

该项目中其他的功能逻辑大致类似，此处不再赘述。有兴趣的读者可以在 GitHub 中下载并翻阅上述代码。

本节项目的源码读者可以从 GitHub 上复制下来并运行，地址是 <https://github.com/khno/react-redux-example.git>。该项目可以通过在终端执行以下命令获取并启动：

```
git clone https://github.com/khno/react-redux-example.git  
npm install  
npm start
```

项目启动后展示效果如图 4.8 所示。

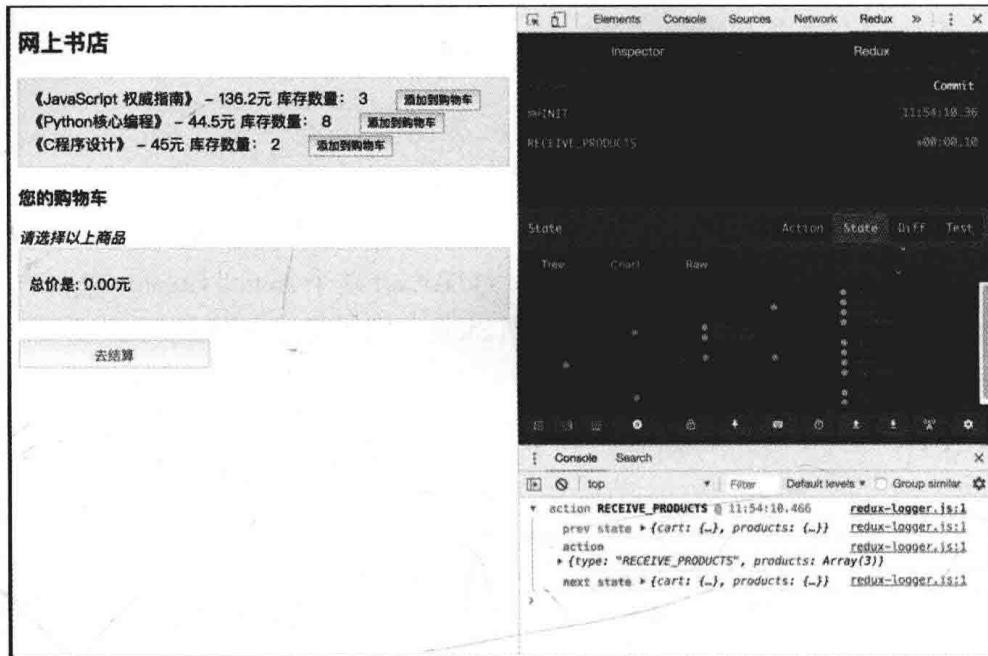


图 4.8 网上书店截图

第 5 章 路由

近几年单页应用（Single Page Web Application，SPA）发展迅猛，前端控制路由变为主流，单页应用也是前端路由的主要使用场景。在传统多页应用开发中，路由的概念往往存在于后端（后端路由），用户每次访问一个新的页面会向后台发起请求，然后服务器去响应。这个过程需要用户等待，从性能和用户体验上来讲，无疑前端路由会比后端路由提升很多。本章将介绍 React 中的路由实现原理和使用方法。

5.1 前端路由简介

在一般 SPA 开发中，路由的管理十分重要，它是 React 技术体系中非常重要的组成部分。

那么，什么是路由呢？路由是根据不同 URL 地址展示不同的内容或页面。

简单来说，假如有一台 Web 服务器地址是 `http://192.168.0.1`，在它下面有 `index` 首页、`lists` 列表页和 `details` 详情页 3 个页面，那么用户访问将是：

```
http://192.168.0.1/index  
http://192.168.0.1/lists  
http://192.168.0.1/details
```

它们的路径是`/index`、`/lists` 和`/details`。当用户访问详情页面的时候，服务器收到这个请求，然后会解析 URL 中的路径`/details`。按照以往的做法就是后台会去配置这几个路径，然后导航到对应的 HTML 页面。如果后台没有配置，那么这个页面就无法正常访问。

前端路由其实就是通过 JavaScript 来配置路由。单页应用基本也是前后端分离的，因此后端不提供路由。前端路由和后端路由实现原理是一样的，只是实现方式不同。

5.2 前端路由的实现原理

前端路由的主要实现方式有两种：Hash 和 HTML 5 的 history API。本节我们来讲解前端路由的具体实现方式。

5.2.1 history API 方式

在 HTML 5 的 history API 之前，前端路由通过 Hash 来实现，它也能兼容低版本浏览器。

先来看看 Mozilla (Mozilla 基金会) 在其官方文档的描述：

```
// 只读属性。返回一个整数，该整数表示会话历史中元素的数目，包括当前加载的页。例如，在一个新的选项卡加载的一个页面中，这个属性返回 1
window.history.length
Returns the number of entries in the joint session history.

// 允许 Web 应用程序在历史导航上显式地设置默认滚动恢复行为。此属性可以是自动的 (auto) 或者手动的 (manual)
window.history.scrollRestoration[=value]
Returns the scroll restoration mode of the current entry in the session history.
Can be set, to change the scroll restoration mode of the current entry in the session history.

// 返回一个表示历史堆栈顶部的状态值。这是一种可以不必等待 popstate 事件而查看状态的方式。
window.history.state
Returns the current serialized state, deserialized into an object.

// 通过当前页面的相对位置从浏览器历史记录（会话记录）加载页面
window.history.go([delta])
Goes back or forward the specified number of steps in the joint session history.
A zero delta will reload the current page.
If the delta is out of range, does nothing.

// 往上一页，用户可单击浏览器左上角的返回按钮模拟此方法，等价于 history.go(-1)
window.history.back()
Goes back one step in the joint session history.
If there is no previous page, does nothing.

// 往下一页，用户可单击浏览器左上角的前进按钮模拟此方法，等价于 history.go(1)
window.history.forward()
Goes forward one step in the joint session history.
If there is no next page, does nothing.

// 按指定的名称和 URL (如果提供该参数) 将数据 push 进会话历史栈，数据被 DOM 进行不透明处理；可以指定任何可以被序列化的 JavaScript 对象
window.history.pushState(data,title[,url])
Pushes the given data onto the session history, with the given title, and, if provided and not null, the given URL.

// 按指定的数据，名称和 URL (如果提供该参数)，更新历史栈上最新的入口。这个数据被 DOM 进行了不透明处理。可以指定任何可以被序列化的 JavaScript 对象。
window.history.replaceState(data,title[,url])
Updates the current entry in the session history to have the given data, title, and, if provided and not null, URL.
```

先来关注上述代码中最下面的 2 个接口，history.pushState() 和 history.replaceState()。这是 history 新增的 API，可以为浏览器提供历史栈。这两个 API 都接收 3 个参数，分别是：

- 状态对象 (state object)：一个 JavaScript 对象，与用 pushState() 方法创建的新历史

记录条目关联。无论何时用户导航到新创建的状态，`popState` 事件都会被触发，并且事件对象的 `state` 属性都包含历史记录条目的状态对象的复制。

- **标题 (title)**：FireFox 浏览器目前会忽略该参数，虽然以后可能会用上。考虑到未来可能会对该方法进行修改，传一个空字符串会比较安全。或者也可以传入一个简短的标题，标明将要进入的状态。
- **地址 (URL)**：新的历史记录条目的地址。浏览器不会在调用 `pushState()` 方法后加载该地址，但之后可能会试图加载，例如用户重启浏览器。新的 URL 不一定是绝对路径；如果是相对路径，它将以当前 URL 为基准；传入的 URL 与当前 URL 应该是同源的，否则，`pushState()` 会抛出异常。该参数是可选的；如果不指定，则为文档当前 URL。

`pushState()` 会增加一条新的历史记录，当使用了这个接口，浏览器的返回按钮能返回到上一次访问的 URL，在浏览器的历史记录中也会多出一条记录。而 `replaceState()` 会替换当前历史记录，浏览器的返回按钮无法返回到上一次的记录，浏览器的历史记录中也将用当前的 URL 替换为上一次的 URL 作为浏览记录储存。读者可以自行在浏览器控制台执行这 2 个方法，然后在浏览器历史记录中查看并验证。

值得注意的是，`pushState()` 和 `replaceState()` 这 2 个接口不会引起页面刷新，这一点同 Hash 一样。这样就实现了页面无刷新情况下改变 URL，这也是前端路由得以实现的关键点！

接下来就只要监听浏览器 URL 改变的事件即可对页面展示内容进行切换，在 `history` 中的事件是 `popState`。在历史记录中切换时就会产生 `popState` 事件。各浏览器之间在实现上会有一定差异，根据不同浏览器做兼容即可。

以上就是 `history` 模式实现路由的基本思路。

注意，`pushState()` 和 `replaceState()` 不支持跨域。比如：

当打开 `https://www.zhihu.com?name=1`，在此域名下执行

```
window.history.pushState(null, null, "https://www.zhihu.com/name/orange");
```

这时控制台将会报如下错误提示：

```
VM462:1 Uncaught DOMException: Failed to execute 'pushState' on 'History': A history state object with URL 'https://www.zhihu.com/name/orange' cannot be created in a document with origin 'https://www.baidu.com' and URL 'https://www.baidu.com/'.
at <anonymous>:1:16
```

5.2.2 Hash 方式

Hash（哈希）也称作锚点，指的是 URL 中 # 号以后的字符。其本身用于页面定位，它可以配合 `id` 的元素显示在可视区域内。同样地，它可以改变浏览器 URL，同时做到不刷新页面。但依旧可以通过 `hashChange` 事件去监听其变化，从而进行页面展示内容的切换。

相比于 history API 模式，Hash 这种方式可以考虑到低版本浏览器，但有人认为浏览器 URL 上多一个#会不美观。这就要根据个人喜好进行选择了。

在 React 项目中，通常使用第三方路由库去使用路由，当然也可以不使用第三方库，简单示例如下。

Hash 方式实现路由示例：

```
import React from 'react'
import { render } from 'react-dom'
// 各个路由对应的组件
const Home = ()=> <div>Home</div>
const About = ()=> <div>About</div>
const Inbox = ()=> <div>Inbox</div>
const App = React.createClass({
  getInitialState() {
    return {
      route: window.location.hash.substr(1) // 获取浏览器 Hash 值，并存储在 state 中
    }
  },
  componentDidMount() {
    // 利用监听 Hash 事件去改变路由值
    window.addEventListener('hashchange', () => {
      this.setState({
        route: window.location.hash.substr(1)
      })
    })
  },
  render() {
    let Child
    // 根据 state 的 route 值来响应当前内容的动态展示
    switch (this.state.route) {
      case '/about': Child = About; break;
      case '/inbox': Child = Inbox; break;
      default: Child = Home;
    }
    return (
      <div>
        <a href="#/about">About</a>
          // 单击 a 标签会改变浏览器地址栏 URL，但不会刷新页面
        <a href="#/inbox">Inbox</a>
        <Child />
      </div>
    )
  }
})
React.render(<App />, document.body)
```

5.3 react-router 路由配置

react-router 是 React 中用来实现路由的第三方 JavaScript 库，也是基于 React 开发的。

它拥有简单 API 和强大的路由处理机制，如代码缓冲加载、动态路由匹配，以及建立正确的过渡处理。它可以快速地在应用中添加视图和数据流，保持页面展示内容和 URL 的同步。

5.3.1 react-router 的安装

react-router 3.x 安装：

```
npm install -save react-router
```

react-router4 版本之后，不再引入 react-router，react-router 4 安装：

```
npm install react-router-dom
```

5.3.2 路由配置

以 Webpack 作为项目的模块管理器为例，react-router 3.x 的配置示例代码如下：

```
// v3
import React from "react";
import ReactDOM from "react-dom";
import { Route, IndexRoute, hashHistory } from "react-router";
import App from "./app/containers/app/";
import Home from "./app/containers/home/";
import User from "./app/containers/user/";
ReactDOM.render(
  <Router history={hashHistory} />
  <Route path="/" component={App}>
    {/* 当 url 为/时渲染 Home */}
    <IndexRoute component={Home} /> // 首页
    <Route path="/user" component={User} />
  </Route>
</Router>,
  document.getElementById("root")
);
```

然后在 App 组件中，使用 this.props.children 属性配置路由的组件：

```
// App.js
const Inbox = React.createClass({
  render() {
    return (
      <div>
        {/* 渲染这个 child 路由组件 */}
        {this.props.children}
      </div>
    )
  }
})
```

上述代码中 Router 标签代表路由器。路由器中的 history 表示 react-router 路由的模式。

<Route>是 react-router 最重要的组件，它的职责是在其 path 属性与某个 location 匹配时呈现指定的视图，用于配置路由和组件的对应关系。

```
<Route path="/list" component={List} />
```

path 表示路由的路径，component 代表路由对应的页面组件。其中/代表应用的根路径，假设当前应用的访问地址是 <http://www.example.com/>，当用户在访问这个地址时，应用会加载 App 这个组件，并且实际上访问的地址会是 <http://www.example.com/#/>。当访问 <http://www.example.com/#/home> 时，会切换到 Home 组件。其他路由以此类推。

子路由也可以不写在 Router 组件内，可以写在 Router 组件的属性内，示例如下：

```
let routes = <Route path="/" component={App}>
  <Route path="/home" component={Home} />
  <Route path="/about" component={About} />
</Route>;
<Router routes={routes} history={hashHistory} />
```

5.3.3 默认路由

在 react-router 3.x 中可以指定默认路由（IndexRoute）。

react-router 3.x 中指定默认路由示例：

```
// v3
<Router>
  <Route path="/" component={App}>
    <IndexRoute component={Home} />
    <Route path="about" component={About} />
    <Route path="help" component={Help} />
  </Route>
</Router>
```

当用户在访问"/"时，App 组件被渲染，App 内部的 this.props.children 会去寻找 IndexRoute，将它作为最高层级的路由。如若不指定 IndexRoute，App 内部的 this.props.children 将为 undefined。

需要注意的是，在 react-router v4 中没有<IndexRoute>，而是通过<Switch>组件提供相似功能。

react-router 4.x 中指定默认路由示例：

```
// v4
const App = () => (
  <Switch>
    <Route exact path='/' component={Home} />
    <Route path='/about' component={About} />
    <Route path='/help' component={Help} />
  </Switch>
)
```

5.3.4 路由嵌套

react-router 可以实现路由的嵌套，嵌套路由被描述成一种树形结构。react-router 会深度遍历整个路由配置来匹配给出的路径。

在 react-router 4.0 以下的版本中，路由的嵌套可以放在<Route>中。

react-router 3.x 的路由嵌套示例：

```
// v3
<Route path="parent" component={Parent}>
  <Route path="child1" component={Child1} />
  <Route path="child2" component={Child1} />
</Route>
```

在 react-router 4.0 以上的版本中，路由嵌套书写方式发生了变化，子<Route>会由父<Route>呈现。

react-router 4.0 以上版本的路由嵌套示例：

```
// v4
<Route path="parent" component={Parent} />
const Parent = () => (
  <div>
    <Route path="child1" component={Child1} />
    <Route path="child2" component={Child1} />
  </div>
)
```

5.3.5 重定向

在 react-router 3.x 的版本中，如果要从一个路径重定向到另一个路径，可以使用<IndexRedirect>。

react-router 3.x 的重定向示例：

```
// v3
<Route path="/" component={App}>
  <IndexRedirect to="/AnotherApp" />
  <Route path="anotherApp" component={ anotherApp }>
</Route>
```

上述代码中，当用户访问"/"根路径时，会自动跳转到/AnotherApp。

而在 react-router 4.x 中，使用方式同样有所改变，改为<Redirect>。

react-router 4.x 的重定向示例：

```
// v4
import { Route, Redirect } from 'react-router';
<Route exact path="/" render={() => (
  loggedIn ? (
    <Redirect to="/homePage"/>
  ) : (
    <div>未登录</div>
  )
)} />
```

```

) : (
  <LoginPage/>
)
)}/>

```

上述代码中，如果用户已登录就跳转到 `homePage`；否则，跳到登录页面。

5.4 react-router 下的 history

`react-router` 中有 `history` 属性，用于监听浏览器地址栏的改变，并解析这个 URL 转化为 `location` 对象，从而匹配 `react-router` 配置的路由去渲染对应的视图。

`react-router` 中有 3 种形式：

- `browserHistory`;
- `hashHistory`;
- `createMemoryHistory`。

5.4.1 `browserHistory` 模式

`browserHistory` 是基于使用浏览器 `history API` 实现的，也是 `react-router` 应用推荐的路由模式。可以从 `react-router` 中引入使用：

```

import { browserHistory } from 'react-router'
render(
  <Router history={browserHistory} routes={routes} />,
  document.getElementById('app')
)

```

这种模式的优点是更像“真实的”URL，形如`/index/homePage`。这种模式的缺点就是当用户在子路由刷新或向服务器直接请求子路由，则会显示找不到，给出 404 报错。这时候就需要服务器去配置并处理这些路由访问，假如读者的服务器是 Nginx，可以使用 `try_files` 指令：

```

server {
  ...
  location / {
    try_files $uri /index.html
  }
}

```

当在服务器上找不到其他文件时，Nginx 服务器可以提供静态资源并指向 `index.html` 文件。

如果是 Apache 服务器，就创建一个 `.htaccess` 文件在文件根目录下：

```

RewriteBase /
RewriteRule ^index\.html$ - [L]

```

```
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . /index.html [L]
```

其他服务器处理方式类似，处理方式的实质都是一样的。

5.4.2 hashHistory 模式

hashHistory 是基于哈希 (#) 实现的。形如#/index/homePage?_k=adsis 。示例如下：

```
import { hashHistory } from 'react-router'
render(
  <Router history={hashHistory} routes={routes} />,
  document.getElementById('app')
)
```

5.4.3 createMemoryHistory 模式

createMemoryHistory 模式用于服务端渲染。它不会在地址栏被操作或读取，但会在内存中进行历史记录的存储。

它与其他两种模式不同的是，需要手动去创建它：

```
const history = createMemoryHistory(location)
```

5.5 react-router 路由切换

当 react-router 路由路径配置完之后，那么读者知道不同路由之间是如何跳转，以及不同路由之间是如何传参的吗？传统多页应用中一般使用 a 标签来做 URL 的跳转。但是在 react-router 中，则需要通过调用路由的 API 来完成不同路径之间的切换。下面针对路由切换进行讲解。

5.5.1 Link 标签

Link 是 react-router 中用于路由相互跳转的其中一种方法。其本质就是一个被处理过的<a>标签，它可以接收 Router 的状态。

Link 标签实现的路由切换示例：

```
render() {
  return (
    <div>
      <ul>
        <li><Link to="/">点击跳转首页</Link></li>
        <li><Link to="/login">点击跳转登录页</Link></li>
```

```

        </ul>
    </div>
)
}

```

Link 可以知道哪个 Route 的链接是激活状态，并可以为该链接添加 activeClassName 或 activeStyle 属性。这就使得当用户在 Tab 切换的时候，可以方便地设置激活时的样式展示。示例如下：

```

<Link to="/home" activeStyle={{color: 'red'}}>Home</Link>
<Link to="/about" activeStyle={{color: 'red'}}>About</Link>

```

或：

```

<Link to="/home" activeClassName="active">Home</Link>
<Link to="/about" activeClassName="active">About</Link>

```

 注意：如果链接到根路由"/"，要使用<IndexLink>。

5.5.2 history 属性

在 react-router 3.x 中，路由的跳转一般这样处理：

- 从 react-router 导出 browserHistory；
- 使用 browserHistory.push() 等方法进行路由跳转。

react-router 3.x 示例中的路由跳转示例：

```

import browserHistory from 'react-router';
...
browserHistory.push('/user');

```

在 react-router 4.x 中，不提供 browserHistory 等方法，而是通过使用高阶组件 withRouter 去使用 history 的方法实现路由跳转。

react-router 4.x 示例中的路由跳转示例：

```

// v4
import React, {Component} from 'react'
import {withRouter} from 'react-router-dom'
class User extends Component{
    constructor(){
        super()
    }
    linkTo(){
        this.props.history.push('/about')
    }
    render(){
        return(
            ...
        )
    }
}
export default withRouter(User);

```

5.5.3 传参

应用跳转的时候可能会涉及参数的传递，在 react-router 中传参也很简单。先通过 Route 的 path 进行配置，然后在 Link 的 to 属性中添加需要的参数，最后在跳转后的页面去获取。

示例 1：react-router 中，使用 this.props.params.query 获取参数。

```
<Router history={history}>
  <Route path="user/:id" component={User} />          // 路由中配置:id
</Router>
...
<Link to={{pathname: '/user/{id}' }} activeClassName="active">
  Click to userPage
</Link>
```

示例 2：react-router 中，使用 this.props.location.query 获取参数。

```
<Router history={history}>
  <Route path="user" component={User} />
</Router>
...
<Link to={{pathname: "/User", query:{id: id} }} activeClassName="active">
  Click to userPage
</Link>
```

示例 3：react-router 中使用 this.params.id 获取参数。

如果采用 history 跳转，那么在 user 页面中可以使用 this.params.id 获取参数。

```
<Router history={hashHistory}>
  <Route path='/user/:id' component={User}></Route>
</Router>
...
hashHistory.push("/user/888")
```

5.6 进入和离开的 Hook

Route 可以定义 onEnter 和 onLeave 两个 Hook，这两个钩子会在路由跳转确认时触发一次。这对于权限验证和路由跳转前数据持久化保存有很大作用。

5.6.1 onEnter 简介

onEnter Hook 会在即将进入路由时触发，会从最外层的父路由开始，直到最下层子路由结束。它可以接收 3 个参数：

```
type EnterHook = (nextState: RouterState, replaceState: RedirectFunction,
  callback?: Function) => any;
```

第1个参数 `nextState` 表示它接收的下一个 router state，第2个参数 `replaceState` function 用于触发 URL 的变化，第3个参数 `callback` 用于设置回调函数，以便于继续往下执行。

⚠ 注意：在 `onEnter` Hook 中使用 `callback` 会让变换过程处于阻塞状态，直到 `callback` 被回调。如果不能快速地回调，这可能会导致整个 UI 失去响应。

5.6.2 `onLeave` 简介

`onLeave` Hook 会在即将离开路由时触发，从最下层的子路由开始，直到最外层父路由结束。它是一个用户自定义的函数，用于在离开时被调用。有兴趣的读者可自行查阅相关资料，此处不再赘述。

第6章 React 的性能及性能优化

React 最神奇的亮点就是虚拟 DOM 和高效的 diff 算法。这让开发者无须过多担心性能上的问题，但并非没有性能问题，不佳的写法也会导致出现问题。本章一起来深入理解 diff 算法，以及如何做到书写高性能的组件。

6.1 diff 算法

在 React 中，UI 界面由组件构成。当前组件的状态发生变化时，真实 DOM 树需要重新更新渲染，而真实 DOM 树来源于 React 的虚拟 DOM 树。React 将虚拟 DOM 树转换为真实 DOM 树的最小计算过程称为调和（reconciliation），而 diff 算法便是调和的具体实现！

简单来说，diff 算法就是给定任意的两棵树从中找到最少的转换步骤，或者说是从上一个渲染转到下一个渲染的最少步骤。

6.1.1 时间复杂度和空间复杂度

说到最小生成树（tree）的算法，就不得不提到计算时间复杂度和空间复杂度这两个概念，这是衡量一个算法优劣的两个方面。

1. 时间复杂度

时间复杂度是一个函数，表示该算法运行耗费的时间。这是一个关于代表算法输入值的字符串长度的函数。时间复杂度常用大 O 符号表示，不包括这个函数的低阶项和首项系数。

一般情况下，算法中基本操作重复执行的次数是问题规模 n 的某个函数，用 $T(n)$ 表示，若有某个辅助函数 $f(n)$ ，使得 $T(n)/f(n)$ 的极限值（当 n 趋近于无穷大时）为不等于零的常数，则称 $f(n)$ 是 $T(n)$ 的同数量级函数。记作 $T(n)=O(f(n))$ ，称 $O(f(n))$ 为算法的渐进时间复杂度，简称时间复杂度。

随着模块 n 的增大，算法执行的时间的增长率和 $f(n)$ 的增长率成正比，所以 $f(n)$ 越小，

算法的时间复杂度越低，算法的效率越高。那么时间复杂度怎么计算呢？

如果算法执行时间不随着 n (n 称为问题的规模) 的增长而增长，执行时间只是一个比较大的常数，那么它的时间复杂度为 $O(1)$ ，举例：

```
i = 100000;
while(i--){
    printf("Bonjour")
}
```

这个算法执行 100 000 次，虽然次数多，但循环内执行时间是常数量，所以它的时间复杂度是 $O(1)$ 。

如果按数量级递增排列，常见的时间复杂度还有：对数阶 $O(\log_2 n)$ 、线性阶 $O(n)$ 、线性对数阶 $O(n \log_2 n)$ 、平方阶 $O(n^2)$ 、立方阶 $O(n^3)$... k 次方阶 $O(n^k)$ 、指数阶 $O(2^n)$ 。可以看出，随着问题规模 n 的不断增大，上述时间复杂度不断增大，算法的执行效率越低。

如果是嵌套循环：

```
count = 0;
for(i=1; i<=n; i++)
    for(j=1; j<=i; j++)
        for(k=1; k<=j; k++)
            count++;
...

```

这个算法中主要执行的是 `count++`，它的执行时间是常数值，但算法的时间复杂度是由嵌套层数最多的循环语句的语句频度决定的。每层每次分别执行的次数是： $[(n-1)]$ ， $[(n-1)+(n-2)\dots]$ ， $[(n-1-1)+(n-2-1)\dots]$ ，所以时间复杂度为 $O(n^3 + \text{剩余低次项}) \approx O(n^3)$ ，这里取的是运行次数函数的最高阶项。

2. 空间复杂度

一个程序的空间复杂度是指运行完一个程序所需内存的大小。利用程序的空间复杂度，可以对程序运行所需要内存的大小有个预先估计。一个程序执行时除了需要存储空间和存储本身所使用的指令、常数、变量和输入数据外，还需要一些对数据进行操作的工作单元和存储一些为现实计算所需信息的辅助空间。程序执行时所需存储空间包括以下两部分。

- 固定部分：这部分空间的大小与输入、输出数据的个数多少、数值无关，主要包括指令空间（代码空间）、数据空间（常量和变量）等所占用的空间，这部分属于静态空间。
- 可变空间：这部分空间主要包括动态分配的空间，以及递归栈所需的空间等，这部分的空间大小与算法有关。一个算法所需的存储空间用 $f(n)$ 表示， $S(n)=O(f(n))$ ，其中 n 为问题的规模， $S(n)$ 表示空间复杂度。

注意：大O符号（Big O notation）是用于描述函数渐近行为的数学符号。更确切地说，它是用另一个（通常更简单的）函数来描述一个函数数量级的渐近上界。在数学中，它一般用来刻画被截断的无穷级数，尤其是渐近级数的剩余项；在计算机科学中，它在分析算法复杂性方面非常有用。

本节理论引用来源：

https://grfia.dlsi.ua.es/ml/algorithms/references/editsurvey_bille.pdf;

<https://baike.baidu.com/item/%E7%A9%BA%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6/9664257?fr=aladdin>;

<https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6>;

<https://www.cnblogs.com/xiong233/p/6704846.html>.

6.1.2 diff 策略

现在读者知道，树的算法不是有了 React 才出现的算法，而是之前就有的。传统算法中（论文参考：https://grfia.dlsi.ua.es/ml/algorithms/references/editsurvey_bille.pdf）的做法是将每个节点一一对比，循环遍历所有子节点，然后判断子节点的更新状态，其复杂度为 $O(n^3)$ （ n 是树中节点的总数），效率很低。有多低呢？打个比方，假如有 1 000 个元素，那么需要计算 10 亿次左右。将此算法应用到计算机用于前端渲染，那么代价太大了。即便当下的 CPU 每秒执行约 30 亿条指令，以最高效实现效果也很差。而 React 将时间复杂度为 $O(n^3)$ 的算法直接转为 $O(n)$ ，砍掉了一部分算法，无疑这样肯定是有牺牲的。但如何保证其性能，它又是怎样实现的呢？

首先要理解 diff 的核心在于两点：对比和修改。React 中它基于两个假设实现了一个启发式的 $O(n)$ 的算法：

- 两个不同类型的元素将产生两个不同的树；
- 同一级的一组子节点，可以从中埋入一个 key 属性用于区分；

事实上，这些假设几乎能作用于所有实际用例。在此基础上 React 大胆采用了 3 种策略：

- DOM 节点跨层级操作特别少，所以可以忽略；
- 拥有相同类的两个组件会生成相似树形结构，拥有不同类的两个组件将会产生不同树形结构；
- 同一级一组子节点通过唯一 id（key）区分。

下面来具体介绍这 3 种策略的具体做法。

1. Tree Diff

Tree Diff 是两棵新旧虚拟 DOM 树按照层级的对应关系，把同一层级的节点遍历一遍，

即同层比较，这样就能快速找到有差异的地方。但有一个前提，它们得是同一父节点下的子节点进行比较，父节点不同也就无须比较下面的节点了，即“同层求异”，如图 6.1 所示。

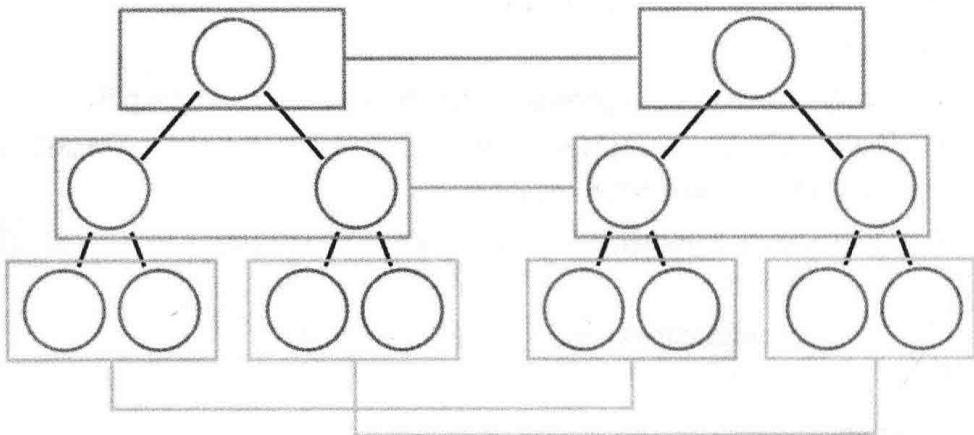


图 6.1 Tree Diff

Tree 的比较适用于界面 DOM 节点跨层级操作少的情形，这样就可以忽略不计层级带来的影响。它的分层比较，层级控制非常高效，当发现子节点不在了将会直接删除该节点以及它下面的所有子节点，也就是只需要遍历一遍。这种方式看似“暴力”，但的确是一种不错的方式。

2. Component Diff

React 构建的应用是以组件组合而成的，以组件为单位的差异对比策略也类似。

对于类型相同的组件，根据 Virtual DOM 树按照原来的策略继续比较 Tree 即可。

对于类型不同的组件，React 会将这个组件内部所有的子节点重新替换。而这个组件在 React 中被称为 dirty component。比如，组件 B 变为组件 C，虽然子组件内容相似，但一旦识别到父组件 B 和 C 不同，就直接删除 B 组件，重新创建 C 组件，如图 6.2 所示。

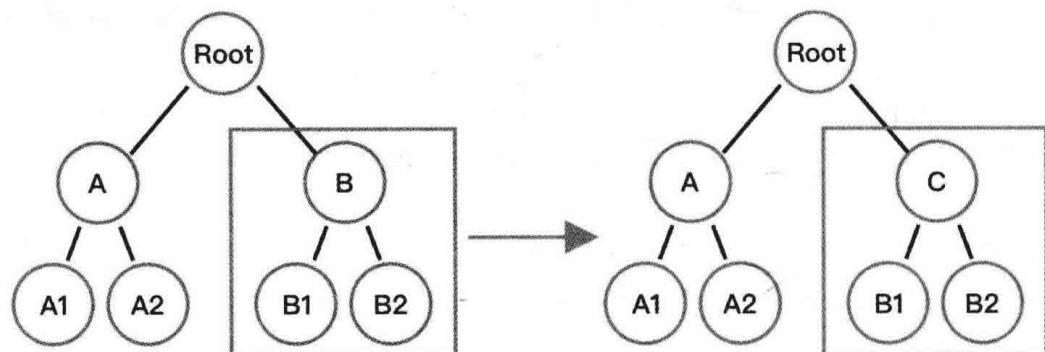


图 6.2 Component Diff

注意：dirty component 可以理解为组件最近一次发生改变，但还未重新渲染的组件。

3. Element Diff

React 在遇到类型相同的组件时，会继续对组件内部元素进行对比，检查内部元素异同，这就是 Element Diff。它可以进行插入、移动、删除这 3 种操作。源码 (<https://github.com/facebook/react/blob/f33f03e3572d11e6810f4ce110eb3af97cbd24a8/src/renderers/shared/stack/reconciler/ReactMultiChild.js#L33>) 如下：

```
function makeInsertMarkup(markup, afterNode, toIndex) {
  return {
    type: 'INSERT_MARKUP',
    content: markup,
    fromIndex: null,
    fromNode: null,
    toIndex: toIndex,
    afterNode: afterNode,
  };
}

function makeMove(child, afterNode, toIndex) {
  return {
    type: 'MOVE_EXISTING',
    content: null,
    fromIndex: child._mountIndex,
    fromNode: ReactReconciler.getHostNode(child),
    toIndex: toIndex,
    afterNode: afterNode,
  };
}

function makeRemove(child, node) {
  return {
    type: 'REMOVE_NODE',
    content: null,
    fromIndex: child._mountIndex,
    fromNode: node,
    toIndex: null,
    afterNode: null,
  };
}
```

下面举例说明，先来看这样一组同级元素，见图 6.3 所示。

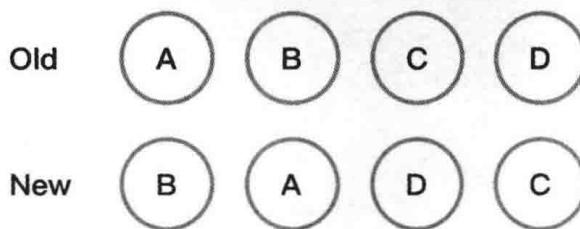


图 6.3 新老节点 diff

旧元素 A、B、C、D 发生变化需要排列为 B、A、D、C，当发现 B 不等于 A 时，则将 B 节点创建并插入至新组建，同时删除 A 节点，以此类推。即使有相同的节点，而且仅仅只是移动了位置，但还是需要删除并重写，无疑这种操作很繁琐低效。

在 React 中，它可以给每个同层节点设置一个唯一的 key，给它们做了标记如图 6.4 所示。同样的问题，当元素 A、B、C、D 发生变化需要排列为 B、A、D、C 时，diff 差异化对比后发现新旧节点存在相同的，则无须进行重新创建和删除，只需将旧的节点集合进行位置移动即可。

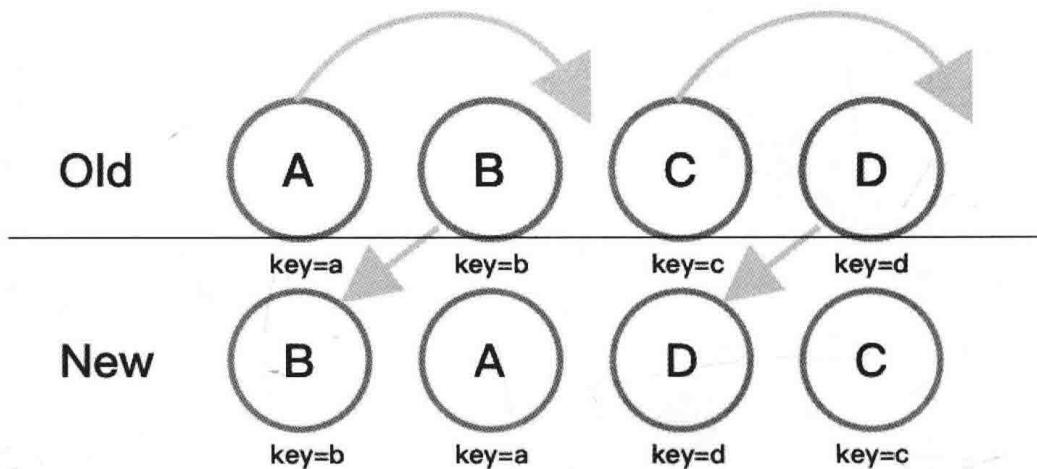


图 6.4 带 key 节点的 diff

其内部具体执行是，先将新的节点集合进行遍历循环，然后通过唯一标记 key 去老的节点集合中寻找是否有命中的标记，如果有，就执行移动操作。需要注意的是，与此同时当前节点在旧集合中索引的值必须小于新集合中当前节点的索引值 lastIndex，因为这样能节省更多不必要的操作从而节省时间，这样可以更加优化算法效率。源码（<https://github.com/facebook/react/blob/f3f03e3572d11e6810f4ce110eb3af97cbd24a8/src/renderers/shared/stack/reconciler/ReactMultiChild.js#L33>）参考如下：

```
// 对要呈现的标记进行更新，并在提供的索引处插入
function makeInsertMarkup(markup, afterNode, toIndex) {
  // NOTE: Null values reduce hidden classes.
  return {
    type: 'INSERT_MARKUP',
    content: markup,
    fromIndex: null,
    fromNode: null,
    toIndex: toIndex,
    afterNode: afterNode,
  };
}
// 将现有元素移动到另一个索引
function makeMove(child, afterNode, toIndex) {
  // NOTE: Null values reduce hidden classes.
  return {
```

```
type: 'MOVE_EXISTING',
content: null,
fromIndex: child._mountIndex,
fromNode: ReactReconciler.getHostNode(child),
toIndex: toIndex,
afterNode: afterNode,
};

}

//删除索引处的元素
function makeRemove(child, node) {
  // NOTE: Null values reduce hidden classes.
  return {
    type: 'REMOVE_NODE',
    content: null,
    fromIndex: child._mountIndex,
    fromNode: node,
    toIndex: null,
    afterNode: null,
  };
}

//设置节点的标记
function makeSetMarkup(markup) {
  // NOTE: Null values reduce hidden classes.
  return {
    type: 'SET_MARKUP',
    content: markup,
    fromIndex: null,
    fromNode: null,
    toIndex: null,
    afterNode: null,
  };
}

// 进行设置文本内容的更新
function makeTextContent(textContent) {
  // NOTE: Null values reduce hidden classes.
  return {
    type: 'TEXT_CONTENT',
    content: textContent,
    fromIndex: null,
    fromNode: null,
    toIndex: null,
    afterNode: null,
  };
}

// 将更新推送到队列
function enqueue(queue, update) {
  if (update) {
    queue = queue || [];
    queue.push(update);
  }
  return queue;
}

// 处理任何排队的更新
```

```

function processQueue(inst, updateQueue) {
  ReactComponentEnvironment.processChildrenUpdates(
    inst,
    updateQueue,
  );
}

// 用新的子节点渲染
updateChildren: function(nextNestedChildrenElements, transaction, context) {
  // Hook used by React ART
  this._updateChildren(nextNestedChildrenElements, transaction, context);
},
_updateChildren:function(nextNestedChildrenElements,transaction,context){
  var prevChildren = this._renderedChildren;
  var removedNodes = {};
  var mountImages = [];
  // 获取新的子元素数组
  var nextChildren = this._reconcilerUpdateChildren(
    prevChildren,
    nextNestedChildrenElements,
    mountImages,
    removedNodes,
    transaction,
    context
  );
  // 如果不存在 nextChildren 并且不存在 prevChildren, 则不需要 diff
  if (!nextChildren && !prevChildren) {
    return;
  }
  var updates = null;
  var name;

  // nextIndex 将为 nextChildren 中的每个子项递增,
  // 但 lastIndex 将是 prevChildren 中访问的最后一个索引
  var nextIndex = 0;
  var lastIndex = 0;
  // nextMountIndex 将为每个新安装的子项递增
  var nextMountIndex = 0;
  var lastPlacedNode = null;
  for (name in nextChildren) {
    if (!nextChildren.hasOwnProperty(name)) {
      continue;
    }
    var prevChild = prevChildren && prevChildren[name];
    var nextChild = nextChildren[name];
    if (prevChild === nextChild) {
      // 同一个引用, 说明使用的是同一个 component, 所以需要做移动的操作
      // 移动已有的子节点
      // 根据 nextIndex, lastIndex 决定是否移动
      updates = enqueue(
        updates,
        this.moveChild(prevChild, lastPlacedNode, nextIndex, lastIndex)
      );
      // 更新 lastIndex
      lastIndex = Math.max(prevChild._mountIndex, lastIndex);
    }
  }
}

```

```

// 更新 component 的.mountIndex 属性
prevChild._mountIndex = nextIndex;
} else {
  if (prevChild) {
    // 通过卸载来取消设置_mountIndex 之前更新 lastIndex
    lastIndex = Math.max(prevChild._mountIndex, lastIndex);
  // 下面的 removedNodes 循环实际上会删除子节点
  }
  // 添加新的子节点在指定的位置上
  updates = enqueue(
    updates,
    this._mountChildAtIndex(
      nextChild,
      mountImages[nextMountIndex],
      lastPlacedNode,
      nextIndex,
      transaction,
      context
    )
  );
  nextMountIndex++;
}
// 更新 nextIndex
nextIndex++;
lastPlacedNode = ReactReconciler.getHostNode(nextChild);
}

// 移除不存在的旧子节点，和旧子节点和新子节点不同的旧子节点
for (name in removedNodes) {
  if (removedNodes.hasOwnProperty(name)) {
    updates = enqueue(
      updates,
      this._unmountChild(prevChildren[name], removedNodes[name])
    );
  }
}
}

...
_mountChildAtIndex: function(
  child,
  mountImage,
  afterNode,
  index,
  transaction,
  context) {
  child._mountIndex = index;
  return this.createChild(child, afterNode, mountImage);
},
// 移除子节点方法
_unmountChild: function(child, node) {
  var update = this.removeChild(child, node);
  child._mountIndex = null;
  return update;
},

```

React diff 算法小结如图 6.5 所示。

- Tree Diff: 采用分层求异的策略，将新旧两棵 DOM 树按照层级对应的关系进行对比，这样只需要对树进行一次遍历，就能够找到哪些元素是需要更新的。
- Component Diff: 查看两个组件的类型是否相同。如果类型不同，则需要更新，更新时先把旧的组件删除，再创建一个新的组件插入之前删除的位置。类型相同时，暂时不需要更新。
- Element Diff: 通过设置唯一 key 值，对元素 diff 进行优化。（组件类型相同时看内部元素）元素发生了改变，则找到需要修改的元素，有针对性进行修改。

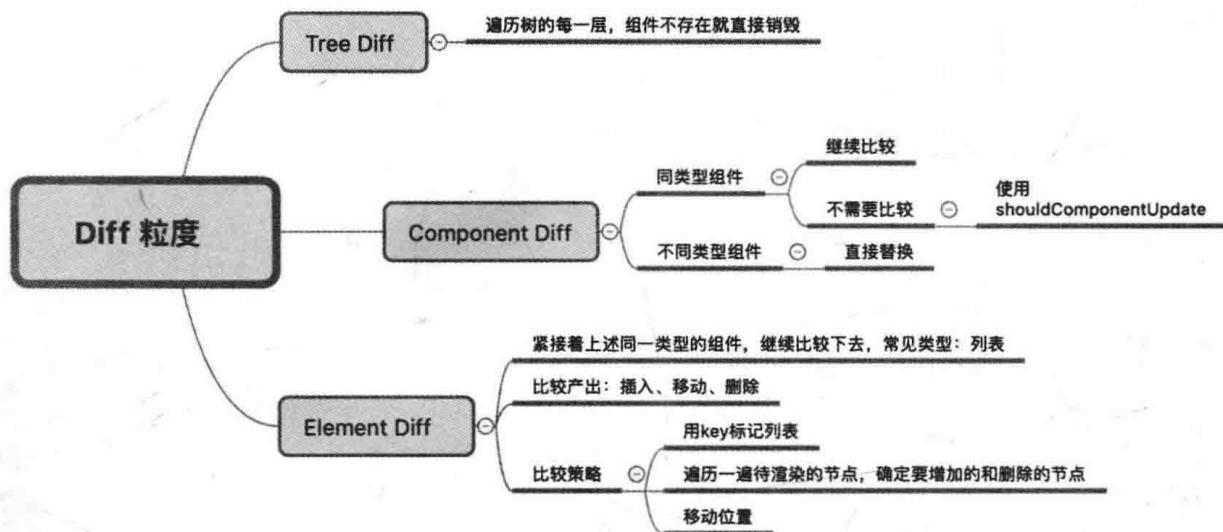


图 6.5 diff 算法小结

6.1.3 key 属性

key 是 React 中一个特殊的属性，不能被开发者获取，而是仅仅用于 React 本身。比如无法获取某个设置 key 节点的 key 值。前面章节已经提到了，在 Element Diff 中采用设置唯一 key 来标记同级节点，用以优化 diff 算法，这是 key 的作用。它就像一个人的身份证，唯一并且稳定，能用这个 id 来作为身份标识。不稳定的 key（比如 Math.random() 函数的结果）可能会让 DOM 节点重渲染时产生非必要的重新创建，造成极大的性能损失。

key 属性的应用示例：

```

this.state = {
  users: [{id:1, name: '张三'}, {id:2, name: '李四'}, {id: 3, name: "王五"}, 
  {id: 4, name: "赵六"}]
}
render()
return(
  <div>

```

```

<h2>User list</h2>
{this.state.users.map(item => <div key={item.id}>{item.name}</div>)}
</div>
)
);

```

假如上述代码中不写 key，React 会默认以数组的 index 作为 key，但是控制台将会报警告，如图 6.6 所示。

```

Warning: Each child in an array or iterator should have a unique "key" prop. react.development.js:225
Check the render method of `App`. See https://fb.me/react-warning-keys for more information.
in ListItems (created by App)
in App

```

图 6.6 不写 key 的 warning

key 的应用场景最多的是由数组动态创建子组件的情景，如果列表数据只是纯展示 map，可以使用 index 作为 key。

列表数据中 key 应用示例：

```

this.state = {
  menu: ['home', 'news', 'about us']
}
render() {
  return(
    <div>
      <h2>Menu</h2>
      {this.state.menu.map((name, index) => <div key={index}>{name}</div>)}
    </div>
  )
}

```

如果不是动态生成的列表，可以不需要 key，如：

```

render() {
  return(
    <div>
      <h2>Menu</h2>
      <div>home</div>
      <div>news</div>
      <div>about us</div>
    </div>
  )
}

```

假如读者在动态生成的组件内使用索引 index 作为 key，虽然不会报警告，但是这种做法是不对的，因为动态生成的节点每次索引会更新，此时的 key 不是唯一也不是稳定的标识。

6.2 组件重新渲染

React 的组件状态发生变化时，就会触发浏览器的重绘（reflow）或重排（repaint），

而重绘和重排是影响网页性能的最大因素。得益于 React 的 Virtual DOM 和 diff 算法，才使得浏览器尽可能地减少重绘和重排。

使用 diff 算法可以把新老虚拟 DOM 进行比较，如果新老虚拟 DOM 树相等，则不重新渲染，否则就重新渲染。这个过程又被称为“子集校正”（reconciliation）。DOM 操作是很耗性能的，所以要提高组件的性能就应该尽可能地减少组件的重新渲染。

在 React 中，任何时刻调用组件的 `setState()` 方法，React 不是立即对其更新，而是先将其标记为 Dirty（脏）状态，如图 6.7 所示。也就是说，`setState()` 之后变更的状态不会立即生效，React 使用了事件轮询对变更做批量处理绘制。

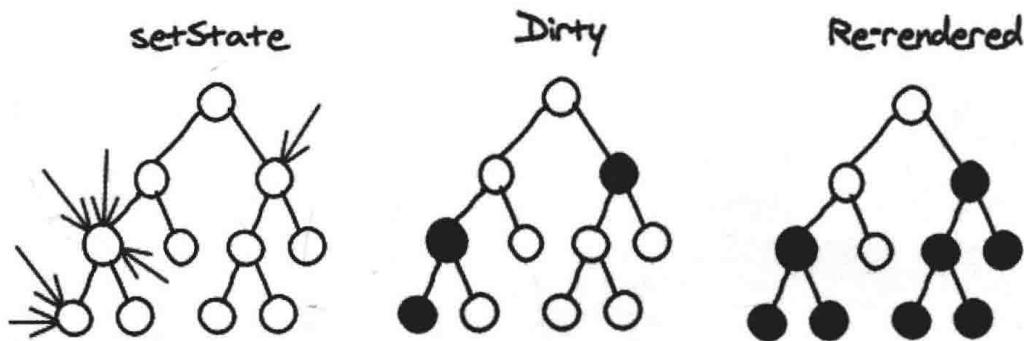


图 6.7 当调用组件的 `setState()` 时，React 将其标记为 Dirty 状态

当事件轮询结束后，React 会将标有 Dirty 的组件及其子节点进行重绘。所有后代节点的 `render()` 方法都会被调用，即使这些后代节点没有任何变化。虽然看起来这样的效率很低，但是现代浏览器对于 JavaScript 处理这类的操作速度已经非常快了。由于这些过程都发生在计算机内存中，而不是真实 DOM，所以不用担心其性能会影响渲染的速度。在前面章节读者已经知道，书写 `render()` 时返回的都是 JSX 或 `ReactElement`。实际上，它返回的只是一个用于描述 DOM 结构的普通 JavaScript 对象。示例代码：

```
{
  type: 'button',
  props: { className: 'btn-primary' },
  children: [
    {
      type: 'em',
      children: 'Confirm'
    }
  ]
}
```

React 就是用上述代码来描述界面中的标签，当状态改变时，就与前一次渲染的对象做比较，如果数据没有变化，就不用更新界面中的 DOM。因此对于 JavaScript 来说，对比这类 JavaScript 对象的成本不算高，计算速度已经足够快了。这好比不足 1 毫秒的渲染时间在提升了 10 倍的速度后依旧还是不足 1 毫秒！

不过 React 还是提供了一种方法对这些后代节点进行优化，可以阻止后代节点树的重绘，那就是重写 `shouldComponentUpdate()` 生命周期。它于组件重新渲染开始前触发，

这个函数默认返回 true，当然如果设置为 false，当前组件将不去重渲染，从而提升更新速度。

```
shouldComponentUpdate(nextProps, nextState) {
  return false;
}
```

可能读者会碰到这样的场景，只想让组件的某个特定的状态值改变时才去做重新渲染。那么可以这样做：

```
class CounterButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 1 };
  }
  shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color !== nextProps.color) {
      return true;
    }
    if (this.state.count !== nextState.count) {
      return true;
    }
    return false;
  }
  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({ count: state.count + 1 }))}
      >
        Count: {this.state.count}
      </button>
    );
  }
}
```

在上述代码中，`shouldComponentUpdate()`只检查 `props.color` 和 `state.count` 的变化。如果这些值没有变化，组件就不会更新。

注意：当使用 `shouldCompoentUpdate()`方法没有对页面造成能用肉眼观察到的速度提升时，就不要过早优化。这不仅会增加代码的复杂度，也可能造成一些衍生出来的其他 bug，并且难以排查。`shouldCompoentUpdate()`相当于是 React 的一扇“后门”，它是保证性能的紧急出口，所以一般不会轻易使用它。只有当读者真正需要的时候，再去使用它。一般情况下，不建议随意使用。

从上述对于 `render()`方法和 `shouldCompoentUpdate()`方法在组件更新时的描述中可以得知，组件的属性在传递时只需传递其需要的 `props` 即可，传的过多或过深会加重 `shouldCompoentUpdate()`内数据对比的负担；复杂的页面不要在一个组件内写完，尽量拆分成组件；适当的组件拆分有利于复用和组件性能优化。

6.3 PureRender 纯渲染

从上一节中可以得知，每当组件状态改变时都会触发其后代组件重渲染，并且必定会触发各组件的 `shouldComponentUpdate()` 这个生命周期，因此可以通过在这个生命周期中进行新旧状态对比看是否有变化来判断是否重新渲染当前组件。但如果要做到充分比较就得进行深比较，这是非常昂贵的操作：

```
shouldComponentUpdate(nextProps, nextState) {
  return isEqual(this.props, nextProps) &&
    isEqual(this.state, nextState);
}
```

出于此考虑，React 在早期就为开发者提供了一个第三方插件 `react-addons-pure-render-mixin`（现在官方使用的是 `react-addons-shallow-compare`）。其原理就是重写了 `shouldComponentUpdate()` 这个生命周期，但它内部的做法是让新旧 `props` 进行浅比较（shallow comparison），`PureRender` 只对对象进行了引用的比较，而没有作值的比较。

浅比较源码：

```
export default function shallowEqual(objA, objB) {
  if (objA === objB) { // 比较其引用，而没有对值进行比较
    return true;
  }
  if (typeof objA !== 'object' || objA === null ||
      typeof objB !== 'object' || objB === null) {
    return false;
  }
  var keysA = Object.keys(objA);
  var keysB = Object.keys(objB);
  if (keysA.length !== keysB.length) {
    return false;
  }
  // Test for A's keys different from B.
  var bHasOwnProperty = Object.prototype.hasOwnProperty.bind(objB);
  for (var i = 0; i < keysA.length; i++) {
    if (!bHasOwnProperty(keysA[i]) || objA[keysA[i]] !== objB[keysA[i]]) {
      return false;
    }
  }
  return true;
}
```

`PureRender` 的使用：

```
import PureRenderMixin from 'react-addons-pure-render-mixin'; // ES 6
class FooComponent extends React.Component {
  constructor(props) {
    super(props);
    this.shouldComponentUpdate = PureRenderMixin.shouldComponentUpdate.
```

```

    bind(this);
}
render() {
  return <div className={this.props.className}>foo</div>;
}
}

```

在React 15.3.0中加入了React.PureComponent，用来替换react-addons-pure-render-mixin：

```

export class FooComponent extends React.PureComponent {
  render() {
    return <div className={this.props.className}>foo</div>;
  }
}

```

简单概括， PureComponent 和 shouldComponentUpdate 的关注点是 UI 界面是否需要更新，而 render 的关注点在于虚拟 DOM 的 diff 的过程。 PureComponent 是 Component 的一个优化组件，通过减少不必要的 render 操作的次数，从而可以提高界面的渲染性能。

注意：React.PureComponent 中的 shouldComponentUpdate 只会对对象进行浅对比。如果对象包含复杂的数据结构，它可能会因深层的数据不一致而产生错误的判断。当组件只拥有简单的 props 和 state 时，才能使用 PureComponent，或者知道深层的数据结构已经发生改变时，可以使用 forceUpdate()。

6.4 Immutable持久性数据结构库

Immutable（持久性数据结构库）是近期 JavaScript 中伟大的发明，它可以让项目性能得到极大的提升，本节将带领读者了解 Immutable，以及如何在项目中使用它提升组件的渲染性能。

6.4.1 Immutable的作用

JavaScript 中对象是可变的（Mutable），引用数据类型中新的引用对象可以改变原始对象。例如：

```

const a = { foo: 'bar' };
const b = a;
b.foo = 'baz';
a === b; // true

```

虽然 b 改变了，但是由于它与 a 引用的是同一个对象，所以它们是相等的。虽然这种做法可以节约内存，但当应用复杂之后就会造成一定的安全隐患。此时 Mutable 带来的优点会得不偿失！为应对这个问题，可以使用深复制或浅复制，但这种做法会造成 CPU 和内存的浪费。

而 Immutable 的出现能够很好地处理这些问题，它存在的意义就是弥补了 JavaScript 没有不可变数据结构的问题。

Immutable.js 是 Facebook 工程师 Lee Byron 花了三年时间打造的。其本质就是 JavaScript 的持久化数据结构的库（Persistent Data Structure），并且数据结构和方法非常丰富，整个包只有 16KB 大小。

Immutable 数据一旦创建就不能被修改。数据对象的任何修改都会返回一个新的 Immutable 对象，同时原来的对象依旧可用且保持不变，这就是持久化数据结构。为了避免 deepCopy 把所有节点都复制一遍带来的性能损耗，Immutable 使用了 Structural Sharing（结构共享），即如果对象树中一个节点发生变化，只修改这个节点和受它影响的父节点，其他节点则进行共享。

虽然通过深复制和浅复制也能解决上述问题，但与 Immutable 相比，其性能较差。比如，每次深复制都要把整个对象递归复制一份，但 Immutable 的实现却不是这样，而是有点像链表。当修改一个节点后，它能把旧节点的父子关系转移到新节点上，这样以来性能就能得以提升。使用 Immutable 后，当橙色节点的 state 变化后，不会再渲染树中的所有节点，而是只渲染中间图中蓝色的部分，如图 6.8 所示。

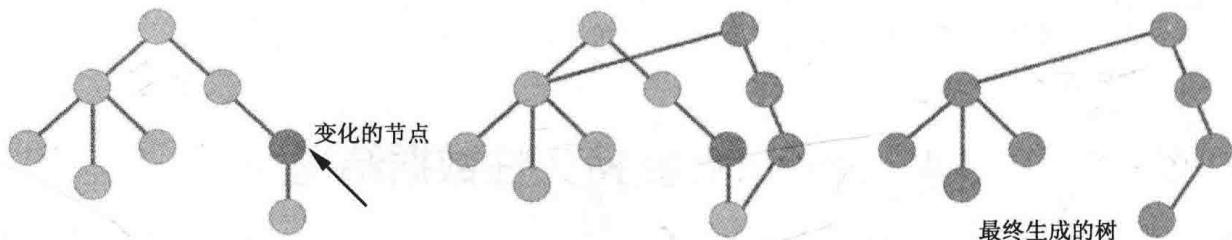


图 6.8 Immutable

当然，Immutable 的强大远不止于此。Immutable.js 提供了如下几种数据类型：

- **List:** 有序索引集，类似 JavaScript 中的 Array。
- **Map:** 无序索引集，类似 JavaScript 中的 Object。
- **OrderedMap:** 有序的 Map，根据数据的 set() 进行排序。
- **Set:** 没有重复值的集合。
- **OrderedSet:** 有序的 Set，根据数据的 add() 进行排序。
- **Stack:** 有序集合，支持使用 unshift() 和 shift() 进行添加和删除操作。
- **Record:** 一个用于生成 Record 实例的类。类似于 JavaScript 的 Object，但是只接收特定字符串为 key，具有默认值。
- **Seq:** 序列，但是可能不能由具体的数据结构支持。
- **Collection:** 是构建所有数据结构的基类，不可以直接构建。

Immutable.js 拥有非常全的 map, filter, groupBy, reduce 和 find 等函数式操作方法，而且其 API 与 Object 或 Array 是类似的。详情见官方文档：<http://facebook.github.io/>

immutable-js/docs/#/。

安装：

```
npm install immutable
```

简单示例：

```
const { Map } = require('immutable')
const map1 = Map({ a: 1, b: 2, c: 3 })
const map2 = map1.set('b', 50)
map1.get('b') + " vs. " + map2.get('b') // 2 vs. 50
```

6.4.2 Immutable 的优缺点

1. 优点

Immutable 的优点主要体现于数据的不可变，这个特征带来的优点如下：

(1) 降低 Mutable 带来的复杂度

Immutable 可以降低 Mutable 带来的复杂度，由于数据是不可变的，所以可以放心地对对象进行任何操作。

(2) 节省内存

虽然引用赋值能节省内存，但当应用复杂之后，可变状态往往会变得难以预测和维护，这会导致很多意想不到的问题。Immutable 使用了 StructureSharing 会尽量复用内存，甚至以前使用的对象也可以再次被复用。没有被引用的对象会被垃圾回收。

(3) 并发安全

由于 JavaScript 是单线程运行的，因此在做并发的时候数据会改变，这很难处理。但使用了 Immutable 就能解决这个问题。因为数据是不会改变的。

(4) 数据易追溯

由于每次数据都是不一样的，所以可以轻易地追溯历史数据。这一点就能让开发者很方便地实现数据变更的撤销之类的功能。

(5) 函数式编程

Immutable 本身就是函数式编程的概念，函数式编程利于组件开发。函数式编程关心数据的映射，同样的输入必定得到同样的输出。

2. 缺点

缺点是需要一定的学习成本和额外引入的静态资源，并且其 API 和原生对象类似，会产生混淆。

6.4.3 Immutable 和原生 JavaScript 对象相互转换

1. 原生 JavaScript 转换为 Immutable

`fromJS()` 用于将一个原生 JS 数据转换为 `Immutable` 类型的数据，如将原生 `Object` 转换为 `Map`，原生 `Array` 转换为 `List`。示例：

```
Immutable.fromJS( value, converter )
```

`value` 是要转变的数据，`converter` 是要做的操作。第 2 个参数可选，默认情况会将数组转换为 `List` 类型，将对象转换为 `Map` 类型，其余不做操作。

将原生 `Object` 转为 `Map`：

```
Immutable.Map( {} )
```

将原生 `Array` 转换为 `List`：

```
Immutable.List( [] )
```

`Immutable.fromJS()` 的好处在于它会嵌套递归执行转换，但 `Map` 与 `List` 不支持深层嵌套转换。

2. Immutable 转换为原生 JavaScript

`toJS()` 方法用于将一个 `Immutable` 数据转换为 `JavaScript` 类型的数据。它会自动判别 `Map` 与 `List` 并转换为原生 `Object` 与 `Array`。

```
ImmutableDate.toJS( )
```

6.4.4 Immutable 中的对象比较

1. 值比较

`is()` 用于将两个对象进行“值比较”。示例：

```
Immutable.is( map1, map2 )
```

和 `JavaScript` 中对象的比较不同，`JavaScript` 中比较两个对象比较的是地址。但是在 `Immutable` 中采用的是字典树（trie）数据结构来存储，比较的是这个对象 `hashCode` 或 `valueOf`，只要两个对象的 `hashCode` 相等，那么值就是一样的。这种算法避免了深度遍历，所以性能非常不错。

2. 引用比较

引用比较中，比较的是内存地址，所以速度非常快，`Immutable` 中示例如下：

```
const map1 = Immutable.Map({a: 1, b: 2, c: 3});
const map2 = Immutable.Map({a: 1, b: 2, c: 3});
map1 === map2; // false
```

6.4.5 Immutable 与 React 配合使用

在 React 中配合使用 Immutable.js 可以更方便、更安全地进行开发，并且这是一个独立的库，无论在哪个框架中都能使用。

在 React 中一旦组件的状态发生变化，就会触发后代组件中 `render()` 方法，从而进行虚拟 DOM 的 diff 算法进行比较差异，即使后代组件中的状态没有更改，也会触发此操作。从这一点上，无疑会造成性能上的浪费。虽然 React 留了一个 `shouldComponentUpdate()` 生命周期来控制是否触发虚拟 DOM 的 diff，但其本质还是将对象进行浅比较，如果碰到深层次的数据结构，那就不顶用了。

因此，这个时候就是 `Immutable.js` 发力的时候！

`Immutable` 提供了简洁高效的判断数据是否变化的方法，只需 `==` 和 `is` 比较就能知道是否需要执行 `render()`，而这个操作几乎是零成本，可以极大提高性能。修改后的 `shouldComponentUpdate` 是这样的：

```
import { is } from 'immutable';
shouldComponentUpdate: (nextProps = {}, nextState = {}) => {
  const thisProps = this.props || {}, thisState = this.state || {};
  if (Object.keys(thisProps).length !== Object.keys(nextProps).length ||
      Object.keys(thisState).length !== Object.keys(nextState).length) {
    return true;
  }
  for (const key in nextProps) {
    if (thisProps[key] !== nextProps[key] || !is(thisProps[key], nextProps[key])) {
      return true;
    }
  }
  for (const key in nextState) {
    if (thisState[key] !== nextState[key] || !is(thisState[key], nextState[key])) {
      return true;
    }
  }
  return false;
}
```

React 中可以把 `state` 当作 `Immutable`，可以这么做：

```
import { Map } from "immutable"; // 引入 immutable
export class App extends React.Component {
  constructor(props) {
    super(props);
```

```
    this.state = {
      userName: Map({firstName: "Zhang", lastName: "San"})
        // 设置 state 为 immutable
    };
}

updateState({target}) {
  // 创建一个新的 immutable 对象存入 state
  let userName = this.state.userName.set(target.name, target.value);
  this.setState({userName});
}
}
```

那么，是否必须得在 React 中使用 Immutable 呢？这取决于组件的 state 有多大、多复杂。如果组件 state 仅仅用于处理 API 返回的数据或者处理静态数据，这时候就没有必要使用 immutable；如果组件需要涉及大量 state 并且对象深层嵌套，那么使用 Immutable 是非常适合的。在这类场景中，使用 Immutable 最大的好处在于其高效。结构共享（Structural Sharing）的数据能让应用在处理数据时变得非常快，避免了维护那些大量让人头疼的数据问题。

第 7 章 React 服务端渲染

前面章节中的示例代码都是基于客户端渲染（CSR）的。随着富客户端应用的发展，各种交互操作变得越来越复杂，对于用户的体验要求也变得越来越高。虽然单页应用解决了很多问题，但也伴随出现了其他问题。比如客户端静态资源加载完成之前的间隙会有白屏出现等。因此出现了对应的解决方案——服务端渲染，本章将带领读者针对服务端渲染内容一探究竟。

7.1 客户端渲染和服务端渲染的区别

很多读者不清楚什么是客户端渲染（Client Side Rendering, CSR）和服务端渲染（Server Side Rendering, SSR），也不清楚它们的区别。因此在介绍服务端渲染之前，先来了解一下什么是客户端渲染和服务端渲染。

服务器端渲染，也即后端渲染。在早期，Web 是由 HTML 和 CSS 构建的静态界面，没有太多的交互，所有的用户行为由服务端来创建和提供。呈现在用户面前的是经过浏览器解析的 HTML 文件，而这些 HTML 文件是由服务端的模板文件生成，如 JSP 和 PHP 等。这些模版的核心设计理念就是在 HTML 文件内放占位符，然后由服务端逻辑替换成真实数据，最后由浏览器呈现给用户。当前依旧有大量的企业网站采用这种“老式”的写法，单击一次，页面会刷新一次，每一次刷新都是在向后台请求新的页面数据。它的好处就是前端只需将 HTML 进行展示，耗时少，利于 SEO；弊端就是占用服务器运算资源，网络传输的数据量大。这可以理解为模板式的服务端渲染。

与早期“模板式”的服务端渲染不同，当下所说的服务端渲染的主要用途是做网页性能加速和搜索引擎优化，服务端渲染无须等待 JS 文件下载执行的过程。它的优势还在于有了中间层（node 层）为客户端发起请求，并由 node 渲染页面，更易于维护，并且服务端和客户端可以共享某些代码。

客户端渲染，也即前端渲染。与服务端渲染不同的是，用户的 HTTP 请求拿到的不是渲染后的网页，而是由 HTML 和 JavaScript 组成的应用。显示的数据是在应用启动之后运行的逻辑，浏览器变成了应用的执行环境，而后端则变成了给前端页面展示服务的数据接口提供者。前端渲染得益于 JavaScript 的兴起，还有 AJAX 的出现，以及后来出现的前、

后端分离等。前、后端交互只需通过约定好的 API 交互，让后端专注于逻辑开发，让前端专注于 UI 开发。它的好处是让网络数据传输量变得更小，从而减轻服务器的压力。弊端是前端耗时增多，并且还不利于 SEO（后端渲染爬虫能看到完整的源码，前端渲染爬虫看不到完整源码）。

从本质上来说，无论是客户端渲染还是服务端渲染，都是一样的。它们做的工作都是将数据渲染进一个固定格式的 HTML 代码中，然后在页面上展示。两端渲染各有利弊，要考虑的是不同业务场景下用哪一端去渲染才是最佳选择。

那么该如何选择使用客户端渲染还是服务端渲染呢？

比如，要做一个注重 SEO 的网站，展示内容占大比重，交互场景很少，那么无疑服务端渲染是最好的选择。如果要做一个后台管理界面，增、删、改、查等业务场景较多，那么适合使用客户端渲染。这样能让后端专注于处理数据和提供接口，让前端专注于交互和 UI，这样分工明确，也便于以后维护。

“成也萧何，败也萧何”，使用了服务端渲染就意味着作为读者得去学习一门后端语言（Node.js），对于企业来说，需要聘用一个既会前端又会 Node 的前端工程师，成本较高。但从另一角度来说服务端和客户端可以共享某些代码。笔者认为，如果仅仅是为了解决首屏加载速度问题，随着近几年浏览器引擎的不断换代和 5G 网络的即将普及，这些都将不会是问题。并且 SEO 也有其它解决方案。目前来看，除了强势的前端团队，小型企业还是很少使用服务端渲染的。

值得注意的是，纵观当下绝大多数互联网开发团队，所有工作在早期全部由后端负责完成，到前、后端（岗位职责）的分离，又到服务端渲染让前端“有机会”介入后端这一系列的演变过程，可以思考一下，前、后端的分离在今后是否又会合二为一？或者职责分离，但岗位应该由一人完成呢？这些问题留给读者思考。

与客户端渲染不同，服务端渲染还有很多知识点需要掌握，本章首先了解服务端渲染所需要的知识点，然后用一个简单的实战训练来完成本章的介绍。

7.2 在 React 中实现服务端渲染

本节将介绍服务端的基础知识，各基础知识点之间没有直接的关联性，读者可以跳过已经掌握的相关知识点。

7.2.1 为何需要服务端渲染

使用服务端渲染主要有以下 3 个优点：

- 利于 SEO，让搜索引擎更容易读取页面内容；
- 让首屏渲染速度更快，无须等待 JS 文件下载执行的过程；

- 更易于维护，服务端和客户端代码共享。

7.2.2 服务端渲染中的 API

先来重点介绍 `renderToString()` 和 `renderToStaticMarkup()` 这两个方法，这两个 API 是服务端渲染的前提基础。它们是 React 用于在服务端渲染中使用的 API，可以配合与 `react-router` 和 `Redux` 一起使用进行首屏渲染。

`renderToString()` 和 `renderToStaticMarkup()` 的作用是将 Virtual DOM 渲染成 HTML 字符串。这两个 API 与 `render()` 一样，都是由 `react-dom` 提供的，但与 `render()` 不同的是，`renderToString()` 和 `renderToStaticMarkup()` 是在 `react-dom/server` 内的。引用如下：

```
import render from 'render';
import {renderToString} from 'react-dom/server';
import {renderToStaticMarkup} from 'react-dom/server';
```

从前面章节已经知道，`render()` 方法接受两个参数，即 `render([react element], [DOM node])`，但在服务端渲染中这两个 API 只接受一个参数，最终返回一段 HTML 字符串：

- `renderToString(react element)`;
- `renderToStaticMarkup(react element)`。

`renderToString()` 方法将 React 组件转换为 HTML 字符串，生成 HTML 的 DOM 渲染时带 `data-reactid` 属性。

`renderToStaticMarkup()` 同样是将 React 组件转换为 HTML 字符串，但生成的 HTML 的 DOM 不会携带 `data-reactid` 属性，可以减少 I/O 传输流量。可以认为它就是一个简化版的 `renderToString()`，当应用为静态文本时使用为宜。

开发中用到比较多的是 `renderToString()` 方法。在服务端，通过这个方法将虚拟 DOM 渲染为 HTML 字符串，然后传到客户端，客户端再做一些事件绑定等操作。在这个流程中，为了保证 DOM 结构的一致性，React 通过 `data-react-checksum` 来做检测。除了在服务端产生“干瘪”的字符串之外，还会额外产生一个 `data-react-checksum` 值，客户端再对这个值进行校验，如果与服务端一致，客户端会进行事件绑定；否则，React 会丢弃服务端返回的 DOM 再去重新渲染。

7.2.3 渲染方法

在 React 15.x 中，服务端渲染的客户端代码与普通写法没有区别，是调用 `ReactDOM.render()` 方法把虚拟 DOM 转换为真实 DOM 后渲染到界面上。

React 15.x 中的渲染方式示例：

```
import React from "react";
import { render } from "react-dom";
// HomePage 组件来自 client/components/homePage/index.js
```

```

import HomePage from "./client/components/homePage/index.js";
render(
  <HomePage />,
  document.getElementById("app")
);

```

要注意的是，React 15.x 中的 ReactDOM.render() 能利用 data-react-checksum 作为标记来复用 ReactDOMServer 的渲染结果，再根据 data-reactid 属性进行事件绑定。

在 React 16.x 版本中，ReactDOMServer 渲染内容移除了 data-react 属性，相当于直接使用服务端渲染的 HTML 结构。因此也就带来了问题，没有了 data-react-checksum 如何判断是否能复用呢？别急，Reactv 16.x 提供了新的 API——hydrate()。服务端输出的是字符串，而客户端需要根据这些字符串为 React 完成初始化工作，因此这个方法可以理解为向服务端输出干瘪的字符串“注水”。注满水后就“充满活力”，能正常运转。

React 16.x 中的 hydrate() 应用示例：

```

import React from "react";
import { hydrate } from "react-dom";
// HomePage 组件来自 client/components/homePage/index.js
import HomePage from "./client/components/homePage/index.js";
hydrate(
  <HomePage />,
  document.getElementById("app")
);

```

在服务器端，通过调用 renderToString() 方法把虚拟 DOM 转为字符串然后返回给客户端。

用 Express 作为服务器示例：

```

import express from 'express';
import fs from 'fs';
import path from 'path';
import React from 'react';
import ReactDOMServer from 'react-dom/server';
// HomePage 组件来自 client/components/homePage/index.js
import HomePage from "./client/components/homePage/index.js";
function handleRender(req, res) {
  // 将组件渲染进 HTML
  const html = ReactDOMServer.renderToString(<HomePage />);
  // 加载 HTML 内容
  fs.readFile('./index.html', 'utf8', function (err, data) {
    if (err) throw err;
    // 在 div 中插入渲染好的 HTML 元素
    const document = data.replace(/<div id="app"><\div>/, '<div id="app">' + html + '</div>');
    // 发送请求结果到客户端
    res.send(document);
  });
}
const app = express();
// 服务器获取静态资源文件位置

```

```

app.use('/build', express.static(path.join(__dirname, 'build')));
// 服务器请求 handleRender 函数
app.get('*', handleRender);
// 服务开始
app.listen(8080);

```

上述代码中，不同端的 App 组件在不同端渲染，但引用的是同一个路径（根目录下的 client/src/index），这样做就达到了一个应用共用一套代码的目的，如图 7.1 所示。

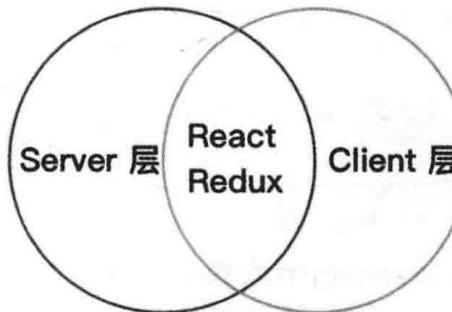


图 7.1 前后同构

上面的服务端渲染使用 res.send()发送字符串内容，这是用于 Express 快速上手的案例，在真实项目中如果这样写，输出完整 HMLT 结构就会显得很麻烦。而 Express 正好支持模板渲染，并且支持多种模板格式。所谓模板就是一组 HTML 标记，启用带有一些特殊的标签可以插入变量或运行一定的逻辑。接下来使用 EJS 作为案例来讲解。

EJS 渲染示例：

下面先给出 EJS 项目结构，以及渲染步骤：

```

├── package.json
├── server.js
└── node_modules
    └── views
        └── index.ejs

```

- (1) 使用 npm install ejs —save 命令安装 EJS，然后安装 npm install express —save。
- (2) 在根目录下创建./views/index.ejs：

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <div>

```

```
<%= data %>
</div>
</body>
</html>
```

(3) 在根目录下创建 server.js，使用模板引擎渲染上面的 index.ejs：

```
var express = require('express');
const app = express();
app.set("view engine", "ejs");
app.get("/", (req, res) => {
    res.render("index", { data: "Welcome" });
});
// 让服务器在 8080 端口执行监听，开始执行后会打印 Listening on port 8080
app.listen(8080, () => {
    console.log("Listening on port 8080");
});
```

(4) 最后在终端上执行 node server.js，或者在 package.json 中修改命令执行 npm start：

```
"scripts": {
    "start": "node server.js"
},
```

(5) 在浏览器中输入 http://localhost:8080，得到如图 7.2 所示结果。



图 7.2 模板引擎

该项目源码地址为 <https://github.com/khno/express-ejs-example>。

7.2.4 状态管理

非私有组件状态依旧采用 Redux 来管理，中间件可以采用 redux-thunk 和 redux-logger 等。为了让客户端与服务端数据同步，需要在服务端把初始状态传入客户端，客户端拿到初始状态后作为预加载状态来创建 Store 的实例。

下面介绍服务端状态管理示例。

服务端：

```
import { renderToString } from 'react-dom/server';
import { Provider } from 'react-redux';
import { createStore } from 'redux';
import App from './App';
import rootReducer from './reducers';
const store = createStore(rootReducer);
async function(ctx) {
    await ctx.render('index', {
        root: renderToString(
            <Provider store={store}>
                <App />
            </Provider>
        ),
        state: store.getState()
    })
}
```

服务端的 index：

```
'<body>
  <div id="root">${content}</div>
  <script>
    window.__REDUX_DATA__ = ${JSON.stringify(data)};
  </script>
</body>'
```

客户端：

```
import { render } from 'react-dom';
import { Provider } from 'react-redux';
import { createStore } from 'redux';
import App from './App';
import rootReducer from './reducers';
const store = createStore(rootReducer, window.__REDUX_DATA__);
render(
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById('root')
)
```

7.2.5 Express 框架简介

为了能让 React 应用在服务器上直接运行，可以使用 Express (<http://expressjs.com/>) 或 Koa (<https://koajs.com/>) 等。当了解了它们之后还可以去了解下 Egg (<https://eggjs.org/>)，笔者此处使用 Express，接下来简单介绍它的物品及应用。

Express 是一个简洁而灵活的 Node.js Web 应用框架，提供了一系列强大特性帮助开发者创建各种 Web 应用和丰富的 HTTP 工具。使用 Express 可以快速地搭建一个完整功能的网站。

Express 框架有以下核心特性：

- 可以设置中间件来响应 HTTP 请求。
- 定义了路由表用于执行不同的 HTTP 请求动作。
- 可以通过向模板传递参数来动态渲染 HTML 页面。

Express 使用示例 1：

(1) 使用 NPM 命令安装 Express：

```
$ npm install express --save
```

执行完上述命令后会在当前项目中额外多出几个 Node 包(在 node_modules 目录下)，它们是安装 Express 所依赖的。此时的项目结构是：

```
├── node_modules
└── package.json
```

(2) 然后在根目录下新建 server.js 文件，代码如下：

```
var express = require('express');
var app = express();

// 主页输出 Hello Express
app.get('/', function (req, res) {
    res.send('Hello Express');
})

app.listen(8888, function () {
    console.log("server is start: 8888")
})
```

项目结构此时变成：

```
├── node_modules
├── server.js
└── package.json
```

(3) 在终端执行 server.js 代码：

```
$ node server.js
```

(4) 然后可以在浏览器中输入 `http://localhost:8080` 访问。在页面中将展示 Hello Express 字样，如图 7.3 所示。

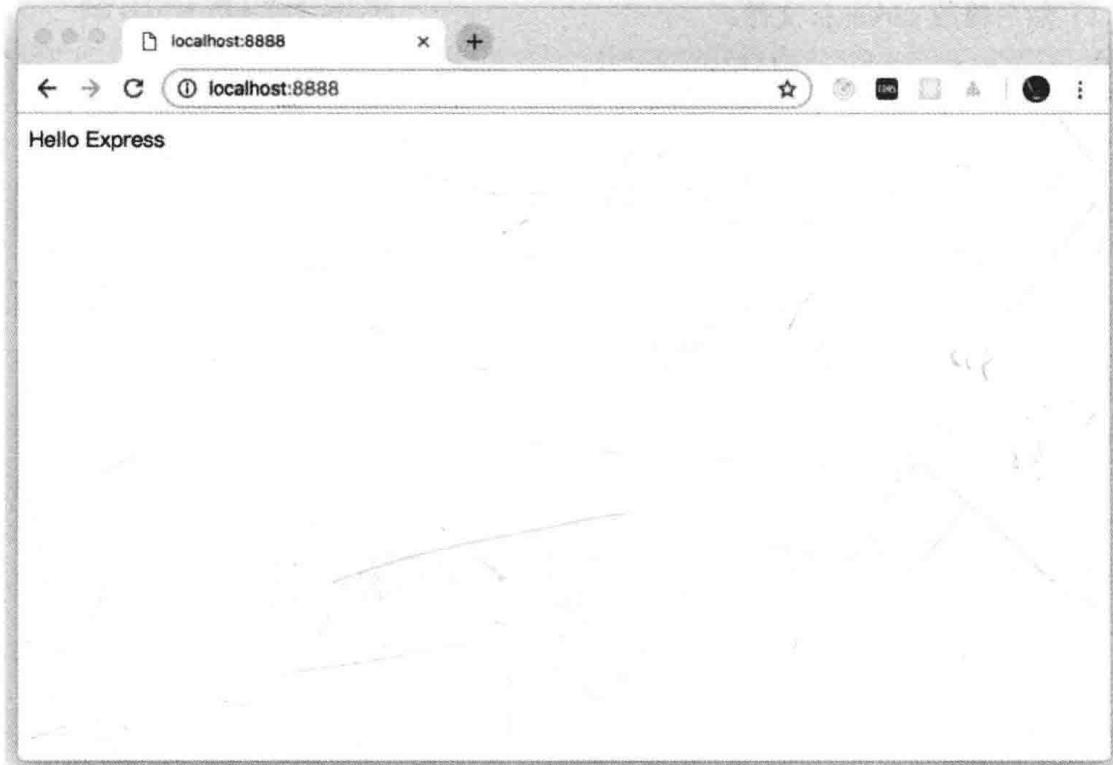


图 7.3 Express demo1 效果

上述代码中，直接通过服务端代码在浏览器中展示文案 Hello Express，那么如何加载服务端的静态资源 `index.html` 文件呢？

很简单，Express 中可以通过 `express.static()` 方法请求来获取服务端的静态资源。Express 提供了内置的中间件 `express.static` 来设置静态文件，如 HTML、JavaScript、CSS 和图片等。

Express 使用示例 2：

(1) 开发者可以使用 `express.static` 中间件来设置静态文件路径。例如，将 HTML、JavaScript、CSS 和图片文件放在 `dist` 目录下，可以这么写：

```
app.use(express.static('dist'));
```

(2) 在根目录中新建一个 `dist` 文件夹，然后在里面新建一个 `index.html`：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
```

```
Hello World  
</body>  
</html>
```

(3) 然后修改 server.js 文件:

```
var express = require("express");
var app = express();
// 设置静态文件目录
app.use(express.static("dist"));
app.get("/index.html", function(req, res) {
    // res.send('Hello Express');
    res.sendFile(__dirname + "/" + "index.html");
});
app.listen(8888, function() {
    console.log("server is start: 8888");
});
```

此时项目结构为：

```
├── node_modules
├── dist                                // 前端静态资源
│   └── index.html
├── server.js
└── package.json
```

(4) 执行 node server.js 文件，在浏览器中输入 `http://localhost:8888` 将会渲染 `dist/index.html` 的内容，如图 7.4 所示。



图 7.4 Express demo2 效果

现在，已经建立起了服务端渲染的基础，后面介绍的内容会在此基础上加以拓展。

7.2.6 路由和 HTTP 请求

在 SPA 中路由是前端控制的，但在服务端渲染中，用户访问的路由是由 Server 层来做。在 Express 中定义了路由表来执行不同 HTTP 的请求动作。在 HTTP 中可以通过路由提取请求的 URL 和 GET 或 POST 参数。

下面介绍一个增加路由示例。

(1) 要添加其他的路由，可以直接在 `server.js` 中添加 `get()` 方法来配置，跟首页配置方法一样。比如增加一个登录页，`./dist/login.html` 内容如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <form action="http://127.0.0.1:8888/login" method="POST">
    账号: <input type="text" name="account"> <br>
    密码: <input type="password" name="password">
    <input type="submit" value="登录">
  </form>
</body>
</html>
```

(2) 书写 `server.js` 文件代码如下：

```
var express = require("express");
var app = express();
// 设置静态文件目录
app.use(express.static("dist"));
app.get("/index.html", function(req, res) {
  // res.send('Hello Express');
  res.sendFile(__dirname + "/" + "index.html");
});
app.get("/login", function(req, res) {
  console.log("/login 响应 get 请求");
  res.sendFile(__dirname + "/" + "login.html");
});
app.listen(8888, function() {
```

```

    console.log("server is start: 8888");
});

```

(3) 在终端再次执行 server.js，在浏览器中输入 <http://localhost:8888/login.html>，就能打开之前创建的新页面。以此类推，新增其他页面。

下面模拟一个登录的 POST 请求示例。

继续来拓展前面的内容，模拟一个登录的 POST 请求。继续修改 server.js 文件如下：

```

var express = require("express");
var app = express();
// express 中间件
var bodyParser = require("body-parser");
// 创建 application/x-www-form-urlencoded 编码解析
var urlencodedParser = bodyParser.urlencoded({ extended: false });
// 设置静态文件目录
app.use(express.static("dist"));
app.get("/index.html", function(req, res) {
    // res.send('Hello Express');
    res.sendFile(__dirname + "/" + "index.html");
});
app.get("/login", function(req, res) {
    console.log("/login 响应 get 请求");
    res.sendFile(__dirname + "/" + "login.html");
});
// POST 请求
app.post("/login", urlencodedParser, function(req, res) {
    // 输出 JSON 格式
    var response = {
        account: req.body.account,
        password: req.body.password
    };
    console.log(response);
    res.end(JSON.stringify(response));
});
app.listen(8888, function() {
    console.log("server is start: 8888");
});

```

 注意：body-parser 是 express 中常用的中间件，作用是对 POST 请求体进行解析。这里不做深入探讨，有兴趣的读者可自行查阅相关资料。

在终端执行 node server.js，在浏览器中输入 <http://localhost:8888/login.html>，展示效果如图 7.5 所示。

输入账号和密码后单击“登录”按钮，页面展示效果如图 7.6 所示。



图 7.5 Express demo3 效果

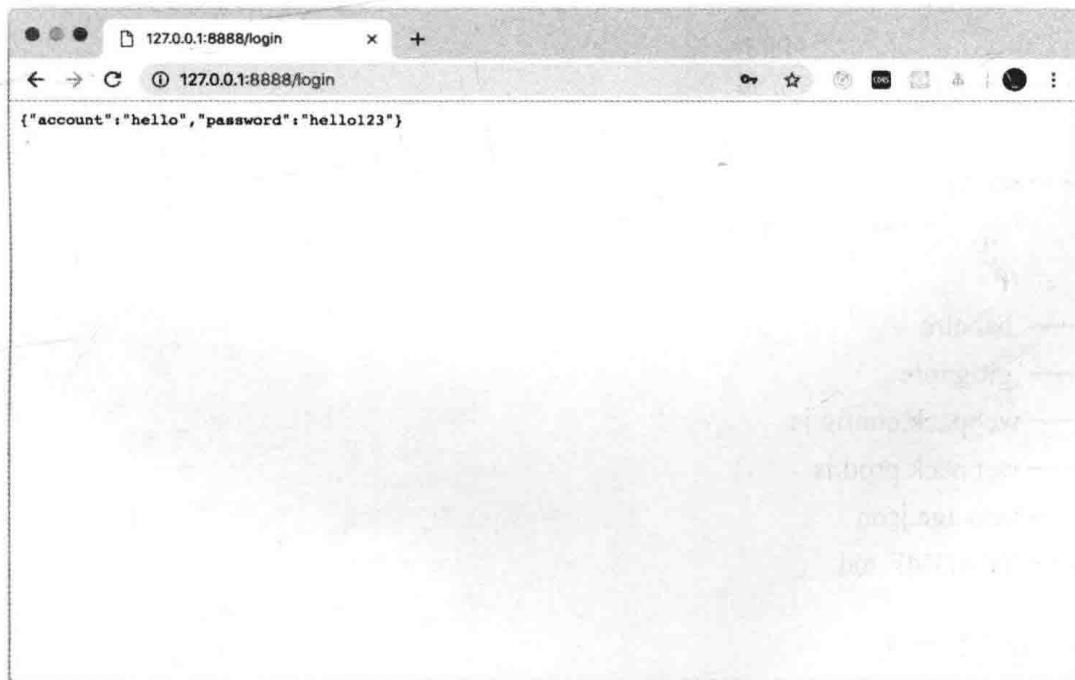


图 7.6 Express demo4 效果

当服务端能得到 HTTP 请求返回结果后，就只剩下对数据库的增、删、改、查操作了。Node.js 中有 Sequelize 等对象关系映射（ORM）技术工具，可以轻松对数据库进行操作，本书暂不展开对数据库操作的解读。

其他相关的 HTTP 请求知识，读者可以自行了解，这里不做探讨。了解了上面案例之后，读者可以尝试完成 cookie 管理、文件上传等操作。

7.3 实战训练——服务端渲染

学习了前面的基础知识之后，本节将通过搭建一个简单的服务端渲染项目来加深读者对知识的理解。

7.3.1 项目结构

新建一个项目，主干目录结构如下：



7.3.2 项目实现

首先来搭建 Webpack 配置。Webpack 配置了两种环境：webpack.config.js 用于开发环境打包配置，webpack.prod.js 用于生产环境的打包配置。先来书写这两个文件。

(1) ./webpack.config.js 文件内容如下：

```
var webpack=require('webpack');
var path=require('path');
```

```

module.exports = {
  entry: {
    app: './client/app.js', // 入口
    vendor: ['react', 'react-dom']
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'js/[name].js',
    publicPath: '/'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        loader: 'babel-loader', // JS 编译
        exclude: path.resolve(__dirname, 'node_modules'),
        include: path.resolve(__dirname, 'client'),
        query: {
          presets: ['latest', 'stage-0', 'react']
        }
      },
      {
        test: /\.tpl$/,
        loader: 'ejs-loader'
      },
      {
        test: /\.css$/,
        loader: 'style-loader!css-loader?importLoaders=1!postcss-loader' // CSS 样式编译
      },
      {
        test: /\.less$/,
        loader: 'style-loader!css-loader!postcss-loader!less-loader' // less 样式编译
      },
      {
        test: /\.scss$/,
        loader: 'style-loader!css-loader!postcss-loader!sass-loader' // scss 样式编译
      },
      {
        test: /\.html$/,
        loader: 'html-loader' // HTML 相关编译
      },
      {
        test: /\.(png|jpg|gif|svg)$/, // 图片编译
        loader: 'file-loader',
        query: {
          name: 'assets/[name]-[hash].[ext]'
        }
      },
      {
        test: /\.(woff|woff2|eot|ttf|otf)$/, // 字体编译
        loader: 'file-loader',
        query: {
          name: 'assets/[name]-[hash].[ext]'
        }
      },
      {
        test: /\.(csv|tsv)$/, // csv/tsv 编译
        loader: 'csv-loader',
        query: {
          name: 'assets/[name]-[hash].[ext]'
        }
      }
    ]
  }
}

```

```

    },
    test: /\.xml$/,
    loader:'xml-loader',
    query:{
      name:'assets/[name]-[hash].[ext]'
    }
  }
],
devServer:{
  hot:true,
  contentBase:path.resolve(__dirname,'dist'),
  publicPath:'/',
},
plugins:[
  // 公用的js库
  new webpack.optimize.CommonsChunkPlugin({
    name:'vendor',
    filename:'vendor.bundle.js'
  }),
  new webpack.HotModuleReplacementPlugin(),
  new webpack.NamedModulesPlugin(),
],
devtool:"inline-source-map"
}

```

(2) ./webpack.prod.js 文件内容如下:

```

var webpack=require('webpack');
var path=require('path');
module.exports={
  entry:{
    main:'./client/app.js', // 打包入口
  },
  output:{
    path:path.resolve(__dirname,'lib'),
    filename:'app.min.js',
    libraryTarget: 'umd',
    publicPath: '/'
  },
  module:{
    loaders:[
      {
        test:/\.js$/,
        loader:'babel-loader', // JS 编译
        exclude:path.resolve(__dirname,'node_modules'),
        include:path.resolve(__dirname,'client'),
        query:{
          presets:['latest','stage-0', 'react']
        }
      },
      {
        test:/\.tpl$/,
        loader:'ejs-loader' // tpl 编译
      },
      {

```

```

test: /\.css$/,
loader: 'style-loader!css-loader?importLoaders=1!postcss-loader'
},
{
  test: /\.less$/,
loader: 'style-loader!css-loader!postcss-loader!less-loader'
},
{
  test: /\.scss$/,
loader: 'style-loader!css-loader!postcss-loader!sass-loader'
},
{
  test: /\.html$/,
loader: 'html-loader'
},
{
  test: /\.(png|jpg|gif|svg)$/,
loader: 'file-loader',
query: {
    name: 'assets/[name]-[hash].[ext]'
}
},
{
  test: /\.woff|woff2|eot|ttf|otf$/,
loader: 'file-loader',
query: {
    name: 'assets/[name]-[hash].[ext]'
}
},
{
  test: /\.csv|tsv$/,
loader: 'csv-loader',
query: {
    name: 'assets/[name]-[hash].[ext]'
}
},
{
  test: /\.xml$/,
loader: 'xml-loader',
query: {
    name: 'assets/[name]-[hash].[ext]'
}
},
],
},
externals: { // 该配置可以让 Webpack 不打包以下 JS 库
'react': 'umd react',
'react-dom': 'umd react-dom'
},
plugins:[
  new webpack.optimize.UglifyJsPlugin({
    compress: {
      warnings: false
    }
  })
],
devtool:"cheap-module-source-map"
}

```

entry 为该项目的入口,也就是./client/app.js, output 为打包后的输出路径。关于 Webpack

的内容，第 1 章有相应介绍，这里不再介绍。

(3) 配置服务./server/server.js:

```
import express from "express";
import React from "react";
import { renderToString } from "react-dom/server";
// HomePage 组件来自客户端组件
import HomePage from "../client/components/homepage/index.js";
let app = express();
// 设置静态资源文件目录
app.use("/dist", express.static("dist"));
app.get("/", (req, res) => {
  res.write(
    "<!DOCTYPE html><html><head><title>Hello HomePage</title></head><body>" +
    "<div id='app'>" +
    renderToString(<HomePage />) +
    "</div></body>" +
    "<script type='text/javascript' client='../dist/vendor.bundle.js'>
    </script><script type='text/javascript' client='../dist/js/app.js'>
    </script>" +
  );
  res.write("</html>");
});
app.listen(8080, () => {
  console.log("server is start:", 8080);
});
```

当执行 server.js 这个脚本文件后，在浏览器中输入 `http://localhost:8080`，就能访问到根目录下的./dist 内的静态资源。Express 的 get 方法中，当访问 `http://localhost:8080` 时，会写入 HTML 的内容，其中的组件通过引用客户端组件 HomePage 用 `renderToString()` 方法来加载：

```
renderToString(<HomePage />)
```

(4) 在 package.json 中，使用 npm run 命令来执行对应的 Webpack 脚本对项目进行编译和打包：

```
"scripts": {
  "server": "babel-node ./server/server.js",
  "compile": "webpack --config webpack.config.js --progress --display-modules --display-reasons --colors",
  "build": "webpack --config webpack.prod.config.js --progress --display-modules --display-reasons --colors"
},
```

`npm run server` 命令用于启动 node 服务，该命令就是用 `babel-node` 执行了 `./server/server.js`。这里之所以采用 `babel-node` 代替 `node` 命令，是因为 `babel-node` 可以直接运行 ES 6 脚本，而 `server.js` 正是用 ES 6 写的。`babel-node` 不用单独安装，而是随着 `babel-cli` 一起安装。然后执行 `babel-node` 就进入 PEPL 环境。`npm run compile` 命令用于编译本地开发，`npm run build` 用于生成生产环境使用的静态资源。

(5) 服务端配置完成后，接下来看./client下面的客户端代码，结构如下：

```
client
└── app.js
└── components
    └── homepage
        └── index.js
```

./client/app.js 为入口文件，React 不同版本的写法略有不同，介绍如下：

React 15.x 版本：

```
import React from "react";
import { render } from "react-dom";
import HomePage from "./components/homepage/index.js";
render(<HomePage />, document.getElementById('app'));
```

React 16.x 版本：

```
import React from "react";
import { hydrate } from "react-dom";
import HomePage from "./components/homepage/index.js";
hydrate(<HomePage />, document.getElementById("app"));
```

(6) React 16.x 版本中渲染方法变更为 `hydrate()`，主要是用于给服务端渲染出的 HTML 结构进行“注水”，由于新版本中 SSR 出的 DOM 节点不再带有 `data-react`，为了能尽可能复用 SSR 的 HTML 内容，所以需要使用新的 `hydrate()` 方法进行事件绑定等客户端独有的操作。

./client/components/homePage/index.js:

```
import React from "react";
export default class HomePage extends React.Component {
  componentDidMount() {
    console.log("渲染了 HomePage");
  }
  render() {
    return <h1>Hello World</h1>;
  }
}
```

client 端的内容开发流程基本没有大的变化，依旧是组件化的概念。

以上就是一个简单的服务端渲染项目，现在读者了解了如何在服务端渲染 React 组件及如何“挂载”。之后可以根据产品业务需要进行不断拓展，比如在此基础上读者还可以添加 Webpack 热更新、添加路由、Redux 状态管理，甚至写接口操作数据库等。

第8章 自动化测试

由于前端项目变得越来越复杂，日常项目的开发和维护中需要修复问题或在原功能基础上不断迭代新功能，为了确保原功能不会引入 bug，就需要自动化测试。本章将介绍什么是前端的自动化测试，以及如何编写 React 测试用例。

8.1 测试的作用

测试是整个开发流程中的最后一环，它用来保障产品的质量。前端开发同样也需要测试，前端偏向于 GUI (Graphical User Interface) 软件的特殊性，很多人目前还是以“人肉”测试为主。很多前端开发者没有编写和维护测试用例的习惯，认为开发完就行了，实则相反。虽然后面的工作有专门的测试部门的测试人员来完成，但开发人员做好自测可以减少开发的 bug，提高代码质量，快速定位问题等。有了自动化测试，能让开发者更加信任自己的代码，减少整个开发流程中测试与开发者反复修改的时间。

自动化测试一般是指软件测试的自动化。软件测试就是在预设条件下运行系统或应用程序，评估运行结果，预先条件应包括正常条件和异常条件。从广义上来说，通过工具的方式来代替或者辅助手工测试的行为都可以理解为自动化测试。

不同种类的测试扮演着不同的角色：接口测试自动化关注点在于 API，用于保障接口不出现问题；功能测试自动化是确保应用程序从用户的角度来看是正常运行的自动化测试；单元测试自动化用于被测试的类或方法，根据类或方法的参数，传入相应的数据，然后得到一个返回结果等。

说明：由于自动化测试包含的内容比较广，本章主要介绍针对 React 的视图层和功能层的自动化测试。

8.2 单元测试简介

单元测试（unit testing），是指对软件中的最小可测试单元进行检查和验证。它就是

一个检测，所以只有通过与不通过两种结果。对于单元测试的定义，维基百科中的解释是这样的：

In computer programming, unit testing is a method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures are tested to determine if they are fit for use.

简而言之，就是以最小可测试单元为单位通过测试工具确认其是否能使用。单元测试是在软件开发过程中进行的最低级别的测试活动，软件的独立单元将在与程序的其他部分相隔离的情况下进行测试。单元测试最大的特点是测试对象的细颗粒度性，即被测对象独立性高、复杂度低。

前端自动化测试一般是指是在预设条件下运行前端页面或逻辑模块，评估运行结果。预设的条件中理应包含正确的条件和异常条件，从而达到能自动运行和减少人工干预测试。通常针对的是 JavaScript 相关的函数、对象和模块所做的测试。如果测试通过，那么这个程序就可以开始使用，否则，需要完善并改进相关代码。

前端单元测试与其他测试最大的区别在于前端单元测试无法避免存在兼容性问题，比如涉及调用浏览器的 API 等。因此前端单元测试需要在“浏览器”环境下运行。

从测试环境上来分，前端单元测试主要有以下 3 种方案。

1. 基于JSDOM

优点：执行速度最快，因为不需要启动浏览器。

缺点：无法测试如 Session 或 Cookie 等相关操作，并且由于不是真实浏览器环境，因此无法保证一些如 DOM 相关和 BOM 相关操作的正确性，并且 JSDOM 未实现 localStorage，如果需要进行覆盖，只能使用第三方库如 node-localStorage（这个库本身对执行环境的判断有一些问题）进行模拟。

2. 基于PhantomJs等无头浏览器

优点：相对较快，并且具有真实的 DOM 环境。

缺点：不在真实浏览器中运行，难以调试，并且项目 issue 非常多，puppeteer 发布后作者宣布不再维护。

3. 使用Karma或puppeteer等工具，调用真实的浏览器环境进行测试

优点：配置简单，能在真实的浏览器中运行测试，并且 karma 能将测试代码在多个浏览器中运行，同时方便调试。

缺点：唯一的缺点就是相对于前两者运行稍慢，但是在单元测试可接受范围内。

8.3 测试工具

什么工作都离不开工具，测试也有很多工具，本节将介绍常用的几款测试工具。

8.3.1 常见的测试工具

1. Karma

Karma 是一个基于 Node.js 的 JavaScript 测试执行过程管理工具（Test Runner）。它的原名是 Testacular，Google 在 2012 年开源了它，2013 年 Testacular 改名为 Karma。该工具可用于测试所有主流 Web 浏览器，也可集成到 CI（Continuous Integration）工具，也可和其他代码编辑器一起使用。这个测试工具的一个强大特性就是它可以监控文件的变化，然后自行执行，通过 `console.log` 显示测试结果。Karma 也是 Google Angular 团队开发的测试运行平台，配置简单灵活，能够很方便地将测试在多个真实浏览器中运行。

2. Mocha

Mocha 是 JavaScript 的一种单元测试框架，既可以在浏览器环境下运行，也可以在 Node.js 环境下运行。它有完善的生态系统，简单的测试组织方式，不对断言库和工具做任何限制，非常灵活。Mocha 既可以测试简单的 JavaScript 函数，又可以测试异步代码，因为异步是 JavaScript 的特性之一。并且可以自动运行所有测试，也可以只运行特定的测试。可以支持 `before`, `after`, `beforeEach` 和 `afterEach` 来编写初始化代码。更多详可请访问：<https://mochajs.org/>查阅。

3. Jasmine

Jasmine 是单元测试框架。它不依赖浏览器、DOM 和其他 JavaScript 框架。它有拥有灵巧而明确的语法可以让开发者轻松地编写测试代码。和 Mocha 语法非常相似，最大的差别是提供了自建的断言、Spy 和 Stub。更多详可请访问：<https://jasmine.github.io/>查阅。

4. Jest

Jest 是 Facebook 的一个专门进行 JavaScript 单元测试的工具，之前仅限他们的前端工程师在公司内部使用，后来进行开源，它是在 Jasmine 测试框架上演变开发而来。更多详情可访问 <https://jestjs.io/en/>查阅。

5. AVA

AVA 是一个简约的测试库，它的优势是 Java 的异步特性和并发运行测试，这反过来提高了性能。利用了 Java 的异步特性优势为测试提供了额外的好处。最主要的好处是优化了在部署时的时间等待。和其他测试框架最大的区别在于多线程，运行速度更快。

前端测试工具比较多，以上只例举了部分测试工具，读者可自行查阅其他工具。在 React 中，官方虽然也有配套的测试工具 `react-addons-test-utils`（React 16.x 中迁移到 `react-dom/test-utils`），但用起来比较繁琐，写出来的测试代码也不易维护。

8.3.2 React 的测试工具

React 本身有内置的测试工具 `react-dom/test-utils`，从目录就可看出它是一个 `react-dom` 的辅助测试工具。这个库的主要作用是遍历 ReactDOM 生成的 DOM 树，方便编写断言。

当测试的对象是事件（如 `click`、`change`、`blur`）时，用 `react-dom/test-utils` 比较合适。缺点是它的 API 比较繁琐，通常使用 Enzyme 等工具来代替它。即便如此，还是有人会选择用 Jest 搭配 `react-dom/test-utils` 来写测试。而且这个工具这可以更好地理解 React 的工作原理。

 注意：`react-dom/test-utils` 需要提供 DOM 环境。

8.3.3 单元测试工具 Jest

Jest 是 Facebook 开源的 JavaScript 单元测试工具，由开源社区的开发者和 Facebook 员工在维护，适合在 React 中使用。Jest 提供了包括内置的测试环境 DOM API 支持、断言库、Mock 库等，还包含了 Snapshot Testing、Instant Feedback 等特性。

Jest 被 Facebook 用来测试包括 React 应用在内的所有 JavaScript 代码。Jest 的理念是，提供一套完整集成的“零配置”测试体验。它有两个特性：

- 能在虚拟 DOM 中运行测试；
- 支持 JSX 语法。

如果项目是使用 `create-react-app` 或 `react-native-init` 创建的，那么此时 Jest 都已经被配置好并可以直接使用了。可以直接在根目录创建 `_tests_` 文件夹下放置测试用例，或者在需要测试的组件内部创建以 `.spec.js` 或 `.test.js` 为后缀的文件。不管选择哪一种方式，Jest 都能找到并且运行它们。

假如没有使用 `create-react-app` 或 `react-native-init`，那么就以下面官方的快照测试为例，来介绍如何使用 Jest。想要确保 UI 不会意外被更改，快照测试是非常有用的工具。如果前后快照不匹配，测试失败，此时就可以判断是否被意外修改了。

 注意：快照测试用于确保 UI 不会有意外的更改。

Jest 使用示例：

(1) 首先新建项目 myjest，在根目录下执行 npm init -y 生成一个 package.json 文件 (-y 表示 yes，跳过 npm init 的提问阶段，直接生成一个 package.json)，如下：

```
{
  "name": "myjest",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

(2) 然后执行 npm install —save-dev jest babel-jest babel-preset-env babel-preset-react react-test-renderer 安装必要的 Node 包。其中，react-test-renderer 作用是负责将组件输出成 JSON 对象以便遍历、断言或进行 snapshot 测试：

```
{
  "name": "myjest",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "babel-jest": "^23.6.0",
    "babel-preset-env": "^1.7.0",
    "babel-preset-react": "^6.24.1",
    "jest": "^23.6.0",
    "react-test-renderer": "^16.5.2"
  }
}
```

(3) 安装 React 相关包，npm install —save react react-dom：

```
"dependencies": {
  "react": "^16.5.2",
  "react-dom": "^16.5.2"
}
```

(4) 在根目录下添加.babelrc，该文件用来设置转码的规则和插件：

```
{
  "presets": ["env", "react"]
}
```

(5) 然后新建一个用于测试的 React 组件 index.js。实现一个非常简单的小功能，鼠标在一个 a 标签上移入和移出，移入 a 标签样式变为 hovered，移出 a 标签样式变为 normal。代码如下：

```
import React from 'react';
const STATUS = {
  HOVERED: 'hovered',
  NORMAL: 'normal',
};
export default class Link extends React.Component {
  constructor(props) {
    super(props);
    this._onMouseEnter = this._onMouseEnter.bind(this);
    this._onMouseLeave = this._onMouseLeave.bind(this);
    this.state = {
      class: STATUS.NORMAL,
    };
  }
  _onMouseEnter() {
    this.setState({
      class: STATUS.HOVERED
    });
  }
  _onMouseLeave() {
    this.setState({
      class: STATUS.NORMAL
    });
  }
  render() {
    return (
      <a
        className={this.state.class}
        href={this.props.url || '#'}
        onMouseEnter={this._onMouseEnter}
        onMouseLeave={this._onMouseLeave}
      >
        {this.props.children}
      </a>
    );
  }
}
```

(6) 接下来使用 react-test-render 和 Jest 快照与组件进行交互，在根目录下创建 test /index.test.js：

```
import React from 'react';
// 引入需要测试的 Link 组件
import Link from './index';
import renderer from 'react-test-renderer';
test('Link changes the class when hovered', () => {
  const component = renderer.create(
    <Link url="https://jestjs.io/">Delightful JavaScript Testing</Link>,
  );
  let tree = component.toJSON();
```

```

expect(tree).toMatchSnapshot();
// 手动触发回调函数
tree.props.onMouseEnter();
// 重渲染
tree = component.toJSON();
expect(tree).toMatchSnapshot();
// 手动触发回调函数
tree.props.onMouseLeave();
// 重渲染
tree = component.toJSON();
expect(tree).toMatchSnapshot();
});

```

上述代码中，`toMatchSnapshot()`方法会对比当前将要生成的结构与上次生成的结构的区别。

(7) 最后，在终端执行 `jest`，执行结果如下：

```

MacBook-Pro:myjest jack$ jest
PASS ./index.test.js
  ✓ Link changes the class when hovered (17ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   3 passed, 3 total
Time:        0.847s, estimated 1s
Ran all test suites.

```

此时项目根目录下生成了一个快照文件 `./_test/_snapshots_/index.test.js.snap`：

```

// Jest Snapshot v1, https://goo.gl/fbAQLP
exports['Link changes the class when hovered 1'] =
<a
  className="normal"
  href="https://jestjs.io/"
  onMouseEnter={[Function]}
  onMouseLeave={[Function]}
>
  Delightful JavaScript Testing
</a>
';
exports['Link changes the class when hovered 2'] =
<a
  className="hovered"
  href="https://jestjs.io/"
  onMouseEnter={[Function]}
  onMouseLeave={[Function]}
>
  Delightful JavaScript Testing
</a>
';
exports['Link changes the class when hovered 3'] =
<a
  className="normal"
  href="https://jestjs.io/"
  onMouseEnter={[Function]}
  onMouseLeave={[Function]}

```

```
>     Delightful JavaScript Testing
</a>
';
```

当下次执行测试命令时，渲染的结果将会和上一次生成的快照进行对比。如果快照测试不通过，就得查看是否有问题；如果符合预期，就执行 `jest-u` 来重写快照。

(8) 如果修改了 Link 组件，在 a 标签内增加一个 p 标签`<p>`，这里新增了 `text</p>`：

```
render() {
  return (
    <a
      className={this.state.class}
      href={this.props.url || '#'}
      onMouseEnter={this._onMouseEnter}
      onMouseLeave={this._onMouseLeave}
    >
      <p>这里新增了 text</p>
      {this.props.children}
    </a>
  );
}
```

(9) 此时，执行 `jest`，终端将会展示如下内容：

```
MacBook-Pro:myjest jack$ jest
FAIL __test__/index.test.js
  ✘ Link changes the class when hovered (26ms)
  ● Link changes the class when hovered
    expect(value).toMatchSnapshot()
      Received value does not match stored snapshot "Link changes the class when hovered 1".
      - Snapshot
      + Received
        @@ -2,7 +2,8 @@
          className="normal"
          href="https://jestjs.io/"
          onMouseEnter={[Function]}
          onMouseLeave={[Function]}
        >
      +  这里新增了 text
        Delightful JavaScript Testing
      </a>
        8 |   );
        9 |   let tree = component.toJSON();
      > 10 |   expect(tree).toMatchSnapshot();
           |
        11 |
        12 |   // 手动触发回调函数
        13 |   tree.props.onMouseEnter();
            at Object.toMatchSnapshot (__test__/index.test.js:10:16)
  ● Link changes the class when hovered
    expect(value).toMatchSnapshot()
      Received value does not match stored snapshot "Link changes the class when hovered 2".
```

```

- Snapshot
+ Received
@@ -2,7 +2,8 @@
  className="hovered"
  href="https://jestjs.io/"
  onMouseEnter={[Function]}
  onMouseLeave={[Function]}
>
+ 这里新增了 text
  Delightful JavaScript Testing
</a>
14 | // 重渲染
15 | tree = component.toJSON();
> 16 | expect(tree).toMatchSnapshot();
| ^
17 |
18 | // 手动触发回调函数
19 | tree.props.onMouseLeave();
at Object.toMatchSnapshot (_test_/index.test.js:16:16)
● Link changes the class when hovered
expect(value).toMatchSnapshot()
Received value does not match stored snapshot "Link changes the class
when hovered 3".
- Snapshot
+ Received
@@ -2,7 +2,8 @@
  className="normal"
  href="https://jestjs.io/"
  onMouseEnter={[Function]}
  onMouseLeave={[Function]}
>
+ 这里新增了 text
  Delightful JavaScript Testing
</a>
20 | // 重渲染
21 | tree = component.toJSON();
> 22 | expect(tree).toMatchSnapshot();
| ^
23 | });
at Object.toMatchSnapshot (_test_/index.test.js:22:16)
> 3 snapshots failed.
Snapshot Summary
> 3 snapshots failed from 1 test suite. Inspect your code changes or re-run
jest with '-u' to update them.
Test Suites: 1 failed, 1 total
Tests:      1 failed, 1 total
Snapshots:  3 failed, 3 total
Time:       1.437s
Ran all test suites.

```

由于更新和修改了组件的内容，这次的组件快照和上一次的组件快照不匹配，所以这次快照会执行失败，并且详细记录此次修改的位置和内容。这对于开发 UI 组件来说是非常有用的。

该测试项目的结构如下：

```

├── .babelrc
└── test
    ├── snapshots
    │   └── index.test.js.snap
    └── index.test.js // 测试脚本
├── index.js // 项目组件
└── package.json

```

更多详情请参考：<https://jestjs.io/docs/en/tutorial-react>。

8.3.4 单元测试工具 Enzyme

Enzyme 是由 Airbnb 开发的用于 React 单元测试的工具。它扩展了 React 的 TestUtils，并通过支持类似于 jQuery 的 find 语法，可以很方便地对 render() 出来的结果做各种断言。Enzyme 模拟类似 jQuery 的 API 非常直观，便于开发者使用和学习。

它提供了 3 种渲染方式，分别介绍如下。

(1) shallow：浅渲染是对官方 shallow rendering 的封装，仅渲染出虚拟节点。它不会返回真实的节点，能极大提高测试性能。而且它只渲染第一层，不会渲染所有子组件，因此速度快。它不适合测试包含子组件及需要测试声明周期的组件。shallow 只渲染本组件内容，引用的外部组件不渲染。

```

import { shallow } from 'enzyme';
const wrapper = shallow(<MyComponent />);
// ...

```

(2) mount：用于将 React 组件加载为真实 DOM 节点。然而真实 DOM 需要一个浏览器环境，为了解决这个问题，会用到 jsdom（使用 JS 模拟的 DOM 环境）。mount 全渲染，会渲染组件内所有内容。

```

import { mount } from 'enzyme';
const wrapper = mount(<MyComponent />);
// ...

```

(3) render：用于将 React 组件渲染成静态的 HTML 字符串，然后分析这段 HTML 代码的结构，返回一个 JS 对象。render 返回的 wrapper 与上面两个 API 类似，不同的是，render 使用了第三方 HTML 解析器和 Cheerio（模拟 DOM 环境，Enzyme 内部使用的全渲染框架）。

```

import { render } from 'enzyme';
const wrapper = render(<MyComponent />);
// ...

```

如果不需要操作和断言子组件，使用 shallow 就可以了。shallow 只渲染当前组件，但只能对当前组件做断言；而 mount 适用于含有 DOM API 交互组件或需要测试含有完整声

明周期的组件，它会渲染当前组件及所有子组件。但 mount 耗时比 shallow 长。

Enzyme 的部分 API 如下：

- .get(index): 返回指定位置的子组件的 DOM 节点；
- .at(index): 返回指定位置的子组件；
- .first(): 返回第一个子组件；
- .last(): 返回最后一个子组件；
- .type(): 返回当前组件的类型；
- .text(): 返回当前组件的文本内容；
- .html(): 返回当前组件的 HTML 代码形式；
- .props(): 返回根组件的所有属性；
- .prop(key): 返回根组件的指定属性；
- .state([key]): 返回根组件的状态。

从上面的 API 可以看出，Enzyme 的 API 用法形似 jQuery，非常简单易懂。下面来演示上述 3 种渲染方式。

Enzyme shallow 示例：

```
import React from 'react';
import { expect } from 'chai';
import { shallow } from 'enzyme';
import sinon from 'sinon';
// 被测试组件引入
import MyComponent from './MyComponent';
import Foo from './Foo';
describe('<MyComponent />', () => {
  it('renders three <Foo /> components', () => {
    const wrapper = shallow(<MyComponent />); // 浅渲染
    expect(wrapper.find(Foo)).to.have.lengthOf(3);
  });
  it('renders an ".icon-star"', () => {
    const wrapper = shallow(<MyComponent />);
    expect(wrapper.find('.icon-star')).to.have.lengthOf(1);
  });
  it('renders children when passed in', () => {
    const wrapper = shallow((
      <MyComponent>
        <div className="unique" />
      </MyComponent>
    ));
    expect(wrapper.contains(<div className="unique" />)).to.equal(true);
  });
  it('simulates click events', () => {
    const onButtonClick = sinon.spy();
    const wrapper = shallow(<Foo onButtonClick={onButtonClick} />);
    wrapper.find('button').simulate('click');
    expect(onButtonClick).to.have.property('callCount', 1);
  });
});
```

Enzyme mount 示例：

```
import React from 'react';
import sinon from 'sinon';
import { expect } from 'chai';
import { mount } from 'enzyme';
// 被测试组件引入
import Foo from './Foo';
describe('<Foo />', () => {
  it('allows us to set props', () => {
    const wrapper = mount(<Foo bar="baz" />);
    expect(wrapper.props().bar).to.equal('baz');
    wrapper.setProps({ bar: 'foo' });
    expect(wrapper.props().bar).to.equal('foo');
  });
  it('simulates click events', () => {
    const onButtonClick = sinon.spy();
    const wrapper = mount((
      <Foo onButtonClick={onButtonClick} />
    ));
    wrapper.find('button').simulate('click');
    expect(onButtonClick).to.have.property('callCount', 1);
  });
  it('calls componentDidMount', () => {
    sinon.spy(Foo.prototype, 'componentDidMount');
    const wrapper = mount(<Foo />);
    expect(Foo.prototype.componentDidMount).to.have.property('callCount', 1);
    Foo.prototype.componentDidMount.restore();
  });
});
```

Enzyme render 示例：

```
import React from 'react';
import { expect } from 'chai';
import { render } from 'enzyme';
// 被测试组件引入
import Foo from './Foo';
describe('<Foo />', () => {
  it('renders three ".foo-bar"s', () => {
    const wrapper = render(<Foo />);
    expect(wrapper.find('.foo-bar')).to.have.lengthOf(3);
  });
  it('renders the title', () => {
    const wrapper = render(<Foo title="unique" />);
    expect(wrapper.text()).to.contain('unique');
  });
});
```

8.4 Jest 和 Enzyme 实战训练

前面学习了单元测试的理论知识，本节将基于第 2.4 节的组件化实例——Todolist（源

码地址 (<https://github.com/khno/react-component-todolist>) 进行单元测试实战。为了方便理解，将之前项目结构稍作改动，改动后的项目结构如下：

```

    └── README.md
    └── index.html
    └── package-lock.json
    └── package.json
    └── src
        ├── app.css
        ├── app.js
        └── components
            ├── Form.js
            ├── Header.js
            └── ListItems.js
    └── webpack.config.js

```

下面给出主要文件的代码。

(1) ./src/app.js 代码清单：

```

import React, { Component } from "react";
import { render } from "react-dom";
import PropTypes from "prop-types";
import "./app.css";
import Header from "./components/Header";
import Form from "./components/Form";
import ListItems from "./components/ListItems";
export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      todoItem: "",
      items: ["吃苹果", "吃香蕉", "喝奶茶"]
    };
    this.onChange = this.onChange.bind(this);
    this.onSubmit = this.onSubmit.bind(this);
  }
  onChange(event) {
    this.setState({
      todoItem: event.target.value
    });
  }
  onSubmit(event) {
    event.preventDefault();
    this.setState({
      todoItem: "",
      items: [...this.state.items, this.state.todoItem]
    })
  }
  render() {
    return (
      <div className="container">
        <Header title="TodoList" />
        <Form
          onSubmit={this.onSubmit}

```

```

        onChange={this.onChange}
        todoItem={this.state.todoItem}
      />
      <ListItems items={this.state.items} />
    </div>
  );
}

App.propTypes = {
  items: PropTypes.array,
  todoItem: PropTypes.string,
  onChange: PropTypes.func
};
render(<App />, document.getElementById("app"));

```

(2) ./src/components/Header.js 代码清单:

```

import React from 'react';
const Header = props => (
  <h1>{props.title}</h1>
);
export default Header;

```

(3) ./src/components/Form.js 代码清单:

```

import React from "react";
const Form = props => (
  <div className="form-wrap">
    <input value={props.todoItem} onChange={props.onChange} />
    <button onClick={props.onSubmit}>Submit</button>
  </div>
);
export default Form;

```

(4) ./src/components/ListItems.js 代码清单:

```

import React from 'react';
const ListItems = props => (
  <ul>
    {
      props.items.map(
        (item, index) => <li key={index}>{item}</li>
      )
    }
  </ul>
);
export default ListItems;

```

本节将基于以上项目继续开发。

8.4.1 Jest 和 Enzyme 的配置

(1) 安装 Jest 和 Enzyme 相关的 Node 包。

```

npm i --save-dev enzyme enzyme-adapter-react-16 jest react-test-renderer
@types/enzyme @types/jest

```

(2) 基于此，读者还需要在根目录下创建以下文件：

- test-setup.js;
- test-shim.js。

test-setup.js 是为 Enzyme 创建 React 16.x 的初始配置，代码如下：

```
/***
 * Defines the React 16 Adapter for Enzyme.
 *
 * @link http://airbnb.io/enzyme/docs/installation/#working-with-react-16
 * @copyright 2017 Airbnb, Inc.
 */
const enzyme = require("enzyme");
const Adapter = require("enzyme-adapter-react-16");
enzyme.configure({ adapter: new Adapter() });
```

test-shime.js 是配置去除有关缺少浏览器 polyfill 的警告，代码如下：

```
/***
 * Get rids of the missing requestAnimationFrame polyfill warning.
 *
 * @link https://reactjs.org/docs/javascript-environment-requirements.html
 * @copyright 2004-present Facebook. All Rights Reserved.
 */
global.requestAnimationFrame = function(callback) {
  setTimeout(callback, 0);
};
```

(3) 创建完以上文件后需要在 package.json 中配置它们，package.json 配置代码如下：

```
"jest": {
  "setupFiles": [
    "<rootDir>/test-shim.js",
    "<rootDir>/test-setup.js"
  ],
  "moduleFileExtensions": [
    "js"
  ],
  "testMatch": [
    "**/_tests_/*.(js)"
  ]
}
```

以上代码中：

- **setupFiles**: 用于运行某些代码以配置或设置测试环境的模块的路径。每个测试文件将运行一次 `setupFile`。由于每个测试都在自己的环境中运行，因此这些脚本将在执行测试代码本身之前立即在测试环境中执行。
- **moduleFileExtensions**: 是模块使用的文件扩展名格式，如果需要模块而不指定文件扩展名，则这些是 Jest 将寻找的扩展。如果使用的是 TypeScript，那么应该写成：

```
["js", "jsx", "json", "ts", "tsx"]
```

- **testMatch**: 用于测试匹配对应的文件，`"**/_tests_/*.(js)"` 能查找项目中以 `_tests_` 命名的文件，然后在里面寻找所有以 `js` 为后缀的文件名进行执行测试。

(4) 由于项目组件中有 css 引入，使用 Jest 测试代码的时候会发生识别报错，因此还需要 identity-obj-proxy 来 mock，它能在引用 class 的地方直接返回 class 的类名。安装命令：

```
npm install --save-dev identity-obj-proxy
```

(5) 在 package.json 的 jest 中配置 moduleNameMapper：

```
"jest": {
  "setupFiles": [
    "<rootDir>/test-shim.js",
    "<rootDir>/test-setup.js"
  ],
  "moduleFileExtensions": [
    "ts",
    "js"
  ],
  + "moduleNameMapper": {
    + "\\\.(css|less)$": "identity-obj-proxy"
  },
  "testMatch": [
    "**/_tests_/*.(ts|js)"
  ]
}
```

(6) 最后在 package.json 的 scripts 中配置 test：

```
"scripts": {
  "start": "webpack-dev-server --open --mode development",
  "build": "webpack -p",
  + "test": "jest"
},
```

然后就可以通过 npm run test 来执行测试命令了。至此项目 Jest 的环境配置完成。

8.4.2 测试 Form 组件视图和单击事件

现在正式编写测试代码。在./src 下新建文件夹 `_test_`，然后新建测试文件 `Form.test.js`：

```
mkdir -p src/_tests_
cd src/_test_
touch Form.test.js
```

先来测试 Form 组件的视图是否正常渲染：

```
import React from "react";
import { shallow } from "enzyme";
// 引入被测试组件
import Form from "../components/Form";
// case1 测试组件是否正常渲染
describe("FormView", () => {
```

```

it("Form Component should be render", () => {
  const wrapper = shallow(<Form />);
  expect(wrapper.find("input").exists()).toBeTruthy();
  expect(wrapper.find("button").exists()).toBeTruthy();
});
});

```

上述代码中，先从./src/components 中引入 Form 组件，用于测试。上面通过查找组件内部是否存在 input 和 button 来测试组件是否正常渲染。其中.find(selector)是 Enzyme 提供的 shallow Rendering 语法，用于查找节点。详细用法见 Enzyme 文档：<http://airbnb.io/enzyme/docs/api/shallow.html>。上面测试案例中 expect(wrapper.find("input").exists()).toBeTruthy() 和 expect(wrapper.find("button").exists()).toBeTruthy() 用了 expect 断言，可以判断组件内视图层结构是否被破坏。

在终端执行测试命令：

```
npm run test
```

打印结果如图 8.1 所示。

```
[AllandeMacBook-Pro:react-jest-enzyme alan$ npm run test
> myfirstapp@1.0.0 test /Users/alan/Desktop/react-jest-enzyme
> jest

PASS  src/__tests__/_Form.test.js
  FormView
    ✓ Form Component should be render (11ms)

  Test Suites: 1 passed, 1 total
  Tests:       1 passed, 1 total
  Snapshots:   0 total
  Time:        1.398s
  Ran all test suites.
```

图 8.1 Form 组件视图测试通过

这就表明执行了一则测试，测试通过，且用时 1.398 秒。假如有开发者误删了按钮，则代码改变：

./src/components/Form.js:

```

import React from "react";
const Form = props => (
  <div className="form-wrap">
    <input value={props.todoItem} onChange={props.onChange} />
    {/* <button onClick={props.onSubmit}>Submit</button> */}
  </div>
);
export default Form;

```

执行 npm run test 会出现的结果，如图 8.2 所示。

```
AllandMacBook-Pro:react-jest-enzyme alan$ npm run test
> myfirstapp@1.0.0 test /Users/alan/Desktop/react-jest-enzyme
> jest

 FAIL  src/__tests__/_Form.test.js
  FormView
    × Form Component should be render (22ms)

      ● FormView › Form Component should be render

        expect(received).toBeTruthy()

        Received: false

          10 |   const wrapper = shallow(<Form />);
          11 |
          > 12 |   expect(wrapper.find("button").exists()).toBeTruthy();
          |   ^
          |   expect(wrapper.find("input").exists()).toBeTruthy();
          | });
          15 | });

      at Object.<anonymous> (src/__tests__/_Form.test.js:12:45)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        1.666s
Ran all test suites.
npm[1] code ELIFECYCLE
npm[1] errno 1
npm[1] myfirstapp@1.0.0 test: `jest`
npm[1] Exit status 1

Failed at the myfirstapp@1.0.0 test script.
This is probably not a problem with npm. There is likely additional logging
output above.

npm[1] A complete log of this run can be found in:
npm[1]   /Users/alan/.npm/_logs/2018-11-22T12_02_24_032Z-debug.log
```

图 8.2 Form 组件视图测试不通过

下面来测试单击事件的功能测试，在`_test__/_Form.test.js`中继续编写第二则测试案例。代码如下：

```
// case2 测试组件单击事件是否能正常执行
describe("executes a handler function on button", () => {
  const mockEvent = {
    onSubmit: jest.fn()
  };
  it("Onsubmit works", () => {
    // 通过 shallow
    const wrapper = shallow(<Form onSubmit={mockEvent.onSubmit} />);
    // 通过 find 查找 button
    const button = wrapper.find("button");
    // 模拟提交
    button.simulate("click");
    expect(mockEvent.onSubmit).toBeCalled();
  });
});
```

上述代码中，用 `jest.fn()` 来模拟单击事件，用 `shallow` 来渲染 Form 组件，找到 button 按钮。当单击按钮时，模拟单击提交事件，用 `expect(mockEvent.onSubmit).toBeCalled()` 来断言是否测试成功。

此时完成对 Form 的组件测试，`Form.test.js` 代码清单：

```
import React from "react";
import { shallow } from "enzyme";
// 引入被测试组件
import Form from "../components/Form";
// case1 测试组件是否正常渲染
// 通过查找，存在 input 和 button，测试组件正常渲染
describe("FormView", () => {
  it("Form Component should be render", () => {
    const wrapper = shallow(<Form />);
    expect(wrapper.find("button").exists()).toBeTruthy();
    expect(wrapper.find("input").exists()).toBeTruthy();
  });
});
// case2 测试组件单击事件是否能正常执行
describe("executes a handler function on button", () => {
  const mockEvent = {
    onSubmit: jest.fn()
  };
  it("Onsubmit works", () => {
    // 通过 shallow
    const wrapper = shallow(<Form onSubmit={mockEvent.onSubmit} />);
    // 通过 find() 查找 button
    const button = wrapper.find("button");
    // 模拟提交
    button.simulate("click");
    expect(mockEvent.onSubmit).toBeCalled();
  });
});
```

8.4.3 测试 ListItems 组件视图

在 `./src/_tests_` 文件内新建文件 `ListItems.test.js`:

```
mkdir -p src/_tests_
cd src/_test_
touch ListItems.test.js
```

编写测试代码如下：

```
import React from "react";
import { shallow } from "enzyme";
// 引入被测试组件
import ListItems from "../components/ListItems";
// case1 测试组件是否正常渲染
describe("ListItemsView", () => {
  it("ListItemsView Component should be render", () => {
```

```
const setup = () => {
  // 模拟 props
  const props = {
    items: [1, 2]
  };
  // 通过 enzyme 提供的 shallow(浅渲染) 创建组件
  const wrapper = shallow(<ListItems {...props} />);
  return {
    props,
    wrapper
  };
};
const { wrapper, props } = setup();
expect(wrapper.find("li").exists()).toBeTruthy();
});
});
```

上述代码中，通过向组件内部传入属性 items 数组查看 li 标签来判断测试是否通过。本节源码可以通过 <https://github.com/khno/react-jest-enzyme> 访问。

第9章 实战——React+Redux 搭建社区项目

通过前面章节的学习，相信读者已经理解了 React 和 Redux 的相关知识。本章通过搭建一个常见的社区项目来加深读者对之前章节的理解和巩固。该项目的目的在于让读者系统地认识和了解一个完整的项目体系，万变不离其宗，完成这个项目能够让读者在今后的工作或项目实践上有所领悟和帮助。

9.1 项目结构

本项目将基于上一章搭建的脚手架继续深入开发，脚手架源码地址为：<https://github.com/khno/react-jest-enzyme>。本项目涉及的技术栈有 React、Redux、react-router v4 和 Webpack 4 等。

从业务上分，社区项目包含首页/列表页、详情页、个人中心，其中包含了各类常见的功能。项目正式开始前，需要搭建基础配置：Less 的编译、路由和 Redux 等。读者可以下载脚手架跟着本章内容一步步完成项目的搭建。

9.2 Less 文件处理

为了方便书写样式，需要在 Webpack 配置文件中配置用于加载.less 后缀文件的 loader。

(1) 安装 loader：

```
npm install less less-loader -dev-save
```

(2) 在 webpack.config.js 的 rules 中增加：

```
{
  test: /\.less$/,
  use: [
    {
      loader: "style-loader",           // 将 JS 字符串生成为 style 节点
    },
  ],
}
```

```

    {
      loader: "css-loader"           // 将 CSS 转化成 CommonJS 模块
    },
    {
      loader: "less-loader"         // 将 Less 编译成 CSS
    }
  ]
}

```

9.3 路由和 Redux 配置

路由是 React 项目中不可或缺的技术组成部分，Redux 的运用可以让项目数据管理更加便利。本节来向读者展示如何在项目中配置和使用路由及 Redux。

9.3.1 前期配置

本例路由采用 react-router v4。

(1) 首先安装路由和 Redux：

```
npm install react-router-dom redux -dev-save
```

(2) 修改 src/app.js 为：

```

import React from "react";
import { render } from "react-dom";
import { HashRouter } from "react-router-dom";
import { Provider } from "react-redux";
import configureStore from "./store/configureStore";           // Redux Store
import AppRouter from "./containers/AppRouter.jsx";           // 路由
const store = configureStore();
render(
  <HashRouter>
    <Provider store={store}>
      <AppRouter />
    </Provider>
  </HashRouter>,
  document.getElementById("app")
);

```

这是 Webpack 打包的入口文件，路由中使用 hashRouter，然后用 Redux 的 Provider 将 Store 注入整个项目。

(3) 根据上面引入的文件，需要在 src 下新建文件夹 store，结构如下：

```

store
├── configureStore.dev.js
└── configureStore.js

```

└─ configureStore.prod.js

其中，configureStore.js 内容为：

```
if (process.env.NODE_ENV === "production") {
    module.exports = require("./configureStore.prod");
} else {
    module.exports = require("./configureStore.dev");
}
```

上述代码表示在开发环境中使用 configureStore.dev.js，在生产环境中使用 configureStore.prod.js。接下来是两个文件的代码。

configureStore.dev.js:

```
import { createStore, applyMiddleware, compose } from "redux";
import thunk from "redux-thunk";
import { hashHistory } from "react-router-dom";
import { routerMiddleware } from "react-router-redux";
import rootReducer from "../reducers";
const router = routerMiddleware(hashHistory);
const enhancer = compose(
    applyMiddleware(thunk, router),
    window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()
);
export default function configureStore(initialState) {
    const store = createStore(rootReducer, initialState, enhancer);
    if (module.hot) {
        module.hot.accept("../reducers", () => {
            const nextReducer = require("../reducers").default; // eslint-
            disable-line global-require
            store.replaceReducer(nextReducer);
        });
    }
    return store;
}
```

configureStore.prod.js:

```
import { createStore, applyMiddleware } from "redux";
import thunk from "redux-thunk";
import rootReducer from "../reducers";
export default function configureStore(initialState) {
    return createStore(rootReducer, initialState, applyMiddleware(thunk));
}
```

(4) 这里需要强调一下，Redux 应用只有一个单一的 Store。当需要拆分数据处理逻辑时，应该使用 Reducer 组合(combine)而不是创建多个 Store。新建文件 src/reducers/index.js：

```
import { combineReducers } from "redux";
import { aReducer, bReducer } from "reducersPath";
```

```
export default combineReducers({
  // aReducer,
  // bReducer
});
```

该文件的作用就是将需要存储的数据根据组件进行拆分，最终整合到全局唯一的 Store 中。

至此，路由和 Redux 在项目中的基本建设已配置完成，接下来测试路由是否可以正常使用。

9.3.2 路由功能的测试

根据 <https://reacttraining.com/react-router/web/guides/quick-start> react-router v4 官网的示例代码来测试。新建路由文件 src/containers/AppRouter.jsx：

```
import React from "react";
import { BrowserRouter as Router, Route, Link } from "react-router-dom";
const Index = () => <h2>Home</h2>;
const About = () => <h2>About</h2>;
const Users = () => <h2>Users</h2>;
const AppRouter = () => (
  <Router>
    <div>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/about/">About</Link>
          </li>
          <li>
            <Link to="/users/">Users</Link>
          </li>
        </ul>
      </nav>
      <Route path="/" exact component={Index} />
      <Route path="/about/" component={About} />
      <Route path="/users/" component={Users} />
    </div>
  </Router>
);
export default AppRouter;
```

此时浏览器的显示效果，如图 9.1 所示，单击链接，如果能正常跳转，就表明路由配置成功。

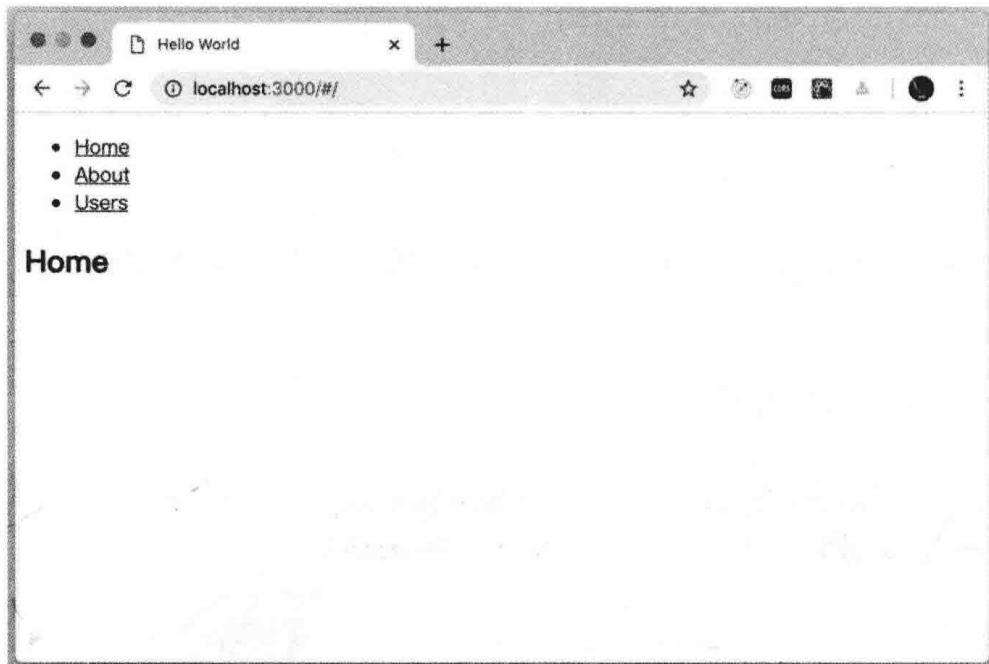


图 9.1 测试路由

9.4 业务入口

接下来开始具体业务代码的开发。我们都知道 React 是组件化的编程，开发前需要规划如何拆分页面模块为组件。这个项目中，页面由 3 块组成，分别是首页/列表页、详情页、个人中心页，3 个页面顶部都用公共头和可以登录的模态框。

业务入口文件配置如下：

```
import React from "react";
import { Route, Switch } from "react-router-dom";
import Header from "../components/Header/index.jsx";
import Home from "./home/index.jsx";
import Details from "./details/index.jsx";
import Mine from "./mine/index.jsx";
class AppRouter extends React.Component {
  render() {
    return (
      <React.Fragment>
        {/* 公用头 */}
        <Header />
        {/* 路由配置 */}
        <Switch>
          {/* 首页/列表 */}
          <Route exact path="/" component={Home} />
          {/* 个人中心 */}
        
      </React.Fragment>
    );
  }
}
```

```

        <Route exact path="/mine" component={Mine} />
        {/* 详情页 */}
        <Route exact path="/details/:id" component={Details} />
    </Switch>
</React.Fragment>
);
}
}
export default AppRouter;

```

这个文件中包含了公共头、登录用的模态框和各个页面的路由配置。其中 path 为"/"的为首页，/details/:id 为从列表页到详情页的传参。

9.5 首 页

根据页面布局来划分组件，首先完成静态页面布局展示。首页布局展示效果如图 9.2 所示。



图 9.2 首页布局（列表数据为假数据）

页面的布局从上往下看可以看出由头部和列表（切换菜单和列表）内容组成。组件划分可以将此页面分为两部分：头部和列表内容。

9.5.1 头部

由于头部固定在页面顶部，并且在多个页面需要公用，因此将头部放在 `src/components` 文件夹下，结构如下：

```

└── components
    └── Header
        ├── index.jsx
        └── index.less

```

先来实现 `Header` 头部组件的布局内容，相关文件代码如下：

(1) `src/components/Header/index.jsx` 的代码：

```

import React from "react";
import { Link } from "react-router-dom";
import "./index.less";

const Header = props => {
  return (
    <header>
      <nav className="header-title">
        {/* logo */}
        <div className="header-logo">
          <Link to="/">
            <i className="iconfont">#xe64b;</i>
          </Link>
        </div>
        {/* 登录按钮 */}
        <div className="header-login">
          <button>登录</button>
        </div>
      </nav>
    </header>
  );
};

export default Header;

```

上述代码利用函数式方法声明了 `Header` 组件，这是一个简单的静态页面。其中，`Link` 属于 `react-router-dom` 的方法，专门用来作为路由的跳转。细心的读者会在浏览器的 HTML 文件中发现，其底层其实就是一个 `a` 标签。

(2) `src/components/Header/index.less` 的代码：

```

@blue : #3296FA;
.header-title {
  height: 52px;
}

```

```
line-height: 50px;
margin-bottom: 10px;
padding: 0 16px;
display: flex;
align-items: center;
justify-content: space-between;
box-shadow: 0px 0px 6px #eaeaea;
position: fixed;
width: 100%;
box-sizing: border-box;
background: #fff;
z-index: 1;
.header-logo {
  display: flex;
  -ms-flex-pack: center;
  justify-content: center;
  i {
    color: @blue;
    font-size: 30px;
  }
}
.header-login {
  display: flex;
  button {
    color: @blue;
    font-size: 16px;
  }
}
```

以上为 Header 组件的 Less 样式文件，Webpack 会通过 Less 相关 loader 将其编译为 CSS 内容。这里用 header-title 这个“顶级”类包裹这个组件当中的所有样式，以防应用中样式相互污染。

9.5.2 列表内容

列表内容由两部分组成：菜单和列表。理论上可以将列表内容组件写成两部分，但由于内容较少，无须划分过细。新建文件如下：

```
└ containers
    └ home
        └ index.jsx
        └ index.less
```

(1) 首先实现列表的静态布局内容，相关文件代码如下所示。

src/containers/home/index.jsx 的代码:

```
import React from "react";
import "./index.less";
```

```

export class Home extends React.Component {
  constructor() {
    super();
    this.state = {};
  }
  render() {
    return (
      <div className="home">
        {/* 顶部菜单 */}
        <div className="fix-header-nav">
          <button className="active">推荐</button>
          <button>生活</button>
          <button>科技</button>
        </div>
        {/* 列表 */}
        <div className="list-warp">
          <a className="article-item">
            <h4>这是列表标题</h4>
            <div className="content">
              <img src="" alt="" />
              <p>这是列表内容</p>
            </div>
            <p className="item-footer">须畅 的创作 3 个赞</p>
          </a>
          <a className="article-item">
            <h4>这是列表标题</h4>
            <div className="content">
              <img src="" alt="" />
              <p>这是列表内容</p>
            </div>
            <p className="item-footer">须畅 的创作 3 个赞</p>
          </a>
          <a className="article-item">
            <h4>这是列表标题</h4>
            <div className="content">
              <img src="" alt="" />
              <p>这是列表内容</p>
            </div>
            <p className="item-footer">须畅 的创作 3 个赞</p>
          </a>
        </div>
      </div>
    );
  }
}
export default Home;

```

上述代码为列表的静态 HTML，布局内容上由列表顶部的 tab 菜单和列表主体两部分组成。

src/containers/home/index.less 的代码：

```

.home {
  background: #fff;

```

```
.list-warp{  
padding-top: 96px;  
}  
.article-item {  
padding: 15px 0 14px;  
margin: 0 15px;  
border-bottom: 0.5px solid #efefef;  
outline: none;  
text-decoration: none;  
display: block;  
color: #333;  
}  
h4 {  
overflow: hidden;  
-webkit-box-orient: vertical;  
text-overflow: ellipsis;  
display: -webkit-box;  
-webkit-line-clamp: 2;  
}  
.content {  
display: flex;  
padding-top: 11px;  
-ms-flex-align: center;  
align-items: center;  
width: 100%;  
}  
p {  
-webkit-line-clamp: 2;  
font-size: 15px;  
overflow: hidden;  
font-weight: 400;  
text-overflow: ellipsis;  
display: -webkit-box;  
line-height: 21px;  
letter-spacing: normal;  
color: #444;  
-webkit-box-orient: vertical;  
}  
.item-footer {  
margin-top: 10px;  
color: #999;  
}  
.fix-header-nav {  
padding: 64px 0 10px;  
position: fixed;  
width: 100%;  
background: #fff;  
button {  
display: inline-block;  
width: 58px;  
height: 24px;  
border-radius: 12px;  
background-color: #fff;  
border: 0.5px solid #ebebeb;  
text-align: center;
```

```

margin: 0px 5px;
color: #999;
font-size: 12px;
line-height: 24px;
}
.active {
color: #444;
font-weight: 600;
}
}
.bottom-tips{
padding: 10px 0;
text-align: center;
font-size: 13px;
color: #999;
}
}
}

```

以上为列表的 Less 文件，外层用.home 这个“顶级”类将其包裹，同样是为了防止样式相互之间被污染。

(2) 接下来将使用接口来动态展示列表内容，接口请求使用 axios，用命令安装：

```
npm install axios --save-dev
```

(3) 请求接口获取数据：

```

// 接口请求放在 componentDidMount 生命周期执行
componentDidMount() {
  // page 为当前页码，type 为列表类型：推荐、生活、科技
  this.fetchList({ page: 1, type: 0 });
}

// 列表数据获取
fetchList = (params, isRefresh) => {
  axios({
    url: "/mock/list",
    params: params
  }).then(res => {
    const { result, success } = res.data;
    if (success) {
      let data;
      if (isRefresh) {
        data = result.data;
      } else {
        data = this.state.data.concat(result.data);
      }
      this.setState({
        data,
        page: result.page,
        hasMore: result.hasMore
      });
    }
  });
}

```

上述代码中，使用 axios()方法实现接口请求，其中的 url 为接口地址，params 为接口

的入参。请求方法在组件 `componentDidMount()` 这个声明周期中去请求，请求成功后使用 `setState()` 方法对数据进行渲染。`isRefresh` 标记为是否刷新当前页面，如果 `isRefresh` 为真，直接将接口返回数据复制给 `data`，否则在原先的数组中继续拼接。最后 `setState()` 当前接口返回的数据，其中 `hasMore` 标记为后端返回，用于判断是否存在下一页，`page` 为当前页面索引。

(4) 由于列表的数据只需在该组件内使用，所以不需要存入全局的 Store。将上面存入 `state` 的列表数据，直接在 `render()` 中展示，如下：

```
render() {
  const { data, hasMore, active } = this.state;
  return (
    <div className="home">
      /* 其他代码 */
      /* 列表 */
      <div className="list-warp">
        {data.map((item, index) => {
          return (
            <a
              className="article-item"
              key={index}
            >
              <h4>{item.title}</h4>
              <div className="content">
                <p>{item.content}</p>
              </div>
              <p className="item-footer">
                {item.name} 的创作 {item.num} 个赞
              </p>
            </a>
          );
        })}
      </div>
    </div>
  );
}
```

(5) 为了让接口请求时有一个 `loading` 加载的效果，可新建一个 `Loading` 的公共组件。此时结构如下：

```

└── components
  └── Loading
    ├── index.js
    └── index.less

```

`src/components>Loading/index.js` 的代码：

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.less";
export default class Loading extends React.Component {
```

```

constructor(props) {
  super(props);
}
render() {
  return (
    <div className="init-loading-wrapper">
      <div className="init-loading">
        <div className="loading-ring">
          <div className="loading-ball-holder">
            <div className="loading-ball1" />
            <div className="loading-ball2" />
            <div className="loading-ball3" />
            <div className="loading-ball4" />
          </div>
        </div>
      </div>
    </div>
  );
}
const showLoading = () => {
  const wrapper = document.createElement("div");
  ReactDOM.render(<Loading />, wrapper);
  document.body.appendChild(wrapper);
  return wrapper;
};
const hideLoading = wrapper => {
  wrapper && document.body.removeChild(wrapper);
};
export const addLoading = function() {
  if (!window.loadingWrapper) {
    window.loadingWrapper = showLoading();
  }
};
export const removeLoading = function() {
  if (window.loadingWrapper) {
    hideLoading(window.loadingWrapper);
    window.loadingWrapper = null;
  }
};

```

以上代码为 loading 效果的布局和对外的方法，其中 addLoading 和 removeLoading 分别为添加 loading 效果和移除 loading 效果。

注意：为了不让 loading 的显示和隐藏给浏览器造成重绘和重排，可以将 Loading 这个组件添加到整个 body 的最后。这样，添加和移除就不会给原先的 DOM 节点造成影响。

src/components>Loading/index.less 的代码：

```

.init-loading-wrapper {
  position: absolute;
  top: 0;
  left: 0;
}

```

```
width: 100%;  
height: 100%;  
z-index: 10000;  
}  
.init-loading{  
position: absolute;  
top: 50%;  
left: 50%;  
margin-top: -25px;  
margin-left: -25px;  
width: 50px;  
height: 50px;  
border-radius: 8px;  
z-index: 9999;  
box-sizing: border-box;  
}  
.loading-ring {  
position: relative;  
width: 48px;  
height: 48px;  
margin: 0 auto;  
border: 2px solid #9C27B0;  
border-radius: 100%;  
border: hidden;  
}  
.loading-ball-holder {  
position: absolute;  
width: 12px;  
height: 48px;  
left: 18px;  
top: 0;  
animation: loading-ball 1.3s linear infinite;  
}  
.loading-ball1 {  
position: absolute;  
width: 12px;  
height: 12px;  
border-radius: 100%;  
background: #F25643;  
}  
.loading-ball2 {  
position: absolute;  
bottom: 0;  
width: 12px;  
height: 12px;  
border-radius: 100%;  
background: #15BC83;  
}  
.loading-ball3 {  
position: absolute;  
top: 18px;
```

```

left: -18px;
width: 12px;
height: 12px;
border-radius: 100%;
background: #3296FA;
}
.loading-ball4{
position: absolute;
top: 18px;
right: -18px;
width: 12px;
height: 12px;
border-radius: 100%;
background: #FF943E;
}
@keyframes loading-ball {
0 {
    transform:rotate(0deg)
}
30% {
    transform: rotate(190deg);
}
100% {
    transform:rotate(360deg)
}
}
}

```

上述代码为 Loading 的 Less 文件，其中用到了 CSS 3 动画。

(6) 将 Loading 组件应用在列表的接口调用中，如下：

```

fetchList = (params, isRefresh) => {
    // 展示 loading 效果
    addLoading();
    axios({
        url: "/api/list",
        params: params
    }).then(res => {
        const { result, success } = res.data;
        if (success) {
            // 移除 loading 效果
            removeLoading();
            let data;
            if (isRefresh) {
                data = result.data;
            } else {
                data = this.state.data.concat(result.data);
            }
            this.setState({
                data,
                page: result.page,
                hasMore: result.hasMore
            });
        }
    })
}

```

```
});  
};
```

当数据请求未返回时，页面展示效果如图 9.3 所示，请求数据返回时展示列表，loading 效果消失。

当然还有更合适的实现方式，那就是将 axios 进行封装，在每次接口请求时都打开 loading 效果，这里不作展开介绍。

以上就是列表部分的展示。当用户单击列表欲进入详情页的时候，根据用户的权限进行拦截，已登录的用户有权限进入详情页，否则，弹出需要用户登录的模态框。当然这个模态框也可通过单击页面顶部右上角的“登录”按钮打开，展示效果相同，如图 9.4 所示。



图 9.3 loading 效果



图 9.4 用户登录模态框

创建模态框，代码在项目中的结构为：

```
└── containers  
    └── login  
        └── ModalContainer.js
```

```

  └── index.jsx
    └── index.less

```

下面展示这个模态框的相关代码。

src/containers/login/index.jsx 的代码:

```

import React, { Component } from "react";
import { connect } from "react-redux";
import { bindActionCreators } from "redux";
import ModalContainer from "./ModalContainer";
import { signinUser, loginModalhide } from "../../actions/index";
import "./index.less";
class Login extends Component {
  constructor(props) {
    super(props);
    this.state = {
      userName: "",
      password: ""
    };
  }
  // 表单 onchange 事件
  handleChange = event => {
    const target = event.target;
    if (target.type === "text") {
      this.setState({
        userName: target.value
      });
    }
    if (target.type === "password") {
      this.setState({
        password: target.value
      });
    }
  };
  // 表单提交
  toLogin = () => {
    const { userName, password } = this.state;
    const { signinUser, loginModalhide } = this.props;
    // 模拟后端验证账号和密码
    if (userName === "" || userName !== "admin") {
      alert("请输入正确账号！");
      return;
    } else if (password === "" || password !== "123456") {
      alert("请输入正确的密码！");
      return;
    }
    signinUser({ userName, password })
      .then(res => {
        // 接口请求成功，关闭弹窗
        if (res.success) {
          loginModalhide();
        }
      })
      .catch(err => {
        // 处理错误逻辑
      });
  };
}

```

```

    // 接口报错处理
    console.log(err);
  });
};

render() {
  const { isLoginModalShow, loginModalhide } = this.props;
  const { visible } = isLoginModalShow;
  const divStyle = {
    zIndex: visible ? "1" : "-1"
  };
  return (
    <ModalContainer>
      <div
        className={`${visible ? "modal fade show" : "modal fade"}`}
        style={divStyle}
      >
        <div className="modal-backdrop" />
        <div className="modal-dialog">
          <div className="modal-content">
            <div className="modal-header">
              <i onClick={loginModalhide} className="iconfont">
                &#xe85c;
              </i>
              <h4 className="modal-title">登录</h4>
            </div>
            <div className="modal-body">
              <input
                type="text"
                placeholder="请输入账号: admin"
                onChange={this.handleChange}
              />
              <input
                type="password"
                placeholder="请输入密码: 123456"
                onChange={this.handleChange}
              />
              <input
                type="submit"
                className="submit-btn"
                value="登录"
                onClick={this.toLogin}
              />
              <a href="">忘记密码? </a>
              <p className="login-tips">登录即表示您同意《用户协议》</p>
            </div>
          </div>
        </div>
      </div>
    </ModalContainer>
  );
}
}

const mapStateToProps = state => ({
  isLoginModalShow: state.isLoginModalShow
})

```

```

    });
const mapDispatchToProps = dispatch => {
  return bindActionCreators(
    {
      signinUser,
      loginModalhide
    },
    dispatch
  );
};

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Login);

```

上述代码中，实现了模态框的布局和业务逻辑，将模态框的显示隐藏的标记 isLoginModalShow 交由 Redux 来完成，这样便于全局共享这个状态。signinUser()用于用户提交登录表单，成功后调用 loginModalhide 这个 Action 来触发 dispatch()修改 Store 中模态框隐藏。

src/containers/login/ModalContainer.js 的代码：

```

import { Component } from "react";
import { createPortal } from "react-dom";
export default class ModalContainer extends Component {
  constructor(props) {
    super(props);
    const doc = window.document;
    this.node = doc.createElement("div");
    doc.body.appendChild(this.node);
    window.document.body.appendChild(window.document.createElement("div"));
  }
  componentWillUnmount() {
    window.document.body.removeChild(this.node);
  }
  render() {
    return createPortal(this.props.children, this.node);
  }
}

```

从上述代码中可以看到，模态框 Login 外层嵌套了一个 ModalContainer 组件，Modal Container 是一个高阶组件，在本例中的作用是让模态框的显示隐藏能够避免不必要的重绘和重排。这里使用了 react-dom 中的 createPortal()方法，这个方法是一个“传送门”，它能够提供一种方式让读者把想要渲染的 DOM 放到其他地方。其实现效果与 9.4.2 节的 Loading 组件类似。

`ReactDOM.createPortal(child, container)`

由于模态框是一个独立的组件，展示在页面中央，所以不应该嵌套在其他业务代码中，可以通过 createPortal()将它放在整个 body 的最底部。有关于 createProtal()更多内容见官方文档：<https://reactjs.org/docs/react-dom.html#createportal>。

src/containers/login/index.less 的代码：

```
.modal {  
  position: fixed;  
  top: 0;  
  right: 0;  
  bottom: 0;  
  left: 0;  
  z-index: 1050;  
  outline: 0;  
}  
.modal-content {  
  height: 80%;  
  position: relative;  
  display: -ms-flexbox;  
  display: -webkit-box;  
  display: flex;  
  -ms-flex-direction: column;  
  -webkit-box-orient: vertical;  
  -webkit-box-direction: normal;  
  flex-direction: column;  
  width: 100%;  
  pointer-events: auto;  
  background-color: #fff;  
  background-clip: padding-box;  
  border-radius: 12px;  
  outline: 0;  
}  
.modal-header {  
  h4 {  
    font-size: 24px;  
    font-weight: 500;  
    margin-bottom: 20px;  
    margin-top: 32px;  
    padding: 0px 30px;  
  }  
  .iconfont {  
    padding: 10px;  
    float: right;  
  }  
}  
.modal-dialog {  
  position: relative;  
  width: auto;  
  margin: 0.5rem;  
  pointer-events: none;  
  display: flex;  
  -webkit-box-align: center;  
  align-items: center;  
  min-height: calc(100% - (0.5rem * 2));  
}  
.modal-body {  
  padding: 20px 30px;  
  input {  
    display: block;  
    width: 100%;  
  }  
}
```

```
height: calc(2.25rem + 2px);
padding: 0.375rem 0.75rem;
font-size: 1rem;
line-height: 1.5;
color: #495057;
background-color: #fff;
background-clip: padding-box;
border: 1px solid #ced4da;
border-radius: 0.25rem;
-webkit-transition: border-color 0.15s ease-in-out,
-webkit-box-shadow 0.15s ease-in-out;
transition: border-color 0.15s ease-in-out,
-webkit-box-shadow 0.15s ease-in-out;
transition: border-color 0.15s ease-in-out, box-shadow 0.15s ease-in-out;
transition: border-color 0.15s ease-in-out, box-shadow 0.15s ease-in-out,
-webkit-box-shadow 0.15s ease-in-out;
margin-bottom: 15px;
}
.submit-btn{
background: #4170ea;
// background: @theme-color;
color: #fff;
}
a{
color: #4170ea;
font-size: 14px;
}
.login-tips{
font-size: 12px;
color: #999;
text-align: center;
margin-top: 55px;
}
.modal-backdrop {
position: fixed;
top: 0;
right: 0;
bottom: 0;
left: 0;
background: rgba(0, 0, 0, 0.5);
}
.fade {
transition: opacity 0.15s linear;
&:not(.show) {
opacity: 0;
}
.modal-dialog {
transition: transform 0.3s ease-out, -webkit-transform 0.3s ease-out;
transform: translate(0, -25%);
}
}
```

```
.show {
  opacity: 1;
  .modal-dialog {
    transform: translate(0, 0);
  }
}
```

以上为模态框的 Less 样式文件。

模态框的展示与隐藏通过 visible 字段进行判断，其值为，真表示展示，为假表示隐藏。这里使用到 Redux，visible 存于全局的 Store 中，以便于组件之间数据的共享。模态框的打开通过 Action 发起，代码如下：

src/actions/index.js

```
// 登录模态框关闭
const LOGIN_MODAL_HIDE = "login_modal_hide";
const LOGIN_MODAL_SHOW = "login_modal_show";
// 登录模态框展示
export const loginModalShow = () => dispatch => {
  dispatch({ type: LOGIN_MODAL_SHOW });
};
// 登录模态框关闭
export const loginModalhide = () => dispatch => {
  dispatch({ type: LOGIN_MODAL_HIDE });
};
```

在业务代码中，将通过调用 loginModalhide 方法来触发 dispatch() 的一个 Action，从而改变 Store 中 isLoginModalShow 的值。Action 本质是一个 JS 对象，除了 type 之外，Action 对象的结构可以自己定义。其中 type 为将要执行的动作，以字符串形式出现，一般情况下会有一个对象用于传递数据的载体（payload）。这里用到一个 type，在下面 Reducer 文件中，根据 type 类型来修改模态框 visible 的值。当业务组件中调用 loginModalhide 这个方法的时候，模态框展示的标记 visible 为 false，将会隐藏。同理反之，调用 loginModalShow 方法将会打开模态框。

src/reducers/auth_reducer.js 的代码：

```
import {
  LOGIN_MODAL_SHOW,
  LOGIN_MODAL_HIDE
} from "../actions/types";
export function isModalShowReducer(state = { visible: false }, action) {
  switch (action.type) {
    case LOGIN_MODAL_SHOW:
      return { ...state, visible: true };
    case LOGIN_MODAL_HIDE:
      return { ...state, visible: false };
    default:
      return state;
  }
}
```

上述代码是整个应用中的其中一个 Reducer，项目中会有很多个 Reducer，一个 Reducer

函数独立负责整个应用中 state 的一部分。所以此时使用 `combineReducers()` 辅助函数的作用是，把一个由多个不同 Reducer 函数作为 value 的 object，合并成一个最终的 Reducer 函数，然后可以对这个 Reducer 调用 `createStore()` 方法。由 `combineReducers()` 返回的 state 对象，会将传入的每个 Reducer 返回的 state 按其传递给 `combineReducers()` 时对应的 key 进行命名。本例中的代码如下：

`src/reducers/index.js`

```
import { combineReducers } from "redux";
import { authReducer, isModalShowReducer } from "./auth_reducer.js";
export default combineReducers({
  auth: authReducer,
  isLoginModalShow: isModalShowReducer
});
```

前面已经定义好了 Action，接下来需要将 Action 与业务组件进行关联，这时会用到 `connect` 方法。需要将打开模态框的动作（Action）关联到组件 Header，当单击“登录”按钮的时候触发这个动作（Action）。Header 组件中代码实现如下：

```
import React from "react";
import { connect } from "react-redux";
import { bindActionCreators } from "redux";
import { Link } from "react-router-dom";
// loginModalShow 方法来自 action
import { loginModalShow } from "../../actions/index";
import "./index.less";
const Header = props => {
  const { loginModalShow } = props;
  return (
    <header>
      <nav className="header-title">
        ...
        {/* 登录按钮 */}
        <div className="header-login">
          <button onClick={loginModalShow}>登录</button>
        </div>
        ...
      </nav>
    </header>
  );
};
const mapDispatchToProps = dispatch => {
  // 注意 bindActionCreators 的使用：当需要把 Action Creator 传到一个组件上时使用
  // 将 loginModalShow 这个 Action Creator 传给当前组件 Header 使用
  return bindActionCreators(
    {
      loginModalShow
    },
    dispatch
  );
};
```

```
export default connect(
  null,
  mapDispatchToProps
)(Header);
```

上述代码中，当单击页面顶部的“登录”按钮，将会调用 loginModalShow 方法，这个方法会触发 dispatch 一个 Action，从而改变 Store 中 visible 的值，当 visible 改变后会触发 Login 组件重渲染，从而打开模态框。

那么，为什么 Store 中 Visible 的值改变会触发 Login 组件重渲染呢？接下来看看 Login 组件（即登录的模态框）：

```
import React, { Component } from "react";
import { connect } from "react-redux";
import ModalContainer from "./ModalContainer";
import "./index.less";
class Login extends Component {
  constructor(props) {
    super(props);
    this.state = {};
  }
  render() {
    const { isLoginModalShow, loginModalhide } = this.props;
    const { visible } = isLoginModalShow;
    return (
      <ModalContainer>
        <div
          className={`${visible ? "modal fade show" : "modal fade"}`}
        >
          ...
        </div>
      </ModalContainer>
    );
  }
}
const mapStateToProps = state => ({
  isLoginModalShow: state.isLoginModalShow
});
export default connect(
  mapStateToProps,
  null
)(Login);
```

上述代码中，通过 mapStateToProps 方法将组件跟 Store 的 state 建立映射关系。将存于 Store 中的 isLoginModalShow 放在当前组件使用，用于判断模态框的展示或隐藏。在 React 中，知道组件自身的 state 变化或外部传入的 props 改变都会触发 render() 的重渲染，所以当 isLoginModalShow 的状态发生变化，会重新渲染当前组件，也就实现了模态框的显示与隐藏。

上面围绕模态框的显示和隐藏的标记 visible 来使用了 Redux，下面来回顾一下有关 Redux 的内容：

- 定义 Action: 用户单击“登录”按钮的方法就是触发 Action, 通过方法触发 dispatch 一个 Action 来改变 Store 中的值。通俗点理解, Action 就是用来描述“发生了什么”。
- 定义 Reducer: 根据 Action 发出的 type 来做某件事。
- Store: 就是把 Reducer 关联到一起的一个对象, 提供了 dispatch(action)方法更新 state, getState 方法获取 state。

要理解 Redux 的使用, 必须先理清 Action、Reducer 和 Store 之间的关系, 简单示例:

```
import book from './reducers'
import { deleteBook } from './action/book'
let store=createStore(book) //关联 reducer
store.dispatch(deleteBook);
```

下面再来看看如何触发 Action, 然后修改 state 起到更新界面的作用:

```
// 顶部 Header 组件
import { loginModalShow } from "../../actions/index";
...
<button onClick={loginModalShow}>登录</button>
...
const mapDispatchToProps = dispatch => {
  // 注意: bindActionCreators 使用: 当需要把 Action Creator 传到一个组件上时使用
  // 将 loginModalShow 这个 Action Creator 传给当前组件 Header 使用
  return bindActionCreators(
    {
      loginModalShow
    },
    dispatch
  );
};
export default connect(null, mapDispatchToProps)(Header);
```

用 connect 将组件包裹, 从而与 Store 建立联系, 再用 mapDispatchToProps 方法将 Action 作为 props 绑定到组件中。当单击“登录”按钮时, 则调用了这个 Action, 修改 Store 中的 visible。

```
// 模态框组件
...
render() {
  const { isLoginModalShow, loginModalhide } = this.props;
  const { visible } = isLoginModalShow;
  return (
    <ModalContainer>
      <div
        className={`${visible ? "modal fade show" : "modal fade"}`}
        style={divStyle}
      >
        ...
      </div>
    </ModalContainer>
```

```

    );
}

...
const mapStateToProps = state => ({
  isLoginModalShow: state.isLoginModalShow
});
export default connect(mapStateToProps, null)(Login);

```

模态框组件中，依旧通过用 `connect` 将组件包裹与 `Store` 建立联系，再用 `mapStateToProps` 方法将 `Store` 中的数据作为 `props` 绑定到组件中。而模态框的展示与否，通过来自 `Store` 的 `visible` 来判断，当 `visible` 改变时，模态框重渲染，发生显示或隐藏。

虽然 `Redux` 的写法有点“绕”，但理清其相互之间的关系，理解起来还是很简单的。

回到本节案例，以上内容实现了单击顶部 `Header` 组件的“登录”按钮操作。当用户没有登录时，没有权限并且无法单击列表到详情页，此时依旧弹出登录模态框用于拦截。相关代码如下：

`src/containers/home/index.jsx:`

```

...
// 详情页跳转，没有登录时需要先登录
toDetails = id => {
  const { authenticated, loginModalShow } = this.props;
  if (authenticated) {
    const { history } = this.props;
    history.push({
      pathname: `/details/${id}`
    });
  } else {
    loginModalShow();
  }
};

...
render() {
  const { data, hasMore, active } = this.state;
  return (
    <div className="home">
      ...
      <div className="list-warp">
        {data.map((item, index) => {
          return (
            <a
              className="article-item"
              key={index}
              onClick={() => this.toDetails(item.id)}
            >
              ...
            </a>
          );
        })}
      </div>
    </div>
  );
}

```

```

}
...

// 将 store 中的数据 authenticated 作为 props 绑定到组件
const mapStateToProps = state => ({
  authenticated: state.auth.authenticated
});
// 将 action 作为 props 绑定到组件中
const mapDispatchToProps = dispatch => {
  return bindActionCreators(
    {
      loginModalShow
    },
    dispatch
  );
};
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Home);

```

单击列表，向方法 toDetails() 传入当前 id，用于跳转后传递给详情页（用于根据 id 调用详情接口）。在 toDetails() 方法中，用存于 Store 中 auth 对象中的 authenticated 属性来判断是否有权限。没有权限则调用 loginModalShow 方法打开登录模态框。

至此，首页内容完成。

9.6 详 情 页

通过单击首页内容可以进入详情页，本节将实现详情页的页面布局、渲染等内容。

9.6.1 静态页面开发

先来看详情页的布局，如图 9.5 所示。

从图 9.5 可以知道，这个页面包含头部及正文，头部内容依旧引用之前写好的组件 Header 来做展示。正文内容含有作者头像、姓名、发布时间这些作者的个人信息，然后是文章的标题和内容，内容底部还有阅读次数、点赞次数。页面内容比较简单，暂时无须分成组件单独处理。

静态页面布局代码如下：

```

import React from "react";
import "./index.less";

class Details extends React.Component {
  constructor() {
    super();
    this.state = {};
  }
}

```

```

}
render() {
  return (
    <div className="details">
      <div className="container">
        <div className="header">
          <a href="">
            
          </a>
        <div className="article-info">
          <span className="user-name">高昌</span>
          <span className="publish-date">3 天前</span>
        </div>
        <h4>这里是标题</h4>
      </div>
      <div className="detail-body">
        <div>这里是内容，还能放入图片。这里是内容，还能放入图片。这里是内容，还能放入图片。</div>
        <span className="read-counts">111 次阅读</span>
        <span className="read-counts">222 次点赞</span>
      </div>
      <div className="detail-footer">
        <i className="iconfont">
          &#xe630;
        </i>
      </div>
    </div>
  );
}
}

export default Details;

```



图 9.5 详情页

样式代码清单如下：

src/containers/details/index.less:

```
.details {  
  padding-top: 70px;  
  padding-left: 10px;  
  padding-right: 10px;  
  .container {  
    background: #fff;  
    padding: 15px;  
    margin-bottom: 10px;  
    border-radius: 4px;  
  }  
  .header {  
    margin-bottom: 10px;  
    border-bottom: 1px solid #efefef;  
    a {  
      display: inline-block;  
      vertical-align: middle;  
      img {  
        width: 40px;  
        height: 40px;  
        border-radius: 50%;  
      }  
    }  
    .article-info {  
      display: inline-block;  
      margin-left: 20px;  
      vertical-align: middle;  
      span {  
        display: block;  
      }  
      .publish-date {  
        font-size: 13px;  
        color: #999;  
        margin-top: 12px;  
      }  
    }  
    h4 {  
      font-size: 25px;  
      line-height: 30px;  
      font-weight: 500;  
      margin: 15px 0;  
    }  
  }  
  .detail-body {  
    border-bottom: 1px solid #efefef;  
    margin-bottom: 20px;  
    img {  
      width: 100%;  
    }  
    p {  
      line-height: 25px;  
      margin-bottom: 16px;  
    }  
  }  
}
```

```
.read-counts {
    margin: 0px 0px 10px;
    display: inline-block;
    font-size: 14px;
    color: #999;
    margin-right: 10px;
}
.detail-footer {
    i {
        font-size: 14px;
        color: #999;
        padding: 6px;
    }
}
```

至此，静态页面布局完成。

9.6.2 根据 id 获取详情

接下来通过接口向后端请求真实数据，开发中依旧使用 mock 数据。在之前路由中配置了如下代码：

```
<Route exact path="/details/:id" component={Details} />
```

也就是说，在其他页面跳转过来会附带 id:

```
const { history } = this.props;
history.push({
  pathname: `/details/${id}`
});
```

然后，在详情页可以根据这个 id 向后端请求对应的详情内容：

```
// 在 componentDidMount 这个生命周期请求数据
componentDidMount() {
    this.fetchDetail();
}

// 根据 id 获取详情
fetchDetail = () => {
    const { id } = this.props.match.params;
    axios({
        url: `/mock/details/${id}`
    }).then(res => {
        const { result, success } = res.data;
        if (success) {
            this.setState({
                author: result.author,
                img: result.img,
                publishDate: result.publishDate,
                title: result.title,
                content: result.content,
                readCount: result.readCount,
                favoriteCount: result.favoriteCount
            });
        }
    });
}
```

```
        hasFavorite: result.hasFavorite
    });
}
});
```

9.6.3 渲染内容

向后端请求的数据需要展示在页面上，数据写入 render() 内，代码如下：

```
render() {
  const {
    author,
    img,
    publishDate,
    title,
    content,
    readCount,
    favoriteCount
  } = this.state;
  return (
    <div className="details">
      <div className="container">
        <div className="header">
          <a href="">
            <img src={img} />
          </a>
          <div className="article-info">
            <span className="user-name">{author}</span>
            <span className="publish-date">{publishDate} 天前</span>
          </div>
          <h4>{title}</h4>
        </div>
        <div className="detail-body">
          <div dangerouslySetInnerHTML={{
            __html: content
          }}>
            <span className="read-counts">{readCount} 次阅读</span>
            <span className="read-counts">{favoriteCount} 次点赞</span>
          </div>
          <div className="detail-footer">
            <i className="iconfont" onClick={this.toFavorite}>
              &#xe630;
            </i>
          </div>
        </div>
      </div>
    );
}
```

由于列表正文内容编写来自富文本编辑器，传给后端会保留 HTML 标签，所以此时请求得到的也是带 HTML 标签的内容，如图 9.6 所示。

```
content: "<p>A Simple Component</p><p>React components implement a render() method
```

图 9.6 接口返回的标签

这里需要使用 `dangerouslySetInnerHTML` 正常渲染上面的内容，有关 `dangerouslySetInnerHTML` 的介绍这里不做展开介绍，预知详情可访问：<https://reactjs.org/docs/dom-elements.html#dangerouslysetinnerhtml>。

完整代码如下：

src/containers/details/index.jsx:

```
import React from "react";
import axios from "axios";
import { removeLoading, addLoading } from "../../components>Loading/index";
import "./index.less";
class Details extends React.Component {
  constructor() {
    super();
    this.state = {};
  }
  componentDidMount() {
    this.fetchDetail();
  }
  // 根据 id 获取详情
  fetchDetail = () => {
    const { id } = this.props.match.params;
    addLoading(); // 展示 loading
    axios({
      url: `https://www.easy-mock.com/mock/590766877a878d73716e4067/mock/details/${id}`
    }).then(res => {
      const { result, success } = res.data;
      if (success) {
        removeLoading(); // loading 隐藏
        this.setState({
          author: result.author,
          img: result.img,
          publishDate: result.publishDate,
          title: result.title,
          content: result.content,
          readCount: result.readCount,
          favoriteCount: result.favoriteCount,
          hasFavorite: result.hasFavorite
        });
      }
    });
  };
  // 点赞功能
  toFavorite = () => {
    console.log("点赞!");
  };
  render() {
    const {
```

```

        author,
        img,
        publishDate,
        title,
        content,
        readCount,
        favoriteCount
    } = this.state;
    return (
        <div className="details">
            <div className="container">
                <div className="header">
                    <a href="">
                        <img src={img} />
                    </a>
                    <div className="article-info">
                        <span className="user-name">{author}</span>
                        <span className="publish-date">{publishDate} 天前</span>
                    </div>
                    <h4>{title}</h4>
                </div>
                <div className="detail-body">
                    <div
                        dangerouslySetInnerHTML={{
                            __html: content
                        }}
                    />
                    <span className="read-counts">{readCount} 次阅读</span>
                    <span className="read-counts">{favoriteCount} 次点赞</span>
                </div>
                <div className="detail-footer">
                    <i className="iconfont" onClick={this.toFavorite}>
                        &#xe630;
                    </i>
                </div>
            </div>
        </div>
    );
}
export default Details;

```

至此，详情页内容开发结束。在本例中仅做了接口调用和内容展示，实际中还会含有点赞、文章收藏、用户之间的评论等功能，考虑到这并不是本节要展示给读者的内容，此处省略这部分内容的开发。

9.7 个人中心

本节将带领读者实现个人中心页面的开发。由于静态页面布局比较简单并且前面章节已经介绍过了，所以这里跳过静态页面布局。为了聚焦 Redux 的使用，本节将跳过其他不

相关功能，只对用户登录和登出操作做具体介绍。

9.7.1 分析页面功能

当在首页登录之后，Header 右上角的“登录”按钮变为用户头像，可以单击用户头像进入个人中心，如图 9.7 所示。



图 9.7 个人中心入口

在个人中心页面，包含了登出、我的发布、我的收藏、我的点赞、设置等功能。但由于聚焦在 React 和 Redux 的使用上，在这里只介绍登出功能，其他功能不在本节需要完成范围内。先来看个人中心页面布局，如图 9.8 所示。



图 9.8 个人中心

9.7.2 模拟用户登录和登出

这个应用中，通过在入口 `src/app.js` 中判断 `localStorage` 本地存储中 `token` 字段是否为真来模拟用户是否登录。代码如下：

`src/app.js:`

```
import React from "react";
import { render } from "react-dom";
import { HashRouter } from "react-router-dom";
import { Provider } from "react-redux";
import configureStore from "./store/configureStore";
import { AUTH_USER } from "./actions/types";
import AppRouter from "./containers/AppRouter.jsx";
import "./styles/index.less";
const store = configureStore();
// localStorage 中 token 是否为真，真为登录过有权限，否则没有登录
const token = localStorage.getItem("token");
if (token) {
  store.dispatch({ type: AUTH_USER });
}
render(
  <HashRouter>
    <Provider store={store}>
      <AppRouter />
    </Provider>
  </HashRouter>,
  document.getElementById("app")
);
```

当 `localStorage` 中 `token` 为真，则发起一个 `dispatch()` 修改 `Store` 中 `authenticated` 的值。

`src/reducers/auth_reducer.js:`

```
export function authReducer(state = {}, action) {
  switch (action.type) {
    case AUTH_USER:
      return { ...state, authenticated: true };
    case UNAUTH_USER:
      return { ...state, authenticated: false };
    case AUTH_ERROR:
      return { ...state, error: action.payload };
    case FETCH_MESSAGE:
      return { ...state, message: action.payload };
    default:
      return state;
  }
}
```

这时候，进入应用的首页/列表页，列表能否跳转到详情页，是根据 `Store` 中 `authenticated` 的真假来判断的；同时 `Header` 头右上角展示用户头像或“登录”按钮，也是通过 `authenticated` 的真假来判断的。实际上，全局需要登录状态的地方都应该从 `Store` 的 `authenticated` 来判

断，这就是 Redux 实现了全局各个组件数据共享的点。

所以在个人中心页面，需要登出的功能也必然是通过发起一个 Action 改变 Store 中的登录状态的标记，同时还需要移除本地存储的 token。

先来看看 Action 方法。

src/actions/index.js:

```
// 登出
export const signoutUser = () => dispatch => {
  return new Promise(resolve => {
    localStorage.removeItem("token");
    dispatch({ type: UNAUTH_USER });
    resolve();
  });
};
```

当单击个人中心的“登出”按钮时，应该执行这个方法：

```
import { signoutUser } from "../../actions/index";
...
signoutUser = () => {
  this.props.signoutUser().then(res => {
    alert("登出成功！");
    const { history } = this.props;
    history.push({
      pathname: `/`
    });
  });
}
...
<a
  onClick={this.signoutUser}
  href="javascript:void(0);"
  className="row-item mt15"
>
  <div className="item-name">退出登录</div>
</a>
...
const mapDispatchToProps = dispatch => {
  return bindActionCreators(
    {
      signoutUser
    },
    dispatch
  );
};
export default connect(
  null,
  mapDispatchToProps
)(Mine);
```

这里用了 Promise 方法来执行成功登出之后需要处理的事情，当成功登出之后，页面弹出“登出成功！”的字样，然后跳转到首页。

9.8 实战项目回顾

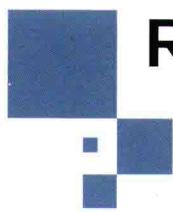
至此，完成了本应用所有内容的开发。本章首先搭建了项目的脚手架，配置了 Less、Redux 和 react-router 等。然后具体业务中聚焦于 React 和 Redux 的搭配使用，完成了首页、详情页、登录、个人中心页的开发。在开发过程中还使用到了后端接口请求、请求过程中 loading 动画效果的封装、登录窗口模态框的封装（含动画效果）等。将重点放在了登录状态用 Redux 在 Store 的储存，出于让读者聚焦学习的目的，放弃了部分应用中应该有的小功能的开发。

万变不离其宗，当读者能理解本章的所有内容后，相信读者能轻松完成其他小功能的开发，也能投身到实际项目中了。

本章项目完整源码地址：<https://github.com/khno/react-starter-kit>。

在本书的最后，留给读者一个问题：前端开发的核心价值是什么？

React+Redux前端开发实战



业内赞誉

近几年前端应用框架发展迅猛，React从中脱颖而出，成为了最受开发者欢迎的技术栈之一。本书中的每个章节都以实战演练的方式展开讲解，让读者在理解理论知识的基础上再加以实践，这对于想要学习React技术栈的朋友们来说是很不错的选择。

——阿里巴巴钉钉前端技术专家 **核心**

React 配合 Redux 具有更加强大的数据管理能力。对于前端开发者而言，这相当于多了一把“利剑”。本书由浅入深、全面细致地介绍了React和Redux前端开发的相关知识，并对其周边开发生态做了介绍。相信读者通过阅读本书可以很好地掌握这些知识，提高前端开发水平。

——宋小菜Scott

React.js是当前最火爆的前端技术栈之一，是一个专注于View层的前端技术栈。由于其独特的设计思想和出众的性能，已经被越来越多的开发者所关注。本书可以带领读者由浅入深、全面系统地学习React.js及Redux生态，迈出作为React全栈开发者的第一步。

——贝贝集团大前端架构负责人 **Early**

本书配套资源获取方式

本书涉及的源代码文件等配套资料需要读者自行下载。请在华章公司的网站www.hzbook.com上搜索到本书，然后单击“资料下载”按钮，即可在本书页面上找到“配书资源”下载链接，单击该链接即可下载。

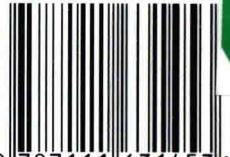
投稿热线：(010) 88379604
购书热线：(010) 68326294
客服热线：(010) 88379426 88361066

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn



上架指导：计算机/前端开发

ISBN 978-7-111-63145-3



9 787111 631453 >

定价：69.00元

2019