Web开发经典丛书

Pro React

React开发实战

使用React以组合方式构建复杂的前端应用程序

[美] Cássio de Sousa Antonio 著
杜伟 柴晓伟 涂曙光 译

Apress®

清华大学出版社

Web 开发经典丛书

React 开发实战

[美] Cássio de Sousa Antonio 著

杜伟 柴晓伟 涂曙光 译

清华大学出版社

北 京

Pro React

By Cássio de Sousa Antonio

EISBN：978-1-4842-1261-5

图书在版编目(CIP)数据

React开发实战/(美)卡西奥 • 德 • 宗萨 • 安东尼奥(Cássio de Sousa Antonio) 著；杜伟，柴晓伟，涂曙光 译. —北京：清华大学出版社，2017

(Web 开发经典丛书)

书名原文：Pro React

ISBN 978-7-302-46197-5

Ⅰ. ①R… Ⅱ. ①卡… ②杜… ③柴… ④涂… Ⅲ. ①移动终端—应用程序—程序设计 Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字(2017)第 019984 号

责任编辑：王　军　韩宏志
装帧设计：孔祥峰
责任校对：曹　阳
责任印制：宋　林

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：三河市溧源装订厂
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：19.5　　**字　　数**：451 千字
版　　次：2017 年 3 月第 1 版　　**印　　次**：2017 年 3 月第 1 次印刷
印　　数：1~4000
定　　价：58.00 元

产品编号：071425-01

第 1 章

React 入门

React 是由 Facebook 创建的一个开源项目。它提供了一种在 JavaScript 中构建用户界面的全新方式。自从它公开发布后，这个库迅速风靡全球，并且围绕着它培育了一个生机勃勃的社区。

通过阅读本书，你将掌握在项目中使用 React 所需的方方面面。因为 React 只关注 UI 界面的渲染，而不会对应用程序的其他模块所使用的技术做任何假设，所以本书同时也将介绍能匹配 React 模式的路由(Routing)和应用程序架构。

在本章中，我们将从一个较高的层面讲述一些主题，以便你能尽快开始创建应用程序。这些主题包括：

- React 的完整定义，以及优点概览
- 如何使用 JSX，这是一个在 React 中用来渲染 UI 的 JavaScript 语法扩展
- 如何创建包含属性和状态的 React 组件

1.1 开始学习之前

React 针对的是现代风格的 JavaScript 开发生态系统。为能亲自尝试本书中的代码示例，你需要安装 Node.js 和 npm。此外，还需要熟悉函数式 JavaScript 语法风格和这门语言最新的特性，如箭头函数(arrow functions)和类。

1.1.1 Node.js 和 npm

JavaScript 自诞生之日起就运行在浏览器上，但是通过 Node.js 的开源命令行工具，可以使 JavaScript 运行在你的本地计算机和服务器上。与 npm(Node Package Manager)一道，Node.js 已经成为一项在本地计算机上进行 JavaScript 应用程序开发的极有用工具，它使得开发人员可以创建脚本来运行任务(例如，复制和移动文件，或是启动一个本地开发服务器)，以及自动下载应用程序所依赖的组件。

如果你尚未安装 Node.js，现在就下载它的 Windows、Mac 或者 Linux 版本，将其安装到你的计算机上。下载地址为 https://nodejs.org/。

1.1.2 JavaScript ES6

JavaScript 是一门自诞生起多年一直在不断进化的语言。最近，JavaScript 技术社区已经认同了一组新的语言特性。有一些最新的浏览器已经能够支持这些特性，React 社区也广泛地使用了这些新的特性(例如，箭头函数、类、展开操作符)。React 同时鼓励在 JavaScript 代码中使用函数式编程模式，所以你也需要熟悉在 JavaScript 中函数和上下文是如何工作的，这样你才能了解 map、reduce 和 assign 等方法。如果你对这些细节感觉有些模糊，可参阅 Apress 网站(www.apress.com/)和本书的 GitHub 页面(http://pro-react.github.io/)上有关这些主题的在线附录说明。

1.2 定义 React

为清楚地说明 React 究竟为何物，我将对它做如下定义：

React 是一个使用 JavaScript 和 XML 技术(可选)构建可组合用户界面的引擎。

下面对 React 定义的每个部分再详加说明：

React 是一个引擎：React 的网站将它定义为一个库，但是我觉得使用“引擎”这个词更能体现出 React 的核心优势：用来实现响应式 UI 渲染的方式。React 的方式是将状态(在一个给定的时间点，所有用来定义应用程序的内部数据)从展现给用户的 UI 中分离出来。在 React 中，你可以声明如何将应用程序的状态表现为 DOM 的虚拟元素，然后自动更新 DOM 以反映出状态的变化。

“引擎”这个术语首先被 Justin Deal 用来描述 React，因为他觉得 React 渲染 UI 界面的方式和游戏引擎渲染的工作方式十分相似(https://zapier.com/engineering/react-js-tutorial-guide-gotchas/)。

创建可组合用户界面：减少创建和维护用户界面的复杂性一直是 React 的核心目标。React 拥抱了这样一种理念：将 UI“打散”成易于重用、扩展和维护的组件与自包含的、关注特定用途的构件(building blocks)。

使用 JavaScript 和 XML 技术(可选)：React 是一个可用于浏览器、服务器和移动设备之上的纯 JavaScript 库。如你将在本章中所见，React 有一种可选的语法来让你可以使用 XML 来描述 UI。一开始你可能会对这种语法感到有些陌生，但使用 XML 对于描述用户界面其实有诸多优点，包括：XML 是声明性的，很容易通过 XML 观察元素之间的关系，也很容易使 UI 的整体结构可视化。

1.3 React 的优点

市面上有许多的 JavaScript MVC 框架。Facebook 有什么理由还要创建 React？你有什么理由要使用 React？下面的三节内容将探索 React 的一些优点，从而回答这两个问题。

1.3.1 简单易学的响应式渲染

在 Web 开发的早期，那时还根本没有单页应用程序这个概念，用户每次在页面上进行一次交互(比如单击了一个按钮)，就算新的页面只和原有页面仅有一点不同，服务器都会将一整页新的页面发送回客户端浏览器。从用户的角度看，体验会非常糟糕，不过程序员倒是很容易规划出用户在某一时刻能看到哪些内容。

单页应用程序持续地从服务器获取新数据，并在用户进行交互时变换 DOM 上面的部分内容。在用户界面逐渐变得复杂时，应用程序也要实现越来越复杂的逻辑来检验应用程序的当前状态，并对 DOM 及时进行所需的修改。

许多 JavaScript 框架(特别是那些在 React 出现之前的框架)都使用了数据绑定技术，来处理这种日益增加的复杂性并保持用户界面和应用程序状态的同步，但是数据绑定在可维护性、可扩展性和性能上，都有一些缺陷。

响应式渲染比传统的数据绑定技术要更容易使用一些。它让我们用一种声明方式，来定义组件的外观和行为。当数据发生变化时，React 在概念上会重新渲染整个用户界面。

当然，每次在状态数据发生变化时就真的重新渲染整个用户界面，在性能上是不可接受的，React 使用了一种存在于内存中的轻量级 DOM 表示法，这被称为“虚拟 DOM”。

处理这种内存中的虚拟 DOM 要比处理真正的 DOM 更快、更有效。当因为用户的交互或者数据的获取而导致应用程序的状态发生改变时，React 快速地将 UI 的当前状态与期望的状态进行比较，然后计算出要对真实 DOM 所需进行的最小更改。这种工作方式使得 React 非常快、非常高效。即使在一个移动设备上，React 应用程序也能轻松地达到 60fps 的刷新率。

1.3.2 使用纯 JavaScript 进行面向组件开发

在一个 React 应用程序中，一切都由组件组成，组件就是应用程序中的自包含的、关注特定用途的基础构件。基于组件来开发应用程序使用了一种“分而治之”的途径，来避免在某个地方有太高的复杂性。由于组件可以组合起来，每个组件都可以尽量保持小巧，通过将更小的组件组合在一起，创建复杂的包含更多功能的组件就变得简单了。

不同于使用特定的模板语言或是传统 Web 应用程序 UI 中用到的 HTML 标记，React 组件由普通的 JavaScript 写就。使用普通 JavaScript 代码编写组件有一个很好的理由：模板限定了你可用来构造 UI 界面时能使用到的功能特性。React 则是使用一门功能完整的编程语言来渲染界面，这相对于使用模板具备很大的优势。

另外，通过使组件成为自包含的、并在相关视图逻辑中使用一个统一的标记，React 组件实现了关注点的分离。在Web开发的早期岁月，关注点的分离是通过在不同部分使用不同的语言来强制实现的：内容结构使用HTML，样式使用CSS，逻辑行为使用JavaScript。这种方法在问世之初工作得很好，因为那个时候Web页面还是静态呈现的。但是现在用户界面变得更有交互性、更复杂，显示逻辑和HTML标记不可避免地被绑定在一起；标记、样式和JavaScript之间的分离变成了仅是技术上的分离，而非关注点的分离。

React 假设显示逻辑和 HTML 标记是高度粘合的；它们同时用来实现 UI 的展现，并通过为每个关注点创建离散的、良好封装的、可重用的组件，来鼓励实现关注点的分离。

1.3.3 灵活的文档模型抽象表现

React 内置了自己的一个 UI 轻量级表现模型，以抽象出 UI 底层的文档模型。这种方式最值得一提的优点，就是不论在 Web 页面，还是在原生的 iOS 和 Android 界面上，它都可以使用同样的原则来渲染 HTML。这种抽象表现同时带来了下面这些特性：

- 事件在所有浏览器和设备上都会以一种统一、标准的方式，自动地使用代理，来达到行为的一致性。
- 为实现 SEO(搜索引擎优化)和更好的性能，React 组件可在服务器上被渲染。

1.4 创建你的第一个 React 应用程序

现在你已经知道了组件是 React UI 的基础构件，但是组件到底看起来是什么样子的？怎么样才能创建一个组件呢？简单来说，一个 React 组件就是一个带有 render 方法，并且返回组件 UI 描述的 JavaScript 类，如下所示：

```
class Hello extends React.Component {
  render() {
    return (
      <h1>Hello World</h1>
    )
  }
}
```

你也许已经注意到了JavaScript代码中间的HTML标记。之前曾经提到过，React有一种称为JSX的JavaScript语法扩展，可以让我们在代码中直接书写XML(以生成HTML)。

是否使用 JSX 是可选的，但是它已经成为一种被广泛接受的在 React 组件中定义 UI 的标准方案，由于 JSX 使用了具有丰富表达能力的声明式语法，以及这些内容最终会被转换成普通的 JavaScript 函数调用，所以这意味着 JSX 并没有影响 JavaScript 语言原本的语义。

我们会在下一章更详尽地介绍 JSX，但是现在要提醒你注意的是，React 需要一个额外的“转换”步骤(或者说是翻译步骤)，将 JSX 转换成 JavaScript 代码。

在现代的 JavaScript 开发生态系统中，有许多工具可处理这种需要额外转换步骤的情况。下面花一点时间来讨论如何为 React 项目搭建出一套流畅的开发流程。

1.4.1 React 开发流程

想当年，我们都是将所有 JavaScript 代码都写到一个文件里面，手工下载一两个 JavaScript 库，然后将自己的代码和第三方的库统统塞到一个页面上。当然，现在你仍然

可将 React 库的压缩 JavaScript 文件通过下载或复制粘贴的方法获得，然后立即用它来运行组件，并在运行时转换 JSX。只不过现在除了小的演示和原型之外，没人会在真正的项目里面这样做。

即使是创建一个最简单的应用程序，我们也希望可以通过开发流程满足如下需求：

- 编写 JSX 并随时将它转换成标准的 JavaScript 代码
- 使用模块化模式编写代码
- 管理依赖性
- 打包多个 JavaScript 文件并使用 source maps 进行调试

为满足上述需求，一个 React 项目的基础结构包含如下内容：

(1) 一个 source 文件夹，里面包含了你所编写的所有 JavaScript 模块。

(2) 一个 index.html 文件。在 React 应用程序中，这个 HTML 页面通常差不多是一个空页面。它仅用来加载应用程序的 JavaScript，并提供一个 div 元素(或者其他任何元素)来让 React 在该元素中渲染出应用程序的组件。

(3) 一个 package.json 文件。这个 package.json 是一个独立的 npm 清单文件，它包含了有关项目的诸多信息，例如名称、描述、作者的信息等。它让开发人员在其中指定依赖关系(通过指定依赖关系可以实现模块的自动下载和安装)并定义脚本任务。

(4) 一个模块打包或 build 工具，用来实现 JSX 转换和模块/依赖项打包。模块的应用使得 JavaScript 代码被组织到了多个文件中，每个模块都声明了它自己的依赖项。模块打包工具在编写完代码之后，就可以根据依赖关系，以正确顺序将所有代码文件打包在一起。有许多工具都可以实现这样的功能，包括 Grunt、Gulp 和 Brunch 等。在任何上述工具的文档中，你都能找到如何配合 React 使用的内容，不过通常来说，React 社区将 webpack 用作完成这项工作的首选工具。本质上，webpack 是一个模块打包工具，但它也可将源代码通过加载器进行转换和编译。

图 1-1 展现了上面所提到的文件和文件夹结构。

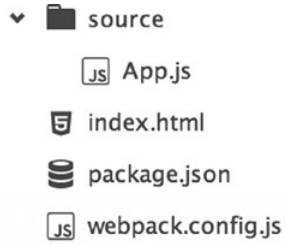

图 1-1 最简单 React 项目的文件和文件夹结构

提示：

在本书的在线资料中，你可以找到一个关于如何使用 webpack 设置一个 React 项目的完整附录说明。附录中涵盖了 webpack 的详细信息，并展现了如何设置高级选项，例如，热重载(Hot Reloading)React 组件。在线附录位于 Apress 网站(www.apress.com)和本书的 GitHub 页面(pro-react.github.io)上。

1. 快速起步

为专注于学习 React 库，本书提供了一个 React 应用程序模板包。你可从 apress.com 或者 GitHub 页面(https://github.com/pro-react/react-app-boilerplate)下载它。模板项目包含了立即开发 React 程序所需的所有基本文件和配置。下载后，你要做的就是安装依赖项并运行开发服务器来在浏览器中测试项目。要自动安装所有依赖项，可打开终端或命令提示符，运行 npm install。要运行开发服务器，输入 npm start。

你已经准备好了。现在可以直接跳过下面的主题，直接开始创建你的第一个 React 组件了。

2. 或者，自己动手

如果想要尝试自己动手，可通过下面的 5 个步骤来手工创建基础的项目结构。由于本书的主要关注点还是 React 库，所以我们现在不会深入讲解每个步骤中的细节，或是讲解那些可选的配置项，但你可通过在线附录阅读到有关它们的更多内容，或是查看 React 应用程序模板项目的源码文件。在线附录和模板项目都可从 Apress 网站(www.apress.com/)或是本书的 GitHub 页面(http://pro-react.github.io/)下载。

(1) 首先创建 source 文件夹(source 和 app 是常用的文件夹名字)。这个文件夹将只包含 JavaScript 模块。不需要被模块打包工具处理的静态资源(包括 index.html、图片文件，CSS 文件等)将放置到根文件夹中。

(2) 在项目的根文件夹中创建 index.html 文件。文件内容如代码清单 1-1 所示。

代码清单 1-1：加载打包后的 JavaScript 代码并为 React 组件渲染提供根 Div 的简单 HTML 页面

```
<!DOCTYPE html>
<html>
  <head>
    <title>First React Component</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="text/javascript" src="bundle.js"></script>
  </body>
</html>
```

(3) 通过在终端或者命令提示符上运行 npm init，然后按照提示进行操作来创建 package.json 文件。你将使用 npm 来进行依赖项管理(下载和安装所有需要的库)。你的项目的依赖项包括了 React、进行 JSX 转换的 Babel 编译器(loader 和 core)以及 webpack(包括 webpack 开发服务器)。按照代码清单 1-2 所示编辑你的 package.json 文件，然后运行 npm install。

代码清单 1-2：一个包含了依赖项的简单 package.json 文件

```
{
  "name": "your-app-name",
  "version": "X.X.X",
  "description": "Your app description",
  "author": "You",
  "devDependencies": {
    "babel-core": "^5.8.*",
    "babel-loader": "^5.3.*",
    "webpack": "^1.12.*",
    "webpack-dev-server": "^1.10.*"
  },
  "dependencies": {
    "react": "^0.13.*"
  }
}
```

(4) 接下来需要配置 webpack，它是你所选择的模块打包工具。代码清单 1-3 显示了配置文件。让我们来看一下配置文件里面的内容。首先，entry 键指向了应用程序主模块。

代码清单 1-3：webpack.config.js 文件

```
module.exports = {
  entry: [
    './source/App.js'
  ],
output: {
    path: __dirname,
    filename: "bundle.js"
  },
  module: {
    loaders: [{
      test: /\.jsx?$/,
      loader: 'babel'
    }]
  }
};
```

下一个键output告诉webpack将按正确顺序把所有模块打包后的单个JavaScript文件保存到什么地方。

最后，在 module 键的 loaders 区域，将所有.js 文件传送给 Babel，这个 JavaScript 编译器会将所有 JSX 转换成标准的 JavaScript 代码。记住，Babel 能做的远不止于此，它可以让你的代码使用最新的 JavaScript 语法，例如箭头函数和类。

(5) 现在是最后一个步骤。项目的结构已经完整。启动一个本地服务器(用于在浏览器中进行测试)的命令是“node_modules/.bin/webpack-dev-server”，不过为避免每次都要手工输入这么长一段命令，你可编辑步骤(3)中创建的 package.json 文件，将这个长命令

定义为一个任务，如代码清单 1-4 所示。

代码清单 1-4：将启动脚本添加到 package.json 文件中

```
{
  "name": "your-app-name",
  "version": "X.X.X",
  "description": "Your app description",
  "author": "You",
  "scripts": {
    "start": "node_modules/.bin/webpack-dev-server --progress"
  },
  "devDependencies": {
    "babel-core": "^5.8.*",
    "babel-loader": "^5.3.*",
    "webpack": "^1.12.*",
    "webpack-dev-server": "^1.10.*"
  },
  "dependencies": {
    "react": "^0.13.*"
  }
}
```

完成这个步骤后，下次想要运行本地的开发服务器时，只需要输入 npm start。

1.4.2 创建你的第一个组件

现在你已经有了一个基本的项目结构，它帮我们管理依赖项，提供了一个模块化系统，并为你转换 JSX。现在可以开始创建一个 Hello World 组件，并将它渲染到页面上了。你将继续使用本章之前展示过的组件的代码，不过在顶部加上了一个 import 声明，这个声明使 React 库被打包到最终要执行的 JavaScript 文件中。

```
import React from 'react';

class Hello extends React.Component {
  render() {
    return (
      <h1>Hello World</h1>
    );
  }
}
```

接下来，将使用 React.render 在页面上显示组件，如下面一行代码和图 1-2 所示：

```
React.render(<Hello />, document.getElementById('root'));
```

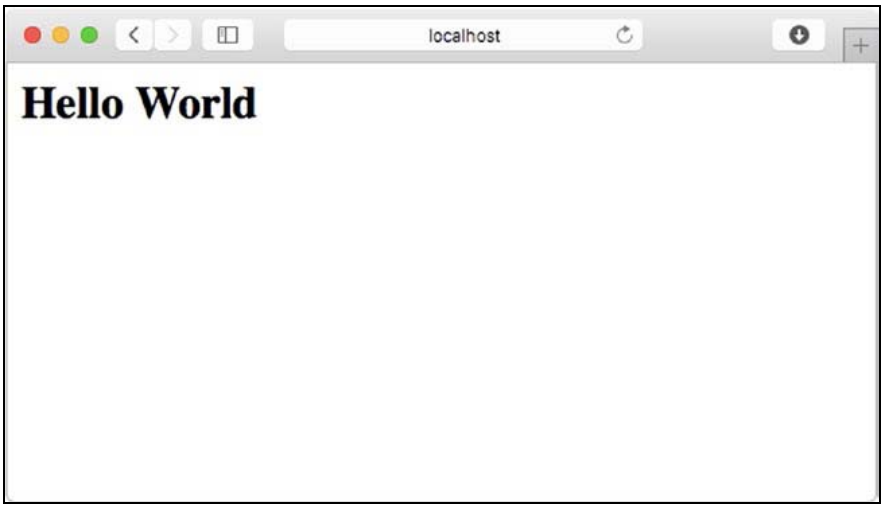

图 1-2 在浏览器中显示的第一个组件

提示：

可在一个页面的文档 body 中直接渲染组件的内容，但是通常来说，在一个子元素(通常是一个 div 元素)中进行渲染要更好一些。许多 JavaScript 库和浏览器扩展都将节点附加到文档 body 中，由于 React 需要完全掌控 DOM 树，如果 React 直接使用文档 body 而非一个子元素进行渲染，第三方库和浏览器扩展的这些行为就会导致不可预见的问题。

1.4.3 减少输入的字符数量

许多开发人员都经常使用一个技巧来减少输入的字符数量，那就是在模块的 import 中使用解构赋值来直接访问模块内部的函数和类。在上面的那个代码示例中，我们可使用下面的这个方法来避免需要输入完整的“React.Component”：

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

class Hello extends Component {
  render() {
    return (
      <h1>Hello World</h1>
    );
  }
}
```

在这里，示例中看起来也没节省太多需要输入的字符，但是在更大项目中，累积起来所节省的字符就比较可观了。

注意：

解构赋值是属于下个版本 JavaScript 规范的一个特性。在 React 中可以使用这种未来 JavaScript 版本才有的特性，在线附录 C 对此进行了详细描述。

1.4.4 动态值

在 JSX 中，位于花括号({})中的值会被当作一个 JavaScript 表达式进行求值，并被渲染到 HTML 标记中。例如，如果想要显示一个本地变量的值，可以这样做：

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

class Hello extends Component {
  render() {
    var place = "World";
    return (
      <h1>Hello {place}</h1>
    );
  }
}

React.render(<Hello />, document.getElementById("root"));
```

1.5 将组件组合起来

React 鼓励开发人员尽量创建简单且可重用的组件，然后通过将组件进行嵌套和组合来创建复杂的 UI。现在你已经了解了一个 React 组件的基本结构，下面介绍如何将组件组合在一起。

1.5.1 props

要让组件可以重用和组合，关键是对它们进行配置，React 提供了属性(简称为 props)来实现组件的配置。props 是 React 中的一种从父组件向子组件传输数据的机制。它们在子组件里面不能被修改；props 由父组件传输出去，也被父组件所“拥有”。

在 JSX 中，props 就像是 HTML 的标签特性那样。作为例子，下面创建一个简单的商店物品列表，它将由两个组件组成，父组件 GroceryList 和子组件 GroceryItem：

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

// Parent Component
class GroceryList extends Component {
  render() {
    return (
      <ul>
        <ListItem quantity="1" name="Bread" />
        <ListItem quantity="6" name="Eggs" />
        <ListItem quantity="2" name="Milk" />
      </ul>
    );
  }
}
```

```
// Child Component
class ListItem extends Component {
  render() {
    return (
      <li>
        {this.props.quantity}× {this.props.name}
      </li>
    );
  }
}

React.render(<GroceryList />,document.getElementById("root"));
```

除了使用命名 props 外，还可以通过 props.children 来引用位于前置标签和后置标签之间的内容：

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

// Parent Component
class GroceryList extends Component {
  render() {
    return (
      <ul>
        <ListItem quantity="1">Bread</ListItem>
        <ListItem quantity="6">Eggs</ListItem>
        <ListItem quantity="2">Milk</ListItem>
      </ul>
    );
  }
}

// Child Component
class ListItem extends Component {
  render() {
    return (
      <li>
        {this.props.quantity}× {this.props.children}
      </li>
    );
  }
}

React.render(<GroceryList />, document.getElementById('root'));
```

1.5.2 呈现看板应用

在本书中，针对每个主题，我们都会创建多个小组件和示例代码。我们还要创建一

个完整的应用程序：一个看板风格的项目管理工具。

在一个看板上，项目活动以卡片形式展现(如图 1-3 所示)。卡片根据它们的状态被归类到不同列表中，卡片可从一个列表移到另一个列表中，以表现一个功能从计划到实现的流程。

图 1-3　一个看板示例

网上有很多现成的看板风格项目管理应用程序。Trello.com 就是这种类型的一个著名网站，虽说你的项目通常没有复杂到要使用 Trello.com 的程度。你的项目最终看起来会如图 1-4 所示，看板应用的数据模型如代码清单 1-5 所示。

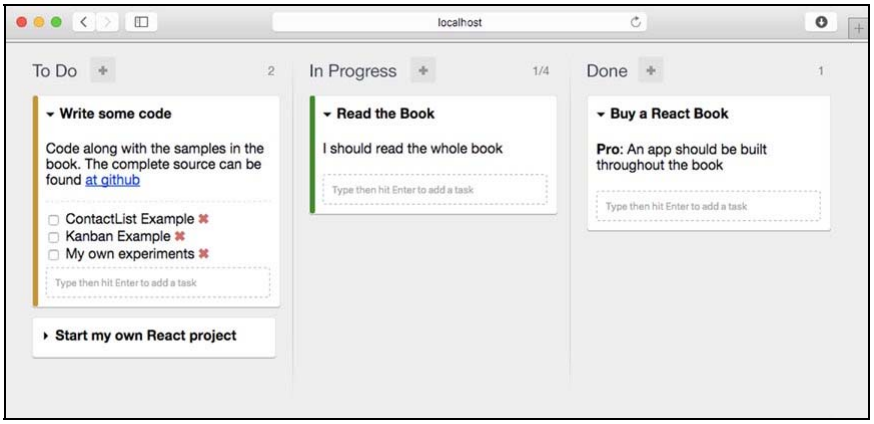

图 1-4　在后续章节中将要创建的看板应用

代码清单 1-5：看板应用的数据模型

```
[
  { id:1,
    title: "Card one title",
    description: "Card detailed description.",
```

```
    status: "todo",
    tasks: [
      {id: 1, name:"Task one", done:true},
      {id: 2, name:"Task two", done:false},
      {id: 3, name:"Task three", done:false}
    ]
  },
  { id:2,
    title: "Card Two title",
    description: "Card detailed description",
    status: "in-progress",
    tasks: []
  },
  { id:3,
    title: "Card Three title",
    description: "Card detailed description",
    status: "done",
    tasks: []
  },
];
```

1.5.3 定义组件的层次关系

第一件要做的事情，就是将页面打散成嵌套的组件。有 3 个要注意的事项。

(1) 记住每个组件都应当小巧且只关注单个功能。换句话说，一个组件只应当完成一件事。如果一个组件不断膨胀，就应当将它打散成多个更小的子组件。

(2) 分析项目的外观框架和布局，这些信息通常能提示我们如何决定组件的层次关系。

(3) 看一看应用程序的数据模型。界面和数据模型倾向于体现相同的信息架构，所以将 UI 界面分割成组件通常是很直观的。被打散的组件应该正好能体现数据模型中的一小块。

如果将上面说的概念应用到看板应用，将推导出如图 1-5 所示的组件层次关系。

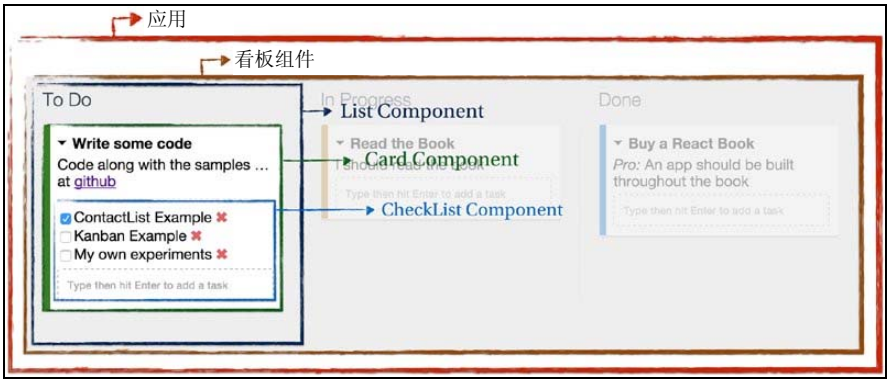

图 1-5 看板应用中的组件层次关系

1.5.4 props 的重要性

props 是实现组件组合的关键。props 是 React 中的一种从父组件向子组件传输数据的机制。它们在子组件中不能被修改；props 由父组件传输出去，也被父组件所“拥有”。

1.5.5 创建组件

确定了界面上的层级关系之后，就该创建组件了。创建组件有两条主要的途径：从上至下或是从下至上。也就是说，既可以首先创建在层级关系上更高的组件(例如 App 组件)，也可以首先创建在层次关系上更低的组件(例如 CheckList 组件)。为能清楚地理顺从父组件向下传递给子组件的 props 数据，以及子组件如何使用它们，将使用从上至下的方式来创建看板应用的组件。

另外，为让项目的文件结构更具有组织性，更容易维护和实现新功能，将为每个组件创建一个单独的 JavaScript 代码文件。

1. App 模块(app.js)

现在你要尽量让app.js文件保持简单。它只会包含数据并只会渲染出一个KanbanBoard组件。在看板应用的第一个迭代中，数据将被硬编码到一个本地变量中，但在后续章节，你会从一个API中获取数据，代码如代码清单 1-6 所示。

代码清单 1-6：一个简单的 app.js 文件

```
import React from 'react';
import KanbanBoard from './KanbanBoard';

let cardsList = [
  {
    id: 1,
    title: "Read the Book",
    description: "I should read the whole book",
    status: "in-progress",
    tasks: []
  },
  {
    id: 2,
    title: "Write some code",
    description: "Code along with the samples in the book",
    status: "todo",
    tasks: [
      {
        id: 1,
        name: "ContactList Example",
        done: true
      },
```

```
        {
          id: 2,
          name: "Kanban Example",
          done: false
        },
        {
          id: 3,
          name: "My own experiments",
          done: false
        }
      ]
    },
  ];

  React.render(<KanbanBoard cards={cardsList} />, document.getElementById
('root'));
```

2. KanbanBoard 组件(KanbanBoard.js)

KanbanBoard 组件将通过 props 接收数据，并负责筛选状态以创建 3 个 List 组件：“To Do”、“In Progress”和“Done”。代码如代码清单 1-7 所示。

注意：

在本章开头曾经说过，React 的组件是使用普通的 JavaScript 代码编写的。它们没有诸如 Mustache 的模板库所具有的分支辅助器之上的对循环的支持，但是由于可以直接使用一门功能完备的编程语言，所以这并不算是一个坏消息。在下面的组件中，会对 cards 数组使用 filter 和 map 函数来对数据进行处理。

代码清单 1-7：KanbanBoard 组件

```
import React, { Component } from 'react';
import List from './List';

class KanbanBoard extends Component {
render(){
   return (
     <div className="app">

       <List id='todo' title="To Do" cards={
         this.props.cards.filter((card) =>card.status === "todo")
       } />

       <List id='in-progress' title="In Progress" cards={
         this.props.cards.filter((card) =>card.status === "in-progress")
       } />

       <List id='done' title='Done' cards={
         this.props.cards.filter((card) =>card.status === "done")
```

```
        } />

      </div>
    );
  }
}

export default KanbanBoard;
```

3. List 组件(List.js)

List 组件只会显示出列表名，然后在组件中渲染出所有 Card 组件。注意，你将对通过 props 接收到的 cards 数组进行映射，然后将映射出的标题和描述等单独信息，以 props 方式传递给 Card 子组件。代码如代码清单 1-8 所示。

代码清单 1-8：List 组件

```
import React, { Component } from 'react';
import Card from './Card';

class List extends Component {
  render() {
    var cards = this.props.cards.map((card) => {
      return<Card  id={card.id}
                   title={card.title}
                   description={card.description}
                   tasks={card.tasks} />
    });

    return (
      <div className="list">
        <h1>{this.props.title}</h1>
        {cards}
      </div>
    );
  }
}

export default List;
```

4. Card 组件(Card.js)

Card 组件是与用户交互最多的组件，它用来显示一个卡片。每个卡片都有一个标题、一个说明和一个代码清单，如图 1-6 所示，代码如代码清单 1-9 所示。

代码清单 1-9：Card 组件

```
import React, { Component } from 'react';
import CheckList from './CheckList';
```

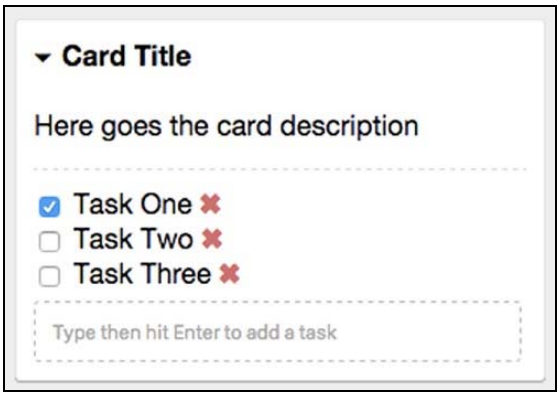

图 1-6 看板应用中的卡片

```
class Card extends Component {
  render() {
    return (
      <div className="card">
        <divclassName="card__title">{this.props.title}</div>
        <div className="card__details">
          {this.props.description}
          <CheckList cardId={this.props.id} tasks={this.props.tasks} />
        </div>
      </div>
    );
  }
}

export default Card;
```

注意 Card 组件中 className 特性的用法。既然 JSX 就是 JavaScript，不鼓励使用类似 class 这样的标识符作为 XML 特性，所以在这里使用了 className。下一章将对该主题再详加讨论。

5. Checklist 组件(CheckList.js)

最后，Checklist 组件用来显示卡片的底部区域。注意，仍然没有一个用来创建新任务的表单，将在之后再创建新建任务的界面。代码如代码清单 1-10 所示。

代码清单 1-10：Checklist 组件

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

class CheckList extends Component {
  render() {
    let tasks = this.props.tasks.map((task) => (
      <li className="checklist__task">
        <input type="checkbox" defaultChecked={task.done} />
        {task.name}
```

```
        <a href="#" className="checklist__task--remove" />
      </li>
    ));

    return (
      <div className="checklist">
        <ul>{tasks}</ul>
      </div>
    );
  }
}

export default CheckList;
```

6. 界面装饰

React 组件的任务已完成。为让界面美观一点，现在编写一个 CSS 来装饰一下界面(如代码清单 1-11 所示)。别忘了创建一个 HTML 文件来加载 JavaScript 和 CSS 文件，并为 React 放置一个 div 元素来渲染所有组件(代码清单 1-12 展示了一个示例)。

代码清单 1-11：CSS 文件

```
*{
  box-sizing: border-box;
}

html,body,#app {
  height:100%;
  margin: 0;
  padding: 0;
}

body {
  background: #eee;
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
}

h1{
  font-weight: 200;
  color: #3b414c;
  font-size: 20px;
}

ul {
  list-style-type: none;
  padding: 0;
  margin: 0;
}
```

```
.app {
  white-space: nowrap;
  height:100%;
}

.list {
  position: relative;
  display: inline-block;
  vertical-align: top;
  white-space: normal;
  height: 100%;
  width: 33%;
  padding: 0 20px;
  overflow: auto;
}

.list:not(:last-child):after{
  content: "";
  position: absolute;
  top: 0;
  right: 0;
  width: 1px;
  height: 99%;
  background: linear-gradient(to bottom, #eee 0%, #ccc 50%, #eee 100%) fixed;
}

.card {
  position: relative;
  z-index: 1;
  background: #fff;
  width: 100%;
  padding: 10px 10px10px 15px;
  margin: 0 0 10px 0;
  overflow: auto;
  border: 1px solid #e5e5df;
  border-radius: 3px;
  box-shadow: 0 1px 0 rgba(0, 0, 0, 0.25);
}

.card__title {
  font-weight: bold;
  border-bottom: solid 5px transparent;
}

.card__title:before {
  display: inline-block;
  width: 1em;
  content: '►';
}
```

```
.card__title--is-open:before {
  content: '▼';
}

.checklist__task:first-child {
  margin-top: 10px;
  padding-top: 10px;
  border-top: dashed 1px #ddd;
}

.checklist__task--remove:after{
  display: inline-block;
  color: #d66;
  content: "+";
}
```

代码清单 1-12：HTML 文件

```
<!DOCTYPE html>
<html>
<head>
  <title>Kanban App</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <div id="root"></div>
  <script type="text/javascript" src="bundle.js"></script>
</body>
</html>
```

如果你在自己的开发环境中一直跟做了上面所有的步骤，那么应该会看到如图 1-7 所示的页面。

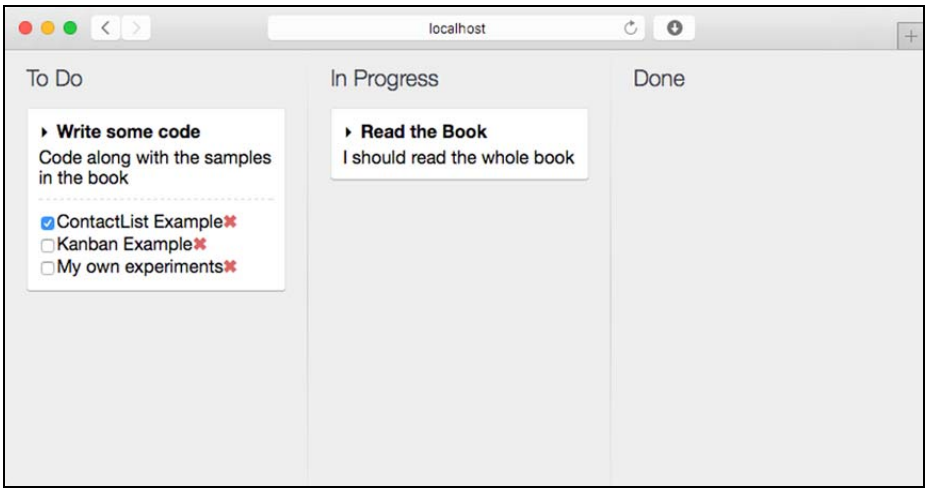

图 1-7　组合的组件界面

1.6 介绍 state

至此，你已经看到了props的用法，它被组件所接收，且是不可变的。这种不可变的特点导致了组件是静态的。如果想要添加行为和交互，组件就需要有可变的数据来体现它的状态(state)。React的组件可以在this.state里面保存可变的数据。注意，this.state对于组件来说是私有的，可通过调用this.setState()函数来修改它的值。

这时就会引出 React 组件的一个重要特性：当 state 被修改时，组件会触发响应式渲染，组件自身及其子组件都会被重新渲染。如前所述，由于 React 使用了一个虚拟的 DOM，这种重新渲染的操作执行得非常快。

看板应用：可切换式卡片

为演示组件中 state 的用法，让我们为看板应用添加一个新功能。你将要使卡片变得可切换。用户可以显示或隐藏卡片的详情。

你可在任何时候为组件设置一个新的 state，但是如果你想要组件有一个初始的 state，你可在类的构造函数里进行设置。现在，Card 组件还没有一个构造函数，只有一个 render 方法。让我们添加一个构造函数，并在组件的 state 中定义一个新的名为 showDetails 的键(key)。代码如代码清单 1-13 所示(为简洁起见，代码清单中省略了 import/export 声明和 render 方法的内容)。

代码清单 1-13：可切换式卡片

```
class Card extends Component {
  constructor() {
    super(...arguments);
    this.state = {
      showDetails: false
    };
  }

  render() {
    return ( ... );
  }
}
```

接下来，可以修改 render 方法中的 JSX 代码，使它只有在 state 的 showDetails 属性为 true 时才渲染卡片的详情。为实现这个功能，定义一个名为 cardDetails 的局部变量，然后只有在当前状态 showDetails 为 true 时才给它赋予真正的数据。在 return 语句中，仅返回这个局部变量的值即可(如果 showDetails 为 false，则变量的值为空)。代码如代码清单 1-14 所示。

代码清单 1-14：Card 组件的 render 方法

```
render() {
  let cardDetails;
  if (this.state.showDetails) {
    cardDetails = (
      <div className="card__details">
        {this.props.description}
        <CheckList cardId={this.props.id} tasks={this.props.tasks} />
      </div>
    );
  };

  return (
    <div className="card">
      <div className="card__title">{this.props.title}</div>
      {cardDetails}
    </div>
  );
}
```

最后一步，让我们添加一个 click 事件处理程序来修改内部 state。使用 JavaScript 的!(非)操作符来切换 showDetails 布尔属性的值(如果它的当前值为 true，应用!操作符后值将是 false，反之亦然)，如代码清单 1-15 所示。

代码清单 1-15：click 事件处理程序

```
render() {
  let cardDetails;
    if (this.state.showDetails) {
      cardDetails = (
        <div className="card__details">
          {this.props.description}
          <CheckList cardId={this.props.id} tasks={this.props.tasks} />
        </div>
      );
    };

  return (
    <div className="card">
      <div className="card__title" onClick={
        ()=>this.setState({showDetails: !this.state.showDetails})
      }>{this.props.title}</div>
      {cardDetails}
    </div>
  );
}
```

当在浏览器上运行这个示例时，卡片上的所有详情默认都会处于关闭状态，通过单

击卡片标题可显示出详情(如图 1-8 所示)。

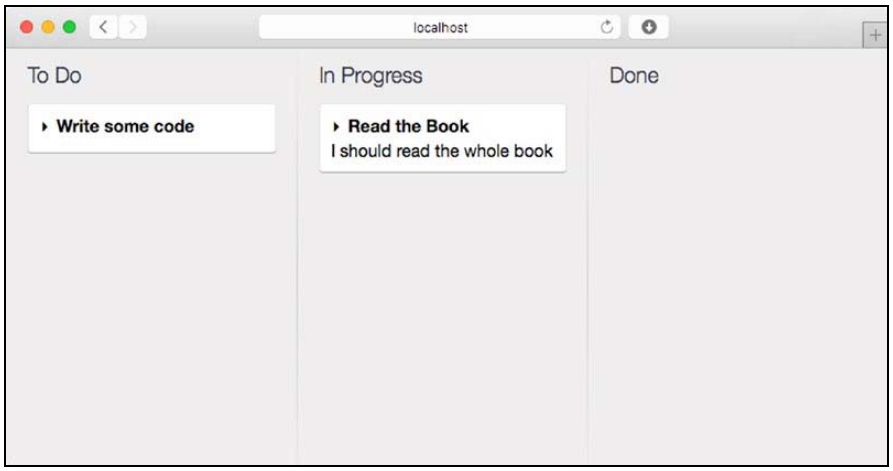

图 1-8 可切换的看板卡片

1.7 本章小结

本章解释了 React 的含义以及它给 Web 开发领域带来的好处(主要好处就是提供了一种高性能、声明式途径，能以组件方式构建应用程序的用户界面)。你还创建了你的第一个组件，并见证了 React 组件的所有基本概念：render 方法、JSX、props 和 state。