果。

第8章我们创建了一个名为storeFactory的工厂类，它是一个可以用来在颜色管理器程序中管理Store创建过程的函数：

```
import { createStore,
         combineReducers,
         applyMiddleware } from 'redux'
import { colors, sort } from './reducers'
```

```
import stateData from '../../data/initialState'

const logger = store => next => action => {
    let result
    console.groupCollapsed("dispatching", action.type)
    console.log('prev state', store.getState())
    console.log('action', action)
    result = next(action)
    console.log('next state', store.getState())
    console.groupEnd()
    return result
}

const saver = store => next => action => {
    let result = next(action)
    localStorage['redux-store'] = JSON.stringify(store.getState())
    return result
}

const storeFactory = (initialState=stateData) =>
    applyMiddleware(logger, saver)(createStore)(
        combineReducers({colors, sort}),
        (localStorage['redux-store']) ?
            JSON.parse(localStorage['redux-store']) :
            initialState
    )

export default storeFactory
```

这个模块导出的一个函数可以用于创建Store。它封装了为颜色管理器创建Store的细节。这个文件包含Reducer、中间件，以及为我们应用程序创建Store必须的默认State。当使用`storeFactory`创建一个Store时，我们可以为新的Store传递一个初始的State，当测试该Store时，这将对用户有所帮助。

jest包含安装和拆卸功能，允许用户在执行每个测试或用例之前执行一些代码。`beforeAll`和`afterAll`方法会在每个测试用例执行前后分别触发。`beforeEach`和`afterEach`方法会在it语句执行之前或者之后触发。

安装和拆卸

编写测试代码时每个测试只允许出现一个断言是个好习惯。[注3]这意味着用户希望避免在单个`it`语句中出现多次调用的情况。通过这种方法，每个断言都可以独立验证，这使得当测试代码无法正确执行时，能够更容易找到症结所在。

jest的安装和拆卸特性可以用来帮助用户遵循这一实践。在一个`beforeAll`语句中执行测试代码，并使用多个`it`语句验证结果。

注3：参考Jay Fields的文章，“Testing: One Assertion per Test”, June 6, 2007（*http://bit.ly/2kuK2Nf*）。

让我们看看如何在测试./__tests__/actions-spec.js文件中addColor的Action生成器的同时，测试Store。接下来的示例将会通过分发一个addColor的Action生成器来测试Store，并对执行结果进行验证：

```
import C from '../src/constants'
import storeFactory from '../src/store'
import { addColor } from '../src/actions'

describe("addColor", () => {

    let store
    const colors = [
    {
        id: "8658c1d0-9eda-4a90-95e1-8001e8eb6036",
        title: "lawn",
        color: "#44ef37",
        timestamp: "Mon Apr 11 2016 12:54:19 GMT-0700 (PDT)",
        rating: 4
    },
    {
        id: "f9005b4e-975e-433d-a646-79df172e1dbb",
        title: "ocean blue",
        color: "#0061ff",
        timestamp: "Mon Apr 11 2016 12:54:31 GMT-0700 (PDT)",
        rating: 2
    },
    {
        id: "58d9caee-6ea6-4d7b-9984-65b145031979",
        title: "tomato",
        color: "#ff4b47",
        timestamp: "Mon Apr 11 2016 12:54:43 GMT-0700 (PDT)",
        rating: 0
    }
]

beforeAll(() => {
    store = storeFactory({colors})
    store.dispatch(addColor("Dark Blue", "#000033"))
})

it("should add a new color", () =>
    expect(store.getState().colors.length).toBe(4))

it("should add a unique guid id", () =>
    expect(store.getState().colors[3].id.length).toBe(36))

it("should set the rating to 0", () =>
    expect(store.getState().colors[3].rating).toBe(0))

it("should set timestamp", () =>
    expect(store.getState().colors[3].timestamp).toBeDefined())
})
```

我们通过storeFactory创建了一个新的Store实例来配置该测试的，该Store中的State包含三种示例颜色。接下来，我们分发了Action生成器addColor添加了第4种颜色到State中：深蓝色。现在每个测试都会对分发的Action进行验证。每个测试中都包含一个expect语句。如果其中任意一个测试未能成功执行，我们将会确切地知道是哪个新的Action出了问题。

这次我们会使用两个新的匹配器：.toBe和.toBeDefined。.toBe匹配器会使用操作符===比较结果。该匹配器还可以用来比较诸如数字或者字符串这类基元类型，而.toEqual匹配器可以用于深入比较对象。

.toBeDefined匹配器可以用来检查现变量或者函数的存在性。在这个测试中，我们会检查是否存在时间戳。

这些测试可以验证我们的Store是否可以使用Action生成器添加新的颜色。这可以提高我们对Store部分程序代码的信心，它能够正常工作。

测试React组件

React组件为React提供了一组在创建和管理DOM时所需的指令集。我们可以通过渲染和检查最终的DOM来测试这些组件。

我们并没有在浏览器中运行测试代码，而是在一个Node.js终端上运行的。Node.js并没有为每个浏览器标准的DOM API提供支持。jest集成了一个名为jsdom的npm软件包，它可以用于在Node.js中模拟浏览器环境，这对于测试React组件至关重要。

配置jest环境

jest为用户提供了在运行任何测试代码之前运行脚本的能力，我们可以在其中设置额外的全局变量，以便可以在后续的测试代码中使用。

比如，假设我们希望将React和一些样本颜色添加到全局作用域上，以便任意测试程序都能访问它们。我们可以创建一个名为*/__tests__/global.js*的文件：

```
import React from 'react'
import deepFreeze from 'deep-freeze'

global.React = React
global._testColors = deepFreeze([
    {
        id: "8658c1d0-9eda-4a90-95e1-8001e8eb6036",
        title: "lawn",
```

```
        color: "#44ef37",
        timestamp: "Sun Apr 10 2016 12:54:19 GMT-0700 (PDT)",
        rating: 4
    },
    {
        id: "f9005b4e-975e-433d-a646-79df172e1dbb",
        title: "ocean blue",
        color: "#0061ff",
        timestamp: "Mon Apr 11 2016 12:54:31 GMT-0700 (PDT)",
        rating: 2
    },
    {
        id: "58d9caee-6ea6-4d7b-9984-65b145031979",
        title: "tomato",
        color: "#ff4b47",
        timestamp: "Fri Apr 15 2016 12:54:43 GMT-0700 (PDT)",
        rating: 0
    }
])
```

该文件添加了React和一些不可变的测试颜色信息到全局作用域。接下来，我们将会告知jest在运行测试代码之前运行这个文件。为此我们只需在*package.json*文件下的jset节点中添加一个setupFiles字段即可：

```
"jest": {
  "setupFiles": ["./ tests /global.js"],
  "modulePathIgnorePatterns": ["global.js"]
}
```

setupFiles字段可以用来提供一个文件名数组，这些文件是Jest在运行测试程序之前，为了搭建全局环境应该执行的内容。modulePathIgnorePatterns字段告知Jest在运行测试时应该忽略*global.js*文件，因为它不包含任何测试用例，它只是一个配置文件。该字段是必需的，因为用户一般都倾向于将*global.js*文件添加到测试文件夹中，即使它不包含任何测试代码。

忽略导入的SCSS文件

如果用户直接将SCSS（CSS或SASS）文件导入了组件，那么在进行代码测试时需要将它们忽略，因为它们会导致测试程序执行失败。

当导入了*.css*、*.scss*或者*.less*这类文件时，可以通过集成一个返回空字符串的模块映射器来忽略它们。我们将安装jest-css-modules这个模块：

```
npm install jest-css-modules --save-dev
```

我们已经安装了这个模块，现在需要告知Jest使用这个模块来替换任意导入的*.scss*文件。我们需要在*package.json*文件中的jest节点下添加一个moduleNameMapper字段：

```
"jest": {
  "setupFiles": ["./ tests /global.js"],
  "modulePathIgnorePatterns": ["global.js"],
  "moduleNameMapper": {
    "\\.(scss)$":   "<rootDir>/node_modules/jest-css-modules"
  }
}
```

这会告知jest使用`jest-css-modules`模块替换任何导入文件后缀为*.scss*的文件。将上述代码添加到*package.json*文件中后，将会避免测试程序因为导入的*.scss*文件而无法正确执行。

Enzyme

我们几乎已经为开始测试React组件做好了准备。在开始编写我们的第一个组件测试程序之前，我们还需要在安装两个npm模块：

```
npm install enzyme react-addons-test-utils --save-dev
```

Enzyme(*http://airbnb.io/enzyme/*)是Airbnb推出的专门用于测试React组件的工具。Airbnb需要用到`react-addons-test-utils`，它们是测试过程中用于渲染UI和与组件交互的一组工具。此外，还需要用到`react-dom`，不过我们将假定用户已经安装了`react-dom`。

Enzyme使得渲染组件和遍历被渲染的输出结果更容易。Enzyme并不是一个测试或者断言框架。它为测试程序处理渲染React组件的任务，并为遍历子元素、验证属性、验证State、模拟事件和DOM查询提供了必要的工具。

它主要包含三种用于渲染的方法：

`shallow`
: shallow为单元测试在一定层级深度上渲染组件。

`mount`
: mount会使用浏览器DOM渲染组件，当用户需要测试完整的组件生命周期，以及子元素的属性或者State时，这是必需的。

`render`
: render用于渲染包含静态HTML标记的组件。通过render，用户可以对组件验证是否能够返回相应的HTML。

比如Star组件：

```
const Star = ({ selected=false, onClick=f=>f }) =>
    <div className={(selected) ? "star selected" : "star"}
         onClick={onClick}>
</div>
```

它应该会渲染一个根据选定属性获得className的div元素。它还会响应click事件。

让我们使用Enzyme为Star组件编写一个测试。将会使用Enzyme渲染组件，并找到包含已被渲染星标的特定DOM元素。我们可以使用shallow方法在一定程度上对上述组件进行渲染：

```
import { shallow } from 'enzyme'
import Star from '../../../src/components/ui/Star'

describe("<Star /> UI Component", () => {

    it("renders default star", () =>
        expect(
            shallow(<Star />)
                .find('div.star')
                .length
        ).toBe(1)
    )

    it("renders selected stars", () =>
        expect(
            shallow(<Star selected={true} />)
                .find('div.selected.star')
                .length
        ).toBe(1)
    )

})
```

Enzyme具有和jQuery类似的某些功能。我们可以使用find方法和选择器语法对最终的DOM进行查找。

在第一个测试中，某个样本星标被渲染了，我们会在DOM中验证其中是否存在一个包含star类的div元素。在第二个测试中，一个选中状态下的样本星标被渲染了，我们可以验证DOM中是否存在一个包含star类和selected类的div元素。检查长度的目的是确保每个测试中只有一个地址被渲染了。

接下来，我们将会测试click事件。Enzyme附带的某些工具允许用户模拟事件，然后验证这些事件是否被触发了。对于这个测试，我们需要一个函数来验证onClick属性是否能够正常运作。我们需要一个模拟函数，并且jest也为用户考虑到了这一点：

```
it("invokes onClick", () => {
```

```
    const _click = jest.fn()

    shallow(<Star onClick={_click} />)
            .find('div.star')
            .simulate('click')

    expect(_click).toBeCalled()

  })
```

在这个测试中的一个模拟函数_click，是通过jest.fn创建的。当我们渲染星标时，将上述模拟函数作为onClick的属性传递给了它。接下来，对被渲染的div元素进行定位，然后使用Enzyme的simulate方法在该元素上模拟了一个click事件。单击星标会触发onClick属性，继而触发执行模拟函数。.toBeCalled匹配器可以用来验证该模拟函数是否被触发了。

Enzyme可以用来帮助用户渲染组件，查找被渲染的DOM元素或者其他组件，并且可以和它们进行交互。

模拟组件

上一个测试引入了模拟的概念：我们使用一个模拟函数测试Star组件。Jest包含丰富的工具，它们可以帮助用户创建和注入各种不同的模拟效果，以便用户可以写出更好的测试代码。模拟对象是一种非常重要的测试技术，它可以帮助用户聚焦于单元测试。模拟对象就是可以用来替换测试中真实对象的对象。[注4]

模拟对象对于测试的世界来说就像是好莱坞的动作替身演员。模拟对象和替身演员都可以用来替代真身（一个组件或者影视明星）。在一部电影中，替身看上去和演员本人没什么两样。在一个测试中，模拟对象可以被当作一个真实的对象。

模拟的目的是允许用户能够聚焦于被测试的组件或者对象，即SUT。模拟对象是用于替换SUT相关的对象、组件或者函数的。这使得用户可以在没有任何来自SUT引用依赖干扰的情况下对SUT进行验证。模拟对象还允许用户在和其他组件隔离的情况下，隔离、构建和测试相关功能。

测试HOC

我们将会用到模拟对象的一个地方是在测试高阶组件时。HOC负责通过属性注入的方

注4：希望深入了解模拟对象技术，可以参考Martin Fowler的博文，"Mocks Aren't Stubs"（*http://bit.ly/2kuR98s*）。

式为组件添加功能。我们可以创建一个模拟组件，然后将它传递给某个HOC，以此来验证HOC添加相应属性到模拟组件后的结果。

让我来看看一个可折叠性测试，回顾一下第7章创建的HOC。为了给这个HOC建立一个测试，我们首先必须创建一个模拟组件，然后发送给这个HOC。MockComponent组件将会是用来替换真实组件的替身：

```
import { mount } from 'enzyme'
import Expandable from '../../../src/components/HOC/Expandable'

describe("Expandable Higher-Order Component", () => {

    let props,
        wrapper,
        ComposedComponent,
        MockComponent = ({collapsed, expandCollapse}) =>
            <div onClick={expandCollapse}>
                {(collapsed) ? 'collapsed' : 'expanded'}
            </div>

    describe("Rendering UI", ... )

    describe("Expand Collapse Functionality", ... )

})
```

MockComponent只是一个非常简单的无状态函数式组件，我们正处于研发过程中。它会返回一个包含onClick事件句柄的div，将会用于测试展开或折叠函数。展开或者折叠的状态也会在模拟组件中显示。这个组件除了在这个测试中使用之外，在其他任何地方是不会使用的。

SUT是可折叠的HOC。在测试前，将会使用模拟组件调用HOC，并对返回的组件进行检查，看它是否应用了相应的属性。

将会使用mount函数替代shallow函数，以便能够检查返回组件的属性和State：

```
describe("Rendering UI", () => {

    beforeAll(() => {
        ComposedComponent = Expandable(MockComponent)
        wrapper = mount(<ComposedComponent foo="foo" gnar="gnar"/>)
        props = wrapper.find(MockComponent).props()
    })

    it("starts off collapsed", () =>
        expect(props.collapsed).toBe(true)
    )

    it("passes the expandCollapse function to composed component", () =>
```

```
        expect(typeof props.expandCollapse)
            .toBe("function")
        )

        it("passes additional foo prop to composed component", () =>
            expect(props.foo)
                .toBe("foo")
        )

        it("passes additional gnar prop to composed component", () =>
            expect(props.gnar)
                .toBe("gnar")
        )

    })
```

一旦我们使用HOC创建了一个合成组件，可以通过挂载它，然后直接检查属性对象的方式来确认该合成组件是否将相应属性添加到了模拟组件中。这个测试将会确保HOC已经添加了collapsed属性，以及用于修改该属性的方法expandCollapse。它还会验证是否可以添加任意属性到合成组件中，比如foo和gnar，以便确保它们能够兼容模拟对象。

接下来，让我验证一下是否可以修改合成组件的collapsed属性：

```
describe("Expand Collapse Functionality", () => {

    let instance

    beforeAll(() => {
        ComposedComponent = Expandable(MockComponent)
        wrapper = mount(<ComposedComponent collapsed={false}/>)
        instance = wrapper.instance()
    })

    it("renders the MockComponent as the root element", () => {
        expect(wrapper.first().is(MockComponent))
    })

    it("starts off expanded", () => {
        expect(instance.state.collapsed).toBe(false)
    })

    it("toggles the collapsed state", () => {
        instance.expandCollapse()
        expect(instance.state.collapsed).toBe(true)
    })

})
```

一旦我们挂载了组件，可以通过wrapper.instance收集被渲染实例的信息。在这种情

况下，我们希望组件的起始状态是已折叠的。 可以检查实例的属性和State，以便确保该组件起始状态是已折叠的。

wrapper还包含以下可以用来遍历DOM的方法。在第一个测试中，我们使用wrapper.first对第一个子元素进行定位，并且验证了该元素是MockComponent组件的一个实例。

HOC是使用模拟对象的极佳场景，因为注入模拟对象的过程非常简单：只需将它作为参数发给HOC即可。私有模拟组件的概念与之类似，不过注入过程需要一定技巧。

jest模拟

jest允许用户将模拟对象注入任何组件，不仅限于HOC。通过jest, 用户可以模拟SUT导入的任何模块。模拟技术允许用户聚焦于正在测试的SUT上，而不是其他可能潜在导致问题的模块。

比如，我们来看看ColorList组件，它导入了Color组件：

```
import { PropTypes } from 'react'
import Color from './Color'
import '../../../stylesheets/ColorList.scss'

const ColorList = ({ colors=[], onRate=f=>f, onRemove=f=>f }) =>
    <div className="color-list">
        {(colors.length === 0) ?
            <p>No Colors Listed. (Add a Color)</p> :
            colors.map(color =>
                <Color key={color.id}
                    {...color}
                        onRate={(rating) => onRate(color.id, rating)}
                        onRemove={() => onRemove(color.id)} />
            )
        }
    </div>

ColorList.propTypes = {
    colors: PropTypes.array,
    onRate: PropTypes.func,
    onRemove: PropTypes.func
}

export default ColorList
```

我们希望确保ColorList组件能够正常运行。我们并不关心Color组件，它应该有自己的单元测试程序。我们可以为ColorList编写一个测试，并使用一个模拟对象替换Color组件：

```
import { mount } from 'enzyme'
import ColorList from '../../../src/components/ui/ColorList'

jest.mock('../../../src/components/ui/Color', () =>
  ({rating, onRate=f=>f}) =>
    <div className="mock-color">
      <button className="rate" onClick={() => onRate(rating)} />
    </div>
)

describe("<ColorList /> UI Component", () => {

    describe("Rating a Color", () => {

        let _rate = jest.fn()

        beforeAll(() =>
            mount(<ColorList colors={_testColors} onRate={_rate} />)
                .find('button.rate')
                .first()
                .simulate('click')
        )

        it("invokes onRate Handler", () =>
            expect(_rate).toBeCalled()
        )

        it("rates the correct color", () =>
            expect(_rate).toBeCalledWith(
                "8658c1d0-9eda-4a90-95e1-8001e8eb6036",
                4
            )
        )

    })

})
```

在这个测试中，我们使用jest.mock注入了一个模拟对象来替换实际的Color组件。传递给jest.mock的第一个参数是用户希望模拟的模块，第二个参数是一个返回模拟组件的函数。在这种情况下，Color的模拟对象是一个精简版的Color组件。这个测试只聚焦于给颜色评分的功能，因此模拟对象只处理了和颜色评分相关的属性。

当运行这个测试时，jest将会使用模拟对象替换Color组件。在渲染ColorList时，将会发送一个前文所述的全局变量_testColors。当ColorList渲染每种颜色时，我们的模拟对象将会被新渲染的元素替代。当在第一个按钮上触发一个click事件后，该事件将会在我们的第一个模拟对象上执行。

该组件被渲染的DOM将会和下列内容类似：

```
<ColorList>
  <div className="color-list">
    <MockColor onRate={[Function]} rating={4}>
      <div className="mock-color">
        <button id="rate" onClick={[Function]} />
      </div>
    </MockColor>
    <MockColor onRate={[Function]} rating={2}>
      <div className="mock-color">
        <button id="rate" onClick={[Function]} />
      </div>
    </MockColor>
    <MockColor onRate={[Function]} rating={0}>
      <div className="mock-color">
        <button id="rate" onClick={[Function]} />
      </div>
    </MockColor>
  </div>
</ColorList>
```

实际的Color组件将会把被渲染的评分数据传递给ColorList，但是我们的模拟对象并没有使用StarRating组件。它不能对颜色评分，相反，它会通过将当前评分值回传给ColorList假装执行了评分操作。在这个测试中我们的重点并不在Color组件上，我们只关心ColorList。ColorList的行为符合预期。单击第一种颜色后，将正确的评分记录传递给了onRate属性。

手工模拟

jest允许用户为模拟对象创建模块。从而替代直接将模拟对象的代码添加到测试程序中的做法。在一个mocks文件夹中，将每个模拟对象代码放在专属于其自身的文件。当然jest还应该可以找到该mocks文件夹。

我们来看看第9章创建的*/src/components/containers.js*文件。该文件包含三个容器。对于接下来的测试，我们将聚焦于Colors容器：

```
import ColorList from './ui/ColorList'

export const Colors = connect(
    state =>
        ({
            colors: [...state.colors].sort(sortFunction(state.sort))
        }),
    dispatch =>
        ({
            onRemove(id) {
                dispatch(removeColor(id))
            },
            onRate(id, rating) {
                dispatch(rateColor(id, rating))
```

```
            }
        })
)(ColorList)
```

Colors容器可以用于将数据从Store连接到ColorList组件。它会对State中的颜色排序，然后将之作为属性发送给ColorList。它还会处理ColorList中onRate和onRemove函数的属性。最后，该容器会依赖于ColorList模块。

我们可以在*__mocks__*文件夹下手动添加一个*<Module>.js*文件。该*__mocks__*文件夹包含用于替换测试过程中实际模块的模拟对象模块。

比如，我们会在ColorList组件同级目录下的*/src/components/ui*文件夹下创建一个*__mocks__*文件夹，然后在其中添加一个ColorList的模拟对象。我们将会把模拟代码和*ColorList.js*放在上述文件下。

我们的模拟对象将只会简单地渲染一个空的div元素。来看看这个*ColorList.js*的模拟代码：

```
const ColorListMock = () => <div className="color-list-mock"></div>

ColorListMock.displayName = "ColorListMock"

export default ColorListMock
```

现在，当我们使用jest.mock对*/src/components/ui/ColorList*组件进行模拟时，jest会从*__mocks__*文件下获取相应的模拟信息。我们将不必在测试代码中直接定义模拟对象。除了手动模拟ColorList，我们还会为Store创建一个模拟对象。Store包含三个重要的函数：dispatch，subscribe和getState。getState函数为模拟函数提供了一个具体实现，可以使用全局的测试颜色返回一个样本State。

我们将会使用该模拟State测试容器。同时还将会把模拟Store作为属性渲染一个Provider组件。我们的容器将会从Store中获取颜色信息，然后对颜色进行排序，最后将它们传递给模拟对象：

```
import { mount, shallow } from 'enzyme'
import { Provider } from 'react-redux'
import { Colors } from '../../../src/components/containers'

jest.mock('../../../src/components/ui/ColorList')

describe("<Colors /> Container ", () => {

    let wrapper
    const _store = {
        dispatch: jest.fn(),
```

```
        subscribe: jest.fn(),
        getState: jest.fn(() =>
            ({
                sort: "SORTED_BY_DATE",
                colors: _testColors
            })
        )
    }

    beforeAll(() => wrapper = mount(
      <Provider store={_store}>
          <Colors />
      </Provider>
    ))

    it("renders three colors", () => {
        expect(wrapper
            .find('ColorListMock')
            .props()
            .colors
            .length
        ).toBe(3)
    })

    it("sorts the colors by date", () => {
        expect(wrapper
            .find('ColorListMock')
            .props()
            .colors[0]
            .title
        ).toBe("tomato")
    })

})
```

在这个测试中，我们使用jest.mock模拟了ColorList组件，不过只给它传递了一个参数：模拟模块的路径。jest知道应该去*__mocks__*文件夹下查找具体的模拟代码实现。我们不再需要使用真实的ColorList了，只需使用光板的模拟组件即可。一旦渲染完毕，DOM元素将会和下列代码类似：

```
<Provider>
  <Connect(ColorListMock)>
    <ColorListMock colors={[...]}
      onRate={[Function]}
      onRemove={[Function]}>
        <div className="color-list-mock" />
    </ColorListMock>
  </Connect(ColorListMock)>
</Provider>
```

如果我们的容器可以正常工作，它将会发送三种颜色到模拟对象中。容器将会根据日

期对这些颜色进行排序。我们可以通过查看“番茄红”是否排在第一位来证明这一点，因为在_testColors中的三种颜色中，它包含最新的时间戳。

让我们再添加一个测试来确保onRate和onRemove能够正常工作：

```
afterEach(() => jest.resetAllMocks())

it("dispatches a REMOVE_COLOR action", () => {
    wrapper.find('ColorListMock')
        .props()
        .onRemove('f9005b4e-975e-433d-a646-79df172e1dbb')

    expect(_store.dispatch.mock.calls[0][0])
        .toEqual({
            id: 'f9005b4e-975e-433d-a646-79df172e1dbb',
            type: 'REMOVE_COLOR'
        })
})

it("dispatches a RATE_COLOR action", () => {
    wrapper.find('ColorListMock')
        .props()
        .onRate('58d9caee-6ea6-4d7b-9984-65b145031979',   5)

    expect(_store.dispatch.mock.calls[0][0])
        .toEqual({
            id: "58d9caee-6ea6-4d7b-9984-65b145031979",
            type: "RATE_COLOR",
            rating: 5
        })
})
```

为了测试onRate和onRemove，我们不需要模拟实际的单击事件。需要我们做的只是使用一些信息来触发这些函数的属性，然后验证Store的dispatch方法被调用时是否使用了正确的数据。

调用onRemove属性将会导致Store分发一个REMOVE_COLOR这样的Action。调用onRate属性将会导致Store分发RATE_COLOR这样一个Action。此外，我们还需要确保每次测试完成后dispatch模拟对象被重置了。

能够轻松地将模拟对象注入用户希望测试的模块是jest最强大的特性之一。模拟对象是将测试聚焦于SUT的一种非常高效的技术。

快照测试

测试驱动开发是一个测试辅助函数、自定义类和数据集的好方法。然而，在对UI进行测试时，TDD可能会变得比较棘手，并且往往是不切实际的。UI频繁的变更，使得维

护UI测试变成了一种需要耗费大量时间的工作。为已经正式上线的UI组件编写测试的做法也越来越普遍。

快照测试为用户提供了一种可以快速测试UI组件的方式，以便确保开发人员没有给组件带来任何不符合预期的变更。jest可以为已渲染的UI保存一个快照，以便可以将它和将来测试中的渲染结果进行比较。这允许用户确保程序更新没有带来任何意想不到的效果，同时开发人员还可以快速推进项目，并且不会因为UI测试的实际工作而陷入困境。此外，当UI变更符合预期时，快照还可以方便地被更新。

我们来看看如何使用快照测试对Color组件进行测试的。首先，让我们考察现有的Color组件代码：

```
import { PropTypes, Component } from 'react'
import StarRating from './StarRating'
import TimeAgo from './TimeAgo'
import FaTrash from 'react-icons/lib/fa/trash-o'
import '../../../stylesheets/Color.scss'

class Color extends Component {

    render() {
        const {
            title, color, rating, timestamp, onRemove, onRate
        } = this.props

        return (
            <section className="color" style={this.style}>
                <h1 ref="title">{title}</h1>
                <button onClick={onRemove}>
                    <FaTrash />
                </button>
                <div className="color"
                    style={{ backgroundColor: color }}>
                </div>
                <TimeAgo timestamp={timestamp} />
                <div>
                    <StarRating starsSelected={rating} onRate={onRate}/>
                </div>
            </section>
        )
    }

}
export default Color
```

如果我们使用特定属性渲染这个组件，预期获得的DOM应该包含基于我们发送的属性获得的特定组件：

```
shallow(
```

```
        <Color title="Test Color"
            color="#F0F0F0"
            rating={3}
            timestamp="Mon Apr 11 2016 12:54:19 GMT-0700 (PDT)"
        />
    ).html()
```

生成的DOM应该和下列代码类似：

```
<section class=\"color\">
  <h1>Test Color</h1>
  <button><svg /></button>
  <div class=\"color\" style=\"background-color:#F0F0F0;\"></div>
  <div class=\"time-ago\">4/11/2016</div>
  <div>
    <div class=\"star-rating\">
      <div class=\"star selected\"></div>
      <div class=\"star selected\"></div>
      <div class=\"star selected\"></div>
      <div class=\"star\"></div>
      <div class=\"star\"></div>
      <p>3 of 5 stars</p>
    </div>
  </div>
</section>
```

快照测试允许用户保存我们首次运行某个测试后生成的DOM快照。然后我们就可以把将来的测试和上述快照进行比较，从而确保输出结果总是能够保持一致。

让我们为Color组件编写一个快照测试程序：

```
import { shallow } from 'enzyme'
import Color from '../../../src/components/ui/Color'

describe("<Color /> UI Component", () => {

    it("Renders correct properties", () =>
        let output = shallow(
            <Color title="Test Color"
                color="#F0F0F0"
                rating={3}
                timestamp="Mon Apr 11 2016 12:54:19 GMT-0700 (PDT)"
            />
        ).html()

        expect(output).toMatchSnapshot()
    )

})
```

在这个测试中，我们使用Enzyme渲染组件，并将收集到的测试结果作为HTML的字符串处理。.toMatchSnapshot是jest用于处理快照测试的匹配器。该测试首次运行时，

jset将会把测试结果的HTML另存为一个快照文件。该文件将会保存到与该测试同级目录下的*__snapshots__*文件夹下。目前，快照文件内容和下列内容类似：

```
exports[`<Color /> UI Component Renders correct properties 1`] =
    `"<section class=\"color\"><h1>Test Color</h1><button><svg ...
```

后续每次运行该测试时，jest将会把测试结果和上述快照进行比较。如果测试结果中存在和快照不一致的内容，那么该测试将无法成功执行。

快照测试允许用户快速推进项目，但是如果移动得太快，可能会导致开发人员写出一些稀奇古怪的测试，或者本应该通过的测试也无法正常执行。捕捉HTML字符的快照能够对测试起作用，但是用户很难验证快照是否能够真实地反映实际情况。让我们通过将输出结果另存为JSX的方式来对快照进行改良。

为此，我们将需要安装enzyme-to-json模块：

```
npm install enzyme-to-json --save-dev
```

该模块提供了一个函数，我们可以通过它将Enzyme包装器渲染成JSX格式，这使得用户可以更方便地验证快照输出结果的正确性。

为了使用enzyme-to-json渲染快照，我们首先会使用Enzyme对Color组件进行浅层渲染，然后将结果发送给toJSON函数，toJSON函数的程序执行结果将会传递给expect函数。我们可能还需要编写一些临时性的代码：

```
expect(
  toJSON(
    shallow(
      <Color title="Test Color"
             color="#F0F0F0"
             rating={3}
             timestamp="Mon Apr 11 2016 12:54:19 GMT-0700 (PDT)"
          />
    )
  )
).toMatchSnapshot()
```

不过这是一个采用合成技术对代码进行改进的绝佳应用场景。还记得合成么？若干小型函数构造了一个更大的函数。我们可以使用Redux的compose函数将shallow、toJSON和expect函数构造成一个更大的函数：

```
import { shallow } from 'enzyme'
import toJSON from 'enzyme-to-json'
import { compose } from 'redux'
import Color from '../../../src/components/ui/Color'
```

```
describe("<Color /> UI Component", () => {

    const shallowExpect = compose(expect,toJSON,shallow)

    it("Renders correct properties", () =>
        shallowExpect(
          <Color title="Test Color"
              color="#F0F0F0"
              rating={3}
              timestamp="Mon Apr 11 2016 12:54:19 GMT-0700 (PDT)"
          />
        ).toMatchSnapshot()
    )

})
```

shallowExpect函数接收一个组件，对它进行浅层渲染并将结果转化为JSON格式，然后将它发送给expect函数，最后返回结果给所有Jest匹配器。

如果我们运行该测试，它将无法正确执行，因为输出结果是JSX格式而非HTML字符串。我们的测试不再需要匹配快照。不过，快照的更新也很容易。我们可以通过再次执行该测试时声明updateSnapshot选项来对快照进行更新。

```
jest --updateSnapshot
```

如果我们在执行Jest命令时附加了watch选项：

```
jest --watch
```

jest将会继续在终端运行，并监听用户对源码和测试程序的变更。如果用户生成了任何新的变更，jest将会再次运行测试。当然用户监控测试时，可以很容易地通过按下u键来更新快照：

```
Snapshot Summary
  › 1 snapshot test failed in 1 test suite. Inspect your code changes or press
  `u` to update them.

Test Suites: 1 failed, 6 passed, 7 total
Tests:       1 failed, 28 passed, 29 total
Snapshots:   1 failed, 1 total Time:   1.407s
Ran all test suites.

Watch Usage
  › Press u to update failing snapshots.
  › Press p to filter by a filename regex pattern.
  › Press q to quit watch mode.
  › Press Enter to trigger a test run.
```

一旦用户更新了快照，测试将会顺序通过。快照文件也发生了变化。与一个长长的HTML字符串相反，快照内容将会和下列内容类似：

```
exports[`<Color /> UI Component Renders correct properties 1`] = `
<section
  className="color">
  <h1>
    Test Color
  </h1>
  <button
    onClick={[Function]}>
    <FaTrashO />
  </button>
  <div
    className="color"
    style={
      Object {
        "backgroundColor": "#F0F0F0",
      }
    } />
  <TimeAgo
    timestamp="Mon Apr 11 2016 12:54:19 GMT-0700 (PDT)" />
  <div>
    <StarRating
      onRate={[Function]}
      starsSelected={3} />
  </div>
</section>
';
```

这个快照的内容更容易阅读理解。我们可以对它进行浏览，以便用户在移动到下一个测试之前验证测试结果的正确性。快照测试是快速添加测试到应用程序的一种非常有效的方法。

代码覆盖率测试

代码覆盖率测试是统计项目源码实际被测试的行数的过程。它提供的一种指标可以帮助开发人员确定编写了足够多的测试程序的时机。

jest和Istanbul一起协同工作，后者是一个用来检查测试程序，并生成一个统计报表，其中描述了有多少语句、分支、函数和代码行被覆盖。

为了让jest执行代码覆盖率统计功能，在执行jest命令时附加coverage选项即可：

```
jest --coverage
```

上述命令将会生成一个当前代码覆盖率的统计报表并显示在终端上：

```
PASS __tests__/components/ui/ColorList.test.js
PASS __tests__/components/containers/Colors.test.js
PASS __tests__/components/ui/Color.test.js
PASS __tests__/components/ui/Star.test.js
PASS __tests__/components/HOC/Expandable.test.js
PASS __tests__/actions.test.js
PASS __tests__/store/reducers/color.test.js

--------------------|--------|----------|----------|----------|----------------|
File                |% Stmts | % Branch |  % Funcs |  % Lines |Uncov'd Lines   |
--------------------|--------|----------|----------|----------|----------------|
All files           |  68.42 |    43.33 |    45.59 |    72.39 |                |
 src                |    100 |      100 |      100 |      100 |                |
  actions.js        |    100 |      100 |      100 |      100 |                |
  constants.js      |    100 |      100 |      100 |      100 |                |
 src/components     |  58.33 |      100 |       40 |    58.33 |                |
  containers.js     |  58.33 |      100 |       40 |    58.33 | 11,13,20,24,26 |
 src/components/HOC |    100 |      100 |      100 |      100 |                |
  Expandable.js     |    100 |      100 |      100 |      100 |                |
 src/components/ui  |  45.65 |    35.29 |       24 |       50 |                |
  AddColorForm.js   |  16.67 |        0 |        0 |    18.18 |... 13,16,18,21 |
  Color.js          |  66.67 |      100 |    33.33 |    66.67 |          40,41 |
  ColorList.js      |   62.5 |       40 |       50 |    83.33 |             13 |
  SortMenu.js       |   37.5 |        0 |        0 |    42.86 |    11,14,18,19 |
  Star.js           |    100 |      100 |      100 |      100 |                |
  StarRating.js     |  33.33 |        0 |        0 |       40 |          5,7,9 |
  TimeAgo.js        |     50 |      100 |        0 |       50 |              4 |
 src/lib            |  58.54 |       15 |    16.67 |    67.65 |                |
  array-helpers.js  |     60 |    33.33 |       60 |    71.43 |            6,8 |
  time-helpers.js   |  58.06 |        0 |        0 |    66.67 |... 43,45,49,54 |
 src/store          |  97.14 |       70 |      100 |    96.77 |                |
  index.js          |    100 |      100 |      100 |      100 |                |
  reducers.js       |  94.12 |    64.71 |      100 |    94.12 |             21 |
--------------------|--------|----------|----------|----------|----------------|

Test Suites: 7 passed, 7 total
Tests:       29 passed, 29 total
Snapshots:   1 passed, 1 total
Time:        1.691s, estimated 2s

Ran all test suites.

Watch Usage
  › Press p to filter by a filename regex pattern.
  › Press q to quit watch mode.
  › Press Enter to trigger a test run.
```

这个报告告诉用户在执行测试过程中每个文件中被执行的代码有多少行，同时还报告了被导入测试的所有文件。

jest还生成了一个可以在浏览器中运行的报告，它提供了更多被测试覆盖的代码的详细信息。在执行jest的代码覆盖率报告命令之后，你会发现在项目根目录新增了一个

名为coverage的文件夹。在浏览器中，打开这个文件：*/coverage/lcov-report/index.html*。它会以交互式的方式向用户展示代码覆盖率报告（见图10-1）。

图10-1：代码覆盖率报告

该报告会告知用户测试程序覆盖了多少代码，以及基于每个子文件夹的单独覆盖率。用户可以深入到子文件夹中查看各个文件的代码覆盖率。比如用户选择了*components/ui*文件夹，将会了解到测试程序对用户接口组件的代码覆盖率（见图10-2）。

用户可以通过单击文件名查看单个文件中被覆盖的代码行数。图10-3展示了ColorList组件被覆盖的代码行数。

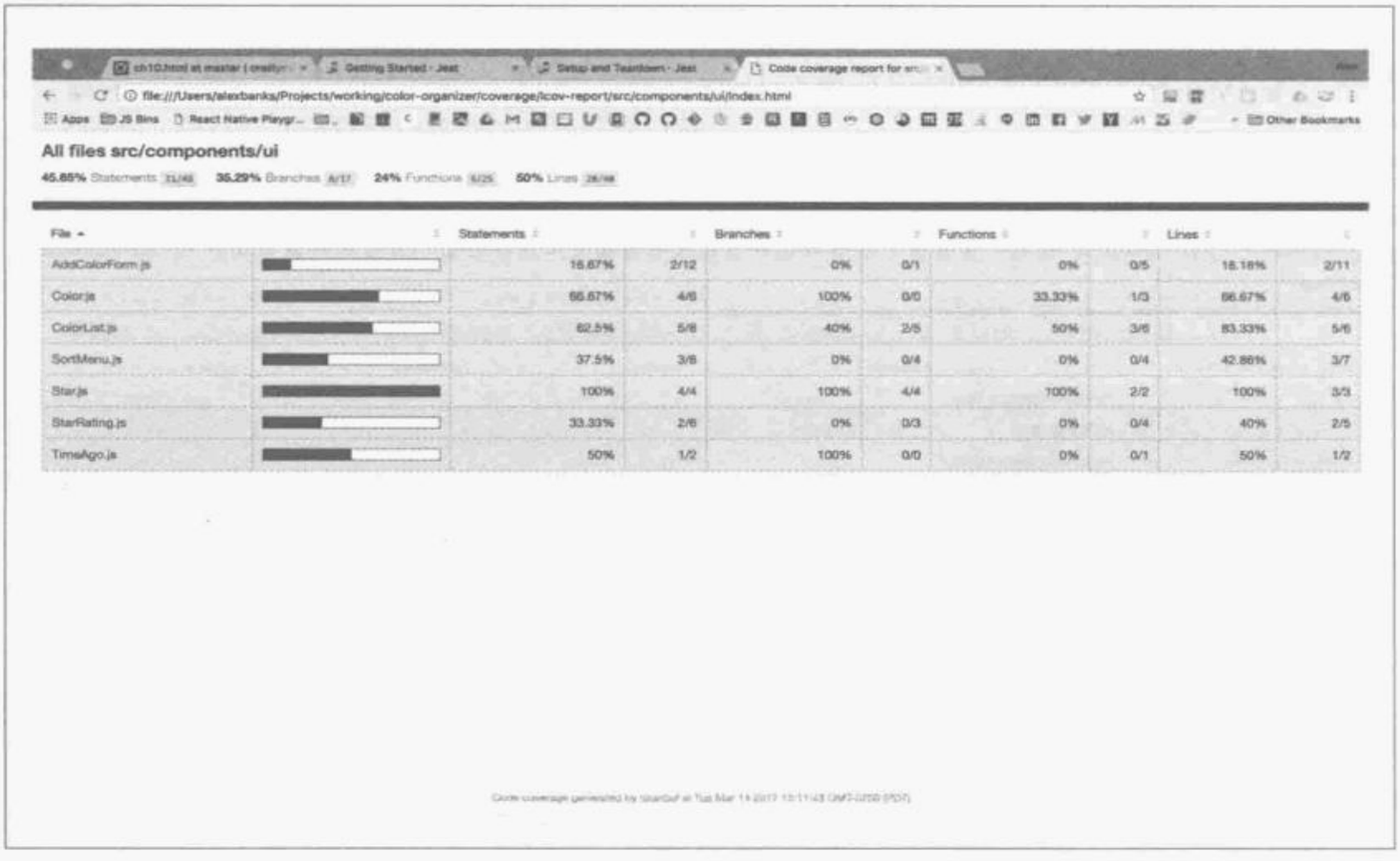


图10-2：UI组件的代码测试覆盖率

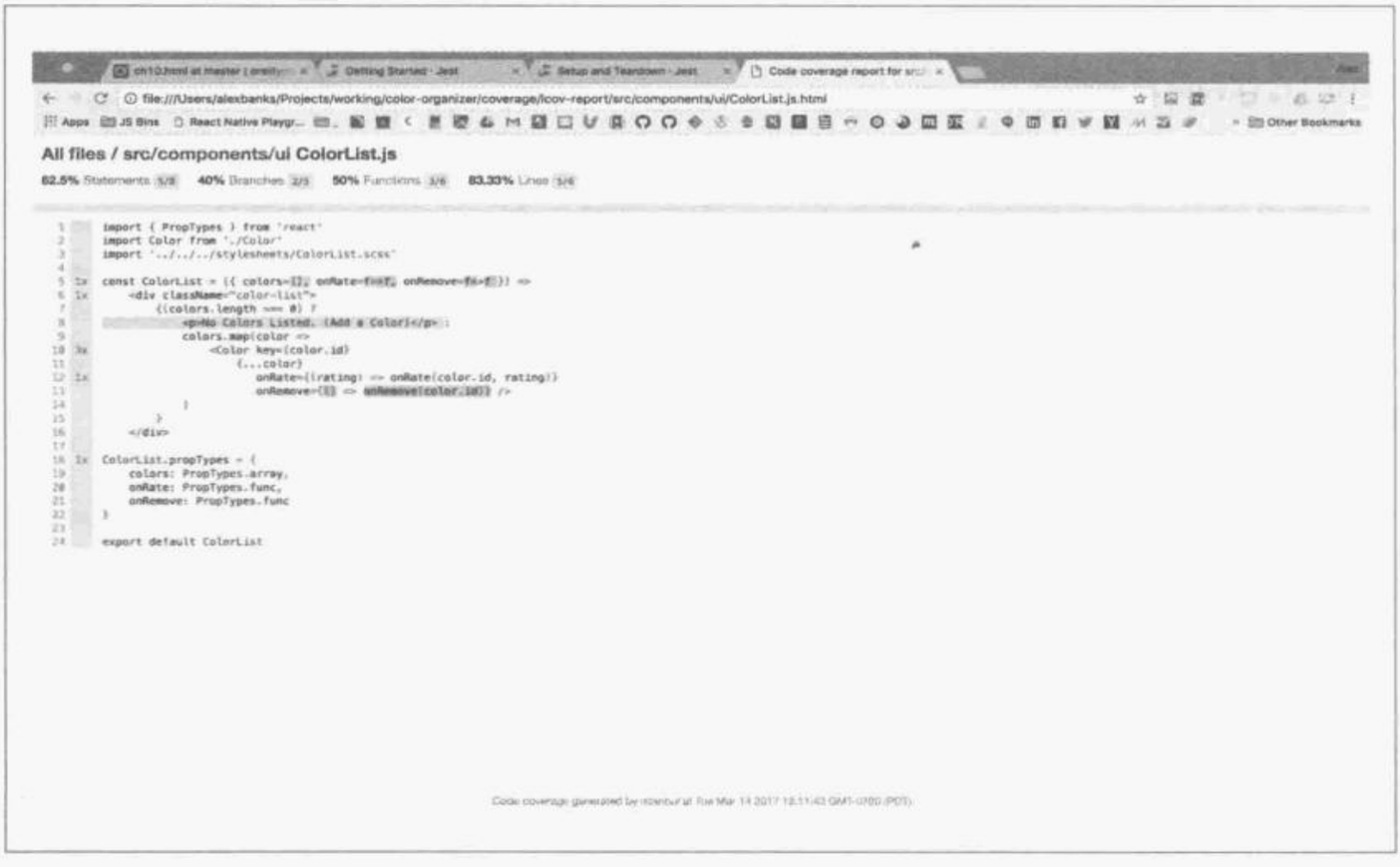


图10-3：ColorList组件的代码覆盖率

ColorList组件已经被很好地测试了。在屏幕左侧的列表中，用户可以发现在一个测试中每行代码被执行了多少次。用黄色和红色高亮显示的代码行表示没有被执行。在这种情况下，看上去我们还没有测试onRemove属性。让我们在*ColorList.test.js*中添加一个测试用例来测试onRemove属性，以便让第13行被覆盖：

```
jest.mock('../../../src/components/ui/Color', () =>
  ({rating, onRate=f=>f, onRemove=f=>f}) =>
    <div className="mockColor">
      <button className="rate" onClick={() => onRate(rating)} />
      <button className="remove" onClick={onRemove} />
    </div>
)

..

describe("Removing a Color", () => {

    let _remove = jest.fn()

    beforeAll(() =>
        mount(<ColorList colors={_testColors} onRemove={_remove} />)
            .find('button.remove')
            .last()
            .simulate('click')
    )

    it("invokes onRemove Handler", () =>
        expect(_remove).toBeCalled()
    )

    it("removes the correct color", () =>
        expect(_remove).toBeCalledWith(
            "58d9caee-6ea6-4d7b-9984-65b145031979"
        )
    )

})
```

onRemove属性被添加到了Color模拟对象中，同时有一个触发该属性的按钮。当我们渲染ColorList时，我们会用测试onRate属性几乎相同的方式测试onRemove属性。将会使用三种测试颜色渲染ColorList组件，单击最后一个“remove”按钮，以便确保正确的ID被传递给了模拟函数_remove。

下次我们生成代码覆盖率报告时，将会发现其中的第13行代码被覆盖了（见图10-4）。

图10-4：通过测试onRemove属性提高了代码覆盖率

看上去第8行代码还没有被覆盖。这是因为我们从来没有使用一个空的颜色数组来渲染ColorList组件。让我们用一个测试用例对第8行代码进行覆盖：

```
describe("Rendering UI", () => {

    it("Defaults properties correctly", () =>
        expect(shallow(<ColorList />).find('p').text())
            .toBe('No Colors Listed. (Add a Color)')
    )

})
```

不提供任何属性的情况下渲染Color组件不仅会覆盖第8行代码，它还会覆盖设置默认属性值的第一行代码（见图10-5）。

测试ColorList组件的代码覆盖率已经非常接近100%了。我们唯一没有测试的部分是onRate和onRemove的默认函数。如果我们不提供这些函数，这些属性将会是必需的。我们可以通过不提供属性渲染ColorList组件来对测试进行改进。我们还希望在第一个评分按钮和最后一个移除按钮上模拟单击事件：

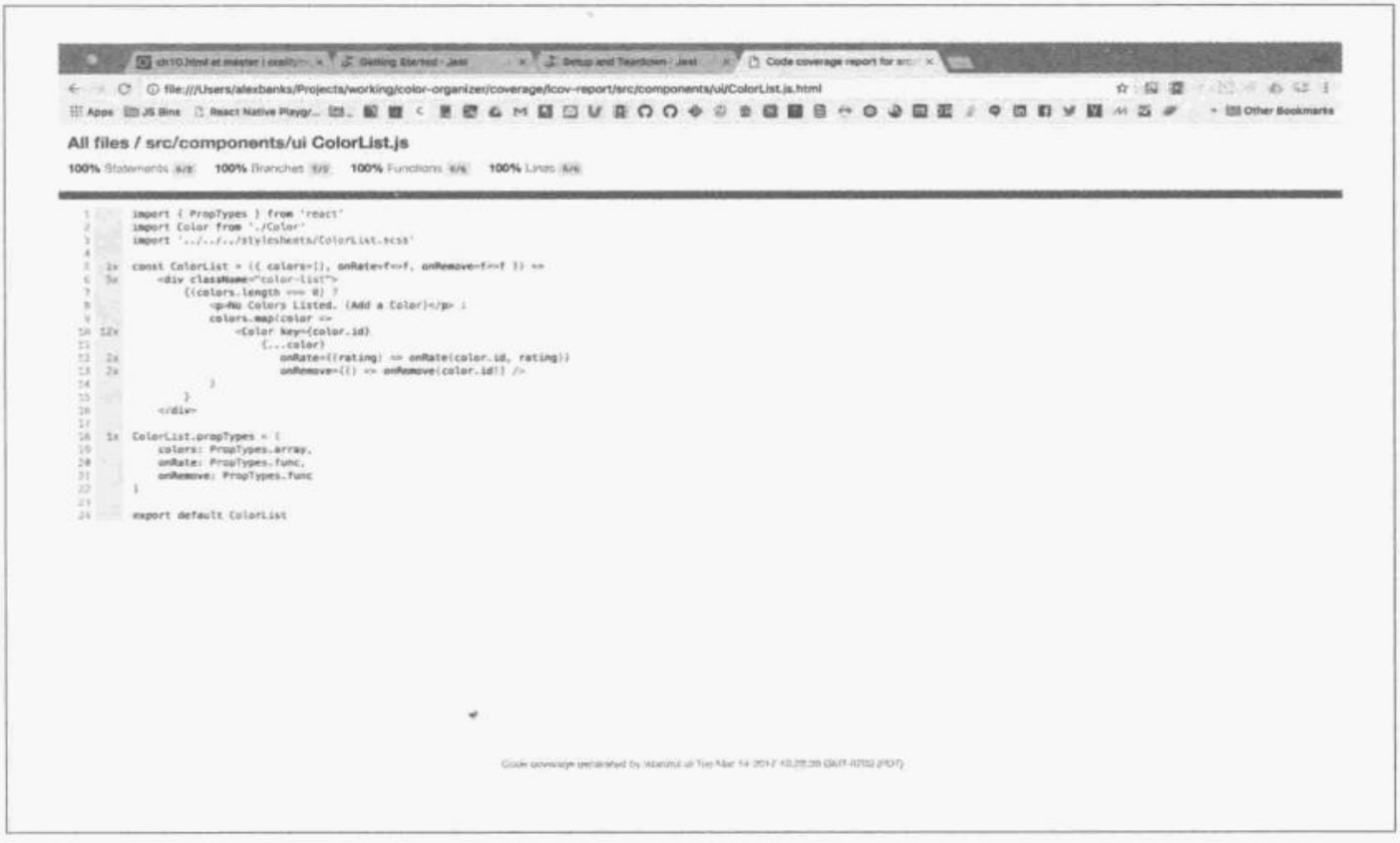


图10-5：通过测试空的颜色值提高了的代码覆盖率

```
describe("Rendering UI", () => {

    it("Defaults properties correctly", () =>
        expect(shallow(<ColorList />).find('p').text())
            .toBe('No Colors Listed. (Add a Color)')
    )

    it("Clicking default rate button does not cause Error", () => {
        mount(<ColorList colors={_testColors} />)
            .find('button.rate')
            .first()
            .simulate('click')
      })

    it("Clicking default remove button does not cause Error", () => {
        mount(<ColorList colors={_testColors} />)
            .find('button.remove')
            .first()
            .simulate('click')
    })
})
```

下次我们在执行jest的代码覆盖率报告命令后，将会发现ColorList组件的代码覆盖率是100%（见图10-6）。

但是我们的项目中的其他组件还有大量的工作要去完成，如图10-7所示。

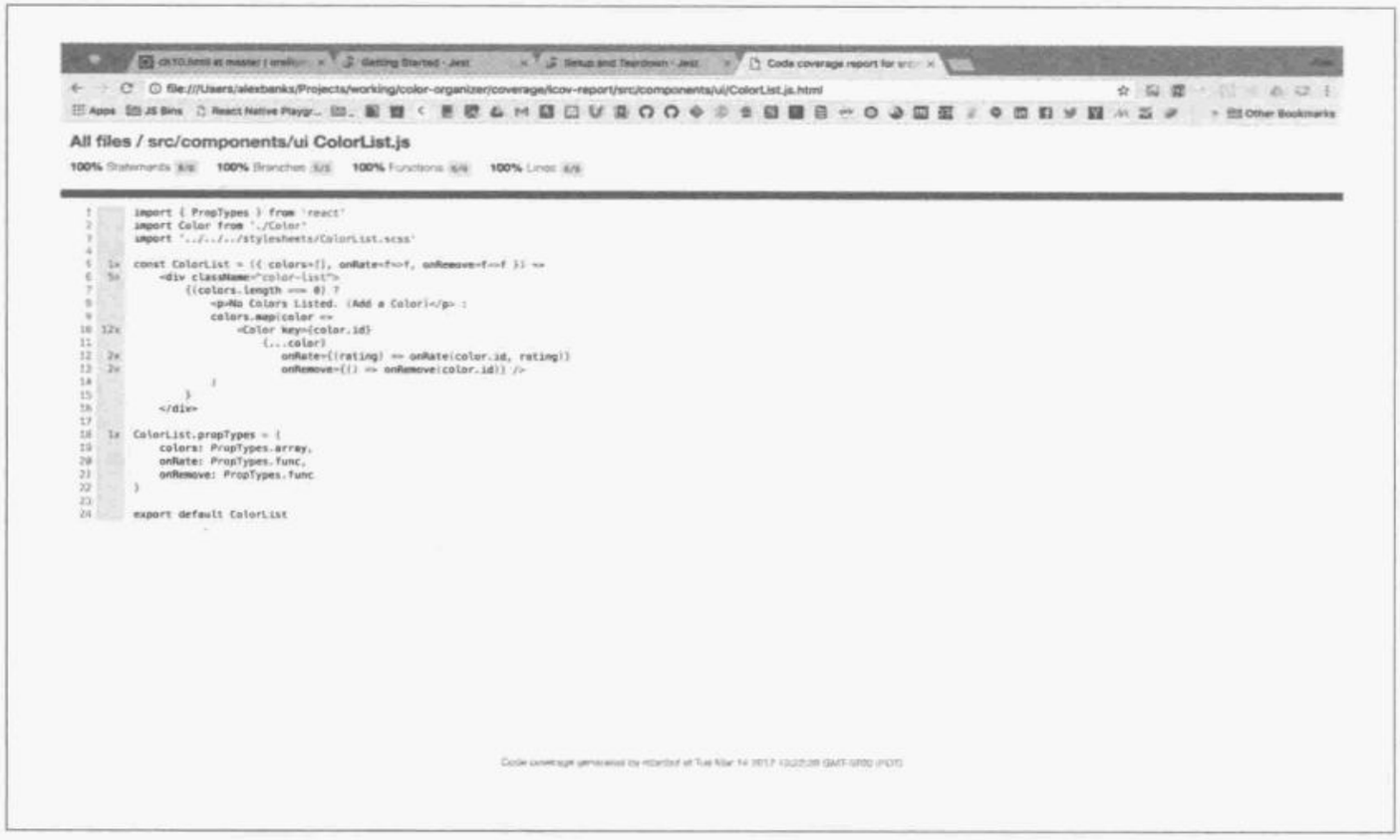


图10-6：ColorList组件的代码覆盖率是100%

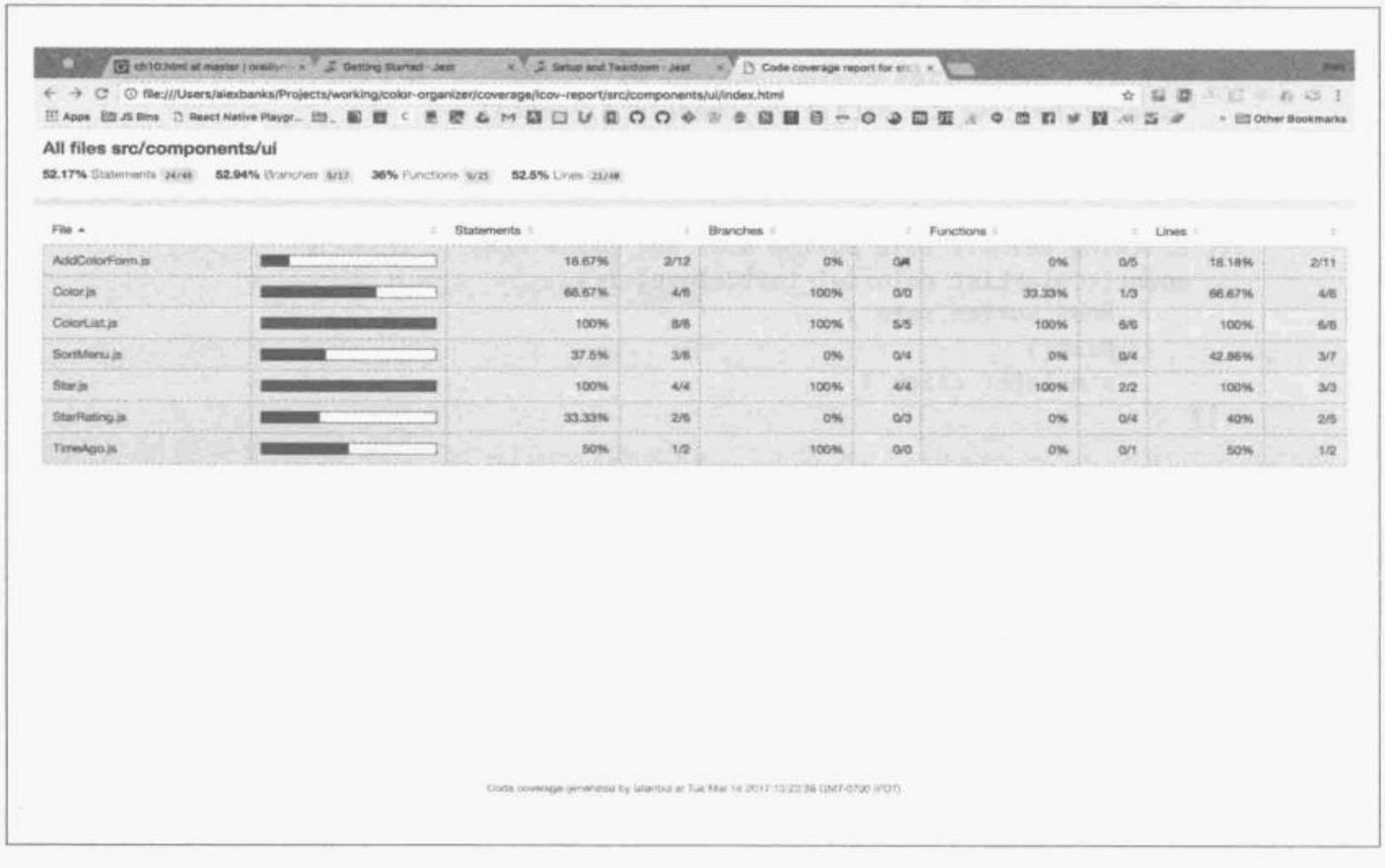


图10-7：为ColorList组件编写新的测试后，UI组件测试的代码覆盖率

用户可以使用这个报告对测试过程进行指导，通过提高测试代码覆盖率来改进测试过程。

用户甚至可以将代码覆盖率选项添加到*package.json*文件中：

```
"jest": {
    "setupFiles": ["./ tests /global.js"],
    "modulePathIgnorePatterns": ["global.js"],
    "moduleNameMapper": {
      "\\.(scss)$": "<rootDir>/node_modules/jest-css-modules"
    },
    "verbose": true,
    "collectCoverage": true,
    "notify": true,
    "collectCoverageFrom": ["src/**"],
    "coverageThreshold": {
      "global": {
        "branches": 80,
        "functions": 80,
        "lines": 80,
        "statements": 80
      }
    }
  }
```

coverageThreshold字段定义了在测试通过之前代码覆盖率应该达到多少。我们为所有分支、函数、代码行和语句声明了代码覆盖率必须达到80%。

collectCoverageFrom字段是用户声明哪些文件应该被覆盖的地方。它会接收一个通配符模式的数组。我们声明了所有在*src*文件目录下，以及其子文件夹下的所有文件都应该被覆盖。

将collectCoverage选项设置为true意味着用户每次在该项目上执行jest命令后都会收集和代码覆盖率相关的数据。notify字段将会根据用户的操作系统向用户显示一个通知信息框。最后，verbose选项会向用户显示每个测试每次执行jest命令的细节报告。“<ColorList /> UI Component”用例的verbose报告将会和下列内容类似：

```
PASS __tests__/components/ui/ColorList.test.js
  <ColorList /> UI Component
    Rendering UI
      ✓ Defaults Properties correctly (2ms)
      ✓ Clicking default rate button do not cause Error (6ms)
      ✓ Clicking default remove button do not cause Error (4ms)
    Rating a Color
      ✓ invokes onRate Handler
      ✓ rates the correct color (1ms)
    Removing a Color
      ✓ invokes onRemove Handler
      ✓ removes the correct color
```

上述内容表明，颜色管理器程序将会需要编写更多测试之后代码覆盖率才能达到100%。GitHub版本库上本章相关的程序代码的代码覆盖率已经达到了100%（见图10-8）。

图10-8：代码覆盖率100%

代码覆盖率统计是衡量代码测试范围的一个很好的工具。它是一种帮助用户了解是否为自己的代码编写了足够多的单元测试的基准。一般来说，每个项目都不可能达到100%的代码覆盖率。达到85%以上是一个很好的目标。[注5]

注5：参见Martin Fowler的文章“Test-Coverage”（*http://bit.ly/2kuXEsb*）。

第11章

React Router

互联网诞生之初，大部分网站都是由用户可以通过请求并打开若干独立文件的一系列页面构成的。当前文件或者资源的位置会显示在浏览器的地址栏上。浏览器的前进和后退按钮能够按照预期工作。对网站中某个页面添加书签后允许用户保存特定文件的引用，以便可以通过客户端请求再次载入该页面。在一个基于页面或者服务端渲染的网站上，浏览器的导航和历史回退功能可以按照预期正常工作。

在一个单页应用中，所有这些特性都会出问题。请务必记住，在一个单页应用中，所有内容都是在同一页面呈现的。JavaScript负责载入信息和修改UI。浏览器的历史记录、书签、前进和后退功能在没有路由转发解决方案的情况下将无法正常工作。路由转发就是为用户的客户端请求定义端点的过程。[注1]这些端点会和浏览器的位置及历史对象一起协同工作。它们可以用于标记请求的内容，以便JavaScript可以加载和渲染相应的用户界面。

与Angular、Ember和Backbone不同之处在于，React并没有附带一个标准的router。鉴于路由转发解决方案的重要性，工程师Michael Jackson和Ryan Florence创造了一个名为React Router的工具。React Router目前已经被社区作为React应用程序广泛采用的路由转发解决方案。[注2]采用它的公司包括Uber、Zendesk、PayPal和Vimeo等。[注3]

在本章中，我们将会介绍React Router，并回顾如何使用HashRouter组件处理客户端的路由转发。

注1：Express.js开发文档，“Basic Routing”（*http://bit.ly/2mJlIt5*）。

注2：该项目在GitHub上已经获得超过20000颗星（*http://bit.ly/2mJt4gk*）。

注3：参见“Sites Using React Router”（*http://bit.ly/2mJbN6X*）。

集成Router

为了演示React Router的功能特性，我们将会构建一个经典的初级网站，其中包括About、Events、Products和Contact Us等段落（见图11-1）。虽然这个网站看上去似乎包含多个页面，但是实际上只有一个：它是一个单页应用（SPA）。

图11-1：公司网站主页

该网站的导航地图包括一个主页，一个显示每个段落的页面，以及用于处理“404 Not Found”的错误提示页面（见图11-2）。

图11-2：页面标题和路由

router将会允许用户为网站的每个段落配置路径。每个路径都是可以在浏览器地址栏输入的端点。当某个路径被请求后，我们可以渲染显示相应的内容。

HashRouter

react-router-dom提供了一对选项用于管理单页应用中的导航历史记录。HashRouter是专门为客户端而设计的。一般来说，地址栏上的哈希字符是用于定义锚点链接的。当在地址栏中使用#时，浏览器并不会向服务端发送请求。在使用HashRouter时，所有路径前面的#将一直是必不可少的。

HashRouter对于新项目或者不需要后端服务的小型客户端网站是非常好的工具。BrowserRouter是大部分准备上线的网站应用首选的解决方案。在第12章讨论通用应用程序时，我们将会详细介绍BrowserRouter。

接下来将安装react-router-dom，这个软件包是我们将router集成到基于浏览器的应用程序所必需的：

```
npm install react-router-dom --save
```

我们还需为网站地图中每个段落或者页面添加几个占位符组件。可以从单个文件中导出这些组件：

```
export const Home = () =>
    <section className="home">
        <h1>[Home Page]</h1>
    </section>

export const About = () =>
    <section className="events">
        <h1>[About the Company]</h1>
    </section>

export const Events = () =>
    <section className="events">
        <h1>[Events Calendar]</h1>
    </section>

export const Products = () =>
    <section className="products">
        <h1>[Products Catalog]</h1>
    </section>

export const Contact = () =>
    <section className="contact">
        <h1>[Contact Us]</h1>
    </section>
```

当应用程序启动后，不会渲染单个App组件，而是将渲染HashRouter组件：

```
import React from 'react'
import { render } from 'react-dom'
```

```
import {
    HashRouter,
    Route
} from 'react-router-dom'

import {
    Home,
    About,
    Events,
    Products,
    Contact
} from './pages'

window.React = React

render(
  <HashRouter>
    <div className="main">
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Route path="/events" component={Events} />
        <Route path="/products" component={Products} />
        <Route path="/contact" component={Contact} />
    </div>
  </HashRouter>,
  document.getElementById('react-container')
)
```

HashRouter组件是作为应用程序的根组件渲染的。每个路径可以在HashRouter中使用Route组件进行定义。

这些路径会告知router当浏览器窗体地址栏发生变化后渲染哪个组件。每个Route组件都包含path和component属性。当浏览器地址和path匹配时，相关的component将会被显示。当路径地址是 / 时，router将会渲染Home组件。当路径地址是/products时，router将会渲染Products组件。

第一个路径（即显示Home组件的那个）包含一个exact属性。这意味着只有当路径和根目录/精确匹配时，才会显示Home组件。

目前为止，我们能够运行该应用程序，并且可以在地址栏手工输入地址来查看页面内容的变化。比如，在地址栏上输入*http://localhost:3000/#/about*，页面将会渲染显示About组件。

我们肯定不希望用户只能在地址栏手动输入地址才能实现对网站页面的导航。react-router-dom提供的Link组件可以用于创建浏览器链接。

让我们对主页进行修改，以便其中包含一个导航菜单，其中包含每条路径的链接：

```
import { Link } from 'react-router-dom'

export const Home = () =>
    <div className="home">
        <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>
```

现在用户可以在网站主页上通过单击链接访问任意内部页面了。浏览器的“后退”按钮将会把用户带回主页面。

Router属性

React Router会将属性传递给它渲染的组件。比如，我们可以通过属性获得当前的位置。接下来将使用当前路径来帮助用户创建一个“404 Not Found”组件：

```
export const Whoops404 = ({ location }) =>
    <div className=" whoops-404" >
        <h1>Resource not found at '{location.pathname}'</h1>
    </div>
```

当用户输入的路径不存在时，router将会向用户渲染显示Whoops404组件。渲染完毕后，router将会把一个location对象作为属性传递给该组件。我们可以获取和使用此对象获取请求路由的当前路径名。将使用这个路径名来通知用户无法找到他们请求的资源。

现在我们将使用一个Route标签将Whoops404组件添加到应用程序中：

```
import {
    HashRouter,
    Route,
    Switch
} from 'react-router-dom'

...

render(
  <HashRouter>
    <div className="main">
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Route path="/events" component={Events} />
        <Route path="/products" component={Products} />
```

```
          <Route path="/contact" component={Contact} />
          <Route component={Whoops404} />
        </Switch>
      </div>
    </HashRouter>,
    document.getElementById('react-container')
  )
```

因为我们只希望在没有其他Route可匹配时才显示Whoops404组件，所以需要使用Switch组件。Switch组件只会显示首个匹配的路由。这可以确保只渲染其中的一个路由。如果没有任何路径可以和一个Route匹配，那么最后一个路由，即包含路径属性的路由将会被渲染显示。如果用户输入的地址是http://localhost:3000/#/profits，那么获得结果如图11-3所示。

图11-3：404 错误页面

本节介绍了React Router的基本使用方法。所有Route组件都需要包装成一个router，在这种情况下，HashRouter将会根据窗体地址栏上的地址渲染显示相关组件。Link组件可以方便导航。这些基础知识可以让读者走得更远，不过它们只是router功能特性的冰山一角。

嵌套路由

Route组件可以用于匹配特定URL地址后才显示相关内容。这个特性允许用户将Web应用组织成一定的层级结构，从而有利于内容复用。

在本节中，我们还将介绍如何将内容组织成包含子菜单的子段落。

使用页面模版

有时，用户在应用程序中导航时，我们希望某些用户界面能够保持不变。以前的解决方案包括页面模版和母版页来帮助开发人员复用UI元素。React组件天生就可以使用子属性合成模版。

让我们看看这个简单的入门网站。一旦打开页面，每个段落都应该显示相同的主菜单。屏幕右侧的内容应该随着用户浏览不同页面段落而变化，但是屏幕左侧的内容应该保持不变（见图11-4）。

让我们创建一个可复用的PageTemplate组件，以便可以将之作为这些内部页面的模版。该组件将会一直显示主菜单，但是将会根据用户浏览网站不同页面段落而渲染嵌套的内容。

首先，我们需要构造MainMenu组件：

```
import HomeIcon from 'react-icons/lib/fa/home'
import { NavLink } from 'react-router-dom'
import './stylesheets/menus.scss'

const selectedStyle = {
    backgroundColor: "white",
    color: "slategray"
}

export const MainMenu = () =>
    <nav className="main-menu">
        <NavLink to="/">
            <HomeIcon/>
        </NavLink>
        <NavLink to="/about" activeStyle={selectedStyle}>
            [About]
        </NavLink>
        <NavLink to="/events" activeStyle={selectedStyle}>
            [Events]
        </NavLink>
        <NavLink to="/products" activeStyle={selectedStyle}>
            [Products]
        </NavLink>
```

```
            <NavLink to="/contact" activeStyle={selectedStyle}>
                [Contact Us]
            </NavLink>
        </nav>
```

图11-4：内部页面：Events

MainMenu组件会用到NavLink组件。NavLink组件可以用来创建链接，并且在链接被激活时可以显示相应的样式。activeStyle属性可以用来设置链接激活或者被选中后的CSS样式。

MainMenu组件将会用在PageTemplate组件中：

```
import { MainMenu } from './ui/menus'

...

const PageTemplate = ({children}) =>
    <div className="page">
        <MainMenu />
        {children}
    </div>
```

PageTemplate组件的子元素是将会是每个段落被渲染的位置。这里，我们将会在MainMenu之后添加子元素。现在我们可以使用PageTemplate组件对相关段落进行合成：

```
export const Events = () =>
    <PageTemplate>
      <section className="events">
        <h1>[Event Calendar]</h1>
      </section>
    </PageTemplate>

export const Products = () =>
    <PageTemplate>
        <section className="products">
            <h1>[Product Catalog]</h1>
        </section>
    </PageTemplate>

export const Contact = () =>
    <PageTemplate>
        <section className="contact">
            <h1>[Contact Us]</h1>
        </section>
    </PageTemplate>

export const About = ({ match }) =>
    <PageTemplate>
        <section className="about">
            <h1>About</h1>
        </section>
    </PageTemplate>
```

如果用户运行该应用程序，将会发现每个段落显示的主菜单是一样的。当用户浏览网站的内部页面时，屏幕右边的内容会随之发生变化。

子段落和子菜单

接下来，将会使用Route组件在About段落下嵌套4个组件（见图11-5）。

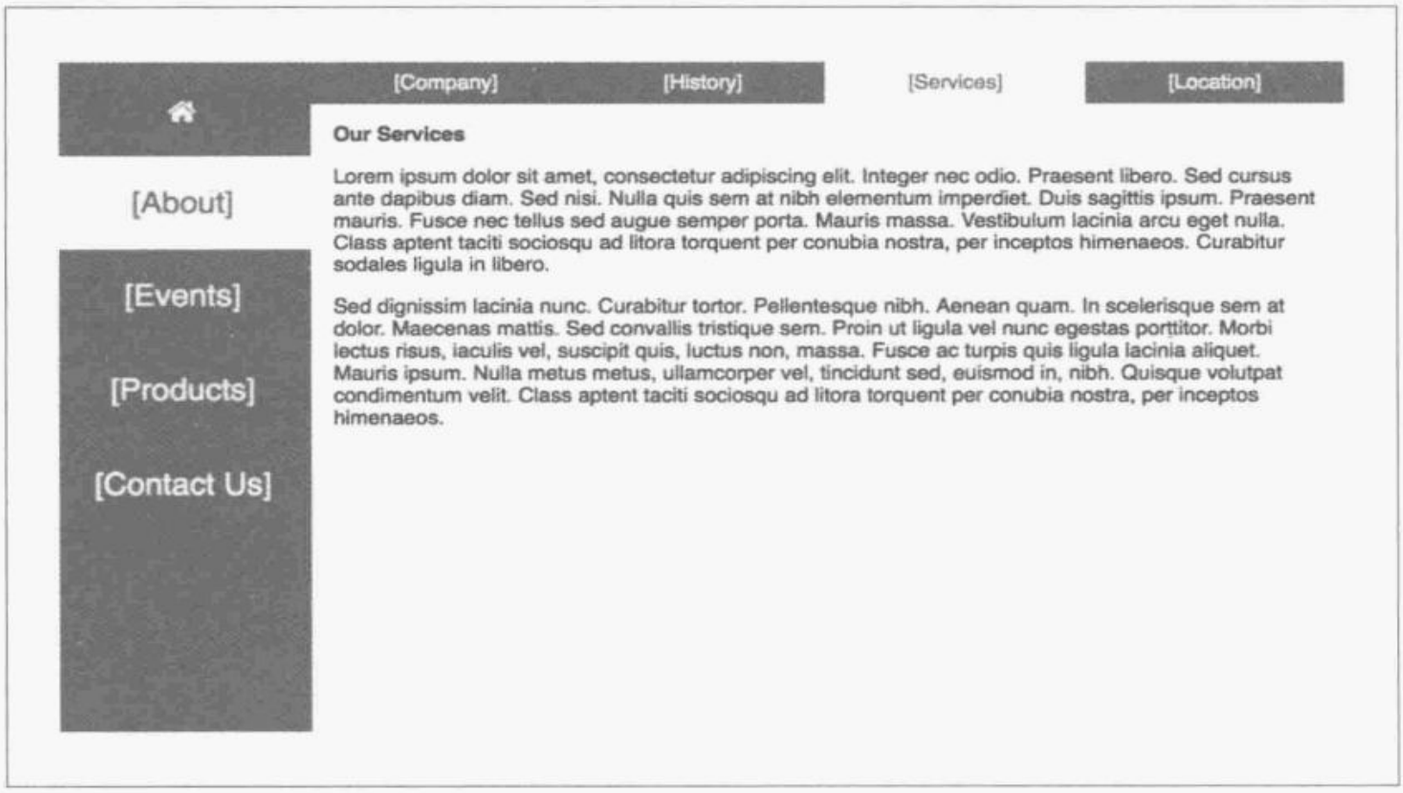


图11-5：About段落下的子页面

我们需要为Company、History、Services和Location等栏目添加页面。当用户选择About段落时，系统默认会给用户显示该段落下的Company页面。大纲如下所示：

- Home Page
  - — **About the Company**
    - — **Company (default)**
    - — **History**
    - — **Services**
    - — **Location**
  - — Events
  - — Products
  - — Contact Us
- 404 Error Page

我们需要创建的新路由将会以下列层级结构表示：

- 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/#/foo-bar

让我们为About段落创建一个子菜单。将会使用NavLink组件，并会采用和MainMenu中一样的activeStyle：

```
export const AboutMenu = ({match}) =>
    <div className="about-menu">
        <li>
            <NavLink to="/about"
                 style={match.isExact && selectedStyle}>
                [Company]
            </NavLink>
        </li>
        <li>
            <NavLink to="/about/history"
                 activeStyle={selectedStyle}>
                [History]
            </NavLink>
        </li>
        <li>
            <NavLink to="/about/services"
                 activeStyle={selectedStyle}>
                [Services]
            </NavLink>
        </li>
        <li>
            <NavLink to="/about/location"
                 activeStyle={selectedStyle}>
                [Location]
            </NavLink>
        </li>
    </div>
```

AboutMenu组件使用NavLink组件引导用户浏览About段落下的内容。这个组件将会使用一个Route渲染表示，这意味着它会接收路由属性。我们将会使用从Route发送到该组件的match属性。

除了第一个之外，所有NavLink组件都会使用activeStyle属性。当路径和链接的路由匹配时，activeStyle将会设置链接的样式属性。比如，当用户导航到http://localhost:3000/about/services时，Services的NavLink将会渲染显示一个白色背景。

第一个NavLink组件并没有使用activeStyle。相反，只有当路由精确匹配/about时，它的样式属性才会被设置为selectedStyle。当路径是/about时，match.isExact的属性值是true，当路径是/about/services时，它的属性值是false。从技术上来说，/about路径和上述两个地址都匹配，不过只有路径是/about时才是精确匹配。

占位符组件

我们还需要注意新增的段落使用了占位符组件：Company、Services、History和Location。它只是简单地显示了一些随机文字：

```
export const Services = () =>
    <section className="services">
        <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>
        <p>
            Sed dignissim lacinia nunc. Curabitur tortor. Pellentesque
            nibh. Aenean quam. In scelerisque sem at dolor. Maecenas
            mattis. Sed convallis tristique sem. Proin ut ligula vel
            nunc egestas porttitor. Morbi lectus risus, iaculis vel,
            suscipit quis, luctus non, massa. Fusce ac turpis quis
            ligula lacinia aliquet. Mauris ipsum.
            Nulla metus metus, ullamcorper vel, tincidunt sed, euismod
            in, nibh. Quisque volutpat condimentum velit. Class aptent
            taciti sociosqu ad litora torquent per conubia nostra, per
            inceptos himenaeos.
        </p>
    </section>
```

现在我们准备将路由添加到About组件：

```
export const About = ({ match }) =>
    <PageTemplate>
        <section className="about">
            <Route component={AboutMenu} />
            <Route exact path="/about" component={Company}/>
```

```
            <Route path="/about/history" component={History}/>
            <Route path="/about/services" component={Services}/>
            <Route path="/about/location" component={Location}/>
          </section>
        </PageTemplate>
```

About组件将会被各个段落复用。路径将会告知应用程序渲染哪个子段落。比如，当地址为http://local host:300/about/history时，About组件内将会渲染显示History组件。

这次我们并没有使用Switch组件。任意和地址匹配的路由都会渲染显示与之相关的组件。第一个Route将会一直显示AboutMenu。此外，存在任何其他匹配的Route时，也会显示它们的组件。

重定向

有时开发人员也许会希望将用户从一个路由重定向到另外一个。比如，我们可以确保任何尝试通过http://localhost:3000/services访问页面的用户能够重定向到正确的路由地址：http://localhost:3000/about/services。

让我们对程序稍作修改，添加重定向功能，以便确保用户可以访问正确的内容：

```
import {
  HashRouter,
  Route,
  Switch,
  Redirect
} from 'react-router-dom'

...

render(
  <HashRouter>
    <div className="main">
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Redirect from="/history" to="/about/history" />
        <Redirect from="/services" to="/about/services" />
        <Redirect from="/location" to="/about/location" />
        <Route path="/events" component={Events} />
        <Route path="/products" component={Products} />
        <Route path="/contact" component={Contact} />
        <Route component={Whoops404} />
      </Switch>
    </div>
  </HashRouter>,
  document.getElementById('react-container')
)
```

Redirect组件允许开发人员将用户重定向到某个特定路由。

当正式上线的应用程序中的某些路由发生变化后，用户仍然可以通过旧的路由访问旧的内容。这通常是由书签产生的。Redirect组件为我们提供了一种加载适当内容的方式，即使用户是通过一个旧的书签访问网站的。

React Router允许用户在应用程序内部任意位置合成Route组件，因为HashRouter是根组件。现在可以通过层级结构的方式对我们的内容进行组织，方便用户浏览。

Router参数

React Router另外一个非常实用的特性是配置路由参数的能力。路由参数是可以从URL中获取参数值的变量。它们在数据驱动的Web应用程序中非常有用，特别是在内容过滤和管理首选项显示方面。

添加颜色详情页面

让我们通过React Router添加每次可以选择和显示一种颜色，来对颜色管理器程序进行改进。当用户单击和选择了一种颜色后，应用程序将会显示该颜色，以及它的标题和十六进制颜色值（见图11-6）。

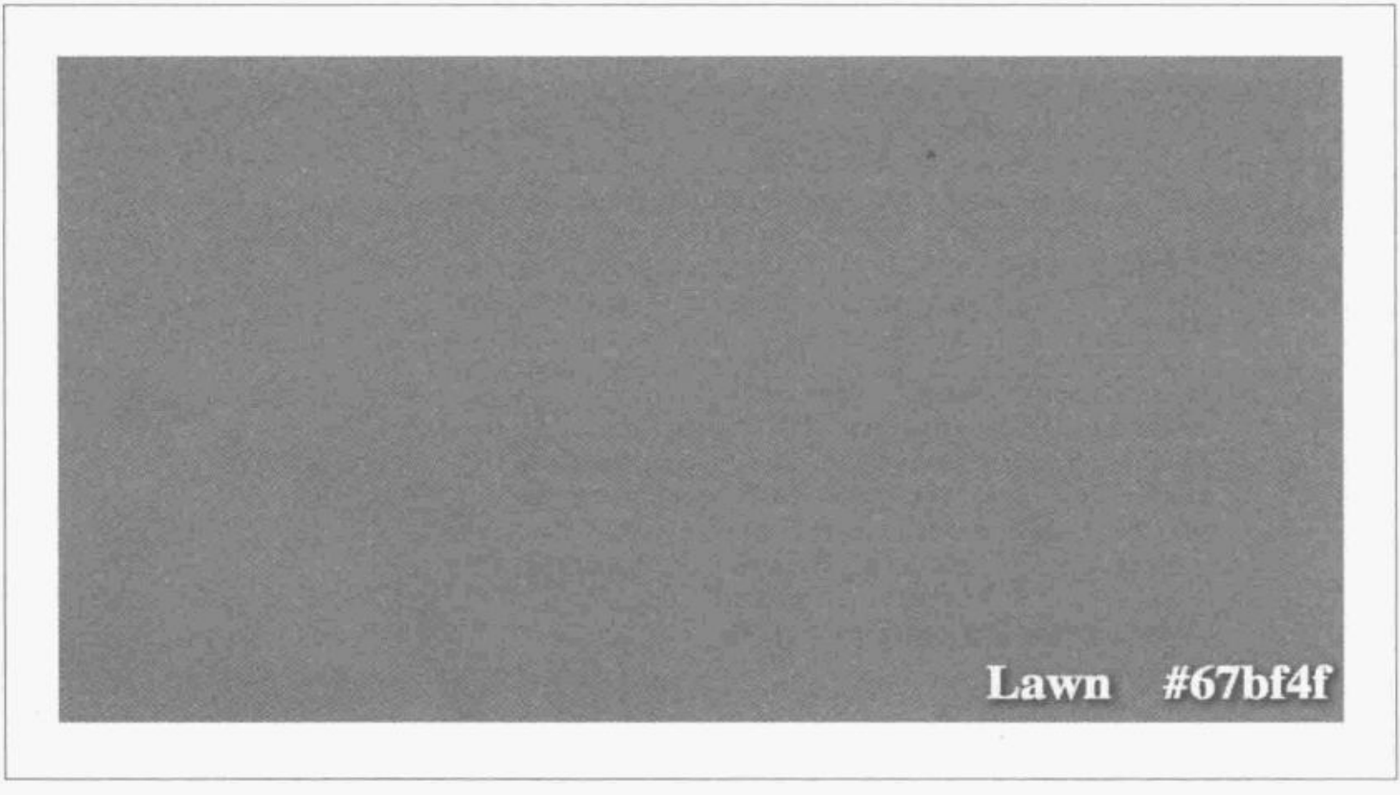


图11-6：颜色详情页面

每种颜色都包含一个唯一ID。这个ID可以用来查找保存在State中的特定颜色。比如，我们可以创建一个findById函数，它会根据id字段在数组中查找相关的对象：

```
import { compose } from 'redux'

export const getFirstArrayItem = array => array[0]

export const filterArrayById = (array, id) =>
    array.filter(item => item.id === id)

export const findById = compose(
    getFirstArrayItem,
    filterArrayById
)
```

findById函数遵循了第2章讨论的函数式编程范式。我们可以看到findById方法首先会根据ID对数组进行过滤，然后从已过滤数组中返回找到的第一个元素。我们可以使用findById函数根据它们的唯一ID在State中对颜色对象进行定位。

通过router，我们可以通过URL获取颜色的ID。比如，我们将要用于显示“草坪绿”的URL，其中包含了该颜色的ID：

```
http://localhost:3000/#/58d9caee-6ea6-4d7b-9984-65b145031979
```

Router参数允许用户获取它们的参数值。它们可以在路由中使用冒号进行定义。比如，我们可以获取唯一id，然后使用Route将它另存为一个名为id的参数中：

```
<Route exact path="/:id" component={UniqueIDHeader} />
```

UniqueIDHeader组件可以从match.params对象中获取id：

```
const UniqueIDHeader = ({ match }) => <h1>{match.params.id}</h1>
```

我们可以随时为从URL中收集到的数据创建参数。

多参数

多参数就是可以在同一参数对象上创建和访问多个参数。下面的示例路由中我们创建了三个参数：

```
<Route path="/members/:gender/:state/:city"
       component="Member" />
```

这三个参数可以通过URL初始化：

```
http://localhost:3000/members/female/california/truckee
```

这三个参数值都会通过match.params传递给Member组件：

```
const Member = ({ match }) =>
    <div className="member">
        <ul>
            <li>gender: {match.params.gender}</li>
            <li>state: {match.params.state}</li>
            <li>city: {match.params.city}</li>
        </ul>
    </div>
```

接下来将会创建一个ColorDetails组件，当用户选择单个颜色后将会渲染显示该组件：

```
const ColorDetails = ({ title, color }) =>
    <div className="color-details"
        style={{backgroundColor: color}}>
        <h1>{title}</h1>
        <h1>{color}</h1>
    </div>
```

ColorDetails组件是一个表现层组件，它希望获得颜色细节的若干属性。因为我们采用了Redux，将会需要添加一个新的容器，以便通过一个路由参数在State中找到被选定的颜色：

```
export const Color = connect(
    (state, props) => findById(state.colors, props.match.params.id)
)(ColorDetails)
```

Color组件是通过connet高阶组件创建的。第一个参数是一个函数，它是用来根据State中的单个颜色对象来设置ColorDetails的属性的。采用了本节前面定义的findById函数后，将会根据从URL中获取的id值来对State中的唯一的颜色对象进行定位。

connect高阶组件将会构造从被定位颜色对象到ColorDetails组件属性之间的数据映射。connect高阶组件还会将Color容器中的任意属性映射到ColorDetails组件中。这意味着router的所有属性也会被传递到ColorDetails组件中。

让我们使用router的history属性给ColorDetails组件添加一个导航数据：

```
const ColorDetails = ({ title, color, history }) =>
    <div className="color-details"
        style={{backgroundColor: color}}
        onClick={() => history.goBack()}>
        <h1>{title}</h1>
        <h1>{color}</h1>
```

```
    </div>
```

当用户单击div.color-details元素时，history.goBack()将会被触发。用户将会被导航到上一个位置。

我们已经有了Color容器，现在需要把它添加到应用程序中。首先，在初始化渲染时，我们需要使用一个HashRouter对App组件进行包装：

```
import { HashRouter } from 'react-router-dom'

...

render(
    <Provider store={store}>
        <HashRouter>
          <App />
        </HashRouter>
    </Provider>,
    document.getElementById('react-container')
)
```

现在我们已经为在应用程序内部任意位置配置路由做好了准备。让我们添加一些路由到App组件中：

```
import { Route, Switch } from 'react-router-dom'
import Menu from './ui/Menu'
import { Colors, Color, NewColor } from './containers'
import '../stylesheets/APP.scss'

const App = () =>
    <Switch>
        <Route exact path="/:id" component={Color} />
        <Route path="/"
            component={() => (
                <div className="app">
                    <Menu />
                    <NewColor />
                    <Colors />
                </div>
            )} />
    </Switch>

export default App
```

Switch组件将用于渲染下列两种路由之一：单个颜色或者主应用程序组件。当id是通过URL传递时，第一个Route将会渲染Color组件。比如该路由匹配的路径是：

```
http://localhost:3000/#/58d9caee-6ea6-4d7b-9984-65b145031979
```

任何其他路径将会匹配 /，并显示主应用程序组件。第2个Route在一个新的匿名无状态函数式组件中对若干组件进行了分组。

因此，用户会根据URL地址的不同，既可以看到单个颜色，也可以看到一组颜色列表。

目前，我们可以通过直接在浏览器地址栏上添加一个id参数对应用程序进行测试。不过，用户也需要一种导航到颜色详情页面的方式。

这一次，NavLink组件将不会用于处理从颜色列表到单个颜色对象细节的导航。相反，我们会直接使用router的history对象进行导航。

让我们给在./ui发现的Color组件添加导航功能。该组件是通过ColorList渲染的。它并不会从Route接收路由属性参数。用户可以沿着组建树向下显式传递这些属性给Color组件，不过使用withRouter函数更方便一些。它是和react-router-dom一起运行的。withRouter可以用来给某个Route下任意被渲染的组件添加路由属性的。

通过withRouter，我们可以通过属性的形式获取router的history对象。我们可以使用它在Color组件内部进行导航浏览：

```
import { withRouter } from 'react-router'

...

class Color extends Component {

    render() {
        const {
            id,
            title,
            color,
            rating,
            timestamp,
            onRemove,
            onRate,
            history } = this.props

        return (
            <section className="color" style={this.style}>
                <h1 ref="title"
                    onClick={() => history.push(`/${id}`)}>
                    {title}
                </h1>
                <button onClick={onRemove}>
                    <FaTrash />
                </button>
                <div className="color"
```

```
                    onClick={() => history.push(`/${id}`)}
                    style={{ backgroundColor: color }}>
                </div>
                <TimeAgo timestamp={timestamp} />
                <div>
                    <StarRating starsSelected={rating}
                        onRate={onRate}/>
                </div>
            </section>
        )
    }

}

export default withRouter(Color)
```

withRouter是一个HOC。当导出Color组件时，我们是将它和Router一起发送的，后者会将它包装成能够传递router属性的组件：match、history和location。

导航功能是直接使用history对象完成的。当某个用户单击颜色标题或者颜色自身时，一个新的路由将会被添加到history对象中。这个新的路由是一个包含颜色id的字符串。将路由添加到history中后将会使得用户可以导航到该颜色的详情页面。

真实单一数据源？

目前，颜色管理器程序的State基本上是由Redux的Store处理的，还有一些State是由router处理的。具体来说，如果路由中包含一个颜色ID，那么应用程序表现层的State是和不包含上述ID的情况有很大不同的。

router处理某些State看上去和Redux要求将State存放于单个对象的原则是矛盾的：真实单一数据源。不过，读者可以将router看作真实单一数据源的浏览器接口。允许router处理任何与站点地图有关的State是绝对正确的，其中还包括数据过滤功能。让其余的State保存在Redux的Store即可。

将State中的排序属性迁移到Router

用户无需限制Router的使用范围。它们的用途远超过在State中过滤和查找特定颜色。它们还可以用于获取渲染UI必需的信息。

Redux的Store是通过sort属性保存如何对颜色排序的信息的。将这个变量从Redux的Store转换成一个路由参数是否合理呢？该变量本身并不是数据，它只是提供了数据应该如何表示的信息。sort变量是一个字符串，这也使其成为路由参数的理想候选。最

后，我们希望用户可以在一个链接中将sort属性发送给其他用户。如果他们偏向于根据颜色评分对颜色排序，那么可以将这些信息放在一个链接中发送给其他用户，或者将这些内容作为浏览器中的书签进行保存。

让我们将State中颜色的sort属性转换成一个路由参数。这些路由可以用来对我们的颜色列表进行排序：

/#/ default

根据日期排序。

/#/sort/title

根据标题排序。

/#/sort/rating

根据颜色评分排序。

首先，我们需要将sort的Redcuer从*./Store/index.js*文件中移除。我们不再需要使用它了。最后：

```
combineReducers({colors, sort})
```

变成了：

```
combineReducers({colors})
```

移除Reducer意味着State变量将会不再由Redux管理。

接下来，我们还可以从*./src/components/containers.js*文件中移除Menu组件的容器。该容器是用来将Redux的Store中的State和Menu表现层组件相连的。sort不再存放于State中了，因为我们也不需要用到这个容器了。

此外，在*containers.js*文件中，我们需要修改Colors容器。它不再从State中接收sort的值了。相反，它会接收一个路由参数作为排序指令，该参数是通过Color组件内部的match属性进行传递的：

```
export const Colors = connect(
    ({colors}, {match}) =>
        ({
            colors: sortColors(colors, match.params.sort)
        }),
    dispatch =>
        ({
            onRemove(id) {
                dispatch(removeColor(id))
```

```
        },
        onRate(id, rating) {
            dispatch(rateColor(id, rating))
        }
    })
)(ColorList)
```

现在颜色会在它们被作为一个属性传递给ColorList之前，通过一个路由参数进行排序。

接下来，我们需要使用一个包含新路由的链接替换Menu组件。与本章前面创建的About菜单类似，菜单的视觉特效将通过NavLink组件的activeStyle属性进行控制：

```
import { NavLink } from 'react-router'

const selectedStyle = { color: 'red' }

const Menu = ({ match }) =>
    <nav className="menu">

        <NavLink to="/" style={match.isExact && selectedStyle}>
            date
        </NavLink>

        <NavLink to="/sort/title" activeStyle={selectedStyle}>
            title
        </NavLink>

        <NavLink to="/sort/rating" activeStyle={selectedStyle}>
            rating
        </NavLink>

    </nav>

export default Menu
```

现在用户可以通过URL对颜色进行排序了。当没有提供可用的排序参数时，颜色将会根据日期进行排序。这个菜单将会通过修改链接颜色的方式告知用户数据被重新排列了。

我们需要修改App组件，让它可以通过路由对颜色进行排序：

```
const App = () =>
  <Switch>
    <Route exact path="/:id" component={Color} />
    <Route path="/"component={() => (
      <div className="app">
        <Route component={Menu} />
        <NewColor />
        <Switch>
```

```
            <Route exact path="/" component={Colors} />
            <Route path="/sort/:sort" component={Colors} />
          </Switch>
        </div>
      )} />
    </Switch>
```

首先，菜单需要用到match属性，以便我们可以通过Route渲染菜单。菜单将始终显示NewColor表单和颜色列表，因为Route并不包含path属性。

在NewColor组件之后，我们希望既可以显示根据默认条件排序的颜色列表，也可以显示根据某个参数进行排序的颜色列表。这些路由被包装到一个Switch组件中，以便确保只渲染一个Colors容器。

当用户导航到主页路由http://localhost:3000时，系统会渲染显示App组件。默认情况下，Colors容器会在App内部渲染显示。sort的参数值是undefined，因此这些颜色会使用默认条件进行排序：

如果用户导航到http://localhost:3000/sort/rating，Colors容器也会被渲染，不过这次sort包含相关的参数值，因此这些颜色将根据上述参数值进行排序。

路由参数是获取影响用户界面表现层数据的理想工具。不过只有当开发人员希望用户通过URL获取这些细节时才应该被采用。比如，在颜色管理器程序中，用户可以通过某个特定字段给其他用户发送对特定颜色或者所有颜色进行排序的链接。用户还可以将这些链接另存为书签，继而返回特定数据。如果希望用户可以在URL中保存表现层的相关信息，那么路由参数是一个不错的解决方案。

在本章中，我们回顾了React Router的基本用法。本章中所有示例都是包含在HashRouter中的。下一章中，我们将会继续通过BrowserRouter讨论路由在服务器端和客户端的使用，并且我们将会使用StaticRouter在服务器端渲染当前路由的上下文。

第12章

React服务器端应用

目前为止，我们已经使用React构建了一些完全可以在浏览器中运行的小型应用程序。它们在浏览器中收集数据，并使用浏览器的storage存储数据。这是合情合理的，因为React是一个视图层。采用它的主要目的是渲染UI。不过，大部分应用程序至少都存在某种后端，并且我们需要理解如何将应用程序和服务器端整合到一起。

即使用户的客户端应用程序完全依赖于使用云服务作为后端，仍然需要从这些服务中读取或者发送数据。在Flux的影响范围内，有特定的位置处理这些传输，并且还有不少类库可以帮助用户处理和HTTP请求有关的延迟问题。

此外，React还支持同构渲染，这意味着它能够兼容浏览器之外的其他平台。同时也说明我们在页面内容抵达浏览器之前，在服务器端可以对UI进行渲染。利用服务器端渲染，我们可以提高应用程序的性能、可移植性和安全性。

从本章开始，我们将会学习同构性和通用性之间的区别，以及这两个概念和React之间的关系。接下来，将会介绍如何使用通用的JavaScript构建一个同构的应用程序。最后，我们会通过添加一个服务器并首先在服务器端渲染UI来改进颜色管理器程序。

同构性和通用性

同构性和通用性这两个术语通常用于描述在客户端和服务器端均可使用的应用程序。虽然这两个术语在描述相同应用程序时可以互换，但是它们之间存在细微的差别。同

构性应用程序是可以在多个平台上渲染的程序。通用性代码意味着完全相同的代码可以在多种环境下运行。[注1]

Node.js允许用户复用相同的代码，比如在浏览器中执行的代码、其他诸如服务器、命令行，甚至移动设备原生应用程序代码。让我们来看看一段通用性JavaScript代码：

```
var printNames = response => {
  var people = JSON.parse(response).results,
      names = people.map(({name}) => `${name.last}, ${name.first}`)
  console.log(names.join('\n'))
}
```

printNames函数是通用的。几乎与之完全相同的代码既可以在客户端执行，也可以在服务器端执行。这意味如果我们使用Node.js构建了一个服务器端，我们可能会在两种环境之间复用大量代码。通用性JavaScript代码是在没有异常出现的情况下，既可以在服务器端执行，也可以在浏览器中执行的JavaScript代码（见图12-1）。

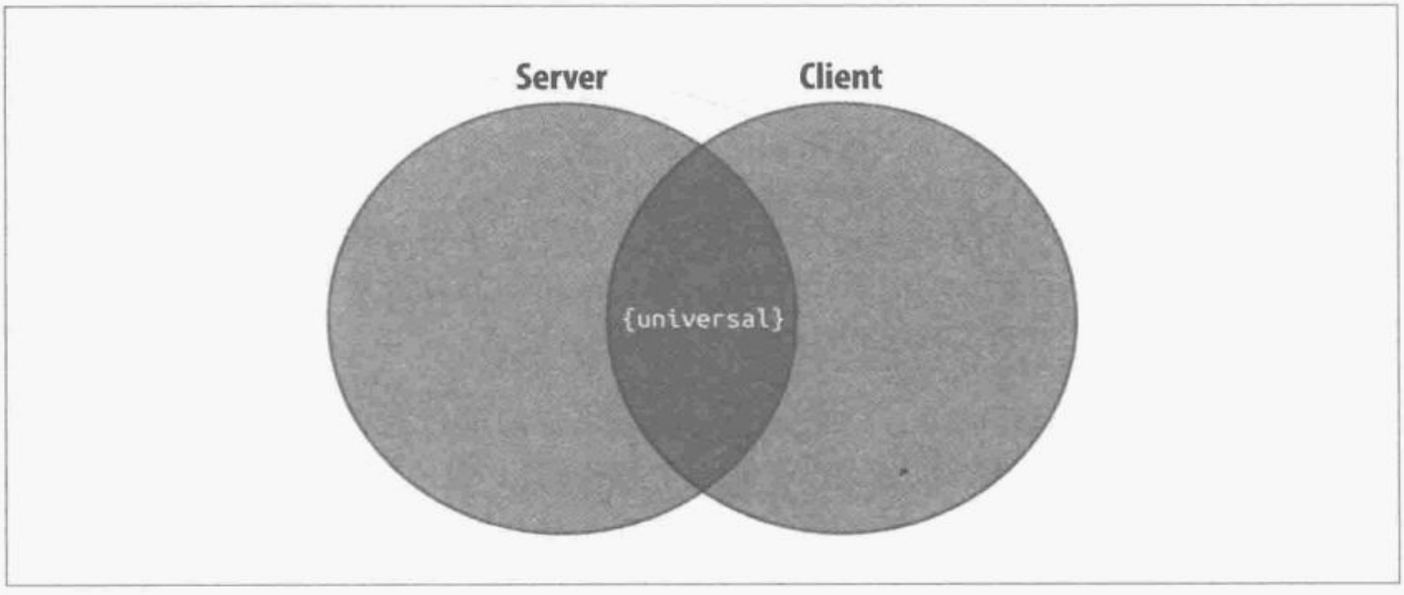


图12-1：客户端和服务器端的作用域

服务器端和客户端是完全不同的作用域，因此用户的JavaScript代码不会自动兼容它们。让我们看看在浏览器中如何创建一个AJAX请求：

```
const request = new XMLHttpRequest()
request.open('GET', 'https://api.randomuser.me/?nat=US&results=10')
request.onload = () => printNames(request.response)
request.send()
```

注1：Gert Hengeveld，“Isomorphism vs Universal JavaScript”，Medium（*http://bit.ly/2m0YDEY*）。

这里我们从*randomuser.me*的API服务中随机请求了10条用户记录。如果我们在浏览器中运行这些代码，将会看到返回的10条随机用户名记录：

```
ford, brianna
henderson, nellie
lynch, lily
gordon, todd
collins, genesis
roberts, suzanne
dixon, rene
ray, rafael
adams, jamie
bowman, mia
```

不过，如果我们尝试在Node.js中运行于上述代码几乎相同的代码，将会看到一个错误提示信息：

```
ReferenceError: XMLHttpRequest is not defined
    at Object.<anonymous> (/Users/...)
    at Module._compile (module.js:541:32)
    at Object.Module._extensions..js (module.js:550:10)
    at Module.load (module.js:458:32)
    at tryModuleLoad (module.js:417:12)
    at Function.Module._load (module.js:409:3)
    at Function.Module.runMain (module.js:575:10)
    at startup (node.js:160:18)
    at node.js:449:3
```

出现这个错误是因为Node.js并没有像浏览器那样包含一个`XMLHttpRequest`对象。在Node.js中，我们可以使用`http`模块创建一个请求：

```
const https = require('https')
https.get(
    'https://api.randomuser.me/?nat=US&results=10',
    res => {

        let results = ''

        res.setEncoding('utf8')
        res.on('data', chunk => results += chunk)

        res.on('end', () => printNames(results))
    }
)
```

在Node.js中从API服务中载入数据需要用到核心模块。它们用到的代码是不同的。在这些示例中，`printNames`函数是通用的，因此相同的函数在上述两个环境下都可以正常工作。

我们可以构建一个既可以在浏览器中，也可以在Node.js应用中将姓名输出到控制台的模块：

```
var printNames = response => {
    var people = JSON.parse(response).results,
    names = people.map(({name}) => `${name.last}, ${name.first}`)
    console.log(names.join('\n'))
}

  if (typeof window !== 'undefined') {

    const request = new XMLHttpRequest()
    request.open('GET', 'http://api.randomuser.me/?nat=US&results=10')
    request.onload = () => printNames(request.response)
    request.send()

  } else {

    const https = require('https')
    https.get(
      'http://api.randomuser.me/?nat=US&results=10',
      res => {
        let results = ''
        res.setEncoding('utf8')
        res.on('data', chunk => results += chunk)
        res.on('end', () => printNames(results))
      }
    )

  }
```

这个JavaScript文件现在是同构的，它包含了通用的JavaScript代码。所有代码并不是通用的，但是这个文件本身可以在上述两种环境下正常工作。它可以在Node.js中运行，也可以在浏览器中嵌入的<script>标签内部执行。

isomorphic-fetch函数

我们经常会优先选择比WHATWG fetch函数的其他实现更好的isomorphic-fetch函数，因为isomorphic-fetch函数可以兼容多种运行环境。

让我们来看看Star组件。这个组件可以变成通用的么？

```
const Star = ({ selected=false, onClick=f=>f }) =>
    <div className={(selected) ? "star selected" : "star"}
         onClick={onClick}>
    </div>
```

当然没问题：请记住，JSX会被编译成JavaScript代码。Star组件只是一个函数：

```
const Star = ({ selected=false, onClick=f=>f }) =>
    React.createElement(
        "div",
        {
            className: selected ? "star selected" : "star",
            onClick: onClick
        }
    )
```

我们可以在浏览器中直接渲染这个组件，或者在其他环境中渲染它并将HTML输出结果转化成一个字符串。ReactDOM有一个renderToString方法，我们可以用来将渲染的UI转化成一个字符串：

```
// 直接在浏览器中渲染html
ReactDOM.render(<Star />)

// 将html渲染成一个字符串
var html = ReactDOM.renderToString(<Star />)
```

我们可以构建能够在不同平台上渲染组件的同构性应用程序，并且可以通过跨越多个环境复用通用JavaScript代码的方式构建这些应用程序。

此外，我们还可以使用其他编程语言，比如Go或者Python来构建同构性应用程序，而不受限于Node.js环境。

服务器端渲染React

使用ReactDOM.renderToString方法允许用户在服务器端渲染UI。服务器的功能非常强大，它能够访问所有资源的能力是浏览器所不具备的。服务器可以非常安全，并且可以安全地访问数据。用户可以在服务器端渲染初始内容的优势之上充分利用这些额外的优点。

让我们使用Node.js和Express构建一个基本的Web服务器。Express是一款脚本库，我们可以利用它快速开发Web服务器：

```
npm install express --save
```

让我们来看一个简单的Express应用。这些代码将会创建一个总是显示“Hello World”信息的Web服务器。首先，和每个请求有关的信息将会输出到控制台上。然后服务器会返回一些HTML页面响应请求。这两个步骤都包含在它们自身的函数中，并且和.use()方法串联到了一起。Express会自动将请求和响应结果作为参数注入这些函数。

```
import express from 'express'
```

```
const logger = (req, res, next) => {
    console.log(`${req.method} request for '${req.url}'`)
    next()
}

const sayHello = (req, res) =>
    res.status(200).send("<h1>Hello World</h1>")

const app = express()
    .use(logger)
    .use(sayHello)

app.listen(3000, () =>
    console.log(`Recipe app running at 'http://localhost:3000'`)
)
```

logger和sayHello函数是中间件。在Express中，中间件函数会和.use()方法链接在一起并整合到某个管道中。[注2]当某个请求发生时，每个中间件函数会依次执行直到返回相关的响应结果。Express应用程序会将每个请求的细节记录输出到控制台，然后发送一个HTML响应：<h1>Hello World</h1>。最后，我们启动Express应用程序时，可以让它监听本地输入请求的3000端口。

在第10章我们曾使用babel-cli执行测试程序。这里我们将会使用babel-cli运行这个Express应用程序，因为它可以兼容当前版本的Node.js不支持的ES6的import语句。

babel-cli并不是在生产环境下运行这个应用程序的绝佳解决方案，而且我们不必使用babel-cli运行每个采用ES6规范的Node.js应用程序。编写本书时，当前版本的Node.js已经支持大部分ES6语法。用户可以选择不使用import语句即可。将来版本的Node.js将会支持import语句。*

另外一种办法是为用户的后端代码创建一个webpack构建。webpack可以导出JavaScript bundle文件，使得它可以在Node.js旧版本上运行。

为了运行babel-node，还需要做一些配置。首先，我们需要安装babel-cli、babel-loader、babel-preset-es2015、babel-preset-react和babel-preset-stage-0等软件包：

```
npm install babel-cli babel-loader babel-preset-env
babel-preset-react babel-preset-stage-0 --save
```

然后，我们需要确保在项目根目录下添加了一个*.babelrc*文件。当运行run babel-node index-server.js命令时，Babel将会查找这个文件并应用我们预设的安装程序：

注2： Express 开发文档，“Using Middleware”（http://bit.ly/2m0Z2ax）。

```
{
  "presets": [
    "env",
    "stage-0",
    "react"
  ]
}
```

最后，我们将会在*package.json*中添加一个启动脚本。如果用户没有*package.json*文件，那么可以通过执行npm init命令创建一个：

```
"scripts": {
    "start": "./node_modules/.bin/babel-node index-server.js"
}
```

现在我们可以使用npm start命令启动运行Express服务器了：

```
npm start

Recipe app running at 'http://localhost:3000'
```

服务器一旦启动，可以在Web浏览器上导航到*http://localhost:3000*。用户将会看到包含“Hello World”信息的页面。

ctrl^c将会终止正在运行的Express服务器。

目前为止，Express应用响应所有请求都是通过相同的字符串：”<h1>Hello World</h1>”。除了渲染这个消息之外，还可以显示第4章和第5章介绍的菜谱应用。我们可以通过使用ReactDOM的renderToString方法向Menu组件中渲染显示一些菜谱数据：

```
import React from 'react'
import express from 'express'
import { renderToString } from 'react-dom/server'
import Menu from './components/Menu'
import data from './assets/recipes.json'

global.React = React

const html = renderToString(<Menu recipes={data}/>)

const logger = (req, res, next) => {
    console.log(`${req.method} request for '${req.url}'`)
    next()
}
```

```
const sendHTMLPage = (req, res) =>
    res.status(200).send(`
<!DOCTYPE html>
<html>
    <head>
        <title>React Recipes App</title>
    </head>
    <body>
        <div  id="react-container">${html}</div>
    </body>
</html>
    `)

const app = express()
    .use(logger)
    .use(sendHTMLPage)

app.listen(3000, () =>
    console.log(`Recipe app running at 'http://localhost:3000'`)
)
```

首先，我们需要导入react、renderToString方法、Menu组件，以及一些用于初始化的菜谱数据。React是全局可以访问的，所以renderToString方法可以正常工作。

接下来，通过调用renderToString函数获取HTML，然后将它发送给Menu组件。

最后，我们可以创建一个新的中间件函数sendHTMLPage，它将会使用一个HTML字符串响应所有请求。这个字符串将服务器端渲染的HTML包装到一个模板中是创建一个页面所必需的。

当用户启动该应用程序，然后在浏览器中导航到*http://localhost:3000*后，将会发现菜谱应用已经被渲染显示了。在这个响应中并没有引用任何JavaScript代码。菜谱已经通过HTML文件的形式展现在页面中了。

目前为止，我们已经拥有了经过服务器端渲染的Menu组件。该应用程序并不是同构的，因为上述组件只能在服务器端渲染。为了实现同构性，我们将会在响应内容中添加一些JavaScript代码，以便相同的组件也能够在浏览器中渲染显示。

我们来创建一个可以在浏览器中运行的*index-client.js*文件：

```
import React from 'react'
import { render } from 'react-dom'
import Menu from './components/Menu'

window.React = React

alert('bundle loaded, Rendering in browser')
```

```
render(
    <Menu recipes={__DATA__} />,
    document.getElementById("react-container")
)

alert('render complete')
```

这个文件将会使用相同的菜谱数据来渲染同一Menu组件。我们之所以知道数据是相同的，是因为它们已经作为响应结果的一个字符串被引用了。当浏览器载入这个脚本时，__DATA__已经存在于全局作用域了。alert方法是用于查看浏览器何时渲染UI元素的。

我们将需要把这个*client.js*文件打包到一个bundle文件中，方便浏览器调用。这里，基本的webpack配置就可以处理这种构建了。

务必记得安装webpack，我们已经安装了babel和必要的预置程序：

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

这里，基本的webpack配置就可以满足需要了：

```
var webpack = require("webpack")

module.exports = {
    entry: "./index-client.js",
    output: {
        path: "assets",
        filename: "bundle.js"
    },
    module: {
        rules: [
            {
                test: /\.js$/,
                exclude: /(node_modules)/,
                loader: 'babel-loader',
                query: {
                    presets: ['env', 'stage-0', 'react']
                }
            }
        ]
    }
}
```

我们希望每次启动应用程序时自动构建客户端的bundle文件，因此将会需要在*package.json*文件中添加一个预启动脚本：

```
"scripts": {
    "prestart": "./node_modules/.bin/webpack --progress",
```

```
    "start": "./node_modules/.bin/babel-node index-server.js"
},
```

最后一步是修改服务器端配置。我们需要将初始的__DATA__作为一个字符串写入响应结果中，还需要添加一个引用客户端bundle文件的script标签。最后，我们需要确保服务器端发送的静态文件是来自*./assets/*目录的：

```
const sendHTMLPage = (req, res) =>
    res.status(200).send(`
<!DOCTYPE html>
<html>
    <head>
        <title>React Recipes App</title>
    </head>
    <body>
        <div id="react-container">${html}</div>
        <script>
            window.__DATA__ = ${JSON.stringify(data)}
        </script>
        <script src="bundle.js"></script>
    </body>
</html>
`)

const app = express()
    .use(logger)
    .use(express.static('./assets'))
    .use(sendHTMLPage)
```

script标签已经直接被添加到响应结果中。被写入的数据会放在第一个script标签中，载入的bundle文件会被放在第二个script标签中。此外。中间件被添加到了请求管道中。当*/bundle.js*文件被请求时，中间件express.static将会使用这个文件替代服务器端渲染的HTML进行响应，因为该文件是在*./assets*文件夹下的。

现在我们可以同构渲染React组件了，首先是在服务器端，然后是在浏览器中。当用户运行该应用程序时，将会看到这些组件在浏览器中被渲染之前和之后弹出相应的alert提示框。用户可能已经注意到在清除第一个alert提示框之前，相关的内容仍然在那里。这是因为它们是在服务器端初始化渲染的。

听上去渲染显示相同的内容两次似乎有点傻，不过它也有不少优点。应用程序可以在所有浏览器中渲染相同的内容，即使客户端禁用了JavaScript。因为页面内容已经和初始化请求一起载入了，用户的网站性能更好，并且可以将必要的内容更快速地分发给移动设备用户。[注3]不需要等待移动设备处理器渲染UI，UI元素已经准备就绪了。此

注3：Andrew H. Farmer, "Should I use React Server-Side Rendering?" (http://bit.ly/2m11mOI)。

外，这个应用程序具备SPA的所有优点。同构性React应程序能够兼顾双方的优点，提供最佳的用户体验。

通用颜色管理器

在前面5章中，我们构建了一个颜色管理器应用程序。到目前为止，我们已经基于这个应用程序编写了大量的代码。我们编写了若干React组件、一个Redux的Store，以及大量的Action生成器和辅助函数。

同时还集成了React Router。如果我们想创建一个Web服务器，那么现在也有大量的代码可以复用。

让我们为这个应用程序创建一个Web服务器，并尽可能复用以前的代码。首先我们将会需要一个模块用于配置Express应用实例,因此接下来将会创建*./server/app.js*文件：

```
import express from 'express'
import path from 'path'
import fs from 'fs'

const fileAssets = express.static(
    path.join(  dirname, '../../dist/assets')
)

const logger = (req, res, next) => {
    console.log(`${req.method} request for '${req.url}'`)
    next()
}

const respond = (req, res) =>
    res.status(200).send(`
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Universal Color Organizer</title>
    </head>
    <body>
        <div id="react-container">ready...</div>
    </body>
</html>
`)

export default express()
    .use(logger)
    .use(fileAssets)
    .use(respond)
```

这个模块是我们通用应用程序的起点。Express配置使用中间件来处理日志和文件资源，并最终通过一个HTML页面响应每个请求。

因为我们能够提供的HTML直接来自这个文件，因此需要将./dist/index.html文件移除。如果该文件还存在的话，它将会在服务器响应之前呈现给用户。

webpack允许用户导入诸如CSS或者图片这类资源文件，不过Node.js将无法处理这些导入的资源。我们将会需要使用ignore-styles库来确保忽略了任何SCSS文件中的import语句。接下来将安装ignore-styles：

```
npm install ignore-styles --save
```

在*./src/server/index.js*文件中，我们将使用Express应用实例并启动服务器。该文件表示Node.js服务器的入口点：

```
import React from 'react'
import ignoreStyles from 'ignore-styles'
import app from './app'

global.React = React

app.set('port', process.env.PORT || 3000)
    .listen(
        app.get('port'),
        () => console.log('Color Organizer running')
    )
```

这个文件将添加React为全局实例并启动服务器。此外，我们已经引入了ignore-styles模块，它将会忽略这些import语句，以便我们可以在Node.js中无障碍地渲染组件。

现在我们已经有了一个启动点：一个基本的Express应用程序配置。当我们需要在该服务器上添加新特性时，用户需要按照一定的方式将这些特性添加到该应用配置模块中。

在本章接下来的内容中，我们将会对这个Express应用程序进行迭代。将会使用更通用的代码构建一个同构/通用版本的颜色管理器。

通用的Redux

Redux库中的所有JavaScript代码都是通用的。用户使用JavaScript编写的Reducer也

不应该包含任何与执行环境有关的代码。Redux的设计初衷是用作浏览器应用程序的State容器的，不过它也适用于所有Node.js应用程序，其中包括CLI、服务器，以及原生的移动应用程序。

我们已经有Redux的Store代码了。接下来将会在服务器上使用这个Store将State的变化另存为一个JSON文件。首先，我们需要修改storeFactory，以便它可以同构运行。目前，storeFactory中包含的日志记录中间件将会在Node.js中导致错误，因为它使用了console.groupCollapsed和console.groupEnd方法。这两个方法都无法在Node.js中运行。如果我们在服务器端创建Store，那么需要使用其他的日志记录器：

```
import { colors } from './reducers'
import {
    createStore, combineReducers, applyMiddleware
} from 'redux'

const clientLogger = store => next => action => {
    let result
    console.groupCollapsed( "dispatching", action.type)
    console.log('prev state', store.getState())
    console.log('action', action)
    result = next(action)
    console.log('next state', store.getState())
    console.groupEnd()
    return result
}

const serverLogger = store => next => action => {
    console.log('\n dispatching server action\n')
    console.log(action)
    console.log('\n')
    return next(action)
}

const middleware = server =>
    (server) ? serverLogger : clientLogger

const storeFactory = (server = false, initialState = {}) =>
    applyMiddleware(middleware)(createStore)(
        combineReducers({colors}),
        initialState
)

export default storeFactory
```

现在storeFactory是同构的了。我们在服务器端创建了专门处理日志的Redux中间件。当storeFactory被调用时，将需要用户告知它希望使用哪一种Store，并将相应的日志记录器添加到新的Store实例中。

现在我们将会使用这个同构storeFactory来创建一个serverStore实例。在Express配置的顶部，我们需导入storeFactory和初始的State数据。可以使用storeFactory从一个JSON文件中读取初始的State数据来创建一个Store：

```
import storeFactory from '../store'
import initialState from '../../data/initialState.json'

const serverStore = storeFactory(true, initialState)
```

现在我们已经有了一个可以运行在服务器端的Store实例。

每次某个Action被分发到该实例时，我们希望确保*initialState.json*文件也被更新了。执行subscribe后，我们可以监听State的变化，并在每次State发生变化后将它另存为一个新的JSON文件：

```
serverStore.subscribe(() =>
    fs.writeFile(
        path.join( dirname, '../../data/initialState.json'),
        JSON.stringify(serverStore.getState()),
        error => (error) ?
            console.log("Error saving state!", error) :
            null
    )
)
```

当Action被分发后，会使用fs模块将新的State数据保存到*initialState.json*文件中。

serverStore现在是主要的真实数据源。任何请求都需要和它交互，以便获取当前最新的颜色列表。还将需要添加一些中间件来将服务器端的Store添加到请求管道中，以便请求过程中其他中间件可以使用它：

```
const addStoreToRequestPipeline = (req, res, next) => {
    req.store = serverStore
    next()
}

export default express()
    .use(logger)
    .use(fileAssets)
    .use(addStoreToRequestPipeline)
    .use(htmlResponse)
```

现在任何在addStoreToRequestPipeline之后的中间件方法都可以在请求对象上访问Store了。我们已经采用了通用的Redux。Store的代码也几乎一样，其中包括若干Reducer，它们将能够兼容多种运行环境。

有一些在为大型应用程序构建Web服务器时有关的难题并没有在这个示例中表现出来。为了数据持久化将数据保存到JSON文件只是权宜之计，但是正式上线的应用程序实际上采用的是数据库。采用Redux技术后可能会满足某些应用程序的需求。但是大型应用程序中和派生节点过程有关的问题仍然需要用户引起重视。用户可以考察Firebase，以及其他类似的云服务提供商的解决方案，以协助用户能够平滑地对数据库进行扩展。

通用路由

在上一章中，我们将react-router-dom添加到了颜色管理器中。router会根据浏览器当前的地址渲染相关的组件。router还可以在服务器端执行渲染任务，只要我们提供相关的地址和路由。

目前为止，我们采用的都是HashRouter。这个router会自动在每个路径前面添加一个#。为了让该router支持同构性，我们需要用BrowserRouter代替HashRouter，前者会将路由前面的#移除。

当渲染应用程序时，我们需要用BrowserRouter替换HashRouter：

```
import { BrowserRouter } from 'react-router-dom'

...

render(
    <Provider store={store}>
        <BrowserRouter>
          <App />
        </BrowserRouter>
    </Provider>,
    document.getElementById('react-container')
)
```

现在颜色管理器程序不需要在每个路径前面添加哈希字符前缀（#）。此时，颜色管理器仍然可以正常工作。启动该程序并选择一种颜色。Color容器仍旧被渲染了，并且它使用ColorDetails组件修改这个屏幕的背景色。

现在地址栏上的信息应该和下列内容类似：

```
http://localhost:3000/8658c1d0-9eda-4a90-95e1-8001e8eb6036
```

路径前面不再包含#前缀。现在让我们在浏览器中刷新这个页面：

```
Cannot GET /8658c1d0-9eda-4a90-95e1-8001e8eb6036
```

刷新该页面后，浏览器会使用当前的路径向服务器端发送一个GET请求。#是用来阻止用户发送上述GET请求的。我们使用BrowserRouter是因为希望将上述GET请求发送到服务器端。为了在服务器端渲染router，我们需要一个地址，我们需要获得路径。在服务器端这个路径将会用于告知router渲染Color容器。当用户希望同构性地渲染路由时才会使用BrowserRouter。

现在我们已经知道用户请求的内容是什么，让我们据此在服务器端渲染UI。为了在服务器端渲染UI，必须对Express配置做一些重大变更。首先，我们将需要导入一些模块：

```
import { Provider } from 'react-redux'
import { compose } from 'redux'
import { renderToString } from 'react-dom/server'
import { StaticRouter } from 'react-router-dom'
```

我们需要Provider、一个compose函数、renderToString函数，以及StaticRouter。在服务器端，当用户希望将组件树渲染成一个字符串时会用到StaticRouter。

为了生成一个HTML响应，需要经过三个步骤：

1. 使用serverStore中的数据创建一个可以运行在客户端的Store。

2. 使用StaticRouter将组件树渲染成HTML形式。

3. 创建将会被发送到客户端的HTML页面。

我们为上述每个步骤创建了一个函数，然后将它合成为单个函数，即htmlResponse：

```
const htmlResponse = compose(
    buildHTMLPage,                      //第3步
    renderComponentsToHTML,             //第2步
    makeClientStoreFrom(serverStore)    //第1步
)
```

在这个合成中，makeClientStoreFrom(serverStore)是一个高阶函数。起初，该函数会被serverStore调用一次。它返回的函数将会被每个请求调用。返回的函数将一直可以访问serverStore。

当htmlResponse被触发后，它需要接收单个参数：即用户请求的url地址。对于第一步，我们将会创建一个高阶函数，使用服务器端Store当前的State，与上述url一起打包创建一个新的客户端Store。Store和url都会被传递给下一个函数，即存放于第二步中的单个对象：

```
const makeClientStoreFrom = store => url =>
    ({
        store: storeFactory(false, store.getState()),
        url
})
```

makeClientStoreFrom函数的输出结果将会成为renderComponentsToHTML函数的输入。该函数预期的结果是url和Store已经被打包存储到单个参数对象中：

```
const renderComponentsToHTML = ({url, store}) =>
    ({
        state: store.getState(),
        html: renderToString(
            <Provider store={store}>
                <StaticRouter location={url} context={{}}>
                    <App />
                </StaticRouter>
            </Provider>
        )
    })
```

renderComponentsToHTML函数会返回一个包含两个属性的对象：state和html。state是从新的客户端Store中获取的，html是由renderToString方法生成的。因为该应用程序在浏览器环境下仍然采用了Redux，Provider会被当作根组件渲染，新的客户端Store会作为一个属性传递给它。

StaticRouter组件是用来根据被请求的路径地址渲染UI的。StaticRouter请求了一个路径和上下文。请求的url是通过location属性传递的，上下文数据是通过一个空对象给出的。当这些组件被渲染成一个HTML字符时，StaticRouter会根据上述路径地址，渲染正确的路由。

该函数会返回两个必需的组件来构建页面：颜色管理器的当前State，以及被渲染UI的HTML字符串。

state和html还可以被最后一个合成函数调用，即buildHTMLPage：

```
const buildHTMLPage = ({html, state}) => `
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Universal Color Organizer</title>
    </head>
    <body>
        <div id="react-container">${html}</div>
        <script>
            window.__INITIAL_STATE__ = ${JSON.stringify(state)}
        </script>
```

```
        <script src="/bundle.js"></script>
    </body>
</html>
`
```

我们的颜色管理器现在是同构的了。它将会在服务器端渲染UI并将它作为文本发送到客户端，还会直接在响应文本中嵌入Store的初始State数据。

浏览器初始显示的UI是从HTML响应文本中获取的。当该bundle文件载入后，它会重新渲染UI并且客户端会接管一切。从这个节点之后，所有的用户交互，包括导航浏览都是在客户端发生的。我们的单页应用程序将会始终具有该功能，直到浏览器页面被刷新，此时服务器端渲染过程又再次重新开始了。

以下是当前Express应用程序的所有模块代码，即整个文件：

```
import express from 'express'
import path from 'path'
import fs from 'fs'
import { Provider } from 'react-redux'
import { compose } from 'redux'
import { StaticRouter } from 'react-router-dom'
import { renderToString } from 'react-dom/server'
import App from '../components/App'
import storeFactory from '../store'
import initialState from '../../data/initialState.json'

const fileAssets = express.static(
    path.join(  dirname, '../../dist/assets')
)

const serverStore = storeFactory(true, initialState)

serverStore.subscribe(() =>
    fs.writeFile(
        path.join( dirname, '../../data/initialState.json'),
        JSON.stringify(serverStore.getState()),
        error => (error) ?
            console.log("Error saving state!", error) :
            null
    )
)

const logger = (req, res, next) => {
    console.log(`${req.method} request for '${req.url}'`)
    next()
}

const addStoreToRequestPipeline = (req, res, next) => {
    req.store = serverStore
    next()
}
```

```
const makeClientStoreFrom = store => url =>
    ({
        store: storeFactory(false, store.getState()),
        url
    })

const renderComponentsToHTML = ({url, store}) =>
    ({
        state: store.getState(),
        css: defaultStyles,
        html: renderToString(
            <Provider store={store}>
                <StaticRouter location={url} context={{}}>
                    <App />
                </StaticRouter>
            </Provider>
        )
    })

const buildHTMLPage = ({html, state}) => `
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Universal Color Organizer</title>
    </head>
    <body>
        <div id="react-container">${html}</div>
        <script>
            window.  INITIAL_STATE   = ${JSON.stringify(state)}
        </script>
        <script src="/bundle.js"></script>
    </body>
</html>
`

const htmlResponse = compose(
    buildHTMLPage,
    renderComponentsToHTML,
    makeClientStoreFrom(serverStore)
)

const respond = (req, res) =>
    res.status(200).send(htmlResponse(req.url))

export default express()
    .use(logger)
    .use(fileAssets)
    .use(addStoreToRequestPipeline)
    .use(respond)
```

我们的应用程序现在支持用户添加书签的URL，以及发送给其他用户的URL都可以进

行同构性渲染了。router会根据URL决定将要渲染显示的内容。在服务器端也可以这么做，这意味着用户可以更快速地访问页面内容。

同构性应用程序能够兼具两方面的优势：它们可以利用服务器端渲染的高速、可控和安全等优点，同时可以从单页应用程序低带宽和速度方面获益。一个同构性React应用程序本质上就是一个服务器端渲染的SPA，这为用户构建构建高效应用程序奠定了坚实的基础，使得这些应用程序既可以变得酷炫，也可以快速高效。

集成服务器端渲染样式

目前，我们已经在服务器端渲染了HTML，但是直到bundle文件在浏览器中加载完毕之前，CSS样式都没有被渲染。这是由于一个奇怪的闪烁问题导致的。初始化时，在CSS文件被载入之前，我们将会在浏览器中看到所有无样式的内容。当浏览器中禁用JavaScript时，用户将无法看到任何CSS样式，因为它们被嵌入JavaScript的bundle文件中了。

解决方案是直接在响应内容中添加样式。为此，我们必须首先将CSS代码从webpack的bundle文件中提取出来，并将之另存为一个单独的文件。用户将需要安装extract-text-webpack-plugin插件：

```
npm install extract-text-webpack-plugin
```

同时还需要在webpack配置文件中引用该插件：

```
var webpack = require("webpack")
var ExtractTextPlugin = require("extract-text-webpack-plugin")
var OptimizeCss = require('optimize-css-assets-webpack-plugin')
```

另外，在webpack配置文件，我们需要使用ExtractTextPlugin插件替换原来的CSS和SCSS加载器：

```
{
    test: /\.css$/,
    loader: ExtractTextPlugin.extract({
        fallback: "style-loader",
        use: [
            "style-loader",
            "css-loader",
            {
                loader: "postcss-loader",
                options: {
                    plugins: () => [require("autoprefixer")]
                }
            }
        ]
```

```
        })
    },
    {
        test: /\.scss/,
        loader: ExtractTextPlugin.extract({
            fallback: "style-loader",
            use: [
                "css-loader",
                {
                    loader: "postcss-loader",
                    options: {
                        plugins: () => [require("autoprefixer")]
                    }
                },
                "sass-loader"
            ]
        })
    }
```

然后我们需要在配置文件中将插件添加到插件数组以便引用该插件。当插件被引用后，我们需要声明希望提取的CSS文件名称：

```
plugins: [
    new ExtractTextPlugin("bundle.css"),
    new OptimizeCss({
        assetNameRegExp: /\.optimize\.css$/g,
        cssProcessor: require('cssnano'),
        cssProcessorOptions: {
            discardComments: {removeAll: true}
        },
        canPrint: true
    })
]
```

现在当我们运行webpack后，它将不会把CSS文件添加到JavaScript的bundle文件中，相反，它会把所有CSS文件提取到独立文件*./assets/bundle.css*中。

我们还需要修改Express配置。当颜色管理器启动后，CSS文件会被另存为一个全局字符串。可以使用文件系统或者fs模块读取文本文件中的内容，然后写入变量staticCSS中：

```
const staticCSS = fs.readFileSync(
    path.join(  dirname, '../../dist/assets/bundle.css')
)
```

现在我们已经对buildHTMLPage函数做了一些修改，直接将CSS代码嵌入响应文本的<style>标签中：

```
const buildHTMLPage = ({html, state}) => `
<!DOCTYPE html>
```

```
<html>
    <head>
        <meta charset="utf-8">
        <title>Universal Color Organizer</title>
        <style>${staticCSS}</style>
    </head>
    <body>
        <div id="react-container">${html}</div>
        <script>
            window.  INITIAL_STATE   = ${JSON.stringify(state)}
        </script>
        <script src="/bundle.js"></script>
    </body>
</html>
```

CSS代码直接嵌入响应文本中。现在无样式的奇怪闪烁问题也被解决。当JavaScript被禁用时，样式仍然可以按照预期运行。我们拥有一个共享大量通用JavaScript代码的同构颜色管理器。

起初，颜色管理器是在服务器端渲染的，但是在客户端页面内容加载完毕之后，它还可以在浏览器中被渲染。当客户端浏览器接管一切后，颜色管理器就可以当作一款单页应用程序。

与服务器端交互

目前，颜色管理器是在服务器端渲染UI，然后在浏览器中重新渲染UI的。一旦浏览器接管一切，颜色管理器程序就可以被视为一个单页应用程序。用户在本地分发Action，本地存储State变化，并且在本地更新UI。在浏览器中所有事情都能够井然有序地进行着，但是上述被分发的Action并没有回传到服务器端。

在下一节中，我们不仅要确保数据可以在服务器端保存，同时还要确保在服务器端创建Action自身，然后将它分发给两边的Store。

在服务器端完成Action

在颜色管理器程序中，我们将会集成一个REST API来处理数据。Action将会在客户端初始化并在服务器端完成，然后分发给两边的Store。`serverStore`将会把新的State保存到JSON文件中，客户端的Store将会触发UI更新。两边的Store将会统一分发相同的Action（见图12-2）。

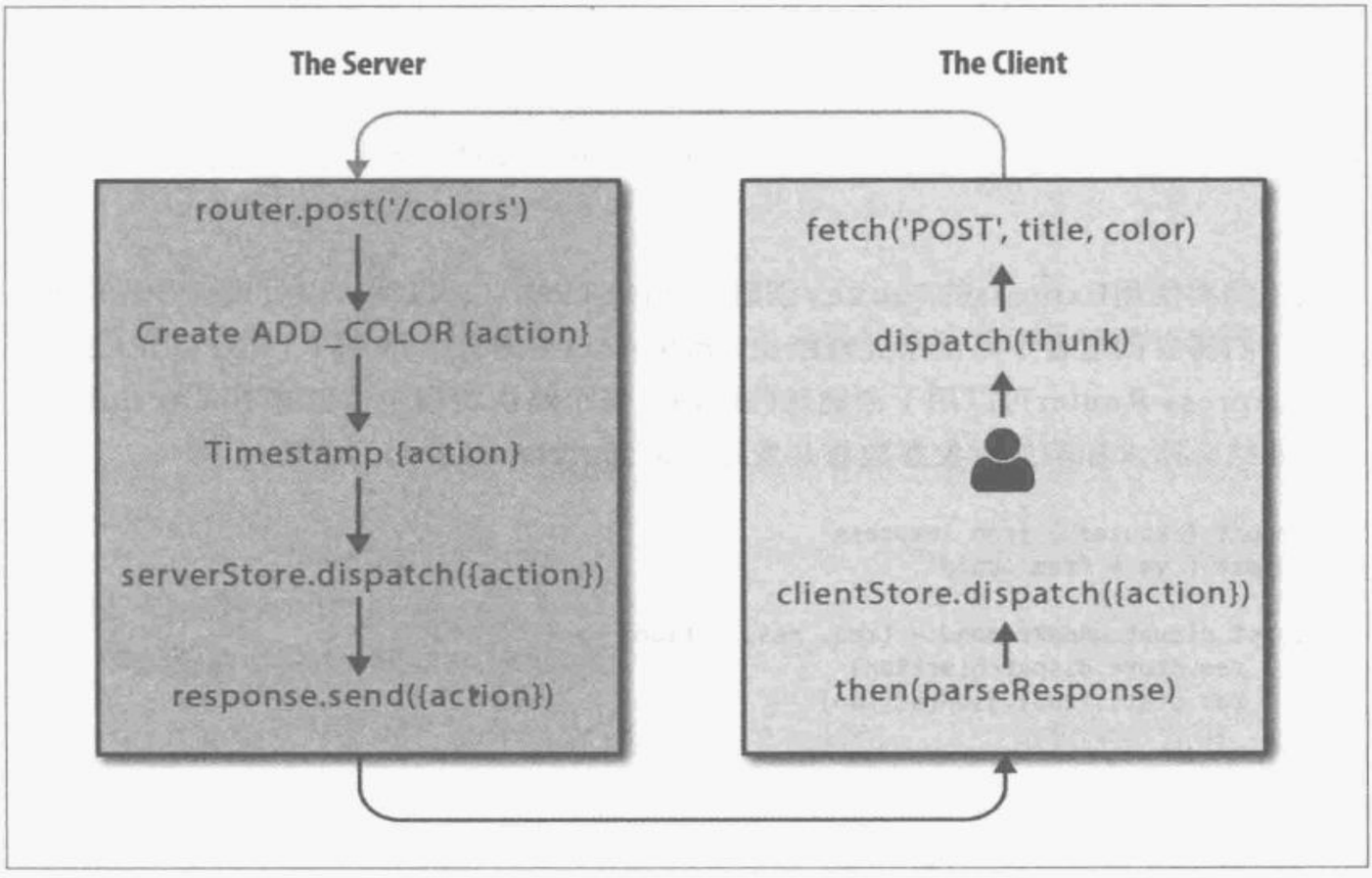


图12-2：创建一个通用的Action

让我们来看一个在常见解决方案中，分发ADD_COLOR这个Action完整过程的示例：

1. 使用新的标题和颜色调用Action生成器addColor()。
2. 在新的POST请求中发送数据到服务器端。
3. 在服务器端创建和分发新的ADD_COLOR这一Action。
4. 在响应体中发送ADD_COLOR这一Action。
5. 解析响应体，然后将ADD_COLOR这一Action分发到客户端。

我们首先需要做的事情是构建REST API。接下来将新建一个名为*./src/server/color-api.js*的文件。

创建每个Action的方式都是一样的：它在服务器端被分发，然后被发送到客户端。让我们创建一个函数，用于将Action分发给服务器端的Store，以及使用相应对象将上述Action发送到客户端：

```
const dispatchAndRespond = (req, res, action) => {
```

```
    req.store.dispatch(action)
    res.status(200).json(action)
}
```

一旦我们创建了一个Action，就可以使用上述函数分发该Action，并向客户端发送响应。

我们还需要使用Express的Router创建一些HTTP端点，以便可以处理多个HTTP请求。我们将会创建若干路由来处理路由*/api/colors*上的GET、POST、PUT和DELETE请求。Express Router可以用于创建这些路由。每个路由将会包含创建不同Action的逻辑，并结合请求和响应对象参数将其发送给dispatchAndRespond函数：

```
import { Router } from 'express'
import { v4 } from 'uuid'

const dispatchAndRespond = (req, res, action) => {
    req.store.dispatch(action)
    res.status(200).json(action)
}

const router = Router()

router.get("/colors", (req, res) =>
    res.status(200).json(req.store.getState().colors)
)

router.post("/colors", (req, res) =>
    dispatchAndRespond(req, res, {
        type: "ADD_COLOR",
        id: v4(),
        title: req.body.title,
        color: req.body.color,
        timestamp: new Date().toString()
    })
)

router.put("/color/:id", (req, res) =>
    dispatchAndRespond(req, res, {
        type: "RATE_COLOR",
        id: req.params.id,
        rating: parseInt(req.body.rating)
    })
)

router.delete("/color/:id", (req, res) =>
    dispatchAndRespond(req, res, {
        type: "REMOVE_COLOR",
        id: req.params.id
    })
)

export default router
```

每个被添加到router对象的函数会根据*http://localhost:3000/api/{route}*处理不同请求：

`GET '/colors'`
: 根据服务器端的State响应返回当前的颜色数组。这个路由是才添加的，因此我们可以查看颜色列表，并且前端并没有调用它。

`POST '/colors'`
: 创建一个新的颜色Action对象，然后将它发送给`dispatchAndRespond`函数。

`PUT '/color/:id'`
: 修改某个颜色的评分数据。该颜色的ID根据路由参数获取，并且在新的Action对象中会用到。

`DELETE '/color/:id'`
: 根据路由参数中传递的ID删除某个颜色。

现在我们已经定义了路由，接下来需要把它们添加到Express应用程序配置中。首先，我们需要安装Express的`body-parser`：

```
npm install body-parser --save
```

`body-parser`是用来解析传入的请求体，并获取发送路由的任意变量的。必须从客户端获取新的颜色和评分信息。我们将会把这个中间件添加到Express应用程序配置中。接下来将会在*./server/app.js*文件中导入`body-parser`和新的路由：

```
import bodyParser from 'body-parser'
import api from './color-api'
```

让我们把`bodyParser`中间件和API服务添加到Express应用程序中。在添加API之前添加`bodyParser`非常重要，以便API处理相关请求时可以解析数据：

```
export default express()
    .use(logger)
    .use(fileAssets)
    .use(bodyParser.json())
    .use(addStoreToRequestPipeline)
    .use('/api', api)
    .use(matchRoutes)
```

`bodyParser.json()`现在可以解析传入的请求体，并将解析结果格式化为JSON。我们的`color-api`被添加到了管道中，并被配置成响应任何前缀为*/api*的路由。比如，可以用来获取当前颜色数组JSON格式的URL是：*http://localhost:3000/api/colors*。

现在Express应用程序已经包含能够响应HTTP请求的若干端点了，我们将会修改前端的Action生成器来和这些端点交互。

采用了Redux Thunk的Action

客户端/服务器端交互的问题之一是延迟问题，或者发送一个请求后用户等待响应时经历的延迟。我们的Action生成器在能够分发Action之前需要等待一个响应，因为在我们的解决方案中，Action自身是从服务器端被发送到客户端的。Redux有一个可以帮助用户处理异步Action的中间件，它被称为redux-thunk。

在下一节中，我们将会使用redux-thunk重写Action生成器代码。这些Action生成器被称为thunks，它允许用户在分发本地Action之前等待服务器端响应。thunks是由一些高阶函数组成的。和Action对象相反，它们会返回其他函数。接下来将安装redux-thunk：

```
npm install redux-thunk --save
```

redux-thunk是中间件，它需要被集成到storeFactory中。首先，在*./src/Store/index.js*文件的顶部，导入redux-thunk：

```
import thunk from 'redux-thunk'
```

storeFactory包含一个名为middleware的函数。它返回的中间件将会以单个数组的形式被集成到新的Store中。我们可以添加任意Redux中间件到这个数组中。每个元素都会被作为参数传递给applyMiddleware函数：

```
const middleware = server => [
    (server) ? serverLogger : clientLogger,
    thunk
]
const storeFactory = (server = false, initialState = {}) =>
    applyMiddleware(...middleware(server))(createStore)(
        combineReducers({colors}),
        initialState
)

export default storeFactory
```

让我们来看看当前添加颜色的Action生成器：

```
export const addColor = (title, color) =>
    ({
        type: "ADD_COLOR",
        id: v4(),
        title,
```

```
            color,
            timestamp: new Date().toString()
        })

...

store.dispatch(addColor("jet", "#000000"))
```

这个Action生成器会返回一个对象，即addColor这一Action。该对象会马上被分发给Store。现在，让我们看看thunk版的addColor:

```
export const addColor = (title, color) =>
    (dispatch, getState) => {

        setTimeout(() =>
            dispatch({
                type: "ADD_COLOR",
                index: getState().colors.length + 1,
                timestamp: new Date().toString()
                title,
                color
            }),
            2000
        )
    }

...

store.dispatch(addColor("jet", "#000000"))
```

即使两个Action生成器的分发方式几乎一样，但是thunk会返回一个函数而不是一个对象。返回的函数是一个回调函数，并且会把Store的dispatch和getState方法作为参数接收。在用户准备好时就可以分发一个Action。在这个示例中，setTimeout函数用于在我们分发一个新的颜色Action之前创建一个2秒的延迟。

除了dispatch之外，thunks还可以访问Store的getState方法。在这个示例中，我们会使用它根据当前State中的颜色数目创建一个索引字段。当创建Action是基于Store中的数据时，上述函数将会非常有用。

并不是所有的Action生成器都必须转换成thunks。redux-thunk中间件能够区分thunks和Action对象之间的差异。Action对象会立刻被分发。

thunks还有另外一个优点。它们可以根据需要多次异步调用dispatch或者getState方法，并且不会受限于只能分发一种Action。在接下来的示例中，thunk将会立刻分发一

个RANDOM_RATING_STARTED，并使用随机数重复分发RATE_COLOR这一Action来给特定颜色评分：

```
export const rateColor = id =>
    (dispatch, getState) => {

        dispatch({ type: "RANDOM_RATING_STARTED" })
        setInterval(() =>
            dispatch({
                type: "RATE_COLOR",
                id,
                rating:  Math.floor(Math.random()*5)
            }),
            1000
        )
    }

...

store.dispatch(
    rateColor("f9005b4e-975e-433d-a646-79df172e1dbb")
)
```

这些thunk都是非常简单的示例。让我们构建一些可以用在颜色管理器上的真正thunk，以便可以替代当前正在使用的Action生成器：

首先，我们将构建一个名为fetchThenDispatch的函数。该函数使用isomorphic-fetch发送一个请求到一个Web服务上，然后自动分发响应结果：

```
import fetch from 'isomorphic-fetch'

const parseResponse = response => response.json()

const logError = error => console.error(error)

const fetchThenDispatch = (dispatch, url, method, body) =>
    fetch(
        url,
        {
            method,
            body,
            headers: { 'Content-Type': 'application/json' }
        }
    ).then(parseResponse)
     .then(dispatch)
     .catch(logError)
```

fetchThenDispatch函数将会需要dispatch函数、一个URL、HTTP的request方法，以及HTTP请求体作为参数。这些信息稍后会用于执行fetch函数。一旦接收到了响应，它将会被解析和分发。任何错误都会被记录到控制台。

我们将会使用fetchThenDispatch函数帮助用户构造thunk。每个thunk将会和任何必需的数据一起向API发送一个请求。因为API的响应内容是Action对象，所以响应结果可以马上被解析并分发：

```
export const addColor = (title, color) => dispatch =>
    fetchThenDispatch(
        dispatch,
        '/api/colors',
        'POST',
        JSON.stringify({title, color})
    )

export const removeColor = id => dispatch =>
    fetchThenDispatch(
        dispatch,
        `/api/color/${id}`,
        'DELETE'
    )

export const rateColor = (id, rating) => dispatch =>
    fetchThenDispatch(
        dispatch,
        `/api/color/${id}`,
        'PUT',
        JSON.stringify({rating})
    )
```

addColor这个thunk会联合新颜色的标题和十六进制颜色值一起发送一个POST请求到*http://localhost:3000/api/colors*。返回的一个ADD_COLOR的Action对象将会被解析和分发。

removeColor这个thunk会连同URL提供的颜色ID一起向API发送一个DELETE请求。返回的一个REMOVE_COLOR的Action对象将会被解析和分发。

rateColor这个thunk会向API发送一个PUT请求。被评分颜色的ID会作为路由参数包含在URL中，并且新的评分数据是由请求体提供的。从服务器端返回的一个RATE_COLOR的Action对象将会被解析成JSON，然后被分发到本地Store。

现在当用户运行该应用程序时，将会在控制台上发现Action被分发给了两边的Store。浏览器的控制台是开发者工具的有机组成部分，服务器端控制台就是服务器启动的终端界面（见图12-3）。

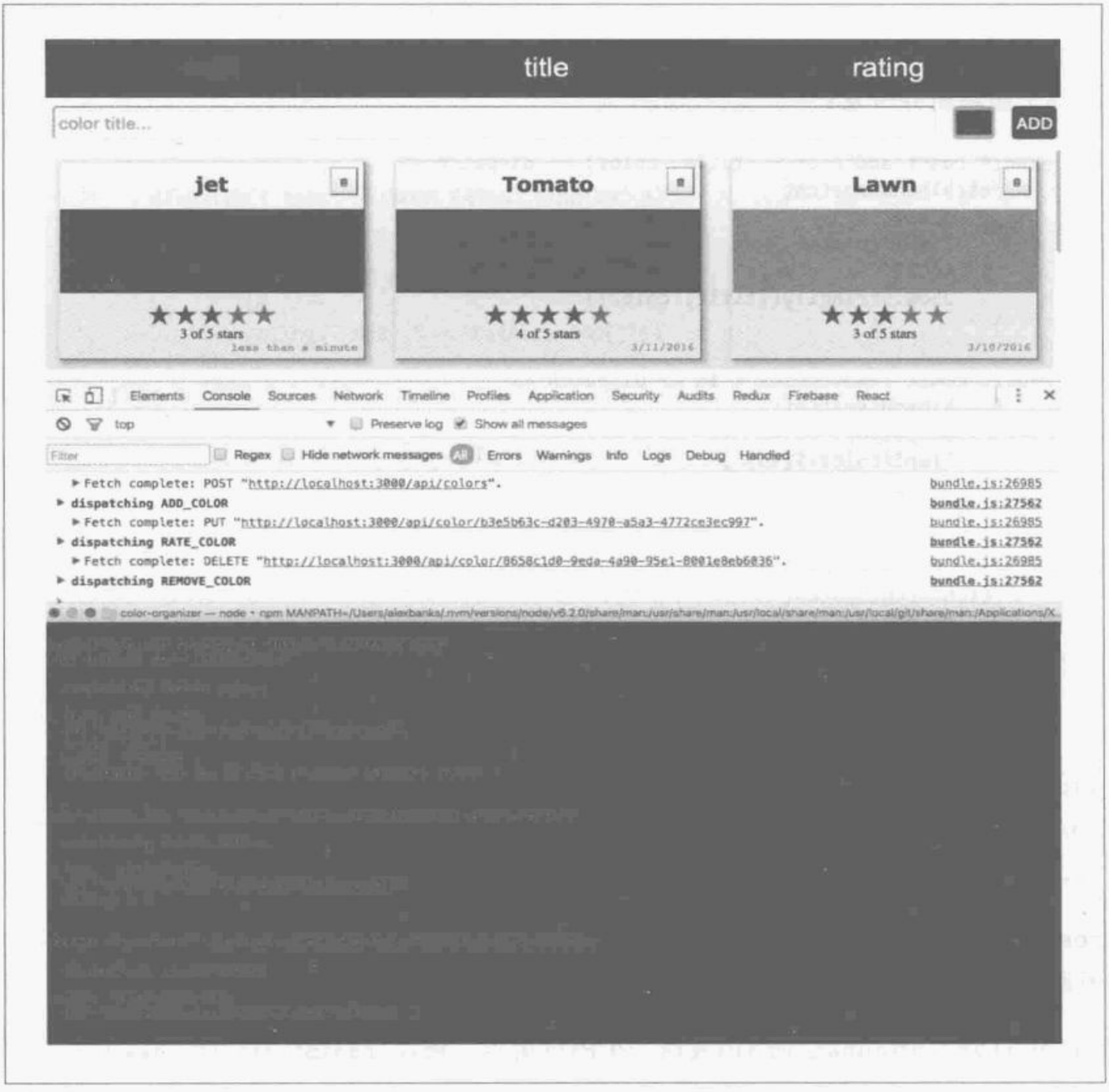


图12-3：浏览器端控制台和服务器端控制台

在thunk中使用websocket

颜色管理器程序是使用REST和服务器端交互的。在thunk中还可以使用websocket来收发信息。websocket在服务器端和客户端之间可以双向通信。websocket可以将数据发送到服务器端，同时它也允许服务器端将数据发送到客户端。

将websocket和thunk一起搭配使用的方法是分发一个Action生成器的连接。比

如，我们希望连接到某个信息服务器：

```
store.dispatch(connectToMessageSocket())
```

thunk可以按照要求分发相关数据。我们可以创建一些thunk来监听传入的信息，然后在接收到新信息后分发NEW_MESSAGE这一Action。接下来的示例将会使用socket.io-client连接到一个socket.io服务器，并监听输入的信息：

```
import io from 'socket.io-client'

const connectToChatSocket = () => dispatch => {

    dispatch({type: "CONNECTING"})

    let socket = io('/message-socket')

    socket.on('connect', () =>
        dispatch({type: "CONNECTED", id: socket.id})
    )

    socket.on('message', (message, user) =>
        dispatch({type: "NEW_MESSAGE", message, user})
    )

}

export default connectToMessageSocket
```

一旦connectToChatSocket被触发执行，一个CONNECTING这样的Action就会被分发。然后我们就可以尝试连接到socket信息服务器。连接成功后，socket将会使用一个connect事件予以响应。当上述结果发生后，我们可以分发一个包含当前socket信息的CONNECTED的Action。

当服务器端发送信息时，socket将会触发相应的信息事件。每次当相关信息从服务器端发送到客户端时，我们可以分发NEW_MESSAGE这样一个Action。

thunk可以兼容任意类型的异步进程，其中包括WebSocket、socket-io、Firebase、setTimeout、transition和animation。

用户构建的每个React应用程序都需要某种Web服务器的存在。有时可能用户只是需要一个服务器来托管该应用程序。其他情况下可能还需要和Web服务交互。然后高流量的应用程序需要兼容多种平台，在这种情况下所需的解决方案是完全不同的。

高级数据获取

如果你正在维护多个平台数据共享的高流量应用程序，则可能需要用到Relay、GraphQL或者Falcor这类框架。这些框架为应用程序的数据提供了更高效和智能的解决方案。

GraphQL（*http://graphql.org/*）是Facebook开发的声明式数据查询解决方案，可用于从多个来源查询数据。所有编程语言和平台都可以使用GraphQL。Relay是一个库（*https://facebook.github.io/relay/*），也是由Facebook开发的，通过将GraphQL查询与React或React Native组件相链接来处理客户端应用程序的数据获取。GraphQL和Relay的学习曲线有一点陡峭，不过如果你真的喜欢声明式编程，那么这是非常值得的。

Falcor（*https://netflix.github.io/falcor/*）是Netflix开发的一个框架，它解决了获取和有效使用数据相关的问题。像GraphQL一样，Falcor允许用户在单个位置查询来自多个服务的数据。然而，Falcor使用JavaScript查询数据，这意味着JavaScript开发人员的学习曲线更平缓。

React开发的关键是在正确的时间选择正确的工具。用户工具箱中已经拥有大量可以构建健壮应用程序的工具。现在只要按需取用即可。如果用户的应用程序没有涉及大量的数据，那么就不必使用Redux。React的State是一个非常适合普通规模应用程序的解决方案。用户的应用程序也许并不需要用到服务器端渲染。不用急于集成它们，直到用户的应用程序人机交互频繁，并且包含大量移动流量时再添加这些特性也不晚。

当读者开发自己的React应用程序时，我们希望本书可以作为参考并为你打下一个良好的基础。虽然React和相关的类库仍然在不断进化，但是这些成熟的工具可以确保读者能够以正确的方式使用它们。使用React、Redux，以及函数式、声明式JavaScript构建应用程序的过程将会充满乐趣，我们对你们构建的程序满怀期待。

作者介绍

Alex Banks是北加利福尼亚州软件开发培训公司Moon Highway的软件工程师、讲师和共同创始人。作为软件顾问，他先后为芝加哥马拉松、MSN和能源部提供过服务。Alex协助开发了软件持续交付课程，该课程会作为Yahoo新员工的培训材料。此外，他还是Lynda.com上若干课程的作者。

Eve Porcello是一名软件架构师和Moon Highway公司的共同创始人。在Moon Highway公司之前，Eve曾就职于1-800-Dentist和 Microsoft。她是一名活跃的企业培训讲师、演讲者，以及Lynda.com的作者。

封面介绍

本书封面上的动物是一头野猪（Sus scrofa）和它的孩子。野猪，也被称为山猪或者欧亚野猪，因为它原产于欧亚大陆、北非，以及大巽他群岛。由于人类的干预，它们是全球分布范围最广的哺乳动物之一。

野猪体型粗壮，四肢短小。它们的脖子粗短，头部较大，占据了身体长度的三分之一。成年野猪的体型和重量会根据环境因素而有所不同，比如获得食物和水的难易程度。不管体型大小，它们奔跑的时速可以达到25英里，跳跃的最大高度约55~59英尺。在冬季，它们的外套由粗糙的毛发构成，覆盖在棕色的表皮之上。这些鬃毛沿着野猪的背部生长，面部和四肢的毛发较短。

野猪拥有高度发达的嗅觉，目前它已经被德国用来检测毒品。它还拥有灵敏的听觉，但视力不佳和色盲，它无法识别30英尺以外站立的人类。

野猪是生活于母系氏族社会下的社会性动物。育种时间从11月持续到次年1月。雄性野猪在为交配做准备过程中身体会经历若干变化，其中包括进化出有助于防御敌人的皮下护甲。它们会长途旅行，途中吃得很少，以便可以找到一头雌性野猪。平均一窝会产4~6头幼崽。

O'Reilly封面上的大部分动物都濒临灭绝，它们对地球都非常重要。希望了解如何为它们提供帮助，可以前往 animals.oreilly.com。

封面图片来自Meyers Kleines Lexicon。

O'REILLY®

React学习手册

如果想学习如何使用React构建高效的用户界面，那么这本书就是为你准备的。本书作者将向你展示如何使用这款小型JavaScript库创建UI，并且可以在不重新加载页面的情况下，在数据驱动的大型网站上轻松显示数据变更。此外，你还将在阅读过程中学习函数式编程和最新的ECMAScript特性的使用。

由Facebook开发，并且被Netflix、Walmart和The New York Times等公司用于构建大部分Web界面，React很快就受到大众的青睐。通过这本实践指南你将学习如何构建React组件，并充分感受React在实际工作中的神奇魅力。

- 用JavaScript学习核心的函数式编程概念。
- 了解React在浏览器中的运行机制。
- 通过挂载和合成React组件来创建应用程序表现层。
- 使用组件树管理数据，减少调试应用程序的时间。
- 探索React组件生命周期，并用它来加载数据和提高UI性能。
- 为浏览器历史、书签和单页应用程序的其他功能选用路由解决方案。
- 学习如何在服务端构建React应用程序。

"本书介绍了React背后的函数式编程理念并使它易于理解，这对于任何希望构建现代JavaScript应用程序的人来说都是非常有用的。它是React应用程序常见问题和解决方案的绝佳指南。"

——Bonnie Eisenman
Twitter软件工程师，
*Learning React Native*的作者

Alex Banks是北加利福尼亚州软件开发培训公司Moon Highway的软件工程师、讲师和共同创始人。他曾先后为芝加哥马拉松、MSN和能源部开发过应用程序。

Eve Porcello是一名软件架构师、培训师和Moon Highway公司的共同创始人。她还曾在斯坦福大学、PayPal、eBay和Lynda.com讲授过JavaScript和Python。

JAVASCRIPT

ISBN 978-7-5198-142

O'Reilly Media, Inc.授权中国电力出版社出版

9 787519 814236

定价：78.00元