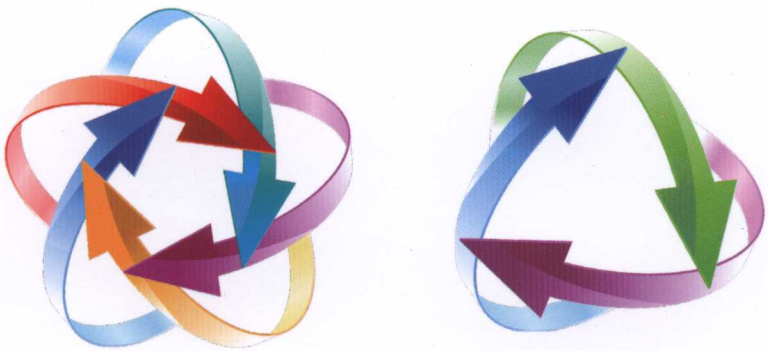
HZ BOOKS
华章IT

资深架构师十余年工程经验结晶

由浅入深介绍如何用React和Redux构建前端项目，产出高质量易维护代码

實戰



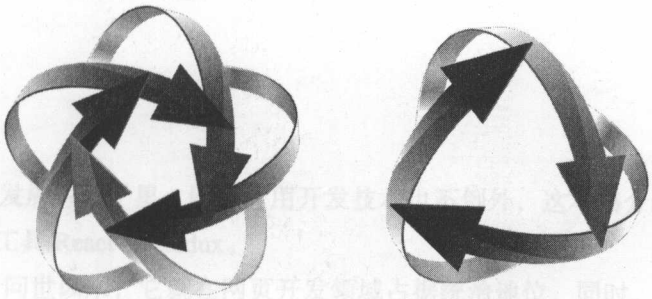
程墨 编著

Dissecting React & Redux

深入浅出 React和Redux

机械工业出版社
China Machine Press

實戰



Dissecting React & Redux

深入浅出 React和Redux

程墨 编著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

深入浅出 React 和 Redux / 程墨编著 . —北京：机械工业出版社，2017.4
（实战）

ISBN 978-7-111-56563-5

I. 深… II. 程… III. JAVA 语言 – 程序设计 IV. TP312.8

中国版本图书馆 CIP 数据核字（2017）第 062853 号

深入浅出 React 和 Redux

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：吴 怡　　责任校对：殷 虹
印　　刷：三河市宏图印务有限公司　　版　　次：2017 年 4 月第 1 版第 1 次印刷
开　　本：186mm × 240mm 1/16　　印　　张：17
书　　号：ISBN 978-7-111-56563-5　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066　　投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259　　读者信箱：hzit@hzbook.com

Preface 前　言

互联网技术发展一日千里，网页应用开发技术也不例外，这本书介绍的是在这一领域备受瞩目的两个工具 React 和 Redux。

自从 jQuery 问世以来，它就在网页开发领域占据统治地位，同时，还有许多 MVC 框架如雨后春笋般出现。但是业界也普遍发现，jQuery 和各种 MVC 框架在开发大型复杂应用时，依然面临很多难以克服的困难。

当 2014 年 Facebook 推出 React 时，给整个业界带来全新的看待网页应用开发的方式，和 React 一同问世的 Flux，也克服传统 MVC 框架的很多弊病。技术在不断发展，在 2015 年，Flux 的一个变体 Redux 出现，进一步优化了 Flux 的功能。

React 和 Redux 的结合，让网页开发的方式耳目一新，写这本书的初衷，是为了让国内读者能够一睹 React 和 Redux 的内在原理并深入实践。

在这里深入介绍 React 和 Redux，绝不是贬抑其他前端框架，事实上，开发者应该接触不同的开发模式，才能融会贯通，对技术有一个全面的认识，若要掌握某种技术，就要深入学习，这就是本书的目的。对 React 和 Redux 的了解不要只是停留在能用的表面功夫，重要的是理解内在的原理。

本书的内容

希望读者把阅读这本书的过程当做一个旅程，由浅入深地了解 React 和 Redux，如果你对 React 和 Redux 技术已经有一些了解，可以直接跳到感兴趣的章节。本书包括 12 章，如下所示。

第 1 章，React 新的前端思维方式。实际操作快速创建一个 React 应用，介绍和传统网页开发相比 React 应用开发的独特方式。

第 2 章，设计高质量的 React 组件。React 提倡基于组件的设计，这一章通过开发一个

ControlPanel 组件的实践，介绍了开发高质量 React 组件的原则，详细介绍 React 组件的生命周期和数据管理方式。

第 3 章，从 Flux 到 Redux。通过 Flux 介绍了单向数据流的框架模式，由此引出比 Flux 更优秀的 Redux 框架，通过用不同框架实现 ControlPanel 应用可以比较框架的优劣。

第 4 章，模块化 React 和 Redux 应用。这一章通过开发一个 Todo 应用介绍将 React 和 Redux 结合的方法。

第 5 章，React 组件的性能优化。通过对 Todo 应用的性能优化，介绍提高 React 组件渲染性的方法，以及提高从 Store 获取数据性能的方法。

第 6 章，React 高级组件。介绍高阶组件和“以函数为子组件”的模式。

第 7 章，Redux 和服务器通信。通过开发一个天气信息应用的实践，介绍应如何在 React 和 Redux 的环境中实现与服务器的通信。

第 8 章，单元测试。介绍针对 React 和 Redux 的单元测试技巧。

第 9 章，扩展 Redux。介绍创建中间件和 Store Enhancer 的技巧。

第 10 章，动画。介绍在 React 中通过 ReactTransitionGroup 和 React-Motion 库实现动画的技巧。

第 11 章，多页面应用。介绍如何创建多页面路由，以及为了提高网页装载性能的代码分片技巧。

第 12 章，同构。创建让 React 组件能够在服务器端和浏览器端渲染的技术。

本书的目标读者

阅读这本书只需要一些基本的 JavaScript、HTML 和 CSS 知识，了解网页应用的工作原理，就足够具备体验 React 和 Redux 这种全新的开发方式。

如果你熟悉传统的 jQuery 应用开发，那么通过阅读本书会让你发现不一样的应用构建模式；如果你之前学习过 Angular.js 或者 Vue.js，那么对理解 React 和 Redux 的工作机理很有帮助，同时有机会体验同样一种思想的不同实现之道。

即使你对 React 和 Redux 已经有了一定认识，相信阅读此书也不会让你觉得是浪费时间，因为书中不只是介绍“如何去做”，更多地还解释了“为什么这么做”，相信阅读此书会让你对 React 和 Redux 会有更多更深的认识。

源代码

本书每章都附带大量的实际代码例子，因为篇幅所限，在书中不可能包含所有代码，

读者可以在 Github（网址 https://github.com/mocheng/react-and-redux）上找到所有代码，代码按照所属章节内容组织。

如果读者发现代码或者书中的错误，可以直接在上面网址对应的代码库中提交问题，请不吝斧正。

致谢

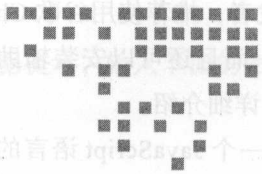
首先要感谢我的家人，没有他们的帮助和理解，这本书不可能完成。

感谢 Hulu 公司，本书中的很多内容都是和 Hulu 的研发团队协同合作中得到的体会。

感谢机械工业出版社的吴怡编辑，因为她的鼓励和帮助，这本书才得以问世。

最后要感谢 React 和 Redux 社区，因为千千万万开发者以开放的心态贡献代码和积极讨论，前端开发技术才获得巨大的飞跃，这个世界才变得更加美好。

目　录 *Contents*



第 1 章 Chapter 1

React 新的前端思维方式

我们先来直观认识 React，对任何一种工具，只有使用才能够熟练掌握，React 也不例外。通过对 React 快速入手，我们会解析 React 的工作原理，并通过与功能相同的 jQuery 程序对比，从而看出 React 的特点。

在这一章中，我们会介绍：

- 如何初始化一个 React 项目；
- 如何创建一个 React 组件；
- React 的工作方式。

让我们开始旅程吧！

1.1 初始化一个 React 项目

为了开发 React 应用，你的电脑是运行微软 Windows 操作系统，还是苹果 Mac，或者是 Linux，并不重要，只需要保证具备以下条件：

- 安装了浏览器，如果是 Windows 操作系统，请保证微软 IE 浏览器版本不低于 8.0 版，因为 React 不支持比 IE 8 更低版本的浏览器；
- 有一个命令行环境，在 Windows 操作系统中有命令行界面，在苹果 Mac 电脑中可以使用 Terminal 应用，对于 Linux 用户，命令行环境我想不用过多解释；
- 一个你最喜欢的代码编辑器，用于编辑 React 应用的代码，本书内容注重实践，只有实际编码才能有深入体会。

作为开发者，推荐使用谷歌 Chrome 浏览器，因为 Chrome 浏览器自带的开发辅助工具非常友好，而且还可以安装辅助 React 和 Redux 的扩展工具，具体的开发工具在第 4 章 4.2 节中有详细介绍。

React 是一个 JavaScript 语言的工具库，在这个 JavaScript 工具铺天盖地的时代，没有意外，你需要安装 Node.js，React 本身并不依赖于 Node.js，但是我们开发中用到的诸多工具需要 Node.js 的支持。

在 Node.js 的官网 (https://nodejs.org/) 可以找到合适的安装方式，安装 Node.js 的同时也就安装了 npm，npm 是 Node.js 的安装包管理工具，因为我们不可能自己开发所有功能，会大量使用现有的安装包，就需要 npm 的帮助。

1.1.1 create-react-app 工具

React 技术依赖于一个很庞大的技术栈，比如，转译 JavaScript 代码需要使用 Babel，模块打包工具又要使用 Webpack，定制 build 过程需要 grunt 或者 gulp……这些技术栈都需要各自的配置文件，还没有开始写一行 React 相关代码，开发人员就已经被各种技术名词淹没。

针对这种情况，React 的创建者 Facebook 提供了一个快速开发 React 应用的工具，名叫 create-react-app，这个工具的目的是将开发人员从配置工作中解脱出来，无需过早关注这些技术栈细节，通过创建一个已经完成基本配置的应用，让开发者快速开始 React 应用的开发。

本书中所有应用实例都由 create-react-app 创建，我们用这种最简单的方式创建可运行的应用，必要的时候才会介绍底层技术栈的细节，毕竟，没有什么比一个能运行的应用更加增强开发者的信心。

create-react-app 是一个通过 npm 发布的安装包，在确认 Node.js 和 npm 安装好之后，命令行中执行下面的命令安装 create-react-app：

```
npm install --global create-react-app
```

安装过程结束之后，你的电脑中就会有 create-react-app 这样一个可以执行的命令，这个命令会在当前目录下创建指定参数名的应用目录。

我们在命令行中执行下面的命令：

```
create-react-app first_react_app
```

这个命令会在当前目录下创建一个名为 first_react_app 的目录，在这个目录中会自动添加一个应用的框架，随后我们只需要在这个框架的基础上修改文件就可以开发 React

应用，避免了大量的手工配置工作：

在 create-react-app 命令一大段文字输出之后，根据提示，输入下面的命令：

```
cd first_react_app
npm start
```

这个命令会启动一个开发模式的服务器，同时也会让你的浏览器自动打开了一个网页，指向本机地址 http://localhost:3000/，显示界面如图 1-1 所示。


注意　本书中的截图是根据 create-react-app 1.0.0 版本所得，其他版本产生的页面可能略有不同。

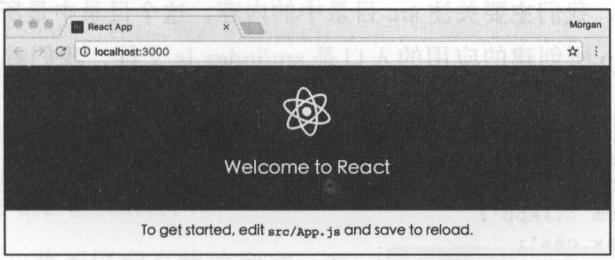


图 1-1　由 create-react-app 创造的 React 应用界面

恭喜你，你的第一个 React 应用诞生了！

接下来，我们会用 React 开发一个简单的功能，让我们继续吧。

1.2　增加一个新的 React 组件

React 的首要思想是通过组件（Component）来开发应用。所谓组件，简单说，指的是能完成某个特定功能的独立的、可重用的代码。

基于组件的应用开发是广泛使用的软件开发模式，用分而治之的方法，把一个大的应用分解成若干小的组件，每个组件只关注于某个小范围的特定功能，但是把组件组合起来，就能够构成一个功能庞大的应用。如果分解功能的过程足够巧妙，那么每个组件可以在不同场景下重用，那样不光可以构建庞大的应用，还可以构建出灵活的应用。打个比方，每个组件是一块砖，而一个应用是一座楼，想要一次锻造就创建一座楼是不现实的。实际上，总是先锻造很多砖，通过排列组合这些砖，才能构建伟大的建筑。

React 非常适合构建用户交互组件，让我们从创建一个 React 组件开始。

学习任何一门语言或者任何一门框架，往往是从写 Hello World 程序开始，不过只是

展示一句 Hello World 并不足以体现 React 的神奇能力，所以，我们要做一个不那么简单的组件，为了体现 React 对交互功能的支持，我们做一个显示点击次数的组件。

我们先看一看 create-react-app 给我们自动产生的代码，在 first-react-app 目录下包含如下文件和目录：

```
src/
public/
README.md

package.json

node_modules/
```

在开发过程中，我们主要关注 src 目录中的内容，这个目录中是所有的源代码。

create-react-app 所创建的应用的入口是 src/index.js 文件，我们看看中间的内容，代码如下：

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './index.css';

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

这个应用所做的事情，只是渲染一个名叫 App 的组件，App 组件在同目录下的 App.js 文件中定义，渲染出来的效果就是在图 1-1 中看到的界面。

我们要定义一个新的能够计算点击数组件，名叫 ClickCounter，所以我们修改 index.js 文件如下：

```
import React from 'react';
import ReactDOM from 'react-dom';
import ClickCounter from './ClickCounter';
import './index.css';

ReactDOM.render(
  <ClickCounter />,
  document.getElementById('root')
);
```

我们接下来会介绍代码的含义。现在我们先来看看如何添加一个新组件，在 src 目录下增加一个新的代码文件 ClickCounter.js，代码如下：

```
import React, { Component } from 'react';
class ClickCounter extends Component {
```

```
  constructor(props) {
    super(props);
    this.onClickButton = this.onClickButton.bind(this);
    this.state = {count: 0};
  }

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

  render() {
    return (
      <div>
        <button onClick={this.onClickButton}>Click Me</button>
        <div>
        Click Count: {this.state.count}
        </div>
      </div>
    );
  }
}
export default ClickCounter;
```

如果你是从上一节不停顿直接读到这一节，而且没有关闭命令行中的 npm start 命令，当你保存完这个文件之后，不需要主动做刷新网页的动作，就会发现网页中的内容已经发生改变，如图 1-2 所示。

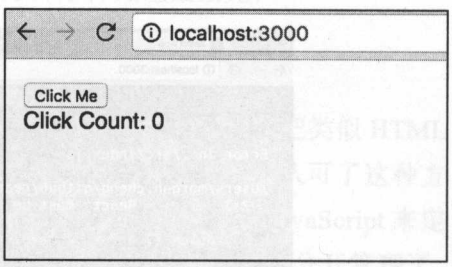


图 1-2　ClickCounter 组件界面效果

去点击那个“Click Me”按钮，可以看到“Click Count”后面的数字会随之增加，每点击一次加一。

恭喜你，现在你已经构建了一个有交互性的组件！

现在让我们来逐步详细解释代码中各部分的要义。

在 index.js 文件中，使用 import 导入了 ClickCounter 组件，代替了之前的 App 组件。

```
import ClickCounter from './ClickCounter';
```

import 是 ES6（EcmaScript 6）语法中导入文件模块的方式，ES6 语法是一个大集合，大部分功能都被最新浏览器支持。不过这个 import 方法却不在广泛支持之列，这没有关系，ES6 语法的 JavaScript 代码会被 webpack 和 babel 转译成所有浏览器都支持的 ES5 语法，而这一切都无需开发人员做配置，create-react-app 已经替我们完成了这些工作。

在 ClickCounter.js 文件的第一行，我们从 react 库中引入了 React 和 Component，如下所示：

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

Component 作为所有组件的基类，提供了很多组件共有的功能，下面这行代码，使用的是 ES6 语法来创建一个叫 ClickCounter 的组件类，ClickCounter 的父类就是 Component：

```
class ClickCounter extends Component {
```

在 React 出现之初，使用的是 React.createClass 方式来创造组件类，这种方法已经被废弃了，但是在互联网上依然存在大量的文章基于 React.createClass 来讲解 React，这些文章中依然有很多真知灼见的部分，但是读者要意识到，使用 React.createClass 是一种过时的方法。在本书中，我们只使用 ES6 的语法来构建组件类。

细心的读者会发现，虽然我们导入的 Component 类在 ClickCounter 组件定义中使用了，可是导入的 React 却没有被使用，难道在这里引入 React 没有必要吗？

事实上，引入 React 非常必要，你可以尝试删掉第一行中的 React，在网页中立刻会出现错误信息，如图 1-3 所示。

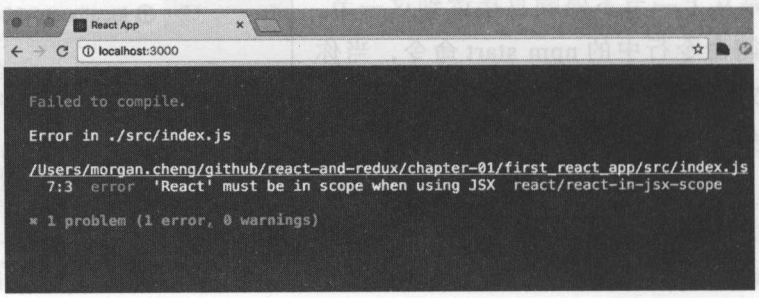


图 1-3 缺失 React 的错误

这个错误信息的含义是："在使用 JSX 的范围内必须要有 React。"

也就是说，在使用 JSX 的代码文件中，即使代码中并没有直接使用 React，也一定要导入这个 React，这是因为 JSX 最终会被转译成依赖于 React 的表达式。

接下来，我们就要认识什么是 JSX。

1.2.1 JSX

所谓 JSX，是 JavaScript 的语法扩展（eXtension），让我们在 JavaScript 中可以编写像 HTML 一样的代码。在 ClickCounter.js 的 render 函数中，就出现了类似这样的 HTML 代码，在 index.js 中，ReactDOM.render 的第一个参数 <App /> 也是一段 JSX 代码。

JSX 中的这几段代码看起来和 HTML 几乎一模一样，都可以使用 <button> 之类的元素，所以只要熟悉 HTML，学习 JSX 完全不成问题，但是，我们一定要明白两者的不同之处。

首先，在 JSX 中使用的“元素”不局限于 HTML 中的元素，可以是任何一个 React 组件，在 App.js 中可以看到，我们创建的 ClickCounter 组件被直接应用在 JSX 中，使用方法和其他元素一样，这一点是传统的 HTML 做不到的。

React 判断一个元素是 HTML 元素还是 React 组件的原则就是看第一个字母是否大写，如果在 JSX 中我们不用 ClickCounter 而是用 clickCounter，那就得不到我们想要的结果。

其次，在 JSX 中可以通过 onClick 这样的方式给一个元素添加一个事件处理函数，当然，在 HTML 中也可以用 onclick（注意和 onClick 拼写有区别），但在 HTML 中直接书写 onclick 一直就是为人诟病的写法，网页应用开发界一直倡导的是用 jQuery 的方法添加事件处理函数，直接写 onclick 会带来代码混乱的问题。

这就带来一个问题，既然长期以来一直不倡导在 HTML 中使用 onclick，为什么在 React 的 JSX 中我们却要使用 onClick 这样的方式来添加事件处理函数呢？

1.2.2　JSX 是进步还是倒退

在 React 出现之初，很多人对 React 这样的设计非常反感，因为 React 把类似 HTML 的标记语言和 JavaScript 混在一起了，但是，随着时间的推移，业界逐渐认可了这种方式，因为大家都发现，以前用 HTML 来代表内容，用 CSS 代表样式，用 JavaScript 来定义交互行为，这三种语言分在三种不同的文件里面，实际上是把不同技术分开管理了，而不是逻辑上的“分而治之”。

根据做同一件事的代码应该有高耦合性的设计原则，既然我们要实现一个 ClickCounter，那为什么不把实现这个功能的所有代码集中在一个文件里呢？

这点对于初学者可能有点难以接受，但是相信当你读完此书后，观点会随之改变。

那么，在 JSX 中使用 onClick 添加事件处理函数，是否代表网页应用开发兜了一个大圈，最终回到了起点了呢？

不是这样，JSX 的 onClick 事件处理方式和 HTML 的 onclick 有很大不同。

即使现在，我们还是要说在 HTML 中直接使用 onclick 很不专业，原因如下：

- onclick 添加的事件处理函数是在全局环境下执行的，这污染了全局环境，很容易产生意料不到的后果；
- 给很多 DOM 元素添加 onclick 事件，可能会影响网页的性能，毕竟，网页需要

的事件处理函数越多，性能就会越低；

- 对于使用 onclick 的 DOM 元素，如果要动态地从 DOM 树中删掉的话，需要把对应的时间处理器注销，假如忘了注销，就可能造成内存泄露，这样的 bug 很难被发现。

上面说的这些问题，在 JSX 中都不存在。

首先，onClick 挂载的每个函数，都可以控制在组件范围内，不会污染全局空间。

我们在 JSX 中看到一个组件使用了 onClick，但并没有产生直接使用 onclick（注意是 onclick 不是 onClick）的 HTML，而是使用了事件委托 (event delegation) 的方式处理点击事件，无论有多少个 onClick 出现，其实最后都只在 DOM 树上添加了一个事件处理函数，挂在最顶层的 DOM 节点上。所有的点击事件都被这个事件处理函数捕获，然后根据具体组件分配给特定函数，使用事件委托的性能当然要比为每个 onClick 都挂载一个事件处理函数要高。

因为 React 控制了组件的生命周期，在 unmount 的时候自然能够清除相关的所有事件处理函数，内存泄露也不再是一个问题。

除了在组件中定义交互行为，我们还可以在 React 组件中定义样式，我们可以修改 ClickCounter.js 中的 render 函数，代码如下：

```
render() {
  const counterStyle = {
    margin: '16px'
  }
  return (
    <div style={counterStyle}>
      <button onClick={this.onClickButton}>Click Me</button>
      <div>
      Click Count: <span id=" clickCount" >{this.state.count}</span>
      </div>
    </div>
  );
}
```

我们在 JavaScript 代码中定义一个 counterStyle 对象，然后在 JSX 中赋值给顶层 div 的 style 属性，可以看到网页中这个部分的 margin 真的变大了。

你看，React 的组件可以把 JavaScript、HTML 和 CSS 的功能在一个文件中，实现真正的组件封装。

1.3 分解 React 应用

前面我们提到过，React 应用实际上依赖于一个很大很复杂的技术栈，我们使用

create-react-app 避免在一开始就费太多精力配置技术栈，不过现在是时候了解一下这个技术栈了。

我们启动 React 应用的命令是 npm start，看一看 package.json 中对 start 脚本的定义，如下所示：

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test --env=jsdom",
  "eject": "react-scripts eject"
}
```

可以看到，start 命令实际上是调用了 react-scripts 这个命令，react-scripts 是 create-react-app 添加的一个 npm 包，所有的配置文件都藏在 node_modules/react-scripts 目录下，我们当然可以钻进这个目录去一探究竟，但是也可以使用 eject 方法来看清楚背后的原理。

你可以发现 package.json 文件中和 start 并列还有其他几个命令，其中 build 可以创建生产环境优化代码，test 用于单元测试，还有一个 eject 命令很有意思。

这个 eject（弹射）命令做的事情，就是把潜藏在 react-scripts 中的一系列技术栈配置都“弹射”到应用的顶层，然后我们就可以研究这些配置细节了，而且可以更灵活地定制应用的配置。

注意　eject 命令是不可逆的，就好像战斗机飞行员选择“弹射”出驾驶舱，等于是放弃了这架战斗机，是不可能再飞回驾驶舱的。所以，当你执行 eject 之前，最好做一下备份。

我们在命令行下执行下面的命令，完成“弹射”操作：

```
npm run eject
```

这个命令会让改变一些文件，也会添加一些文件。

当前目录下会增加两个目录，一个是 scripts，另一个是 config，同时，package.json 文件中的 scripts 部分也发生了变化：

```
"scripts": {
  "start": "node scripts/start.js",
  "build": "node scripts/build.js",
  "test": "node scripts/test.js --env=jsdom"
},
```

从此之后，start 脚本将使用 scripts 目录下的 start.js，而不是 node_modules 目录下的

react-scripts，弹射成功，再也回不去了。

在 config 目录下的 webpack.config.dev.js 文件，定制的就是 npm start 所做的构造过程，其中有一段关于 babel 的定义：

```
{
  test: /\.(js|jsx)$/,
  include: paths.appSrc,
  loader: 'babel',
  query: {
    // This is a feature of `babel-loader` for webpack (not Babel itself).
    // It enables caching results in ./node_modules/.cache/babel-loader/
    // directory for faster rebuilds.
    cacheDirectory: true
  }
},
```

代码中 paths.appSrc 的值就是 src，所以这段配置的含义指的是所有以 js 或者 jsx 为扩展名的文件，都会由 babel 所处理。

并不是所有的浏览器都支持所有 ES6 语法，但是有了 babel，我们就可以不用顾忌太多，因为 babel 会把 ES6 语法的 JavaScript 代码转译（transpile）成浏览器普遍支持的 JavaScript 代码，实际上，在 React 的社区中，不使用 ES6 语法写代码才显得奇怪。

1.4 React 的工作方式

在继续深入学习 React 的其他知识之前，我们先就这个简单的 ClickCounter 组件思考一下 React 的工作方式，要了解一样东西的特点，最好的方法当然是拿这个东西和另一样东西做比较。我们就拿 React 和 jQuery 来比较。

1.4.1 jQuery 如何工作

假设我们用 jQuery 来实现 ClickCounter 的功能，该怎么做呢？首先，我们要产生一个网页的 HTML，写一个 index.html 文件如下所示：

```
<!doctype html>
<html>
  <body>
    <div>
      <button id="clickMe">Click Me</button>
      <div>
        Click Count: <span id="clickCount">0</span>
      </div>
```

```
    </div>
    <script src="http://lib.sinaapp.com/js/jquery/1.9.1/jquery-1.9.1.min.js"></
      script>
    <script src="./clickCounter.js"></script>
  </body>
</html>
```

实际产品中，产生这样的 HTML 可以用 PHP、Java、Ruby on Rails 或者任何一种服务器端语言和框架来做，也可以在浏览器中用 Mustache、Hogan 这样的模板产生，这里我们只是把问题简化，直接书写 HTML。

jQuery 已经发展到 3.x 版，但已经不支持比较老的浏览器了，我们这里使用 1.9.1 的 jQuery 只是为了让这个网页在 IE8 上也能够运行。

上面的 HTML 只是展示样式，并没有任何交互功能，现在我们用 jQuery 来实现交互功能，和 jQuery 的传统一样，我们把 JavaScript 代码写在一个独立的文件 clickCounter.js 里面，如下所示：

```
$(function() {
  $('#clickMe').click(function() {
    var clickCounter = $('#clickCount');
    var count = parseInt(clickCounter.text(), 10);
    clickCounter.text(count+1);
  })
})
```

用浏览器打开上面创造的 index.html，可以看到实际效果和我们写的 React 应用一模一样，但是对比这两段程序可以看出差异。

在 jQuery 的解决方案中，首先根据 CSS 规则找到 id 为 clickCount 的按钮，挂上一个匿名事件处理函数，在事件处理函数中，选中那个需要被修改的 DOM 元素，读取其中的文本值，加以修改，然后修改这个 DOM 元素。

选中一些 DOM 元素，然后对这些元素做一些操作，这是一种最容易理解的开发模式。jQuery 的发明人 John Resig 就是发现了网页应用开发者的这个编程模式，才创造出了 jQuery，其一问世就得到普遍认可，因为这种模式直观易懂。但是，对于庞大的项目，这种模式会造成代码结构复杂，难以维护，每个 jQuery 的使用者都会有这种体会。

1.4.2　React 的理念

与 jQuery 不同，用 React 开发应用是另一种体验，我们回顾一下，用 React 开发的 ClickCounter 组件并没有像 jQuery 那样做“选中一些 DOM 元素然后做一些事情”的动作。

打一个比方，React 是一个聪明的建筑工人，而 jQuery 是一个比较傻的建筑工人，开发者你就是一个建筑的设计师，如果是 jQuery 这个建筑工人为你工作，你不得不事无巨细地告诉 jQuery“如何去做”，要告诉他这面墙要拆掉重建，那面墙上要新开一个窗户，反之，如果是 React 这个建筑工人为你工作，你所要做的就是告诉这个工人“我想要什么样子”，只要把图纸递给 React 这个工人，他就会替你搞定一切，当然他不会把整个建筑拆掉重建，而是很聪明地把这次的图纸和上次的图纸做一个对比，发现不同之处，然后只去做适当的修改就完成任务了。

显而易见，React 的工作方式把开发者从繁琐的操作中解放出来，开发者只需要着重“我想要显示什么”，而不用操心“怎样去做”。

这种新的思维方式，对于一个简单的例子也要编写不少代码，感觉像是用高射炮打蚊子，但是对于一个大型的项目，这种方式编写的代码会更容易管理，因为整个 React 应用要做的就是渲染，开发者关注的是渲染成成什么样子，而不用关心如何实现增量渲染。

React 的理念，归结为一个公式，就像下面这样：

$$UI = render(data)$$

让我们来看看这个公式表达的含义，用户看到的界面（UI），应该是一个函数（在这里叫 render）的执行结果，只接受数据（data）作为参数。这个函数是一个**纯函数**，所谓纯函数，指的是没有任何副作用，输出完全依赖于输入的函数，两次函数调用如果输入相同，得到的结果也绝对相同。如此一来，最终的用户界面，在 render 函数确定的情况下完全取决于输入数据。

对于开发者来说，重要的是区分开哪些属于 data，哪些属于 render，想要更新用户界面，要做的就是更新 data，用户界面自然会做出响应，所以 React 实践的也是“响应式编程”（Reactive Programming）的思想，这也就是 React 为什么叫做 React 的原因。

1.4.3 Virtual DOM

既然 React 应用就是通过重复渲染来实现用户交互，你可能会有一个疑虑：这样的重复渲染会不会效率太低了呢？毕竟，在 jQuery 的实现方式中，我们可以清楚地看到每次只有需要变化的那一个 DOM 元素被修改了；可是，在 React 的实现方式中，看起来每次 render 函数被调用，都要把整个组件重新绘制一次，这样看起来有点浪费。

事实并不是这样，React 利用 Virtual DOM，让每次渲染都只重新渲染最少的 DOM 元素。

要了解 Virtual DOM，就要先了解 DOM，DOM 是结构化文本的抽象表达形式，特

定于 Web 环境中，这个结构化文本就是 HTML 文本，HTML 中的每个元素都对应 DOM 中某个节点，这样，因为 HTML 元素的逐级包含关系，DOM 节点自然就构成了一个树形结构，称为 DOM 树。

浏览器为了渲染 HTML 格式的网页，会先将 HTML 文本解析以构建 DOM 树，然后根据 DOM 树渲染出用户看到的界面，当要改变界面内容的时候，就去改变 DOM 树上的节点。

Web 前端开发关于性能优化有一个原则：**尽量减少 DOM 操作**。虽然 DOM 操作也只是一些简单的 JavaScript 语句，但是 DOM 操作会引起浏览器对网页进行重新布局，重新绘制，这就是一个比 JavaScript 语句执行慢很多的过程。

如果使用 mustache 或者 hogan 这样的模板工具，那就是生成 HTML 字符串塞到网页中，浏览器又要做一次解析产生新的 DOM 节点，然后替换 DOM 树上对应的子树部分，这个过程肯定效率不高。虽然 JSX 看起来很像是一个模板，但是最终会被 Babel 解析为一条条创建 React 组件或者 HTML 元素的语句，神奇之处在于，React 并不是通过这些语句直接构建 DOM 树，而是首先构建 Virtual DOM。

既然 DOM 树是对 HTML 的抽象，那 Virtual DOM 就是对 DOM 树的抽象。Virutal DOM 不会触及浏览器的部分，只是存在于 JavaScript 空间的树形结构，每次自上而下渲染 React 组件时，会对比这一次产生的 Virtual DOM 和上一次渲染的 Virtual DOM，对比就会发现差别，然后修改真正的 DOM 树时就只需要触及差别中的部分就行。

以 ClickCounter 为例，一开始点击计数为 0，用户点击按钮让点击计数变成 1，这一次重新渲染，React 通过 Virtual DOM 的对比发现其实只是 id 为 clickCount 的 span 元素中内容从 0 变成了 1 而已：

```
<span id=" clickCount" >{this.state.count}</span>
```

React 发现这次渲染要做的事情只是更换这个 span 元素的内容而已，其他 DOM 元素都不需要触及，于是执行类似下面的语句，就完成了任务：

```
document.getElementById("clickCount").innerHTML = "1";
```

React 对比 Virtual DOM 寻找差异的过程比较复杂，在第 5 章，我们会详细介绍对比的过程。

1.4.4　React 工作方式的优点

毫无疑问，jQuery 的方式直观易懂，对于初学者十分适用，但是当项目逐渐变得庞大时，用 jQuery 写出的代码往往互相纠缠，形成类似图 1-4 的状况，难以维护。

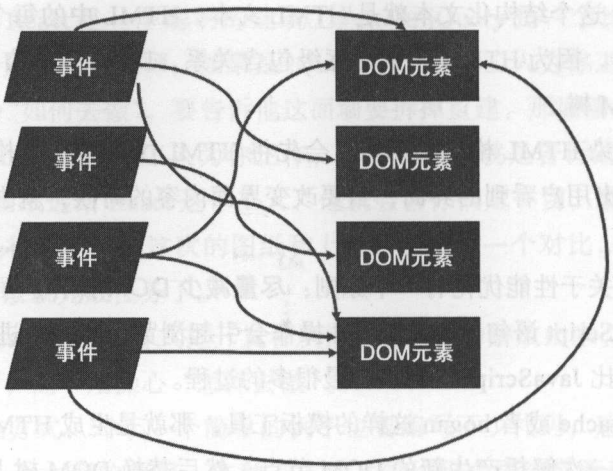


图 1-4 jQuery 方式造成的纠缠代码结构

使用 React 的方式，就可以避免构建这样复杂的程序结构，无论何种事件，引发的都是 React 组件的重新渲染，至于如何只修改必要的 DOM 部分，则完全交给 React 去操作，开发者并不需要关心，程序的流程简化为图 1-5 的样式。

React 利用函数式编程的思维来解决用户界面渲染的问题，最大的优势是开发者的效率会大大提高，开发出来的代码可维护性和可阅读性也大大增强。

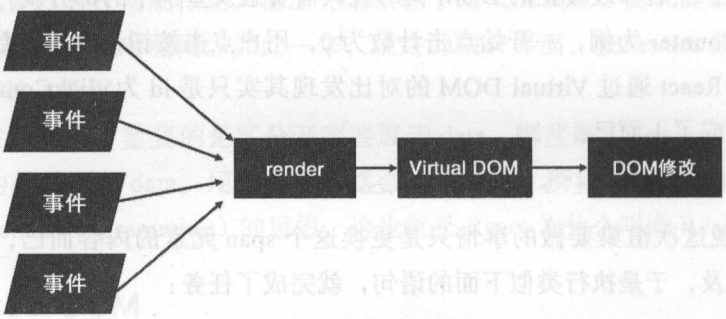


图 1-5 React 的程序流程

React 等于强制所有组件都按照这种由数据驱动渲染的模式来工作，无论应用的规模多大，都能让程序处于可控范围内。

1.5 本章小结

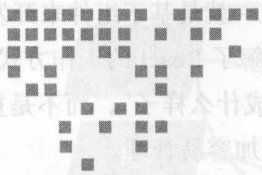
在这一章里，我们用 create-react-app 创造了一个简单的 React 应用，在一开始，我

们就按照组件的思想来开发应用，React 的主要理念之一就是基于组件来开发应用。

通过和同样功能的 jQuery 实现方式对比，我们了解了 React 的工作方式，React 利用声明式的语法，让开发者专注于描述用户界面“显示成什么样子”，而不是重复思考“如何去显示”，这样可以大大提高开发效率，也让代码更加容易管理。

虽然 React 是通过重复渲染来实现动态更新效果，但是借助 Virtual DOM 技术，实际上这个过程并不牵涉太多的 DOM 操作，所以渲染效率很高。

理解 React 的工作方式，是踏入 React 世界的关键一步，接下来我们详细介绍如何构建高质量的 React 组件。



Chapter 2 第 2 章

设计高质量的 React 组件

作为一个合格的开发者，不要只满足于编写出了可以运行的代码，而要了解代码背后的工作原理；不要只满足于自己编写的程序能够运行，还要让自己的代码可读而且易于维护。这样才能开发出高质量的软件。

在这一章里，我们将深入介绍构建高质量 React 组件的原则和方法，包括以下内容：

- 划分组件边界的原则；
- React 组件的数据种类；
- React 组件的生命周期。

本章的内容只是 React 组件设计的基础知识，因为 React 应用都是围绕组件的设计，所以关于组件的设计介绍将贯穿全书。

2.1 易于维护组件的设计要素

任何一个复杂的应用，都是由一个简单的应用发展而来的，当应用还很简单的时候，因为功能很少，可能只有一个组件就足够了，但是，随着功能的增加，把越来越多的功能放在一个组件中就会显得臃肿和难以管理。

就和一个人最好一次只专注做一件事一样，也应该尽量保持一个组件只做一件事。当开发者发现一个组件功能太多代码量太大的时候，就要考虑拆分这个组件，用多个小的组件来代替。每个小的组件只关注实现单个功能，但是这些功能组合起来，也能满足

复杂的实际需求。

这就是“分而治之”的策略，把问题分解为多个小问题，这样既容易解决也方便维护，虽然“分而治之”是一个好策略，但是不要滥用，只有必要的时候才去拆分组件，不然可能得不偿失。

拆分组件最关键的就是确定组件的边界，每个组件都应该是可以独立存在的，如果两个组件逻辑太紧密，无法清晰定义各自的责任，那也许这两个组件本身就不该被拆开，作为同一个组件也许更合理。

虽然组件是应该独立存在的，但是并不是说组件就是孤岛一样的存在，不同组件之间总是会有通信交流，这样才可能组合起来完成更大的功能。

作为软件设计的通则，组件的划分要满足高内聚 (High Cohesion) 和低耦合 (Low Coupling) 的原则。

高内聚指的是把逻辑紧密相关的内容放在一个组件中。用户界面无外乎内容、交互行为和样式。传统上，内容由 HTML 表示，交互行放在 JavaScript 代码文件中，样式放在 CSS 文件中定义。这虽然满足一个功能模块的需要，却要放在三个不同的文件中，这其实不满足高内聚的原则。React 却不是这样，展示内容的 JSX、定义行为的 JavaScript 代码，甚至定义样式的 CSS，都可以放在一个 JavaScript 文件中，因为它们本来就是为了实现一个目的而存在的，所以说 React 天生具有高内聚的特点。

低耦合指的是不同组件之间的依赖关系要尽量弱化，也就是每个组件要尽量独立。保持整个系统的低耦合度，需要对系统中的功能有充分的认识，然后根据功能点划分模块，让不同的组件去实现不同的功能，这个功夫还在开发者身上，不过，React 组件的对外接口非常规范，方便开发者设计低耦合的系统。

上面介绍的只是通用的软件设计原则，对于 React 这个工具，要充分应用这些原则，则需要对 React 的特性有充分的认识。

2.2　React 组件的数据

“差劲的程序员操心代码，优秀的程序员操心数据结构和它们之间的关系。”

——Linus Torvalds，Linux 创始人

毫无疑问，如何组织数据是程序的最重要问题。

React 组件的数据分为两种，prop 和 state，无论 prop 或者 state 的改变，都可能引发组件的重新渲染，那么，设计一个组件的时候，什么时候选择用 prop 什么时候选择用

state 呢？其实原则很简单，prop 是组件的对外接口，state 是组件的内部状态，对外用 prop，内部用 state。

为了演示属性的使用，我们构造一个应用包含两个组件，Counter 组件和 ControlPanel 组件，其中 ControlPanel 组件是父组件，包含若干个 Counter 组件，相关代码在 Github 上对应本书的代码库 https://github.com/mocheng/react-and-redux/tree/master/chapter-02/controlpanel [⊖]中。

图 2-1 ControlPanel 应用效果图

我们可以看到三个 Counter 组件有了不同的初始计数值，点击网页中的“+”按钮可以看到对应一行的计数增加，点击“-”按钮可以看到对应一行的计数减少。

这并不是一个复杂的程序，但是可以看出属性和状态在 React 组件中的作用。

2.2.1 React 的 prop

在 React 中，prop（property 的简写）是从外部传递给组件的数据，一个 React 组件通过定义自己能够接受的 prop 就定义了自己的对外公共接口。

每个 React 组件都是独立存在的模块，组件之外的一切都是外部世界，外部世界就是通过 prop 来和组件对话的。

1. 给 prop 赋值

我们先从外部世界来看，prop 是如何使用的，在下面的 JSX 代码片段中，就使用了 prop：

```
<SampleButton
  id="sample" borderWidth={2} onClick={onButtonClick}
  style={{color: "red"}}
/>
```

⊖ Licensing and the library version of git https://lwn.net/Articles/193245/

在上面的例子中，创建了名为 SampleButton 的组件实例，使用了名字分别为 id、borderWidth、onClick 和 style 的 prop，看起来，React 组件的 prop 很像是 HTML 元素的属性，不过，HTML 组件属性的值都是字符串类型，即使是内嵌 JavaScript，也依然是字符串形式表示代码。React 组件的 prop 所能支持的类型则丰富得多，除了字符串，可以是任何一种 JavaScript 语言支持的数据类型。

比如在上面的 SampleButton 中，borderWidth 就是数字类型，onClick 是函数类型，style 的值是一个包含 color 字段的对象，当 prop 的类型不是字符串类型时，在 JSX 中必须用花括号 {} 把 prop 值包住，所以 style 的值有两层花括号，外层花括号代表是 JSX 的语法，内层的花括号代表这是一个对象常量。

当外部世界要传递一些数据给 React 组件，一个最直接的方式就是通过 prop；同样，React 组件要反馈数据给外部世界，也可以用 prop，因为 prop 的类型不限于纯数据，也可以是函数，函数类型的 prop 等于让父组件交给了子组件一个回调函数，子组件在恰当的实际调用函数类型的 prop，可以带上必要的参数，这样就可以反过来把信息传递给外部世界。

对于 Counter 组件，父组件 ControlPanel 就是外部世界，我们看 ControlPanel 是如何用 prop 传递信息给 Counter 的，代码如下：

```
class ControlPanel extends Component {
  render() {
    return (
      <div>
        <Counter caption="First" initValue={0} />
        <Counter caption="Second" initValue={10} />
        <Counter caption="Third" initValue={20} />
      </div>
    );
  }
}
```

ControlPanel 组件包含三个 Counter 组件实例，在 ControlPanel 的 render 函数中将这三个子组件实例用 div 包起来，因为 React 要求 render 函数只能返回一个元素。

虽然 React 声称将来可以支持 render 返回数组以支持多个组件，但是到本书写作时最新版 v15.4 为止，这个功能依然没有实现。

在每个 Counter 组件实例中，都使用了 caption 和 initValue 两个 prop。通过名为 caption 的 prop，ControlPanel 传递给 Counter 组件实例说明文字。通过名为 initValue 的 prop 传递给 Counter 组件一个初始的计数值。

2. 读取 prop 值

我们再来看 Counter 组件内部是如何接收传入的 prop 的，首先是构造函数，代码如下：

```
class Counter extends Component {
  constructor(props) {
    super(props);

    this.onClickIncrementButton = this.onClickIncrementButton.bind(this);
    this.onClickDecrementButton = this.onClickDecrementButton.bind(this);

    this.state = {
      count: props.initValue || 0
    }
  }
```

如果一个组件需要定义自己的构造函数，一定要记得在构造函数的第一行通过 super 调用父类也就是 React.Component 的构造函数。如果在构造函数中没有调用 super(props)，那么组件实例被构造之后，类实例的所有成员函数就无法通过 this.props 访问到父组件传递过来的 props 值。很明显，给 this.props 赋值是 React.Component 构造函数的工作之一。

在 Counter 的构造函数中还给两个成员函数绑定了当前 this 的执行环境，因为 ES6 方法创造的 React 组件类并不自动给我们绑定 this 到当前实例对象。

在构造函数的最后，我们可以看到读取传入 prop 的方法，在构造函数中可以通过参数 props 获得传入 prop 值，在其他函数中则可以通过 this.props 访问传入 prop 的值，比如在 Counter 组件的 render 函数中，我们就是通过 this.props 获得传入的 caption，render 函数代码如下：

```
render() {
  const {caption} = this.props;
  return (
    <div>
  <button style={buttonStyle} onClick={this.onClickIncrementButton}>+</button>
  <button style={buttonStyle} onClick={this.onClickDecrementButton}>-</button>
  <span>{caption} count: {this.state.count}</span>
  </div>
  );
}
```

在上面的代码中，我们使用了 ES6 的**解构赋值**（destructuring assignment）语法从 this.props 中获得了名为 caption 的 prop 值。

3. propTypes 检查

既然 prop 是组件的对外接口，那么就应该有某种方式让组件声明自己的接口规范。

简单说，一个组件应该可以规范以下这些方面：

- 这个组件支持哪些 prop；
- 每个 prop 应该是什么样的格式。

React 通过 propTypes 来支持这些功能。

在 ES6 方法定义的组件类中，可以通过增加类的 propTypes 属性来定义 prop 规格，这不只是声明，而且是一种限制，在运行时和静态代码检查时，都可以根据 propTypes 判断外部世界是否正确地使用了组件的属性。

比如，对于 Counter 组件的 propTypes 定义代码如下：

```
Counter.propTypes = {
  caption: PropTypes.string.isRequired,
  initValue: PropTypes.number
};
```

其中要求 caption 必须是 string 类型，initValue 必须是 number 类型。可以看到，两者除了类型不同之外，还有一个区别：caption 带上了 isRequried，这表示使用 Counter 组件必须要指定 caption；而 initValue 因为没有 isRequired，则表示如果没有也没关系。

为了验证 propTypes 的作用，可以尝试故意违反 propTypes 的规定使用 Counter 实例，比如在 ControlPanel 的 render 函数中增加下列的代码：

```
<Counter caption={123} initValue={20} />
```

我们在 Chrome 浏览器开发工具的 Console 界面，可以看到一个红色的警告提示，如图 2-2 所示。

```
▶ Warning: Failed prop type: Invalid prop `caption` of type `number` supplied to `Counter`, expected `string`.
    in Counter (at ControlPanel.js:16)
    in ControlPanel (at index.js:7)
                                                    warning.js:36
```

图 2-2　错误 prop 类型的错误提示

这段出错的含义是，caption 属性预期是字符串类型，得到的却是一个数字类型。

我们尝试删掉这个 Counter 实例的 caption 属性，代码如下：

```
<Counter initValue={20} />
```

这时可以看到 Console 选项卡中依然有红色警告信息，如图 2-3 所示。

```
▶ Warning: Failed prop type: The prop `caption` is marked as required in `Counter`, but its value is `undefined`.
    in Counter (at ControlPanel.js:16)
    in ControlPanel (at index.js:7)
                                                    warning.js:36
```

图 2-3　缺失必须存在 prop 的错误提示

提示的含义是，caption 是 Counter 必需的属性，但是却没有赋值。

很明显，有了 propTypes 的检查，可以很容易发现对 prop 的不正确使用方法，可尽早发现代码中的错误。

从上面可以看得出来 propTypes 检查可以防止不正确的 prop 使用方法，那么如果组件根本就没有定义 propTypes 会怎么样呢？

可以尝试在 src/Counter.js 文件中删除掉那一段给 Counter.propTypes 赋值的语句，在浏览器 Console 里可以看到红色警告不再出现。可见，没有 propTypes 定义，组件依然能够正常工作，而且，即使在上面 propTypes 检查出错的情况下，组件依旧能工作。也就是说 propTypes 检查只是一个辅助开发的功能，并不会改变组件的行为。

propTypes 虽然能够在开发阶段发现代码中的问题，但是放在产品环境中就不大合适了。

首先，定义类的 propTypes 属性，无疑是要占用一些代码空间，而且 propTypes 检查也是要消耗 CPU 计算资源的。其次，在产品环境下做 propTypes 检查没有什么帮助，毕竟，propTypes 产生的这些错误信息只有开发者才能看得懂，放在产品环境下，在最终用户的浏览器 Console 中输出这些错误信息没什么意义。

所以，最好的方式是，开发者在代码中定义 propTypes，在开发过程中避免犯错，但是在发布产品代码时，用一种自动的方式将 propTypes 去掉，这样最终部署到产品环境的代码就会更优。现有的 babel-react-optimize 就具有这个功能，可以通过 npm 安装，但是应该确保只在发布产品代码时使用它。

2.2.2 React 的 state

驱动组件渲染过程的除了 prop，还有 state，state 代表组件的内部状态。由于 React 组件不能修改传入的 prop，所以需要记录自身数据变化，就要使用 state。

在 Counter 组件中，最初显示初始计数，可以通过 initValue 这个 prop 来定制，在 Counter 已经被显示之后，用户会点击“+”和“-”按钮改变这个计数，这个变化的数据就要 Counter 组件自己通过 state 来存储了。

1. 初始化 state

通常在组件类的构造函数结尾处初始化 state，在 Counter 构造函数中，通过对 this.state 的赋值完成了对组件 state 的初始化，代码如下：

```
constructor(props) {
  ...
  this.state = {
```

```
      count: props.initValue || 0
    }
  }
```

因为 initValue 是一个可选的 props，要考虑到父组件没有指定这个 props 值的情况，我们优先使用传入属性的 initValue，如果没有，就使用默认值 0。

组件的 state 必须是一个 JavaScript 对象，不能是 string 或者 number 这样的简单数据类型，即使我们需要存储的只是一个数字类型的数据，也只能把它存作 state 某个字段对应的值，Counter 组件里，我们的唯一数据就存在 count 字段里。

注意　在 React 创建之初，使用的是 React.createClass 方法创建组件类，这种方式下，通过定义组件类的一个 getInitialState 方法来获取初始 state 值，但是这种做法已经被废弃了，我们现在都用 ES6 的语法定义组件类，所以不再考虑定义 getInitialState 方法。

由于在 PropType 声明中没有用 isRequired 要求必须有值的 prop，例如上面的 initValue，我们需要在代码中判断所给 prop 值是否存在，如果不存在，就给一个默认的初始值。不过，让这样的判断逻辑充斥在我们组件的构造函数之中并不是一件美观的事情，而且容易有遗漏。我们可以用 React 的 defaultProps 功能，让代码更加容易读懂。

给 Counter 组件添加 defaultProps 的代码如下：

```
Counter.defaultProps = {
  initValue: 0
};
```

有了这样的设定，Counter 构造函数中的 this.state 初始化中可以省去判断条件，可以认为代码执行到这里，必有 initValue 属性值，代码可以简化为这样：

```
this.state = {
  count: props.initValue
}
```

以后，即使 Counter 的使用者没有指定 initValue，在组件中就会收到一个默认的属性值 0。

2. 读取和更新 state

通过给 button 的 onClick 属性挂载点击事件处理函数，我们可以改变组件的 state，以点击“+”按钮的响应函数为例，代码如下：

```
onClickIncrementButton() {
  this.setState({count: this.state.count + 1});
}
```

在代码中，通过 this.state 可以读取到组件的当前 state。值得注意的是，我们改变组件 state 必须要使用 this.setState 函数，而不能直接去修改 this.state。如果不相信，你可以尝试把 onClickIncrementButton 函数修改成下面的样子：

```
onClickIncrementButton() {
  this.state.count = this.state.count + 1;
}
```

既然 this.state 就是一个 JavaScript 对象，上面的代码逻辑看起来也没有什么问题，我们在界面上点击几个按钮看一下实际效果就会发现问题。

首先，在浏览器的 Console 中会有如下的提示：

```
warning Do not mutate state directly. Use setState() react/no-direct-mutation-state
```

这是在警告："不要直接修改 state，应该使用 setState()。"

当你点击"+"按钮，也不会看到后面的计数值有任何变化，但是当你点击"-"按钮，就会立即看到计数值发生变化，而且计数值会发生"跳跃"，比如在初始计数值为 0 的情况下，连续点击"+"按钮三次，计数值没有任何变化依然为 0，点击了一下"-"按钮一次，就会看到计数值一下子变成了 2，这是怎么回事？

直接修改 this.state 的值，虽然事实上改变了组件的内部状态，但只是野蛮地修改了 state，却没有驱动组件进行重新渲染，既然组件没有重新渲染，当然不会反应 this.state 值的变化；而 this.setState() 函数所做的事情，首先是改变 this.state 的值，然后驱动组件经历更新过程，这样才有机会让 this.state 里新的值出现在界面上。

在上面描述的操作中，连续点击三次"+"按钮，this.state 中的 count 字段值已经被增加到了 3，但是没有重新渲染，这时候点击一次"-"按钮，触发的 onClick-DecrementButton 函数依然是用 this.setState 改变组件状态，这个函数调用首先把 this.state 中的 count 值从 3 减少为 2，然后触发重新渲染，于是我们看到界面上的数字一下子从 0 跳跃为 2。

2.2.3 prop 和 state 的对比

总结一下 prop 和 state 的区别：

- prop 用于定义外部接口，state 用于记录内部状态；
- prop 的赋值在外部世界使用组件时，state 的赋值在组件内部；
- 组件不应该改变 prop 的值，而 state 存在的目的就是让组件来改变的。

组件的 state，就相当于组件的记忆，其存在意义就是被修改，每一次通过 this.setState 函数修改 state 就改变了组件的状态，然后通过渲染过程把这种变化体现出来。

但是，组件是绝不应该去修改传入的 props 值的，我们设想一下，假如父组件包含多个子组件，然后把一个 JavaScript 对象作为 props 值传给这几个子组件，而某个子组件居然改变了这个对象的内部值，那么，接下来其他子组件读取这个对象会得到什么值呢？当时读取了修改过的值，但是其他子组件是每次渲染都读取这个 props 的值呢？还是只读一次以后就用那个最初值呢？一切皆有可能，完全不可预料。也就是说，一个子组件去修改 props 中的值，可能让程序陷入一团混乱之中，这就完全违背了 React 设计的初衷。

还记得第 1 章中我们看到的那个公式吗？

$$UI = render(data)$$

React 组件扮演的是 render 函数的角色，应该是一个没有副作用的纯函数。修改 props 的值，是一个副作用，组件应该避免。

严格来说，React 并没有办法阻止你去修改传入的 props 对象。所以，每个开发者就把这当做一个规矩，在编码中一定不要踩这儿红线，不然最后可能遇到不可预料的 bug。

2.3　组件的生命周期

为了理解 React 的工作过程，我们就必须要了解 React 组件的生命周期，如同人有生老病死，自然界有日月更替，每个组件在网页中也会被创建、更新和删除，如同有生命的机体一样。

React 严格定义了组件的生命周期，生命周期可能会经历如下三个过程：

- 装载过程（Mount），也就是把组件第一次在 DOM 树中渲染的过程；
- 更新过程（Update），当组件被重新渲染的过程；
- 卸载过程（Unmount），组件从 DOM 中删除的过程。

三种不同的过程，React 库会依次调用组件的一些成员函数，这些函数称为生命周期函数。所以，要定制一个 React 组件，实际上就是定制这些生命周期函数。

2.3.1　装载过程

我们先来看装载过程，当组件第一次被渲染的时候，依次调用的函数是如下这些：

- constructor
- getInitialState
- getDefaultProps
- componentWillMount

- render
- componentDidMount

我们来逐个地详细解释这些函数的功能。

1. constructor

我们先来看第一个 constructor，也就是 ES6 中每个类的构造函数，要创造一个组件类的实例，当然会调用对应的构造函数。

要注意，并不是每个组件都需要定义自己的构造函数。在后面的章节我们可以看到，无状态的 React 组件往往就不需要定义构造函数，一个 React 组件需要构造函数，往往是为了下面的目的：

- 初始化 state，因为组件生命周期中任何函数都可能要访问 state，那么整个生命周期中第一个被调用的构造函数自然是初始化 state 最理想的地方；
- 绑定成员函数的 this 环境。

在 ES6 语法下，类的每个成员函数在执行时的 this 并不是和类实例自动绑定的。而在构造函数中，this 就是当前组件实例，所以，为了方便将来的调用，往往在构造函数中将这个实例的特定函数绑定 this 为当前实例。

以 Counter 组件为例，我们的构造函数有这样如下的代码：

```
this.onClickIncrementButton = this.onClickIncrementButton.bind(this);
this.onClickDecrementButton = this.onClickDecrementButton.bind(this);
```

这两条语句的作用，就是通过 bind 方法让当前实例中 onClickIncrementButton 和 onClickDecrementButton 函数被调用时，this 始终是指向当前组件实例。

在某些教程中，大家还会看到另一种 bind 函数的方式，类似下面的语句：

```
this.foo = ::this.foo;
```

等同于下面的语句：

```
this.foo = this.foo.bind(this);
```

这里所使用的两个冒号的 :: 操作符叫做 bind 操作符，虽然有 babel 插件支持这种写法，但是 bind 操作符可能不会成为 ES 标准语法的一部分，所以，虽然这种写法看起来很简洁，我们在本书中并不使用它。

2. getInitialState 和 getDefaultProps

getInitialState 这个函数的返回值会用来初始化组件的 this.state，但是，这个方法只有用 React.createClass 方法创造的组件类才会发生作用，本书中我们一直使用的 ES6 语

法，所以这个函数根本不会产生作用。

getDefaultProps 函数的返回值可以作为 props 的初始值，和 getInitialState 一样，这个函数只在 React.createClass 方法创造的组件类才会用到。总之，实际上 getInitialState 和 getDefaultProps 两个方法在 ES6 的方法定义的 React 组件中根本不会用到。

假如我们用 React.createClass 方法定义一个组件 Sample，设定内部状态 foo 的初始值为字符串 bar，同时设定一个叫 sampleProp 的 prop 初始值为数字值 0，代码如下：

```
const Sample = React.createClass({
  getInitialState: function() {
    return {foo: 'bar'};
  },
  getDefaultProps: function() {
    return {sampleProp: 0}
  })
});
```

用 ES6 的话，在构造函数中通过给 this.state 赋值完成状态的初始化，通过给类属性（注意是类属性，而不是类的实例对象属性）defaultProps 赋值指定 props 初始值，达到的效果是完全一样的，代码如下：

```
class Sample extends React.Component {
  constructor(props) {
    super(props);
    this.state = {foo: 'bar'};
  }
});

Sample.defaultProps = {
  return {sampleProp: 0}
};
```

React.createClass 已经被 Facebook 官方逐渐废弃，但是在互联网上还能搜索到很多使用 React.createClass 的教材，虽然强烈建议不再要使用 React.createClass，但是如果读者你真的要用的话，需要注意关于 getInitialState 只出现在装载过程中，也就是说在一个组件的整个生命周期过程中，这个函数只被调用一次，不要在里面放置预期会被多次执行的代码。

3. render

render 函数无疑是 React 组件中最重要的函数，一个 React 组件可以忽略其他所有函数都不实现，但是一定要实现 render 函数，因为所有 React 组件的父类 React.Component 类对除 render 之外的生命周期函数都有默认实现。

通常一个组件要发挥作用，总是要渲染一些东西，render 函数并不做实际的渲染动作，它只是返回一个 JSX 描述的结构，最终由 React 来操作渲染过程。

当然，某些特殊组件的作用不是渲染界面，或者，组件在某些情况下选择没有东西可画，那就让 render 函数返回一个 null 或者 false，等于告诉 React，这个组件这次不需要渲染任何 DOM 元素。

需要注意，render 函数应该是一个纯函数，完全根据 this.state 和 this.props 来决定返回的结果，而且不要产生任何副作用。在 render 函数中去调用 this.setState 毫无疑问是错误的，因为一个纯函数不应该引起状态的改变。我们在后面的章节会对 render 函数做详细的介绍。

4. componentWillMount 和 componentDidMount

在装载过程中，componentWillMount 会在调用 render 函数之前被调用，componentDidMount 会在调用 render 函数之后被调用，这两个函数就像是 render 函数的前哨和后卫，一前一后，把 render 函数夹住，正好分别做 render 前后必要的工作。

不过，我们通常不用定义 componentWillMount 函数，顾名思义，componentWillMount 发生在“将要装载”的时候，这个时候没有任何渲染出来的结果，即使调用 this.setState 修改状态也不会引发重新绘制，一切都迟了。换句话说，所有可以在这个 componentWillMount 中做的事情，都可以提前到 constructor 中间去做，可以认为这个函数存在的主要目的就是为了和 componentDidMount 对称。

而 componentWillMount 的这个兄弟 componentDidMount 作用就大了。

需要注意的是，render 函数被调用完之后，componentDidMount 函数并不是会被立刻调用，componentDidMount 被调用的时候，render 函数返回的东西已经引发了渲染，组件已经被“装载”到了 DOM 树上。

我们还是以 ControlPanel 为例，在 ControlPanel 中有三个 Counter 组件，我们稍微修改 Counter 的代码，让装载过程中所有生命周期函数都用 console.log 输出函数名和 caption 的值，比如，componentWillMount 函数的内容如下：

```
componentWillMount() {
  console.log('enter componentWillMount ' + this.props.caption);
}
```

上面修改并没有添加任何功能，只是通过 console.log 输出一些内容，然后我们刷新网页，在浏览器的 console 里我们能够看见：

```
enter constructor: First

enter componentWillMount First
```

```
enter render First
enter constructor: Second
enter componentWillMount Second
enter render Second
enter constructor: Third
enter componentWillMount Third
enter render Third
enter componentDidMount First
enter componentDidMount Second
enter componentDidMount Third
```

可以清楚地看到，虽然 componentWillMount 都是紧贴着自己组件的 render 函数之前被调用，componentDidMount 可不是紧跟着 render 函数被调用，当所有三个组件的 render 函数都被调用之后，三个组件的 componentDidMount 才连在一起被调用。

之所以会有上面的现象，是因为 render 函数本身并不往 DOM 树上渲染或者装载内容，它只是返回一个 JSX 表示的对象，然后由 React 库来根据返回对象决定如何渲染。而 React 库肯定是要把所有组件返回的结果综合起来，才能知道该如何产生对应的 DOM 修改。所以，只有 React 库调用三个 Counter 组件的 render 函数之后，才有可能完成装载，这时候才会依次调用各个组件的 componentDidMount 函数作为装载过程的收尾。

componentWillMount 和 componentDidMount 这对兄弟函数还有一个区别，就是 component-WillMount 可以在服务器端被调用，也可以在浏览器端被调用；而 component-DidMount 只能在浏览器端被调用，在服务器端使用 React 的时候不会被调用。

到目前为止，我们构造的 React 应用例子都只在浏览器端使用 React，所以看不出区别，但后面第 12 章关于“同构”应用的介绍时，我们会探讨在服务器端使用 React 的情况。

至于为什么只有 componentDidMount 仅在浏览器端执行，这是一个实现上的决定，而不是设计时刻意为之。不过，如果非要有个解释的话，可以这么说，既然“装载”是一个创建组件并放到 DOM 树上的过程，那么，真正的“装载”是不可能在服务器端完成的，因为服务器端渲染并不会产生 DOM 树，通过 React 组件产生的只是一个纯粹的字符串而已。

不管怎样，componentDidMount 只在浏览器端执行，倒是给了我们开发者一个很好的位置去做只有浏览器端才做的逻辑，比如通过 AJAX 获取数据来填充组件的内容。

在 componentDidMount 被调用的时候，组件已经被装载到 DOM 树上了，可以放心获取渲染出来的任何 DOM。

在实际开发过程中，可能会需要让 React 和其他 UI 库配合使用，比如，因为项目前期已经用 jQuery 开发了很多功能，需要继续使用这些基于 jQuery 的代码，有时候其他的

UI 库做某些功能比 React 更合适，比如 d3.js 已经支持了丰富的绘制图表的功能，在这些情况下，我们不得不考虑如何让 React 和其他 UI 库和平共处。

以和 jQuery 配合为例，我们知道，React 是用来取代 jQuery 的，但如果真的要让 React 和 jQuery 配合，就需要是利用 componentDidMount 函数，当 componentDidMount 被执行时，React 组件对应的 DOM 已经存在，所有的事件处理函数也已经设置好，这时候就可以调用 jQuery 的代码，让 jQuery 代码在已经绘制的 DOM 基础上增强新的功能。

在 componentDidMount 中调用 jQuery 代码只处理了装载过程，要和 jQuery 完全结合，又要考虑 React 的更新过程，就需要使用下面要讲的 componentDidUpdate 函数。

2.3.2 更新过程

当组件被装载到 DOM 树上之后，用户在网页上可以看到组件的第一印象，但是要提供更好的交互体验，就要让该组件可以随着用户操作改变展现的内容，当 props 或者 state 被修改的时候，就会引发组件的更新过程。

更新过程会依次调用下面的生命周期函数，其中 render 函数和装载过程一样，没有差别。

- componentWillReceiveProps
- shouldComponentUpdate
- componentWillUpdate
- render
- componentDidUpdate

有意思的是，并不是所有的更新过程都会执行全部函数，下面会介绍到各种特例。

1. componentWillReceiveProps(nextProps)

关于这个 componentWillReceiveProps 存在一些误解。在互联网上有些教材声称这个函数只有当组件的 props 发生改变的时候才会被调用，其实是不正确的。实际上，只要是父组件的 render 函数被调用，在 render 函数里面被渲染的子组件就会经历更新过程，不管父组件传给子组件的 props 有没有改变，都会触发子组件的 componentWillReceiveProps 函数。

注意，通过 this.setState 方法触发的更新过程不会调用这个函数，这是因为这个函数适合根据新的 props 值（也就是参数 nextProps）来计算出是不是要更新内部状态 state。更新组件内部状态的方法就是 this.setState，如果 this.setState 的调用导致 componentWillReceiveProps 再一次被调用，那就是一个死循环了。

让我们对 ControlPanel 做一些小的改进，来体会一下上面提到的规则。

我们首先在 Counter 组件类里增加函数定义，让这个函数 componentWillReceiveProps 在 console 上输出一些文字，代码如下：

```
componentWillReceiveProps(nextProps) {
  console.log('enter componentWillReceiveProps ' + this.props.caption)
}
```

在 ControlPanel 组件的 render 函数中，我们也做如下修改：

```
render() {
  console.log('enter ControlPanel render');
  return (
    <div style={style}>
      . . .
      <button onClick={ () => this.forceUpdate() }>
        Click me to repaint!
      </button>
    </div>
  );
}
```

除了在 ControlPanel 的 render 函数入口处增加 console 输出，我们还增加了一个按钮，这个按钮的 onClick 事件引发一个匿名函数，当这个按钮被点击的时候，调用 this.forceUpdate，每个 React 组件都可以通过 forceUpdate 函数强行引发一次重新绘制。

注意　类似上面的代码，在 JSX 用直接把匿名函数赋值给 onClick 的方法，看起来非常简洁而且方便，其实并不是值得提倡的方法。因为每次渲染都会创造一个新的匿名方法对象，而且有可能引发子组件不必要的重新渲染，原因在后面的章节会有详细介绍。

在网页中，我们去点击那个新增加的按钮，可以看到浏览器的 console 中有如下的输出：

```
enter ControlPanel render
enter componentWillReceiveProps First
enter render First
enter componentWillReceiveProps Second
enter render Second
enter componentWillReceiveProps Third
enter render Third
```

可以看到，引发 forceUpdate 之后，首先是 ControlPanel 的 render 函数被调用，随后第一个 Counter 组件的 componentWillReceiveProps 函数被调用，然后 Counter 组件的 render 函数被调用，随后第二第三个函数的这两个函数也依次被调用。

然而，ControlPanel 在渲染三个子组件的时候，提供的 props 值一直就没有变化，可见 componentWillReceiveProps 并不是当 props 值变化的时候才被调用，所以，这个函数有必要把传入参数 nextProps 和 this.props 作必要对比。nextProps 代表的是这一次渲染传入的 props 值，this.props 代表的上一次渲染时的 props 值，只有两者有变化的时候才有必要调用 this.setState 更新内部状态。

在网页中，我们再尝试点击第一个 Counter 组件的“+”按钮，可以看到浏览器的 console 输出如下：

```
enter render First
```

明显，只有第一个组件 Counter 的 render 函数被调用，函数 componentWillReceiveProps 没有被调用。因为点击“+”按钮引发的是第一个 Counter 组件的 this.setState 函数的调用，就像上面说过的一样，this.setState 不会引发这个函数 componentWillReceiveProps 被调用。

从这个例子中我们也会发现，在 React 的组件组合中，完全可以只渲染一个子组件，而其他组件完全不需要渲染，这是提高 React 性能的重要方式。

2. shouldComponentUpdate(nextProps, nextState)

除了 render 函数，shouldComponentUpdate 可能是 React 组件生命周期中最重要的一个函数了。

说 render 函数重要，是因为 render 函数决定了该渲染什么，而说 shouldComponentUpdate 函数重要，是因为它决定了一个组件什么时候不需要渲染。

render 和 shouldComponentUpdate 函数，也是 React 生命周期函数中唯二两个要求有返回结果的函数。render 函数的返回结果将用于构造 DOM 对象，而 shouldComponentUpdate 函数返回一个布尔值，告诉 React 库这个组件在这次更新过程中是否要继续。

在更新过程中，React 库首先调用 shouldComponentUpdate 函数，如果这个函数返回 true，那就会继续更新过程，接下来调用 render 函数；反之，如果得到一个 false，那就立刻停止更新过程，也就不会引发后续的渲染了。

说 shouldComponentUpdate 重要，就是因为只要使用恰当，他就能够大大提高 React 组件的性能，虽然 React 的渲染性能已经很不错了，但是，不管渲染有多快，如果发现没必要重新渲染，那就干脆不用渲染好了，速度会更快。

我们知道 render 函数应该是一个纯函数，这个纯函数的逻辑输入就是组件的 props 和 state。所以，shouldComponentUpdate 的参数就是接下来的 props 和 state 值。如果我们要定义 shouldComponentUpdate，那就根据这两个参数，外加 this.props 和 this.state 来

判断出是返回 true 还是返回 false。

如果我们给组件添加 shouldCompomentUpdate 函数，那就沿用所有 React 组件父类 React.Component 中的默认实现方式，默认实现方式就是简单地返回 true，也就是每次更新过程都要重新渲染。当然，这是最稳妥的方式，大不了浪费一点，但是绝对不会出错。不过若我们要追求更高的性能，就不能满足于默认实现，需要定制这个函数 shouldComponentUpdate。

让我们尝试来给 Counter 组件增加一个 shouldComponentUpdate 函数。先来看看 props，Counter 组件支持两个 props，一个叫 caption，一个叫 initValue。很明显，只有 caption 这个 prop 改变的时候，才有必要重新渲染。对于 initValue，只是创建 Counter 组件实例时用于初始化计数值，在组件实例创建之后，无论怎么改，都不应该让 Counter 组件重新渲染。

再来看看 state，Counter 组件的 state 只有一个值 count，如果 count 发生了变化，那肯定应该重新渲染，如果 count 没变化，那就没必要了。

现在，让我们给 Counter 组件类增加 shouldComponentUpdate 函数的定义，代码如下：

```
shouldComponentUpdate(nextProps, nextState) {
  return (nextProps.caption !== this.props.caption) ||
    (nextState.count !== this.state.count);
}
```

现在，只有当 caption 改变，或者 state 中的 count 值改变，shouldComponent 才会返回 true。

值得一提的是，通过 this.setState 函数引发更新过程，并不是立刻更新组件的 state 值，在执行到到函数 shouldComponentUpdate 的时候，this.state 依然是 this.setState 函数执行之前的值，所以我们要做的实际上就是在 nextProps、nextState、this.props 和 this.state 中互相比对。

我们在网页中引发一次 ControlPanel 的重新绘制，可以看到浏览器的 console 中的输出是这样：

```
enter ControlPanel render
enter componentWillReceiveProps First
enter componentWillReceiveProps Second
enter componentWillReceiveProps Third
```

可以看到，三个 Counter 组件的 render 函数都没有被调用，因为这个刷新没有改变 caption 的值，更没有引发组件内状态改变，所以完全没有必要重新绘制 Counter。

对于 Counter 这个简单的组件，我们无法感觉到性能的提高，但是，实际开发中会遇到更复杂更庞大的组件，这种情况下避免没必要的重新渲染，就会大大提高性能。

3. componentWillUpdate 和 componentDidUpdate

如果组件的 shouldComponentUpdate 函数返回 true，React 接下来就会依次调用对应组件的 componentWillUpdate、render 和 componentDidUpdate 函数。

componentWillMount 和 componentDidMount，componentWillUpdate 和 componentDidUpdate，这两对函数一前一后地把 render 函数夹在中间。

和装载过程不同的是，当在服务器端使用 React 渲染时，这一对函数中的 Did 函数，也就是 componentDidUpdate 函数，并不是只在浏览器端才执行的，无论更新过程发生在服务器端还是浏览器端，该函数都会被调用。

在介绍 componentDidMount 函数时，我们说到可以利用 componentDidMount 函数执行其他 UI 库的代码，比如 jQuery 代码。当 React 组件被更新时，原有的内容被重新绘制，这时候就需要在 componentDidUpdate 函数再次调用 jQuery 代码。

读者可能会问，componentDidUpdate 函数不是可能会在服务器端也被执行吗？在服务器端怎么能够使用 jQuery 呢？实际上，使用 React 做服务器端渲染时，基本不会经历更新过程，因为服务器端只需要产出 HTML 字符串，一个装载过程就足够产出 HTML 了，所以正常情况下服务器端不会调用 componentDidUpdate 函数，如果调用了，说明我们的程序有错误，需要改进。

2.3.3 卸载过程

React 组件的卸载过程只涉及一个函数 componentWillUnmount，当 React 组件要从 DOM 树上删除掉之前，对应的 componentWillUnmount 函数会被调用，所以这个函数适合做一些清理性的工作。

和装载过程与更新过程不一样，这个函数没有配对的 Did 函数，就一个函数，因为卸载完就完了，没有“卸载完再做的事情”。

不过，componentWillUnmount 中的工作往往和 componentDidMount 有关，比如，在 componentDidMount 中用非 React 的方法创造了一些 DOM 元素，如果撒手不管可能会造成内存泄露，那就需要在 componentWillUnmount 中把这些创造的 DOM 元素清理掉。

2.4 组件向外传递数据

通过构造 ControlPanel 和 Counter，现在我们已经知道如何通过 prop 从父组件传递

数据给子组件，但是，组件之间的交流是相互的，子组件某些情况下也需要把数据传递给父组件，我们接下来看看在 React 中如何实现这个功能。

在 ControlPanel 中，包含三个 Control 子组件实例，每个 Counter 都有一个可以动态改变的计数值，我们希望 ControlPanel 能够即时显示出这三个子组件当前计数值之和。

这个功能看起来很简单，但是要解决一个问题，就是要让 ControlPanel“知道”三个子组件当前的计数值，而且是每次变更都要立刻知道，而 Control 组件的当前值组件的内部状态，如何让外部世界知道这个值呢？

解决这个问题的方法，依然是利用 prop。组件的 prop 可以是任何 JavaScript 对象，而在 JavaScript 中，函数是一等公民，函数本身就可以被看做一种对象，既可以像其他对象一样作为 prop 的值从父组件传递给子组件，又可以被子组件作为函数调用，这样事情就好办了。

2.4.1　应用实例

展示这个功能的代码存在于 https://github.com/mocheng/react-and-redux/ 代码库的 chapter-02/controlpanel_with_summary 目录下，在界面上，可以看到效果如图 2-4 所示：

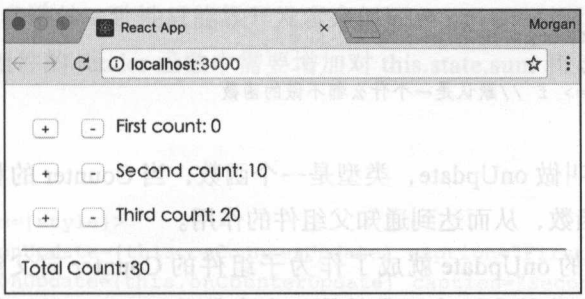


图 2-4　包含总数的 ControlPanel 应用效果图

点击任何一个 Counter 的“+”按钮或者“-”按钮，可以看见除了所属 Counter 的计数变化，底部的总计数也会随之变化，这是因为 Counter 能够把自己状态改变的信息传递给外层的组件。

接下来看实现这个功能的关键代码。

在 Counter 组件中，对于点击“+”和“-”按钮的事件处理方法做了改动，代码如下：

```
onClickIncrementButton() {
  this.updateCount(true);
}
```

```
onClickDecrementButton() {
  this.updateCount(false);
}

updateCount(isIncrement) {
  const previousValue = this.state.count;
  const newValue = isIncrement ? previousValue + 1 : previousValue - 1;
  this.setState({count: newValue})
  this.props.onUpdate(newValue, previousValue)
}
```

现在，onClickIncrementButton 函数和 onClickDecrementButton 的任务除了调用 this.setState 改变内部状态，还要调用 this.props.onUpdate 这个函数，为了避免重复代码，我们对原有代码做了一下重构，提取了共同部分到 updateCount 函数里。

对应的，Counter 组件的 propTypes 和 defaultProps 就要增加 onUpdate 的定义，代码如下：

```
Counter.propTypes = {
  caption: PropTypes.string.isRequired,
  initValue: PropTypes.number,
  onUpdate: PropTypes.func
};

Counter.defaultProps = {
  initValue: 0,
  onUpdate: f => f //默认是一个什么都不做的函数
};
```

新增加的 prop 叫做 onUpdate，类型是一个函数，当 Counter 的状态改变的时候，就会调用这个给定的函数，从而达到通知父组件的作用。

这样，Counter 的 onUpdate 就成了作为子组件的 Counter 向父组件 ControlPanel 传递数据的渠道，我们先约定这个函数的第一个参数是 Counter 更新之后的新值，第二个参数是更新之前的值，至于如何使用这两个参数的值，是父组件 ControlPanel 的逻辑，Counter 不用操心，而且根据两个参数的值足够可以推导出数值是增加还是减少。

从使用 Counter 组件的角度，在 ControlPanel 组件中也要做一些修改，现在 ControlPanel 需要包含自己的 state，首先是构造函数部分，代码如下：

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

  this.onCounterUpdate = this.onCounterUpdate.bind(this);

  this.initValues = [ 0, 10, 20];
  const initSum = this.initValues.reduce((a, b) => a+b, 0);
```

```
    this.state = {
      sum: initSum
    };
  }
```

在 ControlPanel 组件被第一次渲染的时候，就需要显示三个计数器数值的总和，所以我们在构造函数中用 initValues 数组记录所有的 Counter 的初始值，在初始化 this.state 之前，将 initValues 数组中所有值加在一起，作为 this.state 中 sum 字段的初始值。

ControlPanel 传递给 Counter 组件 onUpdate 这个 prop 的值是 onCounterUpdate 函数，代码如下：

```
onCounterUpdate(newValue, previousValue) {
  const valueChange = newValue - previousValue;
  this.setState({ sum: this.state.sum + valueChange});
}
```

onCounterUpdate 函数的参数和 Counter 中调用 onUpdate prop 的参数规格一致，第一个参数为新值，第二个参数为之前的值，两者之差就是改变值，将这个改变作用到 this.state.sum 上就是新的 sum 状态。

遗憾的是，React 虽然有 PropType 能够检查 prop 的类型，却没有任何机制来限制 prop 的参数规格，参数的一致性只能靠开发者来保证。

ControlPanel 组件的 render 函数中需要增加对 this.state.sum 和 onCounterUpdate 的使用，代码如下：

```
render() {
  return (
    <div style={style}>
    <Counter onUpdate={this.onCounterUpdate} caption="First" />
    <Counter onUpdate={this.onCounterUpdate} caption="Second" initValue={this.
      initValues[1]} />
        <Counter onUpdate={this.onCounterUpdate} caption="Third"
          initValue={this.initValues[2]} />
        <hr/>
      <div>Total Count: {this.state.sum}</div>
    </div>
    );
  }
}
```

2.5　React 组件 state 和 prop 的局限

是时候重新思考一下多个组件之间的数据管理问题了。在上面修改的代码中，不

难发现其实实现得并不精妙，每个 Counter 组件有自己的状态记录当前计数，而父组件 ControlPanel 也有一个状态来存储所有 Counter 计数总和，也就是说，数据发生了重复。

数据如果出现重复，带来的一个问题就是如何保证重复的数据一致，如果数据存多份而且不一致，那就很难决定到底使用哪个数据作为正确结果了。

在上面的例子中，ControlPanel 通过 onUpdate 回调函数传递的新值和旧值来计算新的计数总和，设想一下，由于某种 bug 的原因，某个按钮的点击更新没有通知到 ControlPanel，结果就会让 ControlPanel 中的 sum 状态和所有子组件 Counter 的 count 状态之和不一致，这时候，是应该相信 ControlPanel 还是 Counter 呢？

如图 2-5 所示，逻辑上应该相同的状态，分别存放在不同组件中，就会导致这种困局。

对于上面所说的问题，一个直观的解决方法是以某一个组件的状态为准，这个组件是状态的“领头羊”，其余组件都保持和“领头羊”的状态同步，但是在实际情况下这种方法可能难以实施。比如上面的例子中，每个 Counter 记录自己的计数值是很自然的，但是有三个 Counter 组件，也就有三只“领头羊”，让 ControlPanel 跟着三只“领头羊”走，似乎不是一个好主意。

另一种思路，就是干脆不要让任何一个 React 组件扮演“领头羊”的角色，把数据源放在 React 组件之外形成全局状态，如图 2-6 所示，让各个组件保持和全局状态的一致，这样更容易控制。

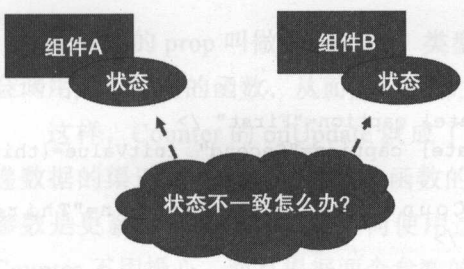


图 2-5 组件状态不一致的困惑

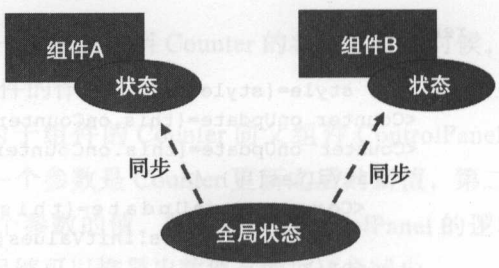


图 2-6 React 中提取出来

图 2-6 中所示，全局状态就是唯一可靠的数据源，在第 3 章中我们会介绍，这就是 Flux 和 Redux 中 Store 的概念。

除了 state，利用 prop 在组件之间传递信息也会遇到问题。设想一下，在一个应用中，包含三级或者三级以上的组件结构，顶层的祖父级组件想要传递一个数据给最低层的子组件，用 prop 的方式，就只能通过父组件中转，如图 2-6 所示。

也许中间那一层父组件根本用不上这个 prop，但是依然需要支持这个 prop，扮演好

搬运工的角色，只因为子组件用得上，这明显违反了低耦合的设计要求。第 3 章中我们会探讨如何解决这样的困局。

2.6　本章小结

在这一章中，我们学习了构建高质量组件的原则，应用 React 一样要以构建高内聚低耦合的组件为目标，而保证组件高质量的一个重要工作就是保持组件对外接口清晰简洁。

React 利用 prop 来定义组件的对外接口，用 state 来代表内部的状态，某个数据选择用 prop 还是用 state 表示，取决于这个数据是对外还是对内。

我们还介绍了 React 的生命周期，了解了装载过程、更新过程和卸载过程涉及的所有生命周期函数。

在本章中我们利用 CountrolPanel 和 Counter 两个组件演示了组件之间的通信方式，包括子组件向父组件传递信息的方式，同时也看出了使用 React 的 state 来存储状态的一个缺点，那就是数据的冗余和重复，这就是我们接下来要解决的问题。

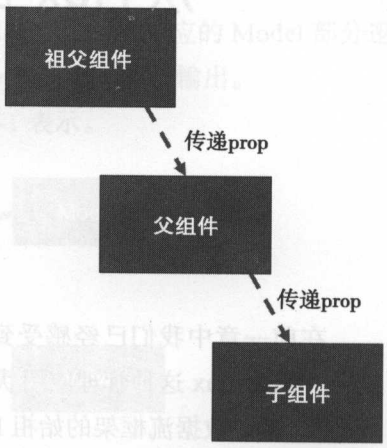
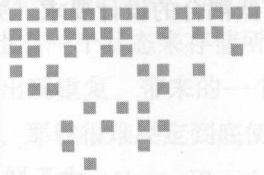


图 2-7　跨级传递 prop 的困局



Chapter 3 第 3 章

从 Flux 到 Redux

在前一章中我们已经感受到完全用 React 来管理应用数据的麻烦，在这一章中，我们将介绍 Redux 这种管理应用状态的框架，包含以下内容。

- 单向数据流框架的始祖 Flux；
- Flux 理念的一个更强实现 Redux；
- 结合 React 和 Redux。

3.1 Flux

要了解 Redux，首先要从 Flux 说起，可以认为 Redux 是 Flux 思想的另一种实现方式，通过了解 Flux，我们可以知道 Flux 一族框架（其中就包括 Redux）贯彻的最重要的观点——单向数据流，更重要的是，我们可以发现 Flux 框架的缺点，从而深刻地认识到 Redux 相对于 Flux 的改进之处。

让我们来看看 Flux 的历史，实际上，Flux 是和 React 同时面世的，在 2013 年，Facebook 公司让 React 亮相的同时，也推出了 Flux 框架，React 和 Flux 相辅相成，Facebook 认为两者结合在一起才能构建大型的 JavaScript 应用。

做一个容易理解的对比，React 是用来替换 jQuery 的，那么 Flux 就是以替换 Backbone.js、Ember.js 等 MVC 一族框架为目的。

在 MVC（Model-View-Controller）的世界里，React 相当于 V（也就是 View）的部分，只涉及页面的渲染一旦涉及应用的数据管理部分，还是交给 Model 和 Controller，不过，

Flux 并不是一个 MVC 框架，事实上，Flux 认为 MVC 框架存在很大问题，它推翻了 MVC 框架，并用一个新的思维来管理数据流转。

3.1.1　MVC 框架的缺陷

MVC 框架是业界广泛接受的一种前端应用框架类型，这种框架把应用分为三个部分：

- Model（模型）负责管理数据，大部分业务逻辑也应该放在 Model 中；
- View（视图）负责渲染用户界面，应该避免在 View 中涉及业务逻辑；
- Controller（控制器）负责接受用户输入，根据用户输入调用对应的 Model 部分逻辑，把产生的数据结果交给 View 部分，让 View 渲染出必要的输出。

MVC 框架的几个组成部分和请求的关系可以用图 3-1 表示。

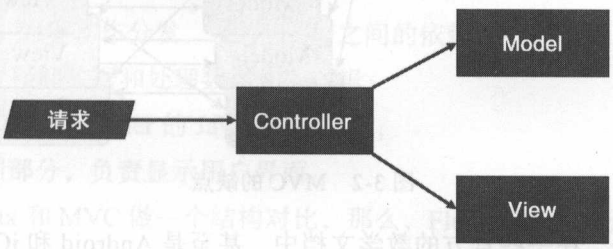


图 3-1　MVC 框架

这样的逻辑划分，实质上与把以一个应用划分为多个组件一样，就是“分而治之”。毫无疑问，相比把业务逻辑和界面渲染逻辑混在一起，MVC 框架要先进得多。这种方式得到了广泛认可，连 Facebook 最初也是用这种框架。

但是，Facebook 的工程部门逐渐发现，对于非常巨大的代码库和庞大的组织，按照他们的原话说就是“ MVC 真的很快就变得非常复杂”。每当工程师想要增加一个新的功能时，对代码的修改很容易引入新的 bug，因为不同模块之间的依赖关系让系统变得“脆弱而且不可预测”。对于刚刚加入团队的新手，更是举步维艰，因为不知道修改代码会造成什么样的后果。如果要保险，就会发现寸步难移；如果放手去干，有可能引发很多 bug。

一句话，MVC 根本不适合 Facebook 的需求。

为何被业界普遍认可的 MVC 框架在 Facebook 眼里却沦落到如此地步呢？

图 3-2 是 Facebook 描述的 MVC 框架，在图中，我们可以看到，Model 和 View 之间缠绕着蜘蛛网一样复杂的依赖关系，根据箭头的方向，我们知道有的是 Model 调用了 View，有的是 View 调用了 Model，好乱。

这就有意思了，怎么图 3-2 的画风和通常我们所描述的 MVC 框架画风很不一样呢？

MVC 框架提出的数据流很理想，用户请求先到达 Controller，由 Controller 调用 Model 获得数据，然后把数据交给 View，但是，在实际框架实现中，总是允许 View 和 Model 可以直接通信，从而出现图 3-2 的情况。

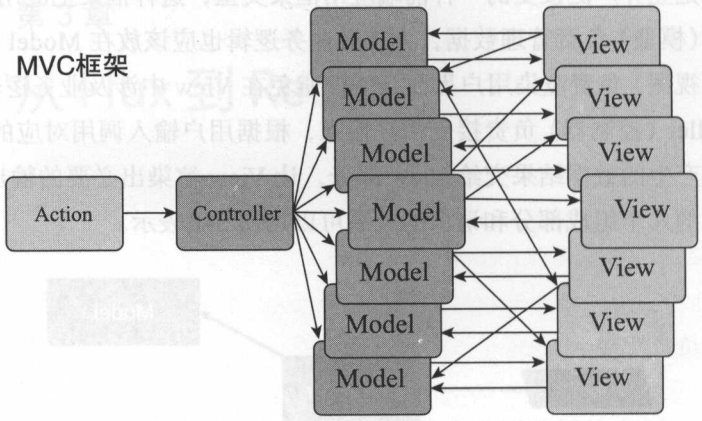


图 3-2 MVC 的缺点

非常遗憾的是，在一些官方的教学文档中，甚至是 Android 和 iOS 的教学文档中的例子中，也会出现 View 和 Model 直接通信的例子。不过这种状况逐渐在改变，因为越来越多的同行发现，在 MVC 中让 View 和 Model 直接对话就是灾难。

当我向以前没接触过 Flux 的朋友介绍 Flux 的时候，发现了一个有意思的现象。凡是只在服务器端使用过 MVC 框架的朋友，就很容易理解和接受 Flux。而对于已经有很多浏览器端 MVC 框架经验的朋友，往往还要费一点劲才能明白 MVC 和 Flux 的差异。

造成这种认知差别的主要原因，就是服务器端 MVC 框架往往就是每个请求就只在 Controller-Model-View 三者之间走一圈，结果就返回给浏览器去渲染或者其他处理了，然后这个请求生命周期的 Controller-Model-View 就可以回收销毁了，这是一个严格意义的单向数据流；对于浏览器端 MVC 框架，存在用户的交互处理，界面渲染出来之后，Model 和 View 依然存在于浏览器中，这时候就会诱惑开发者为了简便，让现存的 Model 和 View 直接对话。

对于 MVC 框架，为了让数据流可控，Controller 应该是中心，当 View 要传递消息给 Model 时，应该调用 Controller 的方法，同样，当 Model 要更新 View 时，也应该通过 Controller 引发新的渲染。

当 Facebook 推出 Flux 时，招致了很多质疑。很多人都说，Flux 只不过是一个对数据流管理更加严格的 MVC 框架而已。这种说法不完全准确，但是一定意义上也说明了

Flux 的一个特点：更严格的数据流控制。

Facebook 已经无心在 MVC 框架上纠缠，他们用 Flux 框架来代替原有的 MVC 框架，他们提出的 Flux 框架大致结构是图 3-3 的模样。

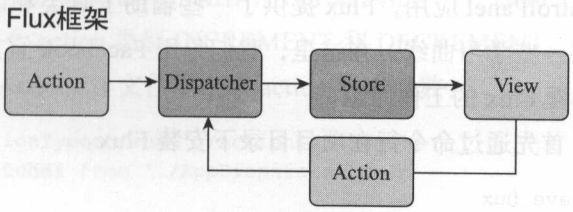


图 3-3　Flux 的单向数据流

一个 Flux 应用包含四个部分，我们先粗略了解一下：

- Dispatcher，处理动作分发，维持 Store 之间的依赖关系；
- Store，负责存储数据和处理数据相关逻辑；
- Action，驱动 Dispatcher 的 JavaScript 对象；
- View，视图部分，负责显示用户界面。

如果非要把 Flux 和 MVC 做一个结构对比，那么，Flux 的 Dispatcher 相当于 MVC 的 Controller，Flux 的 Store 相当于 MVC 的 Model，Flux 的 View 当然就对应 MVC 的 View 了，至于多出来的这个 Action，可以理解为对应给 MVC 框架的用户请求。

在 MVC 框架中，系统能够提供什么样的服务，通过 Controller 暴露函数来实现。每增加一个功能，Controller 往往就要增加一个函数；在 Flux 的世界里，新增加功能并不需要 Dispatcher 增加新的函数，实际上，Dispatcher 自始至终只需要暴露一个函数 Dispatch，当需要增加新的功能时，要做的是增加一种新的 Action 类型，Dispatcher 的对外接口并不用改变。

当需要扩充应用所能处理的“请求”时，MVC 方法就需要增加新的 Controller，而对于 Flux 则只是增加新的 Action。

我们已经基本了解了 Flux 是怎么一回事，接下来就要实践，看看怎么用 Flux 改进一下我们的 React 应用。

3.1.2　Flux 应用

本书讲的是 Redux，但是为什么这里要先介绍 Flux，甚至要用 Flux 来实现一个例子呢？

因为 Redux 其实和 Flux 一脉相承，Redux 出现之后，连 Flux 的创建者都说“喜欢

你创造的 Redux”，Redux 的创建者 Dan Abramov，现在也是在为 Facebook 的 React 库工作。

从 Flux 的例子入手，在理解 Redux 的时候就会感觉非常顺畅。让我们改进一下前面章节中创造的 ControlPanel 应用，Flux 提供了一些辅助工具类和函数，能够帮助创建 Flux 应用，但是需要一些学习曲线。在这里，我们只用 Facebook 官方的基本功能，目的是为了更清晰地看一看 Flux 的工作原理。

为了使用 Flux，首先通过命令行在项目目录下安装 Flux。

```
npm install --save flux
```

利用 Flux 来实现 ControlPanel 应用的相关代码在 https://github.com/mocheng/react-and-redux/tree/master/chapter-03/flux 上可以找到，最终应用界面效果和第 2 章中创造的应用完全一样，通过同一界面不同实现方式的比对，我们可以体会每个方式的优劣。

1. Dispatcher

首先，我们要创造一个 Dispatcher，几乎所有应用都只需要拥有一个 Dispatcher，对于我们这个简单的应用更不例外。在 src/AppDispatcher.js 中，我们创造这个唯一的 Dispatcher 对象，代码如下：

```
import {Dispatcher} from 'flux';

export default new Dispatcher();
```

非常简单，我们引入 flux 库中的 Dispatcher 类，然后创造一个新的对象作为这个文件的默认输出就足够了。在其他代码中，将会引用这个全局唯一的 Dispatcher 对象。

Dispatcher 存在的作用，就是用来派发 action，接下来我们就来定义应用中涉及的 action。

2. action

action 顾名思义代表一个“动作”，不过这个动作只是一个普通的 JavaScript 对象，代表一个动作的纯数据，类似于 DOM API 中的事件（event）。甚至，和事件相比，action 其实还是更加纯粹的数据对象，因为事件往往还包含一些方法，比如点击事件就有 preventDefault 方法，但是 action 对象不自带方法，就是纯粹的数据。

作为管理，action 对象必须有一个名为 type 的字段，代表这个 action 对象的类型，为了记录日志和 debug 方便，这个 type 应该是字符串类型。

定义 action 通常需要两个文件，一个定义 action 的类型，一个定义 action 的构造函数（也称为 action creator）。分成两个文件的主要原因是在 Store 中会根据 action 类型做不同操作，也就有单独导入 action 类型的需要。

在 src/ActionTypes.js 中，我们定义 action 的类型，代码如下：

```
export const INCREMENT = 'increment';
export const DECREMENT = 'decrement';
```

在这个例子中，用户只能做两个动作，一个是点击“+”按钮，一个是点击“-”按钮，所以我们只有两个 action 类型 INCREMENT 和 DECREMENT。

现在我们在 src/Actions.js 文件中定义 action 构造函数：

```
import * as ActionTypes from './ActionTypes.js';
import AppDispatcher from './AppDispatcher.js';

export const increment = (counterCaption) => {
  AppDispatcher.dispatch({
    type: ActionTypes.INCREMENT,
    counterCaption: counterCaption
  });
};

export const decrement = (counterCaption) => {
  AppDispatcher.dispatch({
    type: ActionTypes.DECREMENT,
    counterCaption: counterCaption
  });
};
```

虽然出于业界习惯，这个文件被命名为 Actions.js，但是要注意里面定义的并不是 action 对象本身，而是能够产生并派发 action 对象的函数。

在 Actions.js 文件中，引入了 ActionTypes 和 AppDispatcher，看得出来是要直接使用 Dispatcher。

这个 Actions.js 导出了两个 action 构造函数 increment 和 decrement，当这两个函数被调用的时候，创造了对应的 action 对象，并立即通过 AppDispatcher.dispatch 函数派发出去。

派发出去的 action 对象最后怎么样了呢？在下面关于 Store 的部分可以看到。

3. Store

一个 Store 也是一个对象，这个对象存储应用状态，同时还要接受 Dispatcher 派发的动作，根据动作来决定是否要更新应用状态。

接下来我们创造 Store 相关的代码，因为使用 Flux 之后代码文件数量会增多，再把所有源代码文件都放在 src 目录下就不容易管理了。所以我们在 src 下创建一个子目录 stores，在这个子目录里面放置所有的 Store 代码。

在前面章节的 ControlPanel 应用例子里，有三个 Counter 组件，还有一个统计三个 Counter 计数值之和的功能，我们遇到的麻烦就是这两者之间的状态如何同步的问题，现

在，我们创造两个 Store，一个是为 Counter 组件服务的 CounterStore，另一个就是为总数服务的 SummaryStore。

我们首先添加 CounterStore，放在 src/stores/CounterStore.js 文件中。

在本书中我们避免了使用 Flux 库的任何辅助类或者方法，虽然在 flux/utils 中提供了一些辅助类方便 Store 的开发，但是我们并不使用这些辅助类，因为学习这些辅助类需要一些学习曲线，我们选择原始但是简单的方法来构建 Store，这样能够更清楚地看到 Store 的工作原理。

先看定义 CounterStore 的代码，如下所示：

```
const counterValues = {
  'First': 0,
  'Second': 10,
  'Third': 30
};

const CounterStore = Object.assign({}, EventEmitter.prototype, {
  getCounterValues: function() {
    return counterValues;
  },
  emitChange: function() {
    this.emit(CHANGE_EVENT);
  },
  addChangeListener: function(callback) {
    this.on(CHANGE_EVENT, callback);
  },
  removeChangeListener: function(callback) {
    this.removeListener(CHANGE_EVENT, callback);
  }
});
```

当 Store 的状态发生变化的时候，需要通知应用的其他部分做必要的响应。在我们的应用中，做出响应的部分当然就是 View 部分，但是我们不应该硬编码这种联系，应该用消息的方式建立 Store 和 View 的联系。这就是为什么我们让 CounterStore 扩展了 EventEmitter.prototype，等于让 CounterStore 成了 EventEmitter 对象，一个 EventEmitter 实例对象支持下列相关函数。

- emit 函数，可以广播一个特定事件，第一个参数是字符串类型的事件名称；
- on 函数，可以增加一个挂在这个 EventEmitter 对象特定事件上的处理函数，第一个参数是字符串类型的事件名称，第二个参数是处理函数；
- removeListener 函数，和 on 函数做的事情相反，删除挂在这个 EventEmitter 对象

特定事件上的处理函数，和 on 函数一样，第一个参数是事件名称，第二个参数是处理函数。要注意，如果要调用 removeListener 函数，就一定要保留对处理函数的引用。

对于 CounterStore 对象，emitChange、addChangeListener 和 removeChangeListener 函数就是利用 EventEmitter 上述的三个函数完成对 CounterStore 状态更新的广播、添加监听函数和删除监听函数等操作。

CounterStore 函数还提供一个 getCounterValues 函数，用于让应用中其他模块可以读取当前的计数值，当前的计数值存储在文件模块级的变量 counterValues 中。

注意　严格来说，getCounterValues 这样的 getter 函数，应该返回一个不可改变的（Immutable）数据，这样，调用者即使通过 getCounterValues 获得了当前计数值对象，也不能够修改这个对象从而扰乱其他代码的使用。但是，为了简单起见，这个例子中我们并不使用 Immutable，只是在写代码时要注意，不应该去修改通过 Store 得到的数据。

上面实现的 Store 只有注册到 Dispatcher 实例上才能真正发挥作用，所以还需要增加下列代码：

```
import AppDispatcher from '../AppDispatcher.js';

CounterStore.dispatchToken = AppDispatcher.register((action) => {
  if (action.type === ActionTypes.INCREMENT) {
    counterValues[action.counterCaption] ++;
    CounterStore.emitChange();
  } else if (action.type === ActionTypes.DECREMENT) {
    counterValues[action.counterCaption] --;
    CounterStore.emitChange();
  }
});
```

这是最重要的一个步骤，要把 CounterStore 注册到全局唯一的 Dispatcher 上去。Dispatcher 有一个函数叫做 register，接受一个回调函数作为参数。返回值是一个 token，这个 token 可以用于 Store 之间的同步，我们在 CounterStore 中还用不上这个返回值，在稍后的 SummaryStore 中会用到，现在我们只是把 register 函数的返回值保存在 CounterStore 对象的 dispatchToken 字段上，待会就会用得到。

现在我们来仔细看看 register 接受的这个回调函数参数，这是 Flux 流程中最核心的部分，当通过 register 函数把一个回调函数注册到 Dispatcher 之后，所有派发给 Dispatcher 的 action 对象，都会传递到这个回调函数中来。

比如通过 Dispatcher 派发一个动作，代码如下：

```
AppDispatcher.dispatch({
  type: ActionTypes.INCREMENT,
  counterCaption: 'First'
});
```

那在 CounterStore 注册的回调函数就会被调用，唯一的一个参数就是那个 action 对象，回调函数要做的，就是根据 action 对象来决定改如何更新自己的状态。

作为一个普遍接受的传统，action 对象中必有一个 type 字段，类型是字符串，用于表示这个 action 对象是什么类型，比如上面派发的 action 对象，type 为“increment”，表示是一个计数器的“加一”的动作；如果有必要，一个 action 对象还可以包含其他的字段。上面的 action 对象中还有一个 counterCaption 字段值为“First”，标识名字为“First”的计数器。

在我们的例子中，action 对象的 type 和 counter Caption 字段结合在一起，可以确定是哪个计数器应该做加一或者减一的动作，上面例子中的动作含义就是：“名字为 First 的计数器要做加一动作。”

根据不同的 type，会有不同的操作，所以注册的回调函数很自然有一个模式，就是函数体是一串 if-else 条件语句或者 switch 条件语句，而条件语句的跳转条件，都是针对参数 action 对象的 type 字段：

```
CounterStore.dispatchToken = AppDispatcher.register((action) => {
  if (action.type === ActionTypes.INCREMENT) {
    counterValues[action.counterCaption] ++;
    CounterStore.emitChange();
  } else if (action.type === ActionTypes.DECREMENT) {
    counterValues[action.counterCaption] --;
    CounterStore.emitChange();
  }
});
```

在上面的代码例子中，如果 action.type 是 INCREMENT，就根据 action 对象字段 counterCaption 确定是哪个计数器，把 counterValues 上对应的字段做加一操作；同样，如果发现 action.type 代表 DECREMENT，就做对应的减一的操作。

无论是加一或者减一，最后都要调用 ounterStore.emitChange 函数，假如有调用者通过 Counter.addChangeListner 关注了 CounterStore 的状态变化，这个 emitChange 函数调用就会引发监听函数的执行。

目前，CounterStore 只关注 INCREMENT 和 DECREMENT 动作，所以 if-else 判断也只关照了这两种类型的动作，除此之外，其他 action 对象一律忽略。

接下来，我们再来看看另一个 Store，也就是代表所有计数器计数值总和的 Store，在

src/stores/SummaryStore.js 中有源代码。

SummaryStore 也有 emitChange、addChangeListener 还有 removeChangeListener 函数，功能一样也是用于通知监听者状态变化，这几个函数的代码和 CounterStore 中完全重复，不同点是对获取状态函数的定义，代码如下：

```
function computeSummary(counterValues) {
  let summary = 0;
  for (const key in counterValues) {
    if (counterValues.hasOwnProperty(key)) {
      summary += counterValues[key];
    }
  }
  return summary;
}

const SummaryStore = Object.assign({}, EventEmitter.prototype, {
  getSummary: function() {
    return computeSummary(CounterStore.getCounterValues());
  },
});
```

可以注意到，SummaryStore 并没有存储自己的状态，当 getSummary 被调用时，它是直接从 CounterStore 里获取状态计算的。

CounterStore 提供了 getCounterValues 函数让其他模块能够获得所有计数器的值，SummaryStore 也提供了 getSummary 让其他模块可以获得所有计数器当前值的总和。不过，既然总可以通过 CounterStore.getCounterValues 函数获取最新鲜的数据，SummaryStore 似乎也就没有必要把计数器当前值总和存储到某个变量里。事实上，可以看到 SummaryStore 并不像 CounterStore 一样用一个变量 counterValues 存储数据，SummaryStore 不存储数据，而是每次对 getSummary 的调用，都实时读取 CounterStore.getCounterValues，然后实时计算出总和返回给调用者。

可见，虽然名为 Store，但并不表示一个 Store 必须要存储什么东西，Store 只是提供获取数据的方法，而 Store 提供的数据完全可以另一个 Store 计算得来。

SummaryStore 在 Dispatcher 上注册的回调函数也和 CounterStore 很不一样，代码如下：

```
SummaryStore.dispatchToken = AppDispatcher.register((action) => {
  if ((action.type === ActionTypes.INCREMENT) ||
      (action.type === ActionTypes.DECREMENT)) {
    AppDispatcher.waitFor([CounterStore.dispatchToken]);
    SummaryStore.emitChange();
  }
});
```

SummaryStore 同样也通过 AppDispatcher.register 函数注册一个回调函数，用于接受派发的 action 对象。在回调函数中，也只关注 INCREMENT 和 DECREMENT 类型的 action 对象，并通过 emitChange 通知监听者，注意在这里使用了 waitFor 函数，这个函数解决的是下面描述的问题。

既然一个 action 对象会被派发给所有回调函数，这就产生了一个问题，到底是按照什么顺序调用各个回调函数呢？

即使 Flux 按照 register 调用的顺序去调用各个回调函数，我们也完全无法把握各个 Store 哪个先装载从而调用 register 函数。所以，可以认为 Dispatcher 调用回调函数的顺序完全是无法预期的，不要假设它会按照我们期望的顺序逐个调用。

设想一下，当一个 INCREMENT 类型的动作被派发了，如果首先调用 SummaryStore 的回调函数，在这个回调函数中立即用 emitChange 通知了监听者，这时监听者会立即通过 SummaryStore 的 getSummary 获取结果，而这个 getSummary 是通过 CounterStore 暴露的 getCounterValues 函数获取当前计数器值，计算出总和返回……然而，这时候，INCREMENT 动作还没来得及派发到 CounterStore 啊！也就是说，CounterStore 的 getCounterValues 返回的还是一个未更新的值，那样 SummaryStore 的 getSummary 返回值也就是一个错误的值了。

怎么解决这个问题呢？这就要靠 Dispatcher 的 waitFor 函数了。在 SummaryStore 的回调函数中，之前在 CounterStore 中注册回调函数时保存下来的 dispatchToken 终于派上了用场。

Dispatcher 的 waitFor 可以接受一个数组作为参数，数组中每个元素都是一个 Dispatcherregister 函数的返回结果，也就所谓的 dispatchToken。这个 waitFor 函数告诉 Dispatcher，当前的处理必须要暂停，直到 dispatchToken 代表的那些已注册回调函数执行结束才能继续。

我们知道，JavaScript 是单线程的语言，不可能有线程之间的等待这回事，这个 waitFor 函数当然并不是用多线程实现的，只是在调用 waitFor 的时候，把控制权交给 Dispatcher，让 Dispatcher 检查一下 dispatchToken 代表的回调函数有没有被执行，如果已经执行，那就直接接续，如果还么有执行，那就调用 dispatchToken 代表的回调函数之后 waitFor 才返回。

回到我们上面假设的例子，即使 SummaryStore 比 CounterStore 提前接收到了 action 对象，在 emitChange 中调用 waitFor，也就能够保证在 emitChange 函数被调用的时候，CounterStore 也已经处理过这个 action 对象，一切完美解决。

这里要注意一个事实，Dispatcher 的 register 函数，只提供了注册一个回调函数的功

能，但却不能让调用者在 register 时选择只监听某些 action，换句话说，每个 register 的调用者只能这样请求："当有任何动作被派发时，请调用我。"但不能够这么请求："当这种类型还有那种类型的动作被派发的时候，请调用我。"

当一个动作被派发的时候，Dispatcher 就是简单地把所有注册的回调函数全都调用一遍，至于这个动作是不是对方关心的，Flux 的 Dispatcher 不关心，要求每个回调函数去鉴别。

看起来，这似乎是一种浪费，但是这个设计让 Flux 的 Dispatcher 逻辑最简单化，Dispatcher 的责任越简单，就越不会出现问题。毕竟，由回调函数全权决定如何处理 action 对象，也是非常合理的。

4. View

首先需要说明，Flux 框架下，View 并不是说必须要使用 React，View 本身是一个独立的部分，可以用任何一种 UI 库来实现。

不过，话说回来，既然我们都使用上 Flux 了，除非项目有大量历史遗留代码需要利用，否则实在没有理由不用 React 来实现 View。

存在于 Flux 框架中的 React 组件需要实现以下几个功能：

- 创建时要读取 Store 上状态来初始化组件内部状态；
- 当 Store 上状态发生变化时，组件要立刻同步更新内部状态保持一致；
- View 如果要改变 Store 状态，必须而且只能派发 action。

最后让我们来看看例子中的 View 部分，为了方便管理，所有的 View 文件都放在 src/views 目录里。

先看 src/views/ControlPanel.js 中的 ControlPanel 组件，其中 render 函数的实现和上一章很不一样，代码如下：

```
render() {
  return (
    <div style={style}>
      <Counter caption="First" />
      <Counter caption="Second" />
      <Counter caption="Third" />
      <hr/>
      <Summary />
    </div>
  );
}
```

可以注意到，和以前面章节中的 ControlPanel 不同，Counter 组件实例只有 caption 属性，没有 initValue 属性。因为我们把计数值包括初始值全都放到 CounterStore 中去了，

所以在创造 Counter 组件实例的时候就没必要指定 initValue 了。

接着看 src/views/Counter.js 中定义的 Counter 组件，构造函数中初始化 this.state 的方式有了变化，代码如下：

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

  this.onChange = this.onChange.bind(this);
  this.onClickIncrementButton = this.onClickIncrementButton.bind(this);
  this.onClickDecrementButton = this.onClickDecrementButton.bind(this);

  this.state = {
    count: CounterStore.getCounterValues()[props.caption]
  }
}
```

在构造函数中，CounterStore.getCounterValues 函数获得了所有计数器的当前值，然后把 this.state 初始化为对应 caption 字段的值，也就是说 Counter 组件的 store 来源不再是 prop，而是 Flux 的 Store。

Counter 组件中的 state 应该成为 Flux Store 上状态的一个同步镜像，为了保持两者一致，除了在构造函数中的初始化之外，在之后当 CounterStore 上状态变化时，Counter 组件也要对应变化，相关代码如下：

```
componentDidMount() {
  CounterStore.addChangeListener(this.onChange);
}

componentWillUnmount() {
  CounterStore.removeChangeListener(this.onChange);
}

onChange() {
  const newCount = CounterStore.getCounterValues()[this.props.caption];
  this.setState({count: newCount});
}
```

如上面的代码所示，在 componentDidMount 函数中通过 CounterStore.addChangeListener 函数监听了 CounterStore 的变化之后，只要 CounterStore 发生变化，Counter 组件的 onChange 函数就会被调用。与 componentDidMount 函数中监听事件相对应，在 componentWillUnmount 函数中删除了这个监听。

接下来，要看 React 组件如何派发 action，代码如下：

```
onClickIncrementButton() {
  Actions.increment(this.props.caption);
```

```
  }
  onClickDecrementButton() {
    Actions.decrement(this.props.caption);
  }
  render() {
    const {caption} = this.props;
    return (
      <div>
        <button style={buttonStyle} onClick={this.onClickIncrementButton}> +</
          button>
        <button style={buttonStyle} onClick={this.onClickDecrement-Button}>-</
          button>
        <span>{caption} count: {this.state.count}</span>
      </div>
    );
  }
}
```

可以注意到，在 Counter 组件中有两处用到 CounterStore 的 getCounterValues 函数的地方，第一处是在构造函数中初始化 this.state 的时候，第二处是在响应 CounterStore 状态变化的 onChange 函数中，同样一个 Store 的状态，为了转换为 React 组件的状态，有两次重复的调用，这看起来似乎不是很好。但是，React 组件的状态就是这样，在构造函数中要对 this.state 初始化，要更新它就要调用 this.setState 函数。

有没有更简洁的方法？比如说只使用 CounterStore.getCounterValues 一次？可惜，只要我们想用组件的状态来驱动组件的渲染，就不可避免要有这两步。那么如果我们不利用组件的状态呢？

如果不使用组件的状态，那么我们就可以逃出这个必须在代码中使用 Store 两次的宿命，在接下里的章节里，我们会遇到这种“无状态”组件。

Summary 组件，存在于 src/views/Summary.js 中，和 Counter 类似，在 constructor 中初始化组件状态，通过在 componetDidMount 中添加对 SummaryStore 的监听来同步状态，因为这个 View 不会有任何交互功能，所以没有派发出任何 action。

3.1.3 Flux 的优势

本章的例子和上一章我们只用 React 的实现效果一样，但是工作方式有了大变化。

回顾一下完全只使用 React 实现的版本，应用的状态数据只存在于 React 组件之中，每个组件都要维护驱动自己渲染的状态数据，单个组件的状态还好维护，但是如果多个组件之间的状态有关联，那就麻烦了。比如 Counter 组件和 Summary 组件，Summary 组件需要维护所有 Counter 组件计数值的总和，Counter 组件和 Summary 分别维护自己的

状态，如何同步 Summary 和 Counter 状态就成了问题，React 只提供了 props 方法让组件之间通信，组件之间关系稍微复杂一点，这种方式就显得非常笨拙。

Flux 的架构下，应用的状态被放在了 Store 中，React 组件只是扮演 View 的作用，被动根据 Store 的状态来渲染。在上面的例子中，React 组件依然有自己的状态，但是已经完全沦为 Store 组件的一个映射，而不是主动变化的数据。

在完全只用 React 实现的版本里，用户的交互操作，比如点击“+”按钮，引发的时间处理函数直接通过 this.setState 改变组件的状态。在 Flux 的实现版本里，用户的操作引发的是一个“动作”的派发，这个派发的动作会发送给所有的 Store 对象，引起 Store 对象的状态改变，而不是直接引发组件的状态改变。因为组件的状态是 Store 状态的映射，所以改变了 Store 对象也就触发了 React 组件对象的状态改变，从而引发了界面的重新渲染。

Flux 带来了哪些好处呢？最重要的就是“单向数据流”的管理方式。

在 Flux 的理念里，如果要改变界面，必须改变 Store 中的状态，如果要改变 Store 中的状态，必须派发一个 action 对象，这就是规矩。在这个规矩之下，想要追溯一个应用的逻辑就变得非常容易。

我们已经讨论过 MVC 框架的缺点，MVC 最大的问题就是无法禁绝 View 和 Model 之间的直接对话，对应于 MVC 中 View 就是 Flux 中的 View，对应于 MVC 中的 Model 的就是 Flux 中的 Store，在 Flux 中，Store 只有 get 方法，没有 set 方法，根本可能直接去修改其内部状态，View 只能通过 get 方法获取 Store 的状态，无法直接去修改状态，如果 View 想要修改 Store 状态的话，只有派发一个 action 对象给 Dispatcher。

这看起来是一个“限制”，但却是一个很好的“限制”，禁绝了数据流混乱的可能。

简单说来，在 Flux 的体系下，驱动界面改变始于一个动作的派发，别无他法。

3.1.4 Flux 的不足

任何工具不可能只有优点没有缺点，接下来让我们看看 Flux 的不足之处，只有了解了 Flux 的不足之处，才能理解为什么会出现 Flux 的改进框架 Redux。

1. Store 之间依赖关系

在 Flux 的体系中，如果两个 Store 之间有逻辑依赖关系，就必须用上 Dispatcher 的 waitFor 函数。在上面的例子中我们已经使用过 waitFor 函数，SummaryStore 对 action 类型的处理，依赖于 CounterStore 已经处理过了。所以，必须要通过 waitFor 函数告诉 Dispatcher，先让 CounterStore 处理这些 action 对象，只有 CounterStore 搞定之后 SummaryStore 才继续。

那么，SummaryStore 如何标识 CounterStore 呢？靠的是 register 函数的返回值 dispatchToken，而 dispatchToken 的产生，当然是 CounterStore 控制的，换句话说，要这样设计：

- CounterStore 必须要把注册回调函数时产生的 dispatchToken 公之于众；
- SummaryStore 必须要在代码里建立对 CounterStore 的 dispatchToken 的依赖。

虽然 Flux 这个设计的确解决了 Store 之间的依赖关系，但是，这样明显的模块之间的依赖，看着还是让人感觉不大舒服，毕竟，最好的依赖管理是根本不让依赖产生。

2. 难以进行服务器端渲染

关于服务器端渲染，我们在后面第 12 章“同构”中会详细介绍，在这里，我们只需要知道，如果要在服务器端渲染，输出不是一个 DOM 树，而是一个字符串，准确来说就是一个全是 HTML 的字符串。

在 Flux 的体系中，有一个全局的 Dispatcher，然后每一个 Store 都是一个全局唯一的对象，这对于浏览器端应用完全没有问题，但是如果放在服务器端，就会有大问题。

和一个浏览器网页只服务于一个用户不同，在服务器端要同时接受很多用户的请求，如果每个 Store 都是全局唯一的对象，那不同请求的状态肯定就乱套了。

并不是说 Flux 不能做服务器端渲染，只是说让 Flux 做服务器端渲染很困难，实际上，Facebook 也说的很清楚，Flux 不是设计用作服务器端渲染的，他们也从来没有尝试过把 Flux 应用于服务器端。

3. Store 混杂了逻辑和状态

Store 封装了数据和处理数据的逻辑，用面向对象的思维来看，这是一件好事，毕竟对象就是这样定义的。但是，当我们需要动态替换一个 Store 的逻辑时，只能把这个 Store 整体替换掉，那也就无法保持 Store 中存储的状态。

读者可能会问，有什么使用场景是要替换 Store 呢？

在开发模式下，开发人员要不停地对代码进行修改，如果 Store 在某个状态下引发了 bug，如果能在不毁掉状态的情况下替换 Store 的逻辑，那就最好了，开发人员就可以不断地改进逻辑来验证这个状态下 bug 是否被修复了。

还有一些应用，在生产环境下就要根据用户属性来动态加载不同的模块，而且动态加载模块还希望不要网页重新加载，这时候也希望能够在不修改应用状态的前提下重新加载应用逻辑，这就是热加载 (Hot Load)，在第 12 章会详细介绍如何实现热加载。

可能读者觉得这里所说的“偷梁换柱”一样的替换应用逻辑是不可能做到的。实际上，真的能够做到，Redux 就能做到，所以让我们进入 Redux 的世界吧。

3.2 Redux

这本书的主题是关于 Redux 的，所以我们不要停留在 Flux 上太久。终于，我们要开始接触 Redux 了。

我们把 Flux 看作一个框架理念的话，Redux 是 Flux 的一种实现，除了 Redux 之外，还有很多实现 Flux 的框架，比如 Reflux、Fluxible 等，毫无疑问 Redux 获得的关注最多，这不是偶然的，因为 Redux 有很多其他框架无法比拟的优势。

这里我要强调，本书重点介绍 Redux，绝不是要贬抑其他框架。每个框架都有其优点也都有其缺点，作为开发者，只有兼容并蓄，才能够站得高，看得远。

3.2.1 Redux 的基本原则

2013 年问世的 Flux 饱受争议，而 2015 年 Dan Abramov 提出了在 Flux 基础上的改进框架 Redux，则是一鸣惊人，在所有 Flux 的变体中算是最受关注的框架，没有之一。

Flux 的基本原则是“单向数据流”，Redux 在此基础上强调三个基本原则：

- 唯一数据源（Single Source of Truth）；
- 保持状态只读（State is read-only）；
- 数据改变只能通过纯函数完成（Changes are made with pure functions）。

让我们逐一解释这三条基本原则。

1. 唯一数据源

唯一数据源指的是应用的状态数据应该只存储在唯一的一个 Store 上。

我们已经知道，在 Flux 中，应用可以拥有多个 Store，往往根据功能把应用的状态数据划分给若干个 Store 分别存储管理。比如，在上面的 ControlPanel 例子中，我们创造了 CounterStore 和 SummaryStore。

如果状态数据分散在多个 Store 中，容易造成数据冗余，这样数据一致性方面就会出问题。虽然利用 Dispatcher 的 waitFor 方法可以保证多个 Store 之间的更新顺序，但是这又产生了不同 Store 之间的显示依赖关系，这种依赖关系的存在增加了应用的复杂度，容易带来新的问题。

Redux 对这个问题的解决方法就是，整个应用只保持一个 Store，所有组件的数据源就是这个 Store 上的状态。

注意 Redux 并没有阻止一个应用拥有多个 Store，只是，在 Redux 的框架下，让一个应用拥有多个 Store 不会带来任何好处，最后还不如使用一个 Store 更容易组织代码。

这个唯一 Store 上的状态，是一个树形的对象，每个组件往往只是用树形对象上一部分的数据，而如何设计 Store 上状态的结构，就是 Redux 应用的核心问题，我们接下来会描述具体细节。

2. 保持状态只读

保持状态只读，就是说不能去直接修改状态，要修改 Store 的状态，必须要通过派发一个 action 对象完成，这一点，和 Flux 的要求并没有什么区别。

如果只看这个原则的字面意思，可能会让读者感觉有点费解，还记得那个公式吗？UI=render(state)，我们已经说过驱动用户界面更改的是状态，如果状态都是只读的不能修改，怎么可能引起用户界面的变化呢？

当然，要驱动用户界面渲染，就要改变应用的状态，但是改变状态的方法不是去修改状态上值，而是创建一个新的状态对象返回给 Redux，由 Redux 完成新的状态的组装。

这就直接引出了下面的第三个基本原则。

3. 数据改变只能通过纯函数完成

这里所说的纯函数就是 Reducer，Redux 这个名字的前三个字母 Red 代表的就是 Reducer。按照创作者 Dan Abramov 的说法，Redux 名字的含义是 Reducer+Flux。

Reducer 不是一个 Redux 特定的术语，而是一个计算机科学中的通用概念，很多语言和框架都有对 Reducer 函数的支持。就以 JavaScript 为例，数组类型就有 reduce 函数，接受的参数就是一个 reducer，reduce 做的事情就是把数组所有元素依次做“规约”，对每个元素都调用一次参数 reducer，通过 reducer 函数完成规约所有元素的功能。

下面是一个使用 reducer 函数的例子：

```
[1, 2, 3, 4].reduce(function reducer(accumulation, item) {
  return accumulation + item
}, 0);
```

上面的代码中，reducer（注意不是 reduce）函数接受两个参数，第一个参数是上一次规约的结果，第二个参数是这一次规约的元素，函数体是返回两者之和，所以这个规约的结果就是所有元素之和。

在 Redux 中，每个 reducer 的函数签名如下所示：

```
reducer(state, action)
```

第一个参数 state 是当前的状态，第二个参数 action 是接收到的 action 对象，而 reducer 函数要做的事情，就是根据 state 和 action 的值产生一个新的对象返回，注意 reducer 必须是纯函数，也就是说函数的返回结果必须完全由参数 state 和 action 决定，而且不产生任何副

作用，也不能修改参数 state 和 action 对象。

让我们回顾一下 Flux 中的 Store 是如何处理函数的，代码如下：

```
CounterStore.dispatchToken = AppDispatcher.register((action) => {
  if (action.type === ActionTypes.INCREMENT) {
    counterValues[action.counterCaption] ++;
    CounterStore.emitChange();
  } else if (action.type === ActionTypes.DECREMENT) {
    counterValues[action.counterCaption] --;
    CounterStore.emitChange();
  }
});
```

Flux 更新状态的函数只有一个参数 action，因为状态是由 Store 直接管理的，所以处理函数中会看到代码直接更新 state；在 Redux 中，一个实现同样功能的 reducer 代码如下：

```
function reducer(state, action) => {
  const {counterCaption} = action;

  switch (action.type) {
    case ActionTypes.INCREMENT:
      return {...state, [counterCaption]: state[counterCaption] + 1};
    case ActionTypes.DECREMENT:
      return {...state, [counterCaption]: state[counterCaption] - 1};
    default:
      return state
  }
}
```

可以看到 reducer 函数不光接受 action 为参数，还接受 state 为参数。也就是说，Redux 的 reducer 只负责计算状态，却并不负责存储状态。

我们会在后面的实例中详细解释这个 reducer 的构造。

读到这里，读者可能会有一个疑问，从 Redux 的基本原则来看，Redux 并没有赋予我们强大的功能，反而是给开发者增加了很多限制啊，开发者丧失了想怎么写就怎么写的灵活度。

“如果你愿意限制做事方式的灵活度，你几乎总会发现可以做得更好。”

——John Carmark

作为制作出《Doom》《Quake》这样游戏的杰出开发者，John Carmark 这句话道出了软件开发中的一个真谛。

在计算机编程的世界里，完成任何一件任务，可能都有一百种以上的方法，但是无

节制的灵活度反而让软件难以维护，增加限制是提高软件质量的法门。

3.2.2　Redux 实例

单纯只看书面介绍难以理解 Redux 如何工作的，让我们还是通过例子来介绍。

前面我们用 Flux 实现了一个 ControlPanel 的应用，接下来让我们用 Redux 来重新实现一遍同样的功能，通过对比就能看出二者的差异。

React 和 Redux 事实上是两个独立的产品，一个应用可以使用 React 而不使用 Redux，也可以使用 Redux 而不使用 React，但是，如果两者结合使用，没有理由不使用一个名叫 react-redux 的库，这个库能够大大简化代码的书写。

不过，如果一开始就使用 react-redux，可能对其设计思路完全一头雾水，所以，我们的实例先不采用 react-redux 库从最简单的 Redux 使用方法开始，初步改进，循序渐进地过渡到使用 react-redux。

最基本的 Redux 实现，存在于本书对应 Github 的 chapter-03/redux_basic 目录中，在这里我们只关注使用 Redux 实现和使用 Flux 实现的不同的文件。

首先看关于 action 对象的定义，和 Flux 一样，Redux 应用习惯上把 action 类型和 action 构造函数分成两个文件定义，其中定义 action 类型的 src/ActionTypes.js 和 Flux 版本没有任何差别，但是 src/Actions.js 文件就不大一样了，代码如下：

```
import * as ActionTypes from './ActionTypes.js';

export const increment = (counterCaption) => {
  return {
    type: ActionTypes.INCREMENT,
    counterCaption: counterCaption
  };
};

export const decrement = (counterCaption) => {
  return {
    type: ActionTypes.DECREMENT,
    counterCaption: counterCaption
  };
};
```

和 Flux 的 src/Actions.js 文件对比就会发现，Redux 中每个 action 构造函数都返回一个 action 对象，而 Flux 版本中 action 构造函数并不返回什么，而是把构造的动作函数立刻通过调用 Dispatcher 的 dispatch 函数派发出去。

这是一个习惯上的差别，接下来我们会发现，在 Redux 中，很多函数都是这样不做什么产生副作用的动作，而是返回一个对象，把如何处理这个对象的工作交给调用者。

在 Flux 中我们要用到一个 Dispatcher 对象，但是在 Redux 中，就没有 Dispatcher 这个对象了，Dispatcher 存在的作用就是把一个 action 对象分发给多个注册了的 Store，既然 Redux 让全局只有一个 Store，那么再创造一个 Dispatcher 也的确意义不大。所以，Redux 中“分发”这个功能，从一个 Dispatcher 对象简化为 Store 对象上的一个函数 dispatch，毕竟只有一个 Store，要分发也是分发给这个 Store，就调用 Store 上一个表示分发的函数，合情合理。

我们创造一个 src/Store.js 文件，这个文件输出全局唯一的那个 Store，代码如下：

```
import {createStore} from 'redux';
import reducer from './Reducer.js';

const initValues = {
  'First': 0,
  'Second': 10,
  'Third': 20
};

const store = createStore(reducer, initValues);

export default store;
```

在这里，我们接触到了 Redux 库提供的 createStore 函数，这个函数第一个参数代表更新状态的 reducer，第二个参数是状态的初始值，第三个参数可选，代表 Store Enhancer，在这个简单例子中用不上，在后面的章节中会详细介绍。

确定 Store 状态，是设计好 Redux 应用的关键。从 Store 状态的初始值看得出来，我们的状态是这样一个格式：状态上每个字段名代表 Counter 组件的名（caption），字段的值就是这个组件当前的计数值，根据这些状态字段，足够支撑三个 Counter 组件。

那么，为什么没有状态来支持 Summary 组件呢？因为 Summary 组件的状态，完全可以通过把 Counter 状态数值加在一起得到，没有必要制造冗余数据存储，这也符合 Redux“唯一数据源”的基本原则。记住，Redux 的 Store 状态设计的一个主要原则：避免冗余的数据。

接下来看 src/Reducer.js 中定义的 reducer 函数，代码如下：

```
import * as ActionTypes from './ActionTypes.js';

export default (state, action) => {
  const {counterCaption} = action;

  switch (action.type) {
    case ActionTypes.INCREMENT:
      return {...state, [counterCaption]: state[counterCaption] + 1};
```

```
    case ActionTypes.DECREMENT:
      return {...state, [counterCaption]: state[counterCaption] - 1};
    default:
      return state
  }
}
```

和Flux应用中每个Store注册的回调函数一样，reducer函数中往往包含以action.type为判断条件的if-else或者switch语句。

和Flux不同的是，多了一个参数state。在Flux的回调函数中，没有这个参数，因为state是由Store管理的，而不是由Flux管理的。Redux中把存储state的工作抽取出来交给Redux框架本身，让reducer只用关心如何更新state，而不要管state怎么存。

代码中使用了三个句点组成的扩展操作符(spread operator)，这表示把state中所有字段扩展开，而后面对counterCaption值对应的字段会赋上新值，像下面这样的代码这样：

```
return {...state, [counterCaption]: state[counterCaption] + 1};
```

上面的代码逻辑上等同于下面的代码：

```
const newState = Object.assign({}, state);

newState[counterCaption] ++;
return newState;
```

像上面这样写，创造了一个newState完全复制了state，通过对newState的修改避免了对state的修改，不过这样写显得冗长，使用扩展操作符看起来更清晰简洁。

提示 扩展操作符（spread operator）并不是ES6语法的一部分，甚至都不是ES Next语法的一部分，但是因为其语法简单，已经被广泛使用，因为babel的存在，也不会有兼容性问题，所以我们完全可以放心使用。

和Flux很不一样的是，在reducer中，绝对不能去修改参数中的state，如果我们直接修改state并返回state，代码如下，注意下面的代码不是正确写法：

```
export default (state, action) => {
  const {counterCaption} = action;
  switch (action.type) {
    case ActionTypes.INCREMENT:
      state[counterCaption] ++;
    case ActionTypes.DECREMENT:
      state[counterCaption] --;
  }
  return state;
}
```

像上面这样写，似乎更简单直接，但实际上犯了大错，因为 reducer 应该是一个纯函数，纯函数不应该产生任何副作用。

接下来，我们看 View 部分，View 部分代码都在 src/views 目录下。看看 src/views/ControlPanel.js，作为这个应用最顶层的组件 ControlPanel，内容和 Flux 例子中没有任何区别。然后是 Counter 组件，存在于 src/views/Counter.js 中，这就和 Flux 不大一样了，首先是构造函数中初始化 this.state 的来源不同，代码如下：

```
import store from '../Store.js';

class Counter extends Component {
  constructor(props) {
    super(props);
    . . .
    this.state = this.getOwnState();
  }

  getOwnState() {
    return {
      value: store.getState()[this.props.caption]
    };
  }
```

和 Flux 例子一样，在这个视图文件中我们要引入 Store，只不过这次是我们引入的 Store 不叫 CounterStore，而是一个唯一的 Redux Store，所以名字就叫 store，通过 store.getState() 能够获得 store 上存储的所有状态，不过每个组件往往只需要使用返回状态的一部分数据。为了避免重复代码，我们把从 store 获得状态的逻辑放在 getOwnState 函数中，这样任何关联 Store 状态的地方都可以重用这个函数。

和 Flux 实现的例子一样，仅仅在构造组件时根据 store 来初始化 this.state 还不够，要保持 store 上状态和 this.state 的同步，代码如下：

```
  onChange() {
    this.setState(this.getOwnState());
  }

  componentDidMount() {
    store.subscribe(this.onChange);
  }

  componentWillUnmount() {
    store.unsubscribe(this.onChange);
  }
```

在 componentDidMount 函数中，我们通过 Store 的 subscribe 监听其变化，只要 Store

状态发生变化，就会调用这个组件的 onChange 方法；在 componentWillUnmount 函数中，我们把这个监听注销掉，这个清理动作和 componentDidMount 中的动作对应。

其实，这个增加监听函数的语句也可以写在构造函数中，但是为了让 mount 和 unmount 的对应看起来更清晰，在所有的例子中我们都把加载监听的函数放在 componentDidMount 中。

除了从 store 同步状态，视图中可能会想要改变 store 中的状态，和 Flux 一样，改变 store 中状态唯一的方法就是派发 action，代码如下：

```
onIncrement() {
  store.dispatch(Actions.increment(this.props.caption));
}

onDecrement() {
  store.dispatch(Actions.decrement(this.props.caption));
}
```

上面定义了 onIncrement 和 onDecrement 方法，在 render 函数中的 JSX 中需要使用这两种函数，代码如下：

```
render() {
  const value = this.state.value;
  const {caption} = this.props;

  return (
    <div>
      <button style={buttonStyle} onClick={this.onIncrement}>+</button>
      <button style={buttonStyle} onClick={this.onDecrement}>-</button>
      <span>{caption} count: {value}</span>
    </div>
  );
}
```

在 render 函数中，对于点击“+”按钮和“-”按钮的 onClick 事件，被分别挂上了 onIncrement 函数和 onDecrement 函数，所做的事情就是派发对应的 action 对象出去。注意和 Flux 例子的区别，在 Redux 中，action 构造函数只负责创建对象，要派发 action 就需要调用 store.dispatch 函数。

组件的 render 函数所显示的动态内容，要么来自于 props，要么来自于自身状态。

然后再来看看 src/views/Summary.js 中的 Summary 组件，其中 getOwnState 函数的实现代码如下：

```
getOwnState() {
  const state = store.getState();
  let sum = 0;
```

```
  for (const key in state) {
    if (state.hasOwnProperty(key)) {
      sum += state[key];
    }
  }
  return { sum: sum };
}
```

Summary 组件的套路和 Counter 组件差不多，唯一值得一提的就是 getOwnState 函数的实现。因为 Store 的状态中只记录了各个 Counter 组件的计数值，所以需要在 getOwnState 状态中自己计算出所有计数值总和出来。

在网页中看一下最终效果，和 Flux 例子没有任何区别。

3.2.3 容器组件和傻瓜组件

分析一下上面的 Redux 例子中的 Counter 组件和 Summary 组件部分，可以发现一个规律，在 Redux 框架下，一个 React 组件基本上就是要完成以下两个功能：

- 和 Redux Store 打交道，读取 Store 的状态，用于初始化组件的状态，同时还要监听 Store 的状态改变；当 Store 状态发生变化时，需要更新组件状态，从而驱动组件重新渲染；当需要更新 Store 状态时，就要派发 action 对象；
- 根据当前 props 和 state，渲染出用户界面。

还记得那句话吗？让一个组件只专注做一件事，如果发现一个组件做的事情太多了，就可以把这个组件拆分成多个组件，让每个组件依然只专注做一件事。

如果 React 组件都是要包办上面说的两个任务，似乎做的事情也的确稍微多了一点。我们可以考虑拆分，拆分为两个组件，分别承担一个任务，然后把两个组件嵌套起来，完成原本一个组件完成的所有任务。

这样的关系里，两个组件是父子组件的关系。业界对于这样的拆分有多种叫法，承担第一个任务的组件，也就是负责和 Redux Store 打交道的组件，处于外层，所以被称为容器组件（Container Component）；对于承担第二个任务的组件，也就是只专心负责渲染界面的组件，处于内层，叫做展示组件（Presentational Component）。

外层的容器组件又叫聪明组件（Smart Component），内层的展示组件又叫傻瓜组件（Dumb Component），所谓“聪明”还是“傻瓜”只是相对而言，并没有褒贬的含义。如图 3-4 所示。

图 3-4 容器组件和傻瓜组件的分工

傻瓜组件就是一个纯函数，根据 props 产生

结果。说是“傻瓜”，我倒是觉得这种纯函数实现反而体现了计算机编程中的大智慧，大智若愚。

而容器组件，只是做的事情涉及一些状态转换，虽然名字里有“聪明”，其实做的事情都有套路，我们很容易就能抽取出共同之处，复用代码完成任务，并不需要开发者极其聪明才能掌握。

在我们把一个组件拆分为容器组件和傻瓜组件的时候，不只是功能分离，还有一个比较大的变化，那就是傻瓜组件不需要有状态了。

实际上，让傻瓜组件无状态，是我们拆分的主要目的之一，傻瓜组件只需要根据props来渲染结果，不需要state。

那么，状态哪里去了呢？全都交给容器组件去打点，这是它的责任。容器组件如何把状态传递给傻瓜组件呢？通过props。

值得一提的是，拆分容器组件和傻瓜组件，是设计React组件的一种模式，和Redux没有直接关系。在Flux或者任何一种其他框架下都可以使用这种模式，只不过为了引出后面的react-redux，我们才在这里开始介绍罢了。

我们还是通过例子来感受一下容器组件和傻瓜组件如何协同工作，对应的代码在chapter-03/redux_smart_dumb目录下，是前面chapter-03/redux_basic的改进，只有视图部分代码有改变。

在视图代码src/views/Counter.js中定义了两个组件，一个是Counter，这是傻瓜组件，另一个是CounterContainer，这是容器组件。

傻瓜组件Counter代码的逻辑前所未有的简单，只有一个render函数，代码如下：

```
class Counter extends Component {
  render() {
    const {caption, onIncrement, onDecrement, value} = this.props;

    return (
      <div>
        <button style={buttonStyle} onClick={onIncrement}>+</button>
        <button style={buttonStyle} onClick={onDecrement}>-</button>
        <span>{caption} count: {value}</span>
      </div>
    );
  }
}
```

可以看到，Counter组件完全没有state，只有一个render方法，所有的数据都来自于props，这种组件叫做“无状态”组件。

而CounterContainer组件承担了所有的和Store关联的工作，它的render函数所做的

就是渲染傻瓜组件 Counter 而已，只负责传递必要的 prop，相关代码如下：

```
class CounterContainer extends Component {
  render() {
    return <Counter caption={this.props.caption}
      onIncrement={this.onIncrement}
      onDecrement={this.onDecrement}
      value={this.state.value} />
  }
}

export default CounterContainer;
```

可以看到，这个文件 export 导出的不再是 Counter 组件，而是 CounterContainer 组件，也就是对于使用这个视图的模块来说，根本不会感受到傻瓜组件的存在，从外部看到的就只是容器组件。

对于无状态组件，其实我们可以进一步缩减代码，React 支持只用一个函数代表的无状态组件，所以，Counter 组件可以进一步简化，代码如下：

```
function Counter (props) {
  const {caption, onIncrement, onDecrement, value} = props;

  return (
    <div>
      <button style={buttonStyle} onClick={onIncrement}>+</button>
      <button style={buttonStyle} onClick={onDecrement}>-</button>
      <span>{caption} count: {value}</span>
    </div>
  );
}
```

因为没有状态，不需要用对象表示，所以连类都不需要了，对于一个只有 render 方法的组件，缩略为一个函数足矣。

注意，改为这种写法，获取 props 就不能用 this.props，而是通过函数的参数 props 获得，无状态组件的 props 参数和有状态组件的 this.props 内容和结构完全一样。

还有一种惯常写法，就是把解构赋值（destructuring assignment）直接放在参数部分。

```
function Counter ({caption, onIncrement, onDecrement, value}) {
  // 函数体中可以直接使用caption、onIncrement等变量
}
```

看 src/views/Summary.js 中，内容也被分解为了傻瓜组件 Summary 和 SummaryContainer，方式和 Counter 差不多，不再赘述。

重新审阅代码，我们可以看到 CounterSummary 和 SummaryContainer 代码有很多相

同之处，写两份实在是重复，既然都是套路，完全可以抽取出来，后面的章节会讲如何应用 react-redux 来减少重复代码。

3.2.4　组件 Context

在介绍 react-redux 之前，我们重新看一看现在的 Counter 和 Summary 组件文件，发现它们都直接导入 Redux Store。

```
import store from '../Store.js';
```

虽然 Redux 应用全局就一个 Store，这样的直接导入依然有问题。

在实际工作中，一个应用的规模会很大，不会所有的组件都放在一个代码库中，有时候还要通过 npm 方式引入第三方的组件。想想看，当开发一个独立的组件的时候，都不知道自己这个组件会存在于哪个应用中，当然不可能预先知道定义唯一 Redux Store 的文件位置了，所以，在组件中直接导入 Store 是非常不利于组件复用的。

一个应用中，最好只有一个地方需要直接导入 Store，这个位置当然应该是在调用最顶层 React 组件的位置。在我们的 ControlPanel 例子中，就是应用的入口文件 src/index.js 中，其余组件应该避免直接导入 Store。

不让组件直接导入 Store，那就只能让组件的上层组件把 Store 传递下来了。首先想到的当然是用 props，毕竟，React 组件就是用 props 来传递父子组件之间的数据的。不过，这种方法有一个很大的缺陷，就是从上到下，所有的组件都要帮助传递这个 props。

设想在一个嵌套多层的组件结构中，只有最里层的组件才需要使用 store，但是为了把 store 从最外层传递到最里层，就要求中间所有的组件都需要增加对这个 store prop 的支持，即使根本不使用它，这无疑很麻烦。

还是来看 ControlPanel 这个例子，最顶层的组件 ControlPanel 根本就不使用 store，如果仅仅为了让它传递一个 prop 给子组件 Counter 和 Summary 就要求它支持 state prop，显然非常不合理。所以，用 prop 传递 store 不是一个好方法。

React 提供了一个叫 Context 的功能，能够完美地解决这个问题。如图 3-5 所示。

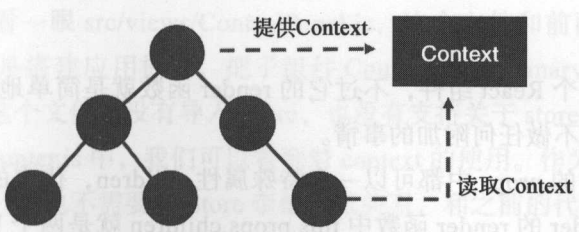


图 3-5　React 的 Context

所谓 Context，就是“上下文环境”，让一个树状组件上所有组件都能访问一个共同的对象，为了完成这个任务，需要上级组件和下级组件配合。

首先，上级组件要宣称自己支持 context，并且提供一个函数来返回代表 Context 的对象。

然后，这个上级组件之下的所有子孙组件，只要宣称自己需要这个 context，就可以通过 this.context 访问到这个共同的环境对象。

我们尝试给 ControlPanel 程序加上 context 功能来优化，相关代码在 chapter-three/redux_with_countext/ 目录中，这个应用是对前面 redux_smart_dumb 的改进，我们这里只关注改变的代码。

因为 Redux 应用中只有一个 Store，因此所有组件如果要使用 Store 的话，只能访问这唯一的 Store。很自然，希望顶层的组件来扮演这个 Context 提供者的角色，只要顶层组件提供包含 store 的 context，那就覆盖了整个应用的所有组件，简单而且够用。

不过，每个应用的顶层组件不同，在我们的 ControlPanel 例子里顶层组件是 ControlPanel，在另一个应用里会有另一个组件。而且，ControlPanel 有它自己的职责，我们来没有理由把它复杂化，没必要非要让它扮演 context 提供者的功能。

我们来创建一个特殊的 React 组件，它将是一个通用的 context 提供者，可以应用在任何一个应用中，我们把这个组件叫做 Provider。在 src/Provider.js 中，首先定义一个名为 Provider 的 React 组件，代码如下：

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

class Provider extends Component {
  getChildContext() {
    return {
      store: this.props.store
    };
  }

  render() {
    return this.props.children;
  }
}
```

Provider 也是一个 React 组件，不过它的 render 函数就是简单地把子组件渲染出来，在渲染上，Provider 不做任何附加的事情。

每个 React 组件的 props 中都可以一个特殊属性 children，代表的是子组件，比如这样的代码，在 Provider 的 render 函数中 this.props.children 就是两个 Provider 标签之间的 <ControlPanel />：

```
<Provider>
  <ControlPanel />
</Provider>
```

除了把渲染工作完全交给子组件，Provider 还要提供一个函数 getChildContext，这个函数返回的就是代表 Context 的对象。我们的 Context 中只有一个字段 store，而且我们也希望 Provider 足够通用，所以并不在这个文件中导入 store，而是要求 Provider 的使用者通过 prop 传递进来 store。

为了让 Provider 能够被 React 认可为一个 Context 的提供者，还需要指定 Provider 的 childContextTypes 属性，代码如下：

```
Provider.childContextTypes = {
  store: PropTypes.object
};
```

Provider 还需要定义类的 childContextTypes，必须和 getChildContext 对应，只有这两者都齐备，Provider 的子组件才有可能访问到 context。

有了 Provider，我们就可以改进一下应用的入口 src/index.js 文件了，代码如下：

```
import store from './Store.js';
import Provider from './Provider.js';

ReactDOM.render(
  <Provider store={store}>
    <ControlPanel />
  </Provider>,
  document.getElementById('root')
);
```

在前面所有的例子中，React.render 的第一个参数就是顶层组件 ControlPanel。现在，这个 ControlPanel 作为子组件被 Provider 包住了，Provider 成为了顶层组件。当然，如同我们上面看到的，Provider 只是把渲染工作完全交给子组件，它扮演的角色只是提供 Context，包住了最顶层的 ControlPanel，也就让 context 覆盖了整个应用中所有组件。

至此，我们完成了提供 Context 的工作，接下来我们看底层组件如何使用 Context。

我们可以顺便看一眼 src/views/ControlPanel.js，这个文件和前面的例子没有任何变化，它做的工作只是搭建应用框架，把子组件 Counter 和 Summary 渲染出来，和 Store 一点关系都没有，这个文件既没有导入 Store，也没有支持关于 store 的 props。

在 src/views/Counter.js 中，我们可以看到对 context 的使用。作为傻瓜组件的 Counter 是一个无状态组件，它也不需要和 Store 牵扯什么关系，和之前的代码一模一样，有变化的是 CounterContainer 部分。

为了让 CounterContainer 能够访问到 context，必须给 CounterContainer 类的 contextTypes 赋值和 Provider.childContextTypes 一样的值，两者必须一致，不然就无法访问到 context，代码如下：

```
CounterContainer.contextTypes = {
  store: PropTypes.object
}
```

在 CounterContainer 中，所有对 store 的访问，都是通过 this.context.store 完成，因为 this.context 就是 Provider 提供的 context 对象，所以 getOwnState 函数代码如下：

```
getOwnState() {
  return {
    value: this.context.store.getState()[this.props.caption]
  };
}
```

还有一点，因为我们自己定义了构造函数，所以要用上第二个参数 context，代码如下：

```
constructor(props, context) {
  super(props, context);
  ...
}
```

在调用 super 的时候，一定要带上 context 参数，这样才能让 React 组件初始化实例中的 context，不然组件的其他部分就无法使用 this.context。

要求 constructor 显示声明 props 和 context 两个参数然后又传递给 super 看起来很麻烦，我们的代码似乎只是一个参数的搬运工，而且将来可能有新的参数出现那样又要修改这部分代码，如果你这样认为的话，可以用下面的方法一劳永逸地解决这个问题，代码如下：

```
constructor() {
  super(...arguments);
}
```

我们不能直接使用 arguments，因为在 JavaScript 中 arguments 表现得像是一个数组而不是分开的一个个参数，但是我们通过扩展标示符就能把 arguments 彻底变成传递给 super 的参数。

在结束之前，让我们重新审视一下 Context 这个功能，Context 这个功能相当于提供了一个全局可以访问的对象，但是全局对象或者说全局变量肯定是我们应该避免的用法，只要有一个地方改变了全局对象的值，应用中其他部分就会受影响，那样整个程序的运

行结果就完全不可预期了。

所以，单纯来看React的这个Context功能的话，必须强调这个功能要谨慎使用，只有对那些每个组件都可能使用，但是中间组件又可能不使用的对象才有必要使用Context，千万不要滥用。

对于Redux，因为Redux的Store封装得很好，没有提供直接修改状态的功能，就是说一个组件虽然能够访问全局唯一的Store，却不可能直接修改Store中的状态，这样部分克服了作为全局对象的缺点。而且，一个应用只有一个Store，这个Store是Context里唯一需要的东西，并不算滥用，所以，使用Context来传递Store是一个不错的选择。

3.2.5 React-Redux

在上面两节中，我们了解了改进React应用的两个方法，第一是把一个组件拆分为容器组件和傻瓜组件，第二是使用React的Context来提供一个所有组件都可以直接访问的Context，也不难发现，这两种方法都有套路，完全可以把套路部分抽取出来复用，这样每个组件的开发只需要关注于不同的部分就可以了。

实际上，已经有这样的一个库来完成这些工作了，这个库就是react-redux。

在本书的chapter-03/react-redux目录下，可以看到利用react-redux实现的Control-Panel版本，因为使用了react-redux，所以它是所有实现方式中代码最精简的一个例子。

我们只看不同的部分。在src/index.js中，代码几乎和react_with_context一模一样，唯一的区别就是我们不再使用自己实现的Provider，而是从react-redux库导入Provider，代码如下：

```
import {Provider} from 'react-redux';
```

有了react-redux，视图文件src/views/Counter.js和src/Summary.js中的代码可以变得相当简洁。

在前面的redux_smart_dumb和redux_with_context例子中，我们实际上分别实现了react-redux的两个最主要功能：

- connect：连接容器组件和傻瓜组件；
- Provider：提供包含store的context。

现在我们直接使用react-redux提供的这两个功能了，让我们分别来详细介绍。

1. connect

以Counter组件为例，和redux_with_context中的代码不同，react-redux的例子中没有定义CounterContainer这样命名的容器组件，而是直接导出了一个这样的语句。

```
export default connect(mapStateToProps, mapDispatchToProps)(Counter);
```

第一眼看去，会让人觉得这不是正常的 JavaScript 语法。其实，connect 是 react-redux 提供的一个方法，这个方法接收两个参数 mapStateToProps 和 mapDispatch-ToProps，执行结果依然是一个函数，所以才可以在后面又加一个圆括号，把 connect 函数执行的结果立刻执行，这一次参数是 Counter 这个傻瓜组件。

这里有两次函数执行，第一次是 connect 函数的执行，第二次是把 connect 函数返回的函数再次执行，最后产生的就是容器组件，功能相当于前面 redux_smart_dumb 中的 CounterContainer。

当然，我们也可以把 connect 的结果赋值给一个变量 CounterContainer，然后再 export 这个 CounterContainer，只是 connect 已经大大简化了代码，习惯上可以直接导出函数执行结果，也不用纠结如何命名这个变量。

这个 connect 函数具体做了什么工作呢？

作为容器组件，要做的工作无外乎两件事：

- 把 Store 上的状态转化为内层傻瓜组件的 prop；
- 把内层傻瓜组件中的用户动作转化为派送给 Store 的动作。

这两个工作一个是内层傻瓜对象的输入，一个是内层傻瓜对象的输出。

这两个工作的套路也很明显，把 Store 上的状态转化为内层组件的 props，其实就是一个映射关系，去掉框架，最后就是一个 mapStateToProps 函数该做的事情。这个函数命名是业界习惯，因为它只是一个模块内的函数，所以实际上叫什么函数都行，如果觉得 mapStateToProps 这个函数名太长，也可以叫 mapState，也是业界惯常的做法。

Counter 组件对应的 mapStateToProps 函数代码如下：

```
function mapStateToProps(state, ownProps) {
  return {
    value: state[ownProps.caption]
  }
}
```

把内层傻瓜组件中用户动作转化为派送给 Store 的动作，也就是把内层傻瓜组件暴露出来的函数类型的 prop 关联上 dispatch 函数的调用，每个 prop 代表的回调函数的主要区别就是 dispatch 函数的参数不同，这就是 mapDispatchToProps 函数做的事情，和 mapStateToProps 一样，这么长的函数名只是习惯问题，mapDispatchToProps 也可以叫做 mapDispatch。

Counter 组件对应的 mapDispatchToProps 函数代码如下：

```
function mapDispatchToProps(dispatch, ownProps) {
```

```
    return {
      onIncrement: () => {
        dispatch(Actions.increment(ownProps.caption));
      },
      onDecrement: () => {
        dispatch(Actions.decrement(ownProps.caption));
      }
    }
}
```

mapStateToProps 和 mapDispatchToProps 都可以包含第二个参数，代表 ownProps，也就是直接传递给外层容器组件的 props，在 ControlPanel 的例子中没有用到，我们在后续章节中会有详细介绍。

2. Provider

我们在 redux_with_context 中已经完整实现了一个 Provider，react-redux 和我们例子中的 Provider 几乎一样，但是更加严谨，比如我们只要求 store 属性是一个 object，而 react-redux 要求 store 不光是一个 object，而且是必须包含三个函数的 object，这三个函数分别是。

- subscribe
- dispatch
- getState

拥有上述三个函数的对象，才能称之为一个 Redux 的 store。

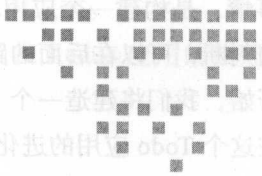
另外，react-redux 定义了 Provider 的 componentWillReceiveProps 函数，在 React 组件的生命周期中，componentWillReceiveProps 函数在每次重新渲染时都会调用到，react-redux 在 componentWillReceiveProps 函数中会检查这一次渲染时代表 store 的 prop 和上一次的是否一样。如果不一样，就会给出警告，这样做是为了避免多次渲染用了不同的 Redux Store。每个 Redux 应用只能有一个 Redux Store，在整个 Redux 的生命周期中都应该保持 Store 的唯一性。

3.3　本章小结

在这一章中，我们首先从 Redux 的鼻祖 Flux 框架出发，通过创造一个 ControlPanel 的例子，了解了 Flux“单向数据流”的原则。如果只由 React 来管理数据流，就很难管理拥有很多组件的大型应用，传统的 MVC 框架也有其缺陷，很容易写乱套，所以 Flux 是应用架构的一个巨大改进，但是 Flux 也有其缺点。

Redux 是 Flux 框架的一个巨大改进，Redux 强调单一数据源、保持状态只读和数据改变只能通过纯函数完成的基本原则，和 React 的 UI=render(state) 思想完全契合。我们在这一章中用不同方法，循序渐进的改进了 ControlPanel 这个应用的例子，为的就是更清晰地理解每个改进背后的动因，最后，我们终于通过 react-redux 完成了 React 和 Redux 的融合。

但是，这只是一个开始。接下来，我们将看到更加深入的 React 和 Redux 实践知识。



第 4 章 Chapter 4

模块化 React 和 Redux 应用

在第一部分中，我们已经了解了 React 的基本工作方式，也知道了 Redux 在组合 React 组件中的作用，但是更多的只是了解其基本原理和使用方法，上手练习的也是一个简单的例子。

实际工作中我们要创建的应用无论结构和大小都要复杂得多，在这一章中，我们要介绍创建一个复杂一点的应用该如何做，包含以下内容：

- 模块化应用的要点；
- 代码文件的组织方式；
- 状态树的设计；
- 开发辅助工具。

4.1 模块化应用要点

在本书中，我们探讨的是如何用 React 和 Redux 来构建前端网页应用，这两者都奉行这样一个公式 UI=render(state) 来产生用户界面。React 才适合于视图层面的东西，但是不能指望靠 React 来管理应用的状态，Redux 才适合担当应用状态的管理工作。

从架构出发，当我们开始一个新的应用的时候，有几件事情是一定要考虑清楚的；

- 代码文件的组织结构；
- 确定模块的边界；
- Store 的状态树设计。

这三件事情，是构建一个应用的基础。如果我们在一开始深入思考这三件事，并作出合乎需要的判断，可以在后面的路上省去很多麻烦。

从本章开始，我们将建造一个“待办事项”（Todo）应用，逐步完善这个应用，增加新的功能，在这个 Todo 应用的进化过程中来学习各个层次的知识。

在这个各种 JavaScript 框架层出不穷的时代，Todo 应用几乎就代替了传统 Hello World 应用的作用，每个框架问世的时候都会用一个 Todo 应用来展示自己的不同，不要小看了这样一个 Todo 应用，它非常适合用于做技术展示，首先，这个应用的复杂度刚刚好，没有复杂到可能要很多篇幅才能解释清楚做什么，也没有简单到只需要几行代码就能够搞定；其次，这样的功能非常利于理解，恰好能够考验一个 JavaScript 框架的表达能力。

确定了我们的应用要做什么之后，不要上来就开始写代码，磨刀不误砍柴工，先要思考上面提到的三个问题。让我们从第一个问题开始吧。

4.2 代码文件的组织方式

4.2.1 按角色组织

如果读者之前曾用 MVC 框架开发过应用程序，应该知道 MVC 框架之下，通常有这样一种代码的组织方式，文件目录列表如下：

```
controllers/
  todoController.js
  filterController.js
models/
  todoModel.js
  filterModel.js
views/
  todo.js
  todoItem.js
  filter.js
```

在 MVC 中，应用代码分为 Controller、Model 和 View，分别代表三种模块角色，就是把所有的 Controller 代码放在 controllers 目录下，把所有的 Model 代码放在 models 目录下，把 View 代码放在 views 目录下。这种组织代码的方式，叫做“按角色组织”（Organized by Roles）。

我们当然不会使用 MVC，在上一章中我们就介绍过 MVC 框架的缺点。和众多前端开发者一样，我们选择 Flux 和 Redux 就是为了克服这些缺点的，但是因为 MVC 框架的

影响非常深远，一些风格依然影响了前端开发人员的思维方式。

因为 MVC 这种“按角色组织”代码文件的影响，在 Redux 应用的构建中，就有这样一种代码组织方法，文件目录列表如下：

```
reducers/
  todoReducer.js
  filterReducer.js
actions/
  todoActions.js
  filterActions.js
components/
  todoList.js
  todoItem.js
  filter.js
containers/
  todoListContainer.js
  todoItemContainer.js
  filterContainer.js
```

和 MVC 的代码组织方式不同，只不过是把 controllers、models 和 views 目录换成了 reducers、actions、components 和 containers，各个目录下代码文件的角色如下：

- reducer 目录包含所有 Redux 的 reducer；
- actions 目录包含所有 action 构造函数；
- components 目录包含所有的傻瓜组件；
- containers 目录包含所有的容器组件。

这种组织方式看起来还不错，把一个类型的代码文件放在了一个目录下，至少比把所有代码全放在一个目录下要有道理。

实际上，在前面章节的所有 ControlPanel 例子中，我们采用的也是类似的方法，当我们发现代码文件变多，全都直接放在一个 src 目录下不合理时，首先想到的就是建一个 views 目录，把所有视图相关的目录移到 views 目录里面去。我们没有移动 action 相关和 reducer 相关的文件，只因为 ControlPanel 应用实在太简单，因为只有一个组件 Counter 可能发出动作，所以只有一个 Action 文件，也只有一个对应的 Reducer 文件，所以到最后我们都没有觉得有必要把它们移到代表各自角色的目录里面去。

在互联网上，很多教学资料也是按照“按角色组织”的方法管理 Redux 应用。虽然“按照角色组织”的方式看起来不错，但是实际上非常不利于应用的扩展。

有过 MVC 框架开发经历的朋友可以回忆一下，当你需要对一个功能进行修改，虽然这个功能只是针对某一个具体的应用模块，但是却牵扯到 MVC 中的三个角色 Controller、Model 和 View，不管你用的是什么样的编辑器，你都得费点劲才能在这三个

目录之间跳转，或者需要滚动文件列表跳过无关的分发器文件才能找到你想要修改的那一个分发器文件。

这真的就是浪费时间。

如果说 MVC 框架下，因为三个角色之间的交叉关系，也只能默默接受，那么在 Redux 框架下，我们已经有机会实行严格模块化的思想，就应该想一想更好的组织文件的方式。

4.2.2 按功能组织

Redux 应用适合于“按功能组织”（Organzied by Feature），也就是把完成同一应用功能的代码放在一个目录下，一个应用功能包含多个角色的代码。在 Redux 中，不同的角色就是 reducer、actions 和视图，而应用功能对应的就是用户界面上的交互模块。

拿 Todo 应用为例子，这个应用的两个基本功能就是 TodoList 和 Filter，所以代码就这样组织，文件目录列表如下：

```
todoList/
  actions.js
  actionTypes.js
  index.js
  reducer.js
  views/
    component.js
    container.js
filter/
  actions.js
  actionTypes.js
  index.js
  reducer.js
  views/
    component.js
    container.js
```

每个基本功能对应的其实就是一个功能模块，每个功能模块对应一个目录，这个例子中分别是 todoList 和 filter，每个目录下包含同样名字的角色文件：

- actionTypes.js 定义 action 类型；
- actions.js 定义 action 构造函数，决定了这个功能模块可以接受的动作；
- reducer.js 定义这个功能模块如何相应 actions.js 中定义的动作；
- views 目录，包含这个功能模块中所有的 React 组件，包括傻瓜组件和容器组件；
- index.js 这个文件把所有的角色导入，然后统一导出。

在这种组织方式下，当你要修改某个功能模块的代码的时候，只要关注对应的目录

就行了，所有需要修改的代码文件都在能这个目录下找到。

表面上看来，“按照角色组织”还是“按照功能组织”只是一个审美的问题，也许你觉得自己已经习惯了 MVC 世界的“按照角色组织”方式，也许你已经有一套很厉害的代码编辑器可以完美解决在不同目录下寻找代码文件困难的问题。但是，开发 Redux 应用你依然应该用“按照功能组织”的方式，为什么呢？我们看看下一条“确定模块的边界”就明白了。

4.3　模块接口

> “在最理想的情况下，我们应该通过增加代码就能增加系统的功能，而不是通过对现有代码的修改来增加功能。”
>
> ——Robert C. Martin

著名软件架构师 Robert C. Martin 这句话阐述了模块化软件的要求。当然，如果要达到这种“最理想的情况”还是有点难度，但是这个目标值得我们去努力奋斗，而 React 和 Redux 这个辅助工具可以帮助我们在这个目标上迈进一大步。

不同功能模块之间的依赖关系应该简单而且清晰，也就是所谓的保持模块之间低耦合性；一个模块应该把自己的功能封装得很好，让外界不要太依赖与自己内部的结构，这样不会因为内部的变化而影响外部模块的功能，这就是所谓高内聚性。

React 组件本身应该具有低耦合性和高内聚性的特点，不过，在 Redux 的游乐场中，React 组件扮演的就是一个视图的角色，还有 reducer、actions 这些角色参与这个游戏。对于整个 Redux 应用而言，整体由模块构成，但是模块不再是 React 组件，而是由 React 组件加上相关 reducer 和 actions 构成的一个小整体。

以我们将要实现实现的 Todo 应用为例，功能模块就是 todoList 和 filter，这两个功能模块分别用各自的 React 组件、reducer 和 action 定义。

可以预期每个模块之间会有依赖关系，比如 filter 模块想要使用 todoList 的 action 构造函数和视图，那么我们希望对方如何导入呢？一种写法是像下面的代码这样：

```
import * as actions from '../todoList/actions';
import container as TodoList from '../todoList/views/container';
```

这种写法当然能够完成功能，但是却非常不合理，因为这让 filter 模块依赖于 todoList 模块的内部结构，而且直接伸手到 todoList 内部去导入想要的部分。

虽然我们在上面的例子中，todoList 和 filter 中的文件名几乎一样，但是这毕竟是模块内部的事情，不应该硬性要求，更不应该假设所有的模块都应该按照这样的文件名命

名。在我们的例子中，存储视图代码文件的目录叫做 views，但是有的开发者习惯把这个功能的目录叫做 components；我们把包含容器组件的文件名叫做 container.js，根据开发者个人习惯也可能叫做 TodoList，这些都没有必要而且也不应该有硬性规定。

现在既然我们把一个目录看做一个模块，那么我们要做的是明确这个模块对外的接口，而这个接口应该实现把内部封装起来。

请注意我们的 todoList 和 filter 模块目录下，都有一个 index.js 文件，这个文件就是我们的模块接口。

比如，在 todoList/index.js 中，代码如下：

```
import * as actions from './actions.js';
import reducer from './reducer.js';
import view from './views/container.js';

export {actions, reducer, view}
```

如果 filter 中的组件想要使用 todoList 中的功能，应该导入 todoList 这个目录，因为导入一个目录的时候，默认导入的就是这个目录下的 index.js 文件，index.js 文件中导出的内容，就是这个模块想要公开出来的接口。

注意 虽然每个模块目录下都会有一个 actionTypes.js 文件定义 action 类型，但是通常不会把 actionTypes 中内容作为模块的接口之一导出，因为 action 类型只有两个部分依赖，一个是 reducer，一个是 action 构造函数，而且只有当前模块的 reducer 和 action 构造函数才会直接使用 action 类型。模块之外，不会关心这个模块的 action 类型，如果模块之外要使用这个模块的动作，也只需要直接使用 action 构造函数就行。

下面就是对应的导入 todoList 的代码：

```
import {actions, reducer, view as TodoList} from '../todoList';
```

当我们想要修改 todoList 的内部代码结构，比如把 views 目录改名为 components 目录，或者把 container.js 改名为 TodoListView.js 或者任何一个我们觉得更加有意义的名字，所要做的只是修改 todoList 目录下的 index.js 内容，而这个文件 export 出来的内容不会有任何改变，也就是说对导入 todoList 的代码不用任何改变。这就是我们确定模块边界想要达到的目的。

还有一种导出模块接口的方式，是不以命名式 export 的方式导出模块接口，而是以 export default 的方式默认导出，就像这样的代码：

```
import * as actions from './actions.js';
import reducer from './reducer.js';
import view from './views/container.js';

export default {actions, reducer, view}
```

如果像上面导出，那么导入时的代码会有一点区别，因为 ES6 语法中，export default 和 export 两种导出方式的导入方式也会不同，代码如下：

```
import TodoListComponent from './actions.js';

const reducer = TodoListComponent.reducer;
const actions = TodoListComponent.actions;
const TodoList = TodlistComponent.view;
```

无论使用哪种导出方式，都请在整个应用中只用一种模块导出方式，保持一致，避免混乱。

在本书中，全部使用 export 的方式，因为从上面的代码看得出来，如果使用 export default 的方式，在导入的时候不可避免要用多行代码才能得到 actions、reducer 和 view，而用导入命名式 export 只用一行就可以搞定，相对而言要更加简洁。

读者可能也注意到了，上面接口代码中导入 actions 的语句和导入 view 和 reducer 不一样，代码如下：

```
import * as actions from './actions.js';
```

我们预期 actions.js 中是按照命名式 export，原因和上面陈述的一样，actions.js 可能会导出很多 action 构造函数，命名式导出是为了导入 actions 方便。对于 view 和 reducer，一个功能模块绝对只有一个根视图模块，一个功能模块也只应该有一个导出的 reducer，所以它们两个在各自代码文件中是以默认方式导出的。

4.4　状态树的设计

上面说的“代码文件组织结构”和“确定模块的边界”更多的只是确定规矩，然后在每个应用中我们只要都遵循这个规矩就足够了，而要注意的第三点“Store 上状态树的设计”，更像是一门技术，需要我们动一动脑子。

因为所有的状态都存在 Store 上，Store 的状态树设计，直接决定了要写哪些 reducer，还有 action 怎么写，所以是程序逻辑的源头。

我们认为状态树设计要遵循如下几个原则：

- 一个模块控制一个状态节点；

- 避免冗余数据；
- 树形结构扁平。

让我们逐个解释这些原则。

4.4.1 一个状态节点只属于一个模块

这个规则与其说是规则，不如说是 Redux 中模块必须遵守的限制，完全无法无视这个限制。

在 Redux 应用中，Store 上的每个 state 都只能通过 reducer 来更改，而我们每个模块都有机会导出一个自己的 reducer，这个导出的 reducer 只能最多更改 Redux 的状态树上一个节点下的数据，因为 reducer 之间对状态树上的修改权是互斥的，不可能让两个 reducer 都可以修改同一个状态树上的节点。

比如，如果 A 模块的 reducer 负责修改状态树上 a 字段下的数据，那么另一个模块 B 的 reducer 就不可能有机会修改 a 字段下的数据。

这里所说的“拥有权”指的是“修改权”，而不是“读取权”，实际上，Redux Store 上的全部状态，在任何时候，对任何模块都是开放的，通过 store.getState() 总能够读取当整个状态树的数据，但是只能更新自己相关那一部分模块的数据。

4.4.2 避免冗余数据

冗余数据是一致性的大敌，如果在 Store 上存储冗余数据，那么维持不同部分数据一致就是一个大问题。

传统的关系型数据库中，对数据结构的各种“范式化”，其实就是在去除数据的冗余。而近年风生水起的 NoSQL 运动，提倡的就是在数据存储中“去范式化”，对数据结构的处理和关系型数据库正好相反，利用数据冗余来减少读取数据库时的数据关联工作。

面向用户的应用出于性能的考虑，倾向于直接使用“去范式化”的应用。但是带来的问题就是维持数据一致性就会困难。

不同的应用当然应该从自己的需要出发，在选择数据库的问题上，选择 SQL 关系型数据库或者 NoSQL 类型的数据库要根据应用特点，这个问题不是我们要在本书中要讨论的。但是要强调的是，不管服务器端数据库用的是“范式化”还是“去范式化”的数据存储方式，在前端 Redux 的 Store 中，一定要避免数据冗余的出现。

并不是说 Redux 应用不需要考虑性能，而是相对于性能问题，数据一致性的问题才更加重要。

在后面的章节中我们会介绍，即使使用“范式化”的无冗余数据结构，我们借助

reselector 等工具一样可以获得很高的性能。

4.4.3　树形结构扁平

理论上，一个树形结构可以有很深的层次，但是我们在设计 Redux Store 的状态树时，要尽量保持树形结构的扁平。

如果树形结构层次很深，往往意味着树形很复杂，一个很复杂的状态树是难以管理的，如果你曾不幸开发过 Windows 操作系统中依赖于“注册表”的应用，就一定深有体会，Windows 中的“注册表”就是一个庞大而且层次很深的树形结果，看起来很灵活，实际上总让软件开发陷入麻烦的泥沼。

从代码的角度出发，深层次树形状态状态结构会让代码冗长。

假设，一个树形从上往下依次有 A、B、C、D 四个节点，为了访问节点 D，就只能通过上面三层逐级访问，不过，谁也不敢保证 A、B、C 三个节点真的存在，为了防止运行时出错，代码就要考虑到所有的可能，最后为了访问 D，代码不得不写成这样：

```
const d = state.A && state.A.B && state.A.B.C && state.A.B.C.D;
```

相信没有开发者会愿意写很多类似上面这样的代码。

4.5　Todo 应用实例

了解上述创建应用的原则之后，我们现在终于可以开始构建 Todo 应用了。

Todo 应用从界面上看应该由三部分组成：

- 待办事项的列表；
- 增加新待办事项的输入框和按钮；
- 待办事项过滤器，可以选择过滤不同状态的待办事项。

看起来需要三个功能模块，但是第一部分和第二部分的关系紧密，可以放在一个模块中，所以最后我们确定有两个功能模块 todos 和 filter，其中 todos 包含第一部分和第二部分的功能。

我们遵循“按照功能组织”的原则来设计代码，创建三个目录来容纳各自的代码文件，每个目录下都有一个 index.js 文件，这是模块的边界。各个模块之间只能假设其他模块包含 index.js 文件，要引用模块只能导入 index.js，不能够直接去导入其他文件，文件目录如下：

```
todos/
  index.js
```

```
filter/
  index.js
```

4.5.1 Todo 状态设计

至于 Todo 应用状态，从界面上看，应用中可以有很多待办事项，并有先后顺序的关系，明显用一个数组很合适。所以，我们的状态树上应该有一个表示待办事项的数组。

至于每个待办事项，应该用一个对象代表，这个对象肯定要包含文字，记录待办事项的内容，因为我们可以把一个待办事项标记为“已完成”，所以还要有一个布尔字段记录是否完成的状态，当我们把一个待办事项标记为“已完成”或者“未完成”时，必须要能唯一确定一个待办事项对象，没有规则说一个待办事项的文字必须唯一，所以我们还需要一个字段来唯一标示一个待办事项，所以一个待办事项的对象格式是这样：

```
{
  id:  // 唯一标示
  text: // 待办事项内容
  completed: // 布尔值，标示待办事项是否已完成
}
```

过滤器选项设定界面上显示什么样状态的代表事项，我们已知过滤器有三种选择：

- 全部待办事项；
- 已完成代办事项；
- 未完成待办事项。

看起来就是一个列举类型的结构，不过 JavaScript 里面并没有原生的 enum 类型支持，所以我们只能用类似常量标示符的方式来定义三种状态。在代码中，可以分别用体现语义的 ALL、COMPLETED 和 UNCOMPLETED 代表这三种状态，但是这三个标示符的实际值的选择，也值得商榷。

最简单的方式，就是让这三个状态标示符的值是是整形，比如这样的代码形式：

```
const ALL=0;
const COMPLETED=1;
const UNCOMPLETED=2;
```

但是，考虑到将来无论是 debug 还是产生 log，一个数字在开发人员眼里不容易看出来代表什么意思，最后还需要对照代码才知道 0 代表 ALL、1 代表 COMPLETED，这样很不方便。所以，开发中一个惯常的方法，就是把这些枚举型的常量定义为字符串，比如这样的代码：

```
const ALL = 'all';
const COMPLETED = 'completed';
```

```
const UNCOMPLETED = 'uncompleted';
```

综合起来看，我们知道 Todo 应用的 Store 状态树大概是这样一个样子，JavaScript 对象的表示形式如下：

```
{
  todos: [
    {
      text: 'First todo',
      completed: false,
      id: 0
    },
    {
      text: 'Second todo',
      completed: false,
      id: 1
    }
  ]
  filter: 'all'
}
```

每当增加一个待办事项，就在数组类型的 todos 中增加一个元素，当要标记一个待办事项为“已完成”或者“未完成”，就更新对应待办事项的 complete 字段值，而哪些待办事项应该显示出来，则要根据 todos 和 filter 共同决定。

在应用的入口文件 src/index.js 中，我们和 ControlPanel 一样，用 Provider 包住最顶层的 TodoApp 模块，这样让 store 可以被所有组件访问到，代码如下：

```
ReactDOM.render(
  <Provider store={store}>
    <TodoApp />
  </Provider>,
  document.getElementById('root')
);
```

而在顶层模块 src/TodoApp.js 中，所要做的只是把两个关键视图显示出来，代码如下：

```
import React from 'react';
import {view as Todos} from './todos/';
import {view as Filter} from './filter/';
function TodoApp() {
  return (
    <div>
      <Todos />
      <Filter />
    </div>
  );
```

```
}

export default TodoApp;
```

4.5.2 action 构造函数

确定好状态树的结构之后，接下来就可以写 action 构造函数了。

在 todos 和 filter 目录下，我们都要分别创造 actionTypes.js 和 actions.js 文件，这两个文件几乎每个功能模块都需要，文件如此命名是大家普遍接受的习惯。

在 src/todos/actionTypes.js 中，我们定义的是 todos 支持的 action 类型。在 Todo 应用中，支持对待办事项的增加、反转和删除三种 action 类型，代码如下：

```
export const ADD_TODO = 'TODO/ADD';
export const TOGGLE_TODO = 'TODO/TOGGLE';
export const REMOVE_TODO = 'TODO/REMOVE';
```

和 index.js 中使用命名式导出而不用默认导出一样，在 actionTypes 中我们也使用命名式导出，这样，使用 actionTypes 的文件可以这样写：

```
import {ADD_TODO, TOGGLE_TODO, REMOVE_TODO} from './actionTypes.js';
```

也同样是为了便于 debug 和输出到 log 里面查看清晰，所有的 action 类型的值都是字符串，字符串还有一个好处就是可以直接通过 === 来比较是否相等，而其他对象使用 === 则要求必须引用同一个对象。

提示 严格说来，使用 Symbol 来代替字符串表示这样的枚举值更合适，但是有的浏览器并不支持 Symbol，我们在这里不作深入探讨。

考虑到应用可以无限扩展，每个组件也要避免命名冲突。所以，最好是每个组件的 action 类型字符串值都有一个唯一的前缀。在我们的例子中，所有 todos 的 action 类型字符串值都有共同前缀“TODO/”，所有 filter 的 action 类型字符串值前缀是“FILTER/”。

在 src/todos/actions.js 中，我们定义 todos 相关的 action 构造函数，代码如下：

```
import {ADD_TODO, TOGGLE_TODO} from './actionTypes.js';

let nextTodoId = 0;

export const addTodo = (text) => ({
  type: ADD_TODO,
  completed: false,
  id: nextTodoId ++,
  text: text
});
```

```
export const toggleTodo = (id) => ({
  type: TOGGLE_TODO,
  id: id
});

export const removeTodo = (id) => ({
  type: REMOVE_TODO,
  id: id
});
```

在上一章中我们已经知道，Redux 的 action 构造函数就是创造 action 对象的函数，返回的 action 对象中必有一个 type 字段代表 action 类型，还可能有其他字段代表这个动作承载的数据。

在 src/todos/actions.js 文件中定义了一个文件级别的变量 nextTodoId，每调用一次 addTodoaction 构造函数就加一，实现为每个产生的待办事项赋予一个唯一 id 的目的。当然，这种方法非常简陋，我们在后面的章节中会改进唯一 id 的生成方法。

读者可能会注意到我们用了一种新的写法，虽然 action 构造函数应该是一个返回 action 对象的方法，我们却看不见 return 的字样。对于只 return 一个对象的函数体，ES6 允许简写为省去 return，直接用圆括号把返回的对象包起来就行，比如上面的 toggleTodo 对象构造器，实际上是下面方法的简写，代码如下：

```
export const toggleTodo = (id) => {
  return {
    type: TOGGLE_TODO,
    id: id
  }
};
```

用简写毕竟少了两行，如果能够看得习惯，我们何乐而不为。

4.5.3　组合 reducer

在 todos 和 filter 目录下，各有一个 reducer.js 文件定义两个功能模块的 reducer。

对于 reducer 我们并不陌生，在第 3 章的 ControlPanel 例子中我们就创建过 reducer。但是在那个例子中，整个应用只有一个 reducer。而在 Todo 应用中，两个功能模块都有自己的 reducer，而 Redux 的 createStore 函数只能接受一个 reducer，那么怎么办？

这是 Redux 最有意思的一部分，虽然 Redux 的 createStore 只接受一个 reducer，却可以把多个 reducer 组合起来，成为一体，然后就可以被 createStore 函数接受。

在 src/Store.js 文件中，我们完成了 reducer 的组合，代码如下：

```
import {createStore, combineReducers} from 'redux';

import {reducer as todoReducer} from './todos';
import {reducer as filterReducer} from './filter';

const reducer = combineReducers({
  todos: todoReducer,
  filter: filterReducer
});

export default createStore(reducer);
```

我们使用了 Redux 提供的一个函数 combineReducers 来把多个 reducer 函数合成为一个 reducer 函数。

combineReducers 函数接受一个对象作为参数，参数对象的每个字段名对应了 State 状态上的字段名（在上面的例子中字段名分别是 todos 和 filter），每个字段的值都是一个 reducer 函数（在上面的例子中分别是 todoReducer 和 filterReducer），combineReducers 函数返回一个新的 reducer 函数，当这个新的 reducer 函数被执行时，会把传入的 state 参数对象拆开处理，todo 字段下的子状态交给 todoReducer，filter 字段下的子状态交给 filter-Reducer，然后再把这两个调用的返回结果合并成一个新的 state，作为整体 reducer 函数的返回结果。

假设，当前 State 上的状态可以用 currentState 代表，这时候给 Store 派发一个 action 对象，combineReducers 产生的这个 reducer 函数就会被调用，调用参数 state 就是 current-State。这个 reducer 函数会分别调用 todoReducer 和 filterReducer，不过传递过去的 state 参数有些变化，调用 todoReducer 的参数 state 值是 currentState.todos，调用 filter-Reducer 的 state 是 currentState.filter，当 todoReducer 和 filterReducer 这两个函数返回结果之后，combineReducers 产生的 reducer 函数就用这两个结果分别去更新 Store 上的 todos 和 filter 字段。

所以，现在我们来看功能模块的 reducer，会发现 state 的值不是 Redux 上那个完整的状态，而是状态上对应自己的那一部分。

在 src/todos/reducer.js 中可以看到，state 参数对应的是 Store 上 todos 字段的值，默认值是一个数组，reducer 函数往往就是一个以 action.type 为条件的 switch 语句构成，代码模式如下：

```
import {ADD_TODO, TOGGLE_TODO, REMOVE_TODO}from './actionTypes.js';

export default (state = [], action) => {
  switch(action.type) {
    //针对action.type所有可能值的case语句
```

```
  }
}
```

先看对于 ADD_TODO 这种 action 类型的处理，代码如下：

```
case ADD_TODO: {
  return [
    {
      id: action.id,
      text: action.text,
      completed: false
    },
    ...state
  ]
}
```

在这里，我们使用了 ES6 的扩展操作符来简化 reducer 的代码，扩展操作符可以用来扩展一个对象，也可以用来扩展一个数组。

现在 state 是一个数组，我们想要返回一个增加了一个对象的数组，就这样写：

```
return [newObject, ...state];
```

为什么我们不直接使用熟悉的数组 push 或者 unshift 操作呢？

绝对不能，因为 push 和 unshift 会改变原来那个数组，还记得吗？ reducer 必须是一个纯函数，纯函数不能有任何副作用，包括不能修改参数对象。

对于 TOGGLE_TODO 这种 action 类型的处理，代码如下：

```
case TOGGLE_TODO: {
  return state.map((todoItem) => {
    if (todoItem.id === action.id) {
      return {...todoItem, completed: !todoItem.completed};
    } else {
      return todoItem;
    }
  })
}
```

扩展操作符可以在一对 {} 符号中把一个对象展开，这样，在 {} 中后面的部分的字段值，可以覆盖展开的部分：

```
return {...todoItem, completed: !todoItem.completed};
```

像上面的代码中，返回了一个新的对象，所有字段都和 todoItem 一样，只是 completed 字段和 todoItem 中的 completed 布尔类型值正好相反。

对于 REMOVE_TODO 这种 action 类型的处理，代码如下：

```
case REMOVE_TODO: {
  return state.filter((todoItem) => {
    return todoItem.id !== action.id;
  })
}
```

对于删除操作，我们使用数组的 filter 方法，将 id 匹配的待办事项过滤掉，产生了一个新的数组。

最后，reducer 中的 switch 语句一定不要漏掉了 default 的条件，代码如下：

```
default: {
  return state;
}
```

因为 reducer 函数会接收到任意 action，包括它根本不感兴趣的 action，这样就会执行 default 中的语句，应该将 state 原样返回，表示不需要更改 state。

在 src/filter/reducer.js 中定义了 filter 模块的 reducer，代码如下：

```
import {SET_FILTER} from './actionTypes.js';
import {FilterTypes} from '../constants.js'

export default (state = FilterTypes.ALL, action) => {
  switch(action.type) {
    case SET_FILTER: {
      return action.filter;
    }
    default:
      return state;
  }
}
```

这个 reducer 更加简单，所做的就是把 Redux Store 上 filter 字段的值设为 action 对象上的 filter 值。

我们来总结一下 Redux 的组合 reducer 功能，利用 combineReducers 可以把多个只针对局部状态的“小的”reducer 合并为一个操纵整个状态树的“大的“reducer，更妙的是，没有两个“小的”reducer 会发生冲突，因为无论怎么组合，状态树上一个子状态都只会被一个 reducer 处理，Redux 就是用这种方法隔绝了各个模块。

很明显，无论我们有多少“小的”reducer，也无论如何组合，都不用在乎它们被因为调用的顺序，因为调用顺序和结果没有关系。

4.5.4 Todo 视图

对于一个功能模块，定义 action 类型、action 构造函数和 reducer 基本上各用一个文

件就行，约定俗成地分别放在模块目录下（actionTypes.js、actions.js 和 reducer.js 文件中）。但是，一个模块涉及的视图文件往往包含多个，因为对于充当视图的 React 组件，我们往往会让一个 React 组件的功能精良、小，导致视图分布在多个文件之中。

既然有多个文件，那么也就没有太大必要保持统一的文件名，反正模块导出的只是一个 view 字段，在模块内部只要文件名能够表达视图的含义就行。

1. todos 视图

对于 todos 模块，在 index.js 中被导出的 view 存在 src/todos/views/todos.js 中，代码如下：

```
import React from 'react';
import AddTodo from './addTodo.js';
import TodoList from './todoList.js';

export default () => {
  return (
    <div className="todos">
      <AddTodo />
      <TodoList />
    </div>
  );
}
```

很简单，就是把 AddTodo 组件和 TodoList 两个组件摆放在一个 div 中，这样的简单组合自然不需要什么状态，所以用一个函数表示成无状态组件就可以了。

在 Todos 的顶层 div 上添加了 className 属性，值为字符串 todos，最后产生的 DOM 元素上就会有 CSS 类 todos。这个类是为了将来定制样式而使用的，并不影响功能。

注意　在 JSX 中指定元素的类名用的不是 HTML 里的 class 属性，而是用 className 属性，说到底 JSX 最终要转译为 JavaScript 代码，而 JavaScript 中 class 是一个保留字，所以 JavaScript 通过操作 DOM 元素的 className 属性来访问元素的类，这种不一致的确容易导致误解，但是 React 官方坚持 JSX 和 JavaScript 方法一致。

对于 AddTodo 组件，涉及处理用户的输入。当用户点击“增加”按钮或者在输入栏 input 中直接回车键的时候，要让 JavaScript 读取到 input 这个 DOM 元素的 value 值，React 为了支持这种方法，提供一个叫 ref 的功能。

在 src/todos/views/addTodo.js 中有对 AddTodo 组件的定义，我们先来看其中 render 函数对 ref 的用法，代码如下：

```
render() {
```

```
    return (
      <div className="add-todo">
        <form onSubmit={this.onSubmit}>
          <input className="new-todo" ref={this.refInput} />
          <button className="add-btn" type="submit">
            添加
          </button>
        </form>
      </div>
    )
  }
}
```

当一个包含 ref 属性的组件完成装载（mount）过程的时候，会看一看 ref 属性是不是一个函数，如果是，就会调用这个函数，参数就是这个组件代表的 DOM 元素。注意，是 DOM 元素，不是 Virtual DOM 元素，通过这种方法，我们的代码可以访问到实际的 DOM 元素。

AddTodo 的 render 函数渲染出来了一个 form，通过 onSubmit 属性把 form 被提交的事件挂在 AddTodo 组件的 onSubmit 函数上。

在上面的例子中，input 元素的 ref 属性是 AddTodo 组件的一个成员函数 refInput，所以当这个 input 元素完成装载之后，refInput 会被调用。

refInput 的函数代码如下：

```
refInput(node) {
  this.input = node;
}
```

当 refInput 被调用时，参数 node 就是那个 input 元素，refInput 把这个 node 记录在成员变量 input 中。

于是，当组件完成 mount 之后，就可以通过 this.input 来访问那个 input 元素。这是一个 DOM 元素，可以使用任何 DOM API 访问元素内容，通过访问 this.input.value 就可以直接读取当前的用户输入。

使用 this.input 的 onSubmit 函数代码如下：

```
onSubmit(ev) {
  ev.preventDefault();

  const input = this.input;
  if (!input.value.trim()) {
    return;
  }
```

```
    this.props.onAdd(input.value);
    input.value = '';
  }
```

在 HTML 中，一个 form 表单被提交的默认行为会引发网页跳转，在 React 应用中当然不希望网页跳转发生，所以我们首先通过调用参数 ev 的 preventDefault 函数取消掉浏览器的默认提交行为。

通过 this.input.value，可以判断出当前输入框里的文字内容。如果是空白，就应该忽略，因为创建一个空白的待办事项没有意义。否则，就调用通过属性 onAdd 传递进来的函数。最后，我们把 input 清空，让用户知道创建成功，而且方便创建下一条待办事项。

文件中的 AddTodo 组件是一个内部组件，按说应该是一个傻瓜组件，但是和我们之前例子中的傻瓜组件不一样，AddTodo 不是一个只有一个 render 函数的组件，AddTodo 甚至有一个逻辑比较复杂的 onSubmit 函数，为什么不把这部分逻辑提取到外面的容器组件中呢?

其实我们可以把 onSubmit 的逻辑提取到 mapDispatchToProps 中。但是，让 AddTodo 组件外面的 mapDispatchToProps 访问到 AddTodo 组件里面的 ref 很困难，得不偿失。

使用 ref 实际上就是直接触及了 DOM 元素，与我们想远离 DOM 是非之地的想法相悖，虽然 React 提供了这个功能，但是还是要谨慎使用，如果要用，我们也尽量让 ref 不要跨越组件的边界。

所以，我们把通过 ref 访问 input.value 放在内部的 AddTodo 中，但是把调用 dispatch 派发 action 对象的逻辑放在 mapDispatchToProps 中，两者一个主内一个主外，各司其职，不要混淆。

注意　并不是只有 ref 一种方法才能够访问 input 元素的 value，我们在这里使用 ref 主要是展示一下 React 的这个功能，在后面的章节中我们会介绍更加可控的方法。

注意，对于 AddTodo，没有任何需要从 Redux Store 的状态衍生的属性，所以最后一行的 connect 函数第一个参数 mapStateToProps 是 null，只是用了第二个参数 mapDispatchToProps。

在 src/todos/views/addTodo.js 中表示 AddTodo 标示符代表的组件和 src/todos/views/todos.js 中 AddTodo 标示符代表的组件不一样，后者是前者用 react-redux 包裹之后的容器组件。

接下来看看待办事项列表组件，定义在 src/todos/view/todoList.js 中，在渲染 TodoList 时，我们的 todos 属性是一个数组，而数组元素的个数是不确定的，这就涉及如何渲染数

量不确定子组件的问题，TodoList 作为无状态组件代码如下：

```
const TodoList = ({todos, onToggleTodo, onRemoveTodo}) => {
  return (
    <ul className="todo-list">
    {
      todos.map((item) => (
        <TodoItem
          key={item.id}
          text={item.text}
          completed={item.completed}
          onToggle={() => onToggleTodo(item.id)}
          onRemove={() => onRemoveTodo(item.id)}
        />
      ))
    }
    </ul>
  );
};
```

这段代码中的 TodoList 真的就只是一个无状态的傻瓜组件了，对于传递进来的 todos 属性，预期是一个数组，通过一个数组的 map 函数转化为一个 TodoItem 组件的数组，这是我们第一次在 JSX 代码片段中创建动态数量的元素。

因为 todos 这个数组的元素数量并不确定，所以很自然想到要用一个循环来产生同样数量的 TodoItem 组件实例。但是，并不能在 JSX 中使用 for 或者 while 这样的循环语句。因为，JSX 中可以使用任何形式的 JavaScript 表达式，只要 JavaScript 表达式出现在符号 {} 之间，但是也只能是 JavaScript“表达式”，for 或者 while 产生的是“语句”而不是“表达式”，所以不能出现 for 或者 while。

归根到底，JSX 最终会被 babel 转移成一个嵌套的函数调用，在这个函数调用中自然无法插入一个语句进去，所以，当我们想要在 JSX 中根据数组产生动态数量的组件实例，就应该用数组的 map 方法。

还有一点很重要，对于动态数量的子组件，每个子组件都必须要带上一个 key 属性，而且这个 key 属性的值一定要是能够唯一标示这个子组件的值。

当我们完成 Todo 应用之后，会回过头来再解释 key 值的作用。

TodoList 的 mapStateToProps 方法需要根据 Store 上的 filter 状态决定 todos 状态上取哪些元素来显示，这个过程涉及对 filter 的 switch 判断。为了防止 mapStateToProps 方法过长，我们将这个逻辑提取到 selectVisibleTodos 函数中，代码如下：

```
const selectVisibleTodos = (todos, filter) => {
  switch (filter) {
    case FilterTypes.ALL:
```

```
      return todos;
    case FilterTypes.COMPLETED:
      return todos.filter(item => item.completed);
    case FilterTypes.UNCOMPPLETED:
      return todos.filter(item => !item.completed);
    default:
      throw new Error('unsupported filter');
  }
}

const mapStateToProps = (state) => {
  return {
    todos: selectVisibleTodos(state.todos, state.filter)
  };
}
```

最后，我们看 TodoList 的 mapDispatchToProps 文件，代码如下：

```
const mapDispatchToProps = (dispatch) => {
  return {
    onToggleTodo: (id) => {
      dispatch(toggleTodo(id));
    },
    onRemoveTodo: (id) => {
      dispatch(removeTodo(id));
    }
  };
};
```

在 TodoList 空间中，看 mapDispatchToProps 函数产生的两个新属性 onToggleTodo 和 onRemoveTodo 的代码遵循一样的模式，都是把接收到的参数作为参数传递给一个 action 构造函数，然后用 dispatch 方法把产生的 action 对象派发出去，这看起来是重复代码。

实际上，Redux 已经提供了一个 bindActionCreators 方法来消除这样的重复代码，显而易见很多 mapDispatchToProps 要做的事情只是把 action 构造函数和 prop 关联起来，所以直接以 prop 名为字段名，以 action 构造函数为对应字段值，把这样的对象传递给 bindActionCreators 就可以了，上面的 mapDispatchToProps 可以简写为这样：

```
import {bindActionCreators} from 'redux';

const mapDispatchToProps = (dispatch) => bindActionCreators({
  onToggleTodo: toggleTodo,
  onRemoveTodo: removeTodo
}, dispatch);
```

更进一步，可以直接让 mapDispatchToProps 是一个 prop 到 action 构造函数的映射，这样连 bindActionCreators 函数都不用，代码如下：

```
const mapDispatchToProps = {
  onToggleTodo: toggleTodo,
  onRemoveTodo: removeTodo
};
```

上面定义的 mapDispatchToProps 传给 connect 函数，产生的效果和之前的写法完全一样。

我们再来看定义了 TodoItem 的 src/todos/views/todoItem.js 文件，代码如下：

```
const TodoItem = ({onToggle, onRemove, completed, text}) => (
  <li
    className="todo-item"
    style={{
      textDecoration: completed ? 'line-through' : 'none'
    }}
  >
    <input className="toggle" type="checkbox" checked={completed ? "checked" :
      ""} readOnly onClick={onToggle} />
    <label className="text">{text}</label>
    <button className="remove" onClick={onRemove}>×</button>
  </li>
)
```

这里 TodoItem 就是一个无状态的组件，把 onToggle 属性挂到 checkbox 的点击事件上，把 onRemove 属性挂到删除按钮的点击事件上。

每个待办事项都包含一个 checkbox 元素和一段文字内容，checkbox 是否勾选并不依赖用户输入，而是根据 completed 属性来判断。同时，对于 completed 状态的待办事项，文字内容中间用横线代表完成。

2. filter 视图

对于过滤器，我们有三个功能类似的链接，很自然就会想到把链接相关的功能提取出来，放在一个叫 Link 的组件中。

在 src/filter/views/filter.js 中，我们定义被导出的 filter 主视图，代码如下：

```
const Filter = () => {
  return (
    <p className="filters">
      <Link filter={FilterTypes.ALL}> {FilterTypes.ALL} </Link>
      <Link filter={FilterTypes.COMPLETED}> {FilterTypes.COMPLETED} </Link>
      <Link filter={FilterTypes.UNCOMPPLETED}> {FilterTypes.UNCOMPPLETED} </Link>
    </p>
  );
};
```

这个 filter 主视图很简单，是一个无状态的函数，列出了三个过滤器，把实际工作都

交给了 Link 组件。

我们在 src/filter/views/link.js 中添加 Link 组件，代码如下：

```
const Link = ({active, children, onClick}) => {
  if (active) {
    return <b className="filter selected">{children}</b>;
  } else {
    return (
      <a href="#" className="filter not-selected" onClick={(ev) => {
        ev.preventDefault();
        onClick();
      }}>
        {children}
      </a>
    );
  }
};
```

作为傻瓜组件的 Link，当传入属性 active 为 true 时表示当前实例就是被选中的过滤器，不该被再次选择，所以不应该渲染超链接标签 a。若否，则渲染一个超链接标签。

一个超链接标签的默认行为是跳转，即 href 属性是 #，所以超链接标签的 onClick 事件处理器第一件事就是用 preventDefault 函数取消默认行为。实际上，我们把 href 设为“#”唯一的目的就是让浏览器给超链接显示出下划线，代表这是一个链接。

我们使用了一个特殊的属性 children，对于任何一个 React 组件都可以访问这样一个属性，代表的是被包裹住的子组件，比如，对于下面的代码：

```
<Foo>

  <Bar>WhatEver</Bar>

</Foo>
```

Foo 组件实例的 children 就是 <Bar>WhatEver</Bar>，而 Bar 组件实例的 children 就是 WhatEver。在 render 函数中把 children 属性渲染出来，是惯常的组合 React 组件的方法。

Link 的 mapStateToProps 和 mapDispatchToProps 函数都很简单，代码如下：

```
const mapStateToProps = (state, ownProps) => {
  return {
    active: state.filter === ownProps.filter
  }
};

const mapDispatchToProps = (dispatch, ownProps) => ({
  onClick: () => {
    dispatch(setFilter(ownProps.filter));
```

```
  }
});
```

4.5.5 样式

我们终于完成了 Todo 应用，在浏览器中可以看到最终效果，如图 4-1 所示。

图 4-1 无定制样式的 Todo 应用界面

这个 Todo 应用功能已经完成，但是并没有定制样式，还是需要通过 CSS 来添加样式。我们在定义视图部分代码时，一些元素上通过 className 属性添加了 CSS 类，现在我们就利用这些类来定义 CSS 规则。

在 src/todos/views/style.css 中，我们定义了 Todo 空间中所有的样式，为了让定义的样式产生效果，在 Todos 组件的最顶层视图文件 src/todos/views/todos.js 中添加下列代码：

```
import './style.css';
```

在 React 应用中，通常使用 webpack 来完成对 JavaScript 的打包，create-react-app 产生的应用也不例外，不过 webpack 不只能够处理 JavaScript 文件，它能够处理 CSS、SCSS 甚至图片文件，因为在 webpack 眼里，一切文件都是“模块”，通过文件中的 import 语句或者 require 函数调用就可以找到文件之间的使用关系，只要被 import 就会被纳入最终打包的文件中，即使被 import 的是一个 CSS 文件。

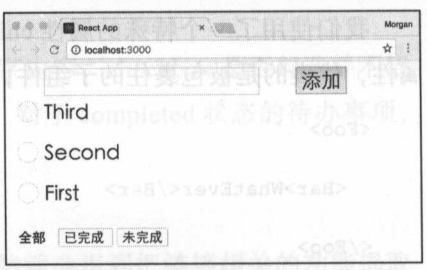


图 4-2 完成定制样式的 Todo 应用界面

如图 4-2 所示，Todo 应用终于拥有样式了。

把 CSS 文件用 import 语句导入，webpack 默认的处理方式是将 CSS 文件的内容嵌入最终的打包文件 bundle.js 中，这毫无疑问会让打包文件变得更大，但是应用所有的逻辑都被包含在一个文件中了，便于部署。

当然，如果不希望将 CSS 和 JavaScript 混在一起，也可以在 webpack 中通过配置完成，在 webpack 的 loader 中使用 extract-text-webpack-plugin，就可以让 CSS 文件在 build 时被放在独立的 CSS 文件中，在第 11 章会介绍定制 webpack 配置的方法。

有意思的是，选择 CSS 和 JavaScript 打包在一起还是分开打包，和代码怎么写没有任何关系，这就是 React 的妙处。代码中只需要描述“想要什么”，至于最终“怎么做”，可以通过配置 webpack 获得多重选择。

如果使用 SCSS 语法，可以简化上面的样式代码，但是 create-react-app 产生的应用默认不支持 SCSS，有兴趣的读者可以通过 eject 方法直接编辑 webpack 配置，应用上 SCSS 加载器。

4.5.6　不使用 ref

在前面的例子中，代码通过 React 的 ref 功能来访问 DOM 中元素，这种功能的需求往往来自于提交表单的操作，在提交表单的时候，需要读取当前表单中 input 元素的值。

毫无疑问，ref 的用法非常脆弱，因为 React 的产生就是为了避免直接操作 DOM 元素，因为直接访问 DOM 元素很容易产生失控的情况，现在为了读取某个 DOM 元素的值，通过 ref 取得对元素的直接引用，不得不说，干得并不漂亮。

有没有更好的办法呢？

有，可以利用组件状态来同步记录 DOM 元素的值，这种方法可以控制住组件不使用 ref。

Todo 应用中的 addTodo.js 文件可以这样修改，首先是 render 函数，代码如下：

```
render() {
  return (
    <div className="add-todo">
      <form onSubmit={this.onSubmit}>
        <input  className="new-todo"  onChange={this.onInputChange}  value={this.
          state.value} />
        <button className="add-btn" type="submit">
          添加
        </button>
      </form>
    </div>
  )
}
```

在这里我们不再使用 ref 来获取对 input 元素的直接访问，而是利用 input 元素上的 onChange 属性挂上一个事件处理函数 onInputChange，这样每当 input 的值发生变化时，onInputChange 函数就会被调用，这样我们总有机会记录下最新的 input 元素内容。

onInputChange 函数的代码如下：

```
onInputChange(event) {
  this.setState({
    value: event.target.value
  });
}
```

在 onInputChange 函数中，通过传入的事件参数 event 可以访问到事件发生的目标，读取到当前值，并把内容存在组件状态的 value 字段上。这样，组件状态上总能够获得最新的 input 元素内容。

然后看 onSubmit 函数的改变，代码如下：

```
onSubmit(ev) {
  ev.preventDefault();

  const inputValue = this.state.value;
  if (!inputValue.trim()) {
    return;
  }

  this.props.onAdd(inputValue);
  this.setState({value: ''});
}
```

当需要提交表单的时候，onSubmit 函数被调用，只需要直接从组件状态上读取 value 字段的值就足够了。

在产品开发中，应该尽量避免 ref 的使用，而换用这种状态绑定的方法来获取元素的值。

4.6 开发辅助工具

工欲善其事，必先利其器。——《论语·卫灵公》

我们已经写了一个比较复杂的基于 React 和 Redux 的应用，将来还会遇到更复杂的应用，是时候引入一些开发辅助工具了，毕竟没有人能够一次把复杂代码写对，必要的工具能够大大提高我们的工作效率。

4.6.1 Chrome 扩展包

Chrome 是网页应用开发者群体最喜爱的浏览器，因为 Chrome 浏览器有丰富的扩展库来帮助网页开发，这里我们介绍三款 Chrome 扩展包，可以说是 React 和 Redux 应用开发必备之物。

- React Devtools，可以检视 React 组件的树形结构，下载地址如下：https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi
- Redux Devtools，可以检视 Redux 数据流，可以将 Store 状态跳跃到任何一个历

史状态，也就是所谓的“时间旅行”功能，下载地址如下：https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd

- React Perf，可以发现 React 组件渲染的性能问题，下载地址如下：https://chrome.google.com/webstore/detail/react-perf/hacmcodfllhbnekmghgdlplbdnahmhmm

如果读者访问谷歌浏览器商店有困难，那就访问 http://chrome-extension-downloader.com/ ，在输入框中输入谷歌浏览器商店官方地址后面那一串字符串，比如对应于 React Devtools 的字符串就是 fmkadmapgofadopljbjfkapdkoienihi，点击“下载扩展包”按钮，下载完成之后安装即可。

4.6.2　redux-immutable-state-invariant 辅助包

我们曾经反复强调过，每一个 reducer 函数都必须是一个纯函数，不能修改传入的参数 state 和 action，否则会让应用重新陷入状态不可预料的境地。

禁止 reducer 函数修改参数，这是一个规则，规则总是会被无心违反的，但是怎么避免开发者不小心违反这个规则呢？

有一个 redux-immutable-state-invariant 包，提供了一个 Redux 的中间件，能够让 Redux 在每次派发动作之后做一个检查。如果发现某个 reducer 违反了作为一个纯函数的规定擅自修改了参数 state，就会给一个大大的错误警告，从而让开发者意识到自己犯了一个错误，必须要修正。

什么是 Redux 的中间件？我们在后面的章节会详细介绍，在这里我们只要理解中间件是可以增强 Redux 的 Store 实例功能的一种方法就可以。

很明显，对于 redux-immutable-state-invariant 的这种检查，在开发环境下很有用。但是在产品环境下，这样的出错提示意义不大，只是徒耗计算资源和增大 JavaScript 代码体积，所以我们通常在产品环境中不启用 redux-immutable-state-invariant。

4.6.3　工具应用

上面我们介绍了辅助开发的 Chrome 扩展包和 redux-immutable-state-invariant 库，对于 React Devtools 来说，启用只是安装一个 Chrome 扩展包的事，但是对于其余几个工具，我们的代码要做一些修改才能配合浏览器使用。

因为 Store 是 Redux 应用的核心，所以所有的代码修改都集中在 src/Store.js 中。

首先需要从 Redux 引入多个函数，代码如下：

```
import {createStore, combineReducers, applyMiddleware, compose} from 'redux';
```

为了使用 React Perf 插件，需要添加如下代码，代码如下：

```
import Perf from 'react-addons-perf'

const win = window;
win.Perf = Perf
```

在这里把 window 赋值给模块级别变量 win，是为了帮助代码缩小器（minifer），在 webpack 中缩小代码的插件叫 UglifyJsPlugin，能够将局部变量名改成很短的变量名，这样功能不受影响但是代码的大小大大缩减。

不过，和所有的代码缩小器一样，UglifyJsPlugin 不敢去改变全局变量名，因为改了就会影响程序的功能。所以当多次引用 window 这样的全局变量时，最好在模块开始将 window 赋值给一个变量，比如 win，然后在代码其他部分只使用 win，这样最终经过 UglifyJsPlugin 产生的缩小代码就只包含一个无法缩小的 window 变量。

我们给 win 上的 Perf 赋值了从 react-addons-perf 上导入的内容，这是为了帮助 React Perf 扩展包的使用，做了这个代码修改之后，React Perf 上的 Start 按钮和 Stop 按钮才会起作用。

图 4-3 就是 React Perf 的界面。

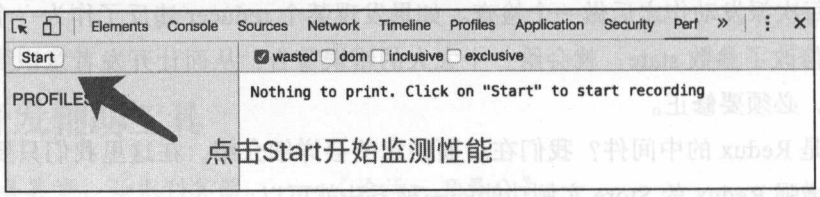


图 4-3 React Perf 工具界面

为了应用 redux-immutable-state-invariant 中间件和 Redux Devtools，需要使用 Redux 的 Store Enhancer 功能，代码如下：

```
const reducer = combineReducers({
  todos: todoReducer,
  filter: filterReducer
});

const middlewares = [];
if (process.env.NODE_ENV !== 'production') {
  middlewares.push(require('redux-immutable-state-invariant')());
}

const storeEnhancers = compose(
  applyMiddleware(...middlewares),
  (win && win.devToolsExtension) ? win.devToolsExtension() : (f) => f,
```

```
);

export default createStore(reducer, {}, storeEnhancers);
```

代码修改的关键在于给 createStore 函数增加了第三个参数 storeEnhaners，这个参数代表 Store Enhancer 的意思，能够让 createStore 函数产生的 Store 对象具有更多更强的功能。

因为 Store Enhancer 可能有多个，在我们的例子中就有两个，所以 Redux 提供了一个 compose 函数，用于把多个 Store Enhancer 组合在一起，我们分别来看这两个 Store Enhancer 是什么。

第一个 Store Enhancer 是一个函数 applyMiddleware 的执行结果，这个函数接受一个数组作为参数，每个元素都是一个 Redux 的中间件。虽然目前我们只应用了一个中间件，但是考虑到将来会扩展更多，所以我们用数组变量 middlewares 来存储所有的中间件，将来要扩展新的中间件只需要往这个数组中 push 新的元素就可以了。

目前，往 middlewares 中 push 的唯一一个中间件就是 redux-immutable-state-invariant。如同上面解释过的，redux-immutable-state-invariant 只有在开发环境下使用才有意义，所以只有当 process.env.NODE_ENV 不等于 production 时才加入这个中间件。

我们一直按照 ES6 的语法导入模块，也就是用 import 关键字导入模块，但是在这里却用了 Node 传统风格的 require，是因为 import 语句不能够存在于条件语句之中，只能出现在模块语句的顶层位置。

第二个 Store Enhancer 就是 Redux Devtools，因为 Redux Devtools 的工作原理是截获所有应用中 Redux Store 的动作和状态变化，所以必须通过 Store Enhancer 在 Redux Store 中加入钩子。

如果浏览器中安装了 Redux Devtools，在全局 window 对象上就有一个 devToolsExtension 代表 Redux Devtools。但是，我们也要让没有安装这个扩展包的浏览器也能正常使用我们的应用，所以要根据 window. devToolsExtension 是否存在做一个判断，如果有就用，如果没有就插入一个什么都不做的函数。

```
(win && win.devToolsExtension) ? win.devToolsExtension() : (f) => f,
```

当所有的代码修改完毕，重新启动 Todo 应用，在浏览器网页里的开发者工具中就可以用上所有关于 React 和 Redux 的工具了。

4.7　本章小结

在这一章，我们学习了构建一个完整 Redux 应用的步骤和需要考虑的方面。

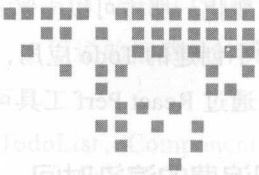
首先，要考虑代码文件的组织方式，对于可以高度模块化的 Redux 应用，使用“按功能组织”的方式要优于传统 MVC 框架下的“按角色组织”的方式。

之后，要考虑 Store 上状态树的设计，因为状态树的结构直接决定了模块的划分，以及 action 类型、action 构造函数和 reducer 的设计。可以说，开始写 Redux 应用第一行代码之前，首先要想好 Store 的状态树长得什么样子。

然后，我们实际构建了一个 Todo 应用，这个应用要比之前的 ControlPanel 应用复杂，利用划分模块的方法解决才是正道，从中我们也学习了 React 的 ref 功能，以及动态数量的子空间必须要包含 key 属性。

最后，我们了解了开发 React 和 Redux 应用必备的几样辅助工具，有了这几件工具，开发 React 和 Redux 应用就会如虎添翼。

这只是一个起点，在接下来的章节中，我们会进一步深入了解 React 和 Redux 的精髓。



第 5 章 Chapter 5

React 组件的性能优化

开发者不仅要让程序能够完成功能，还要让程序运行得足够快。

虽然 React 已经提供了很好的渲染性能，但是要让程序性能发挥到极致，就需要进一步了解性能优化的方法。

这一章中会介绍如下内容：

- 单个 React 组件的性能优化；
- 多个 React 组件的性能优化；
- 利用 reselect 提高数据选取的性能。

但是，这并不是提高性能的全部，一个网页应用的性能由多方面因素决定，其他章节也会涉及性能提高的内容，第 10 章中会介绍动画效果的性能优化，在第 11 章中会介绍代码打包的优化。不过，在这一章中，重点关注的是 React 组件的渲染性能优化。应用由组件构成，要提高应用性能，必须先提高组件性能。

5.1 单个 React 组件的性能优化

React 利用 Virtual DOM 来提高渲染性能，虽然每一次页面更新都是对组件的重新渲染，但是并不是将之前渲染的内容全部抛弃重来，借助 Virtual DOM，React 能够计算出对 DOM 树的最少修改，这就是 React 默认情况下渲染都很迅捷的秘诀。

不过，虽然 Virtual DOM 能够将每次 DOM 操作量减少到最小，计算和比较 Virtual DOM 依然是一个复杂的计算过程。如果能够在开始计算 Virtual DOM 之前就可以判断渲

染结果不会有变化，那样可以干脆不要进行 Virtual DOM 计算和比较，速度就会更快。

在第 4 章中创建的 Todo 应用，其实存在一个影响性能的设计，没有充分发挥 React 的性能优势，通过 React Perf 工具可以发现这个问题。

5.1.1 发现浪费的渲染时间

为了发现这个问题，需要在 Chrome 浏览器中安装 React Perf 扩展，在第 4 章第 6 节介绍了安装 React Perf 的方法。

在 Chrome 浏览器中打开 Todo 应用的页面，执行下列步骤可以发现性能问题所在。

1）添加两个待办事项，文字分别为 First 和 Second；

2）打开开发者工具，切换到 Perf，这是 React Perf 工具界面，确保 wasted 选项被勾选；

3）点击 React Perf 工具左侧的 Start 按钮，开始性能测量，这时按钮会变成 Stop 按钮；

4）勾选 Todo 应用中的 First 那一项，让它变成完成状态；

5）点击 React Perf 工具左侧的 Stop 按钮，结束性能测量，界面上会显示测量结果。

完成上面的步骤之后，应该可以看到如图 5-1 的界面。

图 5-1 React Perf 发现 Todo 应用中的浪费渲染时间

在 React Perf 工具中，可以看见发现了浪费的渲染过程，React Perf 工具记录在点击

Start 按钮和 Stop 按钮之间的所有 React 渲染，如果有组件计算 Virtual DOM 之后发现和之前的 Virtual DOM 相同，那就认为是一次浪费。注意，这里说的浪费是计算 Virtual DOM 的浪费，并不是访问 DOM 树的浪费。

在上面的例子中，发生浪费的那一条 Owner 是 TodoList，Component 是 TodoItem，意思就是父组件 TodoList 渲染一个子组件 TodoItem 是浪费的。

在这个操作过程中，页面上只有 First 和 Second 两个 TodoItem，First 被标记为“完成”状态，对应的界面有改变，所以不可能有浪费；而 Second 没有界面改变，所以浪费只能是渲染 Second 导致的。

5.1.2　性能优化的时机

在 React Perf 的界面中可以在 Inclusive wasted time 这一项中看到浪费掉的时间，这一次浪费了 0.05 毫秒，在页面中 0.05 毫秒是一段不起眼的时间，用户完全感觉不到。

如果说要优化代码避免这样的时间浪费，读者可能觉得没有必要。而且，前辈们已经提醒过我们不要过早进行所谓性能优化，读者可能知道《计算机编程艺术》作者下面的一句名言。

“过早的优化是万恶之源。”——高德纳

表面上看，这句话是说除非性能出了问题，不然就不要去花时间去优化性能。

照这么说，为了一个 0.05 毫秒的时间浪费似乎不值得去优化，但是，目前一个 TodoItem 组件的结构很简单，当一个被浪费的组件很复杂时，这样浪费的时间就会更长，更重要的是，这只是一个组件的浪费时间，而这个组件的数量是可变的，假设 TodoList 包含 1000 个待办事项，那就有 1000 个子组件 TodoItem 组件，每一次用户只能反转一个 TodoItem 组件的完成状态，也就是说会有 999 个 TodoItem 的渲染实际上是浪费掉的，那就浪费掉大约 50 毫秒的时间，这已经是一个能够被用户感知到的延迟时间，如果 TodoItem 增加更丰富的功能，产生的 DOM 结构更复杂，浪费的时间会更多。

难道前辈们说的话是错误的吗？并不是，只是前辈们的话被断章取义了，这句话的全文其实是这样[⊖]。

“我们应该忘记忽略很小的性能优化，可以说 97% 的情况下，过早的优化是万恶之源，而我们应该关心对性能影响最关键的那另外 3% 的代码。”

——高德纳

⊖ Structured Programing with go to Statements, Donald E. Kunth，http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.6084&rep=rep1&type=pdf

看到了全文，就会对这句话有个全新的认识。

高德纳并没有否认性能优化的必要，而是说不要将性能优化的精力浪费在对整体性能提高不大的代码上，而对于性能有关键影响的部分，优化并不嫌早。因为，对性能影响最关键的部分，往往涉及解决方案核心，决定整体的架构，将来要改变的时候牵扯更大。

对于合并多个字符串，是直接使用 + 操作符合并？还是将所有字符串放在一个数组之后用 join 方法合并？不同 JavaScript 引擎和不同数量的字符串数量下，性能结果可能不一样，但是使用什么方法不大可能对整个应用造成关键的性能影响，这就是高德纳说的 97% 的情况；而选择用什么样的方式定义组件的接口，如何定义 state 到 prop 的转变，使用什么样的算法来比对 Virtual DOM, 这些决定对性能和架构的影响是巨大，就是那关键的 3%。

所谓的“过早的优化”，指的是没有任何量化证据情况下开发者对性能优化的猜测，没有可测量的性能指标，就完全不知道当前的性能瓶颈在何处，完成优化之后也无法知道性能优化是否到了预期的结果。虽然我们也不知道高德纳在这句话里如何量化 97% 和 3% 这两个数字，但是在自己的工作中一定要有量化的性能指标，在上面的例子中，React Perf 已经对 wasted time 这个指标给出了精确到毫秒小数点后两位的量化结果。

总之，虽然现在我们发现的浪费时间只有 0.05 毫秒，但这并不能成为我们无视这个性能问题的理由。

对于实例数量是动态的组件，一点点的浪费时间会因为实例数量增多而累积成很长的时间。开发者应该能认识到实际产品中会出现比开发过程中复杂得多的使用情况，还是以 Todo 应用为例，开发者每一次测试可能只有几个 TodoItem，但是实际用户可能会创建成百上千的 TodoItem，开发者永远要考虑到极端情况。

接下来，我们就来进一步改进这个 Todo 应用，其中会引入 React 组件的构建模式，记住这些模式，自然就会避免渲染的浪费时间。这也许会被认为是“过早的优化”，但是已经成为模式的优化方式，有利于代码的可读性。高德纳认为过早的优化是万恶之源，是因为这些优化往往让代码过于复杂而且难以维护，但是如果早期的优化能够让代码结构更加合理更加容易维护，何乐而不为呢？

5.1.3 React-Redux 的 shouldComponentUpdate 实现

在前面介绍过，shouldCompnoentUpdate 可能是 React 组件生命周期函数中除了 render 函数之外最重要的一个函数了，render 函数决定了“组件渲染出什么”，而 shouldComponentUpdate 函数则决定“什么时候不需要重新渲染”。

React 组件类的父类 Component 提供了 shouldComponentUpdate 的默认实现方式，但

这个默认实现就是简单地返回一个 true。也就是说，默认每次更新的时候都要调用所有的生命周期函数，包括调用 render 函数，根据 render 函数的返回结果计算 Virtual DOM。

当然，默认方法是一个兜底的保险方法。毕竟 React 并不知道各个组件的细节，把组件重新渲染一遍就算浪费，至少绝对不会出错。

但是要达到更好的性能，有必要定义好我们的组件的 shouldComponentUpdate 函数，让它在必要的时候返回 false，告诉 React 不用继续更新，就会节省大量的计算资源。每个 React 组件的内在逻辑都有自己的特点，需要根据组件逻辑来定制 shouldComponentUpdate 函数的行为。

之所以出现图 5-1 所示的浪费时间，就是因为 TodoItem 是一个无状态函数，所以使用的是 React 默认的 shouldComponentUpdate 函数实现，也就是永远返回 true 的实现。

TodoList 在重复渲染时，会引发所有 TodoItem 子组件的更新过程，即使传递给这些子组件的 prop 没有任何变化。既然 TodoItem 的 shouldComponentUpdate 行为是返回 true，那 React 也就会调用更新过程中所有的生命周期函数，产生 Virtual DOM，但是最后通过 Virtual DOM 的比对发现 Virtual DOM 没有变化，其实根本不需要修改 DOM。

看起来，要做的就是给 TodoItem 组件增加定制的 shouldComponentUpdate 函数，这样就会让 TodoItem 代码从一个独立的函数变成一个 ES6 class，增加的 shouldComponentUpdate 函数代码如下：

```
shouldComponentUpdate(nextProps, nextState) {

  return (nextProps.completed !== this.props.completed) ||

    (nextProps.text !== this.props.text);

}
```

因为对于一个 TodoItem 组件而言，影响渲染内容的 prop 只有 completed 和 text，只要确保这两个 prop 没有变化，shouldComponentUpdate 函数就可以返回 false。

如果每个 React 组件都需要定制自己的 shouldComponentUpdate 函数，从写代码的角度来看也是一件很麻烦的事情。但是如果使用了 react-redux 库，一切都很简单。

回顾一下前面 ControlPanel 和 Todo 应用的例子，使用 react-redux 库，我们把完成一个功能的 React 组件分成两部分：

- 第一部分是一个傻瓜组件，只管负责视图部分，处理的是“组件看起来怎样”的事情。这个傻瓜组件往往用一个函数的无状态组件就足够表示，甚至都不需要是一个类的形态，只需要定义一个函数就足够。
- 第二部分是一个容器组件，负责逻辑部分，处理的是“组件如何工作”的事情，

这个容器组件有状态，而且保持和 Redux Store 上状态的同步，但是 react-redux 的 connect 函数把这部分同步的逻辑封装起来了，我们甚至在代码中根本看不见这个类的样子，往往直接导出 connect 返回函数的执行结果就行。

使用 react-redux，一个典型的 React 组件代码文件最后一个语句代码是这样：

```
export default connect(mapStateToProps, mapDispatchToProps)(Foo);
```

可以看到，往往都没有必要把产生的容器组件赋值给一个变量标示符，直接把 connect 的结果 export 导出就可以了。

虽然代码上不可见，但是 connect 的过程中实际上产生了一个无名的 React 组件类，这个类定制了 shouldComponentUpdate 函数的实现，实现逻辑是比对这次传递给内层傻瓜组件的 props 和上一次的 props，因为负责“组件看起来怎样”的傻瓜视图是一个无状态组件，它的渲染结果完全由传入的 props 决定，如果 props 没有变化，那就可以认为渲染结果肯定一样。

例如，我们有一个 Foo 组件，代码如下：

```
import React, {PropTypes} from 'react';
import {connect} from 'react-redux';

const Foo = ({text}) => (
  <div>{text}</div>
)

const mapStateToProps = (state) => (
  text: state.text
);
export default connect(mapStateToProps)(Foo);
```

这个组件简单得实在不能更简单，就是把 Redux Store 的状态树上的 text 字段显示出来。内部的傻瓜组件 Foo 只有一个 props 属性 text，通过 react-redux 的 connect 方法，这个文件导出的是封装过的容器组件。

在这个例子中，导出的容器组件的 shouldComponentUpdate 所做的事情，就是判断这一次渲染的 text 值和上一次的 text 值是否相同，如果相同，那就没有必要重新渲染了，可以返回 false；否则就要返回 true。

这样，我们编写的 Foo 依然是一个无状态组件，但是当要渲染 Foo 组件实例时，只要 Redux Store 上的对应 state 没有改变，Foo 就不会经历无意义的 Virutal DOM 产生和比对过程，也就避免了浪费。

同样的方法也可以应用在 TodoItem 组件上。不过，因为 TodoItem 没有直接从 Redux Store 上读取状态，但我们依然可以使用 react-redux 方法，只是 connect 函数的调

用不需要任何参数，要做的只是将定义 TodoItem 组件的代码最后一行改成如下代码，代码如下：

```
export default connect()(TodoItem);
```

在上面的例子中，在 connect 函数的调用没有参数，没有 mapStateToProps 和 mapDispatchToProps 函数，使用 connect 来包裹 TodoItem 的唯一目的就是利用那个聪明的 shouldComponentUpdate 函数。

使用 React Perf 重复检查性能，可以发现并没有真正解决问题，依然存在渲染浪费的问题，如图 5-2 所示。

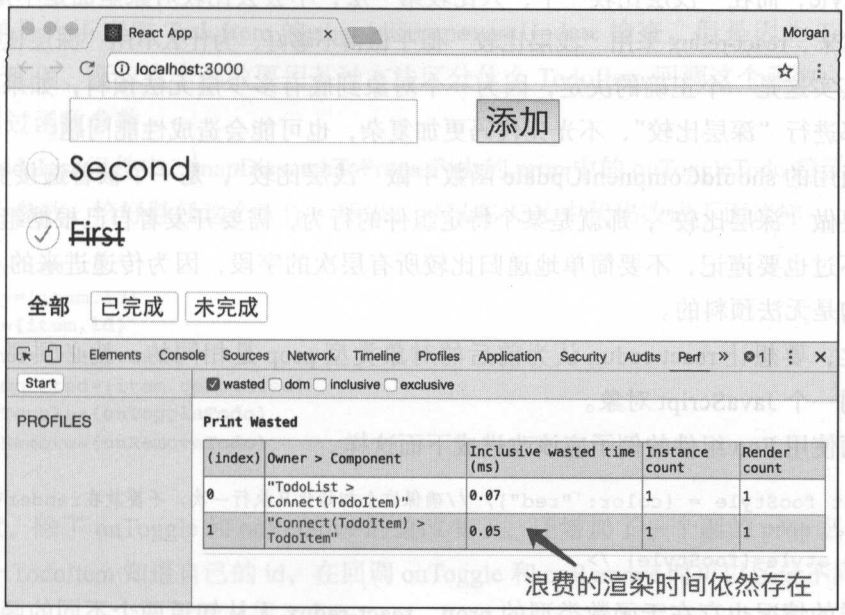


图 5-2　使用 connect 之后依然存在渲染时间浪费

和图 5-1 的不同之处只是产生浪费的组件发生了变化，TodoList 渲染 Connect（TodoItem）是浪费，根源是 Connect（TodoItem）渲染 TodoItem 浪费，其中 Connect（TodoItem）就是利用 react-redux 的 connect 函数产生的容器组件。

要理解为什么会出现这个现象，就要理解 react-redux 提供的是 shouldCompnentUpdate 是怎样的实现方式。

相对于 React 组件的默认 shouldComponentUpdate 函数实现，react-redux 的实现方式当然是前进了一大步。但是在对比 prop 和上一次渲染所用 prop 方面，依然用的是尽量简单的方法，做的是所谓“浅层比较”（shallow compare）。简单说来就是用 JavaScript 的 ===

操作符来进行比较，如果 prop 的类型是字符串或者数字，只要值相同，那么“浅层比较”也会认为二者相同，但是，如果 prop 的类型是复杂对象，那么“浅层比较”的方式只看这两个 prop 是不是同一对象的引用，如果不是，哪怕这两个对象中的内容完全一样，也会被认为是两个不同的 prop。

比如，JSX 中使用组件 Foo 的时候给名为 style 的 prop 赋值，代码如下：

```
<Foo style={{color: "red"}} />
```

像上面这样使用方法，Foo 组件利用 react-redux 提供的 shouldComponentUpdate 函数实现，每一次渲染都会认为 style 这个 prop 发生了变化，因为每次都会产生一个新的对象给 style，而在“浅层比较”中，只比较第一层，不会去比较对象里面是不是相等。

看起来，react-redux 采用“浅层比较”似乎做得不够好，为什么不用“深层比较”呢？

但其实这是一个正确的决定，因为一个对象到底有多少层无法预料，如果递归对每个字段都进行“深层比较”，不光让代码更加复杂，也可能会造成性能问题。

在通用的 shouldCompnentUpdate 函数中做“浅层比较”，是一个被普遍接受的做法；如果需要做“深层比较”，那就是某个特定组件的行为，需要开发者自己根据组件情况去编写。不过也要谨记，不要简单地递归比较所有层次的字段，因为传递进来的 prop 对象什么结构是无法预料的。

总之，要想让 react-redux 认为前后的对象类型 prop 是相同的，就必须要保证 prop 是指向同一个 JavaScript 对象。

上面使用 Foo 组件的例子应该改进成下面这样。

```
const fooStyle = {color: "red"}; //确保这个初始化只执行一次，不要放在render中

<Foo style={fooStyle} />
```

同样的情况也存在于函数类型的 prop，react-redux 无从知道两个不同的函数是不是做着一样的事情，要想让它认为两个 prop 是相同的，就必须让这这两个 prop 指向同样一个函数，如果每次传给 prop 的都是一个新创建的函数，那肯定就没法让 prop 指向同一个函数了。

看 TodoList 传递给 TodoItem 的 onToggle 和 onRemove 这个 prop 是如何写的，在 JSX 中的代码如下：

```
onToggle={() => onToggleTodo(item.id)}

onRemove={() => onRemoveTodo(item.id)}
```

这里赋值给 onClick 的是一个匿名的函数，而且是在赋值的时候产生的。也就是说，每次渲染一个 TodoItem 的时候，都会产生一个新的函数，这就是问题所在。

虽然每次产生的匿名函数做的都是同样的事情，但是 react-redux 只认函数类型 prop 是不是指向同一个函数对象，每次都新产生的函数怎么可能通过这个检验呢，所以，即使 TodoItem 组件被 react-redux 库装备上了巧妙的 shouldCompoentUpdate 实现，依然躲不过每次更新过程都要重新渲染的命运。

怎么办呢？办法就是不要让 TodoList 每次都传递新的函数给 TodoItem。

在 Todo 应用这个例子中，TodoItem 组件实例的数量是不确定的，而每个 TodoItem 的点击事件函数又依赖于 TodoItem 的 id，所以处理起来有点麻烦这里有两种方式，下面以 onToggle 这个 prop 为例展示一下，onRemove 使用类似的方法。

第一种方式，TodoList 保证传递给 TodoItem 的 onToggle 永远只指向同一个函数对象，这样是为了应对 TodoItem 的 shouldComponentUpdate 检查，但是因为 TodoItem 可能有多个实例，所以这个函数要用某种方法区分什么 TodoItem 回调这个函数，区分的方法只能通过函数参数。

在 TodoList 组件中，mapDispatchToProps 产生的 prop 中的 onToggleTodo 接受 TodoItem 的 id 作为参数，恰好胜任这个工作，所以，可以在 JSX 中代码改成下面这样：

```
<TodoItem
  key={item.id}
  id={item.id}
  text={item.text}
  completed={item.completed}
  onToggle={onToggleTodo}
  onRemove={onRemoveTodo}
/>
```

注意，除了 onToggle 和 onRemove 的值改变了，还增加了一个新的 prop 名为 id，这是让每个 TodoItem 知道自己的 id，在回调 onToggle 和 onRemove 时可以区分不同的 TodoItem 实例。

TodoList 的代码简化了，但是 TodoItem 组件也要做对应改变，对应 TodoItem 组件的 mapDispatchToProps 函数代码如下：

```
const mapDispatchToProps = (dispatch, ownProps) => ({
  onToggleItem : () => ownProps.onToggle(ownProps.id)
});
```

以前我们只使用过 mapDispatchToProp 这个函数的第一个参数 dispatch，其实这个函数还有第二个参数 ownProps，也就是父组件渲染当前组件时传递过来的 props，通过访问 ownProps.id 就能够得到父组件传递过来的名为 id 的 prop 值。

上面的 mapDispatchToProps 函数给 TodoItem 组件增加了名为 onToggleItem 的

prop，调用 onToggle，传递当前实例的 id 作为参数，在 TodoItem 的 JSX 中就应该使用 onToggleItem，而不是直接使用 TodoList 提供的 onToggle。

第二种方式，干脆让 TodoList 不要给 TodoItem 传递任何函数类型 prop，点击事件完全由 TodoItem 组件自己搞定。

在 TodoList 组件的 JSX 中，渲染 TodoItem 组件的代码如下：

```
<TodoItem
  key={item.id}
  id={item.id}
  text={item.text}
  completed={item.completed}
/>
```

可以看到不需要 onToggle 和 onRemove 这些函数类型 prop，但依然有名为 id 的 prop。

在 TodoItem 组件中，需要自己通过 react-redux 派发 action，需要改变的代码如下：

```
const mapDispatchToProps = (dispatch, ownProps) => {
  const {id} = ownProps;
  return {
    onToggle: () => dispatch(toggleTodo(id)),
    onRemove: () => dispatch(removeTodo(id))
  }
};
```

对比两种方式，可以看到无论如何 TodoItem 都需要使用 react-redux，都需要定义产生定制 prop 的 mapDispatchToProps，都要求 TodoList 传入一个 id，区别只在于 actions 是由父组件导入还是由组件自己导入。

相比而言，没有多大必要让 action 在 TodoList 导入然后传递一个函数给 TodoItem，第二种方式让 TodoItem 处理自己的一切事务，更符合高内聚的要求。

在 https://github.com/mocheng/react-and-redux 的 chapter-05/todo_perf 目录下，可以看到用第二种方式实现的完整代码，在代码中有意把傻瓜组件 TodoItem 变成了一个 ES6 class，这么做完全是为了在这个组件的构造函数中写上 console.log，在 render 函数中也使用了 console.log 语句。这样，在演示改进的 Todo 应用时，根据浏览器 Console 上的输出，就能知道 TodoItem 组件是否经历了装载过程和更新过程。

重新启动改进后的 Todo 应用，添加三个待办事项，分别是 First、Second 和 Third，让 First 和 Third 反转为完成状态，清空浏览器的 Console。

这时候，我们点击“已完成”过滤器，待办事项就只显示 First 和 Third，在 Console 上看不到输出，说明这个过程中没有任何 TodoItem 组件被创建和更新。

然后我们再点击“全部”过滤器，这时在浏览器 Console 中可以有如下输出：

```
enter TodoItem constructor: Second
enter TodoItem render: Second
```

这个过程涉及 TodoList 组件的更新，但是 First 和 Third 两个 TodoItem 实例的 render 函数并没有被调用，说明 react-redux 的 shouldComponentUpdate 函数起了作用，避免了没有必要的重新渲染，使用 React Perf 工具，也不再出现浪费的渲染时间。

读者可能好奇，在“全部”和“未完成”之间切换的时候，似乎只有 Second 这个 TodoItem 经历了装载过程，React 如何知道 First 和 Third 这两个 TodoItem 不需要渲染的呢？这就是下一节多个 React 组件性能优化要讨论的问题。

5.2　多个 React 组件的性能优化

在第 2 章中介绍过 React 组件的生命周期，当一个 React 组件被装载、更新和卸载时，组件的一系列生命周期函数会被调用。不过，这些生命周期函数是针对某一个特定 React 组件的函数，在一个应用中，从上到下有很多 React 组件组合起来，它们之间的渲染过程要更加复杂。

还是拿 Todo 应用为例，从顶层 Provider 组件算起，首先有 TodoApp 组件，然后 TodoApp 组件包含 Todos 和 Filter 组件，Todos 组件包含 AddTodo 组件和 TodoList 组件，而 TodoList 组件包含动态数量的 TodoItem 组件，连 Filter 组件也包含若干个 Link 组件，如图 5-3 所示。

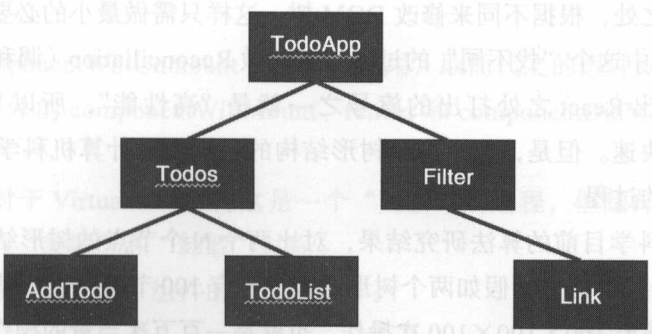


图 5-3　Todo 应用的组件结构

我们现在考虑的不是单个 React 组件内的渲染过程，而是多个 React 组件之间组合的渲染过程。和单个 React 组件的生命周期一样，React 组件也要考虑三个阶段：装载阶段、更新阶段和卸载阶段。

其中，装载阶段基本没有什么选择，当一个 React 组件第一次出现在 DOM 树中时，

无论如何是要彻底渲染一次的，从这个 React 组件往下的所有子组件，都要经历一遍 React 组件的装载生命周期，因为这部分的工作没有什么可以省略的，所以没有多少性能优化的事情可做。

至于卸载阶段，只有一个生命周期函数 componentWillUnmount，这个函数做的事情只是清理 componentDidMount 添加的事件处理监听等收尾工作，做的事情比装载过程要少很多，所以也没有什么可优化的空间。

所以值得关注的过程，就只剩下了更新过程。

5.2.1 React 的调和 (Reconciliation) 过程

在装载过程中，React 通过 render 方法在内存中产生了一个树形的结构，树上每一个节点代表一个 React 组件或者原生的 DOM 元素，这个树形结构就是所谓的 Virtual DOM。React 根据这个 Virtual DOM 来渲染产生浏览器中的 DOM 树。

在装载过程结束后，用户就可以对网页进行交互，用户操作引发了界面的更新，网页中需要更新界面，React 依然通过 render 方法获得一个新的树形结构 Virtual DOM，这时候当然不能完全和装载过程一样直接用 Virtual DOM 去产生 DOM 树，不然就和最原始的字符串模板一个做法。而且，在真实的应用中，大部分网页内容的更新都是局部的小改动，如果每个改动都是推倒重来，那样每次都重新完全生成 DOM 树，性能肯定不可接受。

实际上，React 在更新阶段很巧妙地对比原有的 Virtual DOM 和新生成的 Virtual DOM，找出两者的不同之处，根据不同来修改 DOM 树，这样只需做最小的必要改动。

React 在更新中这个“找不同”的过程，就叫做 Reconciliation（调和）。

Facebook 推出 React 之处打出的旗号之一就是“高性能”，所以 React 的 Reconciliation 过程必须快速。但是，找出两个树形结构的区别，从计算机科学的角度来说，真的不是一件快速的过程。

按照计算机科学目前的算法研究结果，对比两个 N 个节点的树形结构的算法，时间复杂度是 $O(N^3)$，打个比方，假如两个树形结构上各有 100 节点，那么找出这两个树形结构差别的操作，需要 100×100×100 次操作，也就是一百万次当量的操作，假如有一千个节点，那么需要相当于进行相当于 1000×1000×1000 次操作，这是一亿次的操作当量，这么巨大数量的操作在强调快速反应的网页中是不可想象的，所以 React 不可能采用这样的算法。

React 实际采用的算法需要的时间复杂度是 O(N)，因为对比两个树形怎么着都要对比两个树形上的节点，似乎也不可能有比 O(N) 时间复杂度更低的算法。

注意　O(N) 时间复杂度的算法并不是说一定会有 N 次指令操作，$O(N^3)$ 时间复杂度的算法也不是说这个算法一定会执行 N 的三次方数量的指令操作，时间复杂度只是对一个算法最好和最差情况下需要的指令操作数量级的估量，本书中不仔细介绍时间复杂度的定义，但是读者应该了解通常一个 $O(N^3)$ 时间复杂度的算法意味着不适合于高性能要求的场合。

React 采用的算法肯定不是最精准的，但是对于 React 应对的场景来说，绝对是性能和复杂度的最好折衷，让这个算法发挥作用，还需要开发者一点配合，让我们来看一看 React 的取属性结果差异算法。

其实 React 的 Reconciliation 算法并不复杂，当 React 要对比两个 Virtual DOM 的树形结构的时候，从根节点开始递归往下比对，在树形结构上，每个节点都可以看作一个这个节点以下部分子树的根节点。所以其实这个对比算法可以从 Virtual DOM 上任何一个节点开始执行。

React 首先检查两个树形的根节点的类型是否相同，根据相同或者不同有不同处理方式。

1. 节点类型不同的情况

如果树形结构根节点类型不相同，那就意味着改动太大了，也不要去费心考虑是不是原来那个树形的根节点被移动到其他地方去了，直接认为原来那个树形结构已经没用，可以扔掉，需要重新构建新的 DOM 树，原有的树形上的 React 组件会经历“卸载”的生命周期。

这时候，componentWillUnmount 方法会被调用，取而代之的组件则会经历装载过程的生命周期，组件的 componentWillMount、render 和 componentDidMount 方法依次被调用。

也就是说，对于 Virtual DOM 树这是一个“更新”的过程，但是却可能引发这个树结构上某些组件的“装载”和“卸载”过程。

举个例子，在更新之前，组件的结构是这样：

```
<div>
  <Todos />
</div>
```

我们想要更新成这样：

```
<span>
  <Todos />
</span>
```

那么在做比较的时候，一看根节点原来是 div，新的根节点是 span，类型就不一样，那么这个算法就认为必须要废掉之前的 div 节点，包括下面所有的子节点，一切推倒重来，重新构建一个 span 节点以及其子节点。

很明显，这是一个巨大的浪费，因为 div 和 span 的子节点其实是一模一样的 Todos 组件，顶层的元素实际上不做什么实质的功能，但是仅仅因为类型不同就把本可以重用的 Todos 组件卸载掉，然后重新再把这个组件装载一遍。

虽然是浪费，但是为了避免 $O(N^3)$ 的时间复杂度，React 必须要选择一个更简单更快捷的算法，也就只能采用这种方式。

作为开发者，很显然一定要避免上面这样浪费的情景出现。所以，一定要避免作为包裹功能的节点类型被随意改变，像上面的例子中，把 div 换成 span 只会带来没有必要的组件重新装载。

如果 React 对比两个树形结构的根节点发现类型相同，那么就觉得可以重用原有的节点，进入更新阶段，按照下一小节的步骤来处理。

2. 节点类型相同的情况

如果两个树形结构的根节点类型相同，React 就认为原来的根节点只需要更新过程，不会将其卸载，也不会引发根节点的重新装载。

这时，有必要区分一下节点的类型，节点的类型可以分为两类：一类是 DOM 元素类型，对应的就是 HTML 直接支持的元素类型，比如 div、span 和 p；另一类是 React 组件，也就是利用 React 库定制的类型。

对于 DOM 元素类型，React 会保留节点对应的 DOM 元素，只对树形结构根节点上的属性和内容做一下比对，然后只更新修改的部分。

比如原本的节点用 JSX 表示是这样：

```
<div style={{color: "red", fontSize: 15}} className="welcome">
  Hello World
</div>
```

改变之后的 JSX 表示是这样：

```
<div style={{color: "green", fontSize: 15}} className="farewell">
  Good Bye
</div>
```

可以看到，二者的差别是 div 中的文字发生了改变，另外 className 也发生了变化。style 中有一个字段 color 发生了改变。React 可以对比发现这些属性和内容的变化，在操作 DOM 树上节点的时候，只去修改这些发生变化的部分，让 DOM 操作尽可能少。

如果属性结构的根节点不是 DOM 元素类型，那就只可能是 React 组件类型，那么 React 做的工作类似，只是 React 此时并不知道如何去更新 DOM 树，因为这些逻辑还在 React 组件之中，React 能做的只是根据新节点的 props 去更新原来根节点的组件实例，引发这个组件实例的更新过程，也就是按照顺序引发下列函数：

- shouldComponentUpdate
- componentWillReceiveProps
- componentWillUpdate
- render
- componentDidUpdate

在这个过程中，如果 shouldComponentUpdate 函数返回 false 的话，那么更新过程就此打住，不再继续。所以为了保持最大的性能，每个 React 组件类必须要重视 shouldComponentUpdate，如果发现根本没有必要重新渲染，那就可以直接返回 false。

在处理完根节点的对比之后，React 的算法会对根节点的每个子节点重复一样的动作，这时候每个子节点就成为它所覆盖部分的根节点，处理方式和它的父节点完全一样。

3. 多个子组件的情况

当一个组件包含多个子组件的情况，React 的处理方式也非常简单直接。

拿 Todo 应用中的待办事项列表作为例子，假如最初的组件形态用 JSX 表示是这样：

```
<ul>
  <TodoItem text="First" completed={false}>
  <TodoItem text="Second" completed={false}>
</ul>
```

在更新之后，用 JSX 表示是这样：

```
<ul>
  <TodoItem text="First" completed={false}>
  <TodoItem text="Second" completed={false}>
  <TodoItem text="Third" completed={false}>
</ul>
```

那么 React 会发现多出了一个 TodoItem，会创建一个新的 TodoItem 组件实例，这个 TodoItem 组件实例需要经历装载过程，对于前两个 TodoItem 实例，React 会引发它们的更新过程，但是只要 TodoItem 的 shouldComponentUpdate 函数实现恰当，检查 props 之后就返回 false 的话，就可以避免实质的更新操作。

上面的例子是在 TodoItem 序列后面增加一个新的 TodoItem 实例，接下来我们看另外一个例子，在序列前面增加一个 TodoItem 实例，就会暴露出一个问题，JSX 代码如下：

```
<ul>
  <TodoItem text="Zero" completed={false}>
  <TodoItem text="First" completed={false}>
  <TodoItem text="Second" completed={false}>
</ul>
```

从直观上看，内容是“Zero”的新加待办事项被插在了第一位，只需要创造一个新的组件 TodoItem 实例放在第一位，剩下两个内容为“First”和“Second”的 TodoItem 实例经历更新过程，但是因为 props 没有改变，所以 shouldComponentUpdate 可以帮助这两个组件不做实质的更新动作。

可是实际情况并不是这样。如果要让 React 按照上面我们构想的方式来做，就必须要找出两个子组件序列的不同之处，现有的计算出两个序列差异的算法时间是 $O(N^2)$，虽然没有树形结构比较的 $O(N^3)$ 时间复杂度那么夸张，但是也不适合一个对性能要求很高的场景，所以 React 选择看起来很傻的一个办法，不是寻找两个序列的精确差别，而是直接挨个比较每个子组件。

在上面的新 TodoItem 实例插入在第一位的例子中，React 会首先认为把 text 为 First 的 TodoItem 组件实例的 text 改成了 Zero，text 为 Second 的 TodoItem 组件实例的 text 改成了 First，在最后面多出了一个 TodoItem 组件实例，text 内容为 Second。这样操作的后果就是，现存的两个 TodoItem 实例的 text 属性被改变了，强迫它们完成一个更新过程，创造出来的新的 TodoItem 实例用来显示 Second。

理想情况下只需要增加一个 TodoItem 组件，实际上引发了两个 TodoItem 实例的更新，而且，假设有 100 个 TodoItem 实例，那就会引发 100 个 TodoItem 实例的更新，这明显就是一个浪费。

看起来的确很傻，但电脑不是人类，一个简单的算法就只能用这种方式处理问题。

当然，React 并不是没有意识到这个问题，所以 React 提供了方法来克服这种浪费，不过需要开发人员在写代码的时候提供一点小小的帮助，这就是 key 的作用。

5.2.2 Key 的用法

React 不会使用一个 $O(N^2)$ 时间复杂度的算法去找出前后两列子组件的差别，默认情况下，在 React 的眼里，确定每一个组件在组件序列中的唯一标识就是它的位置，所以它也完全不懂哪些子组件实际上并没有改变，为了让 React 更加“聪明”，就需要开发者提供一点帮助。

如果在代码中明确地告诉 React 每个组件的唯一标识，就可以帮助 React 在处理这个问题时聪明很多，告诉 React 每个组件“身份证号”的途径就是 key 属性。

假如让待办事项列表用 JSX 表示的代码如下：

```
<ul>
  <TodoItem key={1} text="First" completed={false}>
  <TodoItem key={2} text="Second" completed={false}>
</ul>
```

前面代码的区别是每个 TodoItem 增加了名为 key 的 prop，而且每个 key 是这个 TodoItem 实例的唯一 id，当新的待办事项列表变成下面这样的时候，React 的处理方式也会不一样，现在第一位增加一个 TodoItem 实例，我们给他一个唯一的 key 值 0，代码如下：

```
<ul>
  <TodoItem key={0} text="Zero" completed={false}>
  <TodoItem key={1} text="First" completed={false}>
  <TodoItem key={2} text="Second" completed={false}>
</ul>
```

React 根据 key 值，可以知道现在的第二和第三个 TodoItem 实例其实就是之前的第一和第二个实例，所以 React 就会把新创建的 TodoItem 实例插在第一位，对于原有的两个 TodoItem 实例只用原有的 props 来启动更新过程，这样 shouldComponentUpdate 就会发生作用，避免无谓的更新操作。

庆幸的是，React 会提醒开发者不要忘记使用 key，同类型子组件出现多个实例时如果没有 key 的话，React 在运行时会给出警告，例如，在 Todo 应用中，我们把 TodoList 中关于 key 的那一行删掉，让每个 TodoItem 都没有 key 属性，JSX 代码如下：

```
<TodoItem
  id={item.id}
  text={item.text}
  completed={item.completed}
/>
```

在浏览器的 Console 中就可以看见如下所示的错误警告，提示开发者在 TodoList 中不要忘了使用 key：

```
▶ Warning: Each child in an array or iterator should have a unique "key" prop. Check the render method of `TodoList`. See https://fb.me/react-warning-keys for more information.    warning.js:36
    in Connect(TodoItem) (at todoList.js:11)
    in TodoList (created by Connect(TodoList))
    in Connect(TodoList) (at todos.js:11)
    in div (at todos.js:9)
    in Unknown (at TodoApp.js:8)
    in div (at TodoApp.js:7)
    in TodoApp (at index.js:10)
    in Provider (at index.js:9)
```

理解了 key 属性的作用，也就知道，在一列子组件中，每个子组件的 key 值必须唯一，不然就没有帮助 React 区分各个组件的身份，这并不是一个很难的问题，一般很容易

给每个组件找到一个唯一的 id。

但是这个 key 值只是唯一还不足够，这个 key 值还需要是稳定不变的，试想，如果 key 值虽然能够在每个时刻都唯一，但是变来变去，那么就会误导 React 做出错误判断，甚至导致错误的渲染结果。

如果通过数组来产生一组子组件，一个常见的错误就是将元素在数组中的下标值作为 key，下面的代码是错误的例子：

```
<ul>
  {
    todos.map((item, index) => (
      <TodoItem
        key={index}
        text={item.text}
        completed={item.completed}
      />
    ))
  }
</ul>
```

这么做非常危险，因为，假如没有使用 key 的话 React 会在运行时输出一个错误提示，但是错误地使用 key 值 React 就不会给出错误提示了，因为 React 无法发现开发者的错误。

用数组下标作为 key，看起来 key 值是唯一的，但是却不是稳定不变的，随着 todos 数组值的不同，同样一个 TodoItem 实例在不同的更新过程中在数组中的下标完全可能不同，把下标当做 key 就让 React 彻底乱套了。

需要注意，虽然 key 是一个 prop，但是接受 key 的组件并不能读取到 key 的值，因为 key 和 ref 是 React 保留的两个特殊 prop，并没有预期让组件直接访问。

5.3 用 reselect 提高数据获取性能

在前面的例子中，都是通过优化渲染过程来提高性能，既然 React 和 Redux 都是通过数据驱动渲染过程，那么除了优化渲染过程，获取数据的过程也是一个需要考虑的优化点。

在 src/todos/views/todoList.js 文件中，通过 mapStateToProps 调用 selectVisibleTodos 函数从 Redux Store 提供的 state 中产生渲染需要的数据，代码如下：

```
const selectVisibleTodos = (todos, filter) => {
  switch (filter) {
    case FilterTypes.ALL:
```

```
        return todos;
      case FilterTypes.COMPLETED:
        return todos.filter(item => item.completed);
      case FilterTypes.UNCOMPPLETED:
        return todos.filter(item => !item.completed);
      default:
        throw new Error('unsupported filter');
    }
  }

  const mapStateToProps = (state) => {
    return {
      todos: selectVisibleTodos(state.todos, state.filter)
    };
  }
```

作为从 Redux Store 上获取数据的重要一环，mapStateToProps 函数一定要快，从代码看来，运算本身并没有什么可优化空间，要获取当前应该显示的待办事项，就是要根据 Redux Store 状态树上的 todos 和 filter 两个字段上的值计算出来。不过这个计算过程要遍历 todos 字段上的数组，当数组比较大的时候，对于 TodoList 组件的每一次重新渲染都重新计算一遍，就会显得负担过重了。

5.3.1　两阶段选择过程

既然这个 selectVisibleTodos 函数的计算必不可少，那如何优化呢？

实际上，并不是每一次对 TodoList 组件的重新渲染都必须要执行 selectVisibleTodos 中的计算过程，如果 Redux Store 状态树上代表所有待办事项的 todos 字段没有变化，而且代表当前过滤器的 filter 字段也没有变化，那么实在没有必要重新遍历整个 todos 数组来计算一个新的结果，如果上一次的计算结果被缓存起来的话，那就可以重用缓存的数据。

这就是 reselect 库的工作原理：只要相关状态没有改变，那就直接使用上一次的缓存结果。

reselect 库被用来创造“选择器”，所谓选择器，就是接受一个 state 作为参数的函数，这个选择器函数返回的数据就是我们某个 mapStateToProps 需要的结果。

在前面的章节，我们已经强调过 React 组件的渲染函数应该是一个纯函数，Redux 中的 reducer 函数也应该是一个纯函数，mapStateToProps 函数也应该是纯函数，纯函数让问题清晰而且简化。不过，现在这个“选择器”函数可不是纯函数，它是一种有“记忆力”的函数，运行选择器函数会有副作用，副作用就是能够根据以往的运行“记忆”返

回“记忆”中的结果。

reselect 认为一个选择器的工作可以分为两个部分，把一个计算过程分为两个步骤：

步骤 1，从输入参数 state 抽取第一层结果，将这第一层结果和之前抽取的第一层结果做比较，如果发现完全相同，就没有必要进行第二部分运算了，选择器直接把之前第二部分的运算结果返回就可以了。注意，这一部分做的“比较”，就是 JavaScript 的 === 操作符比较，如果第一层结果是对象的话，只有是同一对象才会被认为是相同。

步骤 2，根据第一层结果计算出选择器需要返回的最终结果。

显然，每次选择器函数被调用时，步骤一都会被执行，但步骤一的结果被用来判断是否可以使用缓存的结果，所以并不是每次都会调用步骤二的运算。

选择器就是利用这种缓存结果的方式，避免了没有必要的运算浪费。

剩下的事情就是确定选择器步骤一和步骤二分别进行什么运算。原则很简单，步骤一运算因为每次选择器都要使用，所以一定要快，运算要非常简单，最好就是一个映射运算，通常就只是从 state 参数中得到某个字段的引用就足够，把剩下来的重活累活都交给步骤二去做。

在 TodoList 这个具体例子中，todos 和 filter 的值直接决定应该显示什么样的待办事项，所以，很显然步骤一是获取 todos 和 filter 的值，步骤二就是根据这两个值进行计算。

使用 reselect 需要安装对应的 npm 包：

```
npm install --save reselect
```

在 src/todos/selector.js 文件中，选择器函数的代码如下：

```
import {createSelector} from 'reselect';
import {FilterTypes} from '../constants.js';

export const selectVisibleTodos = createSelector(
  [getFilter, getTodos],
  (filter, todos) => {
    switch (filter) {
      case FilterTypes.ALL:
        return todos;
      case FilterTypes.COMPLETED:
        return todos.filter(item => item.completed);
      case FilterTypes.UNCOMPPLETED:
        return todos.filter(item => !item.completed);
      default:
        throw new Error('unsupported filter');
    }
  }
);
```

reselect 提供了创造选择器的 createSelector 函数，这是一个高阶函数，也就是接受函数为参数来产生一个新函数的函数。

第一个参数是一个函数数组，每个元素代表了选择器步骤一需要做的映射计算，这里我们提供了两个函数 getFilte 和 getTodos，对应代码如下：

```
const getFilter = (state) => state.filter;
const getTodos = (state) => state.todos;
```

上面说过，步骤一的运算要尽量简单快捷，所以往往一个 Lambda 表达式就足够。

createSelector 函数的第二个参数代表步骤二的计算过程，参数为第一个参数的输出结果，里面的逻辑和之前 TodoList 中的逻辑没有什么两样，只是这第二个函数不是每次都会被调用到。

现在，我们可以在 TodoList 模块中改用新定义的选择器来获取待办事项数据了，代码如下：

```
import {selectVisibleTodos} from '../selector.js';

const mapStateToProps = (state) => {
  return {
    todos: selectVisibleTodos(state)
  };
}
```

Redux 要求每个 reducer 不能修改 state 状态，如果要返回一个新的状态，就必须返回一个新的对象。如此一来，Redux Store 状态树上某个节点如果没有改变，那么我们就有信心这个节点下数据没有改变，应用在 reselect 中，步骤一的运算就可以确定直接缓存运算结果。

虽然 reselect 的 createSelector 创造的选择器并不是一个纯函数，但是 createSelector 接受的所有函数参数都是纯函数，虽然选择器有“记忆”这个副作用，但是只要输入参数 state 没有变化，产生的结果也就没有变化，表现得却类似于纯函数。

只要 Redux Store 状态树上的 filter 和 todos 字段不变，无论怎样触发 TodoList 的渲染过程，都不会引发没有必要的遍历 todos 字段的运算，性能自然更快。

虽然 reselect 库以 re 开头，但是逻辑上和 React/Redux 没有直接关系。实际上，在任何需要这种具有记忆的计算场合都可以使用 reselect，不过，对于 React 和 Redux 组合的应用，reselect 无疑能够提供绝佳的支持。

5.3.2 范式化状态树

在第 4 章中已经介绍过，Redux Store 的状态树应该设计的尽量扁平，在使用了 reselect

之后，我们可以进一步认为，状态树的设计应该尽量范式化（Normalized）。

所谓范式化，就是遵照关系型数据库的设计原则，减少冗余数据。如果读者做过关系型数据库（比如 MySQL 或者 PostgreSQL）的表设计，就知道范式化的数据结构设计就是要让一份数据只存储一份，数据冗余造成的后果就是难以保证数据一致性。

与范式化相对，还存在“反范式化”的数据库设计，这在风生水起的 NoSQL 领域是一个惯常的设计风格。反范式化是利用数据冗余来换取读写效率，因为关系型数据库的强项虽然是保持一致，但是应用需要的数据形式往往是多个表 join 之后的结果，而 join 的过程往往耗时而且在分布式系统中难以应用。

介绍范式化或者反范式化的数据库设计不是本书讨论范围内，这里只是用数据库的两种设计理念来类比 Redux Store 状态树的设计策略。

假设我们给 Todo 应用做一个更大的改进，增加一个 Type 的概念，可以把某个 TodoItem 归为某一个 Type，而且一个 Type 有特有的名称和颜色信息，在界面上，用户可以看到 TodoItem 显示为自己所属 Type 对应的颜色，TodoItem 和 Type 当然是多对多的关系。

用哪种方式设计状态树合适呢？更具体一点，如何设计代表 TodoItem 的状态树结构？

如果使用反范式化的设计，那么状态树上的数据最好是能够不用计算拿来就能用，在 Redux Store 状态树的 todos 字段保存的是所有待办事项数据的数组，对于每个数组元素，反范式化的设计会是类似下面的对象：

```
{
  id: 1, //待办事项id
  text: "待办事项1",          //待办事项文字内容
  completed: false,          //是否已完成
  type: { // 种类
    name: "紧急",            //种类的名称
    color: "red"             //种类的显示颜色
  }
}
```

在原有 TodoItem 对应状态中增加了 type 字段，包含 name 和 color 两个数据。当然，如果要达到数据扁平化的目的，应该是增加两个字段，分别为 typeName 和 typeColor。

这种状态设计的好处就是在渲染 TodoItem 组件时，从 Redux Store 上获得的状态可以直接使用 name 和 color 数据。但这也有缺点，当需要改变某种类型的名称和颜色时，不得不遍历所有 TodoItem 数据来完成改变。反范式化数据结构的特点就是读取容易，修改比较麻烦。

如果使用范式化的数据结构设计，那么 Redux Store 上代表 TodoItem 的一条数据是类似下面的对象：

```
{
  id: 1,                    //待办事项id
  text: "待办事项1",          //待办事项文字内容
  completed: false,         //是否已完成
  typeId: 1                 //待办事项所属的种类id
}
```

用一个 typeId 代表类型，然后在 Redux Store 上和 todos 平级的根节点位置创建一个 types 字段，内容是一个数组，每个数组元素代表一个类型，一个种类的数据是类似下面的对象：

```
{
  id: 1,                    //种类id
  name: "紧急",              //种类的名称
  color: "red"              //种类的显示颜色
}
```

当 TodoItem 组件要渲染内容时，从 Redux Store 状态树的 todos 字段下获取的数据是不够的，因为只有 typeId。为了获得对应的种类名称和颜色，需要做一个类似关系型数据库的 join 操作，到状态树的 types 字段下去寻找对应 typeId 的种类数据。

这个过程当然要花费一些时间，但是当要改变某个种类的名称或者颜色时，就异常地简单，只需要修改 types 中的一处数据就可以了。

对比反范式方式和范式方式的优劣，不难看出范式方式更合理。因为虽然 join 数据需要花费计算时间，但是应用了 reselect 之后，大部分情况下都会命中缓存，实际上也就没有花费很多计算时间了。

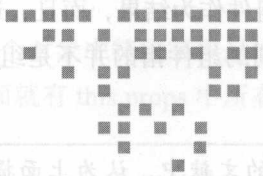
5.4　本章小结

在本章中，我们了解了利用 react-redux 提供的 shouldComponentUpdate 实现来提高组件渲染功能的方法，一个要诀就是避免传递给其他组件的 prop 值是一个不同的对象，不然会造成无谓的重复渲染。

通过了解 React 的 Reconciliation 过程，我们了解了 React 为什么能够在 O(N) 时间复杂度下完成 Virutal DOM 的比较，但是为了让 React 高效无误地完成工作，需要开发者做一些配合。首先，不能随意修改一个作为容器的 HTML 节点的类型。其次，对于动态数量的同类型子组件，一定要使用 key 这个 prop。

React 的 Reconciliation 算法缺点是无法发现某个子树移动位置的情况，如果某个子树移动了位置，那 React 就会重新创建这个子树。当然，通常应用中不会出现这种情况。

最后，我们学习了利用 reselect 库来实现高效的数据获取。因为 reselect 的缓存功能，开发者不用顾忌范式化的状态树会存在性能问题，Redux Store 的状态树应该按照范式化原则来设计，减少数据冗余，这样利于保持数据一致。



第 6 章 Chapter 6

React 高级组件

我们已经掌握构建 React 组件的方法，当我们开发出更多 React 组件时，就会遇到这样的问题，多个组件都需要某个功能，而且这个功能和界面并没有关系，所以也不能简单地抽取成一个新的组件，但是如果让同样的逻辑在各个组件里各自实现，无疑会导致重复代码。

“重复是优秀系统设计的大敌。”——Robert C.Martin

在这一章中，我们就会探讨如何构建更易于复用、更灵活的 React 高级组件，包含下面两种方式：

- 高阶组件的概念及应用；
- 以函数为子组件的模式。

这两种方式的最终目的都是为了重用代码，只是策略不同，各有优劣，开发者在实际工作中要根据实际情况决定采用何种方式。

6.1 高阶组件

高阶组件（Higher Order Component，HOC）并不是 React 提供的某种 API，而是使用 React 的一种模式，用于增强现有组件的功能。

简单来说，一个高阶组件就是一个函数，这个函数接受一个组件作为输入，然后返

回一个新的组件作为结果，而且，返回的新组件拥有了输入组件所不具有的功能。

这里提到的组件指的并不是组件实例，而是一个组件类，也可以是一个无状态组件的函数。

提示 在有的文献中，认为上面提到的函数返回的结果才应该叫“高阶组件”，而这个函数本身应该叫做“高阶组件工厂函数”，这样的定义更加严谨。但是在本书中，我们遵循更普遍的定义，把增强功能的函数叫做“高阶组件”，读者只需要知道存在这两种说法就可以。

我们先看一个非常简单的高阶组件的例子，感受一下高阶组件是如何工作的，代码如下：

```
import React from 'react';

function removeUserProp(WrappedComponent) {
  return class WrappingComponent extends React.Component {
    render() {
      const {user, ...otherProps} = this.props;
      return <WrappedComponent {...otherProps} />
    }
  }
}

export default removeUserProp;
```

在上面的代码中定义了一个函数 removeUserProp，这个函数就是我们定义的第一个高阶组件。

这样一个高阶组件做的工作非常简单，它有一个参数，名叫 WrappedComponent，代表一个组件类，这个函数返回一个新的组件，所做的事情和 WrappedComponent 一模一样，只是忽略名为 user 的 prop。也就是说，如果 WrappedComponent 能够处理名为 user 的 prop，这个高阶组件返回的组件则完全无视这个 prop。

函数 removeUserProp 直接返回了一个 class 的定义，这个 class 名叫 WrappingComponent。其实这个 class 叫什么名字并不重要，因为它只是一个局部变量，这个 WrappingComponent 继承自 React.Component，表明返回的结果依然是一个组件类，像上面说过的，高阶组件就是根据一个组件类产生一个新的组件类。

WrappingComponent 作为一个 React 组件，必须要有自己的 render 函数，在它的 render 函数中，利用扩展操作符将 this.props 分为两部分：user，以及不是 user 的所有其他 props，即下面的代码这样：

```
const {user, ...otherProps} = this.props;
```

这是一个利用 ES6 语法的技巧，可以简洁地把一个对象中特定字段过滤掉，结果赋值给一个新的对象。经过上面的语句，otherProps 里面就有 this.props 中所有的字段，除了 user。

然后，render 函数把渲染的工作完全交给了 removeUserProp 函数的参数，只是使用了过滤掉了 user 的 otherProps，把这个 WrappedComponent 组件渲染的结果作为 render 函数返回的结果。

假如我们现在不希望某个组件接收到 user 的 prop，那么我们就不要直接使用这个组件，而是把这个组件作为参数传递给 removeUserProp 函数，然后我们把这个函数的返回结果当做组件来使用：

```
const NewComponent = removeUserProp(SampleComponent);
```

在上面的代码中，NewComponent 拥有和 SampleComponent 完全一样的行为，唯一的区别就是即使传递 user 属性给它，它也会当没有 user 来处理。当然，这看起来是削弱 SampleComponent 的功能，而不是“增强”其功能，不过，只要把“屏蔽某个 prop”也看做一种功能增强，那这个高阶组件已经完成了“增强”的工作。

定义高阶组件的意义何在呢？

首先，重用代码。有时候很多 React 组件都需要公用同样一个逻辑，比如说 react-redux 中容器组件的部分，没有必要让每个组件都实现一遍 shouldComponentUpdate 这些生命周期函数，把这部分逻辑提取出来，利用高阶组件的方式应用出去，就可以减少很多组件的重复代码。

其次，修改现有 React 组件的行为。有些现成 React 组件并不是开发者自己开发的，来自于第三方，或者，即使是我们自己开发的，但是我们不想去触碰这些组件的内部逻辑，这时候高阶组件有了用武之地。通过一个独立于原有组件的函数，可以产生新的组件，对原有组件没有任何侵害。

现在我们对高阶组件有了一个直观认识，但是上面的例子只是冰山一角，高阶组件可以有很多种实现方式，也可以应用于很多场景之下。

根据返回的新组件和传入组件参数的关系，高阶组件的实现方式可以分为两大类：

- 代理方式的高阶组件；
- 继承方式的高阶组件。

接下来我们就来介绍这两大类实现，然后介绍高阶组件的显示名，最后讲一段历史故事。

6.1.1 代理方式的高阶组件

上面的 removeUserProp 例子就是一个代理方式的高阶组件，特点是返回的新组件类直接继承自 React.Component 类。新组件扮演的角色是传入参数组件的一个“代理”，在新组建的 render 函数中，把被包裹组件渲染出来，除了高阶组件自己要做的工作，其余功能全都转手给了被包裹的组件。

如果高阶组件要做的功能不涉及除了 render 之外的生命周期函数，也不需要维护自己的状态，那也可以干脆返回一个纯函数，像上面的 removeUserProp，代码可以简写成下面这样：

```
function removeUserProp(WrappedComponent) {
  return function newRender(props) {
    const {user, ...otherProps} = props;
    return <WrappedComponent {...otherProps} />
  }
}
```

虽然这样写代码更少，但是为了代码逻辑更加清晰，在本章其他的例子中，我们还是统一让高阶组件返回一个 class，而不只是一个函数。

代理方式的高阶组件，可以应用在下列场景中：

- 操纵 prop；
- 访问 ref；
- 抽取状态；
- 包装组件。

下面分别介绍各种场景。

1. 操纵 prop

代理类型高阶组件返回的新组件，渲染过程也被新组件的 render 函数控制，而 render 函数相当于一个代理，完全决定如何使用被包裹的组件。

在 render 函数中，this.props 包含新组件接收到的所有 prop，最简单的方式当然是把 this.props 原封不动地传递给被包裹组件，当然，高阶组件也可以增减、删除、修改传递给包裹组件的 props 列表。

上面的 removeUserProp 就是一个删除了特定 prop 的高阶组件，我们再看一个增加 prop 的例子：

```
const addNewProps = (WrappedComponent, newProps) => {
  return class WrappingComponent extends React.Component {
    render() {
```

```
        return <WrappedComponent {...this.props} {...newProps} />
      }
    }
  }
```

和 removeUserProp 相反，这个高阶组件 addNewProps 增加了传递给被包裹组件的 prop，而不是删除 prop，我们让 addNewProps 不只接受被包裹参数，还增加了一个 newProps 参数，这样高阶组件的复用性更强，利用这样一个高阶组件可以给不同的组件扩充不同的新属性，代码如下：

```
const FooComponent = addNewPropsHOC(DemoComponent, {foo: 'foo'});
const BarComponent = addNewPropsHOC(OtherComponent, {bar: 'bar'});
```

在上面的代码中，新创造的 FooComponent 被添加了名为 foo 的 prop，BarComponent 被添加了名为 bar 的 prop。除此之外，两者的功能也不一样，因为一个是通过 DemoComponent 产生，另一个是根据 OtherComponent 产生。由此可见，一个高阶组件可以在重用在不同的组件上，减少代码的重复。

2. 访问 ref

在第 4 章介绍 ref 时我们就已经强调，访问 ref 并不是值得推荐的 React 组件使用方式，在这里我们只是展示应用高阶组件来实现这种方式的可行性。

我们写一个名为 refsHOC 的高阶组件，能够获得被包裹组件的直接应用 ref，然后就可以根据 ref 直接操纵被包裹组件的实例，代码如下：

```
const refsHOC = (WrappedComponent) => {
  return class HOCComponent extends React.Component {
    constructor() {
      super(...arguments);
      this.linkRef = this.linkRef.bind(this);
    }

    linkRef(wrappedInstance) {
      this._root = wrappedInstance;
    }

    render() {
      const props = {...this.props, ref: this.linkRef};
      return <WrappedComponent {...props}/>;
    }
  };
};
```

这个 refsHOC 的工作原理其实也是增加传递给被包裹组件的 props，只是利用了 ref 这个特殊的 prop，ref 这个 prop 可以是一个函数，在被包裹组件的装载过程完成的时候

被调用，参数就是被装载的组件本身。

传递给被包裹组件的 ref 值是一个成员函数 linkRef，当 linkRef 被调用时就得到了被包裹组件的 DOM 实例，记录在 this._root 中。

这样的高阶组件有什么作用呢？可以说非常有用，也可以说没什么用。说它非常有用，是因为只要获得了对被包裹组件的 ref 引用，那它基本上就无所不能，因为通过这个引用可以任意操纵一个组件的 DOM 元素。说它没什么用，是因为 ref 的使用非常容易出问题，我们已经知道最好能用“控制中的组件”（Controlled Component）来代替 ref。

软件开发中一个问题可以有很多解法，可行的解法很多，但是合适的解法不多，了解一种可能性并不表示一定要使用它，我们只需要知道高阶组件有访问 ref 这种可能，并不意味着我们必须要使用这种高阶组件。

3. 抽取状态

其实，我们已经使用过“抽取状态”的高阶组件了，就是 react-redux 的 connect 函数，注意 connect 函数本身并不是高阶组件，connect 函数执行的结果是另一个函数，这个函数才是高阶组件。

在傻瓜组件和容器组件的关系中，通常让傻瓜组件不要管理自己的状态，只要做一个无状态的组件就好，所有状态的管理都交给外面的容器组件，这个模式就是“抽取状态”。

我们尝试实现一个简易版的 connect 高阶组件，代码结构如下：

```
const doNothing = () => ({});

function connect(mapStateToProps=doNothing, mapDispatchToProps=doNothing) {
  return function(WrappedComponent) {
    class HOCComponent extends React.Component {
      //在这里定义HOCComponent的生命周期函数
    };

    HOCComponent.contextTypes = {
      store: React.PropTypes.object
    }
    return HOCComponent;
  };
}
```

创造的新组件 HOCComponent 需要利用 React 的 Context 功能，所以定义了 contextTypes 属性，从 Context 中获取名为 store 的值。

和 react-redux 中的 connect 方法一样，我们定义的 connect 方法接受两个参数，分别是 mapStateToProps 和 mapDispatchToProps。返回的 React 组件类预期能够访问一个叫 store 的 context 值，在 react-redux 中，这个 context 由 Provider 提供，在组件中我们通过

this.context.store 就可以访问到 Provider 提供的 store 实例。

我们在前面的章节中也实现了一个 Provider 的功能，在这里就不再赘述。

为了实现类似 react-redux 的功能，HOCComponent 组件需要一系列的成员函数来维持内部状态和 store 的同步，代码如下：

```
constructor() {
  super(...arguments);
  this.onChange = this.onChange.bind(this);
  this.store = {};
}

componentDidMount() {
  this.context.store.subscribe(this.onChange);
}

componentWillUnmount() {
  this.context.store.unsubscribe(this.onChange);
}

onChange() {
  this.setState({});
}
```

在上面的代码中，借助 store 的 subscribe 和 unsubscribe 函数，HOCComponent 保证每当 Redux 的 Store 上状态发生变化的时候，都会驱动这个组件的更新。

虽然应该返回一个有状态的组件，但反正真正的状态存在 Redux Store 上，组件内的状态存什么真的不重要，我们使用组件状态的唯一原因是可以通过 this.setState 函数来驱动组件的更新过程，所以这个组件的状态实际上是一个空对象就足够。

每当 Redux Store 上的状态发生变化时，就通过 this.setState 函数重设一遍组件的状态，每次都创造一个新的对象，虽然 state 没有任何有意义的值，却能够驱动组件的更新过程。

再来看 HOCComponent 的 render 函数，代码如下：

```
render() {
  const store = this.context.store;
  const newProps = {
    ...this.props,
    ...mapStateToProps(store.getState()),
    ...mapDispatchToProps(store.dispatch)
  }

  return <WrappedComponent {...newProps} />;
}
```

render 中的逻辑类似“操纵 Props”的方式，虽然渲染工作完全交给了 Wrapped-

Component，但是却控制住了传给 WrappedComponent 的 props，因为 WrappedComponent 预期是一个无状态的组件，所以能够渲染什么完全由 props 决定。

传递给 connect 函数的 mapStateToProps 和 mapDispatchToProps 两个参数在 render 函数中被使用，根据执行 store.getState 函数来得到 Redux Store 的状态，通过 store.dispatch 可以得到传递给 mapDispatchToProps 的 dispatch 方法。

使用扩展操作符，this.props 和 mapStateToProps、mapDispatchToProps 的返回结果被结合成一个对象，传递给了 WrappedCompoent，就是最终的渲染结果。

注意，这里我们实现的并不是完整的 connect 实现逻辑。比如，没有实现 shouldComponentUpdate 函数，缺少 shouldComponentUpdate 会导致每次 Store 状态变化都走一遍完整的更新过程，读者可以尝试着实现这个高阶组件中的 shouldComponentUpdate 函数。

4. 包装组件

到目前为止，通过高阶组件产生的新组件，render 函数都是直接返回被包裹组件，修改的只是 props 部分。其实 render 函数的 JSX 中完全可以引入其他的元素，甚至可以组合多个 React 组件，这样就会得到更加丰富多彩的行为。

一个实用的例子是给组件添加样式 style，代码如下：

```
const styleHOC = (WrappedComponent, style) => {

  return class HOCComponent extends React.Component {

    render() {
      return (
        <div style={style}>
          <WrappedComponent {...this.props}/>
        </div>
      );
    }
  };
};
```

把一个组件用 div 包起来，并且添加一个 style 来定制其 CSS 属性，可以直接影响被包裹的组件对应 DOM 元素的展示样式。

有了这个 styleHOC，就可以给任何一个组件补充 style 的样式，代码如下：

```
const style= {color: 'red'};
const NewComponent = styleHOC(DemoComponent, style);
```

6.1.2 继承方式的高阶组件

继承方式的高阶组件采用继承关系关联作为参数的组件和返回的组件，假如传入的

组件参数是 WrappedComponent，那么返回的组件就直接继承自 WrappedComponent。

我们用继承方式重新实现一遍 removeUserProp 这个高阶组件，代码如下：

```
function removeUserProp(WrappedComponent) {
  return class NewComponent extends WrappedComponent {
    render() {
      const {user, ...otherProps} = this.props;
      this.props = otherProps;
      return super.render();
    }
  };
}
```

代理方式和继承方式最大的区别，是使用被包裹组件的方式。

在代理方式下，render 函数中的使用被包裹组件是通过 JSX 代码：

```
return <WrappedComponent {...otherProps} />
```

在继承方式下，render 函数中渲染被包裹组件的代码如下：

```
return super.render();
```

因为我们创造的新的组件继承自传入的 WrappedComponent，所以直接调用 super.render 就能够得到渲染出来的元素。

需要注意，在代理方式下 WrappedComponent 经历了一个完整的生命周期，但在继承方式下 super.render 只是一个生命周期中的一个函数而已；在代理方式下产生的新组件和参数组件是两个不同的组件，一次渲染，两个组件都要经历各自的生命周期，在继承方式下两者合二为一，只有一个生命周期。

在上面的例子中我们直接修改了 this.props，这实在不是一个好做法，实际上，这样操作可能产生不可预料的结果。

继承方式的高阶组件可以应用于下列场景：

- 操纵 prop；
- 操纵生命周期函数。

1. 操纵 Props

继承方式的高阶组件也可以操纵 props，除了上面不安全的直接修改 this.props 方法，还可以利用 React.cloneElement 让组件重新绘制：

```
const modifyPropsHOC = (WrappedComponent) => {
  return class NewComponent extends WrappedComponent {
    render() {
      const elements = super.render();
      const newStyle = {
```

```
        color: (elements && elements.type === 'div') ? 'red' : 'green'
      }
      const newProps = {...this.props, style: newStyle};
      return React.cloneElement(elements, newProps,lements.props.children);
    }
  };
};
```

上面的高阶组件实现的功能是首先检查参数组件的 render 函数返回结果，如果顶层元素是一个 div，就将其增加一个 color 为 red 的 style 属性，否则增加 color 为 green 的 style 属性。最后，我们用 React.cloneElement 来传入新的 props，让这些产生的组件重新渲染一遍。

虽然这样可行，但是过程实在非常复杂，唯一用得上的场景就是高阶组件需要根据参数组件 WrappedComponent 渲染结果来决定如何修改 props。否则，实在没有必要用继承方式来实现这样的高阶组件，使用代理方式实现操纵 prop 的功能更加清晰。

2. 操纵生命周期函数

因为继承方式的高阶函数返回的新组件继承了参数组件，所以可以重新定义任何一个 React 组件的生命周期函数。

这是继承方式高阶函数特用的场景，代理方式无法修改传入组件的生命周期函数，所以不具备这个功能。

例如，我们可以定义一个高阶组件，让参数组件只有在用户登录时才显示，代码如下：

```
const onlyForLoggedinHOC = (WrappedComponent) => {
  return class NewComponent extends WrappedComponent {
    render() {
      if (this.props.loggedIn) {
        return super.render();
      } else {
        return null;
      }
    }
  }
}
```

又例如，我们可以重新定义 shouldComponentUpdate 函数，只要 prop 中的 useCache 不为逻辑 false 就不做重新渲染的动作，代码如下：

```
const cacheHOC = (WrappedComponent) => {
  return class NewComponent extends WrappedComponent {
    shouldComponentUpdate(nextProps, nextState) {
      return !nextProps.useCache;
```

```
    }
  }
}
```

从上面例子的比较可以看出来，各方面看来代理方式都要优于继承方式。

业界有一句老话："优先考虑组合，然后才考虑继承。"（Composition over Inheritance）。前人诚不欺我，我们应该尽量使用代理方式来构建高阶组件。

6.1.3　高阶组件的显示名

每个高阶组件都会产生一个新的组件，使用这个新组件就丢失掉了参数组件的"显示名"，为了方便开发和维护，往往需要给高阶组件重新定义一个"显示名"，不然，在 debug 和日志中看到的组件名就会莫名其妙。增加"显示名"的方式就是给高阶组件类的 displayName 赋上一个字符串类型的值。

以 react-redux 的 connect 为例，我们希望高阶组件的名字包含 Connect，同时要包含参数组件 WrappedComponent 的名字，所以我们对 connect 高阶组件做如下修改：

```
function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName ||
    WrappedComponent.name ||
    'Component';
  }
HOCComponent.displayName=`Connect(${getDisplayName(WrappedComponent)})`;
```

上面给 displayName 赋值使用了 ES6 的模板字符串语法，注意是用反撇号 ` 代替普通字符串的单引号或者双引号。

增加 displayName 的定义之后，在 React Perf 等工具中看到的组件类名就更有意义。例如，当我们用 connect 返回的结果去包裹一个叫 DemoComponent 的组件时，得到的新组建的 displayName 就是字符串 Connect（DemoComponent）。

6.1.4　曾经的 React Mixin

除了上面提到的高阶组件，React 还有一个可以提供组件之间复用代码的功能叫 Mixin，不过这已经是一个不建议使用的功能，在这里讲一段历史故事为的是让大家理解从灵活到不灵活的原因。

我们可以定义这样一个包含 shouldComponentUpdate 函数的 Mixin，代码如下：

```
const ShouldUpdateMixin = {
  shouldComponentUpdate: function() {
    return !this.props.useCache;
  }
}
```

但是 Mixin 只能在用 React.createClass 方式创建的组件类中才能使用，不能在通过 ES6 语法创建的 React 组件中使用。

下面是一个使用 Mixin 的代码样例：

```
const SampleComponent = React.createClass({
  mixins: [ShouldUpdateMixin],

  render: function() {
    // 实现render函数
  }
});
```

使用 React.createClass 创建出来的 React 组件 SampleComponent，因为有 mixins 字段的存在，成员方法中就“混入”了 ShouldUpdateMixin 这个对象里的方法。

第一眼看过去，Mixins 似乎使用非常灵活。但是，Mixin 因为太灵活，导致难以管理，而且作为一项设计原则，应该尽量把 state 从组件里抽取出来，尽量构建无状态组件，Mixins 却反其道而行之，鼓励往 React 组件中加入状态，这也是一个很大的缺陷。

对于 Mixins 的批评不胜枚举，不用过多描述。我们只需要知道，在 ES6 的 React 组件类定义方法中不能使用 Mixin，React 官方也很明确声明 Mixin 是应该被废弃的方法。所以我们只需要知道在 React 的历史上，曾经有这样一个重用代码的解决方法就足够了。

6.2 以函数为子组件

高阶组件并不是唯一可用于提高 React 组件代码重用的方法。在上一节的介绍中可以体会到，高阶组件扩展现有组件功能的方式主要是通过 props，增加 props 或者减少 props，或者修改原有的 props。以代理方式的高阶组件为例，新产生的组件和原有的组件说到底是两个组件，是父子关系，而两个 React 组件之间通信的方式自然是 props。因为每个组件都应该通过 propTypes 声明自己所支持的 props，高阶组件利用原组件的 props 来扩充功能，在静态代码检查上也占优势。

但是，高阶组件也有缺点，那就是对原组件的 props 有了固化的要求。也就是说，能不能把一个高阶组件作用于某个组件 X，要先看一下这个组件 X 是不是能够接受高阶组件传过来的 props，如果组件 X 并不支持这些 props，或者对这些 props 的命名有不同，或者使用方式不是预期的方式，那也就没有办法应用这个高阶组件。

举个例子，假如有这样一个高阶组件 addUserProp，它读取 loggedinUser，这是“当前登录用户”的数据，把这个数据作为新的名为 user 的 prop 传递给参数组件，代码如下：

```
const addUserProp = (WrappedComponent) => {
```

```
  class WrappingComponent extends React.Component {
    render() {
      const newProps = {user: loggedinUser};
      return <WrappedComponent {...this.props} {...newProps} />
    }
  }
  return WrappingComponent;
}
```

要使用这个高阶组件，作为参数的组件必须要能够接受名为 user 的 prop，不然应用这个高阶组件就完全没有用处。当然，我们可以给高阶组件增加其他参数，让使用者可以定制代表“当前登录用户”prop 的名字。但是，作为层层传递的 props，高阶组件这种要求参数组件必须和自己有契约的方式，会带来很多麻烦。

“以函数为子组件”的模式就是为了克服高阶组件的这种局限而生的。在这种模式下，实现代码重用的不是一个函数，而是一个真正的 React 组件，这样的 React 组件有个特点，要求必须有子组件的存在，而且这个子组件必须是一个函数。在组件实例的生命周期函数中，this.props.children 引用的就是子组件，render 函数会直接把 this.props.children 当做函数来调用，得到的结果就可以作为 render 返回结果的一部分。

我们把 addUserProp 高阶函数用“以函数为子组件”的方式重新实现，代码如下：

```
const loggedinUser = 'mock user';

class AddUserProp extends React.Component {
  render() {
    const user = loggedinUser;
    return this.props.children(user);
  }
}

AddUserProp.propTypes = {
  children: React.PropTypes.func.isRequired
}
```

AddUserProp 首字母大写，因为现在我们创造的是一个真正的组件类，而不是一个函数。这个类的代码，和被增强组件的唯一联系就是 this.props.children，而且 this.props.children 是函数类型，在 render 函数中直接调用 this.props.children 函数，参数就是我们希望传递下去的 user。

使用这个 AddUserProp 的灵活之处在于它没有对被增强组件有任何 props 要求，只是传递一个参数过去，至于怎么使用，完全由作为子组件的函数决定。

例如，我们想要让一个组件把 user 显示出来，代码如下：

```
<AddUserProp>
```

```
  { (user) => <div>{user}</div> }
</AddUserProp>
```

或者，我们想要把 user 作为一个 prop 传递给一个接受 user 名 prop 的组件 Foo，那只需要用另一个函数作为 AddUserProp 的子组件就可以，代码如下：

```
<AddUserProp>
  {
    (user) => <Foo user={user} />
  }
</AddUserProp>
```

再或者，有一个组件 Bar 也可以接受 prop，但是它接受的 prop 名为 currentUser，使用高阶组件就会很麻烦，但是使用 AddUserProp 在子组件函数中就可以完成从 user 到 currentUser 的转换，代码如下：

```
<AddUserProp>
  {
    (user) => <Bar currentUser={user} />
  }
</AddUserProp>
```

从上面三个使用样例可以看得出来，利用这种模式非常灵活，因为 AddUserProp 预期子组件是一个函数，而函数使得一切皆有可能。

作为 AddUserProp 子组件的函数，成为了连接父组件和底层组件的桥梁。一个函数可以包含各种逻辑，这样就给使用 AddUserProp 提供了最大的灵活性。“以函数为子组件”模式没有高阶组件那么多分类和应用场景，因为以函数为连接桥梁的方式已经提供了无数种用例。

6.2.1 实例 CountDown

上面的 AddUserProp 太简单，让我们利用“以函数为子组件”模式来构建一个复杂一点的例子 CountDown，实现倒计时的通用功能。

倒计时功能可以应用在很多场合中：在一个演示页面中，只是简单变化一个数字而已；在临近新年钟声响起的时候，倒计时的表现形式就是在大屏幕上显示动画的 10、9、8……3、2、1 倒数；在一个游戏场景里，倒计时的表现形式是一个炸弹上的电子计时器逐渐减少为 0。

在这些场景下，如果我们要求所有子组件都接受一个固定的 prop，那就显得太不灵活了，该是“以函数为子组件”模式上场的时候了。

我们要定义一个名为 CountDown 的组件，这个 CountDown 可以接受一个初始值作为当前数字，然后每隔一秒钟调用一次函数形式的子组件，同时把当前数字减 1，如此

重复，直到当前数字减为 0。

下面是 CountDown 组件的代码，首先 CounterDown 是一个 React 组件，构造函数代码如下：

```
class CountDown extends React.Component {
  constructor() {
    super(...arguments);
    this.state = {count: this.props.startCount};
  }
}
```

在我们的 CountDown 中，需要有一个倒数的开始值，所以还需要一个名为 startCount 的 prop。CountDown 发挥作用靠的是持续驱动子组件重新渲染。要让子组件重新渲染，CountDown 的更新过程要被驱动，所以我们需要 CountDown 包含状态，这个状态叫做 count，在构造函数里我们把它初始化为和 startCount 一样的 prop 值。

当 CountDown 组件完成装载时，在 componentDidMount 函数中通过 setInterval 函数启动每秒钟更新内部状态的操作，代码如下：

```
componentDidMount() {
  this.intervalHandle = setInterval(() => {
    const newCount = this.state.count - 1;
    if (newCount >= 0) {
      this.setState({count: newCount});
    } else {
      window.clearInterval(this.intervalHandle);
    }
  }, 1000);
}
```

为了让 CountDown 每隔一秒驱动一次更新过程，我们就要利用 JavaScript 的 setInterval 函数，我们必须记录下 setInterval 的返回结果。因为不能无限制地重复倒数，当状态 count 减小为 0，或者 CountDown 组件实例被卸载的时候，就必须取消 setInterval 引发的重复操作，我们把 setInterval 返回的结果记录在组件类的成员变量 intervalHandle 上。

为什么我们用组件类的成员变量记录 intervalHandle，而用组件状态记录 count 呢？

虽然成员变量和组件状态都是特定于某个组件实例的数据，但是组件状态的改变可以引发组件的更新过程，而普通的成员变量不会，所以实在没有理由把 intervalHandle 放在组件状态中。

值得一说的是，setInterval 帮我们把第一个函数参数中的环境 this 设为组件实例，我们之所以能够在那个函数中直接通过 this 访问 this.state 和 this.setState，是因为我们 setInterval 第一个参数是 ES6 的箭头函数形式，箭头函数会自动将自身的 this 绑定为所处环境的 this，

因为这个箭头函数所处环境是 componentDidMount，所以 this 自然就是组件实例本身。

在 componentDidUnmount 里面一定要取消并且清理掉 intervalHandle，因为一个 CountDown 组件完全可能在没有倒计时到 0 的时候就被卸载，如果不在 componentDidUnmount 取消 setInterval 引发的定时间隔操作，那么这个定时间隔操作会一直继续，继续去驱动子组件的重新执行，这会引发无法预料的后果。componentDidUnmount 函数的代码如下：

```
componentWillUnmount() {
  if (this.intervalHandle) {
    window.clearInterval(this.intervalHandle);
  }
}
```

在 CountDown 的 render 函数中，是这个模式最重要的部分，调用 this.props.children，把需要传递进去的 this.state.count 作为参数带上，代码如下：

```
render() {
  return this.props.children(this.state.count);

}
```

CountDown 组件需要两个 prop，其中 children 代表函数类型的子组件，startCount 为初始化的数值，PropType 的代码如下：

```
CountDown.propTypes = {
  children: React.PropTypes.func.isRequired,
  startCount: React.PropTypes.number.isRequired
}
```

定义好 CountDown 组件之后，就可以将其应用于任何需要倒计时的场合，所要做的只是定义恰当的函数作为 CountDown 子组件而已。

一个简单的显示倒计时从 10 到 0 的例子如下：

```
<CountDown startCount={10}>
  {
    (count) => <div>{count}</div>
  }
</CountDown>
```

类似于新年倒计时的例子，当倒计时为 0 的时候，显示“新年快乐”：

```
<CountDown startCount={10}>
  {
    (count) => <div>{ count > 0 ? count : '新年快乐'}</div>
  }
</CountDown>
```

类似于炸弹倒计时的例子，完全把倒数数字交给 Bomb 子类：

```
<CountDown startCount={10}>
  {
    (count) => <Bomb countdown={count} / >
  }
</CountDown>
```

可以看到，使用 CountDown 的方法非常灵活。

从 CountDown 的例子中我们看出一点端倪，这种“以函数为子组件”的模式非常适合于制作动画，类似 CountDown 这样的例子决定动画每一帧什么时候绘制，绘制的时候是什么样的数据，作为子组件的函数只要专注于使用参数来渲染就可以了。

实际上，React 实际中的动画库 react-motion 就大量使用了“以函数为子组件”的模式，我们在第 10 章中会详细介绍。

6.2.2 性能优化问题

“以函数为子组件”模式可以让代码非常灵活，但是凡事都有优点也有缺点，这种模式也不例外，这种模式的缺点就是难以做性能优化。

每次外层组件的更新过程，都要执行一个函数获得子组件的实际渲染结果，这个函数确实提供了灵活性。但是也因为每次渲染都要调用函数，无法利用 shouldComponentUpdate 来避免渲染浪费，使用高阶组件则可以直接使用 shouldComponentUpdate 来避免无谓的重新渲染。

可以让外层组件定制自己的 shouldComponentUpdate，但是每个组件对这个生命周期函数的定义都不同，无法使用 react-redux 那种放之四海而皆准的标准。

为了便于理解，我们设想一下如何给 CountDown 做性能优化。要知道，CountDown 也会成为其他组件的子组件，作为子组件，更新过程是可以被父组件触发的，所以我们要预期到 CountDown 在没有更改 props 的情况下被要求开始更新过程，为了防止无谓的重新渲染，我们需要借助 shouldComponentUpdate 函数：

```
shouldComponentUpdate(nextProps, nextState) {

  return nextState.count !== this.state.count;
}
```

CountDown 组件还是比较简单，因为传递给子组件的参数只有 this.state.count，所以我们在 shouldComponentUpdate 中只要看新 state 上的 count 值和当前 count 值是否相同就可以了。

另一个性能问题来自于函数形式的子组件，在上面的例子中，为了代码清晰，我们都使用在 JSX 直接定义一个箭头函数或者 Lambda 表达式的方式：

```
<CountDown startCount={10}>
```

```
  {
    (count) => <div>{count}</div>
  }
</CountDown>
```

这样用起来当然很方便，但是等于是每次渲染都会重新定义一个新的函数，那么 CountDown 的 shouldComponentUpdate 函数应该不应该把 this.props.children 和 nextProps.children 比较呢？

严格说是需要的，因为两个函数都不一样了，当然应该重新渲染，但是，如果这么做的话，就要求使用 CountDown 的地方不能使用匿名函数，比如，在使用 CountDown 的组件类中定义一个 showCount 函数，接受一个 count 参数，代码如下：

```
showCount(count) {

  return <div>{count}</div>

}
```

然后 CountDown 就可以把 showCount 作为子组件使用，代码如下：

```
<CountDown startCount={10}>

  {showCount}

</CountDown>
```

这样就失去了代码的灵活性，总之鱼与熊掌不可兼得，开发者要权衡使用。

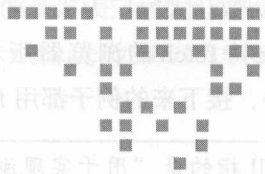
虽然“以函数为子组件”有这样性能上的潜在问题，但是它依然是一个非常棒的模式，实际上，在 react-motion 动画库中大量使用这种模式，也没有发现特别大的性能问题。要知道，对于动画功能，性能是最重要的因素，react-motion 的用户群没有反映存在性能问题，可见这种模式是性能和灵活性的一个恰当折中。

6.3 本章小结

在这一章中我们学习了 React 高级组件的用法，包括高阶组件和“以函数为子组件”的模式，这两种用法的目的都是为了重用代码。当我们发现有的功能需要在多个组件中重复时，就可以考虑构建一个高阶组件或者应用“以函数为子组件”的模式。

关于高阶组件，有两种实现方式：一种是代理方式，另一种是继承方式。通过比较不难发现，代理方式更加容易实现和控制，继承方式的唯一优势是可以操纵特定组件的生命周期函数。

和高阶组件相比，“以函数为子组件”的模式更加灵活，因为有函数的介入，连接两个组件的方式可以非常自由，在第 10 章中，我们会介绍这种模式在动画中的应用。



第 7 章 Chapter 7

Redux 和服务器通信

无论 Redux 还是 React，工作方式都是靠数据驱动，到目前为止，本书例子中的数据都是通过用户输入产生，但现实中，应用的数据往往存储在数据库中，通过一个 API 服务器暴露出来，网页应用要获得数据，就需要与服务器进行通信。

在这一章中，我们会介绍：

- React 组件访问服务器的方式；
- Redux 架构下访问服务器的方式。

React 组件访问服务器的方式适用于简单的网页应用；对于复杂的网页应用，自然会采用 Redux 来管理数据，所以在 Redux 环境中访问服务器会是我们介绍的重点。

7.1 React 组件访问服务器

我们先考虑最直接最简单的场景，在一个极其简单的网页应用中，有可能只需要单独使用 React 库，而不使用 Redux 之类的数据管理框架，这时候 React 组件自身也可以担当起和服务器通信的责任。

访问服务器本身可以使用任何一种支持网络访问的 JavaScript 库，最传统的当然是 jQuery 的 $.ajax 函数，但是我们都用上了 React 了，那也就没有必要使用 jQuery，实在没有理由再为了一个 $.ajax 函数引入一个 jQuery 库到网页里面来。

一个趋势是在 React 应用中使用浏览器原生支持的 fetch 函数来访问网络资源，fetch 函数返回的结果是一个 Promise 对象，Promise 模式能够让需要异步处理的代码简洁清晰，

这也是 fetch 函数让大家广为接受的原因。

对于不支持 fetch 的浏览器版本，也可以通过 fetch 的 polyfill 来增加对 fetch 的支持。在本书中，接下来的例子都用 fetch 来访问服务器数据资源。

polyfill 指的是“用于实现浏览器不支持原生功能的代码”，比如，现代浏览器应该支持 fetch 函数，对于不支持的浏览器，网页中引入对应 fetch 的 polyfill 后，这个 polyfill 就给全局的 window 对象上增加一个 fetch 函数，让这个网页中的 JavaScript 可以直接使用 fetch 函数了，就好像浏览器本来就支持 fetch 一样。在这个链接上 https://github.com/github/fetch 可以找到 fetch polyfill 的一个实现。

我们用一个具体实例来说明如何让 React 访问服务器。

根据前面章节中的 Todo 应用的学习得知，如果要做成一个完整的网站应用，那么应该把待办事项数据存储在服务器端，这样待办事项能够持久化存储下来，而且从任何一个浏览器访问都能够看到待办事项，当然服务器端如何存储不是本书要讨论的范围，我们的重点是开发一个新的应用来展示 React 访问服务器的方式。

近年来，天气问题成了大家都关注的问题，我们做一个能够展示某个城市天气的 React 组件，这样很有实际意义。

我们不想浪费篇幅去介绍如何构建一个服务器端 API，而是利用互联网上现成的 API 来支持我们的应用。中国天气网（http://www.weather.com.cn/）提供了 RESTful API 用于访问某个城市的天气信息，在例子中我们会利用这个 API 来获得天气数据。

7.1.1 代理功能访问 API

我们利用 create-react-app 创建一个新的应用，名叫 weather_react，读者可以在本书对应 Github 代码库 https://github.com/mocheng/react-and-redux 的 chapter-07 目录下找到完整代码。

首先我们要确定访问什么样的 API 能够获得天气信息，中国天气网提供的 RESTful API 中有访问一个城市当前天气情况的 API，规格如表 7-1 所示。

表 7-1 中国天气网获取城市天气 API 规格

规 格	描 述
主机地址	http://www.weather.com.cn/
访问方法	GET
路径	data/cityinfo/{ 城市编号 }.html
返回结果	JSON 格式，包含城市的名称 city、最低气温 temp1、最高气温 temp2

根据这样的 API 规格，如果要访问北京的天气情况，先确定北京的城市编号是 101010100，用 GET 方法访问 http://www.weather.com.cn/data/cityinfo/101010100.html，就能够得到类似下面的 JSON 格式结果：

```
{
  "weatherinfo":{
    "city":"北京",
    "cityid":"101010100",
    "temp1":"-2℃",
    "temp2":"16℃",
    "weather":"晴",
    "img1":"n0.gif",
    "img2":"d0.gif",
    "ptime":"18:00"
  }
}
```

但是，我们的网页应用中不能够直接访问中国天气网的这个 API，因为从我们本地的网页访问 weather.com.cn 域名下的网络资源属于跨域访问，而中国天气网的 API 并不支持跨域访问，所以在我们的应用中如果直接像下面那样使用 fetch 访问这个 API 资源，肯定无法获得我们预期的 JSON 格式结果：

```
fetch(http://www.weather.com.cn/data/cityinfo/101010100.html)
```

解决跨域访问 API 的一个方式就是通过代理（Proxy），让我们的网页应用访问所属域名下的一个服务器 API 接口，这个服务器接口做的工作就是把这个请求转发给另一个域名下的 API，拿到结果之后再转交给发起请求的浏览器网页应用，只是一个“代理”工作。

因为对于跨域访问 API 的限制是针对浏览器的行为，服务器对任何域名下的 API 的访问不受限制，所以这样的代理工作可以成功实现对跨域资源的访问。

在本地开发的时候，网页应用的域名是 localhost，对应的服务器域名也是 localhost，所以要在 localhost 服务器上建立一个代理。好在 create-react-app 创造的应用已经具备了代理功能，所以并不用花费时间来开发一个代理服务。

在 weather_react 应用的根目录 package.json 中添加如下一行：

```
  "proxy": "http://www.weather.com.cn/",
```

这一行配置告诉 weather_react 应用，当接收到不是要求本地资源的 HTTP 请求时，这个 HTTP 请求的协议和域名部分替换为 http://www.weather.com.cn/ 转手发出去，并将收到的结果返还给浏览器，这样就实现了代理功能。

例如，假如服务器收到了一个网页发来的 http://localhost/data/cityinfo/101010100.

html 的请求，就会发送一个请求到 http://www.weather.com.cn/data/cityinfo/101010100.html，并把这个请求结果返回给网页。

至此，我们就准备好了一个 API。

create-react-app 生成应用的 proxy 功能只是方便开发，在实际的生产环境中，使用这个 proxy 功能就不合适了，应该要开发出自己的代理服务器来满足生产环境的需要。

7.1.2 React 组件访问服务器的生命周期

在 weather_react 应用中，我们创造一个名为 Weather 的组件，这个组件可以在本书的 Github 代码库 https://github.com/mocheng/react-and-redux/ 的 chapter-07/weather_react 目录下找到。

这个 Weather 组件将要显示指定城市的当前天气情况，这个组件封装了两个功能：

- 通过服务器 API 获得天气情况数据；
- 展示天气情况数据。

现在面临的首要问题是如何关联异步的网络请求和同步的 React 组件渲染。

访问服务器 API 是一个异步操作。因为 JavaScript 是单线程的语言，不可能让唯一的线程一直等待网络请求的结果，所以所有对服务器的数据请求必定是异步请求。

但是，React 组件的渲染又是同步的，当开始渲染过程之后，不可能让 Weather 组件一边渲染一边等待服务器的返回结果。

总之，当 Weather 组件进入装载过程的时候，即使此时 Weather 立刻通过 fetch 函数发起对服务器的请求，也没法等待服务器的返回数据来进行渲染。因为 React 组件的装载过程和更新过程中生命周期函数是同步执行的，没有任何机会等待一个异步操作。

所以，可行的方法只能是这样，分两个步骤完成：

步骤 1，在装载过程中，因为 Weather 组件并没有获得服务器结果，就不显示结果。或者显示一个“正在装载”之类的提示信息，但 Weather 组件这时候要发出对服务器的请求。

步骤 2，当对服务器的请求终于获得结果的时候，要引发 Weather 组件的一次更新过程，让 Weather 重新绘制自己的内容，这时候就可以根据 API 返回结果绘制天气信息了。

从上面的过程可以看得出来，为了显示天气信息，必须要经历装载过程和更新过程，至少要渲染 Weather 组件两次。

还有一个关键问题，在装载过程中，在什么时机发出对服务器的请求呢？

通常我们在组件的 componentDidMount 函数中做请求服务器的事情，因为当生命周期函数 componentDidMount 被调用的时候，表明装载过程已经完成，组件需要渲染的内容已经在 DOM 树上出现，对服务器的请求可能依赖于已经渲染的内容，在 componentDidMount 函数中发送对服务器请求是一个合适的时机。

另外，componentDidMount 函数只在浏览器中执行，在第 12 章介绍同构时，我们会介绍 React 在服务器端渲染的过程，当 React 组件在服务器端渲染时，肯定不希望它发出无意义的请求，所以 componentDidMount 是最佳的获取初始化组件内容请求的时机。

万事俱备，我们来看一看定义 Weather 组件的 weather.js 文件内容，首先是构造函数，代码如下：

```
constructor() {
  super(...arguments);
  this.state = {weather: null};
}
```

因为 Weather 组件要自我驱动更新过程，所以 Weather 必定是一个有状态的组件，状态中包含天气情况信息，在状态上有一个 weather 字段，这个字段的值是一个对象，格式和服务器 API 返回的 JSON 数据中的 weatherinfo 字段一致。

在 render 函数中，所要做的是渲染 this.state 上的内容，代码如下：

```
render() {
  if (!this.state.weather) {
    return <div>暂无数据</div>;
  }

  const {city, weather, temp1, temp2} = this.state.weather;
  return (
    <div>
      {city} {weather} 最低气温 {temp1} 最高气温 {temp2}
    </div>
  )
}
```

在构造函数中，我们将组件状态上的 weather 字段初始化为 null。这样，在装载过程引发的第一次 render 函数调用时，就会直接渲染一个“暂无数据”的文字；但是当 state 上包含 weather 信息时，就可以渲染出实际的天气信息。

通过 API 获得数据的工作交给 componentDidMount，代码如下：

```
componentDidMount() {
  const apiUrl = `/data/cityinfo/${cityCode}.html`;
  fetch(apiUrl).then((response) => {
    if (response.status !== 200) {
```

```
        throw new Error('Fail to get response with status ' + response.status);
      }
      response.json().then((responseJson) => {
        this.setState({weather: responseJson.weatherinfo});
      }).catch((error) => {
        this.setState({weather: null});
      });
    }).catch((error) => {
      this.setState({weather: null});
    });
  }
```

fetch 函数执行会立刻返回，返回一个 Promise 类型的对象，所以后面会跟上一大串 then 和 catch 的语句。每个 Promise 成功的时候，对应的 then 中的回调函数会被调用；如果失败，对应 catch 中的回调函数也被调用。

值得注意的是，fetch 的参数 apiUrl 中只有 URL 的路径部分，没有协议和域名部分，代码如下：

```
const apiUrl = `/data/cityinfo/${cityCode}.html`;
```

这样是为了让 fetch 根据当前网页的域名自动配上协议和域名，如果当前网页地址是 http://localhost，那么 fetch 请求的 URL 就是 http://localhost/data/cityinfo/101010100.html；如果当前网页地址是 http://127.0.0.1，那么 URL 就是 http://127.0.0.1/data/cityinfo/ 101010100.html，好处就是网页代码中无需关心当前代码被部署在什么域名下。

遗憾的是，中国天气网没有提供根据访问者 IP 映射到城市的功能。在这个例子中我们硬编码了北京市的城市代码，读者在实际操作的时候，也可以把 weather.js 中的模块级变量 cityCode 改为你所在的城市，在 http://www.weather.com.cn/ 页面上搜索城市名，网页跳转后 URL 上的数字就是城市对应的城市代码。

componentDidMount 中这段代码看起来相当繁杂。不过没有办法，输入输出操作就是这样，因为 fetch 的过程是和另一个计算机实体通信，而且通信的介质也是一个无法保证绝对可靠的互联网，在这个通信过程中，各种异常情况都可能发生，服务器可能崩溃没有响应，或者服务器有响应但是返回的不是一个状态码为 200 的结果，再或者服务器返回的是一个状态码为 200 的结果，结果的实际内容可能并不是一个合法的 JSON 数据。正因为每一个环节都可能出问题，所以每一个环节都需要判断是不是成功。

虽然被 fetch 广为接受，大有取代其他网络访问方式的架势，但是它有一个特性一直被人诟病，那就是 fetch 认为只要服务器返回一个合法的 HTTP 响应就算成功，就会调用 then 提供的回调函数，即使这个 HTTP 响应的状态码是表示出错了的 400 或者 500。正因为 fetch 的这个特点，所以我们在 then 中，要做的第一件事就是检查传入参数 response

的 status 字段，只有 status 是代表成功的 200 的时候才继续，否则以错误处理。

当 response.status 为 200 时，也不能直接读取 response 中的内容，因为 fetch 在接收到 HTTP 响应的报头部分就会调用 then，不会等到整个 HTTP 响应完成。所以这时候也不保准能读到整个 HTTP 报文的 JSON 格式数据。所以，response.body 函数执行并不是返回 JSON 内容，而是返回一个新的 Promise，又要接着用 then 和 catch 来处理成功和失败的情况。如果返回 HTTP 报文内容是一个完整的 JSON 格式数据就会成功，如果返回结果不是一个 JSON 格式，比如是一堆 HTML 代码，那就会失败。

当历经各种检查最后终于获得了 JSON 格式的结果时，我们通过 Weather 组件的 this.setState 函数把 weatherinfo 字段赋值到 weather 状态上去，如果失败，就把 weather 设为 null。

处理输入输出看起来的确很麻烦，但是必须要遵照套路把所有可能出错的情况都考虑到，对任何输入输出操作只要记住一点：**不要相信任何返回结果。**

至此，Weather 功能完成了，我们打开网页刷新察看最终效果，可以看到网页最开始显示“暂无数据”，这是装载过程的渲染结果，过了一会，当通过代理调用中国天气网远端 API 返回的时候，网页上就会显示北京市的天气情况，这是 API 返回数据驱动的 Weather 组件更新过程的渲染结果，显示界面如图 7-1 所示。

图 7-1　组件 Weather 的界面

7.1.3　React 组件访问服务器的优缺点

通过上面的例子，我们可以感受到让 React 组件自己负责访问服务器的操作非常直接简单，容易理解。对于像 Weather 这样的简单组件，代码也非常清晰。

但是，把状态存放在组件中其实并不是一个很好的选择，尤其是当组件变得庞大复杂了之后。

Redux 是用来帮助管理应用状态的，应该尽量把状态存放在 Redux Store 的状态中，而不是放在 React 组件中。同样，访问服务器的操作应该经由 Redux 来完成。

接下来，我们就看一看用 Redux 来访问服务器如何做到。

7.2 Redux 访问服务器

为了展示更丰富的功能，我们扩展前面的展示天气信息应用的功能，让用户可以在若干个城市之中选择，选中某个城市，就显示某个城市的天气情况，这次我们用 Redux 来管理访问服务器的操作。

对应的代码可以在本书 Github 代码库 https://github.com/mocheng/react-and-redux/ 的 chapter-07/weather_redux 目录下找到。

我们还是用 create-react-app 创造一个新的应用，叫 weather_redux，这个应用创建 Store 的部分将使用一个叫 redux-thunk 的 Redux 中间件。

7.2.1 redux-thunk 中间件

使用 Redux 访问服务器，同样要解决的是异步问题。

Redux 的单向数据流是同步操作，驱动 Redux 流程的是 action 对象，每一个 action 对象被派发到 Store 上之后，同步地被分配给所有的 reducer 函数，每个 reducer 都是纯函数，纯函数不产生任何副作用，自然是完成数据操作之后立刻同步返回，reducer 返回的结果又被同步地拿去更新 Store 上的状态数据，更新状态数据的操作会立刻被同步给监听 Store 状态改变的函数，从而引发作为视图的 React 组件更新过程。

这个过程从头到尾，Redux 马不停蹄地一路同步执行，根本没有执行异步操作的机会，那应该在哪里插入访问服务器的异步操作呢？

Redux 创立之初就意识到了这种问题，所以提供了 thunk 这种解决方法，但是 thunk 并没有作为 Redux 的一部分一起发布，而是存在一个独立的 redux-thunk 发布包中，我们需要安装对应的 npm 包：

```
npm install --save redux-thunk
```

实际上，redux-thunk 的实现极其简单，只有几行代码，将它作为一个独立的 npm 发布而不是放在 Redux 框架中，更多的只是为了保持 Redux 框架的中立性，因为 redux-thunk 只是 Redux 中异步操作的解决方法之一，还有很多其他的方法，具体使用哪种方法开发人员可以自行决定，在后面章节会介绍 Redux 其他支持异步操作的方法。

读者可能想问 thunk 这个命名是什么含义，thunk 是一个计算机编程的术语，表示辅助调用另一个子程序的子程序，听起来有点绕，不过看看下面的例子就会体会到其中的含义。

假如有一个 JavaScript 函数 f 如下定义：

```
const f = (x) => {
  return x() + 5;
}
```

f 把输入参数 x 当做一个子程序来执行，结果加上 5 就是 f 的执行结果，那么我们试着调用一次 f：

```
const g = () => {
  return 3 + 4;
}

f(g); //结果是 (3+4)*5 = 37
```

上面代码中函数 g 就是一个 thunk，这样使用看起来有点奇怪，但有个好处就是 g 的执行只有在 f 实际执行时才执行，可以起到延迟执行的作用，我们继续看 redux-thunk 的用法来理解其意义。

按照 redux-thunk 的想法，在 Redux 的单向数据流中，在 action 对象被 reducer 函数处理之前，是插入异步功能的时机。

在 Redux 架构下，一个 action 对象在通过 store.dispatch 派发，在调用 reducer 函数之前，会先经过一个中间件的环节，这就是产生异步操作的机会，实际上 redux-thunk 提供的就是一个 Redux 中间件，我们需要在创建 Store 时用上这个中间件。如图 7-2 所示。

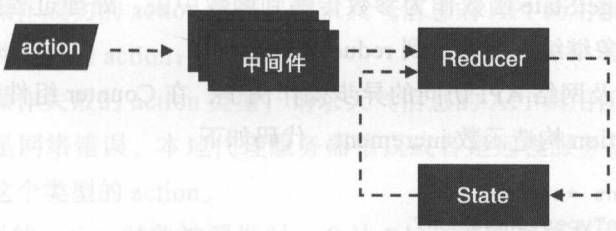


图 7-2　Redux 的 action 处理流程

在第 4 章，我们已经接触过了中间件，redux-immutable-state-invariant 这个中间件帮助开发者发现 reducer 里不应该出现的错误，现在我们要再加一个 redux-thunk 中间件来支持异步 action 对象。

我们创建的 Store.js 文件基本和 Todo 应用中基本一致，区别就是引入了 redux-thunk，代码如下：

```
import thunkMiddleware from 'redux-thunk';

const middlewares = [thunkMiddleware];
```

之前我们用一个名为 middlewares 的数组来存储所有中间件，现在只要往这个数组里加一个元素就可以了，之后，如果需要用到更多的中间件，只需要导入中间件放在 middlewares 数组中就可以。

7.2.2 异步 action 对象

当我们想要让 Redux 帮忙处理一个异步操作的时候，代码一样也要派发一个 action 对象，毕竟 Redux 单向数据流就是由 action 对象驱动的。但是这个引发异步操作的 action 对象比较特殊，我们叫它们“异步 action 对象”。

前面例子中的 action 构造函数返回的都是一个普通的对象，这个对象包含若干字段，其中必不可少的字段是 type，但是“异步 action 对象”不是一个普通 JavaScript 对象，而是一个函数。

如果没有 redux-thunk 中间件的存在，这样一个函数类型的 action 对象被派发出来会一路发送到各个 reducer 函数，reducer 函数从这些实际上是函数的 action 对象上是无法获得 type 字段的，所以也做不了什么实质的处理。

不过，有了 redux-thunk 中间件之后，这些 action 对象根本没有机会触及到 reducer 函数，在中间件一层就被 redux-thunk 截获。

redux-thunk 的工作是检查 action 对象是不是函数，如果不是函数就放行，完成普通 action 对象的生命周期，而如果发现 action 对象是函数，那就执行这个函数，并把 Store 的 dispatch 函数和 getState 函数作为参数传递到函数中去，处理过程到此为止，不会让这个异步 action 对象继续往前派发到 reducer 函数。

举一个并不涉及网络 API 访问的异步操作例子，在 Counter 组件中存在一个普通的同步增加计数的 action 构造函数 increment，代码如下：

```
const increment = () => ({
  type: ActionTypes.INCREMENT,
});
```

派发 increment 执行返回的 action 对象，Redux 会同步更新 Store 状态和视图，但是我们现在想要创造一个功能，能够发出一个“让 Counter 组件在 1 秒之后计数加一”的指令，这就需要定义一个新的异步 action 构造函数，代码如下：

```
const incrementAsync = () => {
  return (dispatch) => {
    setTimeout(() => {
      dispatch(increment());
    }, 1000);
  };
};
```

异步 action 构造函数 incrementAsync 返回的是一个新的函数，这样一个函数被 dispatch 函数派发之后，会被 redux-thunk 中间件执行，于是 setTimeout 函数就会发生作用，在 1 秒之后利用参数 dispatch 函数派发出同步 action 构造函数 increment 的结果。

这就是异步 action 的工作机理，这个例子虽然简单，但是可以看得出来，异步 action 最终还是要产生同步 action 派发才能对 Redux 系统产生影响。

redux-thunk 要做的工作也就不过如此，但因为引入了一次函数执行，而且这个函数还能够访问到 dispatch 和 getState，就给异步操作带来了可能。

action 对象函数中完全可以通过 fetch 发起一个对服务器的异步请求，当得到服务器结果之后，通过参数 dispatch，把成功或者失败的结果当做 action 对象再派发出去。这一次派发的是普通的 action 对象，就不会被 redux-thunk 截获，而是直接被派发到 reducer，最终驱动 Store 上状态的改变。

7.2.3　异步操作的模式

有了 redux-thunk 的帮助，我们可以用异步 action 对象来完成异步的访问服务器功能了，但是在此之前，我们先想一想如何设计 action 类型和视图。

一个访问服务器的 action，至少要涉及三个 action 类型：

- 表示异步操作已经开始的 action 类型，在这个例子里，表示一个请求天气信息的 API 请求已经发送给服务器的状态；
- 表示异步操作成功的 action 类型，请求天气信息的 API 调用获得了正确结果，就会引发这种类型的 action；
- 表示异步操作失败的 action 类型，请求天气信息的 API 调用任何一个环节出了错误，无论是网络错误、本地代理服务器错误或者是远程服务器返回的结果错误，都会引发这个类型的 action。

当这三种类型的 action 对象被派发时，会让 React 组件进入各自不同的三种状态，如下所示。

- 异步操作正在进行中；
- 异步操作已经成功完成；
- 异步操作已经失败。

不管网络的传输速度有多快，也不管远程服务器响应有多快，我们都不能认为“异步操作正在进行中”状态会瞬间转换为“异步操作已经成功完成”或者“异步操作已经失败”状态。前面说过，网络和远程服务器都是外部实体，是靠不住的。在开发环境下可能速度很快，所以感知不到状态转换，但在其他环境下可能会明显感觉存在延迟，所

以有必要在视图上体现三种状态的区别。

作为一种模式，我们需要定义三种 action 类型，还要定义三种对应的状态类型。

相关代码在本书的 Github 代码库 https://github.com/mocheng//react-and-redux 的 chapter-07/weather_redux 目录下可以找到。

和以往的习惯一样，我们为 Weather 组件创建一个放置所有代码的目录 weather，对外接口的文件是 src/weather/index.js，把这个功能模块的内容导出，代码如下：

```
import * as actions from './actions.js';
import reducer from './reducer.js';
import view from './view.js';

export {actions, reducer, view};
```

在 src/weather/actionTypes.js 中定义异步操作需要的三种 action 类型：

```
export const FETCH_STARTED = 'WEATHER/FETCH_STARTED';
export const FETCH_SUCCESS = 'WEATHER/FETCH_SUCCESS';
export const FETCH_FAILURE = 'WEATHER/FETCH_FAILURE';
```

在 src/weather/status.js 文件中定义对应的三种异步操作状态：

```
export const LOADING = 'loading';
export const SUCCESS = 'success';
export const FAILURE = 'failure';
```

读者可能觉得 actionTypes.js 和 status.js 内容重复了，因为三个 action 类型和三个状态是一一对应的。虽然看起来代码重复，但是从语义上说，action 类型只能用于 action 对象中，状态则是用来表示视图。为了语义清晰，还是把两者分开定义。

接下来我们看 src/weather/actions.js 中 action 构造函数如何定义，首先是三个普通的 action 构造函数，代码如下：

```
import {FETCH_STARTED, FETCH_SUCCESS, FETCH_FAILURE} from './actionTypes.js';

export const fetchWeatherStarted = () => ({
  type: FETCH_STARTED
});
export const fetchWeatherSuccess = (result) => ({
  type: FETCH_SUCCESS,
  result
})
export const fetchWeatherFailure = (error) => ({
  type: FETCH_FAILURE,
  error
})
```

三个普通的 action 构造函数 fetchWeatherStarted、fetchWeatherSuccess 和 fetchWeatherFailure 没有什么特别之处，只是各自返回一个有特定 type 字段的普通对象，它们的作用是驱动 reducer 函数去改变 Redux Store 上 weather 字段的状态。

关键是随后的异步 action 构造函数 fetchWeather，代码如下：

```
export const fetchWeather = (cityCode) => {
  return (dispatch) => {
    const apiUrl = `/data/cityinfo/${cityCode}.html`;

    dispatch(fetchWeatherStarted())

    fetch(apiUrl).then((response) => {
      if (response.status !== 200) {
        throw new Error('Fail to get response with status ' + response.status);
      }
      response.json().then((responseJson) => {
        dispatch(fetchWeatherSuccess(responseJson.weatherinfo));
      }).catch((error) => {
        throw new Error('Invalid json response: ' + error)
      });
    }).catch((error) => {
      dispatch(fetchWeatherFailure(error));
    })
  };
}
```

异步 action 构造函数的模式就是函数体内返回一个新的函数，这个新的函数可以有两个参数 dispatch 和 getState，分别代表 Redux 唯一的 Store 上的成员函数 dispatch 和 getState。这两个参数的传入是 redux-thunk 中间件的工作，至于 redux-thunk 如何实现这个功能，我们在后面关于中间件的章节会详细介绍。

在这里，我们只要知道异步 action 构造函数的代码基本上都是这样的套路，代码如下：

```
export const sampleAsyncAction = () => {
  return (dispatch, getState) => {
    //在这个函数里可以调用异步函数，自行决定在合适的时机通过dispatch参数
    //派发出新的action对象。
  }
}
```

在我们的例子中，异步 action 对象返回的新函数首先派发由 fetchWeatherStarted 的产生的 action 对象。这个 action 对象是一个普通 action 对象，所以会同步地走完单向数据流，一直走到 reducer 函数中，引发视图的改变。同步派发这个 action 对象的目的是将视

图置于“有异步 action 还未结束”的状态，完成这个提示之后，接下来才开始真正的异步操作。

这里使用 fetch 来做访问服务器的操作，和前面介绍的 weather_react 应用中的代码几乎一样，区别只是 this.setState 改变组件状态的语句不见了，取而代之的是通过 dispatch 来派发普通的 action 对象。也就是说，访问服务器的异步 action，最后无论成败，都要通过派发 action 对象改变 Redux Store 上的状态完结。

在 fetch 引发的异步操作完成之前，Redux 正常工作，不会停留在 fetch 函数执行上，如果有其他任何 action 对象被派发，Redux 照常处理。

我们来看一看 src/weather/reducer.js 中的 reducer 函数，代码如下：

```
export default (state = {status: Status.LOADING}, action) => {
  switch(action.type) {
    case FETCH_STARTED: {
      return {status: Status.LOADING};
    }
    case FETCH_SUCCESS: {
      return {...state, status: Status.SUCCESS, ...action.result};
    }
    case FETCH_FAILURE: {
      return {status: Status.FAILURE};
    }
    default: {
      return state;
    }
  }
}
```

在 reducer 函数中，完成了上面提到的三种 action 类型到三种状态类型的映射，增加一个 status 字段，代表的就是视图三种状态之一。

这里没有任何处理异步 action 对象的逻辑，因为异步 action 对象在中间件层就被 redux-thunk 拦截住了，根本没有机会走到 reducer 函数中来。

最后来看看 src/weather/view.js 中的视图，也就是 React 组件部分，首先是无状态组件函数，代码如下：

```
const Weather = ({status, cityName, weather, lowestTemp, highestTemp}) => {
  switch (status) {
    case Status.LOADING: {
      return <div>天气信息请求中...</div>;
    }
    case Status.SUCCESS: {
      return (
        <div>
```

```
            {cityName} {weather} 最低气温 {lowestTemp} 最高气温 {highestTemp}
          </div>
        )
      }
      case Status.FAILURE: {
        return <div>天气信息装载失败</div>
      }
      default: {
        throw new Error('unexpected status ' + status);
      }
    }
  }
```

和 weather_react 中的例子不同，因为现在状态都是存储在 Redux Store 上，所以这里 Weather 是一个无状态组件，所有的 props 都是通过 Redux Store 状态获得。

在渲染函数中，根据三种不同的状态，显示出来的内容也不一样。当视图状态为 LOADING 时，表示一个对天气信息的 API 请求刚刚发出，还没有结果返回，这时候界面上就显示一个“天气信息请求中…”字样；当视图状态为 SUCCESS 时，根据状态显示对应的城市和天气信息；当视图状态为 FAILURE 时，显示“天气信息装载失败”。

这个组件对应的 mapStateToProps 函数代码如下：

```
const mapStateTopProps = (state) => {
  const weatherData = state.weather;

  return {
    status: weatherData.status,
    cityName: weatherData.city,
    weather: weatherData.weather,
    lowestTemp: weatherData.temp1,
    highestTemp: weatherData.temp2
  };
}
```

为了驱动 Weather 组件的 action，我们另外创建一个城市选择器控件 CitySelector，CitySelector 很简单，也不是这个应用功能的重点，我们只需要它提供一个作为视图的 React 组件就可以。

在 CitySelector 组件中，定义了四个城市的代码，代码如下：

```
const CITY_CODES = {
  '北京': 101010100,
  '上海': 101020100,
  '广州': 101280101,
  '深圳': 101280601
};
```

CitySelector 组件的 render 函数根据 CITY_CODES 的定义画出四个城市的选择器，代码如下：

```
render() {
  return (
    <select onChange={this.onChange}>
      {
        Object.keys(CITY_CODES).map(
          cityName => <option key={cityName} value={CITY_CODES[cityName]}>
            {cityName}</option>
        )
      }
    </select>
  );
}
}
```

其中使用到的 onChange 函数使用 onSelectCity 来派发出 action，代码如下：

```
onChange(ev) {
  const cityCode = ev.target.value;
  this.props.onSelectCity(cityCode)
}
```

为了让网页初始化的时候就能够获得天气信息，在 componentDidMount 中派发了对应第一个城市的 fetchWeatheraction 对象，代码如下：

```
componentDidMount() {
  const defaultCity = Object.keys(CITY_CODES)[0];
  this.props.onSelectCity(CITY_CODES[defaultCity]);
}
```

CitySelector 的 mapDispatchToProps 函数提供了名为 onSelectCity 的函数类型 prop，代码如下：

```
const mapDispatchToProps = (dispatch) => {
  return {
    onSelectCity: (cityCode) => {
      dispatch(weatherActions.fetchWeather(cityCode));
    }
  }
};
```

这个 city_selector 提供的视图导入了 weather 功能组件导出的 actions，显示出北京、上海、广州、深圳四个城市的选择器，当用户选中某个城市的时候，就会派发 fetchWeather 构造器产生的 action 对象，让 Weather 组件去服务器获取对应城市的天气信息。

完成全部代码之后，我们在网页中就可以看到最终效果，如图 7-3 所示。

当我们选择另一个城市之后，可以看到会有短暂的显示“天气信息请求中 ...”，然后才显示出对应城市的天气，因为访问服务器总是会有时间延迟。

我们也可以试着关闭命令行上中断 npm start 这个命令，等于是关闭了代理服务器，这样对 API 的访问必然失败，然后切换城市，可以看到“天气信息装载失败”。

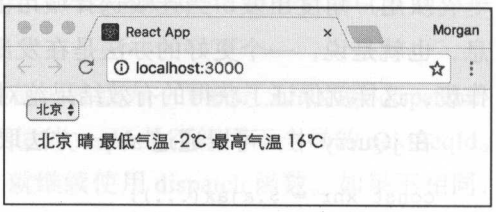


图 7-3　天气应用最终效果

7.2.4　异步操作的中止

对于访问服务器这样的异步操作，从发起操作到操作结束，都会有段时间延迟，在这段延迟时间中，用户可能希望中止异步操作。

从执行一个 fetch 语句发出请求，到获得服务器返回的结果，可能很快只有几十毫秒，也可能要花上好几秒甚至十几秒，如果没有超时限制的话，就算是等上几分钟也完全是可能的。也就是说，从一个请求发出到获得响应这个过程中，用户可能等不及了，或者改变主意想要执行另一个操作，用户就会进行一些操作引发新的请求发往服务器，而这就是我们开发者需要考虑的问题。

在 weather_redux 这个应用中，如果当前城市是“北京”，用户选择了“上海”之后，不等服务器返回，立刻又选择“广州”，那么，最后显示出来的天气是什么呢？

结果是难以预料的，用户的两次选择城市操作，引发了两次 API 请求，最后结果就看哪个请求先返回结果。假如是关于“上海”的请求先返回结果，界面上就会先显示上海的天气信息，然后关于“广州”的请求返回，界面上又自动更新为广州的天气信息。假如是关于“广州”的请求先返回，关于“上海”的请求后返回，那么结果就正相反，最后显示的是上海的天气信息。此时界面上会出现严重的信息不一致，城市选择器上显示的是“广州”，但是天气信息部分却是“上海”。

两次 API 的请求顺序是“上海”“广州”，有可能返回的顺序是“广州”“上海”吗？完全可能，访问服务器这样的输入输出操作，复杂就复杂在返回的结果和时间都是不可靠的，即使是访问同样一个服务器，也完全可能先发出的请求后收到结果。

要解决这种界面上显示不一致的问题，一种方法是在视图上做文章，比如当一个 API 请求发出去，立刻将城市选择器锁住，设为不可改变，直到 API 请求返回结果才解锁。这种方式虽然可行，但是给用户的体验可能并不好，用户希望随时能够选择城市，而服务器的响应时间完全不可控，锁住城市选择器的时间可能很长，而且这个时间由服务器响应时间决定，不在代码控制范围内，如果服务器要等 10 秒钟才返回结果，锁住城

市选择器的时间就有 10 秒，这是不可接受的。

从用户角度出发，当连续选择城市的时候，总是希望显示最后一次选中的城市的信息，也就是说，一个更好的办法是在发出 API 请求的时候，将之前的 API 请求全部中止作废，这样就保证了获得的有效结果绝对是用户的最后一次选择结果。

在 jQuery 中，可以通过 abort 方法取消掉一个 AJAX 请求：

```
const xhr = $.ajax(...);

xhr.abort(); //取消掉已经发出的AJAX请求
```

但是，很不幸，对于 fetch 没有对应 abort 函数的功能，因为 fetch 返回的是一个 Promise 对象，在 ES6 的标准中，Promise 对象是不存在“中断”这样的概念的。

既然 fetch 不能帮助我们中止一个 API 请求，那就只能在应用层实现“中断”的效果，有一个技巧可以解决这个问题，只需要修改 action 构造函数。

我们对 src/weather/actions.js 进行一些修改，代码如下：

```
let nextSeqId = 0;
export const fetchWeather = (cityCode) => {
  return (dispatch) => {
    const apiUrl = `/data/cityinfo/${cityCode}.html`;
    const seqId = ++ nextSeqId;
    const dispatchIfValid = (action) => {
      if (seqId === nextSeqId) {
        return dispatch(action);
      }
    }
    dispatchIfValid(fetchWeatherStarted())
    fetch(apiUrl).then((response) => {
      if (response.status !== 200) {
        throw new Error('Fail to get response with status ' + response.status);
      }
      response.json().then((responseJson) => {
        dispatchIfValid(fetchWeatherSuccess(responseJson.weatherinfo));
      }).catch((error) => {
        dispatchIfValid(fetchWeatherFailure(error));
      });
    }).catch((error) => {
      dispatchIfValid(fetchWeatherFailure(error));
    })
  };
}
```

在 action 构造函数文件中定义一个文件模块级的 nextSeqId 变量，这是一个递增的整数数字，给每一个访问 API 的请求做序列编号。

在 fetchWeather 返回的函数中，fetch 开始一个异步请求之前，先给 nextSeqId 自增

加一，然后自增的结果赋值给一个局部变量 seqId，这个 seqId 的值就是这一次异步请求的编号，如果随后还有 fetchWeather 构造器被调用，那么 nextSeqId 也会自增，新的异步请求会分配为新的 seqId。

然后，action 构造函数中所有的 dispatch 函数都被替换为一个新定义的函数 dispatchIfValid，这个 dispatchIfValid 函数要检查一下当前环境的 seqId 是否等同于全局的 nextSeqId。如果相同，说明 fetchWeather 没有被再次调用，就继续使用 dispatch 函数。如果不相同，说明这期间有新的 fetchWeather 被调用，也就是有新的访问服务器的请求被发出去了，这时候当前 seqId 代表的请求就已经过时了，直接丢弃掉，不需要 dispatch 任何 action。

虽然不能真正“中止”一个 API 请求，但是我们可以用这种方法让一个 API 请求的结果被忽略，达到了中止一个 API 请求一样的效果。

在这个例子中 Weather 模块只有一种 API 请求，所以一个 API 调用编号序列就足够，如果需要多种 API 请求，则需要更多类似 nextSeqId 的变量来存储调用编号。

拥有异步操作中止功能的代码，在本书 Github 代码库 https://github.com/mocheng/react-and-redux/ 的 chapter-07/weather_redux_improved 目录下找到，启动对应的应用，可以看到无论如何选择城市，最终显示的天气信息和选中的城市都是一致的。

7.3　Redux 异步操作的其他方法

上述的 redux-thunk 并不是在 Redux 中处理异步操作的唯一方式，只不过 redux-thunk 应该是应用最简单，也是最容易被理解的一种方式。

在 Redux 的社区中，辅助进行异步操作的库还有：

- redux-saga
- redux-effects
- redux-side-effects
- redux-loop
- redux-observable

上面列举的只是最负盛名的一些库，并不是完整清单，而且随着更多的解决方法出现，这个列表肯定还将不断增长。

7.3.1　如何挑选异步操作方式

所有这些辅助库，都需要通过一个 Redux 中间件或者 Store Enhancer 来实现 Redux 对异步操作的支持，每一个库都足够写一本书出来讲解，所以没法在这里一一详细介绍，在这里我们只是列出一些要点，帮助读者研究让 Redux 支持异步操作的库时需要考虑哪些方面。

第一，在 Redux 的单向数据流中，什么时机插入异步操作？

Redux 的数据流转完全靠 action 来驱动，图 7-2 显示了数据流转的过程，对于 redux-thunk，切入异步操作的时机是在中间件中，但是这不是唯一的位置。

通过定制化 Store Enhancer，可以在 action 派发路径上任何一个位置插入异步操作，甚至作为纯函数的 reducer 都可以帮助实现异步操作。异步操作本身就是一种副作用，reducer 的执行过程当然不应该产生异步操作，但是 reducer 函数的返回值却可以包含对异步操作的“指示”。也就是说，reducer 返回的结果可以用纯数据的方式表示需要发起一个对服务器资源的访问，由 reducer 的调用者去真正执行这个访问服务器资源的操作，这样不违背 reducer 是一个纯函数的原则，在 redux-effects 中使用的就是这种方法。

很遗憾，很多库的文档并没有解释清楚自己切入异步操作的位置，这就容易导致很多误解，需要开发者自己去发掘内在机制。只有确定了切入异步操作的位置，才能了解整个流程，不会犯错。

第二，对应库的大小如何？

有的库看起来功能很强大，单独一个库就有几十 KB 大小的体积，比如 redux-saga，发布的最小化代码有 25KB，经过 gzip 压缩之后也有 7KB，要知道 React 本身被压缩之后也不过是 45KB 大小。

不同的应用对 JavaScript 的体积有不同的要求。比如，对于视频类网站，观看视频本来就要求访问者的网络带宽比较优良，那多出来的这些代码大小就不会有什么影响。但是对于一些预期会在网络环境比较差的情况下访问的网站，可能就需要计较一下是否值得引入这些库。

第三，学习曲线是不是太陡？

所有这些库都涉及一些概念和背景知识，导致学习曲线比较陡，比如 redux-saga 要求开发者能够理解 ES6 的 async 和 await 语法，redux-observable 是基于 Rx.js 库开发的，要求开发者已经掌握响应式编程的技巧。

如果一个应用只有一个简单的 API 请求，而且使用 redux-thunk 就能够轻松解决问题，那么选择一个需要较陡学习曲线的辅助库就显得并不是很恰当；但是如果应用中包含大量的 API 请求，而且每个请求之间还存在复杂的依赖关系，这时候也许就是考虑使用某个辅助库的时机。

切记，软件开发是团队活动，选用某种技术的时候，不光要看自己能不能接受，还也要考虑团队中其他伙伴是否容易接受这种技术。毕竟，软件开发的终极目的是满足产品需求，不要在追逐看似更酷更炫的技术中迷失了初心。

第四，是否会和其他 Redux 库冲突？

所有这些库都是以 Redux 中间件或者 Redux Store Enhancer 的形态出现，在用 Redux

的 createStore 创建 store 实例时，可能会组合多个中间件和多个 Store Enhancer，在 Store 这个游戏场上，不同的玩家之间可能会发生冲突。

总之，使用任何一个库在 Redux 中实现异步操作，都需要多方面的考虑，到目前为止，业界都没有一个公认的最佳方法。

相对而言，虽然 redux-thunk 容易产生代码臃肿的问题，但真的是简单又易用，库也不大，只有几行代码而已，在第 9 章中我们会详细介绍 redux-thunk 的实现细节。

7.3.2　利用 Promise 实现异步操作

除了 redux-thunk，还有另一种异步模式，将 Promise 作为特殊处理的异步 action 对象，这种方案比 redux-thunk 更加易用，复杂度也不高。

fetch 函数返回的结果也是一个 Promise 对象，用 Promise 来连接访问 API 操作和 Redux，是天作之合。

不过，对于 Promise 在 Redux 中应该如何使用，也没有形成统一观点，相关的库也很多，但是都很简单，用一个 Redux 中间件就足够实现：

- redux-promise
- redux-promises（名字只比上面的多了一个表示复数的 s）
- redux-simple-promise
- redux-promise-middleware

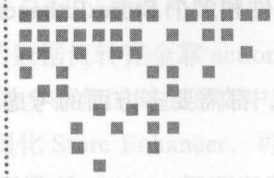
同样，这样一个清单可能也会不断增长，所以我们也不逐一介绍，在第 9 章中，我们会创造自己的基于 promise 的中间件来实现异步功能。

7.4　本章小结

在这一章中我们介绍了一个网页应用必须具备的功能，通过 API 访问获取服务器数据资源。

无论是从服务器获取数据，还是向服务器提交数据，都是一个异步的过程。在一个 React 组件中，我们可以利用 componentDidMount，在装载过程结束时发起对服务器的请求来获取数据填充组件内容。

在一个 Redux 应用中，状态要尽量存在 Redux 的 Store 上，所以单个 React 组件访问服务器的方案就不适用了，这时候我们需要在 Redux 中实现异步操作。最简单直接的方法是使用 redux-thunk 这个中间件，但是也有其他的库作为选择，每种方案都有其优缺点，开发者要了解权衡决定哪种库适合自己的应用。



Chapter 8 第 8 章

单元测试

“我发现写单元测试实际上提高了我的编程速度。”——Martin Fowler

作为验证程序质量的重要手段之一，测试是一个很大的主题，不过，在这一章中我们重点介绍的是特定于 React 和 Redux 应用的单元测试方法。

在本章会介绍

- 单元测试原则；
- React 和 Redux 的单元测试环境；
- 单元测试 React 组件的方法；
- 单元测试 Redux 各个部分的方法。

因为 React 和 Redux 基于函数式编程的思想，所以应用功能更容易拆分成容易测试的模块，对应产出代码的可测试性也更高。

8.1 单元测试的原则

从不同的角度，可将测试划分为如下不同的种类：

- 从人工操作还是写代码来操作的角度，可以分为手工测试和自动化测试；
- 从是否需要考虑系统的内部设计角度，可以分为白盒测试和黑盒测试；
- 从测试对象的级别，可以分为单元测试、集成测试和端到端测试；

- 从测试验证的系统特性，又可以分为功能测试、性能测试和压力测试。

……

上述的测试种类中，有很多与被测试程序是基于什么语言、什么框架没有任何关系，比如端到端测试，对使用 React 和 Redux 写的应用，和对 Backbone.js 和 jQuery 写的应用，测试本身没有什么区别。在这里，我们只探讨特定于 React 和 Redux 的测试技巧，而体现特殊性的就是单元测试。

单元测试是一种自动化测试，测试代码和被测的对象非常相关，比如测试 React 组件的代码就和测试 jQuery 插件的代码完全不是一回事。

单元测试代码一般都由编写对应功能代码的开发者来编写，开发者提交的单元测试代码应该保持一定的覆盖率，而且必须永远能够运行通过。可以说，单元测试是保证代码质量的第一道防线。

既然说到单元测试，就不得不说测试驱动开发（Test Driven Development，TDD），有的开发者对测试驱动开发奉为神器，严格实践先写单元测试测试用例后写功能代码，而且单元测试也保证其他开发者不会因为失误破坏原有的功能；也有开发者对测试驱动开发不以为然，因为写单元测试的时间是写功能代码的好几倍，当需求发生改变的时候，除了要维护功能代码，还要维护测试代码，苦不堪言。

这里我们也不要争论测试驱动开发是否有必要，我们只需要正视一点，那就是单元测试应该是让开发者的工作更轻松更高效，而不是成为开发过程中的包袱。

每个开发团队会根据自身特点决定是否要求测试驱动开发，也可以设定恰当的单元测试覆盖阈值，不过要注意以下几点：

首先，即使单元测试覆盖率达到了 100%，也不表示程序是没有 bug 的，实现高质量的软件有多方面要求，单元测试只是手段之一，不要对单元测试覆盖率有太过偏执的要求。

另外，程序架构的可测试性非常重要，开发者不喜欢写单元测试代码一个很主要的原因就是发现单元测试“太难写”，比如为了写一个单元测试要写太多的模仿对象（Mock），涉及复杂的流程难以全部条件分支。

要克服单元测试“太难写”的问题，就需要架构能把程序拆分成足够小到方便测试的部分，只要每个小的部分被验证能够正确地各司其职，组合起来能够完成整体功能，那么开发者编写的单元测试就可以专注于测试各个小的部分就行，这就是更高的“可测试性”。

只要应用得当，React 和 Redux 应用的可测试性非常高，因为对应单元测试写出来大多就是对纯函数的测试。

在本书前面的例子中，都是尽量不让 React 存储状态，把状态存储到 Redux 的 Store 上，也就是让 React 组件只是一个根据数据负责渲染的纯函数就好，这样的 React 组件是非常方便测试的，每个写过单元测试的开发者都会有体会，测试一个根据输入返回输出的纯函数，要比测试一个包含很多状态的对象容易得多。因为纯函数的结果根据输入完全可以预测，而为了测试一个对象在某个状态下的行为，还要首先让这个对象处于那个状态，测试代码自身就会变得很长，让开发者担心测试代码本身就可能因为复杂化而出问题。

既然 React 组件变成了纯函数，那 Redux 就承担了复杂的状态管理，那是否 Redux 部分的可测试性会变低呢？并没有，因为 Redux 的功能由众多函数组成，这些函数中不少依然是纯函数，所以测试依然非常简单。

使用 Redux 的应用，开发者书写的大部分代码都属于 action 构造函数、reducer 或者选择器，其中普通的 action 构造函数和 reducer 就是纯函数，异步 action 构造函数有副作用所以并不是纯函数，在接下来我们会介绍如何测试异步 action 构造函数；选择器虽然包含缓存的副作用，但是对于同样的输入也有同样的输出，测试难度不比测试纯函数更高。

接下来，我们就介绍单元测试 React 和 Redux 程序的方法。

8.2 单元测试环境搭建

要实现单元测试，需要搭建单元测试环境，包括下面几个方面。

- 单元测试框架；
- 单元测试代码组织；
- 辅助工具。

8.2.1 单元测试框架

构建 React 和 Redux 单元测试环境，首先要确定测试框架，有很多选择，最常见的是以下两种：

- 用 Mocha 测试框架，但是 Mocha 并没有断言库，所以往往还要配合 Chai 断言库来使用，也就是 Mocha+Chai 的组合。
- 使用 React 的本家 Facebook 出品的 Jest，Jest 自带了断言等功能，相当于包含了 Mocha 和 Chai 的功能，不过 Jest 的语法和 Chai 并不一致。

这两种方法各有千秋，没有哪一种具备绝对优势。

在 create-react-app 创建的应用中自带了 Jest 库，所以本书的代码库中单元测试都是基于 Jest 框架编写的代码，在任何一个 create-react-app 产生的应用代码目录下，用命令行执行下列代码，就会进入单元测试的界面：

```
npm run test
```

上面的命令在执行完单元测试后不会结束，而是进入待命状态，如图 8-1 所示。

在待命状态下，任何对相关代码的修改，都会触发单元测试的运行，而且 Jest 很智能，只运行修改代码影响的单元测试。同时也提供了强行运行所有单元测试代码、选择只运行满足过滤条件的单元测试用例等高级功能。

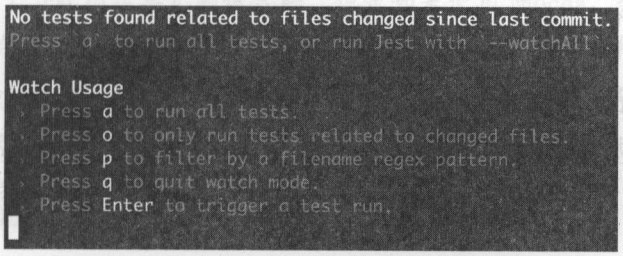


图 8-1 Jest 的命令行界面

Jest 会自动在当前目录下寻找满足下列任一条件的 JavaScript 文件作为单元测试代码来执行。

- 文件名以 .test.js 为后缀的代码文件；
- 存于 __test__ 目录下的代码文件。

一种方式，是在项目的根目录上创建一个名为 test 的目录，和存放功能代码的 src 目录并列，在 test 目录下建立和 src 对应子目录结构，每个单元测试文件都以 .test.js 后缀，就能够被 Jest 找到。这种方法可以保持功能代码 src 目录的整洁，缺点就是单元测试中引用功能代码的路径会比较长。例如，功能代码在 src/todo/actions.js 文件中，对应的单元测试代码放在 test/todo/actions.test.js 文件中，这样一来，在 actions.test.js 文件中导入被测试的功能代码，就需要一个很长的路径名，代码如下：

```
import * as actions from '../../src/todos/actions.js';
```

另一种存放单元测试代码的策略是在每一个目录下创建 __test__ 子目录，用于存放对应这个目录的单元测试。这种方法因为功能代码和测试代码放在一起，容易比对，缺点就是散布在各个目录下的 __test__ 看起来不是很整洁。

具体用何种方式布局单元测试代码文件，开发者可以自行决定，在本书的例子中，更注重功能代码目录的整洁，所以测试代码都放在和 src 并排的 test 目录中，文件以 .test.js

后缀结尾。

8.2.2 单元测试代码组织

单元测试代码的最小单位是测试用例（test case），每一个测试用例考验的是被测试对象在某一个特定场景下是否有正确的行为。在 Jest 框架下，每个测试用例用一个 it 函数代表，it 函数的第一个参数是一个字符串，代表的就是测试用例名称，第二参数是一个函数，包含的就是实际的测试用例过程。

一个很简单的单元测试用例代码如下：

```
it('should return object when invoked', () => {
  //增加断言语句
});
```

测试用例用 it 函数代表，这个函数名 it 指代的“它”就是被测对象，所以第一个参数的用例名称就应该描述“它”的预期行为，比较好的测试用例名遵循这样的模式：“（它）在什么样的情况下是什么行为”，应该尽量在 it 函数的第一个参数中使用这样有意义的字符串。

为了测试被测对象在多种情况下的行为，就需要创建多个单元测试用例，因此，接下来的问题就是如何组织多个 it 函数实例，也就是测试套件（test suite）的构建。

一个测试套件由测试用例和其他测试套件构成，很明显，测试套件可以嵌套使用，于是测试套件和测试用例形成了一个树形的组织结构。当执行某个测试套件的时候，按照从上到下从外到里的顺序执行所有测试用例。

在 Jest 中用 describe 函数描述测试套件，一个测试套件的代码例子如下：

```
describe('actions', () => {
  it('should return object when invoked', () => {
  });
  //可以有更多的it函数调用
});
```

describe 函数包含与 it 函数一样的参数，两者主要的区别就是 describe 可以包含 it 或者另一个 describe 函数调用，但是 it 却不能。

将多个 it 放到一个 describe 中的主要目的是为了重用共同的环境设置。比如一组 it 中都需要创建一个 Redux Store 实例作为测试的前提条件，让每个 it 中都进行这个操作就是重复代码，这时就应该把这些 it 放在一个 describe 中，然后利用 describe 下的 beforeEach 函数来执行共同的创建 Redux Store 工作。

describe 中有如下特殊函数可以帮助重用代码。

- beforeAll，在开始测试套件开始之前执行一次；
- afterAll，在结束测试套件中所有测试用例之后执行一次；
- beforeEach，每个测试用例在执行之前都执行一次；
- afterEach，每个测试用例在执行之后都执行一次。

假设一个 descibe 中包含上面所描述的四个函数，并包含两个 it 测试用例，那么首先执行 beforeAll 函数，随后执行 beforeEach 函数，接下来执行第一个 it 函数，接着执行 afterEach 函数，接下来，又是依次执行 beforeEach、it 和 afterEach 函数，最后执行的是 afterAll 函数。

提示 本书中基于 Jest 构建单元测试代码，但是如果使用 Mocha，会发现代码很相似，一样也是使用 describe 和 it 来代表测试套件和测试用例，一样利用 beforeEach 和 beforeEnd 执行通用的代码。但是 Mocha 没有 beforeAll 和 afterAll，取而代之的是 before 和 after 函数，这是两个测试框架的微小区别。

8.2.3 辅助工具

特定于 React 和 Redux 的单元测试，还需要几个辅助类。

1. Enzyme

要方便地测试 React 组件，就需要用到 Enzyme，有意思的是 Enzyme 并不是 Facebook 出品，而是 AirBnB 贡献出来的开源项目。

要使用 Enzyme，需要安装对应的 npm 包：

```
npm install —save-dev enzyme react-addons-test-utils
```

上面的 react-addons-test-utils 是 Facebook 提供的单元测试辅助库，Enzyme 依赖这个库，但是这个库的本身功能没有 Enzyme 那么强大。

测试 React 组件，需要将 React 组件渲染出来看一看结果，不过 Enzyme 认为并不是所有的测试过程都需要把 React 组件的 DOM 树都渲染出来，尤其对于包含复杂子组件的 React 组件，如果深入渲染整个 DOM 树，那就要渲染所有子组件，可是子组件可能会有其他依赖关系，比如依赖于某个 React Context 值，为了渲染这样的子组件需要耗费很多精力准备测试环境，这种情况下啊，针对目标组件的测试只要让它渲染顶层组件就好了，不需要测试子组件。

Enzyme 支持三种渲染方法：

- shallow，只渲染顶层 React 组件，不渲染子组件，适合只测试 React 组件的渲染行为；

- mount，渲染完整的 React 组件包括子组件，借助模拟的浏览器环境完成事件处理功能；
- render，渲染完整的 React 组件，但是只产生 HTML，并不进行事件处理。

例如，对于 Filter 组件，代码如下：

```
const Filter = () => (
  <p className="filters">
    <Link filter={FilterTypes.ALL}> {FilterTypes.ALL} </Link>
    <Link filter={FilterTypes.COMPLETED}> {FilterTypes.COMPLETED} </Link>
    <Link filter={FilterTypes.UNCOMPLETED}> {FilterTypes.UNCOMPLETED} </Link>
  </p>
);
```

在测试 Filter 组件的时候，如果只专注于 Filter 的功能，只要保证这个渲染结果包含三个 Filter 组件就足够了，没有必要把 Link 组件内容渲染出来，因为那是 Link 组件的单元测试应该做的事情，Enzyme 这种“浅层渲染”的方法叫 shallow。

如果想要渲染完整的 DOM 树，甚至想要看看 Link 中的点击是否获得预期效果，可以选择 Enzyme 的方法 mount，mount 不光产生 DOM 树，还会加上所有组件的事件处理函数，可以模拟一个浏览器中的所有行为。

如果只想检查 React 组件渲染的完整 HTML，不需要交互功能，可以使用 Enzyme 提供的 render 函数。

2. sinon.js

React 和 Redux 已经尽量让单元测试面对的是纯函数，但是还是不能避免有些被测试的对象依赖于一些其他因素。比如，对于异步 action 对象，就会依赖于对 API 服务器的网络请求，毫无疑问在单元测试中不能真正地访问一个 API 服务器，所以需要模拟网络访问的结果。

开源社区存在很多模拟网络请求的单元测试辅助工具，不过对于“模拟”这件事，不应该只是局限于网络请求，所以这里我们使用一个全能的模拟工具 sinon.js。

sinon.js 功能强大，可以改变指定对象的行为，甚至改变测试环境的时钟设置。

为了使用 sinons.js，需要如下安装对应的 npm 包：

```
npm install —save-dev sinon
```

3. redux-mock-store

虽然 Redux 简单易用，但是在某些情况下并不需要完整的 Redux 功能，一个模拟的 Redux Store 使用起来更加方便。比如对于测试一个异步 action 构造函数时，异步 action 构造函数会往 Store 中连续派发 action 对象，从测试角度并不需要 action 对象被派发到

reducer 中，只要能够检查 action 对象被派发就足够了，这样就能够用上 redux-mock-store。

安装 redux-mock-store 的方法如下：

```
npm install —save-dev redux-mock-store
```

8.3 单元测试实例

单元测试的要义是一次只测试系统的一个功能点，现在我们就开始看一看在 React 和 Redux 的应用中各个功能点如何测试吧。

8.3.1 action 构造函数测试

普通的 action 构造函数是很简单的单元测试，简单到往往让开发者觉得很无聊。

对应于第 5 章的 todo_with_selector/src/todos/actions.js 的单元测试代码，存放在文件 todo_with_selector/test/todos/actions.test.js 中，首先看测试套件的组织，代码如下：

```
describe('todos/actions', () => {
  describe('addTodo', () => {
    //在这里添加it测试用例
  });
});
```

因为测试目标是 todos 功能模块的 action 构造函数，最顶层的 describe 名为“todos/actions”，不过 todos 包含 addTodo、toggleTodo 和 removeTodo 这些 action 构造函数。为了便于阅读，我们为每一个构造函数单独创建一个 describe 测试套件。

接下来我们看看针对 addTodo 的测试用例，首先需要验证 addTodo 函数执行返回的结果是预期的对象，代码如下：

```
it('should create an action to add todo', () => {
  const text = 'first todo';
  const action = addTodo(text);

  expect(action.text).toBe(text);
  expect(action.completed).toBe(false);
  expect(action.type).toBe(actionTypes.ADD_TODO);
});
```

多次调用 addTodo 函数返回的对象具有一样的内容，只有 id 值是不同的，因为每一个新创建的待办事项都要有唯一的 id，对应的单元测试代码如下：

```
it('should have different id for different actions', () => {
  const text = 'first todo';
  const action1 = addTodo(text);
```

```
  const action2 = addTodo(text);
  expect(action1.id !== action2.id).toBe(true);
});
```

从上面代码可以看出单元测试的基本套路如下：

1）预设参数；

2）调用纯函数；

3）用 expect 验证纯函数的返回结果。

8.3.2 异步 action 构造函数测试

异步 action 构造函数因为存在副作用，所以单元测试会比普通 action 构造函数复杂。

一个异步 action 对象就是一个函数，被派发到 redux-thunk 中间件时会被执行，产生副作用，以第 7 章中的 weather_redux/src/weather/actions.js 异步 action 对象构造器 fetchWeather 为例，产生的异步动作被派发之后，会连续派发另外两个 action 对象代表 fetch 开始和 fetch 结束，单元测试要做的就是验证这样的行为。

被测试对象 fetchWeather 发挥作用需要使用 redux-thunk 中间件，所以需要一个 Redux Store，在 fetchWeather 函数中还需要调用 dispatch 函数，而 dispatch 函数也是来自于一个 Redux Store。但是我们并没有必要创建一个完整功能的 Redux Store，使用 redux-mock-store 更加合适，因为在单元测试环境下，dispatch 函数最好不要做实际的派发动作，只要能够把被派发的对象记录下来，留在验证阶段读取就可以了。

使用 redux-mock-store 的代码如下：

```
import thunk from 'redux-thunk';
import configureStore from 'redux-mock-store';
const middlewares = [thunk];
const createMockStore = configureStore(middlewares);
```

最后得到的是一个 createMockStore 函数，在测试用例中我们会用这个函数而不是 redux 提供的 createStore 函数，注意 createMockStore 可以使用 Redux 中间件，添加了 redux-thunk 之后可以处理异步 action 对象。

fetchWeather 函数中会调用 fetch 函数，这个函数的行为是去访问指定的 URL 来获取资源。单元测试应该独立而且稳定，当然不应该在单元测试中访问网络资源，所以需要“篡改”fetch 函数的行为，感谢 sinon，这样的篡改工作非常简单，代码如下：

```
import {stub} from 'sinon';
describe('fetchWeather', () => {
  let stubbedFetch;
  beforeEach(() => {
```

```
    stubbedFetch = stub(global, 'fetch');
  });
  afterEach(() => {
    stubbedFetch.restore();
  });
});
```

通过 sinon 提供的 stub 函数来“篡改”函数行为，stub 第一个参数是一个对象，第二个参数是这个函数的字符串名，返回一个 stub 对象，通过这个 stub 上的对象可以指定被“篡改”函数的行为。通过 stub 函数实际上可以“篡改”任何一个函数的行为，对 fetch 这样的全局函数也不例外，因为全局函数相当于在 global 对象上的一个函数。

需要注意的是，每一个单元测试都应该把环境清理干净。所以对一个测试套件惯常的做法是在 beforeEach 中创造 stub 对象，在 afterEach 函数中用 stub 对象的 restore 方法恢复被“篡改”函数原本的行为。

fetchWeather 函数的测试用例代码如下：

```
it('should dispatch fetchWeatherSuccess action type on fetch success', () => {
  const mockResponse= Promise.resolve({
    status: 200,
    json: () => Promise.resolve({
      weatherinfo: {}
    })
  });
  stubbedFetch.returns(mockResponse);

  return store.dispatch(actions.fetchWeather(1)).then(() => {
    const dispatchedActions = store.getActions();
    expect(dispatchedActions.length).toBe(2);
    expect(dispatchedActions[0].type).toBe(actionTypes.FETCH_STARTED);
    expect(dispatchedActions[1].type).toBe(actionTypes.FETCH_SUCCESS);
  });
});
```

在上面的测试用例中，利用 beforeEach 中创造的 stub 对象 stubbedFetch 规定 fetch 函数被调用时返回一个指定的 mockReposonse。这样，fetchWeather 函数中的 fetch 函数行为就完全被操纵，毕竟我们并不需要测试 fetch 函数的行为，所以只需要让 fetch 函数返回我们想要的结果就行。

fetchWeather 是一个异步 action 构造函数，测试一个涉及异步的被测函数时，就不能像测试普通函数一样预期被测函数执行结束就可以验证结果了。

在上面的例子中，虽然 mockResponse 是通过 Promise.resolve 函数产生的“创造即已经完结的”Promise 对象，但是其 then 指定的函数依然要等到 Node.js 的下一个时钟周

期才执行，所以也不能在 fetchWeather 函数执行完之后就认为异步操作就已经完结。

在 Jest 中测试异步函数有两种方法，一种是代表测试用例的函数增加一个参数，习惯上这个参数叫做 done，Jest 发现这个参数存在就会认为这个参数是一个回调函数，只有这个回调函数被执行才算是测试用例结束。

测试用例使用 done 参数的例子如下：

```
it('should timeout', (done) => {
});
```

在上面的例子中，这个 it 测试用例最终会因为超时而失败，因为没有任何代码去调用 done 函数。

除了使用 done 参数，还有另外一个方法，就是让测试用例函数返回一个 Promise 对象，这样也等于告诉 Jest 这个测试用例是一个异步过程，只有当返回的 Promise 对象完结的时候，这个测试用例才算结束。

在 fetchWeather 的测试用例中，就使用了返回 Promise 对象的方法。

注意，fetchWeather 的测试用例返回的并不是 store.dispatch 函数返回的那个 Promise 对象，而是经过 then 函数产生的一个新的 Promise 对象，所以当 Jest 获取的 Promise 对象被认为是完结时，在 then 函数中的所有断言语句绝对已经执行完毕了。

断言部分我们使用了 redux-mock-store 所创造 Store 的 getActions 函数，注意这个函数并不是 Redux 的功能，但能够帮助我们读取到所有派发到 Store 上的 actions，在单元测试中非常适用。

8.3.3 reducer 测试

reducer 是纯函数，所以测试非常简单，所要做的就是创造 state 和 action 对象，传递给 reducer 函数，验证结果即可。

以第 7 章中的 weather_redux/src/weather/reducer.js 为例，对应的单元测试代码放在 weather_redux/test/weather/reducer.test.js 中，代码如下：

```
it('should return loading status', () => {

  const action = actions.fetchWeatherStarted();
  const newState = reducer({}, action);
  expect(newState.status).toBe(Status.LOADING);
});
```

8.3.4 无状态 React 组件测试

对于一个无状态的 React 组件，可以使用 Enzyme 的 shallow 方法来渲染，因为 shallow

方法只渲染一层，所以不会牵涉子组件的 React 组件渲染，将单元测试专注于被测试的 React 组件本身。

以第 5 章 todo_with_selector/src/filter/views/filters.js 为例，对于这样的无状态组件，单元测试代码放在 todo_with_selector/test/filter/views/filters.test.js 中，代码如下：

```
describe('filters', () => {
  it('should render three link', () => {
    const wrapper = shallow(<Filters />);

    expect(wrapper.contains(<Link filter={FilterTypes.ALL}> {FilterTypes.ALL}
      </Link>)).toBe(true);
    expect(wrapper.contains(<Link filter={FilterTypes.COMPLETED}> {FilterTypes.
      COMPLETED} </Link>)).toBe(true);
    expect(wrapper.contains(<Link filter={FilterTypes.UNCOMPLETED}>
      {FilterTypes.UNCOMPLETED} </Link>)).toBe(true);
  });
});
```

习惯上，把 Enzyme 函数渲染的结果命名为 wrapper，对 wrapper 可以使用 contains 函数判断是否包含某个子组件。在这里，shallow 并没有渲染产生子组件 Link 的 DOM 元素，所以完全可以用 contains 来判断是否包含 Link 组件。

这种单元测试不深入渲染 React 子组件，主要的意义是可以简化测试过程，因为 React 子组件的完全渲染可能引入其他的依赖关系。

设想一下，有一个 Parent 组件，只是一个和 Redux 无关的无状态组件，Parent 包含一个 Child 组件，但是 Child 组件通过 react-redux 连接到了 Redux Store 上。那么，如果在测试 Parent 组件时要渲染 Child 组件，那就必须创造一个 Store 对象，还要用 Provider 创造一个 Context。创造这样的环境，对测试 Parent 组件本身没有任何帮助，但是如果使用 shallow 浅层渲染，只要在渲染过程中知道创造了 Child 组件，传递给 Child 的 prop 都对，这就足够了，至于 Child 功能是否正确，那就交给 Child 的单元测试去验证，不是 Parent 单元测试的责任。

8.3.5 被连接的 React 组件测试

如果将应用状态存放在 Redux Store 上，配合使用 react-redux 库，所有有状态的 React 组件都是通过 connect 函数产生的组件，被称为“被连接的组件”。

以第 5 章的 todo_with_selector/src/todos/views/todoList. js 为例，这是一个被链接的组件，在 todo_with_selector/test/todos/views/todoList.test. js 文件中可以看到全部单元测试代码。

TodoList 这样一个组件依赖于一个 Redux Store 实例，而且能够实实在在地提供内容，所以不再使用 redux-mock-store，而是使用一个货真价实的 Redux Store，需要创造一个 store，代码如下：

```
const store = createStore(
  combineReducers({
    todos: todosReducer,
    filter: filterReducer
  }), {
    todos: [],
    filter: FilterTypes.ALL
  });
```

为了将这个 Store 放在 React Context，还需要创造 Provider，使用 Enzyme 的 mount 方法渲染的是 Provider 包裹起来的 TodoList 组件，代码如下：

```
const subject = (
  <Provider store={store}>
    <TodoList />
  </Provider>
);
const wrapper = mount(subject);
```

最终，通过调用 store.dispatch 函数派发 action，然后就可以验证 wrapper 对象上的渲染元素是否发生了预期改变：

```
store.dispatch(actions.addTodo('write more test'));
expect(wrapper.find('.text').text()).toEqual('write more test');
```

这个单元测试中，我们想要模拟一次完整的 action 对象处理周期，派发出一个 addTodoaction 对象之后，应该在 DOM 树中存在对应的待办事项文字，待办事项文字存在于 TodoItem 组件中。为了检查 DOM 中的文字，就需要用 mount 函数而不是 shallow 函数。

上面创造 Provider 的过程看起来有一点麻烦，主要是因为 TodoList 组件还包含了 TodoItem 组件也是连接到 Store 的组件，如果被测试组件并不包含任何其他链接到 Store 的子组件，那就可以直接在组件渲染中用名为 store 的 prop。

例如，对于 TodoItem 组件的单元测试，就可以在 JSX 中直接这样写，不用 Provider：

```
const subject = <TodoItem store={store} {...otherProps} />;
const wrapper = mount(subject);
```

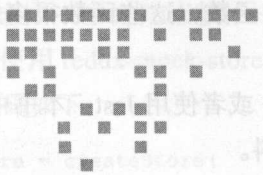
8.4 本章小结

本章介绍了单元测试的概念，说明了 React 和 Redux 应用可测试性强的原因，主要

就是因为 React 和 Redux 的功能可以分解成很多小的函数，这些函数很多还是纯函数，容易测试。

选择单元测试框架，可以使用 Mocha+Chai 组合，或者使用 Jest，本书中不一一介绍细节，读者可以通过相关工具的官方文档了解详细资料。

回顾单元测试例子，可以看出来 React 和 Redux 的单元测试往往非常直观，几乎可以认为是多余的。的确，重要的不是代码容易测试，而是程序的结构非常简单，简单到单元测试都显得没有必要的地步。



Chapter 9 第 9 章

扩展 Redux

Redux 本身提供了很强大的数据流管理功能，但是 Redux 更强大之处在于它提供了扩展自身功能的可能性。当 Redux 库中功能不满足我们要求的时候，开发者可以扩展增强 Redux 的功能。实际上，Redux 之所以获得广泛接受，就是因为社区围绕 Redux 创建了很多扩展功能，形成了一个生态系统。

在这一章中，将介绍两种扩展 Redux 的方法。

- 中间件；
- Store Enhancer。

9.1 中间件

在前面的章节中我们已经见识过中间件，现在我们来看看 Redux 中间件的具体工作机制，然后我们就要学会自己创造中间件。

如果读者接触过 Node.js 世界中最著名的服务器端框架 Express，应该对“中间件”这个名词并不陌生。在 Express 框架中，中间件是一些函数，用于定制对特定请求的处理过程。作为中间件的函数互相是独立的，可以提供对记录日志、返回特定响应报头、压缩等等请求的处理操作，中间件这种架构设计使得可以重用通用逻辑，通过组合不同中间件可以完成复杂功能。

Redux 和 Express 是两种功能不同的框架，中间件的概念也不完全一样。但是两者也有一些共同之处，如果把 Redux 和 Express 都看做一个对请求的处理框架的话，那么

Redux 中的 action 对象对应于 Express 中的客户端请求，而所有的中间件就组成处理请求的“管道”。

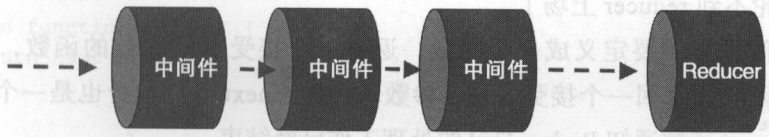


图 9-1　中间件“管道”

如图 9-1 所示，派发给 Redux Store 的 action 对象，要被 Store 上的中间件依次处理。如果把 action 和当前 state 交给 reducer 处理的过程看做是默认存在的中间件，那所有对 action 的处理都可以由中间件组成。

中间件的特点是：

- 中间件是独立的函数；
- 中间件可以组合使用；
- 中间件有一个统一的接口。

所谓中间件是独立的函数，指的是中间件之间不应该有依赖关系，每个中间件应该能够独立被一个 Redux Store 使用，完成某一个特定功能。

正因为每个中间件只完成某一个特定功能，所以为了满足比较丰富的应用需求，应该能够在一个 Redux Store 上组合使用多个中间件。这些中间件按照指定顺序依次处理传入的 action，只有排在前面的中间件完成任务之后，后面的中间件才有机会继续处理 action。当然，每个中间件都可以中断对于某个 action 的“管道”，也就是不用后面的中间件继续处理了。

不同中间件之所以能够组合使用，是因为 Redux 要求所有中间件必须提供统一的接口，每个中间件的实现逻辑都不一样，但是只有遵守统一的接口才能够和 Redux 和其他中间件对话。

我们先从 Redux 中间件的接口出发，来看看中间件如何来定制。

9.1.1　中间件接口

在 Redux 框架中，中间件处理的是 action 对象，而派发 action 对象的就是 Store 上的 dispatch 函数，之前介绍过通过 dispatch 派发的 action 对象会进入 reducer。其实这不完全正确，因为在 action 对象进入 reducer 之前，会经历中间件的管道。

在这个中间件管道中，每个中间件都会接收到 action 对象，在处理完毕之后，就会把 action 对象交给下一个中间件来处理，只有所有的中间件都处理完 action 对象之后，

才轮到 reducer 来处理 action 对象，然而，如果某个中间件觉得没有必要继续处理这个 action 对象了，就不会把 action 对象交给下一个中间件，对这个 action 对象的处理就此中止，也就轮不到 reducer 上场了。

每个中间件必须要定义成一个函数，返回一个接受 next 参数的函数，而这个接受 next 参数的函数又返回一个接受 action 参数的函数。next 参数本身也是一个函数，中间件调用这个 next 函数通知 Redux 自己的处理工作已经结束。

听起来有点绕，通过一个实例会看的更明白。

一个实际上什么事都不做的中间件代码如下：

```
function doNothingMiddleware({dispatch, getState}) {
  return function(next) {
    return function(action) {
      return next(action);
    }
  }
}
```

这个 doNothingMiddleware 接受一个对象作为参数，对象参数上有两个字段 dispatch 和 getState，分别代表的就是 Redux Store 上的两个同名函数，不过并不是所有中间件都用得上这两个函数。

函数 doNothingMiddleware 返回的函数接受一个 next 类型的参数，这个 next 是一个函数。如果调用它，代表这个中间件完成了自己的功能，把控制权交给了下一个中间件，但是这个函数还不是处理 action 对象的函数，它返回的那个以 action 为参数的函数才是。

以 action 为参数的函数对传入的 action 对象进行处理，因为 JavaScript 支持闭包（Clousure），在这个函数里可以访问上面两层函数的参数，所以可以根据需要做很多事情，包括以下功能：

- 调用 dispatch 派发出一个新 action 对象；
- 调用 getState 获得当前 Redux Store 上的状态；
- 调用 next 告诉 Redux 当前中间件工作完毕，让 Redux 调用下一个中间件；
- 访问 action 对象 action 上的所有数据。

具有上面这些功能，一个中间件足够获取 Store 上的所有信息，也具有足够能力控制数据的流转。

当然，在上面的简单例子中，doNothingMiddleware 中间件什么都没有做，只是简单地调用 next 并把 action 交给下一个中间件。

读者可能会有一个问题，为什么中间件的接口定义需要这么多层的函数呢？为什么不干脆把接口设计成一个函数接受两个参数 store 和 next，然后返回一个对 action 处理的

函数呢？如果要那样做，上面的 doNothingMiddleware 就是下面这样定义：

```
function doNothingMiddleware({dispatch, getState}, next) {

  return function(action) {
    return next(action);
  }
}
```

如果按照上面这种方法定义中间件接口，似乎更容易理解，但是 Redux 并没有这么设计，因为 Redux 是根据函数式编程（Functional Programming）的思想来设计的，函数式编程的一个重要思想就是让每个函数的功能尽量小，然后通过函数的嵌套组合来实现复杂功能，在 Redux 中我们会看到很多这样的现象。

在前面的章节中，我们使用过 redux-thunk 中间件来实现异步 action 对象，现在让我们来揭开 redux-thunk 的神秘面纱，其实 redux-thunk 极其简单，代码如下：

```
function createThunkMiddleware(extraArgument) {
  return ({ dispatch, getState }) => next => action => {
    if (typeof action === 'function') {
      return action(dispatch, getState, extraArgument);
    }
    return next(action);
  };
}

const thunk = createThunkMiddleware();
export default thunk;
```

在这里，使用了 ES6 的箭头方式表示函数，连续的 => 符号表示的是返回函数的函数，比如下面的代码写法，实际效果和上面的 doNothinigMiddleware 一样：

```
({dispatch, getState})=>next=>action=>next(action)
```

我们看 redux-thunk 这一串函数中最里层的函数，也就是实际处理每个 action 对象的函数。首先检查参数 action 的类型，如果是函数类型的话，就执行这个 action 函数，把 dispatch 和 getState 作为参数传递进去，否则就调用 next 让下一个中间件继续处理 action，这个处理过程和 redux-thunk 文档中描述的功能一致。

值得一提，每个中间件最里层处理 action 参数的函数返回值都会影响 Store 上 dispatch 函数的返回值。但是，每个中间件中这个函数返回值可能都不一样，像上面的 redux-thunk，有可能返回的是下一个中间件返回的结果，也可能返回的是 action 参数作为函数执行的结果，同时在一个应用中使用的中间件也可能发生变化。所以，dispatch 函数调用的返回结果完全是不可控的，我们在代码中最好不要依赖于 dispatch 函数的返回值。

9.1.2 使用中间件

使用中间件有两种方法，两种方法都离不开 Redux 提供的 applyMiddleware 函数。

第一种方法是用 Redux 提供的 applyMiddleware 来包装 createStore 产生一个新的创建 Store 的函数，以使用 redux-thunk 中间件为例，代码如下：

```
import {createStore, applyMiddleware} from 'redux';
import thunkMiddleware from 'redux-thunk'

const configureStore = applyMiddleware(thunkMiddleware)(createStore);
const store = configureStore(reducer, initialState);
```

为了使用 thunkMiddleware，将 thunkMiddlware 作为参数传递给 applyMiddleware，产生了一个新的函数，新产生的函数实际上是一个 Store Enhancer（很快就会介绍到 Store Enhancer，在这里只需要理解 Store Enhancer 是一中能够增强 Store 功能的函数就行），这个 Store Enhancer 函数又将 Redux 的 createStore 作为参数，产生了一个加强版的创造 store 的函数，习惯上将这个增强的 createStore 命名为 configureStore，利用 configureStore 创造的 Store 将具有 thunkMiddleware 中间件的功能。

对于没有使用其他 Store Enhancer 的场景，上述的方法还可以一用，但是实际中应用基本都会需要其他 Store Enhancer 的辅助，比如便于 Redux 开发的 Redux Devtools 增强器，在有其他增强器出现的情况下，再用这种方法就显得很不方便，所以上面介绍的第一种方法现在很少使用了，取而代之的是第二种方法。

第二种方法也就是把 applyMiddleware 的结果当做 Store Enhancer，和其他 Enhancer 混合之后作为 createStore 参数传入。

以同时使用 redux-thunk 和 Redux Devtools 增强器为例，代码如下：

```
import {createStore, applyMiddleware, compose} from 'redux';
import thunkMiddleware from 'redux-thunk'
const win = window;
const storeEnhancers = compose(
  applyMiddleware(...middlewares),
  (win && win.devToolsExtension) ? win.devToolsExtension() : f => f
);
const store = createStore(reducer, storeEnhancers);
```

可以看到，applyMiddlware 函数的执行结果，和其他的 Store Enhancer 处于同级的位置，Redux 提供了一个 compose 函数，可以把多个 Store Enhancer 串接起来成为一个函数，因为 createStore 只能接受一个 Store Enhancer 的参数。

createStore 最多可以接受三个参数，第一个参数是 reducer，第二个参数如果是对象，就会被认为是创建 Store 时的初始状态，这样第三个参数如果存在就是增强器。但如

果第二个参数是函数类型，那就认为没有初始状态，直接把第三个参数当做增强器处理。

值得一提的是，使用这种方法应用中间件的时候，一定要把 applyMiddlware 的结果作为 compose 的第一个参数，也就是要放在所有其他 Store Enhancer 之前应用。这是因为增强器的顺序也就是它们处理 action 对象的顺序，applyMiddleware 配合 redux-thunk 或者其他中间件要处理异步 action 对象。如果不是优先把他们摆在前面的话，异步 action 对象在被它们处理之前就会被其他 Store Enhancer 处理，无法理解异步对象的增强器就会出错。

在上面的例子中，如果把 applyMiddleware 放在 win.devToolsExtension() 后面的话，遇到异步对象就无法工作，在浏览器 Console 中可以看到如下错误信息：

```
Uncaught Error: Actions must be plain objects. Use custom middleware for async
    actions.
```

这就是因为 action 对象在被 redux-thunk 处理之前就被 Redux Devtools 捕获了，而 Redux Devtools 无法处理函数类型的异步 action 对象。

9.1.3 Promise 中间件

理解中间件的最好办法就是自己开发一个中间件，在上一章中提到过，替换 redux-thunk 中间件来实现异步 action 对象还有一个办法是利用 Promise，Promise 更加适合于输入输出操作，而且 fetch 函数返回的结果就是一个 Promise 对象，接下来我们就实现一个用于处理 Promise 类型 action 对象的中间件。

一个最简单的 Promise 中间件实现方式是这样：

```
function isPromise(obj) {
  return obj && typeof obj.then === 'function';
}

export default function promiseMiddleware({dispatch}) {
  return function(next) {
    return function(action) {
      return isPromise(action) ? action.then(dispatch) : next(action);
    }
  }
}
```

逻辑很简单，类似于 redux-thunk 判断 action 对象是否是函数，在这里判断 action 对象是不是一个 Promise。如果不是，那就放行通过 next 让其他中间件继续处理。如果是，那就让这个 Promise 类型的 action 对象在完成时调用 dispatch 函数把完成的结果派发出去。

这个实现很简单，但并不十分实用。

我们在第 7 章的 Weather 应用中说到过，做一个完备的异步操作，要考虑三个状态，分别是“异步操作进行中”、“异步操作成功完成”和“异步操作失败”。上面的 Promise 中间件只考虑了第二种状态“异步操作成功完成”，这是不完整的，使用这样的 Promise 中间件，并不比 redux-thunk 省事。

我们需要重新思考一下如何使用 Promise。

对于 redux-thunk，实现异步就是要在中间件层执行一个指定子程序，而 redux-thunk 就是用 action 对象是不是函数类型来判断是否调用子程序。严格来说，我们完全可以写一个中间件，通过判断 action 对象上的 async 或者什么其他字段，代码如下：

```
const thunk = ({ dispatch, getState }) => next => action => {
  if (typeof action.async === 'function') {
    return action.async(dispatch, getState);
  }

  return next(action);
};
```

然后只要约定异步 action 对象依然是普通 JavaScript 对象，只是多一个名为 async 的函数类型字段。

如果能够这样理解 action 对象，那么我们也没有必要要求 Promise 中间件处理的异步 action 对象是 Promise 对象了，只需要 action 对象某个字段是 Promise 字段就行，而 action 对象可以拥有其他字段来包含更多信息。

改进版本的 Promise 中间件是这样：

```
export default function promiseMiddleware({dispatch}) {
  return (next) => (action) => {
    const {types, promise, ...rest} = action;
    if (!isPromise(promise) || !(action.types && action.types.length === 3)) {
      return next(action);
    }

    const [PENDING, DONE, FAIL] = types;
    dispatch({...rest, type: PENDING});
    return action.promise.then(
      (result) => dispatch({...rest, result, type: DONE}),
      (error) => dispatch({...rest, error, type: FAIL})
    );
  };
}
```

这个中间件处理的异步对象必须包含一个 promise 字段和一个 types 字段，前者当然

是一个 Promise 对象，后者则必须是一个大小为 3 的数组，依次分别代表异步操作的进行中、成功结束和失败三种 action 类型。具体是什么值，应该在对应功能组件的 actionTypes.js 文件中定义：

一个被 Promise 中间件处理的异步 action 对象的例子是这样：

```
{
  promise: fetch(apiUrl),
  types: ['pending', 'success', 'failure']
}
```

如果传入的 action 对象不满足这种格式，就直接通过 next 交给其他中间件处理。

如果确定传入的 action 对象满足条件，那么 Promise 中间件就从 action 对象的 types 字段提取出三个 action 类型，第一个是表示异步操作进行中的 PENDING，不用多说，先制造一个 type 为 PENDING 的 action 对象派发出去，告诉系统这个异步动作已经开始了。

接下来，通过 then 和 catch 分别连接上 promise 字段，当这个字段代表的 Proimse 对象成功完成时，派发一个 type 为 DONE 的 action 对象，失败的时候派发一个 type 为 FAIL 的 action 对象。

在这里 PENDING、DONE 和 FAIL 都是变量，是最初异步对象中 types 字段提取出来的，由异步动作的发起者决定，所以这一个 Promise 中间件能够支持任意种类的异步请求。

这里也有一个约定，所有表示异步操作成功的 action 对象由 result 字段记录成功结果，表示异步操作失败的 action 对象用 error 字段记录失败原因。

有了上面的 Promise 中间件，可以大大简化使用异步动作的模块代码。

在第 7 章中的 Weather 应用，如果我们用这个 Promise 中间件取代 redux-thunk，那么 weather 组件的 actions.js 文件可以改写，对应 fetchWeather 的动作构造函数代码如下：

```
export const fetchWeather = (cityCode) => {
  const apiUrl = `/data/cityinfo/${cityCode}.html`;
  return {
    promise: fetch(apiUrl).then(response => {
      if (response.status !== 200) {
        throw new Error('Fail to get response with status ' + response.status);
      }
      return response.json().then(responseJson => responseJson.weatherinfo);
    }),
    types: [FETCH_STARTED, FETCH_SUCCESS, FETCH_FAILURE]
  };
}
```

原来的三个 action 构造函数 fetchWeatherStarted、fetchWeatherSuccess 和 fetchWeatherFailure 都不见了，因为他们的逻辑都被抽象出来放到 Promise 中间件中去了，在这个文件中只需要通过异步 action 对象的 types 字段把三个定义好的 action 类型设上就可以。

对比 redux-thunk 中间件和这个 Promise 中间件可以发现区别，如果应用 redux-thunk，实际发起异步操作的语句是在中间件中调用的；而如果应用 Promise 中间件，异步操作是在中间件之外引发的，因为只有异步操作发生了才会有 Promise 对象，而 Promise 中间件只是处理这个对象而已。

读者可能会发现，Promise 中间件要求“异步操作成功”和“异步操作失败”两个 action 对象必须用 result 和 error 字段记录结果，这样缺乏灵活性，的确是的。有一个改进方法是不用 types 传入 action 类型，而是用例如 actionCreators 名字的字段传入三个 action 构造函数，这样就能够更灵活地产生 action 对象，代价就是每个模块需要定义那三个 action 构造函数，读者可以尝试实现这种中间件。

9.1.4 中间件开发原则

开发一个 Redux 中间件，首先要明确这个中间件的目的，因为中间件可以组合使用，所以不要让一个中间件的内容太过臃肿，尽量让一个中间件只完成一个功能，通过中间件的组合来完成丰富的功能。

上面的例子中，都只使用了一个中间件的情况，实际中 applyMiddleware 函数可以接受任意个参数的中间件，每个通过 dispatch 函数派发的动作组件按照在 applyMiddleware 中的先后顺序传递给各个中间件，比如可以把 redux-thunk 和我们写的 Promise 中间件组合使用：

```
applyMiddleware(thunkMiddleware, promiseMiddleware)
```

这样，每个派发的 action 对象会先交给 thunkMiddleware，如果不是函数类型的 action 对象，就会顺延交给 promiseMiddelware，在这里，两个中间件所处理的 action 对象不是一个类别，所以先后顺序并不重要。

每个中间件必须是独立存在的，但是要考虑到其他中间件的存在。

所谓独立存在，指的是中间件不依赖于和其他中间件的顺序，也就是不应该要求其他中间件必须出现在它前面或者后面，否则事情会复杂化。

所谓考虑到其他中间件的存在，指的是每个中间件都要假设应用可能包含多个中间件，尊重其他件可能存在的事实。当发现传入的 action 对象不是自己感兴趣的类型，或者对 action 对象已经完成必要处理的时候，要通过调用 next（action）将 action 对象交回给

中间件管道，让下一个中间件有机会来完成自己的工作，千万不能不明不白地丢弃一个 action 对象，这样处理“管道”就断了。

对于异步动作中间件，等于是要“吞噬”掉某些类型的 action 对象，这样的 action 对象不会交还给管道。不过，中间件会异步产生新的 action 对象，这时候不能够通过 next 函数将 action 对象还给管道了，因为 next 不会让 action 被所有中间件处理，而是从当前中间件之后的“管道”位置开始被处理。

一个中间件如果产生了新的 action 对象，正确的方式是使用 dispatch 函数派发，而不是使用 next 函数。

9.2　Store Enhancer

中间件可以用来增强 Redux Store 的 dispatch 方法，但也仅限于 dispatch 方法，也就是从 dispatch 函数调用到 action 对象被 reducer 处理这个过程中的操作，如果想要对 Redux Store 进行更深层次的增强定制，就需要使用 Store Enhancer。

在前面的章节我们看到，applyMiddleware 函数的返回值被作为增强器传入 createStore，所以其实中间件功能就是利用增强器来实现的，毕竟定制 dispatch 只是 Store Enhancer 所能带来的改进可能之一，利用 Store Enhancer 可以增强 Redux Store 的各个方面。

9.2.1　增强器接口

Redux 提供的创建 Store 的函数叫 createStore，这个函数除了可以接受 reducer 和初始状态（preloadedState）参数，还可以接受一个 Store Enhancer 作为参数，Store Enhancer 是一个函数，这个函数接受一个 createStore 模样的函数为参数，返回一个新的 createStore 函数。

所以，一个什么都不做的 Store Enhancer 长得这个样子，代码如下：

```
const doNothingEnhancer = (createStore) => (reducer, preloadedState, enhancer) => {
  const store = createStore(reducer, preloadedState, enhancer);
  return store;
};
```

上面的例子中，最里层的函数体可以拿到一个 createStore 函数对象，还有应该传递给这个 createStore 函数对象的三个参数。所以，基本的套路就是就是利用所给的参数创造出一个 store 对象，然后定制 store 对象，最后把 store 对象返回去就可以。


注意 在实际执行过程中，这里的 createStore 参数未必是 Redux 默认的 createStore 函数，因为多个增强器也可以组合使用，所以这里接收到的 createStore 参数可能是已经被另一个增强器改造过的函数。当然，每个增强器都应该互相独立，只要遵循统一接口，各个增强器可以完全不管其他增强器的存在。

实现一个 Store Enhancer，功夫全在于如何定制产生的 store 对象。

一个 store 对象中包含下列接口：

- dispatch
- subscribe
- getState
- replaceReducer

每一个接口都可以被修改，当然，无论如何修改，最后往往还是要调用原有对应的函数。

例如，如果我们想要增强器给每个 dispatch 函数的调用都输出一个日志，那么就实现一个 logEnhancer，代码如下：

```
const logEnhancer = (createStore) => (reducer, preloadedState, enhancer) => {
  const store = createStore(reducer, preloadedState, enhancer);

  const originalDispatch = store.dispatch;
  store.dispatch = (action) => {
    console.log('dispatch action: ', action);
    originalDispatch(action);
  }

  return store;
};
```

增强器通常都使用这样的模式，将 store 上某个函数的引用存下来，给这个函数一个新的实现，但是在完成增强功能之后，还是要调用原有的函数，保持原有的功能。在上例中，我们想要增强 dispatch 函数，先把 store.dispatch 的实现存在 originalDispatch 变量中，这样在新的 dispatch 函数中就可以调用，保证不破坏 Redux Store 的默认功能了。

增强器还可以给 Store 对象增加新的函数，下面我们就举一个这样的例子。

9.2.2 增强器实例 reset

在实际的 Redux 应用中，有这样一个场景，一个页面已经完成了初始化，Redux Store 已经创建完毕，甚至用户都已经做了一些交互动作。这时候页面要在不刷新的情况

下切入另一个界面，这个界面拥有全新的功能模块，也就是说有全新的 reducer、action 构造函数和 React 组件。但是我们还想保持原有的 Store 对象，因为网页并没有刷新，一个应用本来就应该只有一个 Store 对象。

对于 action 构造函数和 React 组件好说，按照我们前面模块化的思想，React 组件尽量不存储状态，状态都存在 Redux Store 上，action 构造函数也直接替换就行，所以这个替换功能模块的问题根本上就是替换 reducer 和 Redux Store 上状态的问题。

Redux 的 store 上有一个 replaceReducer 函数，可以帮助我们完成替换 reducer 的工作，但是 store 只有 getState，可并没有 replaceState。Redux 这样设计就是杜绝对 state 的直接修改，我们都知道改变一个 Redux Store 状态的只能通过 reducer，而驱动 reducer 就要派发一个 action 对象。

一个应用可能包含很多功能模块，即使每个模块都提供了修改属于自己 store 状态的 action 对象，那么这个替换过程也要给所有功能模块派发一个 action 对象，这样既复杂又不通用，无法实现。

所以，我们创造一个增强器，给创造出来的 store 对象一个新的函数 reset，通过这一个函数就完成替换 reducer 和状态的功能，代码如下：

```
const RESET_ACTION_TYPE = '@@RESET';

const resetReducerCreator = (reducer, resetState) => (state, action) => {
  if (action.type === RESET_ACTION_TYPE) {
    return resetState;
  } else {
    return reducer(state, action);
  }
};

const reset = (createStore) => (reducer, preloadedState, enhancer) => {
  const store = createStore(reducer, preloadedState, enhancer);

  const reset = (resetReducer, resetState) => {
    const newReducer = resetReducerCreator(resetReducer, resetState);
    store.replaceReducer(newReducer);
    store.dispatch({type: RESET_ACTION_TYPE, state: resetState});
  };

  return {
    ...store,
    reset
  };
};

export default reset;
```

即使是 Store Enhancer，也无法打破 Redux 内在的一些限制，比如对 state，增强器也不可能直接去修改 state 的值，依然只能通过派发一个 action 对象去完成。在这里定义一个特殊的 action 类型 RESET_ACTION_TYPE，表示要重置整个 store 上的值，为了能够处理这个 action 类型，我们也要在 reducer 上做一些定制。

在 reset 函数中，先通过 replaceReducer 函数替换 store 原有的 reducer，然后通过 store 的 dispatch 函数派发一个 type 为 RESET_ACTION_TYPE 的 action 对象。这个 action 对象可没有预期让任何应用的功能模块捕获到，完全是这个增强器自己消耗。

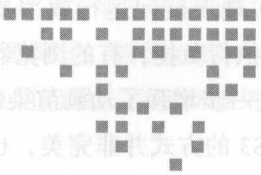
通过 replaceReducer 替换的新 reducer 在所有功能模块 reducer 之前被执行，所有通过了中间件管道的 action 对象先被 resetReducerCreator 函数的返回函数处理。在这个函数中，如果发现 action 对象的 type 为 RESET_ACTION_TYPE，那就直接返回 resetState 作为整个 Store 的新状态，其他的 action 对象则交给其他 reducer 来处理。

9.3 本章小结

在这一章中，我们了解了扩展 Redux 功能的两种方法：中间件和 Store Enhancer。

中间件用于扩展 dispatch 函数的功能，多个中间件实际构成了一个处理 action 对象的管道，action 对象被这个管道中所有中间件依次处理过之后，才有机会被 reducer 处理。在这个过程里中间件就可以扩展对 action 对象处理的功能，比如异步 action 对象，我们利用 Promise 的特性实现了一个新的处理异步 action 对象的中间件。

Store Enhancer 的功能比中间件更加强大，它可以完全定制一个 store 对象的所有接口，我们通过一个增强器实例演示了重置 store 上 reducer 和状态的功能。



第 10 章 Chapter 10

动　画

在面向用户的网页应用中，动画是很重要的一个元素，动画的作用往往并不是应用的核心功能，但是恰当应用动画，会大大提高用户体验。

在这一章中，我们将介绍以下功能。

- 在网页中动画的实现方式；
- React 提供的动画辅助工具 ReactCSSTransitionGroup；
- React-Motion 动画库。

10.1　动画的实现方式

在网页中，实现动画无外乎两种方式。

- CSS3 方式，也就是利用浏览器对 CSS3 的原生支持实现动画；
- 脚本方式，通过间隔一段时间用 JavaScript 来修改页面元素样式来实现动画。

接下来我们就分别介绍这两种方式的原理，让大家先对这两种方式有一个直观认识，了解各自的优缺点，然后我们再分别介绍在 React 中应用这两种动画方式的技巧。

10.1.1　CSS3 方式

CSS3 的方式下，开发者一般在 CSS 中定义一些包含 CSS3 transition 语法的规则。在某些特定情况下，让这些规则发生作用，于是浏览器就会将这些规则应用于指定的 DOM 元素上，产生动画的效果。

这种方式毫无疑问运行效率要比脚本方式高，因为浏览器原生支持，省去了 JavaScript 的解释执行负担，有的浏览器（比如 Chrome 浏览器）甚至还可以充分利用 GPU 加速的优势，进一步增强了动画渲染的性能。

不过 CSS3 的方式并非完美，也有不少缺点。

首先，CSS3 Transition 对一个动画规则的定义是基于时间和速度曲线（Speed Curve）的规则。换句话来说，就是 CSS3 的动画过程要描述成“在什么时间范围内，以什么样的运动节奏完成动画”。

在本书的 Github 代码库 https://github.com/mocheng/react-and-redux 中的文件 chapter-10/animation_types/ css3_sample.html 文件中可以找到一个简单例子来说明 CSS3 的工作方式，CSS 代码如下：

```
.sample {
  position: absolute;
  left: 0px;
  width: 100px;
  height: 100px;
  transition-property: left;
  transition-duration: 0.5s;
  transition-timing-function: ease
}
.sample:hover {
  left: 420px;
}
```

在上面的例子中，sample 类的元素定义了这样的动画属性：“left 属性会在 0.2 秒内以 ease 速度曲线完成动画”。

transition 只定义了动画涉及的属性、时间和速度曲线，并不定义需要修改的具体值。sample 类的 left 属性默认值为 0，当鼠标移到 sample 类元素上时，left 属性就拥有新的值 420px。这时候 transition 定义的规则发生作用，让 left 属性以 ease 速度曲线在 0.2 秒的时间完成从 0 变成 420px 的转化过程，这个过程中，用户看到的就是 sample 类元素向右移动 420 个像素的动画过程。

因为 CSS3 定义动画的方式是基于时间和速度曲线，可能不利于动画的流畅，因为动画是可能会被中途打断的，在上面的例子中，鼠标移到 sample 类元素上的时候开始动画，但是在 0.2 秒的动画时间内，用户的鼠标可能会移出这个 sample 类元素，这时候 CSS3 还会以 ease 速度曲线的节奏让 sample 类元素回到原位。从用户体验角度来说，中途 sample 类元素回到原位的动作，语义上是“取消操作”的含义，但却依然以同样的时间和 ease 节奏来完成“取消操作”的动画，这并不合理。

时间和速度曲线的不合理是 CSS3 先天的属性，更让开发者头疼的就是开发 CSS3 规则的过程，尤其是对 transition-duration 时间很短的动画调试，因为 CSS3 的 transition 过程总是一闪而过，捕捉不到中间状态，只能一遍一遍用肉眼去检验动画效果，用 CSS3 做过复杂动画的开发者肯定都深有体会。

虽然 CSS3 有这样一些缺点，但是因为其无与伦比的性能，用来处理一些简单的动画还是不错的选择。

React 提供的 ReactCSSTransitionGroup 功能，使用的就是 CSS3 的方式来实现动画，在后面的章节会详细介绍。

10.1.2 脚本方式

相对于 CSS3 方式，脚本方式最大的好处就是更强的灵活度，开发者可以任意控制动画的时间长度，也可以控制每个时间点上元素渲染出来的样式，可以更容易做出丰富的动画效果。

脚本方式的缺点也很明显，动画过程通过 JavaScript 实现，不是浏览器原生支持，消耗的计算资源更多。如果处理不当，动画可能会出现卡顿滞后现象，本来使用动画是为了创造更好的用户体验，如果出现卡顿，反而对用户体验带来不好的影响。

最原始的脚本方式就是利用 setInterval 或者 setTimeout 来实现，每隔一段时间一个指定的函数被执行来修改界面的内容或者样式，从而达到动画的效果。

在本书的 Github 代码库 https://github.com/mocheng/react-and-redux 中的文件 chapter-10/animation_types/ setInterval_animation.html 文件中可以找到一个例子，JavaScript 部分代码如下：

```
var animatedElement = document.getElementById("sample");
var left = 0;
var timer;
var ANIMATION_INTERVAL = 16;

timer = setInterval(function() {
  left += 10;
  animatedElement.style.left = left + "px";
  if ( left >= 400 ) {
    clearInterval(timer);
  }
}, ANIMATION_INTERVAL);
```

在上面的例子中，有一个常量 ANIMATION_INTERVAL 定义为 16，setInterval 以这个常量为间隔，每 16 毫秒计算一次 sample 元素的 left 值，每次都根据时间推移按比例

增加 left 的值，直到 left 大于 400。

为什么要选择 16 毫秒呢？因为每秒渲染 60 帧（也叫 60fps，60 Frame Per Second）会给用户带来足够流畅的视觉体验，一秒钟有 1000 毫秒，1000□60≈16，也就是说，如果我们做到每 16 毫秒去渲染一次画面，就能够达到比较流畅的动画效果。

对于简单的动画，setInterval 方式勉强能够及格，但是对于稍微复杂一些的动画，脚本方式就顶不住了，比如渲染一帧要花去超过 32 毫秒的时间，那么还用 16 毫秒一个间隔的方式肯定不行。实际上，因为一帧渲染要占用网页线程 32 毫秒，会导致 setInterval 根本无法以 16 毫秒间隔调用渲染函数，这就产生了明显的动画滞后感，原本一秒钟完成的动画现在要花两秒钟完成，所以这种原始的 setInterval 方式是肯定不适合复杂的动画的。

出现上面问题的本质原因是 setInterval 和 setTimeout 并不能保证在指定时间间隔或者延迟的情况下准时调用指定函数。所以可以换一个思路，当指定函数调用的时候，根据逝去的时间计算当前这一帧应该显示成什么样子，这样即使因为浏览器渲染主线程忙碌导致一帧渲染时间超过 16 毫秒，在后续帧渲染时至少内容不会因此滞后，即使达不倒 60fps 的效果，也能保证动画在指定时间内完成。

下面是一个这种方法实现动画的例子，首先我们实现一个 raf 函数，raf 是 request animation frame 的缩写，代码如下：

```
var lastTimeStamp = new Date().getTime();
function raf(fn) {
  var currTimeStamp = new Date().getTime();
  var delay  = Math.max(0, 16 - (currTimeStamp - lastTimeStamp));
  var handle = setTimeout(function(){
    fn(currTimeStamp);
  }, delay);
  lastTimeStamp = currTimeStamp;
  return handle;
}
```

在上面定义的 raf 中，接受的 fn 函数参数是真正的渲染过程，raf 只是协调渲染的节奏。

raf 尽量以每隔 16 毫秒的速度去调用传染的 fn 参数，如果发现上一次被调用时间和这一次被调用时间相差不足 16 毫秒，就会保持 16 毫秒一次的渲染间隔继续，如果发现两次调用时间间隔已经超出了 16 毫秒，就会在下一次时钟周期立刻调用 fn。

还是让 id 为 sample 的元素向右移动的例子，我们定义渲染每一帧的函数 render，代码如下：

```
var left = 0;
```

```
var animatedElement = document.getElementById("sample");
var startTimestamp = new Date().getTime();
function render(timestamp) {
  left += (timestamp - startTimestamp) / 16;
  animatedElement.style.left = left + 'px';
  if (left < 400) {
    raf(render);
  }
}
```

上面的 render 函数中根据当前时间和开始动画的时间差来计算 sample 元素的 left 属性，这样无论 render 函数何时被调用，总能够渲染出正确的结果。

最后，我们将 render 作为参数传递给 raf，启动了动画过程：

```
raf(render);
```

上面的例子在本书的 Github 代码库 https://github.com/mocheng/react-and-redux 的文件 chapter-10/animation_types/ simulate_requestAnimationFrame.html 文件中可以找到。

实际上，现代浏览器提供了一个新的函数 requestAnimationFrame，采用的就是上面描述的思路，不是以固定 16 毫秒间隔的时间去调用渲染过程，而是让脚本通过 requestAnimationFrame 传入一个回调函数，表示想要渲染一帧画面，浏览器会决定在合适的时间来调用给定的回调函数，而回调函数的工作是要根据逝去的时间来决定将界面渲染成什么样子。

这样一来，渲染动画的方式就改成按需要来渲染，而不是每隔 16 毫秒渲染固定的帧内容。

不是所有浏览器都支持 requestAnimationFrame，对于不支持这个函数的浏览器，可以使用上面 raf 函数的方式模拟 requestAnimationFrame 的行为。

在后面介绍的 react-motion 库中，使用的就是 requestAnimationFrame 这样的方式。

10.2 ReactCSSTransitionGroup

React 提供了一个叫做 ReactCSSTransitionGroup 的功能帮助实现动画，为了使用这个功能，首先要通过 npm 安装 react-addons-css-transition-group 这个库，之后就可以导入这个库的内容：

```
import TransitionGroup from 'react-addons-css-transition-group';
```

在 JavaScript 中，可以把 react-addons-css-transition-group 库导入成任何变量名，因为这个库以默认方式导出 React 组件，这里我们将其命名为 TransitionGroup。同时，为

了文字简略起见，在本章中我们将 ReactCSSTransitionGruop 简称为 TransitionGroup。

TransitionGroup 借助 CSS3 的功能实现动画，读者可能就会问，既然是用 CSS3 来实现动画，那还需要 JavaScript 代码来操作这样一个 TransitionGruop 干吗？

对于动画而言，单纯的 CSS3 就够了。但是，在 React 中使用 CSS3 来实现动画，不得不考虑一下 React 中组件的生命周期。每个 React 都会经历装载过程、更新过程和卸载过程。对于更新过程，要实现动画就是改变组件渲染内容中的样式，可以完全由 CSS3 实现，不在 TransitinGroup 能够帮忙的范围之内。不过，对于装载过程和卸载过程就不那么简单了。

假如我们希望 React 组件在装载过程中展示动画，一个问题就是，我们什么时候去修改元素的 CSS 样式从而触发动画？必须是在组件产生的 DOM 元素已经渲染在 DOM 树上之后，大约就是在 compomentDidMount 生命周期函数执行的时候，如果让每个需要动画的组件都要去实现自己的 componentDidMount，那就是重复工作，而是用 TransitionGroup 就可以轻松解决这个问题。

对于装载过程的处理还算简单，对于卸载过程的处理就更复杂了，假如我们希望 React 组件在卸载过程中展示动画，也就是希望一个组件从 DOM 树上删除时不要瞬间一下子消失，而是以一个渐进动画的方式慢慢消失，但是卸载就是把组件的 DOM 元素删掉，我们的组件又怎么能够让组件元素被删掉之前还能展示动画呢？

TransitionGroup 的工作就是帮助组件实现装载过程和卸载过程的动画，而对于更新过程，并不是 TransitionGroup 要解决的问题。

10.2.1 Todo 应用动画

为了展示 TransitionGroup 的用法，我们来增强前面章节创造的 Todo 应用，当用户添加一个待办事项时，这个待办事项以淡入的动画方式显示，当用户删除一个待办事项时，以淡出的动画方式显示。

增加动画效果，需要修改的文件只有 src/todos/views/todoList.js 文件，相关代码可以在本书的 Github 代码库 https://github.com/mocheng/react-and-redux/ 的目录 chapter-10/todo_animated 下找到。

其中作为傻瓜组件的 TodoList 组件的代码如下：

```
const TodoList = ({todos}) => {
  return (
    <ul>
      <TransitionGroup transitionName="fade" transitionEnterTimeout={500}
        transitionLeaveTimeout={200}>
```

```
        {
          todos.map((item) => (
            <TodoItem
              key={item.id}
              id={item.id}
              text={item.text}
              completed={item.completed}
            />
          ))
        }
        </TransitionGroup>
      </ul>
    );
};
```

这个 TodoList 组件和前面章节版本的区别就是使用了 TransitionGroup，TodoItem 组件的数组被当做了 TransitionGroup 的子组件。

TransitionGroup 中使用了几个 prop，其中 transitionName 是所有动画相关 prop 的核心，因为它的值关联了 TransitionGroup 和 CSS 规则。在这里，transitionName 的值为字符串 fade，代表这个 TransitionGroup 相关的 CSS 类都要以 fade 为前缀。

在 TodoList 所在的文件中，还要导入相关动画 CSS 规则，代码如下：

```
import './todoItem.css';
```

导入的 CSS 规则定义在同目录的 todoItem.css 文件中，内容如下：

```
.fade-enter{
  opacity: 0.01;
}

.fade-enter.fade-enter-active {
  opacity: 1;
  transition: opacity 500ms ease-in;
}
.fade-leave {
  opacity: 1;
}

.fade-leave.fade-leave-active {
  opacity: 0.01;
  transition: opacity 200ms ease-in;
}
```

可以看到，相关 CSS 规则中所有的类都是以 fade 为前缀，和 transitionName 的值一致，然后每个类名都由 enter、leave、active 这些单词构成，含义在下一节中会详细介绍。

现在，用浏览器访问改造过的 Todo 应用，可以看到增加删除待办事项会有动画的效果。

10.2.2 ReactCSSTransitionGroup 规则

从上面 Todo 的应用不难看出，TransitionGroup 并不能代替 CSS。恰恰相反，它离不开 CSS，其扮演的角色是让 React 组件在生命周期的特定阶段使用不同的 CSS 规则，而连接 React 组件和 CSS 需要遵守一些规则。

1. 类名规则

配合 TransitionGroup 中的 transitionName 属性，对应的 CSS 规则中类名遵从统一的规则。再类名由 - 符号把几个单词连接起来，除了 transitionName 的值，还可以有这几个单词：enter 代表“装载”开始时的状态，leave 代表“卸载”开始时的状态，active 代表动画结束时的状态。

假设 transitionName 为 sample，那么定制相关 React 组件的类名就是：

- sample-enter
- sample-enter-active
- sample-leave
- sample-leave-active

其中 active 后缀类名的作用比较特殊，因为用 CSS3 的 transition 功能实现动画，必须定义“开始状态”和“结束状态”，只有存在这两个状态，CSS3 才知道如何将元素属性从“开始状态”在指定的时间按照指定的速度曲线转化为“结束状态”，这两个状态必须定义在两个不同的 CSS 类中，否则 CSS3 无法区分。例如，首先给一个元素名为 sample-enter 的 CSS 类，然后再给它一个名为 sample-enter-active 的 CSS 类，因为这两个类分两次赋予这个元素，CSS3 就能够发现两者的差别，以 transition 的方式动画显示这个转化过程，这也就是为什么对于 sample-enter 需要一个 sample-enter-active 类，而 sample-leave 需要一个 sample-leave-active 的原因，-active 后缀的类代表的就是动画结束的状态。

在 Todo 应用的例子中，当 TransitionGroup 的任何一个子组件（在这个例子中就是 TodoItem 组件）被装载时，这个组件会被加上两个类名，fade-enter 和 fade-enter-active。这两个类并不是同时加上去的，React 会先让 TodoItem 先具有 fade-enter 类，然后在 JavaScript 下一个时钟周期才加上 fade-enter-active 类，分两次进行，这样才会产生从 fade-enter 到 fade-enter-active 描述的样式的转化过程，结果就是让 TodoItem 组件在 500 毫秒内以 ease-in 的速度曲线将不透明度从 0.01 变成 1，这就是一个淡入的效果。

当 TransitionGroup 的某个子组件被删除卸载时，这个组件会被先后加上两个类名，fade-leave 和 fade-leave-active，这两个类的会让 TodoItem 组件在 200 毫秒内以 ease-in 的速度曲线将不透明度从 1 变成 0.01，然后才彻底删除，这就是一个淡出的效果。

2. 动画时间长度

使用 TransitionGroup，动画持续的时间在两个地方都要指定，第一个是在 TransitionGroup 中以 Timeout 为结尾的属性，比如 transitionEnterTimeout 和 transitionLeave-Timeout，第二个地方是在 CSS 文件中的 transition-duration 规则。

一般来说，这两处地方的时间应该是一致的，不过我们先搞清楚为什么要分两处来指定动画时间。

以 Todo 应用中的 enter 过程为例，transitionEnterTimeout 值为 500 是告诉 TransitionGroup 每个新加入的 TodoItem 组件的动画时间持续 500 毫秒，那样 TransitionGroup 会在这个动画开始之初给 TodoItem 组件加上 fade-enter 和 fade-enter-active 类。500 毫秒之后，就会把这两个类删除掉，然后就是 TodoItem 正常的显示方式。

再看 CSS 中的规则，它只管在 500 毫秒内以指定节奏完成动画过程，因为真正的动画的时间是由 CSS 规则控制的。

假如两处的设定不一致，比如 transitionEnterTimeout 的值为 250，CSS 规则中依然是 500ms，那么 CSS 依然会以 500 毫秒的时间节奏将 TodoItem 的不透明度从 0.01 变成 1，但是，在进行到 250 毫秒的时候，transitionEnterTimeout 的设定就会让 .fade-enter 和 .fade-enter-active 类被删掉，这时候 TodoItem 组件会一下子进入普通显示状态，也就是不透明度为 1 的状态，这样就会感觉到不透明度有一个明显的跳跃，而不是平滑过渡。

一般来说，没有用户体验会希望有这样的“突变”效果，所以，通常两处的设定必须一致。

当然，同一个值需要在两处设定，这明显是重复代码，这是 TransitionGroup 的一个缺点。

3. 装载时机

读者可能会有一个疑问，为什么用 TransitionGroup 在 todoList.js 文件中包住所有 TodoItem 组件实例的数组，而不是让 TransitionGroup 在 todoItem.js 文件中包住单个 TodoItem 组件呢？

看起来应该能实现同样效果，但实际上这样做不行。因为 TransitionGroup 要发挥作用，必须自身已经完成装载了。这很好理解，TransitionGroup 也只是一个 React 组件，功能只有在被装载之后才能发挥，它自己都没有被装载，怎么可能发挥效力呢？

假如在 todoItem.js 中使用 TransitionGroup，也就是说把 TransitionGroup 当成了 TodoItem 组件的子组件，那么只有 TodoItem 完成装载时才被装载。如此一来，当一个 TodoItem 进入装载过程的时候，它内部的所有子组件还没有装载呢，包括 TransitionGroup，这样根本就不会有动画效果。

当我们要给一个数量变化的组件集体做动画的时候，TransitionGroup 总是要包住这整个结合，就像 TodoList 上包住一个 map 函数产生的 TodoItem 组件实例数组一样，这也就是为什么 TransitionGroup 命名中带一个 Group 的原因。

4. 首次装载

有一个有趣的现象，在 Todo 应用中，TransitionGroup 自身被装载的时候，可能已经包含了若干个 TodoItem 组件实例，但是这些 TodoItem 组件实例虽然经历了装载过程，却没有动画效果，只有在 TransitionGroup 被装载之后新加入的 TodoItem 组件才有动画效果。

似乎类 fade-enter 和 fade-enter-active 中定义的 CSS 规则对于随 TransitionGroup 一同装载的 TodoItem 组件实例无效。这是因为 enter 过程并不包括 TransitionGroup 的首次装载，顾名思义，enter 就是“进入”TransitionGroup，TransitionGroup 实例装载完成之后，新加入的 TodoItem 组件算是“进入”，但是随 TransitionGroup 实例一起装载的 TodoItem 组件不算“进入”。

如果我们就想让随 TransitionGroup 实例一起装载的子组件也有动画呢？那就要使用 appear 过程，appear 过程代表的就是随 TransitionGroup 一起“出现”的过程。

TransitionGroup 的 appear 也有对应的 transitionAppearTimout 属性，对应的 CSS 类符合一样的模式，以 -appear 和 -appear-active 为结尾，对应 Todo 应用中的例子，就是 fade-appear 和 fade-apppear-active 类。

不过，appear 还是有一点特殊，因为通常对于首次装载没有必要有动画，所以默认控制动画的开关是关闭的。所有的动画过程，包括 active、leave 和 appear 都有对应的 TransitionGroup 动画开关属性，分别是 transitionEnter、tranisitionLeave 和 transitionAppear，这三个属性是布尔属性，前两个默认是 true，但第三个 transitionAppear 的值默认是 false。

为了启用这个过程的动画，我们要在 TransitioGroup 的属性中显示地设定 transitionAppear 的值为 true，代码如下：

```
<TransitionGroup transitionName="fade" transitionEnterTimeout={500} transitionLeave
  Timeout={200}
transitionAppear={true}
```

```
transitionAppearTimeout={500}>
```

在 todoItem.css 中，我们可以让 appear 和 enter 具有一样的 CSS 规则：

```
.fade-enter,
.fade-appear {
  opacity: 0.01;
}

.fade-enter.fade-enter-active,
.fade-appear.fade-appear-active{
  opacity: 1;
  transition: opacity 500ms ease-in;
}
```

在浏览器中刷新 Todo 应用网页，可以看到这时候现成的三个待办事项也是以渐入的方式出现的。当然，每次切换过滤器，TodoItem 组件都以动画方式出现，看起来并不是很好的效果。

通常应用中不需要使用 appear 过程，这也就是为什么 transitionAppear 属性默认是 false 的原因。

10.3 React-Motion 动画库

在前一节我们了解了 React 官方提供的 TransitonGroup 动画库，是典型的 CSS3 方式，现在我们来了解另一个动画库 react-motion。react-motion 是很优秀的动画库，它采用的动画方式和 TransitionGroup 不同，是用脚本的方式。

10.3.1 React-Motion 的设计原则

要了解 react-motion，先要了解和 CSS3 方式不同的两个观点。

首先，大部分情况下，友好的 API 比性能更重要。这并不是说性能不重要，而是在大部分情况下，性能并不会因为采用了脚本方式不用 CSS3 方式而引起显著的性能下降。这时候，只要性能足够好，一个友好的 API 会比难以调试的 CSS3 更让开发者愿意接受，也更利于提高开发效率。

其次，不要以时间和速度曲线来定义动画过程，缺点在前面的章节已经介绍过，react-motion 提出用另外两个参数来定义动画，一个是刚度（stiffness），另一个是阻尼（damping），当然，为了方便开发者使用，react-motion 提供了一组预设的这两个参数的组合。

在第 6 章我们介绍过“以函数为子组件”的模式，在 react-motion 中就大量使用了这样的模式，react-motion 提供的组件，都预期接受一个函数作为子组件。

我们来看一个例子，这个例子显示一个倒计时，从 100 倒数为 0，可以看得出来，模式和第 6 章中我们创造的倒计时组件 CountDown 如出一辙，代码如下：

```
<Motion
  defaultStyle={{x:100}}
  style={
    {x: spring(0, {stiffness: 100, damping: 100})}
  }>
  {value => <div>{Math.ceil(value.x)}</div>}
</Motion>
```

Motion 组件的 defaultStyle 属性指定了一个初始值，style 属性指定了目标值，然后就通过刚性参数 100 和阻尼 100 的节奏，把初始值变为目标值，期间不断调用作为子组件的函数，完成动画的过程。

在第 6 章的 CountDown 组件中，我们使用的是 setInterval，我们已经知道 setInterval 方法并不是一个很好的选择。在 Motion 中，利用的是 requestAnimationFrame 函数来触发子组件函数。

很明显，Motion 其实并不直接参与动画的绘制，它只是提供参数。具体的绘制过程，由作为子组件的绘制函数来完成，很明显这种“以函数为子组件”的模式带来了很大的灵活度。

Motion 专注于提供动画的数据，子组件函数专注于绘制过程，这又是一个“每个组件只专注做一件事情”的例子。

只是有一点看起来可能有点奇怪，虽然 Motion 的两个属性都有 style 字样，但其实完全可能和 style 无关。在上面的例子中，传递到子组件的函数参数并没有用来当 style，而是直接作为内容渲染出来，这一点读者只要接受 style 在这里代表变化的参数这个概念就好了。

10.3.2 Todo 应用动画

我们在 Todo 应用中使用 react-motion 来实现添加和删除待办事项的动画，可以和上一节的 TransitionGroup 对比一下，体会两者不同。

相关代码可以在本书的 Github 代码库 https://github.com/mocheng/react-and-redux/ 的目录 chapter-10/todo_react_motion 下找到。

和前面章节中的 Todo 应用差别只有 src/todos/views/todoList.js 文件，而且这次我们不需要使用 CSS 文件，首先从 react-motion 库导入 spring 和 TransitionMotion，代码如下：

```
import {spring, TransitionMotion} from 'react-motion';
```

spring 函数用于产生动画属性的开始和结束状态，用于取代 CSS3 的 transition 方式；而 TransitionMotion 是一个组件，用于处理装载过程和卸载过程，对应于前文的 TransitionGroup，但两者使用方式有较大差别。

修改之后的 TodoList 组件的代码如下：

```
const TodoList = ({todos}) => {
  const styles = getStyles(todos);
    return (
    <TransitionMotion willLeave={willLeave} willEnter={willEnter} styles={styles}>
      {
       interpolatedStyles =>
       <ul className="todo-list">
         {
           interpolatedStyles.map(config => {
             const {data, style, key} = config;
             const item = data;
             return (<TodoItem style={style} key={key} id={item.id}
               text={item.text} completed={item.completed} />);
           })
         }
       </ul>
      }
      </TransitionMotion>
  );
};
```

TransitionGroup 应该直接包含需要动画效果的子组件，子组件可以是一组。例如，在 Todo 应用中，TransitionGroup 直接包含多个 TodoItem 组件，TransitionGroup 本身作为 ul 的子组件；但是，使用 TransitionMotion 就不可以这样，因为 TransitionMotion 只能包含函数作为子组件，而且这个函数只能返回一个元素，不能是一个数组作为元素，在上面的代码中我们可以看到，TranistionMotion 不是 ul 的子组件，ul 被移到了 TranistionMotion 的子组件函数内部，用于把所有 TodoItem 包裹为一个元素返回。

TransitonMotion 要求有一个名为 styles 的数组属性，这个数组的每个元素代表一个子组件。在 Todo 应用中，styles 中每个元素包含显示一个子组件的所有样式和数据内容，这个 styles 数组的每个元素都是一个对象，包含 key 和 style 两个必有的属性，还有一个可选的 data 属性，获取 styles 的函数如下：

```
const getStyles = (todos) => {
  return todos.map(item => ({
    key: item.id.toString(),
    data: item,
    style: {
      height: spring(60),
```

```
        opacity: spring(1)
      }
    })
  }
```

其中，key 属性是 React 要求动态数量的子组件必须包含的属性，在第 5 章对 key 属性有详细介绍，在 Todo 应用中，我们将其设为待办事项的 id。不过，这个属性要求是一个字符串类型，所以我们用 toString 将 id 转为字符串。

至于 style 属性，就是子元素动画的“目标状态”，我们利用 spring 函数指定了每个 TodoItem 组件的动画状态，含义是“以默认的刚性和阻尼设定，把 height 属性变成 60，把 opacity 属性变成 1：

```
style: {
  height: spring(60),
  opacity: spring(1)
}
```

data 属性的内容和动画无关，是子组件才理解的数据，react-motion 对这个数据也并不关心，只是扮演一个搬运工的角色，把 data 传递给内层的函数。

注意，styles 属性的每个元素的所有值都可能不同，只是在 Todo 应用中，每个 TodoItem 只有 key 和 data 不同，而 style 都是相同的。

在这里我们再次感受到 styles 命名的尴尬，虽然名为 styles，却包含了 key 和 data 这样和样式无关的数据，真正和样式相关的只有 style 一个字段而已。

在 style 中定义的只是动画的“目标状态”，但是要产生画面的运动，还需要定义动画的“初始状态”，在 TransitonMotion 中定义“初始状态”的是 willEnter。

在我们的例子中，willEnter 函数代码如下：

```
const willEnter = () => {
  return {
    height: 0,
    opacity: 0
  };
};
```

TransitionMotion 延续了 TransitionGroup 的命名习惯，用 enter 代表一个新的组件“加入”到 TransitionMotion 之中，leave 代表一个子组件“离开”TransitionMotion 被卸载。

willEnter 属性的值是一个函数，当一个组件“加入”TransitionMotion 的时候，TransitionMotion 就调用这个函数，获得这个新组件的动画初始状态。在 Todo 应用中，TodoItem 的初始状态在 willEnter 函数中定义，返回的内容让 height 和 opacity 属性都为 0。于是，TransitionMotion 就让新“加入”的组件展示 height 从 0 到 60 而 opacity 从 0 到

1 的动画过程。

willLeave 属性的值同样是一个函数，返回当一个组件“离开”TransitionMotion 时的“结束状态”。注意，在“离开”的过程开始的时候，可以认为“进入”过程完毕了，这时候 TodoItem 的状态已经是 style 中定义的最终状态，所以“离开”动画过程的“初始状态”实际上就是“进入”状态的结束状态。要形成动画效果，willLeave 返回的结果就不能像 willEnter 那样只是返回纯数字的结果，而是要用 spring 明确动画的节奏，在代码中我们可以看到 willEnter 和 willLeave 两个函数返回结果的不同。

我们的例子中对应的 willLeave 函数代码如下：

```
const willLeave = () => {
  return {
    height: spring(0),
    opacity: spring(0)
  };
}
```

对比 TransitionGroup，可以看到 TransitonMoton 不需要 CSS 的协助，完全通过 JavaScript 代码就可以实现动画的效果。

读者可能会问，在 TransitionGroup 中有一个 appear 定制第一次装载过程的动画，那么在 TransitionMotion 中如何定制这个过程呢？

TransitionMotion 还支持一种属性叫 defaultStyles，可以满足类似的要求，defaultStyles 的格式类似 styles，也是一个数组，只是每个数组元素中的 style 字段代表的就是第一次加载时的样式。

读者可以尝试给 Todo 应用的 TransitonMotion 加上下面的 defaultStyles 属性值，代码如下：

```
const defaultStyles = todos.map(item => {
  return {
    key: item.id.toString(),
    data: item,
    style: {
      height: 0,
      opacity: 0
    }
  };
});
```

最终的效果就是刷新网页之后，所有预先存在的待办事项都以动画方式出现。

在这里只介绍了 React-Motion 很小的一部分功能，也就是用 TransitionMotion 管理装载和卸载 React 组件的动画功能，实际上 React-Motion 的功能远不止于此。在 React-

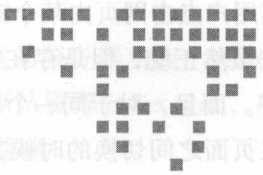
Motion 中，还有一个组件 Motion 提供普通的动画功能，一个组件 StaggeredMotion 提供固定数量组件相互依赖的动画展示功能，和 TransitionMotion 一样，React-Motion 提供的这些组件都遵循“以函数为子组件”的模式，只要掌握了“以函数为子组件”模式，就理解了全部 React-Motion 的原则。

10.4 本章小结

在这一章中，我们了解了网页动画的两种实现方式，CSS3 方式和脚本方式，在 React 的世界，也有对应这两种方式的动画解决方案。

React 官方的 ReactCSSTransitionGroup，能够帮助定制组件在装载过程和卸载过程中的动画，对于更新过程的动画，则不在 ReactCSSTransitionGroup 考虑之列，可以直接用 CSS3 来实现。

React-Motion 库提供了更强大灵活的动画实现功能，利用“以函数为子组件”的模式，React-Motion 只需要提供几个组件，这些组件通过定时向子组件提供动画参数，就可以让开发者自由定义动画的功能。



第 11 章 Chapter 11

多页面应用

现实中，应用往往都会包含很多功能，这些功能无法通过一个视觉层面上的页面展示，所以应用往往是“多页面应用”。而且，用户会在这些页面之间来回切换，开发者要做的就是保证用户的操作流畅。最好的解决方法就是虽然逻辑上是“多页面应用”，但是页面之间切换并不引起页面刷新，实际上是“单页应用”。

本章将介绍以下内容：

- 单页应用的目标；
- 实现多页面路由的 React-Router 库；
- 多页面的代码分片。

11.1 单页应用

如果使用传统的多页面实现方式，那就是每次页面切换都是一次网页的刷新，每次切换页面的时候都遵照以下步骤：

1）浏览器的地址栏发生变化指向新的 URL，于是浏览器发起一个 HTTP 请求到服务器获取页面的完整 HTML；

2）浏览器获取到 HTML 内容后，解析 HTML 内容；

3）浏览器根据解析的 HTML 内容确定还需要下载哪些其他资源，包括 JavaScript 和 CSS 资源；

4）浏览器会根据 HTML 和其他资源渲染页面内容，然后等待用户的其他操作；

然后，当用户点击网页内某个链接引起 URL 的改变，又会重复上面的步骤。

上述方法虽然正统，但是存在很大的浪费。每个页面切换都要刷新一次页面，用户体验不会很好。而且，对于同一个应用，不同页面之间往往有共同点，比如共同的顶栏和侧栏，当在页面之间切换的时候，最终结果只是局部的内容变化，却要刷新整个网页，实在是没有必要。

业界已经有很多提高多页面应用的方案，让用户能够感觉是在不同“页面”之间切换，但是实际上页面并没有刷新，只是局部更新，这种看起来多页面但是其实只有一个页面的应用，被称为“单页应用”（Single Page Application），虽然名为“单页”，但其实目的是制造视觉上的“多页”。

在本章中，我们来探讨如何用 React 和 Redux 实现单页应用。

首先我们来确定单页应用要达到的目标，如下所示：

- ❑ 不同页面之间切换不会造成网页的刷新；
- ❑ 页面内容和 URL 保持一致。

第一点好理解，是单页应用的基本要求。第二点“页面内容和 URL 保持一致”分两个方面：第一个方面是指当页面切换的时候，URL 会对应改变，这通过浏览器的 History API 可以实现在不刷新网页的情况下修改 URL；另一方面，用户在地址栏直接输入某个正确的 URL 时，网页上要显示对应的正确内容，这一点非常重要。

当用户在单页应用上浏览时，发现一个特别精彩的内容，可以把这一页的 URL 复制保存下来，回头重新打开这个 URL，看到的内容应该就是当初保存这个 URL 时的内容，而不只是这个网页应用的默认页面，这也就是所谓的“可收藏”（Bookmarkable）应用。

举个例子，假设某个应用包含两个“页面”，路径分别为 foo 和 bar，两个页面都包含指向对方的链接。当用户访问 foo 页面时，浏览器地址栏显示的 URL 地址部分为 foo。当用户点击指向 bar 页面的链接时，URL 地址部分必须转换成 bar，同时页面内容也转换为 bar 页面的内容，这时候利用浏览器的刷新按钮强行刷新页面，用户应该看到的还是 bar 页面的内容。

如果一个应用能够有上述的表现，那就是一个合格的单页应用。不过，要做到这一点，服务器必须要能够响应所有正确的 URL 请求，毕竟，只有服务器对 URL 请求回应 HTML，我们的 JavaScript 代码才有可能发挥作用。

到目前为止，本书中例子的 React 应用代码都是在浏览器中执行的，服务器返回的 HTML 实际上没有任何可视的内容，作用只是引入 JavaScript 代码，以及创建一个空的 div 作为供 React 应用大展拳脚的竞技场而已。这样一来，对于任何一个 URL，只需要返回一个同样的 HTML 页面就可以了，反正一切应用逻辑都在浏览器执行的 JavaScript 代

码中。

由 create-react-app 创建的 React 应用天生具有上述的服务器功能，当访问一个 public 目录下存在的资源时，就返回这个资源，否则都返回默认资源 index.html，所以开发者什么都不用做，就已经具备了一个支持单页应用的服务器端。

接下来，我们构建一个简单的单页应用，我们需要用上 React-Router 库。

11.2　React-Router

React-Router 库可以帮助我们创建 React 单页应用，因为要处理多个页面，所以首先要了解几个概念。

每个 URL 都包含**域名部分**和**路径（path）部分**，例如对于 URL http://localhost:3000/home 来说，路径部分是 home，因为应用可能被部署到任何一个域名上，所以决定一个 URL 显示什么内容的只有路径部分，和域名以及端口没有关系，根据路径找到对应应用内容的过程，也就是 React-Router 的重要功能——**路由**（Routing）。

本章的应用实例基于 React-Router v3.0.0 的 API，预计 React-Router v4 版的 API 会有巨大的变化，可以认为和之前的版本是完全不同的一个库，截止本书写作完稿时，React-Router v4 版还没有正式发布。不过，即使 v4 版发布之后，我们可以在 package.json 中指定 react-router 的版本为 3.0.0 来使用本章的例子。

11.2.1　路由

React-Router 库提供了两个组件来完成路由功能，一个是 Router，另一个是 Route。前者 Router 在整个应用中只需要一个实例，代表整个路由器。后者 Route 则代表每一个路径对应页面的路由规则，一个应用中应该会有多个 Route 实例。

我们所要做的单页应用很简单，只包含三个页面，第一个是代表主页的 Home，对应的路径是 home，第二个是代表说明页的 About，对应的路径是 about，当地址栏为不被支持的路径时，我们也应该提示用户资源不存在，这第三个页面就是 NotFound。

我们用 create-react-app 创造一个新的应用 react_redux_basic，在本书的 Github 代码库 https://github.com/mocheng/react-and-redux/ 的 chapter-11/react_router_basic 目录下可以找到完整代码。

在 src/pages 目录下分别创建 Home.js、About.js 和 NotFound.js，每个文件都包含一个 React 组件，内容几乎相同，只是显示的文字不同，比如 src/pages/Home.js 中定义的 Home 组件代码如下：

```
const Home = () => {
  return (
    <div>Home</div>
  );
};
```

可以看到，Home 虽然概念上是一个“页面”，但是实现上是一个 React 组件，功能上没有任何特别之处。React-Router 库认为每个页面就是一个 React 组件，当然这个组件可以包含很多子组件来构成一个复杂的页面。

当准备好三个页面之后，我们就可以开始定义路由规则了。

在 src/Routes.js 文件中，我们添加如下代码：

```
import React from 'react';
import {Router, Route, browserHistory} from 'react-router';

import Home from './pages/Home.js';
import About from './pages/About.js';
import NotFound from './pages/NotFound.js';

const history=browserHistory;
const Routes = () => (
  <Router history={browserHistory}>
    <Route path="home" component={Home} />
    <Route path="about" component={About} />
    <Route path="*" component={NotFound} />
  </Router>
);

export default Routes;
```

这个文件导出一个函数，函数返回一个 Router 组件实例，所以这个文件导出的其实也是一个 React 组件，正因为 React 组件提供了一个很好的界面功能封装，连路由功能也可以用组件形式表达。

Router 实例的 history 属性值被赋为 browserHistory，这样路由的改变就和浏览器 URL 的历史产生了关联，在后面的章节我们可以了解到 history 属性还可以是其他值。

然后 Router 实例包含三个 Route 子组件，分别完成三个路由规则，路径 home 被映射到 Home 组件，路径 about 被映射到 About 组件，一个 * 通配符代表的是所有路径，被映射到了 NotFound 组件。注意，路径为 * 通配符的这个 Route 实例必须放在最后。因为 React-Router 按照 Route 在代码中的先后顺序决定匹配的顺序，也就是首先将路径和 home 比较，如果不匹配才和 about 比较，依然不匹配才和 * 通配符比较，如果包含 * 通配符的 Route 放在了最前面，那它对任何路径都是匹配成功的，后面的 Route 规则根本

不会有机会被匹配，那样所有的路径都会被映射到页面 NotFound 上，这明显不是我们想要的结果。

最后，我们把应用的入口 src/index.js 文件修改成这样：

```
import React from 'react';
import ReactDOM from 'react-dom';
import Routes from './Routes.js';

ReactDOM.render(
  <Routes />,
  document.getElementById('root')
);
```

在 render 函数中，渲染的最顶层组件是从 src/Routes.js 中导入的 Routes 组件，这样，应用中所有的组件都居于 Router 的保护伞之下，每个组件能否被显示，都由 Router 来决定。

现在，我们通过命令行 npm start 启动这个单页应用，在浏览器中可以通过直接访问不同 URL 看到不同界面，http://localhost:3000/home 或者 http://localhost:3000/about 可以看到对应文字，如图 11-1 所示。

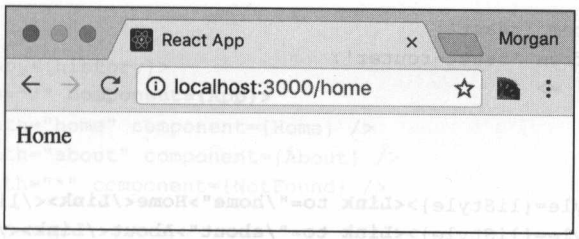


图 11-1　没有链接的 Home 页面效果

如果我们胡乱输入一个路径，比如 http://localhost:3000/noway ，看到的就是 NotFound 页面，如图 11-2 所示。

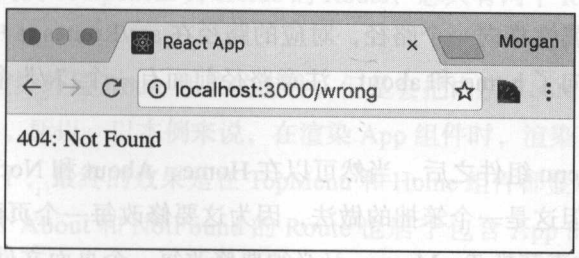


图 11-2　Not Found 页面效果

这说明 React-Router 的路由发挥了作用。然而目前这个应用很不完整，我们甚至无法验证不同页面之间切换是不是真的没有网页刷新，所以，我们需要在网页中增加一些链接。

11.2.2 路由链接和嵌套

如果在这个单页应用中增加一个顶栏，包含所有页面的链接，这样只要所有网页都包含这个顶栏，那就可以方便地在不同页面之间切换了。

不过，我们不能直接使用 HTML 的链接，因为 HTML 的链接在用户点击的时候，默认行为是网页跳转，这样并不是一个“单页应用”应有的行为。

React-Router 提供了一个名为 Link 的组件来支持路由链接，Link 的作用是产生 HTML 的链接元素，但是对这个链接元素的点击操作并不引起网页跳转，而是被 Link 截获操作，把目标路径发送给 Router 路由器，这样 Router 就知道可以让哪个 Route 下的组件显示了。

顶栏作为一个功能组件，没有复杂功能，我们在 src/TopMenu/index.js 中直接定义它的视图，代码如下：

```
import React from 'react';
import {Link} from 'react-router';
const view = () => {
  return (
    <div>
      <ul>
        <li style={liStyle}><Link to="/home">Home</Link></li>
        <li style={liStyle}><Link to="/about">About</Link></li>
      </ul>
    </div>
  );
};

export {view};
```

Link 组件的 to 属性指向一个路径，对应的路径在 src/Routes.js 中应该有定义，在这里两个 Link 分别指向了 home 和 about，注意路径前面有一个“/”符号，代表从根路径开始匹配。

有了顶栏 TopMenu 组件之后，当然可以在 Home、About 和 NotFound 组件中引用这个 TopMenu 组件，但这是一个笨拙的做法，因为这要修改每一个页面文件。如果将来我们想要用另一个组件来替换 TopMenu，又必须要修改每一个页面文件，非常不合适。

React-Router 的 Route 提供了嵌套功能，可以很方便地解决上面的问题。

在 src/pages/App.js 中我们定义个新的 React 组件 App，这个组件包含 TopMenu 组件，同时会渲染出 Home、About 或者 NotFound 页面，代码如下：

```
import React from 'react';
import {view as TopMenu} from '../components/TopMenu';

const App = ({children}) => {
  return (
    <div>
      <TopMenu />
      <div>{children}</div>
    </div>
  );
};
export default App;
```

上面的代码中看不见 Home、About 或者 NotFound，因为它们都会作为 App 的子组件，children 代表的就是 App 的子组件，所以 App 的工作其实就是给其他页面增加一个 TopMenu 组件。

在 src/Routes.js 中，我们导入 App，利用一个 Route 组件将根路径映射到 App 组件上。

```
const Routes = () => (
  <Router history={history}>
    <Route path="/" component={App}>
      <Route path="home" component={Home} />
      <Route path="about" component={About} />
      <Route path="*" component={NotFound} />
    </Route>
  </Router>
);
```

现在，假如在浏览器中访问 http://localhost:3000/home，那么 React-Router 在做路径匹配时，会根据路径“/home”的“/”前缀先找到 component 为 App 的 Route，然后根据剩下来的“home”找到 component 为 Home 的 Route，总共有两个 Route，那应该渲染哪一个呢？

React-Router 会渲染外层 Route 相关的组件，但是会把内层 Route 的组件作为 children 属性传递给外层组件。所以，以本例来说，在渲染 App 组件时，渲染 children 属性也就把 Home 组件渲染出来了，最终的效果是在 TopMenu 和 Home 组件都显示在页面上。

同样，因为包含 About 和 NotFound 的 Route 也居于包含 App 的 Route 之下，所以这两个页面上也可以看见 TopMenu 组件，如图 11-3 所示。

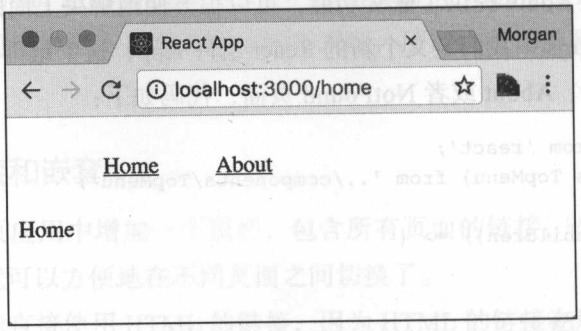


图 11-3 包含 TopMenu 的 Home 页面效果

通过点击 TopMenu 组件上的链接，用户可以在不同页面之间切换，可以注意到页面之间的切换并没有带来网页刷新。

建立 Route 组件之间的父子关系，这种方式，就是路由的嵌套。

嵌套路由的一个好处就是每一层 Route 只决定到这一层的路径，而不是整个路径，所以非常灵活，例如，我们可以修改 App 的 Route 属性 path 如下：

```
<Route path="/root" component={App}>
```

上面的 Route 规则把关于 App 的 Route 规则改成路径为“/root”，那么，访问 Home 的路径就变成了 http://localhost:3000/root/home，而关于 Home、About 和 NotFound 的 Route 却并不用做任何修改，因为每个 Route 都只匹配自己这一层的路径，当 App 已经匹配 root 部分之后，Home 只需要匹配 home 部分。

11.2.3 默认链接

到目前为止，这个应用还有一个缺陷，就是当路径部分为空时，React-Router 只匹配中了 App 组件而已，而 App 组件的角色只是一个包含 TopMenu 的壳子，直接访问 http://localhost:3000 看到的除了 TopMenu 之外什么都没有。

在路径为空的时候，应用也应该显示有意义的内容，通常对应主页内容。在这个应用中，我们希望路径为空的时候显示 Home 组件。

React-Router 提供了另外一个组件 IndexRoute，就和传统上 index.html 是一个路径目录下的默认页面一样，IndexRoute 代表一个 Route 下的默认路由，代码如下：

```
<Router history={history}>
  <Route path="/" component={App}>
    <IndexRoute component={Home} />
    <Route path="home" component={Home} />
    <Route path="about" component={About} />
```

```
    <Route path="*" component={NotFound} />
  </Route>
</Router>
```

这样一来，无论 http://localhost:3000 还是 http://localhost:3000/home 访问的都是包含 Home 的主页内容。

11.2.4　集成 Redux

为了实现路由功能，有 React-Router 就足够，和 Redux 并没有什么关系。但是我们依然希望用 Redux 来管理应用中的状态，所以要把 Redux 添加到应用中去。

首先在 src/Store.js 中添加创建 Redux Store 的代码，代码如下：

```
import {createStore, compose} from 'redux';

const reducer = f => f;
const win = window;
const storeEnhancers = compose(
  (win && win.devToolsExtension) ? win.devToolsExtension() : (f) => f,
);

const initialState = {};
export default createStore(reducer, initialState, storeEnhancers);
```

上面定义的 Store 只是一个例子，并没有添加实际的 reducer 和初始状态，主要是使用了 Redux Devtools。

使用 React-Redux 库的 Provider 组件，作为数据的提供者，Provider 必须居于接受数据的 React 组件之上。换句话说，要想让 Provider 提供的 store 能够被所有组件访问到，必须让 Provider 处于组件树结构的顶层，而 React-Router 库的 Router 组件，也有同样的需要那么，两者都希望自己处于顶端，如何处理呢?

一种方法是让 Router 成为 Provider 的子组件，例如在应用的入口函数 src/index.js 中代码修改成下面这样：

```
ReactDOM.render(
  <Provider store={store} >
    <Routes />,
  </Provider>
  document.getElementById('root')
);
```

Router 可以是 Provider 的子组件，但是，不能够让 Provider 成为 Router 的子组件，因为 Router 的子组件只能是 Route 或者 IndexRoute，否则运行时会报错。

另一个方法，是使用 Router 的 createElement 属性，通过给 createElement 传递一个

函数，可以定制创建每个 Route 的过程，这个函数第一个参数 Component 代表 Route 对应的组件，第二个参数代表传入组件的属性参数。

加上 Provider 的 createElement 可以这样定义，代码如下：

```
import store from './Store.js';
const createElement = (Component, props) => {
  return (
    <Provider store={store}>
      <Component {...props} />
    </Provider>
  );
};
const Routes = () => (
  <Router history={history} createElement={createElement}>
...
```

需要注意的是，Router 会对每个 Route 的构造都调用一遍 createElement，也就是每个组件都创造了一个 Provider 来提供数据，这样并不会产生性能问题，但如果觉得这样过于浪费的话，那就使用第一种方法。

在第 3 章我们刚刚接触 Redux 概念的时候就知道，Redux 遵从的一个重要原则就是“唯一数据源”，唯一数据源并不是说所有的数据都要存储在一个地方，而是说一个特定数据只存在一个地方，以路由为例，使用 React-Router，即使结合了 Redux，当前路由的信息也是存储在浏览器的 URL 上，而不是像其他数据一样存储在 Redux 的 Store 上，这样做并不违背“唯一数据源”的原则，获取路由信息的唯一数据源就是当前 URL。

不过，如果不是所有应用状态都存在 Store 上，就会有一个很大的缺点，就是当利用 Redux Devtools 做调试时，无法重现网页之间的切换，因为当前路由作为应用状态根本没有在 Store 状态上体现，而 Redux Devtools 操纵的只有状态。

为了克服这个缺点，我们可以利用 react-router-redux 库来同步浏览器 URL 和 Redux 的状态。显然，这违反了“唯一数据源”的规则，但是只要两者绝对保持同步，就并不会带来问题。而如果两个数据源内容不一致，那就会出大问题。

react-router-redux 库的工作原理是在 Redux Store 的状态树上 routing 字段中保存当前路由信息，因为修改 Redux 状态只能通过 reducer，所以先要修改 src/Store.js 中代码，增加 routing 字段上的规约函数 routerReducer：

```
import {routerReducer} from 'react-router-redux';

const reducer = combineReducers({
  routing: routerReducer
});
```

reducer 需要由 action 对象驱动，我们在 src/Routes.js 文件中要修改传给 Router 的 history 变量，让 history 能够协同 URL 和 Store 上的状态，代码如下：

```
import {syncHistoryWithStore} from 'react-router-redux';

const history = syncHistoryWithStore(history, store);
```

react-router-redux 库提供的 syncHistoryWithStore 方法将 React-Router 提供的 browserHistory 和 store 关联起来，当浏览器的 URL 变化的时候，会向 store 派发 action 对象，同时监听 store 的状态变化，当状态树下 routing 字段发生变化时，反过来会更新浏览器 URL。

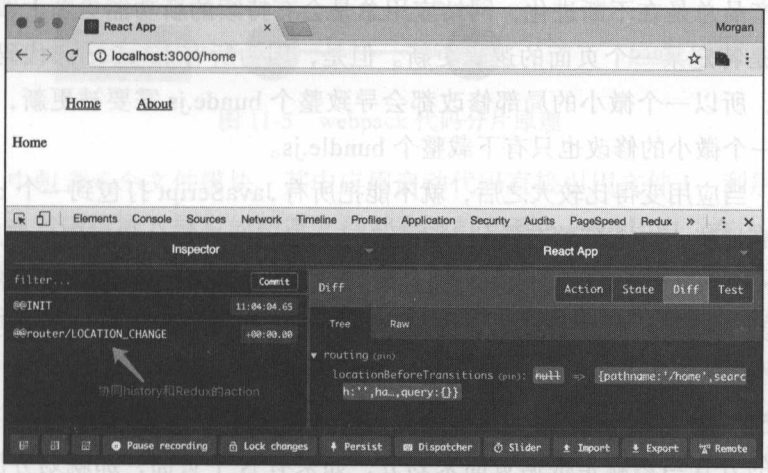


图 11-4　react-router-redux 加入的 LOCATION_CHANGE 类型 action

如图 11-4 所示，在 Redux Devtools 界面上，每当页面发生切换的时候，可以看到有一个 type 为 @@router/LOCATION_CHANGE 的 action 对象被派发出来，通过跳转到不同的 action 对象，浏览器的 URL 和界面也会对应变化。

11.3　代码分片

借助 React-Router 我们可以将需要多页面的应用构建成“单页应用”，在服务器端对任何页面请求都返回同样一个 HTML，然后由一个打包好的 JavaScript 处理所有路由等应用逻辑，在 create-react-app 创造的应用中，由 webpack 产生的唯一打包 JavaScript 文件被命名为 bundle.js。

对于小型应用，按照上面的方式就足够了，但是，对于大型应用，把所有应用逻辑打包在一个 bundle.js 文件中的做法就显得不大合适了，因为会影响用户感知的性能。

在大型应用中，因为功能很多，若把所有页面的 JavaScript 打包到一个 bundle.js 中，那么用户访问任何一个网页，都需要下载整个网站应用的功能。虽然浏览器的缓存机制可以避免下次访问时下载重复资源，但是给用户的第一印象却打了折扣，而第一印象对用户体验很重要。

例如，对于一个需要用户注册登录的网站应用，用户未登录情况下显示注册或者登录页面，但用户登录完成之后，通常很长时间都不会再访问注册登录页面，这样注册和登录页面访问量肯定很少。如果整个网站应用都只用一个 bundle.js，那么用户光是注册或者登录就要下载整个网站应用的 JavaScript，长时间的下载时间会降低用户体验。

另外，产品总是在不断进化，网站应用总是会有持续的新功能更新上线，可能某次功能更新只是特定某一个页面的逻辑更新。但是，因为所有 JavaScript 代码都打包到一个 bundle.js，所以一个微小的局部修改都会导致整个 bunde.js 需要被更新，用户浏览器为了获取那一个微小的修改也只有下载整个 bundle.js。

很明显，当应用变得比较大之后，就不能把所有 JavaScript 打包到一个 bundle.js 中。

为了提高性能，一个简单有效的方法是对 JavaScript 进行分片打包，然后按需加载。也就是把 JavaScript 转译打包到多个文件中，每一个文件的大小可以被控制得比较小。这样，访问某个网页的时候，只需要下载必需的 JavaScript 代码就行，不用下载整个应用的逻辑。

那么，按照什么原则来对代码进行分片呢？

最自然的方式当然就是根据页面来划分，如果有 N 个页面，那就划分出 N 个分片，每个页面对应一个，每个页面也只需要加载那一个分片就行。不过，现实中各个网页之间肯定有交叉的部分。比如，A 页面和 B 页面虽然不同，但是却都使用了一个共同的组件 X，而且对于 React 应用来说，每个页面都依赖于 React 库，所以至少都有共同的 React 库部分代码，这些共同的代码没有必要在各个分片里重复，需要抽取出来放在一个共享的打包文件中。

最终，理想情况下，当一个网页被加载时，它会获取一个应用本身的 bundle.js 文件，一个包含页面间共同内容的 common.js 文件，还有一个就是特定于这个页面内容的 JavaScript 文件。

为了实现代码分片，传统上需要开发者利用配置文件规定哪些代码被打包到哪个文件，这是一件费力且有可能出错的事情。感谢 webpack，有了 webpack 的帮助，实现分片非常简单。因为 webpack 的工作方式就是根据代码中的 import 语句和 require 方法确定模块之间的依赖关系，所以 webpack 完全可以发掘所有模块文件的依赖图表，从这个图表中不难归结出分片需要的信息。

图 11-5 展示了 webpack 实现代码分片的原理。

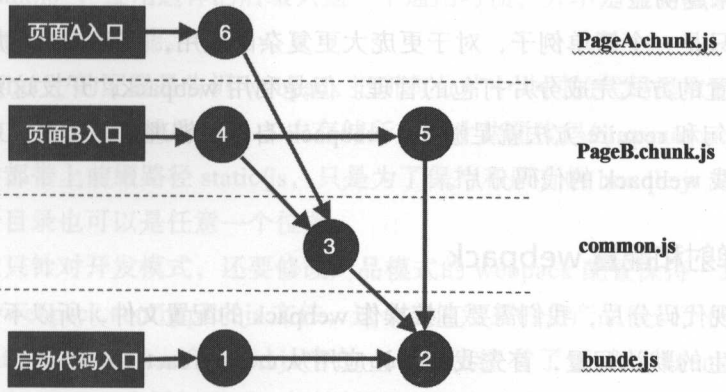


图 11-5　webpack 代码分片原理

图 11-5 中包含 6 个文件模块，其中应用启动代码直接引用文件 1，利用 import 或者 require 直接或者间接被 1 导入的所有文件都会被打包到 bundle.js 中，在上面的例子中，1 号和 2 号文件属于 bundle.js。

不过，还有的文件并不通过启动代码直接或者间接导入，比如上面例子中的 3、4、5、6 号文件。其中 6 号文件包含页面 A 的逻辑，4 号文件包含页面 B 的逻辑，webpack 会将这两个文件分别放到这两个页面各自的打包文件中。

还有 5 号文件只被页面 B 导入，页面 3 既被页面 A 导入也被页面 B 导入，这样 5 号文件就会被放在页面 B 专属的打包文件中，但是 3 号文件因为被两个页面共享，会放到一个共享的打包文件中。

其中 2 号文件既被 3 号文件也被 5 号文件导入，但是 2 号文件已经被打包到启动打包文件中，所以没有必要在其他打包文件中重复一份了。

所以，最后总共生成四个打包文件：启动代码的 bundle.js 包含 1 号和 2 号文件，所有页面共享的打包文件 common.js 包含 3 号文件，页面 A 生成的打包文件 PageA.chunk.js 包含 6 号文件，页面 B 生成的打包文件 PageB.chunk.js 包含 4 号和 5 号文件。

这样，当浏览器访问页面 A 时，只需要加载 PageA.bundle.js、common.js 和 bundle.js 三个文件，和页面 A 无关的 4 号和 5 号文件则根本不被加载，节省了代码下载量。

当然，提高网页性能的另一个重要原则是减少 HTTP 请求数，虽然代码分片减少了每个页面的代码下载量，却也增加了引用的 JavaScript 资源数，但是这只影响用户访问的第一个页面，例如，用户访问的第一个页面是 A，下载 PageA.bundle.js、common.js 和 bundle.js 三个文件，随后当页面切换到 B 时，因为浏览器的缓存作用，common.js 和

bundle.js 不用重新下载，所以新下载的文件只有 PageB.bundle.js，当应用中页面越多，这种优化效果越明显。

上面还只是一个简单例子，对于更庞大更复杂的应用，依赖关系更加复杂，肯定无法用人工配置的方式完成分片打包的管理。但是利用 webpack，开发这要做的只是管理好 import 语句和 require 方法就足够了，webpack 自然能够理清复杂的依赖关系，接下来我们就来实践 webpack 的代码分片。

11.3.1 弹射和配置 webpack

为了实现代码分片，我们需要直接操作 webpack 的配置文件，所以不能再使用 create-react-app 产生的默认配置，首先我们要让应用从 create-react-app 制造的“安全舱”里弹射出来，在命令行执行下列命令完成“弹射”：

```
npm run eject
```

“弹射”是不可逆的操作，上面的命令会要求你确认是否下了决心。当然，为了配置 webpack 我们别无选择，只能确认。

在命令行中这个命令完成的时候，会发现应用目录下多了 scripts 和 config 两个目录，分别包含脚本和配置文件，同时应用目录下的 package.json 文件也发生了变化，包含了更多的内容，至此，“弹射”完成，但是功能和“弹射”之前别无二致，要改进功能还需要手工修改一些文件。

有两个 webpack 配置，分别代表开发环境和产品环境，我们首先处理开发模式也就是 npm start 命令启动的模式下的 webpack 配置。

我们打开 config/webpack.config.dev.js，找到给 module.exports 赋值的语句，在给 module.exports 赋值的对象中，找到 output 这个字段，在其中添加上关于 chunkFilename 的一行。然后找到 plugins 字段，这是一个数组，在里面添加一个元素增加 CommonsChunkPlugin，代码修改如下：

```
module.exports = {
  ...
  output: {
    ...
    chunkFilename: 'static/js/[name].chunk.js',
    ...
  }
  plugins: [
    ...
    new webpack.optimize.CommonsChunkPlugin('common', 'static/js/common.js'),
  ]
}
```

增加的 output 配置，是告诉 webpack 给每个分片都产生一个文件，文件名包含模块名和后缀“.chunk.js”，使用这样的后缀只是一个通用习惯，并不是强制要求，如果改成“.page.js”之类也不影响功能。

增加在 plugins 中的配置是告诉 webpack 把所有分片中共同的代码提取出来，放在名为 common.js 的文件中，也就是图 11-5 中存储所有分片共同代码的 common.js。

生成的文件都带上前缀路径 static/js，只是为了保持和原有的 bundle.js 文件所在目录一致，其实这个目录也可以是任意一个位置。

上面的修改只针对开发模式，还要修改产品模式的 webpack 配置保持一致。

打开 config/webpack.config.prod.js 文件，这个文件定义的是产品模式下的 webpack 配置，不过在 config/webpack.config.prod.js 中的 output 已经有了正确的 chunkFileName 配置，所以只需要在 plugins 中添加下面一行就行：

```
new webpack.optimize.CommonsChunkPlugin('common', 'static/js/common.[chunkhash:8].js')
```

产品环境的配置和开发环境有些不同，多出了 [chunkhash:8] 的部分，这是为了让浏览器缓存在文件内容改变时失去效果。

因为产品环境下打包的文件部署出去之后预期会被浏览器长时间缓存，所以不能使用固定的文件名，否则后续部署的代码更新无法被浏览器发现。所以每个文件名都会包含一个 8 位的根据文件内容产生的哈希结果，这样当文件内容发生改变时，文件名也就发生了变化，对应文件的 URL 也就发生了变化，浏览器就会去下载最新的 JavaScript 打包资源。

11.3.2　动态加载分片

针对 webpack 的配置只是告诉 webpack 分片打包，但是 webpack 没有“页面”的概念，还是需要修改 JavaScript 代码来确定怎样按照页面分片。

继续我们的应用实例，我们希望 Home、About 和 NotFound 页面每个都是按需加载的，这三个页面都应该有自己的分片，它们的内容也就不应该包含在主体的 bundle.js 文件中。

因为 webpack 的工作方式是根据代码中的 import 语句和 require 函数来找到所有的文件模块，所以，要让这三个页面不出现在 bundle.js 文件中，我们就不能再直接使用 import 命令来导入它们。

对 ES 语法有一个提议是增加 import 函数从而实现动态的 import，注意动态 import 是函数形式，代码类似 import('./pages/Home.js')，和不带括号的静态的 import 语句不同。目前这个动态 import 的提议处于 ES 的 Stage3 阶段，本书中的例子没有使用这种语法，读者可以尝试使用动态 import 修改本书的例子取代 require.ensure。

在 src/Stores.js 中，我们首先注释或者删掉针对 Home、About 和 NotFound 的 import 语句，代码如下：

```
import App from './pages/App.js';
//import Home from './pages/Home.js';
//import About from './pages/About.js';
//import NotFound from './pages/NotFound.js';
```

然后，我们在 src/Routes.js 中要利用 Route 的 getComponent 属性异步加载 React 组件，代码如下：

```
const getHomePage = (location, callback) => {
  require.ensure([], function(require) {
    callback(null, require('./pages/Home.js').default);
  }, 'home');
};
const getAboutPage = (location, callback) => {
  require.ensure([], function(require) {
    callback(null, require('./pages/About.js').default);
  }, 'about');
};
const getNotFoundPage = (location, callback) => {
  require.ensure([], function(require) {
    callback(null, require('./pages/NotFound.js').default);
  }, '404');
};
```

Route 的 getComponent 函数有两个参数，第一个参数 nextState 代表匹配到当前 Route 的信息，不过这里我们用不上这个参数；第二个参数是一个回调函数，回调函数遵从 Node.js 的回调函数风格，第一个参数代表是否有错误，第二个参数代表装载成功的组件，正因为这个回调函数的存在，使得异步加载组件成为可能，我们会在异步加载了对应组件之后再调用这个回调函数。

在这里，异步加载模块的方法使用 require.ensure，ensure 是 require 对象的一个属性，实际是一个函数。当 webpack 在做静态代码分析时，除了特殊处理 import 和 require，也会特殊处理 require.ensure，当遇到 require.ensure 函数调用，就知道需要产生一个动态加载打包文件。

require.ensure 函数有三个参数，第一个参数是一个数组，第二个参数是一个函数，第三个参数是分片模块名。require.ensure 所做的事情就是确保第二个函数参数被调用时，第一个参数数组中所有模块都已经被装载了。对于我们的应用，页面模块没有特殊的依赖关系，所以第一个参数保持一个空数组就行，要做的只是在第二个参数中通过 require 语句来装载对应的页面文件。

至于 require.ensure 函数的第三个参数，代表的是模块名，对应的就是上面在 webpack 配置文件中添加的 chunkFilename 参数中的 [name] 值，如果第三个参数没有的话，webpack 会给每个模块分配一个数字代表的唯一 ID 作为 chunk 的名字，为了让分片文件名清晰，我们在代码中分别指定了模块名为 home、about 和 404，那么我们预期最终会有三个分片文件产生，名字分别是 home.chunk.js、about.chunk.js 和 404.chunk.js。

在 Routes 函数中使用 getComponent 而不是 componet 属性来使用异步加载页面的函数，代码如下：

```
const Routes = () => (
  <Router history={history} createElement={createElement}>
    <Route path="/" component={App}>
      <IndexRoute getComponent={getHomePage} />
      <Route path="home" getComponent={getHomePage} />
      <Route path="about" getComponent={getAboutPage} />
      <Route path="*" getComponent={getNotFoundPage} />
    </Route>
  </Router>
);
```

所以，完成动态加载分片要两个方面。

- 使用 require.ensure 让 webpack 产生分片打包文件；
- 使用 React-Router 的 getComponent 异步加载页面分片文件。

这个神奇的过程，关键在于 webpack 会对 require.ensure 函数调用做特殊处理，而且 React-Router 通过 getComponent 函数支持异步加载组件。

读者看到上面的代码可能会有一个疑问，我们给每个页面的 getComponent 属性都定义了一个函数，分别是 getHomeComponent、getAboutComponent 和 getNotFoundComponent，这三个函数中的代码几乎相同，唯一的区别就是页面组件的文件位置和模块名。出于避免重复代码的目的，为什么不把这三个函数的共同部分提取成一个函数，把差别体现的函数参数中呢？比如使用像下面这样的代码：

```
const getPageCompnent = (pagePath, chunkName) => (nextState, callback) => {
  require.ensure([], function(require) {
    callback(null, require(pagePath).default);
  }, chunkName);
};

<Route path="home" getComponent={getPageCompnent('./pages/Home.js', 'home')}
/>
```

然而，并不能这样做。因为 webpack 的打包过程是对代码静态扫描的过程，也就是说，webpack 工作的时候，我们缩写的代码并没有运行，webpack 看到的 import 和

require 的参数如果不是字符串而是一个需要运算的表达式，webpack 就无从知道表达式运算结果是什么。

如果 require 的参数是字符串，那 webpack 就可以明确知道对应的文件模块位置，一切顺利；如果 require 的参数是一个变量，那么 webpack 就无法在静态扫描状态下确定哪些文件应该放在对应分片中，分片也就失效了。

现在，我们刷新网页，访问 http://localhost:3000，在浏览器的网络工具中可以看到下载了三个文件，分别是 common.bundle.js、bundle.js 和 home.chunk.js，其中 home.chunk.js 就是特定于 Home 的分片文件，当我们通过点击顶栏的 About 链接时，可以看到只有一个新下载的文件 about.chunk.js，效果如图 11-6 所示。

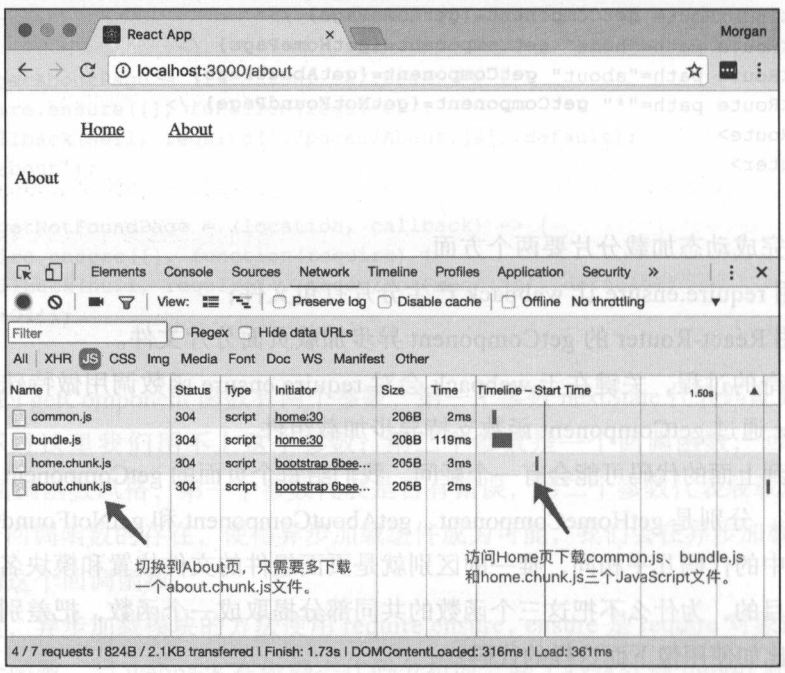


图 11-6　动态分页的效果

本节关于动态加载分片的代码，在本书 Github 代码库 https://github.com/mocheng/react-and-redux/ 的目录 chapter-11/ react_router_chunked 下可以找到。

11.3.3　动态更新 Store 的 reducer 和状态

在第 4 章中，我们学习过将应用分解为若干功能模块，每个功能模块除了包含 React 组件，还可以有自己的 reducer 和被这个 reducer 修改的 Store 上的状态。

当实现动态加载分片之后，功能模块会被 webpack 分配到不同的分片文件中，包含在功能模块中的 reducer 代码也会被分配到不同的分片文件。这样，应用的 bundle.js 文件中就没有这些 reducer 函数的定义，每个应用都有一个唯一的 Redux Store，当应用启动创建 Store 时，并不知道这个应用中所有的 reducer 函数如何定义。所以，当切换到某个页面的时候，除了要加载对应的 React 组件，还要加载对应的 reducer，否则功能模块无法正常工作。

功能模块依赖于 Store 上的状态，所以当页面切换时，除了要更新 reducer，Store 上的状态树也可能需要做对应改变，才能支持新加载的功能组件。

感谢 Redux 的结构，让我们把应用逻辑分散在视图和 reducer，而这两者并不负责状态存储，状态存储在一个全局状态树上，才使得这种动态加载变得容易。

我们在第 8 章中介绍过 Store Enhancer，并且创造 reset 增强器，可以在 store 上添加一个名为 reset 的函数，这个函数可以替换当前 store 上的状态和 reducer，现在我们就要把 reset 增强器应用到我们的例子中去。

为了演示更新 reducer 和状态的功能，我们在代码中增加一个具有 reducer 的功能模块，在第 3 章中我们实现过一个计数器功能，在这里我们稍微修改，以第 4 章功能模块的定义方式放在 src/components/Counter 目录下。

在 src/components/Counter/index.js 文件中定义功能模块的接口，代码如下：

```
import * as actions from './actions.js';
import reducer from './reducer.js';
import view, {stateKey} from './view.js';

export {actions, reducer, view, stateKey};
```

之前我们在功能模块中只导出 actions、reducer 和 view，现在多了一个 stateKey，代表的是这个功能模块需要占据的 Redux 全局状态树的子树字段名，具体值是字符串 coutner，在 view.js 中定义，因为视图中的 mapStateToProps 函数往往需要直接访问这个 stateKey 值。

在 src/components/Counter/actionTypes.js 文件中定义 action 类型对象，代码如下：

```
export const INCREMENT = 'counter/increment';
export const DECREMENT = 'counter/decrement';
```

在 src/components/Counter/acitons.js 文件中定义 action 构造函数，代码如下：

```
import * as ActionTypes from './actionTypes.js';

export const increment = () => ({
```

```
  type: ActionTypes.INCREMENT
});

export const decrement = () => ({
  type: ActionTypes.DECREMENT
});
```

在 src/componets/Counter/reducer.js 中定义 reducer，代码如下：

```
import {INCREMENT, DECREMENT} from './actionTypes.js';

export default (state = {}, action) => {
  switch (action.type) {
    case INCREMENT:
      return state + 1;
    case DECREMENT:
      return state - 1;
    default:
      return state
  }
}
```

最后，在 src/components/Counter/view.js 中定义视图，对于 Counter 无状态组件的定义，代码如下：

```
function Counter({onIncrement, onDecrement, value}) {
  return (
    <div>
      <button style={buttonStyle} onClick={onIncrement}>+</button>
      <button style={buttonStyle} onClick={onDecrement}>-</button>
      <span>Count: {value}</span>
    </div>
  );
}
```

这个 Counter 组件会直接从 Store 的 counter 字段读取当前的计数值，所以需要定义对应的 mapStateToProps 和 mapDispatchToProps 函数，代码如下：

```
export const stateKey = 'counter';

const mapStateToProps = (state) => ({
  value: state[stateKey] || 0
})

const mapDispatchToProps = (dispatch) => bindActionCreators({
  onIncrement: increment,
  onDecrement: decrement
}, dispatch);
```

```
export default connect(mapStateToProps, mapDispatchToProps)(Counter);
```

可以看到 mapStateToProps 函数需要直接访问 stateKey，所以 stateKey 虽然在 index.js 文件中导出，但通常要在视图文件中定义值。

在 src/components/ 目录下定义的都是功能模块，我们将功能模块和页面严格区分开，为了展示出 Counter 模块，我们先在 src/pages/CounterPage.js 文件中增加对应的页面定义，代码如下：

```
import {view as Counter} from '../components/Counter';

const CounterPage = () => {
  return (
    <div>
      <div>Counter</div>
      <Counter caption="any"/>
    </div>
  );
};

export default CounterPage;
```

页面中只显示了一个 caption 为 any 的计数器，除此之外和其他页面的定义没有什么两样。

最后，我们在 src/Routes.js 中增加对 CounterPage 的 Route 规则，代码如下：

```
const getCounterPage = (nextState, callback) => {
  require.ensure([], function(require) {
    callback(null, require('./pages/CounterPage.js').default);
  }, 'counter');
};

...
<Route path="home" getComponent={getHomePage} />
<Route path="counter" getComponent={getCounterPage} />
<Route path="about" getComponent={getAboutPage} />
```

最后，在 src/componets/TopMenu/index.js 顶栏中增加对 counter 路径的链接，在任何一个页面都可以通过顶栏的链接切换到 Counter 页面，代码如下：

```
<li style={liStyle}><Link to="/counter">Counter</Link></li>
```

现在，我们可以启动应用，在浏览器中可以看到顶栏上新添加了一个 Counter 链接，可以点击这个链接，显示出计数器页面，如图 11-7 所示。

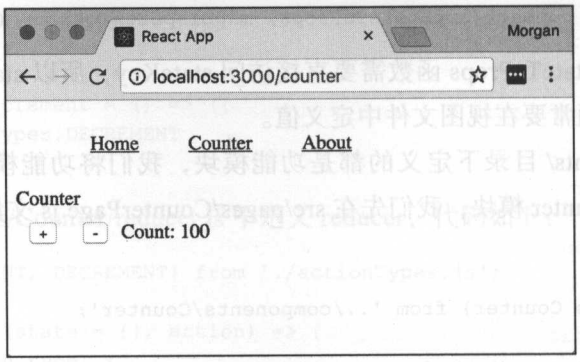


图 11-7 增加的 Counter 页面效果

不过，在计数器页面中点击“+”按钮和“-”按钮，计数并不会变化。这很正常，因为上面的代码只处理了视图，却没有处理 reducer，reducer 不会自动把自己添加到 Redux 中。所以，要想具有完整的计数器功能，我们还要继续加工代码。

在 src/pages/CounterPage.js 文件中虽然导入了 Counter 功能组件，但是只使用了 view，却没有使用 reducer。然而，在 CounterPage.js 中直接操作 Redux Store 似乎超出了它的职责范围。所以我们只是让 CounterPage.js 导出内容中增加 reducer，由使用它的模块来操作 Redux 就好，进行这个操作的模块还需要更新 Redux 的状态树，所以 CounterPage.js 还需要提供页面的初始状态 initialState，以及需要初始状态挂靠的状态树字段名 stateKey。

最终的 src/pages/CounterPage.js 是这样：

```
import {view as Counter, stateKey, reducer} from '../components/Counter';
const page = () => {
  return (
    <div>
      <div>Counter</div>
      <Counter />
    </div>
  );
};

const initialState = 100;
export {page, reducer, initialState, stateKey};
```

为了让初始计数值有别于默认的 0，我们 Counter 组件的初始计数为 100。

在 src/Store.js 中，我们引入了第 8 章中定义的名为 reset 的 Store Enhancer，这样创造出来的 store 上有一个函数 reset，可以更新 reducer 和状态，代码如下：

```
import resetEnhancer from './enhancer/reset.js';
```

```
const originalReducers = {
  routing: routerReducer
}
const reducer = combineReducers(originalReducers);

const storeEnhancers = compose(
  resetEnhancer,
  applyMiddleware(...middlewares),
  (win && win.devToolsExtension) ? win.devToolsExtension() : (f) => f,
);

const initialState = {};
const store = createStore(reducer, initialState, storeEnhancers);
store._reducers = originalReducers;
export default store;
```

在上面的例子中，store 上增加了一个 _reducers 字段，这是因为无论如何更改 Redux 的 reducer，都应该包含应用启动时的 reducer，所以我们把最初的 reducer 存储下来，在之后每次 store 的 reset 函数被调用时，都会用 combineReducers 来将启动时的 reducer 包含进来。

最后，我们在 src/Routes.js 文件中要更新一下 getCounterPage 函数的实现，代码如下：

```
import {combineReducers} from 'redux';

const getCounterPage = (nextState, callback) => {
  require.ensure([], function(require) {
    const {page, reducer, stateKey, initialState} = require('./pages/CounterPage.
      js');

    const state = store.getState();
    store.reset(combineReducers({
      ...store._reducers,
      counter: reducer
    }), {
      ...state,
      [stateKey]: initialState
    });

    callback(null, page);
  }, 'counter');
};
```

在新的 getCounterPage 函数中，从 CounterPage.js 文件中不仅获得了代表视图的 page，还有 reducer，以及代表页面组件初始状态的 initialState 和状态树字段名 stateKey。

在调用 callback 通知 Router 装载页面完成之前，要完成更新 Redux 的 reducer 和状态树的操作。因为 reset 增强器的帮助，现在 store 上有一个 reset 函数，第一个参数是新

的 reducer，第二个参数是新的状态。我们不能破坏应用初始化时的规约逻辑，也不想丢弃状态树上现有的状态，所以使用扩展操作符增量添加了新的 reducer 和状态，然后调用 reset 函数。

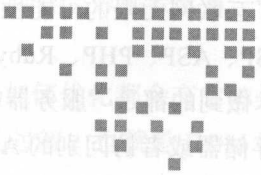
现在，在浏览器中重新切换到 Counter 页面时，计数器初始值是 100，说明我们的初始状态设置成功，点击“+”和“-”按钮能够引起计数数值的变化，说明 reducer 也更新成功，我们实现了完整的动态加载功能模块。

11.4 本章小结

本章介绍了构建多页面复杂应用的方法，借助 React-Router 库的帮助，可以将应用中的不同路径映射到 React 组件。

当应用变得庞大时，需要考虑对应用的 JavaScript 代码进行分片管理，这样用户访问某个页面的时候需要下载的 JavaScript 大小就可以控制在一个可以接受的范围内。

代码分片带来的一个问题是如何使用分片中定义的 reducer 和初始状态，借助于 Store Enhancer，我们可以在更新视图的同时，完成对 reducer 和状态树的更新。



第 12 章 Chapter 12

同　构

在前面的所有章节中，React 和 Redux 都是完全在浏览器中运行的，其实，React 作为一个产生用户界面的 JavaScript 库，Redux 作为一个管理应用数据的框架，两者也可以在服务器端运行。

理想情况下，一个 React 组件或者说功能组件既能够在浏览器端渲染也可以在服务器端渲染产生 HTML，这种方式叫做“同构”（Isomorphic），也就是同一份代码可以在不同环境下运行。

本章将介绍以下内容。

- 服务器端渲染和浏览器端渲染的对比；
- 如何构建渲染动态内容的服务器；
- 如何实现 React 和 Redux 的同构应用。

12.1　服务器端渲染 vs 浏览器端渲染

我们首先回顾一下万维网的历史，万维网技术是从服务器端渲染发展起来。最初，所有的网页都是静态网页，服务器上存储的只有一些静态 HTML 文件，浏览器通过 HTTP 协议访问到的就是这些静态 HTML。这样虽然可以建立纯文档的万维网，但是没法提供动态内容，这无法满足利用网页制作动态内容应用的需要。

随后，CGI 技术出现，服务器端终于可以产生动态内容的 HTML 了，这是一个巨大

的进步，给了互联网无限的可能性。因为需求存在，所以服务器端各种语言、各种框架层出不穷，JSP、ASP、PHP、Ruby on Rails、Python……可以列出一个超长的技术名单，所有这些技术做到的都是“服务器端渲染”，也就是对于来自浏览器的 HTTP 请求，服务器通过访问存储器或者访问别的 API 服务之类的方式获得数据，然后根据数据渲染产生 HTML 返回给浏览器，浏览器只要把 HTML 渲染出来，就是用户想要看的结果。

服务器渲染方式统治了网页应用开发很长时间，到现在也依然有很大影响力，但是，用户对网页应用的要求也越来越高，传统的服务器渲染对于每个 HTTP 请求都要产生一份全新的 HTML，这样每个 HTTP 请求都保持独立，这种方式当然有利于功能扩展，但是一个缺陷就是会有浪费。毕竟很多网页切换只需要一个局部的更新，却依然要网页重新下载刷新，体验会有一点差。而用户希望更好的体验，于是就有了 AJAX，可以在不刷新网页的情况下通过局部更新提高用户体验，随 AJAX 而来的就是 Web 2.0 的浪潮。

技术没有止步于 AJAX，最初使用 AJAX 的应用大多依然通过服务器渲染产生一个包含可渲染内容的 HTML 网页，只有局部的更新使用 AJAX 完成，于是，有的开发者已经开始思考，是不是可以干脆不用服务器端返回有内容的 HTML，是不是可以让应用功能完全用 JavaScript 在浏览器端产生 HTML 呢?

2009 年，Twitter 网站就进行过这样的尝试，服务器返回的 HTML 只包含几个无内容的元素作为页面框架，但是，网页中引用的 JavaScript 文件会直接访问 API 服务器来获取数据，获取的数据再通过模板库产生 HTML 字符串，把 HTML 字符串插入到页面框架中，就产生了用户最终看到的界面，这就是“浏览器端渲染”的方式。

反思当年 Twitter 的尝试，并不算十分成功，因为模板库的效率并不是很高，加上用户需要等待 API 请求成功之后才能看到第一条有意义内容。所以非但没有提高用户体验，反而让用户感觉更慢了，最终 Twitter 放弃了这种纯靠浏览器渲染的方式，直到今天，Twitter 网站依然是服务器端渲染配合浏览器端渲染的工作方式。

不过，整个行业并没有被这点挫折吓倒，更多的支持完全“浏览器端渲染”的方案提了出来。

传统上，一个浏览器端渲染的方案，一般要包含这几个部分：

- 一个应用框架，包含路由和应用结构功能，例如 Backbone.js 就是这样的 MVC 框架，当然 Redux 这样遵循单向数据流的框架配合 React-Router 也可以胜任；
- 一个模板库，比如 mustache，通过模板库开发者可以定义模板，模板以数据为输入，输出的就是 HTML 字符串，可以插入到网页之中，React 可以替换模板库的功能；
- 服务器端的 API 支持，因为应用代码完全部署在页面的 JavaScript 中，获取数据

不能像服务器端渲染那样有直接访问数据库的选择，只能要求有一个提供数据的 API 服务器，通常就是一个 RESTful API。

传统的模板库就是生硬的字符串替换操作，无论如何优化都会有它的极限，而且模板的输出依然是字符串，将 HTML 字符串插入网页的过程，也就是 DOM 树的操作，性能也无法优化。在前面的章节中我们介绍过 React 的 Virtual DOM 工作原理，配合生命周期函数的应用，性能不是字符串替换的模板库能够比拟的。

React 可以将网页内容（HTML）、动态行为（JavasScript）和样式（CSS）全部封装在一个组件中，把浏览器端渲染的应用发挥到了极致。

虽然完全的浏览器端渲染很有市场，但是这种方式有一个难以摆脱的阴影，那就是首页性能。

为了便于量化网页性能，我们定义两个指标：

- TTFP（Time To First Paint）：指的是从网页 HTTP 请求发出，到用户可以看到第一个有意义的内容渲染出来的时间差；
- TTI（Time To Interactive）：指的是从网页 HTTP 请求发出，到用户可以对网页内容进行交互的时间。

TTFP 这个时间差当然越短越好，也就是说应用要尽早显示有意义的内容给用户，不要让用户只对着一个空白屏幕发呆；TTI 肯定要比 TTFP 要长一些，因为没有内容哪里会有交互，当渲染出内容之后，还要 JavaScript 代码给 DOM 元素添加事件处理函数才会有交互结果，这个过程需要一些时间。

在一个完全靠浏览器渲染的应用中，当用户在浏览器中打开一个页面的时候，最坏情况下没有任何缓存，需要等待三个 HTTP 请求才能到达 TTFP 的时间点：

- 向服务器获取 HTML，虽然这个 HTML 只是一个无内容的空架子，但是皮之不存毛将焉附，这个 HTML 就是皮，在其中运行的 JavaScript 就是毛，所以这个请求是不可省略的；
- 获取 JavaScript 文件，大部分情况下，如果这是浏览器第二次访问这个网站，就可以直接读取缓存，不会发出真正的 HTTP 请求；
- 访问 API 服务器获取数据，得到的数据将由 JavaScript 加工产生之后用来填充 DOM 树，如果应用的是 React，那就是通过修改组件的状态或者属性来驱动渲染。

总共有三个 HTTP 请求，在三个不可预料的网络请求之后，才可以渲染第一个有意义内容，即使第二个 HTTP 请求利用了缓存，那也需要两个 HTTP 来回的时间。

实际上，按照 Progressive Web App 的规格，可以通过 Manifest 和 Service Worker 技术进一步优化，避免第一个获取 HTML 的请求和第三个访问 API 的请求，但是这种技术超出了本书讨论的范围，而且这些技术也并不适用于所有网页应用，在这里只讨论一般情况。

对于一个服务器端渲染，因为获取 HTTP 请求就会返回有内容的 HTML，所以在一个 HTTP 的周期之后就会提供给浏览器有意义的内容，所以首次渲染时间 TTFP 会优于完全依赖于浏览器端渲染的页面。

除了更短的 TTFP，服务器端渲染还有一个好处就是利于搜索引擎优化，虽然某些搜索引擎已经能够索引浏览器端渲染的网页，但是毕竟不是所有搜索引擎都能做到这一点，让搜索引擎能够索引到应用页面的最直接方法就是提供完整 HTML。

上面的性能对比当然只是理论上的分析，实际中，采用服务器端是否获得更好的 TTFP 有多方面因素。

首先，服务器端获取数据快还是浏览器端获取数据块？虽然服务器端渲染减少了一个浏览器和服务器之间的 HTTP 访问周期，也就是获取数据的过程，但是在网页服务器上一样有获取数据的过程，如果在网页服务器上访问数据存储或者 API 服务器的延时要比从浏览器端更短，那么才能体现服务器端渲染的优势。当然，大多数情况下，在网页服务器上获取数据的确更快，开发者只需要确认这一点就行。

其次，服务器端产生的 HTML 过大是否会影响性能？因为服务器渲染让网页服务器返回包含内容的 HTML，那样首页下载的 HTML 要比纯浏览器端渲染要大，这样下载 HTML 的时间也会增长，这样导致服务器渲染未必能获得更好的性能。特定于 React 应用，服务器端渲染需要在网页中包含“脱水数据”，除了 HTML 之外还要包含输入给 React 重新绘制的数据，这样导致页面的大小比最传统的服务器端渲染产生的页面还要大，更要考虑页面大小对性能的影响。

最后，服务器端渲染的运算消耗是否是服务器能够承担得起的？在浏览器渲染的方案下，服务器只提供静态资源，无论 HTML 还是 JavaScript，提供静态资源对服务器压力较小，甚至可以交给 CDN 来承担，这样服务器基本无压力，产生网页 HTML 的运算压力被分摊到了访问用户的浏览器中；如果使用服务器端渲染，那么页面请求都要产生 HTML 页面，这样服务器的运算压力也就增大了。对应 React，使用服务器端渲染可能对服务器的运算压力很大，因为 Facebook 已经明确说 React 并不是给服务器端渲染设计的，Facebook 本身也没有在实际产品中应用 React 的服务器端渲染。

总结一下，我们既列举了服务器端渲染优点，也分析了服务器端渲染潜在的风险。那么，应不应该使用 React 服务器端渲染呢？

没人能够给一个明确的答案，至今业内很多人觉得 React 的“同构”没有多大意义，但也有很多有志之士开发了很多服务器端渲染的实例。所以，在这里只是介绍“同构”的方法，每个应用都有每个应用的特点，开发者需要根据应用特点作出判断。

如果应用对 TTFP 没有那么高的要求，也并不希望对 React 页面进行搜索引擎优化，那就真没有必要使用“同构”来增加应用复杂度。

如果希望应用的性能百尺竿头更进一步，而且服务器端运算资源充足，那么可以试一试服务器端渲染。

React 被发明出来就是为了满足浏览器端渲染的需要，我们前面章节的例子也只考虑了 React 在浏览器端运行的情况，现在我们考虑如何在服务器端使用 React。

12.2 构建渲染动态内容服务器

虽然 Facebook 声称 React 并不是给服务器端渲染设计的，但是 React 真的很适合来做同构，因为 React 用声明的方式定义了用户界面如何渲染，就是这个我们反复强调的公式描述的：

$$UI = render(state)$$

React 组件描述了 render 过程，往里面塞进去 state 就能够得到用户界面 UI，这个过程既可以在浏览器端进行，也可以在服务器端进行，一份代码足矣，不像其他的服务器端渲染，需要分别开发服务器端渲染的代码和浏览器端代码。

现在我们就来看一看怎样用 React 实现同构，不过首先我们要搭建起服务器端渲染的框架。

因为 React 是 JavaScript 语言所写，我们的应用的也是 JavaScript 代码，那么服务器端要重用一样的代码当然也要支持 JavaScript 的运行，那就毫无悬念地应该选择 Node.js 作为服务器端的运行环境。

Node.js 只是一个 JavaScript 的运行环境，提供的 API 非常底层，所以我们还需要一个基于 Node.js 的应用框架来方便构建服务器端，这里我们选择最成熟的 Express 框架。

我们基于第 11 章中最后构建的单页应用来实践同构，不过同构是一个比较复杂过程，我们先实现一个小目标，那就是用 Node.js 和 Express 来代替 create-react-app 提供的启动脚本。

12.2.1 设置 Node.js 和 Express

在第 11 章中，我们已经对 create-react-app 产生的应用进行了“弹出”操作，这样我们终于获得了任意定制 webpack 的自由，不过应用启动脚本 npm start 依然是 create-react-app 生成的，启动之后虽然也有一个基于 Node.js 的服务器端在运行，但是只是简单地对所有请求返回 public 目录下的 index.html 文件，无法定制服务器端内容，所以只能自己动手创造新的启动脚本。

这一节相关代码可以在本书的 Github 库 https://github.com/mocheng/react-and-redux 的目录 chapter-12/express_server 下找到。

Node.js 提供的 API 都非常的底层，快速开发一个网络应用必须要借助 Node.js 之上框架的支持，在这里我们使用 Express 框架，还有在 Express 中我们要使用 ejs 模板工具，首先要安装相关 npm 包：

```
npm install --save express ejs
```

然后，我们在项目根目录下创造一个 server 目录，这个目录用于存放所有只在服务器端运行的代码，我们先要定义服务器端入口文件 server/index.js，代码如下：

```
const isProductionMode = (process.env.NODE_ENV === 'production');
const app = isProductionMode ? require('./app.prod.js'): require('./app.dev.
js');

if (!isProductionMode) {
  process.env.NODE_ENV = 'development';
}

const PORT = process.env.PORT || 9000;

app.listen(PORT, function() {
   console.log('running in ' + (isProductionMode ? 'producition' :
'development') + ' mode');
  console.log('listening on port: ' + PORT);
});
```

这个文件根据当前环境变量选的“开发模式”或者“产品模式”运行，因为两种模式下代码大不一样，所以主要代码分别放在 app.dev.js 和 app.prod.js 文件中。

为了避免和原有的 npm start 启动脚本冲突，程序监听的端口不是 3000 而是 9000。

相对而言，“产品模式”下的代码简单一些，我们先来看 server/app.prod.js 如何实现，代码如下：

```
const express = require('express');
const path = require('path');
```

```
const app = express();
const assetManifest = require(path.resolve(__dirname, '../build/asset-
  manifest.json'));

app.use(express.static(path.resolve(__dirname, '../build')));
app.get('*', (req, res) => {
  return res.render('index', {
    title: 'Sample React App',
    PUBLIC_URL: '/',
    assetManifest: assetManifest
  });
});

app.set('view engine', 'ejs'); // 使用ejs作为渲染模板。
app.set('views', path.resolve(__dirname, 'views')); //模板文件目录。

module.exports = app;
```

要运行“产品模式”，必须先通过 npm run build 命令编译产生所有的浏览器端 JavaScript 打包文件，这些打包文件都已经做过优化处理，而且文件名中包含 8 个字符的哈希值，所以要想确定这些打包文件实际的文件名，需要去读取转译过程中产生的描述文件，描述文件存放在项目目录下的 build/asset-manifest.json 中，这个文件内容是 JSON 形式，大概如下：

```
{
  "404.js": "static/js/404.d5ac31a0.chunk.js",
  "404.js.map": "static/js/404.d5ac31a0.chunk.js.map",
  "about.js": "static/js/about.8642419c.chunk.js",
  "about.js.map": "static/js/about.8642419c.chunk.js.map",
  ...
  "main.js.map": "static/js/main.f3a22285.js.map"
}
```

上面只是一个例子，每个文件名中的 8 位哈希值随文件内容改变而改变，任何一点代码逻辑修改都会导致文件名的不同。

要获得页面应该引用的 JavaScript 文件，只需读取 build/asset-manifest.json 文件中 main.js 和 common.js 对应的路径就行，其他的分片文件比如 home.js 和 404.js，打包文件 main.js 会按需去动态加载，也就是 require.ensure 函数被调用的时候去加载，无需我们操心。

在 app.prod.js 中，对于所有 HTTP 请求，先去 static 目录下匹配静态资源。如果找不到，就会用 app.get 指定的一个默认路径处理，和 React-Router 一样，“*”代表任何路径，默认路径的处理方式就是用 ejs 模板返回一个定制的 HTML 网页。

在 server/views/index.ejs 文件中是模板文件，代码如下：

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="shortcut icon" href="<%= PUBLIC_URL %>favicon.ico">
    <title><%= title %></title>
  </head>
  <body>
    <div id="root"></div>
    <script src="<%= PUBLIC_URL + assetManifest['common.js'] %>"></script>
    <script src="<%= PUBLIC_URL + assetManifest['main.js'] %>"></script>
  </body>
</html>
```

目前，这个模板文件只渲染了一个 div 作为浏览器端 React 的舞台，另外引入了 common.js 和 main.js 对应的实际 JavaScript 路径，产生的内容和 npm start 没有差别，这只是一个阶段性小目标，后面我们会让这个模板包含真正的服务器端渲染内容。

最后我们在 package.json 中的 scripts 部分增加一个指令：

```
"start_prod": "NODE_ENV=production node server/index.js",
```

就可以在命令行通过 npm run start_prod 来启动“产品模式”的应用了，这次的应用链接是在 http://localhost:9000 上，但是每次这样修改代码之后，都要先运行 npm run build 编译产生打包文件，实在很麻烦，所以大部分时间开发者还是要在‘开发模式’下运行程序，为了达到便于开发的目的，相对应的代码 server/app.dev.js 就要麻烦很多。

12.2.2 热加载

在 create-react-app 产生应用的 npm start 指令下，每次对任何代码的修改，都会让浏览器中页面自动刷新，这样当然会让我们省去了手动刷新的麻烦，但是，有时候开发者并不想这样，比如，我们发现了一个 Bug，一个最普通不过的 Debug 过程就是，修改一点代码，看看问题修复没有，没有就再修改一点代码，看看问题修复没有……直到问题消失了，不过，有的 Bug 很诡异，要在同一网页装载完成中做很多次操作之后才能复现，如果每改一点代码网页就刷新一次，开发者又要重复多次操作来看看是否修复了这个 Bug，如果没有重来，又要手工重复多次操作……这样页面自动刷新就显得很烦。

试想，既然在 React 中遵守 *UI=render(state)* 这样的公式，在代码更新的时候，如果只更新 render 的逻辑，而不去碰 state，那该多好！那样当发现 render 在某种 state 下暴露出 bug，只需保持 state 不变，反复替换 render 就可以验证 bug 是否被修复。

对于上面 Debug 的过程，如果每次代码更改的时候，不要去刷新网页，而是让网页

中的 React 组件渲染代码换成新的就行。因为在 Redux 框架下，我们把状态存在了 Store 上而不是在 React 组件中，所以这种替换完全可行。这种方式，叫做**热加载**（Hot Load）。目前 create-react-app 不支持热自动加载，但是我们现在自建服务器端，也就不受影响可以自己实现热自动加载了。

为了支持热自动加载，我们需要两个 Express 的中间件，一个叫 webpack-dev-middleware，用于动态运行 webpack 生成打包文件，另一个叫 webpack-hot-middleware，这个用于处理来自浏览器端的热加载请求，还需要一个 babel 装载起 react-hot-loader，用于处理 React 组件的热自动加载。注意这两个中间件是 Express 的中间件，并不是 Redux 的中间件。

首先在命令行安装这三个 npm 库：

```
npm install --save-dev webpack-dev-middleware webpack-hot-middleware react-hot-loader
```

然后我们要修改 config/webpack.config.dev.js，我们的“开发模式”服务器依然以这个文件作为 webpack 的配置。

默认的开发模式没有产生静态资源说明文件，但是我们的页面模板文件需要说明文件来获取 JavaScript 打包文件路径，所以，首先在文件顶部导入 ManifestPlugin，然后在 plugins 部分添加 ManifestPlugin 实例，参数指定了产生 asseet-manifest.json 文件，这和 npm build 脚本产生的资源说明文件是一致的：

```
var ManifestPlugin = require('webpack-manifest-plugin');

plugins: [
  ...
  new ManifestPlugin({
    fileName: 'asset-manifest.json'
  })
]
```

然后，在 entry 部分删掉或者注释掉原有的 webpackHotDevClient，因为我们不想要网页自动刷新了，用另一个 webpack-hot-middleware/client 来取代它：

```
//require.resolve( 'react-dev-utils/webpackHotDevClient' ),
 'webpack-hot-middleware/client' ,
```

在 loaders 部分，增加 react-hot 的应用：

```
{
  test: /\.js$/,
  include: paths.appSrc,
  loader: 'react-hot'
},
```

为了让热加载有效，还需要保证 webpack.HotModuleReplacementPlugin 存在于 plugins 中，默认配置中已经有这个插件，所以无需修改。

最后，我们增加新的 server/app.dev.js 文件，和 server/app.prod.js 相比多了不少内容，先是要导入 webpack-dev-middleware，代码如下：

```
const express = require('express');
const path = require('path');

const webpack = require('webpack');
const webpackConfig = require('../config/webpack.config.dev.js');
const compiler = webpack(webpackConfig);
const webpackDevMiddleware = require('webpack-dev-middleware')(
  compiler,
  {
    noInfo: true,
    publicPath: webpackConfig.output.publicPath
  });
```

为了让 express 服务器支持开发者的工作，需要使用新安装的两个中间件，而这两个中间件都需要 webpack 作为支持，我们在 JavaScript 代码中创建 webpack 实例，指定配置文件就是 config 目录下的 webpac.config.dev.js，和 npm start 使用的配置文件相同，习惯上，webpack 实例的变量名被称为 compiler。

在 Express 服务器启动的时候，webpack-dev-middleware 根据 webpack 来编译生成打包文件，之后每次相关文件修改的时候，就会对应更新打包文件。因为更新过程只需要重新编译更新的文件，这个速度会比启动时的完全编译过程快很多，当项目文件量很大的时候尤其突出，这就是在开发模式中使用 webpack-dev-middleware 的意义。

而且，webpack-dev-middleware 并没有将产生的打包文件存放在真实的文件系统中，而是存放在内存中的虚拟文件系统，所以要获取资源描述文件不能像产品环境那样直接 require 就行，而是要读取 webpack-dev-middleware 实例中的虚拟文件系统，对应的函数定义如下：

```
function getAssetManifest() {
  const content = webpackDevMiddleware.fileSystem.readFileSync(__dirname +
    '/../build/asset-manifest.json');
  return JSON.parse(content);
}
```

上面定义的 getAssetMannifest 文件读取的就是 webpack-dev-middleware 的虚拟文件系统中的打包说明文件。

虽然 webpack-dev-middleware 中间件能够完成实时更新打包文件，但是这只发生在服务器端，只有当浏览器刷新重新向服务器请求资源时才能得到更新的打包文件，而

webpack-hot-middleware 就更进一步，无需网页刷新，能够把代码更新“推送”到网页之中。

使用两个 express 中间件的代码如下：

```
const app = express();
app.use(express.static(path.resolve(__dirname, '../build')));
app.use(webpackDevMiddleware);
app.use(require('webpack-hot-middleware')(compiler, {
  log: console.log,
  path: '/__webpack_hmr',
  heartbeat: 10 * 1000
}));
```

webpack-hot-middleware 的工作原理是让网页建立一个 websocket 链接到服务器，服务器支持 websocket 的路径由 path 参数指定，在我们的例子中就是 /__webpack_hmr。每次有代码文件发生改变，就会有消息推送到网页中，网页就会发出请求获取更新的内容。

最后，和“产品模式”的 server/app.prod.js 一样，需要用 app.get 指定一个默认的路由处理方式，对于所有非静态资源都返回一个 ejs 模板的渲染结果，代码如下：

```
app.get('*', (req, res) => {
  const assetManifest = getAssetManifest();
  return res.render('index', {
    title: 'Sample React App',
    PUBLIC_URL: '/',
    assetManifest: assetManifest
  });
});
app.set('view engine', 'ejs');
app.set('views', path.resolve(__dirname, 'views'));

module.exports = app;
```

在 package.json 中的 scripts 部分增加下面的一个指令：

```
"start:isomorphic ": "NODE_ENV=development node server/index.js",
```

然后，我们就可以在命令行用 npm start:isomorphic 命令启动开发模式的应用。

在浏览器中访问 http://localhost:9000，可以看到程序界面，这时候我们尝试改变一个 React 组件的源代码，比如将 src/pages/Home.js 文件中的“ Home ”改成“ 主页 ”，可以发现，在浏览器中无需刷新网页，对应页面自动发生了变化。

在浏览器的 Console 中，可以看到整个自动更新的过程，如下所示：

```
[HMR] connected
[HMR] bundle rebuilding
[HMR] bundle rebuilt in 339ms
[HMR] Checking for updates on the server...
[HMR] Updated modules:
[HMR]  - ./src/pages/Home.js
[HMR] App is up to date.
▶ Warning: [react-router] You cannot change <Router routes>; it will be ignored
```

HMR 代表的就是 Hot-Module-Reload，从日志中可以看到修改 Home.js 文件的更新被浏览器端发现并做了自动更新。

不过，也可以发现这样一行错误警告：

```
Warning: [react-router] You cannot change <Router routes>; it will be ignored
```

这个错误警告是因为 React-Router 在热加载时的报警，这并不会产生什么实质危害，但是这个错误警告总是出现也很让人讨厌。我们可以修改 src/Routes.js，把路由规则从 Router 的子组件中抽取出来作为独立变量，就可以解决这个问题，代码如下：

```
const routes = (
  <Route path="/" component={App}>
    <IndexRoute getComponent={getHomePage} />
    <Route path="home" getComponent={getHomePage} />
    <Route path="counter" getComponent={getCounterPage} />
    <Route path="about" getComponent={getAboutPage} />
    <Route path="*" getComponent={getNotFoundPage} />
  </Route>
);

const Routes = () => (
  <Router history={history} createElement={createElement}>
    {routes}
  </Router>
);
```

改完之后，这个错误提示就消失了。

现在我们已经有了服务器端渲染的开发模式和产品模式，接下来我们要实现真正的同构。

12.3 React 同构

为了实现同构，需要实现这些功能。

- ❑ 在服务器端根据 React 组件产生 HTML；
- ❑ 数据脱水和注水；

- 服务器端管理 Redux Store；
- 支持服务器和浏览器获取共同数据源。

这是一个复杂的过程，让我们一一道来。

12.3.1 React 服务器端渲染 HTML

React 在服务器端渲染使用函数和浏览器端不一样，在浏览器端的函数是 render，接受两个参数，第一个是一个 React 组件，第二个是这个组件需要装载的 DOM 节点位置，代码模式是这样：

```
import ReactDOM from 'react-dom';
React.render(<RootComponent />,  document.getElementById('root'));
```

当 render 函数被调用的时候，必须保证 id 为 root 的 DOM 元素作为容器真的存在，render 函数的工作就是启动 RootComponent 的装载过程，最后产生的 DOM 元素就存在 root 容器之中。

在浏览器端，最终的产出是 DOM 元素，而在服务器端，最终产出的是字符串，因为返回给浏览器的就是 HTML 字符串，所以服务器端渲染不需要指定容器元素，只有一个返回字符串的函数 renderToString，使用这个函数的代码模式是这样：

```
import ReactDOMServer from 'react-dom/server';
const appHtml = ReactDOMServer.renderToString(<RootComponent />);
```

renderToString 函数的返回结果就是一个 HTML 字符串，至于这个字符串如何处理，要由开发者来决定，当然，在这个例子中，为了和浏览器端一致，我们会把返回的字符串嵌在 id 为 root 的元素中。

当然，只是把 renderToString 返回的字符串在浏览器中渲染出来，用户看到的只是纯静态的 HTML 而已，并不具有任何动态的交互功能。要让渲染的 HTML“活”起来，还需要浏览器端执行 JavaScript 代码。因为 React 将 HTML、样式和 JavaScript 封装在一个组件中的特点，所以让 React 组件渲染出的 HTML“活”起来的代码就存在于 React 组件之中，也就是说，如果应用 React 服务器端渲染，一样要利用 React 浏览器端渲染。

过程是这样，服务器端渲染产生的 React 组件 HTML 被下载到浏览器网页之中，浏览器网页需要使用 render 函数重新渲染一遍 React 组件。这个过程看起来会比较浪费，不过在浏览器端的 render 函数结束之前，用户就已经可以看见服务器端渲染的结果了，所以用户感知的性能提高了。

为了避免不必要的 DOM 操作，服务器端在渲染 React 组件时会计算所生成 HTML 的校验和，并存放在根节点的属性 data-react-checksum 中。在浏览器渲染过程中，在重

新计算出预期的 DOM 树之后，也会计算一遍校验和，和服务器计算的校验和做一个对比。如果发现二者相同，就没有必要做 DOM 操作了，如果不同，那就应用浏览器端产生的 DOM 树，覆盖掉服务器产生的 HTML。

很明显，如果服务器端渲染和浏览器端渲染产生的内容不一样，用户会先看到服务器端渲染的内容，随后浏览器端渲染会重新渲染内容，用户就会看到一次闪烁，这样给用户的体验很不好。所以，实现同构很重要的一条，就是一定要保证服务器端和浏览器端渲染的结果要一模一样。

如果我们能保证服务器端和浏览器使用的 React 组件代码是一致的，那导致渲染结果不一致的唯一可能就是提供给 React 组件的数据不一致。

为了让两端数据一致，就要涉及“脱水”和“注水”的概念。

12.3.2 脱水和注水

“那你只能脱水了。”周文王说，一手用兽皮扇着风。

“脱水以后，你不会扔下我吧？”

“当然不会，我保证把你带到朝歌。”

——《三体》

在《三体》[⊖]的科幻故事中，三体星被三个太阳操控，随时可能陷入酷热的境地，所以三体人进化出一种“脱水”的能力，当炎热缺水时，三体人可以脱水进入休眠状态，脱水之后的体重变轻，可以由其他三体人搬运，当水量充足时，只要重新“注水”，三体人就能够复活过来。

React 同构中的“脱水”（Dehydrate）和“注水”（Rehydrate）就和三体人的这种奇异能力相似。

服务器端渲染产出了 HTML，但是在交给浏览器的网页中不光要有 HTML，还需要有“脱水数据”，也就是在服务器渲染过程中给 React 组件的输入数据，这样，当浏览器端渲染时，可以直接根据“脱水数据”来渲染 React 组件，这个过程叫做“注水”。使用脱水数据可以避免没有必要的 API 服务器请求，更重要的是，保证了两端渲染的结果一致，这样不会产生网页内容的闪动。

脱水数据的传递方式一般是在网页中内嵌一段 JavaScript，内容就是把传递给 React 组件的数据赋值给某个变量，这样浏览器就可以直接通过这个变量获取脱水数据。

使用 EJS 作为服务器端模板，渲染脱水数据的代码模式基本这样，其中 appHTML

⊖ 《三体》第 7 章《周文王 . 长夜》，刘慈欣著 .

是 React 的 renderToString 返回的 HTML 字符串，而 dehydrateState 就是脱水数据的 JSON 字符串表示形式，赋值给了全局变量 DEHYDRATED_STATE，在浏览器端，可以直接读取到这个变量：

```
<div id="root"><%- appHtml %></div>
<script>
  var DEHYDRATED_STATE = <%- dehydratedState %>
</script>
```

需要注意的是，使用脱水数据要防止跨站脚本攻击（XSS Attack），因为脱水数据有可能包含用户输入的成分，而用户的输入谁也保不准包含什么，假如 dehydrateState 包含用户可以控制的字符串，那就可能被利用，产生下面的网页输出：

```
<script>
  var DEHYDRATED_STATE = "...</script><script>doBadThing()">
</script>
```

既然我们使用 Redux 来管理应用数据，那么脱水数据的就应该是来自于 Redux 的 Store，借助于 react-redux 的 Provider 帮助，可以很容易地让所有 React 都从一个 store 获得数据，然后我们在调用服务器端渲染的函数 renderToString 之后调用 store 的 getState 函数，得到的结果就可以作为“脱水数据”：

```
const appHtml = ReactDOMServer.renderToString(
  <Provider store={store}>
    <RouterContext {...renderProps} />
  </Provider>
);

const dehydratedState= store.getState();
```

就像三体人脱水之后就应该变得很轻一样，脱水数据一定不能太大，因为脱水数据要占用网页的大小，如果脱水数据过大，可能会影响性能，让服务器端渲染失去意义。

由此，我们再次看出让 Redux Store 上的不要存冗余数据的必要性，只要我们保证 Store 上状态没有冗余，在产生脱水数据的时候就轻松太多，不然很难分辨哪些数据不必要放在脱水数据中。

12.3.3 服务器端 Redux Store

在服务器端使用 Redux，必须要对每个请求都创造一个新的 Store，这是和浏览器渲染的最大区别。

在浏览器端，网页只需要满足一个用户的需要，所以一个 Store 就足够了，但是在服务器端会接受到很多浏览器端的请求，毕竟我们的服务器不会设计成只满足一个用户

在线使用，既然特定每个请求的数据存在 Store 里，当然对每个请求都要重新构建一个 Store 实例。

所以，我们要修改一下 src/Store.js 的实现，把构建 Store 的代码放在一个函数中：

```
const configureStore = () => {
  const store = createStore(reducer, {}, storeEnhancers);
  store._reducers = originalReducers;
  return store;
}
export {configureStore};
```

因为 src/Store.js 导出的不再是一个 Store 实例而是一个 configureStore 函数，对应导入这个文件的代码也要对应改变。

12.3.4 支持服务器和浏览器获取共同数据源

脱水数据只在浏览器访问的首页发挥作用，之后，用户可以操作网页跳转，这时候网页需要自己获得数据提供给 React 组件。

举个例子，我们希望 Counter 页面显示的计数初始值由服务器端确定，而不是一个代码中固定的数字。用户首先直接访问 http://localhost:9000/home 页面，然后又通过点击顶栏的“Counter”链接进入 http://localhost:9000/counter 页面，因为这是一个单页应用，Counter 页面的 HTML 是完全由浏览器端渲染的，所以没有服务器的脱水数据，所以浏览器需要一个 API 请求来获取初始值。同样，用户可能在 Counter 页面点击浏览器刷新按钮，或者在另一个窗口直接输入 Counter 页面的地址，这时网页的 HTML 是服务器端渲染产生的，这样服务器也需要有能够获取初始值的能力。

很明显，最简单的方法，就是有一个 API 服务器提供接口让服务器和浏览器都能够访问，这样无论是什么样的场景，服务器和浏览器获得数据都是一致的。

API 服务器作为独立的服务器可能并不在网页服务器同一个域名下，这样如果要浏览器端访问，就要在 API 服务器配置跨域访问策略，有的时候，API 服务器并不在一个产品的控制范围内，无法配置跨域策略，这样，就需要网页服务器同域名下搭起一个代理，把请求转发到 API 服务器。

在我们的例子中，我们在 server/app.dev.js 中增加一个路由规则，模拟一个 API 服务器，提供一个计数初始值的接口：

```
app.use('/api/count', (req, res) => {
  res.json({count: 100});
});
```

到现在为止，我们的理论准备工作差不多了，接下来就要实践同构。

12.3.5 服务器端路由

因为浏览器端使用了 React-Router 作为路由，没有理由不在服务器端使用一致的方法，不过在服务器端使用 React-Router 的方式和浏览器端不一样，在浏览器端，整个 Router 作为一个 React 组件传递一个 ReactDOM 的 render 函数，Router 可以自动和 URL 同步，但是对于服务器端的过程，URL 对应到路由规则的过程需要用 match 函数：

```
import {match, RouterContext} from 'react-router';

match({routes: routes, location: requestUrl}, function(err, redirect,
  renderProps) {
  if (err) {
    return res.status(500).send(err.message);
  }
  if (redirect) {
    return res.redirect(redirect.pathname + redirect.search);
  }
  if (!renderProps) {
    return res.status(404).send('Not Found');
  }
  const appHtml = ReactDOMServer.renderToString(
    <RouterContext {...renderProps} />
  );
});
```

match 函数接受一个对象和一个回调函数作为参数，对象参数中的 routes 就是 Route 构成的路由规则树，这里根本用不上 Router 类，所以也用不上 Router 的 history 属性，这就是和浏览器端渲染的最大区别。match 是通过对象参数中的 location 字段来确定路径的，不是靠和浏览器地址栏关联的 history。

当 match 函数根据 location 和 routes 匹配完成之后，就会调用第二个回调函数参数，根据回调函数第一个参数 err 和第二个参数 redirect 可以判断匹配是否错误或者是一个重定向。一切顺利的话，第一第二个参数都是空，有用的就是第三个参数 renderProps，这个 renderProps 包含路由的所有信息，把它用扩展操作符展开作为属性传递给 RouterContext 组件，渲染的结果就是服务器端渲染产生的 HTML 字符串。

如果应用没有使用代码分片，浏览器端的路由部分就无需任何改变，不过在我们的例子中已经应用了代码分片，所以应用了服务器端渲染之后，浏览器端渲染也要做对应修改，使用 match 函数来完成匹配，否则，服务器端和浏览器中产生的 HTML 会不一致，这种不一致不是脱水数据问题导致，而是产生两端的代码不一致导致的：

```
match({history, routes}, (err, redirectLocation, renderProps) => {
    ReactDOM.render(
      <Router {...renderProps} />
      domElement
    );
  });
```

可以注意到，到浏览器端 match 函数确定当前路径的参数又是用 history，不像服务器端那样使用 URL 字符串。

12.4 同构实例

现在我们来看一看实现同构的完整例子。

首先，CounterPage 的定义要发生变化，之前 CounterPage 导出一个 intialState 是固定的值，现在我们希望服务器和浏览器端共用一个的数据源，这个数据源就是之前我们定义的 API 接口，所以将 intialState 改为 initState 函数，这个函数返回一个 Promise 实例，在下面的代码中可以看到，使用 Promise 可以大大简化代码的结构。

在 src/pages/CounterPage.js 文件中，initState 函数根据环境变量 HOST_NAME 决定 API 地址，如果没有 HOST_NAME 那就是用一个指向本地开发环境的域名，这是一个常用的技巧。

可以看到，initState 返回的是一个 Promise 对象，代码如下：

```
const END_POINT = process.env.HOST_NAME || 'localhost:9000';
const initState = () => {
  return fetch(`http://${END_POINT}/api/count`).then(response => {
    if (response.status !== 200) {
      throw new Error('Fail to fetch count');
    }
    return response.json();
  }).then(responseJson => {
    return responseJson.count;
  });
}
```

而且，src/pages/CounterPage.js 的导出语句和其他页面文件不一样，对于 CounterPage，不只要导出视图 page，还要导出 reducer、stateKey 和 initState，代码如下：

```
export {page, reducer, initState, stateKey};
```

处理完 CounterPage，接下来就要准备服务器端的路由逻辑。

因为实际的渲染工作和路由关系紧密，所以要把这两个功能都集中在文件 server/routes.

Server.js 中，让 server/app.dev.js 和 server/app.prod.js 只需要提供 assetManifest，把 req 和 res 参数传递给 renderPage 函数，代码如下：

```
const renderPage = require('./routes.Server.js').renderPage;

app.get('*', (req, res) => {
  if (!assetManifest) {
    assetManifest = getAssetManifest();
  }
  return renderPage(req, res, assetManifest);
});
```

所以，主要的功能其实是在 server/routes.Server.js 文件中，我们来看看这个文件。

在服务器端渲染，没有必要使用分片，自然也不需要动态加载模块，所有的页面都是直接导入，比如导入 Home 页面的代码就是这样：

```
import Home from '../src/pages/Home.js';
```

对于 App、Home、About 和 NotFound 页面用上面的方法 import 就可以，但是 CounterPage 导入有一点特殊，因为不仅要导入视图，还要导入这个页面对应的 reducer、stateKey 和初始状态 initState，代码如下：

```
import {page as CounterPage, reducer, stateKey, initState} from '../src/pages/
    CounterPage.js';
```

路由规则，因为不需要分片加载，所以非常简单，可以直接用一个 routes 变量代表，代码如下：

```
const routes = (
  <Route path="/" component={App}>
    <IndexRoute component={Home} />
    <Route path="home" component={Home} />
    <Route path="counter" component={CounterPage} />
    <Route path="about" component={About} />
    <Route path="*" component={NotFound} />
  </Route>
);
```

最关键的就是 renderPage 函数，这个函数承担了来自浏览器请求的路由和渲染工作。

服务器端路由使用 React-Router 提供的 match 函数，如果匹配路由成功，那就调用另一个函数 renderMatchedPage 渲染页面结果，代码如下：

```
export const renderPage = (req, res, assetManifest) => {
  match({routes: routes, location: req.url}, function(err, redirect, renderProps) {
    // 检查error和redirect，如果存在就让res结束
    return renderMatchedPage(req, res, renderProps, assetManifest);
```

```
  });
};
```

当 renderMatchedPage 函数被调用时，代表已经匹配中了某个路由，接下来要做的工作就是获得相关数据把这个页面渲染出来。

之前已经说过，为了保证服务器对每个请求的独立处理，必须每个请求都建一个 Store，所以 renderMathedPage 函数做的第一件事就是创建一个 Store，然后获取初始化 Store 状态的 Promise 对象，代码如下：

```
function renderMatchedPage(req, res, renderProps, assetManifest) {
  const store = configureStore();
  // 获取匹配Page的initStata函数
  const statePromise = (initState) ? initState() : Promise.resolve(null);
```

在这个应用中，只有 CounterPage 才提供 initState，如果匹配的页面不提供 initState，那就使用 Promise.resolve 产生一个 Promise 对象，这个对象不提供任何数据。但是，在没有 initState 的情况下也能给 statePromise 变量一个 PromiseB 对象，从而使得后续处理代码不用关心如何获得数据。

当 statePromise 的 then 函数被调用时，代表页面所需的文件已经准备好，这时候首先要设置 Store 上的状态，同时也要更新 Store 上的 reducer，通过我们定义的 reset enhancer 可以完成，代码如下：

```
statePromise.then((result) => {
  if (stateKey) {
    const state = store.getState();
    store.reset(combineReducers({
      ...store._reducers,
      [stateKey]: reducer
    }), {
      ...state,
      [stateKey]: result
    });
  }
```

至此，Store 已经准备好了，接下来就通过 React 提供的服务器端渲染函数 renderToString 产生对应的 HTML 字符串，存放在 appHtml 变量中，代码如下：

```
const appHtml = ReactDOMServer.renderToString(
  <Provider store={store}>
    <RouterContext {...renderProps} />
  </Provider>
);
```

返回给浏览器的 HTML 中只包含 appHtml 还不够，还需要包含“脱水数据”，所以

在渲染完 HTML 之后，要立即把 Store 上的状态提取出来作为“脱水数据”，因为服务器端 React 组件用的是同样的状态，所以浏览器端用这样“脱水数据”渲染出来的结果绝对是一样的。

获取“脱水数据”的代码如下：

```
const dehydratedState= store.getState();
```

到这里，React 组件产生的 HTML 准备好了，“脱水数据”也准备好了，接下来就通过 Express 提供的 res.render 渲染结果就可以，代码如下：

```
res.render('index', {
  title: 'Sample React App',
  PUBLIC_URL: '/',
  assetManifest: assetManifest,
  appHtml: appHtml,
  dehydratedState: safeJSONstringify(dehydratedState)
});
```

这里，“脱水数据” dehydratedState 通过函数 safeJSONstringify 被转化为字符串，这样在 ejs 模板文件中直接渲染这个字符串就行。注意，这里不能直接使用 JSON.stringify 转化为字符串，因为“脱水数据”可能包含不安全的字符，需要避免跨站脚本攻击的漏洞。

safeJSONstringify 函数的代码如下：

```
function safeJSONstringify(obj) {
  return JSON.stringify(obj).replace(/<\/script/g, '<\\/script').replace(/<!--/g,
    '<\\!--');
}
```

最后，让我们看一看 ejs 模板文件 server/views/index.ejs，服务器端渲染产生的 appHtml 字符串和脱水数据 dehydratedState 在这里被渲染到返回给浏览器的 HTML 中，代码如下：

```
<body>
    <div id="root"><%- appHtml %></div>
    <script>
      var DEHYDRATED_STATE = <%- dehydratedState %>
    </script>
    <script src="<%= PUBLIC_URL + assetManifest['common.js'] %>"></script>
    <script src="<%= PUBLIC_URL + assetManifest['main.js'] %>"></script>
  </body>
</html>
```

其中 dehydratedState 是被渲染到内嵌的 script 中，赋值给一个 DEHYDRATED_STATE 变量，这个变量在浏览器端可以被访问到，利用这个变量就可以对 React 组件“注水”，让它们重生，接下来我们看看浏览器端如何实现“注水”。

和服务器端一样，入口函数 src/index.js 把渲染的工作完全交给 Routes.js，所做的只是提供装载 React 组件的 DOM 元素。

```
import {renderRoutes} from './Routes.js';
renderRoutes(document.getElementById('root'));
```

在 src/Routes.js 文件中是更新的浏览器端路由和渲染功能，主要的改变在 getCounterPage 函数中，代码如下：

```
const getCounterPage = (nextState, callback) => {
  require.ensure([], function(require) {
    const {page, reducer, stateKey, initState} = require('./pages/CounterPage.
      js');
    const dehydratedState = (win && win.DEHYDRATED_STATE);
    const state = store.getState();
    const mergedState = {...dehydratedState, ...state};
    const statePromise = mergedState[stateKey]
      ? Promise.resolve(mergedState[stateKey])
      : initState();
    // 继续处理statePromise
  }
```

和服务器端类似，首先要获取一个 statePromise，优先从“脱水数据”中获得初始状态，只有没有脱水初始状态的时候，才使用 initState 函数去异步获取初始化数据。

当 statePromise 完成时，一样可以使用 reset 功能设置 Store 的状态和 reducer，代码如下：

```
statePromise.then((result) => {
  store.reset(combineReducers({
    ...store._reducers,
    [stateKey]: reducer
  }), {
    ...state,
    [stateKey]: result
  });
```

因为使用了服务器端渲染，同时浏览器端使用了 React-Router 的代码分片功能，所以浏览器端也需要用 match 函数来实现路由，代码如下：

```
export const renderRoutes = (domElement) => {
  match({history, routes}, (err, redirectLocation, renderProps) => {
    ReactDOM.render(
      <Provider store={store}>
        <Router {...renderProps} />
      </Provider>,
      domElement
    );
```

```
  });
}
```

至此，一个同构应用完成了，以 http://localhost:9000/counter 页面为例，如果在浏览器中直接访问这个页面，在服务器端就会通过 API 服务器获得初始计数值，返回的网页中包含完整 HTML 和脱水的初始计数值；如果直接访问其他页面，然后通过顶栏链接切换到 Counter 页面，浏览器就会通过一个 API 请求获得初始数据完成浏览器端渲染。

12.5 本章小结

在这一章中，我们首先对比了服务器端渲染和浏览器端渲染各自的优缺点，React 本身很适合创建同构的应用，因为一个功能模块全部由 JavaScript 实现，既可以在服务器端运行，也可以在浏览器端运行。

为了实现服务器端渲染，对 create-react-app 产生的应用配置要做一系列修改，要使用 express 来作为服务器框架。

为了实现同构，服务器端除了提供渲染出的 HTML 字符串，还要提供脱水数据，才能保持两端渲染的内容一致。

和浏览器不同，服务器要对每个请求都创建一个 Store 实例，这样才能保持各个请求互不干扰。

结　语

恭喜你，你已经读到了本书的结尾，但是，作为对 React 和 Redux 技术的探索和应用，旅程才刚刚开始。

作为一个作者，本人感谢你花时间阅读我的文字，React 和 Redux 的应用是无限的，技术的发展和进化也是飞速向前，也许，这本书中介绍的具体技能很快就会过时，但是我们探讨的原理则永远都不会过时。

在最后，让我们闭上眼睛想一想 React 和 Redux 给我们留下了什么印象，请用一个词来描述，每个人可能会找到不同的词语，也许你会想到“强大”，也许你会想到“难学”，也许你会想到“深奥”。

让我告诉你我想到的词语，那就是“限制”。

“限制”在这里绝不是贬义词，恰恰相反，是对技术框架的最高夸奖，因为限制能够确保程序按照可控的方式进化。

在计算机软件世界里，造物主就是人类自己，没有物理化学的限制，一切皆有可能。也正因为一切皆有可能，一个问题即使没有无数种解法，也会有很多很多种解法，也许你会觉得这是天赐之福，毕竟，拥有多个解决方法让我们可以从中选出最合适的方案。但是，拥有很多方案并不表示我们应该使用所有的方案。

对于工程化的开发，使用太多的解决方法可能是一个灾难。

软件要由人来维护和开发，人的脑力是有局限的，当程序极度复杂的时候，牵一发而动全身，bug 层出不穷，即使最专业的程序员也会丧失勇气，而造成复杂的根源之一就是采用过多的解法，让系统结构失去了控制。

作为程序员，我们一直追求工具的“强大”。但最后可能会发现所谓的“强大”，并不是这个工具能够帮我们做所有的事情，而是这个工具能够保证所有人按照一个有纪律有规矩的方式去完成任务。

以往我们以为靠惯例可以实现纪律，事实证明不行，程序员也是人，人就会违反惯

例，所以，与其苦口婆心说服程序员遵守规则，不如让技术框架本身施加限制，让程序员没有其他选择，只有规定的一条路可走。

React 和 Redux 技术框架最大的好处，并不是让我们无所不能，而是设定了一套规矩，让开发者只能按照这套规矩来完成代码。这样，只要理解了这套规矩，无论产生的代码由谁来维护由谁来继续开发都不会有大问题。

有人会问，“限制”会抑制程序员的创造力吗？无稽之谈！如果程序员的创造力能够被这些限制给抑制住，那程序员也就不能称为程序员了。实际上，在 React 和 Redux 的技术框架上，众多程序员发挥更大的创造力，开发出了更多更强大的工具和应用。

推崇“限制”可能违背普通人的思维，但是我相信你作为一个软件工程师，最终一定能够理解其中的含义。

在电影《黑客帝国》系列故事的最后，主角“救世主”尼奥终于面对了母体的架构师，这两个人代表了两种不同的思想，尼奥代表了不断挑战母体系统的黑客精神，架构师则通过各种手段来限制居于母体中的人类，保证系统的稳定。

在现实世界里，我们通常都尊崇同为人类的尼奥，但在软件开发的世界里，程序员的角色更应该像是尼奥，还是母体的架构师呢？

相信经过思考，你会有自己的答案。

再次感谢你阅读这本书，再见！

實戰

“React和Redux帮助我们重新思考前端网页的构建方式，希望更多的开发者能够熟悉这两种技术，阅读这本书就是一个很好的开始。”

——诸葛越，Hulu公司全球研发副总裁，中国研发中心总经理

“程墨在Velocity China上的演讲很精彩，他能够把复杂的技术问题讲解得透彻、幽默。希望他的这本书也能给读者带来不同以往的技术书籍阅读体验，像Head First系列一样生动有趣。”

—— Douglas Wan，Chief Editor of O’Reilly Media Inc. Beijing

“阅读程墨的书，不仅能获得对一种技术的理解，而且能深入体会这种技术选择背后的动因，知其然也要知其所以然，这本书在这方面做得相当出色。”

——秦适，微博易、云鸟配送联合创始人兼CTO

本书主要特点：

- 如何避免复杂的技术栈配置，快速上手前端开发。
- 如何学习实用的空间和应用构建方式。
- 介绍框架演进的过程，揭示其背后的发展规律。
- 深入讲解React高阶组件的用法。
- 充分探讨Redux数据管理技巧。
- 全面介绍同构的架构设计。
- 本书所有代码在Github上，网址 https://github.com/mocheng/react-and-redux，读者可以随时下载学习。

上架指导：计算机\程序设计
ISBN 978-7-111-56563-5
9 787111 565635 >
定价：69.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn