

# React.js 小书

[Github](#) [关于作者](#)

这是一本关于 React.js 的小书。

因为工作中一直在使用 React.js，也一直以来想总结一下自己关于 React.js 的一些知识、经验。于是把一些想法慢慢整理书写下来，做成一本开源、免费、专业、简单的入门级别的小书，提供给社区。希望能够帮助到更多 React.js 刚入门朋友。

由于水平有限，编写的过程难免会有诸多错误，也希望大家在看的过程中发现了问题，可以在 [Github](#) 上给该项目发 Pull Request。衷心希望可以有更多的人参与到本书的编写当中。

2017-06-09 更新：本书所有练习题已经在 [ScriptOJ](#) 上线，读者朋友们可以直接在线进行练习。

## 本书介绍

本书为有一点前端基础的并且是 React.js 零基础的同学而作，帮助他们掌握 React.js 并且灵活地把 React.js 应用到实际项目当中。如果你有一定的 HTML、CSS、JavaScript 基础并且希望学习 React.js，而又觉得 React.js 当中有些概念比较难以接受和理解，希望能够从零开始学习，那么本书很适合你。但如果你已经对前端已经非常熟悉并且用过不少的前端框架和相关的组件化技术，建议你直接看官网文档。

本书并不会文档式地包含所有知识点，只会提炼实战经验中基础的、重要的、频繁的知识进行重点讲解，让你能用最少的精力深入了解实战中最需要的 React.js 知识和套路，轻装上路。如果需要更多更全面的知识点，可以参看更多的官方文档或者其他丰富的资料。

**另外，本书全书采用 ECMAScript 2015，阅读之前请确保自己已经掌握了 ECMAScript 2015 的基本语法，否则阅读起来会非常困难。**

本书初定分为三个阶段，每个阶段最后会有实战分析，把该阶段的知识点应用起来。

**第一个阶段：**希望能让读者掌握 React.js 的基本概念和基础知识。包括问题的根源：前端组件的复用性问题、数据和视图的同步问题。了解清楚问题以后再了解 React.js 的基础知识，包括 JSX、事件监听、state、props、列表渲染等。看看 React.js 是怎么解决这些问题的。这个阶段结束以后，读者就可以运用 React.js 构建简单的页面功能。

**第二个阶段：**让读者更进一步了解 React.js，包括组件生命周期及其含义、state 和 props 的进阶概念、props 验证及其意义、组件组合进阶、如何和 DOM 打交道、并且开始引入前端应用状态管理所存在的问题。

**第三个阶段：**让读者掌握 React.js 较为高级的概念，包括高阶组件、context。并且通过引入 React-redux 来协助我们构建较为完整的前端应用，还会开始深入讨论前端应用状态管理的问题；关于 React-router 也会有所提及。

## 目录

### 第一个阶段 已完成 100%

- Lesson 1 - React.js 简介
- Lesson 2 - 前端组件化（一）：从一个简单的例子讲起
- Lesson 3 - 前端组件化（二）：优化 DOM 操作
- Lesson 4 - 前端组件化（三）：抽象出公共组件类
- Lesson 5 - React.js 基本环境安装
- Lesson 6 - 使用 JSX 描述 UI 信息
- Lesson 7 - 组件的 render 方法
- Lesson 8 - 组件的组合、嵌套和组件树
- Lesson 9 - 事件监听
- Lesson 10 - 组件的 state 和 setState
- Lesson 11 - 配置组件的 props
- Lesson 12 - state vs props
- Lesson 13 - 渲染列表数据
- Lesson 14 - 实战分析：评论功能（一）
- Lesson 15 - 实战分析：评论功能（二）
- Lesson 16 - 实战分析：评论功能（三）

### 第二个阶段 已完成 100%

- Lesson 17 - 前端应用状态管理 — 状态提升
- Lesson 18 - 挂载阶段的组件生命周期（一）
- Lesson 19 - 挂载阶段的组件生命周期（二）
- Lesson 20 - 更新阶段的组件生命周期
- Lesson 21 - ref 和 React.js 中的 DOM 操作
- Lesson 22 - props.children 和容器类组件
- Lesson 23 - dangerouslySetHTML 和 style 属性
- Lesson 24 - PropTypes 和组件参数验证
- Lesson 25 - 实战分析：评论功能（四）
- Lesson 26 - 实战分析：评论功能（五）
- Lesson 27 - 实战分析：评论功能（六）

## 第三个阶段 已完成 100%

- Lesson 28 - 高阶组件 (Higher-Order Components)
- Lesson 29 - React.js 的 context
- Lesson 30 - 动手实现 Redux (一) : 优雅地修改共享状态
- Lesson 31 - 动手实现 Redux (二) : 抽离 store 和监控数据变化
- Lesson 32 - 动手实现 Redux (三) : 纯函数 (Pure Function) 简介
- Lesson 33 - 动手实现 Redux (四) : 共享结构的对象提高性能
- Lesson 34 - 动手实现 Redux (五) : 不要问为什么的 reducer
- Lesson 35 - 动手实现 Redux (六) : Redux 总结
- Lesson 36 - 动手实现 React-redux (一) : 初始化工程
- Lesson 37 - 动手实现 React-redux (二) : 结合 context 和 store
- Lesson 38 - 动手实现 React-redux (三) : connect 和 mapStateToProps
- Lesson 39 - 动手实现 React-redux (四) : mapDispatchToProps
- Lesson 40 - 动手实现 React-redux (五) : Provider
- Lesson 41 - 动手实现 React-redux (六) : React-redux 总结
- Lesson 42 - 使用真正的 Redux 和 React-redux
- Lesson 43 - Smart 组件 vs Dumb 组件
- Lesson 44 - 实战分析: 评论功能 (七)
- Lesson 45 - 实战分析: 评论功能 (八)
- Lesson 46 - 实战分析: 评论功能 (九)

---

## 特别鸣谢

特别感谢以下朋友对本书所做的审校工作，给本书提出了很多宝贵的改进意见：

- 邝伟科 - 腾讯 Web 前端工程师
- 杨海波 - 百度 Web 高级前端工程师
- 谢军令 - 天猫 Web 前端工程师
- 戴嘉年华 - 前微信 Web 前端工程师

---

## 联系作者

邮箱: huzidaha@126.com

知乎: [@胡子大哈](#)

专栏: [@前端大哈](#)

有问题上知乎给我私信留言

---

## License

本作品采用 [署名-禁止演绎 4.0 国际许可协议](#) 进行许可

## 《React.js 小书》的PDF版本说明

本《React.js小书》的PDF版本

是由 若川 <https://lxchuan12.cn>

使用node 库 puppeteer爬虫生成， 仅供学习交流，严禁用于商业用途。

文章 前端使用puppeteer 爬虫生成《React.js 小书》PDF并合并：

<https://juejin.im/post/5b86732451882542af1c8082>

项目源代码地址：<https://github.com/lxchuan12/learn-nodejs/tree/master/src/puppeteer/reactMiniBook.js>

---

React.js 小书

[<-- 返回首页](#)

## 1. React.js 简介

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson1>
- 转载请注明出处，保留原文链接和作者信息。

React.js 是一个帮助你构建页面 UI 的库。如果你熟悉 MVC 概念的话，那么 React 的组件就相当于 MVC 里面的 View。如果你不熟悉也没关系，你可以简单地理解为，React.js 将帮助我们将界面分成了各个独立的小块，每一个块就是组件，这些组件之间可以组合、嵌套，就成了我们的页面。

一个组件的显示形态和行为有可能是由某些数据决定的。而数据是可能发生改变的，这时候组件的显示形态就会发生相应的改变。而 React.js 也提供了一种非常高效的方式帮助我们做到了数据和组件显示形态之间的同步。

React.js 不是一个框架，它只是一个库。它只提供 UI (view) 层面的解决方案。在实际的项目当中，它并不能解决我们所有的问题，需要结合其它的库，例如 Redux、React-router 等来协助提供完整的解决方法。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：2. 前端组件化（一）：从一个简单的例子讲起](#)

React.js 小书

[← 返回首页](#)

## 2. 前端组件化（一）：从一个简单的例子讲起

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson2>
- 转载请注明出处，保留原文链接和作者信息。

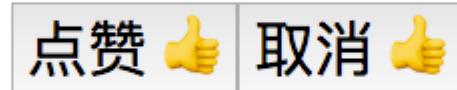
很多课程一上来就给大家如何配置环境、怎么写 React.js 组件。但是本课程还是希望大家对问题的根源有一个更加深入的了解，其实很多的库、框架都是解决类似的问题。只有我们对这些库、框架解决的问题有深入的了解和思考以后，我们才能得心应手地使用它们，并且有新的框架出来也不会太过迷茫——因为其实它们解决都是同一个问题。

这两节课我们来探讨一下是什么样的问题导致了我们需要前端页面进行组件化，前端页面的组件化需要解决什么样的问题。后续课程我们再来看看 React.js 是怎么解决这些问题的。

所以这几节所讲的内容将和 React.js 的内容没有太大的关系，但是如果你能顺利了解这几节的内容，那么后面那些对新手来说很复杂的概念对你来说就是非常自然的事。

### 一个简单的点赞功能

我们会从一个简单的点赞功能讲起。假设现在我们需要实现一个点赞、取消点赞的功能。



如果你对前端稍微有一点了解，你就顺手拈来：

HTML：

```
<body>
  <div class='wrapper'>
    <button class='like-btn'>
      <span class='like-text'>点赞</span>
      <span>👍</span>
    </button>
  </div>
</body>
```

为了模拟现实当中的实际情况，所以这里特意把这个 `button` 里面的 HTML 结构搞得稍微复杂一些。有了这个 HTML 结构，现在就给它加入一些 JavaScript 的行为：

JavaScript：

```
const button = document.querySelector('.like-btn')
const buttonText = button.querySelector('.like-text')
let isLiked = false
button.addEventListener('click', () => {
  isLiked = !isLiked
  if (isLiked) {
    buttonText.innerHTML = '取消'
  } else {
    buttonText.innerHTML = '点赞'
  }
}, false)
```

功能和实现都很简单，按钮已经可以提供点赞和取消点赞的功能。这时候你的同事跑过来了，说他很喜欢你的按钮，他也想用你写的这个点赞功能。这时候问题就来了，你就会发现这种实现方式很致命：你的同事要把整个 `button` 和里面的结构复制过去，还有整段 JavaScript 代码也要复制过去。这样的实现方式没有任何可复用性。

## 结构复用

现在我们来重新编写这个点赞功能，让它具备一定的可复用。这次我们先写一个类，这个类有 `render` 方法，这个方法里面直接返回一个表示 HTML 结构的字符串：

```
class LikeButton {
  render () {
    return `
      <button id='like-btn'>
        <span class='like-text'>赞</span>
        <span>👍</span>
      </button>
    `
  }
}
```

然后可以用这个类来构建不同的点赞功能的实例，然后把它们插到页面中。

```
const wrapper = document.querySelector('.wrapper')
const likeButton1 = new LikeButton()
wrapper.innerHTML = likeButton1.render()

const likeButton2 = new LikeButton()
wrapper.innerHTML += likeButton2.render()
```



这里非常暴力地使用了 `innerHTML`，把两个按钮粗鲁地插入了 `wrapper` 当中。虽然你可能会对这种实现方式非常不满意，但我们还是勉强实现了结构的复用。我们后面再来优化它。

## 实现简单的组件化

你一定会发现，现在的按钮是死的，你点击它它根本不会有什么反应。因为根本没有往上面添加事件。但是问题来了，`LikeButton` 类里面是虽然说有一个 `button`，但是这玩意根本就是在字符串里面的。你怎么能往一个字符串里面添加事件呢？DOM 事件的 API 只有 DOM 结构才能用。

我们需要 DOM 结构，准确地说：**我们需要这个点赞功能的 HTML 字符串表示的 DOM 结构**。假设我们现在有一个函数 `createDOMFromString`，你往这个函数传入 HTML 字符串，但是它会把相应的 DOM 元素返回给你。这个问题就可以解决了。

```
// ::String => ::Document
const createDOMFromString = (domString) => {
  const div = document.createElement('div')
  div.innerHTML = domString
  return div
}
```

先不用管这个函数应该怎么实现，先知道它是干嘛的。拿来用就好，这时候用它来改写一下 `LikeButton` 类：

```

class LikeButton {
  render () {
    this.el = createDOMFromString(` 
      <button class='like-button'>
        <span class='like-text'>点赞</span>
        <span>👍</span>
      </button>
    `)
    this.el.addEventListener('click', () => console.log('click'), false)
    return this.el
  }
}

```

现在 `render()` 返回的不是一个 html 字符串了，而是一个由这个 html 字符串所生成的 DOM。在返回 DOM 元素之前会先给这个 DOM 元素上添加事件再返回。

因为现在 `render` 返回的是 DOM 元素，所以不能用 `innerHTML` 暴力地插入 `wrapper`。而是要用 DOM API 插进去。

```

const wrapper = document.querySelector('.wrapper')

const likeButton1 = new LikeButton()
wrapper.appendChild(likeButton1.render())

const likeButton2 = new LikeButton()
wrapper.appendChild(likeButton2.render())

```

现在你点击这两个按钮，每个按钮都会在控制台打印 `click`，说明事件绑定成功了。但是按钮上的文本还是没有发生改变，只要稍微改动一下 `LikeButton` 的代码就可以完成完整的功能：

```

class LikeButton {
  constructor () {
    this.state = { isLiked: false }
  }

  changeLikeText () {
    const likeText = this.el.querySelector('.like-text')
    this.state.isLiked = !this.state.isLiked
    likeText.innerHTML = this.state.isLiked ? '取消' : '点赞'
  }

  render () {
    this.el = createDOMFromString(` 
      <button class='like-button'>
        <span class='like-text'>点赞</span>
        <span>👍</span>
      </button>
    `)
  }
}

```

```
  `)
  this.el.addEventListener('click', this.changeLikeText.bind(this), false)
  return this.el
}
}
```

这里的代码稍微长了一些，但是还是很好理解。只不过是在给 `LikeButton` 类添加了构造函数，这个构造函数会给每一个 `LikeButton` 的实例添加一个对象 `state`，`state` 里面保存了每个按钮自己是否点赞的状态。还改写了原来的事件绑定函数：原来只打印 `click`，现在点击的按钮的时候会调用 `changeLikeText` 方法，这个方法会根据 `this.state` 的状态改变点赞按钮的文本。

现在这个组件的可复用性已经很不错了，你的同事们只要实例化一下然后插入到 DOM 里面去就好了。

下一节我们继续优化这个例子，让它更加通用。

---

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：3. 前端组件化（二）：优化 DOM 操作](#)

[上一节：1. React.js 简介](#)

React.js 小书

[<-- 返回首页](#)

## 3. 前端组件化（二）：优化 DOM 操作

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson3>
- 转载请注明出处，保留原文链接和作者信息。

看看上一节我们的代码，仔细留意一下 `changeLikeText` 函数，这个函数包含了 DOM 操作，现在看起来比较简单，那是因为现在只有 `isLiked` 一个状态。由于数据状态改变会导致需要我们去更新页面的内容，所以假想一下，如果你的组件依赖了很多状态，那么你的组件基本全部都是 DOM 操作。

一个组件的显示形态由多个状态决定的情况非常常见。代码中混杂着对 DOM 的操作其实是一种不好的实践，手动管理数据和 DOM 之间的关系会导致代码可维护性变差、容易出错。所以我们的例子这里还有优化的空间：如何尽量减少这种手动 DOM 操作？

### 状态改变 -> 构建新的 DOM 元素更新页面

这里要提出的一种解决方案：一旦状态发生改变，就重新调用 `render` 方法，构建一个新的 DOM 元素。这样做好处是什么呢？好处就是你可以在 `render` 方法里面使用最新的 `this.state` 来构造不同 HTML 结构的字符串，并且通过这个字符串构造不同的 DOM 元素。页面就更新了！听起来有点绕，看看代码怎么写，修改原来的代码为：

```
class LikeButton {
  constructor () {
    this.state = { isLiked: false }
  }

  setState (state) {
    this.state = state
    this.el = this.render()
  }

  changeLikeText () {
    this.setState({
      isLiked: !this.state.isLiked
    })
  }

  render () {
    this.el = createDOMFromString(`
```

```

<button class='like-btn'>
  <span class='like-text'>${this.state.isLiked ? '取消' : '点赞'}</span>
  <span>赞</span>
</button>
`)
this.el.addEventListener('click', this.changeLikeText.bind(this), false)
return this.el
}
}

```

其实只是改了几个小地方：

1. `render` 函数里面的 HTML 字符串会根据 `this.state` 不同而不同（这里是用了 ES6 的模版字符串，做这种事情很方便）。
2. 新增一个 `setState` 函数，这个函数接受一个对象作为参数；它会设置实例的 `state`，然后重新调用一下 `render` 方法。
3. 当用户点击按钮的时候，`changeLikeText` 会构建新的 `state` 对象，这个新的 `state`，传入 `setState` 函数当中。

这样的结果就是，用户每次点击，`changeLikeText` 都会调用改变组件状态然后调用 `setState`；`setState` 会调用 `render`，`render` 方法会根据 `state` 的不同重新构建不同的 DOM 元素。

也就是说，**你只要调用 `setState`，组件就会重新渲染**。我们顺利地消除了手动的 DOM 操作。

## 重新插入新的 DOM 元素

上面的改进不会有什么效果，因为你仔细看一下就会发现，其实重新渲染的 DOM 元素并没有插入到页面当中。所以在这个组件外面，你需要知道这个组件发生了改变，并且把新的 DOM 元素更新到页面当中。

重新修改一下 `setState` 方法：

```

...
setState (state) {
  const oldEl = this.el
  this.state = state
  this.el = this.render()
  if (this.onStateChange) this.onStateChange(oldEl, this.el)
}
...

```

使用这个组件的时候：

```
const likeButton = new LikeButton()
wrapper.appendChild(likeButton.render()) // 第一次插入 DOM 元素
likeButton.onStateChange = (oldEl, newEl) => {
  wrapper.insertBefore(newEl, oldEl) // 插入新的元素
  wrapper.removeChild(oldEl) // 删除旧的元素
}
```

这里每次 `setState` 都会调用 `onStateChange` 方法，而这个方法是实例化以后时候被设置的，所以你可以自定义 `onStateChange` 的行为。这里做的事是，每当 `setState` 中构造完新的 DOM 元素以后，就会通过 `onStateChange` 告知外部插入新的 DOM 元素，然后删除旧的元素，页面就更新了。这里已经做到了进一步的优化了：现在不需要再手动更新页面了。

非一般的暴力，因为每次 `setState` 都重新构造、新增、删除 DOM 元素，会导致浏览器进行大量的重排，严重影响性能。不过没有关系，这种暴力行为可以被一种叫 Virtual-DOM 的策略规避掉，但这不是本文所讨论的范围。

这个版本的点赞功能很不错，我可以继续往上面加功能，而且还不需要手动操作DOM。但是有一个不好的地方，如果我要重新另外做一个新组件，譬如说评论组件，那么里面的这些 `setState` 方法要重新写一遍，其实这些东西都可以抽出来，变成一个通用的模式。[下一节](#)我们把这个通用模式抽离到一个类当中。

---

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：4. 前端组件化（三）：抽象出公共组件类](#)

[上一节：2. 前端组件化（一）：从一个简单的例子讲起](#)

React.js 小书

[<-- 返回首页](#)

## 4. 前端组件化（三）：抽象出公共组件类

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson4>
- 转载请注明出处，保留原文链接和作者信息。

为了让代码更灵活，可以写更多的组件，我们把这种模式抽象出来，放到一个 `Component` 类当中：

```
class Component {
  setState (state) {
    const oldEl = this.el
    this.state = state
    this._renderDOM()
    if (this.onStateChange) this.onStateChange(oldEl, this.el)
  }

  _renderDOM () {
    this.el = createDOMFromString(this.render())
    if (this.onClick) {
      this.el.addEventListener('click', this.onClick.bind(this), false)
    }
    return this.el
  }
}
```

这个是一个组件父类 `Component`，所有的组件都可以继承这个父类来构建。它定义的两个方法，一个是我们已经很熟悉的 `setState`；一个是私有方法 `_renderDOM`。  
`_renderDOM` 方法会调用 `this.render` 来构建 DOM 元素并且监听 `onClick` 事件。所以，组件子类继承的时候只需要实现一个返回 HTML 字符串的 `render` 方法就可以了。

还有一个额外的 `mount` 的方法，其实就是把组件的 DOM 元素插入页面，并且在 `setState` 的时候更新页面：

```
const mount = (component, wrapper) => {
  wrapper.appendChild(component._renderDOM())
  component.onStateChange = (oldEl, newEl) => {
    wrapper.insertBefore(newEl, oldEl)
    wrapper.removeChild(oldEl)
```

```

        }
    }
}
```

这样的话我们重新写点赞组件就会变成：

```

class LikeButton extends Component {
    constructor () {
        super()
        this.state = { isLiked: false }
    }

    onClick () {
        this.setState({
            isLiked: !this.state.isLiked
        })
    }

    render () {
        return (
            <button class='like-btn'>
                <span class='like-text'>${this.state.isLiked ? '取消' : '点赞'}</span>
                <span>👍</span>
            </button>
        )
    }

    mount(new LikeButton(), wrapper)
}
```

这样还不够好。在实际开发当中，你可能需要给组件传入一些自定义的配置数据。比如说想配置一下点赞按钮的背景颜色，如果我给它传入一个参数，告诉它怎么设置自己的颜色。那么这个按钮的定制性就更强了。所以我们可以给组件类和它的子类都传入一个参数 `props`，作为组件的配置参数。修改 `Component` 的构造函数为：

```

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

继承的时候通过 `super(props)` 把 `props` 传给父类，这样就可以通过 `this.props` 获取到配置参数：

```

class LikeButton extends Component {
    constructor (props) {
        super(props)
        this.state = { isLiked: false }
```

```

        }

        onClick () {
            this.setState({
                isLiked: !this.state.isLiked
            })
        }

        render () {
            return (
                <button class='like-btn' style="background-color: ${this.props.bgColor}">
                    <span class='like-text'>
                        ${this.state.isLiked ? '取消' : '点赞'}
                    </span>
                    <span>👍</span>
                </button>
            )
        }
    }

    mount(new LikeButton({ bgColor: 'red' }), wrapper)

```

这里我们稍微修改了一下原有的 `LikeButton` 的 `render` 方法，让它可以根据传入的参数 `this.props.bgColor` 来生成不同的 `style` 属性。这样就可以自由配置组件的颜色了。

只要有了上面那个 `Component` 类和 `mount` 方法加起来不足40行代码就可以做到组件化。如果我们需要写另外一个组件，只需要像上面那样，简单地继承一下 `Component` 类就好了：

```

class RedBlueButton extends Component {
    constructor (props) {
        super(props)
        this.state = {
            color: 'red'
        }
    }

    onClick () {
        this.setState({
            color: 'blue'
        })
    }

    render () {
        return (
            <div style='color: ${this.state.color};'>${this.state.color}</div>
        )
    }
}

```

简单好用，现在可以灵活地组件化页面了。`Component` 完整的代码可以在这里找到 [reactjs-in-40](#)。

## 总结

我们用了很长的篇幅来讲一个简单的点赞的例子，并且在这个过程里面一直在优化编写的方式。最后抽离出来了一个类，可以帮助我们更好的做组件化。在这个过程里面我们学到了什么？

组件化可以帮助我们解决前端结构的复用性问题，整个页面可以由这样的不同的组件组合、嵌套构成。

一个组件有自己的显示形态（上面的 HTML 结构和内容）行为，组件的显示形态和行为可以由数据状态（state）和配置参数（props）共同决定。数据状态和配置参数的改变都会影响到这个组件的显示形态。

当数据变化的时候，组件的显示需要更新。所以如果组件化的模式能提供一种高效的方式自动化地帮助我们更新页面，那也就可以大大地降低我们代码的复杂度，带来更好的可维护性。

好了，课程结束了。你已经学会了怎么使用 React.js 了，因为我们已经写了一个——当然我是在开玩笑，但是上面这个 `Component` 类其实和 React 的 `Component` 使用方式很类似。掌握了这几节的课程，你基本就掌握了基础的 React.js 的概念。

接下来我们开始正式进入主题，开始正式介绍 React.js。你会发现，有了前面的铺垫，下面讲的内容理解起来会简单很多了。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：5. React.js 基本环境安装](#)

[上一节：3. 前端组件化（二）：优化 DOM 操作](#)

React.js 小书

[<-- 返回首页](#)

## 5. React.js 基本环境安装

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson5>
- 转载请注明出处，保留原文链接和作者信息。

### 安装 React.js

React.js 单独使用基本上是不可能的事情。不要指望着类似于 `jQuery` 下载放到 `<head />` 标签就开始使用。使用 React.js 不管在开发阶段生产阶段都需要一堆工具和库辅助，编译阶段你需要借助 `Babel`；需要 `Redux` 等第三方的状态管理工具来组织代码；如果你要写单页面应用那么你需要 `React-router`。这就是所谓的“React.js 全家桶”。

本课程不会教大家如何配置这些东西，因为这不是课程的重点，网上有很多的资料，大家可以去参考那些资料。我们这里会直接使用 React.js 官网所推荐使用的工具 `create-react-app` 工具。它可以帮助我们一键生成所需要的工程目录，并帮我们做好各种配置和依赖，也帮我们隐藏了这些配置的细节。也就是所谓的“开箱即用”。

工具地址：<https://github.com/facebookincubator/create-react-app>

### Create React App

Create React apps with no build configuration.

- [Getting Started](#) – How to create a new app.
- [User Guide](#) – How to develop apps bootstrapped with Create React App.

Create React App works on macOS, Windows, and Linux.

If something doesn't work please [file an issue](#).

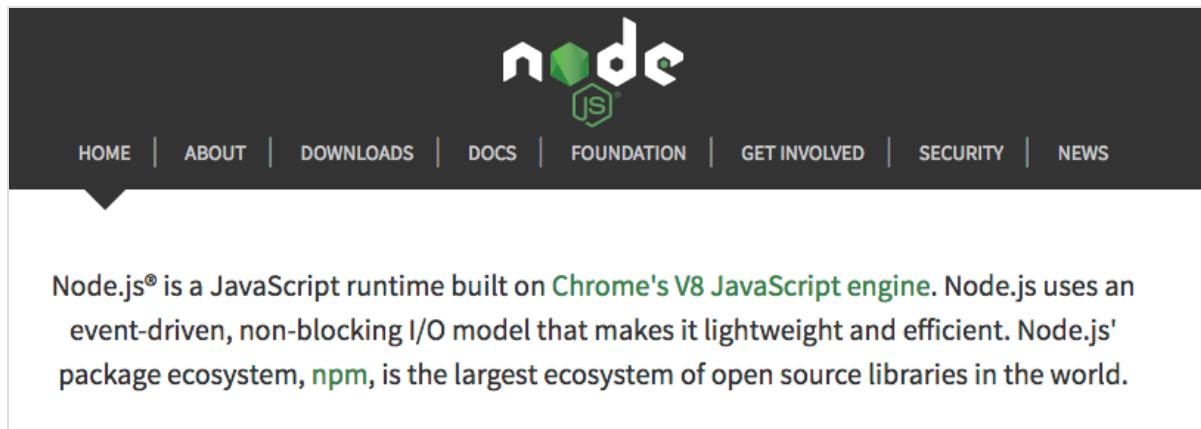
#### tl;dr

```
npm install -g create-react-app
create-react-app my-app
cd my-app/
npm start
```

Then open <http://localhost:3000/> to see your app.

When you're ready to deploy to production, create a minified bundle with `npm run build`.

在安装之前要确认你的机器上安装了 `node.js` 环境包括 `npm`。如果没有安装的同学可以到 `node.js` 的官网下载自己电脑的对应的安装包来安装好环境。



安装好环境以后，只需要按照官网的指引安装 `create-react-app` 即可。

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

这条命令会往我们的机器上安装一条叫 `create-react-app` 的命令，安装好以后就可以直接使用它来构建一个 react 的前端工程：

```
create-react-app hello-react
```

这条命令会帮我们构建一个叫 `hello-react` 的工程，并且会自动地帮助我们安装所需要的依赖，现在只需要安静地等待它安装完。

额外的小贴士：

如果有些同学安装过程比较慢，那是很有可能是因为 npm 下载的时候是从国外的源下载的缘故。所以可以把 npm 的源改成国内的 taobao 的源，这样会加速下载过程。在执行上面的命令之前可以先修改一下 npm 的源：

```
npm config set registry https://registry.npm.taobao.org
```

下载完以后我们就可以启动工程了，进入工程目录然后通过 npm 启动工程：

```
cd hello-react  
npm start
```

终端提示成功：

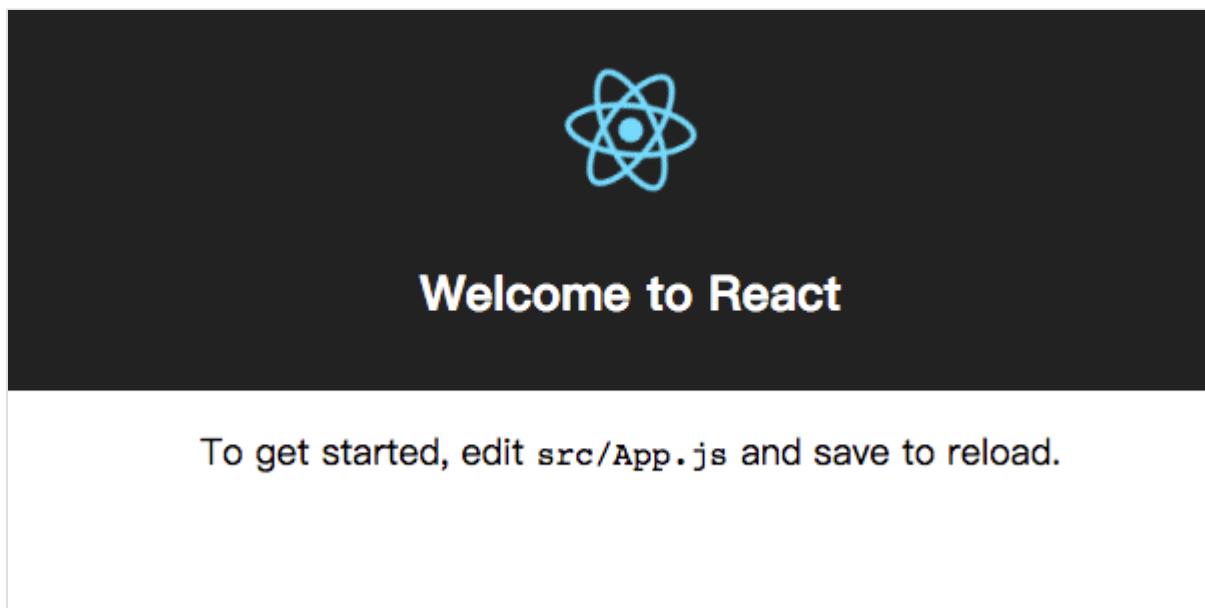
```
Compiled successfully!
```

```
The app is running at:
```

```
http://localhost:3000/
```

```
Note that the development build is not optimized.  
To create a production build, use npm run build.
```

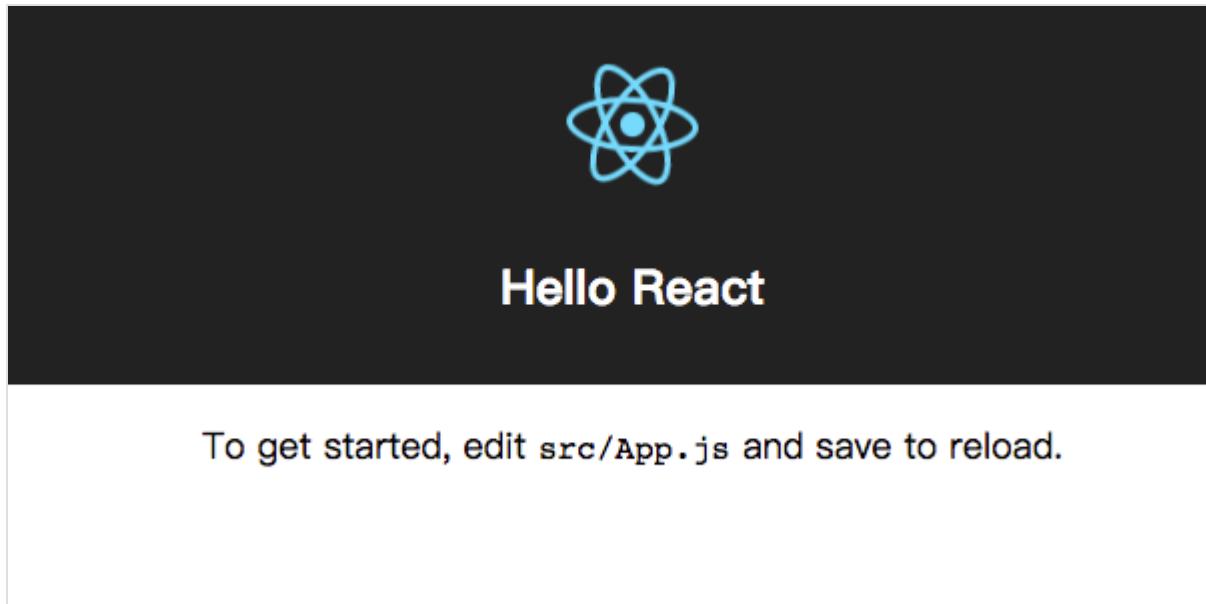
并且会自动打开浏览器，就可以看到 React 的工程顺利运行的效果：



这时候我们把 `src/App.js` 文件中的 `<h2>` 标签的内容修改为 `Hello React`，

```
<h2>Hello React</h2>
```

保存一下，然后户就会发现浏览器自动刷新，并且我们的修改也生效了：



到这里我们的环境已经安装好了，并且顺利地运行了我们第一个例子。接下来我们会探讨 `React.js` 的组件的基本写法。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

---

下一节：6. 使用 JSX 描述 UI 信息

上一节：4. 前端组件化（三）：抽象出公共组件类

React.js 小书

[<-- 返回首页](#)

## 6. 使用 JSX 描述 UI 信息

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson6>
- 转载请注明出处，保留原文链接和作者信息。

这一节我们通过一个简单的例子讲解 React.js 描述页面 UI 的方式。把 `src/index.js` 中的代码改成：

```
import React, { Component } from 'react'
import ReactDOM from 'react-dom'
import './index.css'

class Header extends Component {
  render () {
    return (
      <div>
        <h1>React 小书</h1>
      </div>
    )
  }
}

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

我们在文件头部从 `react` 的包当中引入了 `React` 和 `React.js` 的组件父类 `Component`。记住，只要你要写 `React.js` 组件，那么就必须要引入这两个东西。

`ReactDOM` 可以帮助我们把 `React` 组件渲染到页面上去，没有其它的作用了。你可以发现它是从 `react-dom` 中引入的，而不是从 `react` 引入。有些朋友可能会疑惑，为什么不把这些东西都包含在 `react` 包当中呢？我们稍后会回答这个问题。

接下来的代码你看起来会比较熟悉，但又会有点陌生。你看其实它跟我们前几节里面讲的内容其实很类似，一个组件继承 `Component` 类，有一个 `render` 方法，并且把这个组件的 `HTML` 结构返回；这里 `return` 的东西就比较奇怪了，它并不是一个字符串，看起来像是纯 `HTML` 代码写在 `JavaScript` 代码里面。你也许会说，这不就有语

法错误了么？这完全不是合法的 JavaScript 代码。这种看起来“在 JavaScript 写的标签的”语法叫 JSX。

## JSX 原理

为了让大家深刻理解 JSX 的含义。有必要简单介绍了一下 JSX 稍微底层的运作原理，这样大家可以更加深刻理解 JSX 到底是什么东西，为什么要有这种语法，它是经过怎么样的转化变成页面的元素的。

思考一个问题：如何用 JavaScript 对象来表现一个 DOM 元素的结构，举个例子：

```
<div class='box' id='content'>
  <div class='title'>Hello</div>
  <button>Click</button>
</div>
```

每个 DOM 元素的结构都可以用 JavaScript 的对象来表示。你会发现一个 DOM 元素包含的信息其实只有三个：标签名，属性，子元素。

所以其实上面这个 HTML 所有的信息我们都可以用合法的 JavaScript 对象来表示：

```
{
  tag: 'div',
  attrs: { className: 'box', id: 'content' },
  children: [
    {
      tag: 'div',
      attrs: { className: 'title' },
      children: ['Hello']
    },
    {
      tag: 'button',
      attrs: null,
      children: ['Click']
    }
  ]
}
```

你会发现，HTML 的信息和 JavaScript 所包含的结构和信息其实是一样的，我们可以用 JavaScript 对象来描述所有能用 HTML 表示的 UI 信息。但是用 JavaScript 写起来太长了，结构看起来又不清晰，用 HTML 的方式写起来就方便很多了。

于是 React.js 就把 JavaScript 的语法扩展了一下，让 JavaScript 语言能够支持这种直接在 JavaScript 代码里面编写类似 HTML 标签结构的语法，这样写起来就方便很多了。编译的过程会把类似 HTML 的 JSX 结构转换成 JavaScript 的对象结构。

上面的代码：

```
import React, { Component } from 'react'
import ReactDOM from 'react-dom'
import './index.css'

class Header extends Component {
  render () {
    return (
      <div>
        <h1 className='title'>React 小书</h1>
      </div>
    )
  }
}

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

经过编译以后会变成：

```
import React, { Component } from 'react'
import ReactDOM from 'react-dom'
import './index.css'

class Header extends Component {
  render () {
    return (
      React.createElement(
        "div",
        null,
        React.createElement(
          "h1",
          { className: 'title' },
          "React 小书"
        )
      )
    )
  }
}

ReactDOM.render(
  React.createElement(Header, null),
  document.getElementById('root')
);
```

`React.createElement` 会构建一个 JavaScript 对象来描述你 HTML 结构的信息，包括标签名、属性、还有子元素等。这样的代码就是合法的 JavaScript 代码了。所以使用 React 和 JSX 的时候一定要经过编译的过程。

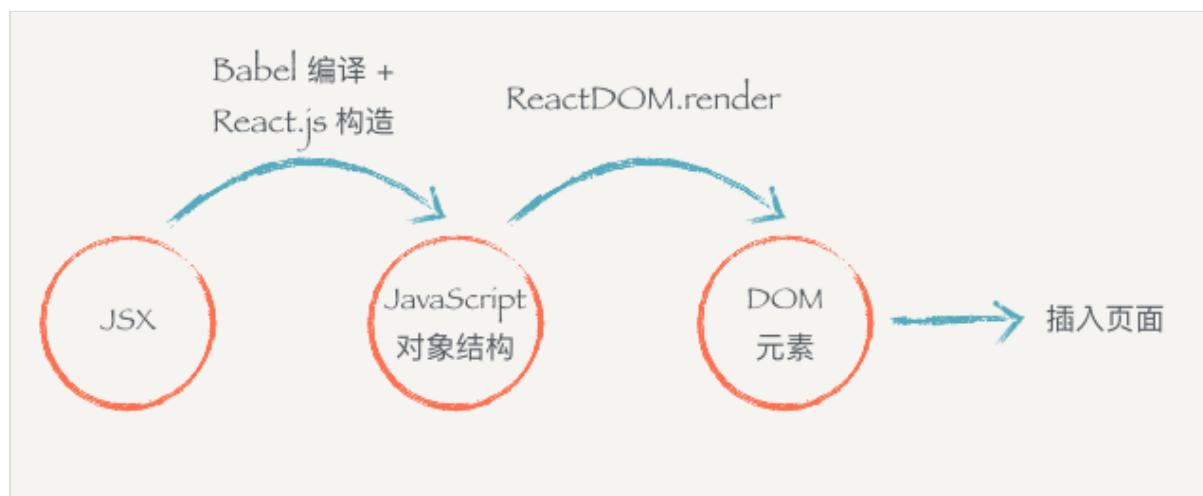
这里再重复一遍：所谓的 JSX 其实就是 JavaScript 对象。每当在 JavaScript 代码中看到这种 JSX 结构的时候，脑子里面就可以自动做转化，这样对你理解 React.js 的组件写法很有好处。

有了这个表示 HTML 结构和信息的对象以后，就可以拿去构造真正的 DOM 元素，然后把这个 DOM 元素塞到页面上。这也是我们最后一段代码中 `ReactDOM.render` 所干的事情：

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

`ReactDOM.render` 功能就是把组件渲染并且构造 DOM 树，然后插入到页面上某个特定的元素上（在这里是 id 为 `root` 的 `div` 元素）。

所以可以总结一下从 JSX 到页面到底经过了什么样的过程：



有些同学可能会问，为什么不直接从 JSX 直接渲染构造 DOM 结构，而是要经过中间这么一层呢？

第一个原因是，当我们拿到一个表示 UI 的结构和信息的对象以后，不一定会把元素渲染到浏览器的普通页面上，我们有可能把这个结构渲染到 canvas 上，或者是手机 App 上。所以这也是为什么要把 `react-dom` 单独抽离出来的原因，可以想象有一个叫 `react-canvas` 可以帮我们把 UI 渲染到 canvas 上，或者是有一个叫 `react-app` 可以帮我们把它转换成原生的 App（实际上这玩意叫 `ReactNative`）。

第二个原因是，有了这样一个对象。当数据变化，需要更新组件的时候，就可以用比较快的算法操作这个 JavaScript 对象，而不用直接操作页面上的 DOM，这样可以尽

量少的减少浏览器重排，极大地优化性能。这个在以后的章节中我们会提到。

## 总结

要记住几个点：

1. JSX 是 JavaScript 语言的一种语法扩展，长得像 HTML，但并不是 HTML。
2. React.js 可以用 JSX 来描述你的组件长什么样的。
3. JSX 在编译的时候会变成相应的 JavaScript 对象描述。
4. `react-dom` 负责把这个用来描述 UI 信息的 JavaScript 对象变成 DOM 元素，并且渲染到页面上。

## 课后练习题

- [用 React.js 在页面上渲染标题](#)

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：7. 组件的 render 方法](#)

[上一节：5. React.js 基本环境安装](#)

React.js 小书

[<-- 返回首页](#)

## 7. 组件的 render 方法

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson7>
- 转载请注明出处，保留原文链接和作者信息。

React.js 中一切皆组件，用 React.js 写的其实就是 React.js 组件。我们在编写 React.js 组件的时候，一般都需要继承 React.js 的 `Component`（还有别的编写组件的方式我们后续会提到）。一个组件类必须要实现一个 `render` 方法，这个 `render` 方法必须要返回一个 JSX 元素。但这里要注意的是，必须要用一个外层的 JSX 元素把所有内容包裹起来。返回并列多个 JSX 元素是不合法的，下面是错误的做法：

```
...
render () {
  return (
    <div>第一个</div>
    <div>第二个</div>
  )
}
...
...
```

必须要用一个外层元素把内容进行包裹：

```
...
render () {
  return (
    <div>
      <div>第一个</div>
      <div>第二个</div>
    </div>
  )
}
...
...
```

### 表达式插入

在 JSX 当中你可以插入 JavaScript 的表达式，表达式返回的结果会相应地渲染到页面上。表达式用 `{}` 包裹。例如：

```

...
render () {
  const word = 'is good'
  return (
    <div>
      <h1>React 小书 {word}</h1>
    </div>
  )
}
...

```

页面上就显示“React 小书 is good”。你也可以把它改成 `{1 + 2}`，它就会显示“React 小书 3”。你也可以把它写成一个函数表达式返回：

```

...
render () {
  return (
    <div>
      <h1>React 小书 {(function () { return 'is good'})()}</h1>
    </div>
  )
}
...

```

简而言之，`{}` 内可以放任何 JavaScript 的代码，包括变量、表达式计算、函数执行等等。`render` 会把这些代码返回的内容如实地渲染到页面上，非常的灵活。

表达式插入不仅仅可以用在标签内部，也可以用在标签的属性上，例如：

```

...
render () {
  const className = 'header'
  return (
    <div className={className}>
      <h1>React 小书</h1>
    </div>
  )
}
...

```

这样就可以为 `div` 标签添加一个叫 `header` 的类名。

注意，直接使用 `class` 在 React.js 的元素上添加类名如 `<div class="xxx">` 这种方式是不合法的。因为 `class` 是 JavaScript 的关键字，所以 React.js 中定义了一种新的方式：`className` 来帮助我们给元素添加类名。

还有一个特例就是 `for` 属性，例如 `<label for='male'>Male</label>`，因为 `for` 也是 JavaScript 的关键字，所以在 JSX 用 `htmlFor` 替代，即 `<label htmlFor='male'>Male</label>`。而其他的 HTML 属性例如 `style`、`data-*` 等就可以像普通的 HTML 属性那样直接添加上去。

## 条件返回

{ } 上面说了，JSX 可以放置任何表达式内容。所以也可以放 JSX，实际上，我们可以在 `render` 函数内部根据不同条件返回不同的 JSX。例如：

```
...
render () {
  const isGoodWord = true
  return (
    <div>
      <h1>
        React 小书
        {isGoodWord
          ? <strong> is good</strong>
          : <span> is not good</span>
        }
      </h1>
    </div>
  )
}
...
...
```

上面的代码中定义了一个 `isGoodWord` 变量为 `true`，下面有个用 {} 包含的表达式，根据 `isGoodWord` 的不同返回不同的 JSX 内容。现在页面上是显示 `React 小书 is good`。如果你把 `isGoodWord` 改成 `false` 然后再看页面上就会显示 `React 小书 is not good`。

如果你在表达式插入里面返回 `null`，那么 React.js 会什么都不显示，相当于忽略了该表达式插入。结合条件返回的话，我们就做到显示或者隐藏某些元素：

```
...
render () {
  const isGoodWord = true
  return (
    <div>
      <h1>
        React 小书
        {isGoodWord
          ? <strong> is good</strong>
          : null
        }
      </h1>
    </div>
  )
}
...
```

```

    </div>
)
}
...

```

这样就相当于在 `isGoodWord` 为 `true` 的时候显示 `<strong>is good</strong>`，否则就隐藏。

条件返回 JSX 的方式在 `React.js` 中很常见，组件的呈现方式随着数据的变化而不一样，你可以利用 JSX 这种灵活的方式随时组合构建不同的页面结构。

如果这里有些同学觉得比较难理解的话，可以回想一下，其实 JSX 就是 JavaScript 里面的对象，转换一下角度，把上面的内容翻译成 JavaScript 对象的形式，上面的代码就很好理解了。

## JSX 元素变量

同样的，如果你能理解 JSX 元素就是 JavaScript 对象。那么你就可以联想到，JSX 元素其实可以像 JavaScript 对象那样自由地赋值给变量，或者作为函数参数传递、或者作为函数的返回值。

```

...
render () {
  const isGoodWord = true
  const goodWord = <strong> is good</strong>
  const badWord = <span> is not good</span>
  return (
    <div>
      <h1>
        React 小书
        {isGoodWord ? goodWord : badWord}
      </h1>
    </div>
  )
}
...

```

这里给把两个 JSX 元素赋值给了 `goodWord` 和 `badWord` 两个变量，然后把它们作为表达式插入的条件返回值。达到效果和上面的例子一样，随机返回不同的页面效果呈现。

再举一个例子：

```

...
renderGoodWord (goodWord, badWord) {
  const isGoodWord = true
  return isGoodWord ? goodWord : badWord

```

```
}

render () {
  return (
    <div>
      <h1>
        React 小书
        {this.renderGoodWord(
          <strong> is good</strong>,
          <span> is not good</span>
        )}
      </h1>
    </div>
  )
}

...
```

这里我们定义了一个 `renderGoodWord` 函数，这个函数接受两个 JSX 元素作为参数，并且随机返回其中一个。在 `render` 方法中，我们把上面例子的两个 JSX 元素传入 `renderGoodWord` 当中，通过表达式插入把该函数返回的 JSX 元素插入到页面上。

## 课后练习

- [用 React.js 构建未读消息组件](#)
- [JSX元素变量](#)

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：8. 组件的组合、嵌套和组件树](#)

[上一节：6. 使用 JSX 描述 UI 信息](#)

React.js 小书

[<-- 返回首页](#)

## 8. 组件的组合、嵌套和组件树

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson8>
- 转载请注明出处，保留原文链接和作者信息。

继续拓展前面的例子，现在我们已经有了 `Header` 组件了。假设我们现在构建一个新的组件叫 `Title`，它专门负责显示标题。你可以在 `Header` 里面使用 `Title` 组件：

```
class Title extends Component {
  render () {
    return (
      <h1>React 小书</h1>
    )
  }
}

class Header extends Component {
  render () {
    return (
      <div>
        <Title />
      </div>
    )
  }
}
```

我们可以直接在 `Header` 标签里面直接使用 `Title` 标签。就像是一个普通的标签一样。React.js 会在 `<Title />` 所在的地方把 `Title` 组件的 `render` 方法表示的 JSX 内容渲染出来，也就是说 `<h1>React 小书</h1>` 会显示在相应的位置上。如果现在我们在 `Header` 里面使用三个 `<Title />`，那么就会有三个 `<h1 />` 显示在页面上。

```
<div>
  <Title />
  <Title />
  <Title />
</div>
```

这样可复用性非常强，我们可以把组件的内容封装好，然后灵活在使用在任何组件内。另外这里要注意的是，**自定义的组件都必须要用大写字母开头，普通的 HTML 标签都用小写字母开头。**

现在让组件多起来。我们来构建额外的组件来构建页面，假设页面是由 `Header` 、 `Main` 、 `Footer` 几个部分组成，由一个 `Index` 把它们组合起来。

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

class Title extends Component {
  render () {
    return (
      <h1>React 小书</h1>
    )
  }
}

class Header extends Component {
  render () {
    return (
      <div>
        <Title />
        <h2>This is Header</h2>
      </div>
    )
  }
}

class Main extends Component {
  render () {
    return (
      <div>
        <h2>This is main content</h2>
      </div>
    )
  }
}

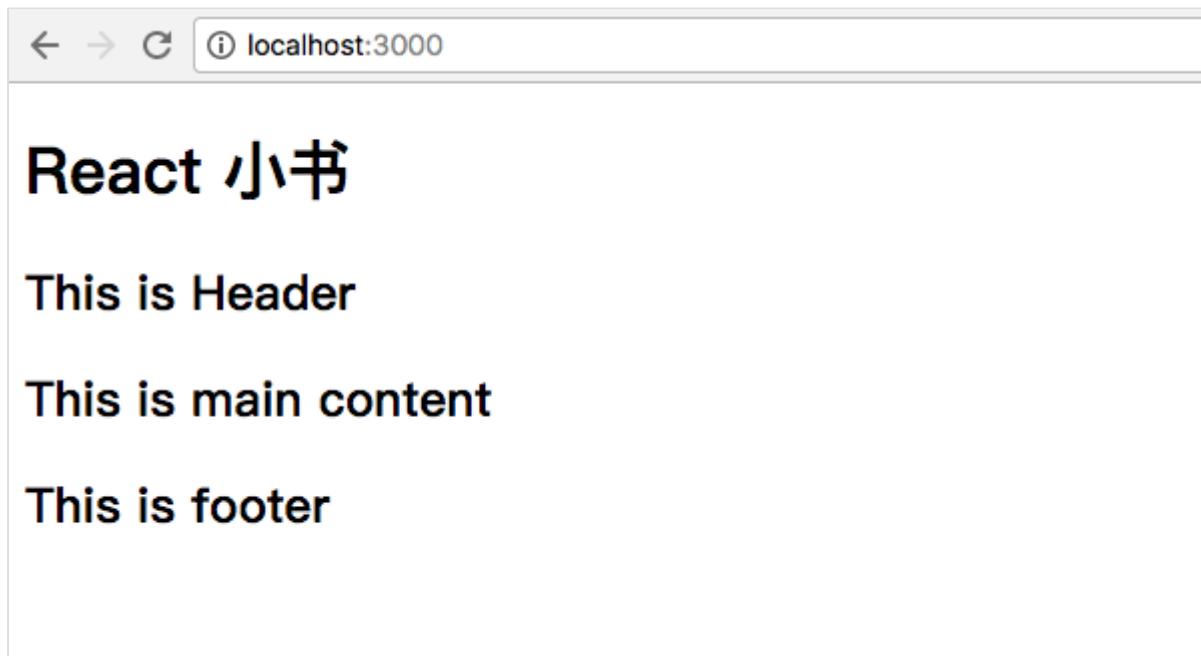
class Footer extends Component {
  render () {
    return (
      <div>
        <h2>This is footer</h2>
      </div>
    )
  }
}

class Index extends Component {
  render () {
```

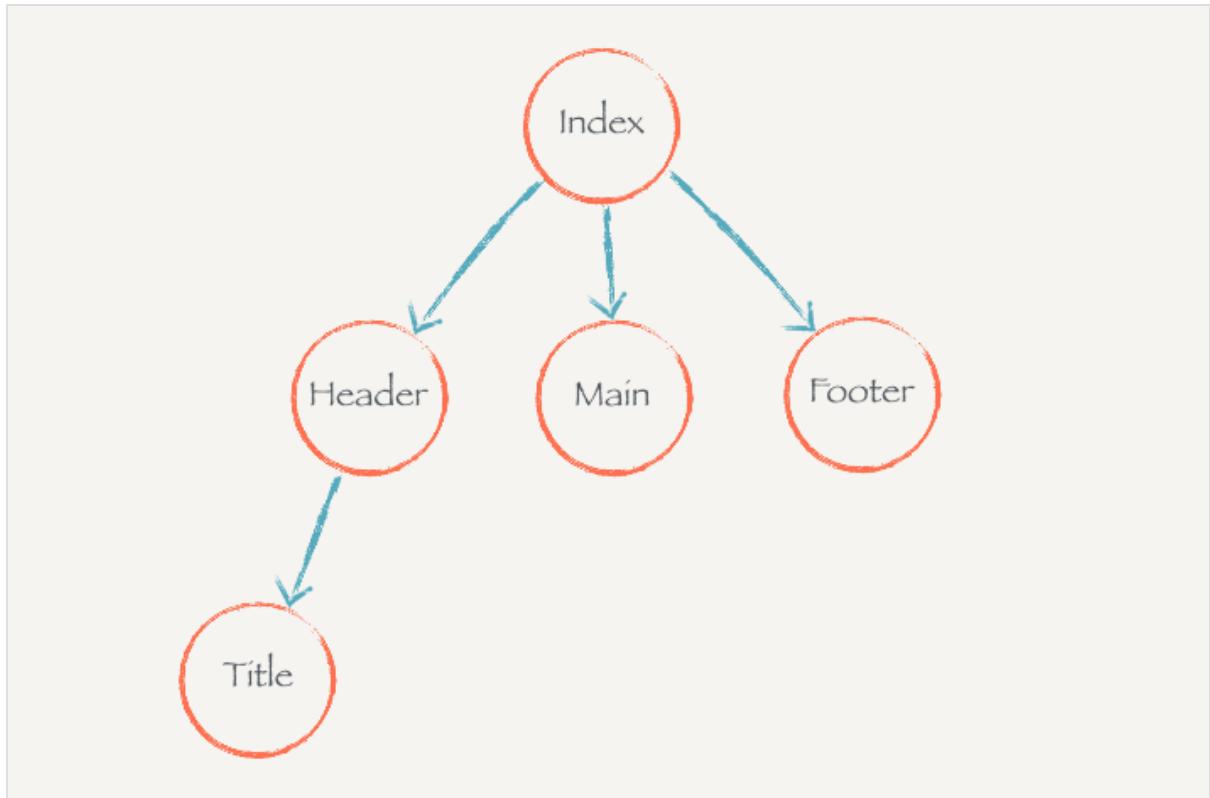
```
return (
  <div>
    <Header />
    <Main />
    <Footer />
  </div>
)
}

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

最后页面会显示内容：



组件可以和组件组合在一起，组件内部可以使用别的组件。就像普通的 HTML 标签一样使用就可以。这样的组合嵌套，最后构成一个所谓的组件树，就正如上面的例子那样，`Index` 用了 `Header`、`Main`、`Footer`，`Header` 又使用了 `Title`。这样用这样的树状结构表示它们之间的关系：



这里的结构还是比较简单，因为我们的页面结构并不复杂。当页面结构复杂起来，有许多不同的组件嵌套组合的话，组件树会相当的复杂和庞大。理解组件树的概念对后面理解数据是如何在组件树内自上往下流动过程很重要。

## 课后练习

- [用 React.js 组建的房子](#)

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

下一节：9. 事件监听

上一节：7. 组件的 render 方法

React.js 小书

[<-- 返回首页](#)

## 9. 事件监听

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson9>
- 转载请注明出处，保留原文链接和作者信息。

在 React.js 里面监听事件是很容易的事情，你只需要给需要监听事件的元素加上属性类似于 `onClick`、`onKeyDown` 这样的属性，例如我们现在要给 `Title` 加上点击的事件监听：

```
class Title extends Component {  
  handleClickOnTitle () {  
    console.log('Click on title.')  
  }  
  
  render () {  
    return (  
      <h1 onClick={this.handleClickOnTitle}>React 小书</h1>  
    )  
  }  
}
```

只需要给 `h1` 标签加上 `onClick` 的事件，`onClick` 紧跟着是一个表达式插入，这个表达式返回一个 `Title` 自己的一个实例方法。当用户点击 `h1` 的时候，React.js 就会调用这个方法，所以你在控制台就可以看到 `Click on title.` 打印出来。

在 React.js 不需要手动调用浏览器原生的 `addEventListener` 进行事件监听。React.js 帮我们封装好了一系列的 `on*` 的属性，当你需要为某个元素监听某个事件的时候，只需要简单地给它加上 `on*` 就可以了。而且你不需要考虑不同浏览器兼容性的问题，React.js 都帮我们封装好这些细节了。

React.js 封装了不同类型的事件，这里就不一一列举，有兴趣的同学可以参考官网文档：[SyntheticEvent - React](#)，多尝试不同的事件。另外要注意的是，这些事件属性名都必须用驼峰命名法。

没有经过特殊处理的话，这些 `on*` 的事件监听只能用在普通的 HTML 的标签上，而不能用在组件标签上。也就是说，`<Header onClick={...} />` 这样的写法不会有什么效果的。这一点要注意，但是有办法可以做到这样的绑定，以后我们会提及。现在只要

记住一点就可以了：这些 `on*` 的事件监听只能用在普通的 HTML 的标签上，而不能用在组件标签上。

## event 对象

和普通浏览器一样，事件监听函数会被自动传入一个 `event` 对象，这个对象和普通的浏览器 `event` 对象所包含的方法和属性都基本一致。不同的是 React.js 中的 `event` 对象并不是浏览器提供的，而是它自己内部所构建的。React.js 将浏览器原生的 `event` 对象封装了一下，对外提供统一的 API 和属性，这样你就不用考虑不同浏览器的兼容性问题。这个 `event` 对象是符合 W3C 标准（[W3C UI Events](#)）的，它具有类似于 `event.stopPropagation`、`event.preventDefault` 这种常用的方法。

我们来尝试一下，这次尝试当用户点击 `h1` 的时候，把 `h1` 的 `innerHTML` 打印出来：

```
class Title extends Component {
  handleClickOnTitle (e) {
    console.log(e.target.innerHTML)
  }

  render () {
    return (
      <h1 onClick={this.handleClickOnTitle}>React 小书</h1>
    )
  }
}
```

再看看控制台，每次点击的时候就会打印”React 小书“。

## 关于事件中的 `this`

一般在某个类的实例方法里面的 `this` 指的是这个实例本身。但是你在上面的 `handleClickOnTitle` 中把 `this` 打印出来，你会看到 `this` 是 `null` 或者 `undefined`。

```
...
handleClickOnTitle (e) {
  console.log(this) // => null or undefined
}
...
```

这是因为 React.js 调用你所传给它的方法的时候，并不是通过对象方法的方式调用 (`this.handleClickOnTitle`)，而是直接通过函数调用 (`handleClickOnTitle`)，所以事件监听函数内并不能通过 `this` 获取到实例。

如果你想在事件函数当中使用当前的实例，你需要手动地将实例方法 `bind` 到当前实例上再传入给 `React.js`。

```
class Title extends Component {
  handleClickOnTitle (e) {
    console.log(this)
  }

  render () {
    return (
      <h1 onClick={this.handleClickOnTitle.bind(this)}>React 小书</h1>
    )
  }
}
```

`bind` 会把实例方法绑定到当前实例上，然后我们再把绑定后的函数传给 `React.js` 的 `onClick` 事件监听。这时候你再看看，点击 `h1` 的时候，就会把当前的实例打印出来：

```
▼ Title
  ► _reactInternalInstance: ReactCompositeComponent
  ► context: Object
  ► isMounted: (...)
  ► props: Object
  ► refs: Object
  ► replaceState: (...)
  ► state: null
  ► updater: Object
  ► __proto__: ReactComponent
```

你也可以在 `bind` 的时候给事件监听函数传入一些参数：

```
class Title extends Component {
  handleClickOnTitle (word, e) {
    console.log(this, word)
  }

  render () {
    return (
      <h1 onClick={this.handleClickOnTitle.bind(this, 'Hello')}>React 小书</h1>
    )
  }
}
```

这种 `bind` 模式在 `React.js` 的事件监听当中非常常见，`bind` 不仅可以帮我们把事件监听方法中的 `this` 绑定到当前组件实例上；还可以帮助我们在渲染列表元素的时候，把列表元素传入事件监听函数当中—这个将在以后的章节提及。

如果有些同学对 JavaScript 的 `this` 模式或者 `bind` 函数的使用方式不是特别了解到话，可能会对这部分内容会有些迷惑，可以补充对 JavaScript 的 this 和 bind 相关的知识再回来回顾这部分内容。

## 总结

为 React 的组件添加事件监听是很简单的事情，你只需要使用 React.js 提供了一系列的 `on*` 方法即可。

React.js 会给每个事件监听传入一个 `event` 对象，这个对象提供的功能和浏览器提供的功能一致，而且它是兼容所有浏览器的。

React.js 的事件监听方法需要手动 `bind` 到当前实例，这种模式在 React.js 中非常常用。

## 课后练习

- 不能摸的狗（一）

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

下一节：10. 组件的 `state` 和 `setState`

上一节：8. 组件的组合、嵌套和组件树

React.js 小书

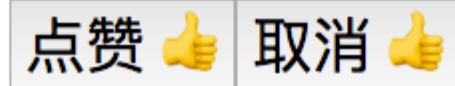
[<-- 返回首页](#)

## 10. 组件的 state 和 setState

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson10>
- 转载请注明出处，保留原文链接和作者信息。

### state

我们前面提到过，一个组件的显示形态是可以由它数据状态和配置参数决定的。一个组件可以拥有自己的状态，就像一个点赞按钮，可以有“已点赞”和“未点赞”状态，并且可以在这两种状态之间进行切换。React.js 的 `state` 就是用来存储这种可变化的状态的。



我们还是拿点赞按钮做例子，它具有已点赞和未点赞两种状态。那么就可以把这个状态存储在 `state` 中。修改 `src/index.js` 为：

```
import React, { Component } from 'react'
import ReactDOM from 'react-dom'
import './index.css'

class LikeButton extends Component {
  constructor () {
    super()
    this.state = { isLiked: false }
  }

  handleClickOnLikeButton () {
    this.setState({
      isLiked: !this.state.isLiked
    })
  }
}

ReactDOM.render(, document.getElementById('root'))
```

```

        })
    }

    render () {
        return (
            <button onClick={this.handleClickOnLikeButton.bind(this)}>
                {this.state.isLiked ? '取消' : '点赞'} 
            </button>
        )
    }
}
...

```

`isLiked` 存放在实例的 `state` 对象当中，这个对象在构造函数里面初始化。这个组件的 `render` 函数内，会根据组件的 `state` 的中的 `isLiked` 不同显示“取消”或“点赞”内容。并且给 `button` 加上了点击的事件监听。

最后构建一个 `Index`，在它的 `render` 函数内使用 `LikeButton`。然后把 `Index` 渲染到页面上：

```

...
class Index extends Component {
    render () {
        return (
            <div>
                <LikeButton />
            </div>
        )
    }
}

ReactDOM.render(
    <Index />,
    document.getElementById('root')
)

```

## setState 接受对象参数

在 `handleClickOnLikeButton` 事件监听函数里面，大家可以留意到，我们调用了 `setState` 函数，每次点击都会更新 `isLiked` 属性为 `!isLiked`，这样就可以做到点赞和取消功能。

`setState` 方法由父类 `Component` 所提供。当我们调用这个函数的时候，`React.js` 会更新组件的状态 `state`，并且重新调用 `render` 方法，然后再把 `render` 方法所渲染的最新的内容显示到页面上。

注意，当我们要改变组件的状态的时候，不能直接用 `this.state = xxx` 这种方式来修改，如果这样做 `React.js` 就没办法知道你修改了组件的状态，它也就没有办法更新页面。所以，一定要使用 `React.js` 提供的 `setState` 方法，**它接受一个对象或者函数作为参数。**

传入一个对象的时候，这个对象表示该组件的新状态。但你只需要传入需要更新的部分就可以了，而不需要传入整个对象。例如，假设现在我们有另外一个状态 `name`：

```
...
constructor (props) {
  super(props)
  this.state = {
    name: 'Tomy',
    isLiked: false
  }
}

handleClickOnLikeButton () {
  this.setState({
    isLiked: !this.state.isLiked
  })
}
...
```

因为点击的时候我们并不需要修改 `name`，所以只需要传入 `isLiked` 就行了。`Tomy` 还是那个 `Tomy`，而 `isLiked` 已经不是那个 `isLiked` 了。

## setState 接受函数参数

这里还有要注意的是，当你调用 `setState` 的时候，`React.js` **并不会马上修改 state**。而是把这个对象放到一个更新队列里面，稍后才会从队列当中把新的状态提取出来合并到 `state` 当中，然后再触发组件更新。这一点要好好注意。可以体会一下下面的代码：

```
...
handleClickOnLikeButton () {
  console.log(this.state.isLiked)
  this.setState({
    isLiked: !this.state.isLiked
  })
  console.log(this.state.isLiked)
}
...
```

你会发现两次打印的都是 `false`，即使我们中间已经 `setState` 过一次了。这并不是什么 bug，只是 `React.js` 的 `setState` 把你的传进来的状态缓存起来，稍后才会帮

你更新到 `state` 上，所以你获取到的还是原来的 `isLiked`。

所以如果你想在 `setState` 之后使用新的 `state` 来做后续运算就做不到了，例如：

```
...
handleClickOnLikeButton () {
  this.setState({ count: 0 }) // => this.state.count 还是 undefined
  this.setState({ count: this.state.count + 1 }) // => undefined + 1 = NaN
  this.setState({ count: this.state.count + 2 }) // => NaN + 2 = NaN
}
...
```

上面的代码的运行结果并不能达到我们的预期，我们希望 `count` 运行结果是 `3`，可是最后得到的是 `Nan`。但是这种后续操作依赖前一个 `setState` 的结果的情况并不罕见。

这里就自然地引出了 `setState` 的第二种使用方式，可以接受一个函数作为参数。

`React.js` 会把上一个 `setState` 的结果传入这个函数，你就可以使用该结果进行运算、操作，然后返回一个对象作为更新 `state` 的对象：

```
...
handleClickOnLikeButton () {
  this.setState((prevState) => {
    return { count: 0 }
  })
  this.setState((prevState) => {
    return { count: prevState.count + 1 } // 上一个 setState 的返回是 count 为 0
  })
  this.setState((prevState) => {
    return { count: prevState.count + 2 } // 上一个 setState 的返回是 count 为 1
  })
  // 最后的结果是 this.state.count 为 3
}
...
```

这样就可以达到上述的利用上一次 `setState` 结果进行运算的效果。

## setState 合并

上面我们进行了三次 `setState`，但是实际上组件只会重新渲染一次，而不是三次；这是因为在 `React.js` 内部会把 `JavaScript` 事件循环中的消息队列的同一个消息中的 `setState` 都进行合并以后再重新渲染组件。

深层的原理并不需要过多纠结，你只需要记住的是：在使用 `React.js` 的时候，并不需要担心多次进行 `setState` 会带来性能问题。

## 课后练习

- 不能摸的狗（二）

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

下一节：11. 配置组件的 props

上一节：9. 事件监听

React.js 小书

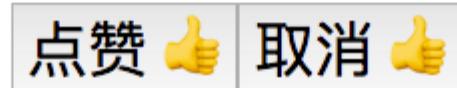
[<-- 返回首页](#)

## 11. 配置组件的 props

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson11>
- 转载请注明出处，保留原文链接和作者信息。

组件是相互独立、可复用的单元，一个组件可能在不同地方被用到。但是在不同的场景下对这个组件的需求可能会根据情况有所不同，例如一个点赞按钮组件，在我这里需要它显示的文本是“点赞”和“取消”，当别的同事拿过去用的时候，却需要它显示“赞”和“已赞”。如何让组件能适应不同场景下的需求，我们就要让组件具有一定“可配置”性。

React.js 的 `props` 就可以帮助我们达到这个效果。每个组件都可以接受一个 `props` 参数，它是一个对象，包含了所有你对这个组件的配置。就拿我们点赞按钮做例子：



下面的代码可以让它达到上述的可配置性：

```
class LikeButton extends Component {
  constructor () {
    super()
    this.state = { isLiked: false }
  }

  handleClickOnLikeButton () {
    this.setState({
      isLiked: !this.state.isLiked
    })
  }
}
```

```

render () {
  const likedText = this.props.likedText || '取消'
  const unlikedText = this.props.unlikedText || '点赞'
  return (
    <button onClick={this.handleClickOnLikeButton.bind(this)}>
      {this.state.isLiked ? likedText : unlikedText} 
    </button>
  )
}
}

```

从 `render` 函数可以看出来，组件内部是通过 `this.props` 的方式获取到组件的参数的，如果 `this.props` 里面有需要的属性我们就采用相应的属性，没有的话就用默认的属性。

那么怎么把 `props` 传进去呢？**在使用一个组件的时候，可以把参数放在标签的属性当中，所有的属性都会作为 `props` 对象的键值：**

```

class Index extends Component {
  render () {
    return (
      <div>
        <LikeButton likedText='已赞' unlikedText='赞' />
      </div>
    )
  }
}

```

就像你在用普通的 `HTML` 标签的属性一样，可以把参数放在表示组件的标签上，组件内部就可以通过 `this.props` 来访问到这些配置参数了。



前面的章节我们说过，`JSX` 的表达式插入可以在标签属性上使用。所以其实可以把任何类型的数据作为组件的参数，包括字符串、数字、对象、数组、甚至是函数等等。例如现在我们把一个对象传给点赞组件作为参数：

```
class Index extends Component {
  render () {
    return (
      <div>
        <LikeButton wordings={{likedText: '已赞', unlikedText: '赞'}} />
      </div>
    )
  }
}
```

现在我们把 `likedText` 和 `unlikedText` 这两个参数封装到一个叫 `wordings` 的对象参数内，然后传入点赞组件中。大家看到 `{}{likedText: '已赞', unlikedText: '赞'}}` 这样的代码的时候，不要以为是什么新语法。之前讨论过，JSX 的 `{}` 内可以嵌入任何表达式，`{}{}` 就是在 `{}` 内部用对象字面量返回一个对象而已。

这时候，点赞按钮的内部就要用 `this.props.wordings` 来获取到参数了：

```
class LikeButton extends Component {
  constructor () {
    super()
    this.state = { isLiked: false }
  }

  handleClickOnLikeButton () {
    this.setState({
      isLiked: !this.state.isLiked
    })
  }

  render () {
    const wordings = this.props.wordings || {
      likedText: '取消',
      unlikedText: '点赞'
    }
    return (
      <button onClick={this.handleClickOnLikeButton.bind(this)}>
        {this.state.isLiked ? wordings.likedText : wordings.unlikedText} 
      </button>
    )
  }
}
```

甚至可以往组件内部传入函数作为参数：

```
class Index extends Component {
  render () {
    return (
      <div>
```

```

<LikeButton
    wordings={{likedText: '已赞', unlikedText: '赞'}}
    onClick={() => console.log('Click on like button!')}/>
</div>
)
}
}

```

这样可以通过 `this.props.onClick` 获取到这个传进去的函数，修改 `LikeButton` 的 `handleClickOnLikeButton` 方法：

```

...
handleClickOnLikeButton () {
  this.setState({
    isLiked: !this.state.isLiked
  })
  if (this.props.onClick) {
    this.props.onClick()
  }
}
...

```

当每次点击按钮的时候，控制台会显示 `Click on like button!`。但这个行为不是点赞组件自己实现的，而是我们传进去的。所以，一个组件的行为、显示形态都可以用 `props` 来控制，就可以达到很好的可配置性。

## 默认配置 `defaultProps`

上面的组件默认配置我们是通过 `||` 操作符来实现。这种需要默认配置的情况在 React.js 中非常常见，所以 React.js 也提供了一种方式 `defaultProps`，可以方便的做到默认配置。

```

class LikeButton extends Component {
  static defaultProps = {
    likedText: '取消',
    unlikedText: '点赞'
  }

  constructor () {
    super()
    this.state = { isLiked: false }
  }

  handleClickOnLikeButton () {
    this.setState({
      isLiked: !this.state.isLiked
    })
  }
}

```

```

        }

      render () {
        return (
          <button onClick={this.handleClickOnLikeButton.bind(this)}>
            {this.state.isLiked
              ? this.props.likedText
              : this.props.unlikedText} 
          </button>
        )
      }
    }
  }
}

```

注意，我们给点赞组件加上了以下的代码：

```

static defaultProps = {
  likedText: '取消',
  unlikedText: '点赞'
}

```

`defaultProps` 作为点赞按钮组件的类属性，里面是对 `props` 中各个属性的默认配置。这样我们就不需要判断配置属性是否传进来了：如果没有传进来，会直接使用 `defaultProps` 中的默认属性。所以可以看到，在 `render` 函数中，我们会直接使用 `this.props` 而不需要再做判断。

## props 不可变

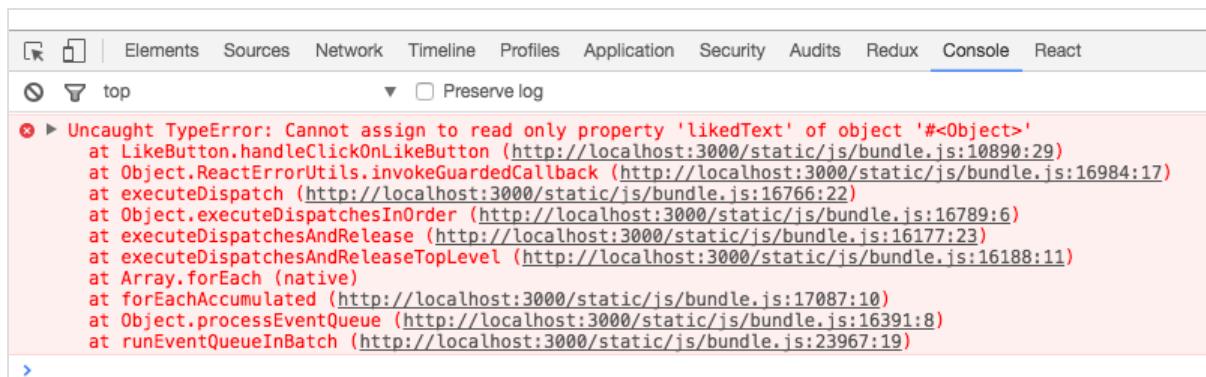
`props` 一旦传入进来就不能改变。修改上面的例子中的 `handleClickOnLikeButton`：

```

...
handleClickOnLikeButton () {
  this.props.likedText = '取消'
  this.setState({
    isLiked: !this.state.isLiked
  })
}
...

```

我们尝试在用户点击按钮的时候改变 `this.props.likedText`，然后你会看到控制台报错了：



你不能改变一个组件被渲染的时候传进来的 `props`。React.js 希望一个组件在输入确定的 `props` 的时候，能够输出确定的 UI 显示形态。如果 `props` 渲染过程中可以被修改，那么就会导致这个组件显示形态和行为变得不可预测，这样可能会给组件使用者带来困惑。

但这并不意味着由 `props` 决定的显示形态不能被修改。组件的使用者可以**主动地通过重新渲染的方式**把新的 `props` 传入组件当中，这样这个组件中由 `props` 决定的显示形态也会得到相应的改变。

修改上面的例子的 `Index` 组件：

```
class Index extends Component {
  constructor () {
    super()
    this.state = {
      likedText: '已赞',
      unlikedText: '赞'
    }
  }

  handleClickOnChange () {
    this.setState({
      likedText: '取消',
      unlikedText: '点赞'
    })
  }

  render () {
    return (
      <div>
        <LikeButton
          likedText={this.state.likedText}
          unlikedText={this.state.unlikedText} />
        <div>
          <button onClick={this.handleClickOnChange.bind(this)}>
            修改 wordings
          </button>
        </div>
      </div>
    )
  }
}
```

```
)  
}  
}
```

在这里，我们把 `Index` 的 `state` 中的 `likedText` 和 `unlikedText` 传给 `LikeButton`。`Index` 还有另外一个按钮，点击这个按钮会通过 `setState` 修改 `Index` 的 `state` 中的两个属性。

由于 `setState` 会导致 `Index` 重新渲染，所以 `LikedButton` 会接收到新的 `props`，并且重新渲染，于是它的显示形态也会得到更新。这就是通过重新渲染的方式来传入新的 `props` 从而达到修改 `LikedButton` 显示形态的效果。

## 总结

1. 为了使得组件的可定制性更强，在使用组件的时候，可以在标签上加属性来传入配置参数。
2. 组件可以在内部通过 `this.props` 获取到配置参数，组件可以根据 `props` 的不同来确定自己的显示形态，达到可配置的效果。
3. 可以通过给组件添加类属性 `defaultProps` 来配置默认参数。
4. `props` 一旦传入，你就不能在组件内部对它进行修改。但是你可以通过父组件主动重新渲染的方式来传入新的 `props`，从而达到更新的效果。

## 课后练习

- [打开和关闭电脑](#)

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

下一节：12. state vs props

上一节：10. 组件的 state 和 setState

React.js 小书

[← 返回首页](#)

## 12. state vs props

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson12>
- 转载请注明出处，保留原文链接和作者信息。

我们来一个关于 `state` 和 `props` 的总结。

`state` 的主要作用是用于组件保存、控制、修改自己的可变状态。`state` 在组件内部初始化，可以被组件自身修改，而外部不能访问也不能修改。你可以认为 `state` 是一个局部的、只能被组件自身控制的数据源。`state` 中状态可以通过 `this.setState` 方法进行更新，`setState` 会导致组件的重新渲染。

`props` 的主要作用是让使用该组件的父组件可以传入参数来配置该组件。它是外部传进来的配置参数，组件内部无法控制也无法修改。除非外部组件主动传入新的 `props`，否则组件的 `props` 永远保持不变。

`state` 和 `props` 有着千丝万缕的关系。它们都可以决定组件的行为和显示形态。一个组件的 `state` 中的数据可以通过 `props` 传给子组件，一个组件可以使用外部传入的 `props` 来初始化自己的 `state`。但是它们的职责其实非常明晰分明：`state` 是让组件控制自己的状态，`props` 是让外部对组件自己进行配置。

如果你觉得还是搞不清 `state` 和 `props` 的使用场景，那么请记住一个简单的规则：尽量少地用 `state`，尽量多地用 `props`。

没有 `state` 的组件叫无状态组件（stateless component），设置了 `state` 的叫做有状态组件（stateful component）。因为状态会带来管理的复杂性，我们尽量多地写无状态组件，尽量少地写有状态的组件。这样会降低代码维护的难度，也会在一定程度上增强组件的可复用性。前端应用状态管理是一个复杂的问题，我们后续会继续讨论。

React.js 非常鼓励无状态组件，在 0.14 版本引入了函数式组件——一种定义不能使用 `state` 组件，例如一个原来这样写的组件：

```
class HelloWorld extends Component {  
  constructor() {  
    super()  
  }  
}
```

```
sayHi () {
  alert('Hello World')
}

render () {
  return (
    <div onClick={this.sayHi.bind(this)}>Hello World</div>
  )
}
}
```

用函数式组件的编写方式就是：

```
const HelloWorld = (props) => {
  const sayHi = (event) => alert('Hello World')
  return (
    <div onClick={sayHi}>Hello World</div>
  )
}
```

以前一个组件是通过继承 `Component` 来构建，一个子类就是一个组件。而用函数式的组件编写方式是一个函数就是一个组件，你可以和以前一样通过 `<HelloWorld />` 使用该组件。不同的是，函数式组件只能接受 `props` 而无法像跟类组件一样可以在 `constructor` 里面初始化 `state`。你可以理解函数式组件就是一种只能接受 `props` 和提供 `render` 方法的类组件。

但本书全书不采用这种函数式的方式来编写组件，统一通过继承 `Component` 来构建组件。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：13. 渲染列表数据](#)

[上一节：11. 配置组件的 props](#)

React.js 小书

[← 返回首页](#)

## 13. 渲染列表数据

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson13>
- 转载请注明出处，保留原文链接和作者信息。

列表数据在前端非常常见，我们经常要处理这种类型的数据，例如文章列表、评论列表、用户列表...一个前端工程师几乎每天都需要跟列表数据打交道。

React.js 当然也允许我们处理列表数据，但在使用 React.js 处理列表数据的时候，需要掌握一些规则。我们这一节会专门讨论这方面的知识。

### 渲染存放 JSX 元素的数组

假设现在我们有这么一个用户列表数据，存放在一个数组当中：

```
const users = [
  { username: 'Jerry', age: 21, gender: 'male' },
  { username: 'Tomy', age: 22, gender: 'male' },
  { username: 'Lily', age: 19, gender: 'female' },
  { username: 'Lucy', age: 20, gender: 'female' }
]
```

如果现在要把这个数组里面的数据渲染页面上要怎么做？开始之前要补充一个知识。之前说过 JSX 的表达式插入 `{}` 里面可以放任何数据，如果我们往 `{}` 里面放一个存放 JSX 元素的数组会怎么样？

```
...
class Index extends Component {
  render () {
    return (
      <div>
        {[  

          <span>React.js </span>,
          <span>is </span>,
          <span>good</span>
        ]}
      </div>
    )
  }
}
```

```

    }

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

我们往 JSX 里面塞了一个数组，这个数组里面放了一些 JSX 元素（其实就是 JavaScript 对象）。到浏览器中，你在页面上会看到：



React.js is good

审查一下元素，看看会发现什么：

```

▼<div data-reactroot class="wrapper">
  ▼<div>
    <span>React.js </span>
    <span>is </span>
    <span>good</span>
  </div>
</div>
  
```

React.js 把插入表达式数组里面的每一个 JSX 元素一个个罗列下来，渲染到页面上。所以这里有个关键点：**如果你往 {} 放一个数组，React.js 会帮你把数组里面一个个元素罗列并且渲染出来。**

## 使用 map 渲染列表数据

知道这一点以后你就可以知道怎么用循环把元素渲染到页面上：循环上面用户数组里面的每一个用户，为每个用户数据构建一个 JSX，然后把 JSX 放到一个新的数组里面，再把新的数组插入 render 方法的 JSX 里面。看看代码怎么写：

```

const users = [
  { username: 'Jerry', age: 21, gender: 'male' },
  { username: 'Tomy', age: 22, gender: 'male' },
  { username: 'Lily', age: 19, gender: 'female' },
  { username: 'Lucy', age: 20, gender: 'female' }
  
```

```
]

class Index extends Component {
  render () {
    const usersElements = [] // 保存每个用户渲染以后 JSX 的数组
    for (let user of users) {
      usersElements.push( // 循环每个用户，构建 JSX，push 到数组中
        <div>
          <div>姓名: {user.username}</div>
          <div>年龄: {user.age}</div>
          <div>性别: {user.gender}</div>
          <hr />
        </div>
      )
    }
  }

  return (
    <div>{usersElements}</div>
  )
}
}

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

这里用了一个新的数组 `usersElements`，然后循环 `users` 数组，为每个 `user` 构建一个 `JSX` 结构，然后 `push` 到 `usersElements` 中。然后直接用表达式插入，把这个 `userElements` 插到 `return` 的 `JSX` 当中。因为 `React.js` 会自动化帮我们把数组当中的 `JSX` 罗列渲染出来，所以可以看到页面上显示：

姓名: Jerry

年龄: 21

性别: male

---

姓名: Tomy

年龄: 22

性别: male

---

姓名: Lily

年龄: 19

性别: female

---

姓名: Lucy

年龄: 20

性别: female

---

但我们一般不会手动写循环来构建列表的 JSX 结构，可以直接用 ES6 自带的 `map`（不了解 `map` 函数的同学可以先了解相关的知识再来回顾这里），代码可以简化成：

```
class Index extends Component {
  render () {
    return (
      <div>
        {users.map((user) => {
          return (
            <div>
              <div>姓名: {user.username}</div>
              <div>年龄: {user.age}</div>
              <div>性别: {user.gender}</div>
              <hr />
            </div>
          )
        })}
      </div>
    )
  }
}
```

这样的模式在 JavaScript 中非常常见，一般来说，在 React.js 处理列表就是用 `map` 来处理、渲染的。现在进一步把渲染单独一个用户的结构抽离出来作为一个组件，继续优化代码：

```
const users = [
  { username: 'Jerry', age: 21, gender: 'male' },
```

```

    { username: 'Tomy', age: 22, gender: 'male' },
    { username: 'Lily', age: 19, gender: 'female' },
    { username: 'Lucy', age: 20, gender: 'female' }
]

class User extends Component {
  render () {
    const { user } = this.props
    return (
      <div>
        <div>姓名: {user.username}</div>
        <div>年龄: {user.age}</div>
        <div>性别: {user.gender}</div>
        <hr />
      </div>
    )
  }
}

class Index extends Component {
  render () {
    return (
      <div>
        {users.map((user) => <User user={user} />)}
      </div>
    )
  }
}

ReactDOM.render(
  <Index />,
  document.getElementById('root')
)

```

这里把负责展示用户数据的 JSX 结构抽离成一个组件 `User`，并且通过 `props` 把 `user` 数据作为组件的配置参数传进去；这样改写 `Index` 就非常清晰了，看一眼就知道负责渲染 `users` 列表，而用的组件是 `User`。

## key! key! key!

现在代码运作正常，好像没什么问题。打开控制台看看：

```

✖ ▶ Warning: Each child in an array or iterator should have a unique "key" prop. Check the render method of `Index`. See http://fb.me/react-warning-keys for more information.
  in User (at index.js:183)
  in Index (at index.js:209)

```

React.js 报错了。如果需要详细解释这里报错的原因，估计要单独写半本书。但可以简单解释一下。

React.js 的是非常高效的，它高效依赖于所谓的 Virtual-DOM 策略。简单来说，能复用的话 React.js 就会尽量复用，没有必要的话绝对不碰 DOM。对于列表元素来说

也是这样，但是处理列表元素的复用性会有一个问题：元素可能会在一个列表中改变位置。例如：

```
<div>a</div>
<div>b</div>
<div>c</div>
```

假设页面上有这么3个列表元素，现在改变一下位置：

```
<div>a</div>
<div>c</div>
<div>b</div>
```

c 和 b 的位置互换了。但其实 React.js 只需要交换一下 DOM 位置就行了，但是它并不知道其实我们只是改变了元素的位置，所以它会重新渲染后面两个元素（再执行 Virtual-DOM 策略），这样会大大增加 DOM 操作。但如果给每个元素加上唯一的标识，React.js 就可以知道这两个元素只是交换了位置：

```
<div key='a'>a</div>
<div key='b'>b</div>
<div key='c'>c</div>
```

这样 React.js 就简单的通过 key 来判断出来，这两个列表元素只是交换了位置，可以尽量复用元素内部的结构。

这里没听懂没有关系，后面有机会会继续讲解这部分内容。现在只需要记住一个简单的规则：**对于用表达式套数组罗列到页面上的元素，都要为每个元素加上 key 属性，这个 key 必须是每个元素唯一的标识。**一般来说，key 的值可以直接后台数据返回的 id，因为后台的 id 都是唯一的。

在上面的例子当中，每个 user 没有 id 可以用，可以直接用循环计数器 i 作为 key：

```
...
class Index extends Component {
  render () {
    return (
      <div>
        {users.map((user, i) => <User key={i} user={user} />)}
      </div>
    )
  }
}
```

再看看，控制台已经没有错误信息了。但这是不好的做法，这只是掩耳盗铃（具体原因大家可以自己思考一下）。记住一点：在实际项目当中，如果你的数据顺序可能发生变化，标准做法是最好是后台数据返回的 `id` 作为列表元素的 `key`。

## 课后练习

- [打印章节标题](#)

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：14. 实战分析：评论功能（一）](#)

[上一节：12. state vs props](#)

React.js 小书

[<-- 返回首页](#)

## 14. 实战分析：评论功能（一）

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson14>
- 转载请注明出处，保留原文链接和作者信息。

课程到这里大家已经掌握了 React.js 的基础知识和组件的基本写法了。现在可以把我所学到的内容应用于实战当中。这里给大家提供一个实战的案例：一个评论功能。效果如下：

用户名：

评论内容：

*Jerry*：你说得没错，这里确实有一个 Bug

*Tomy*：@Jerry 那 fix 以后再给我发一个 PR 吧，我看看再合进去。记得 git rebase 把这几个 commit 都 squash 成一个 commit。另外，以后写功能的时候最好根据功能不同切一个新的分支出来，这样比较好管理。

*Jerry*：收到！

*Lucy*：@Tomy 昨天那个 patch 麻烦也帮忙看看，谢谢

*Tomy*：@Lucy 好的，马上看看！

## 在线演示地址

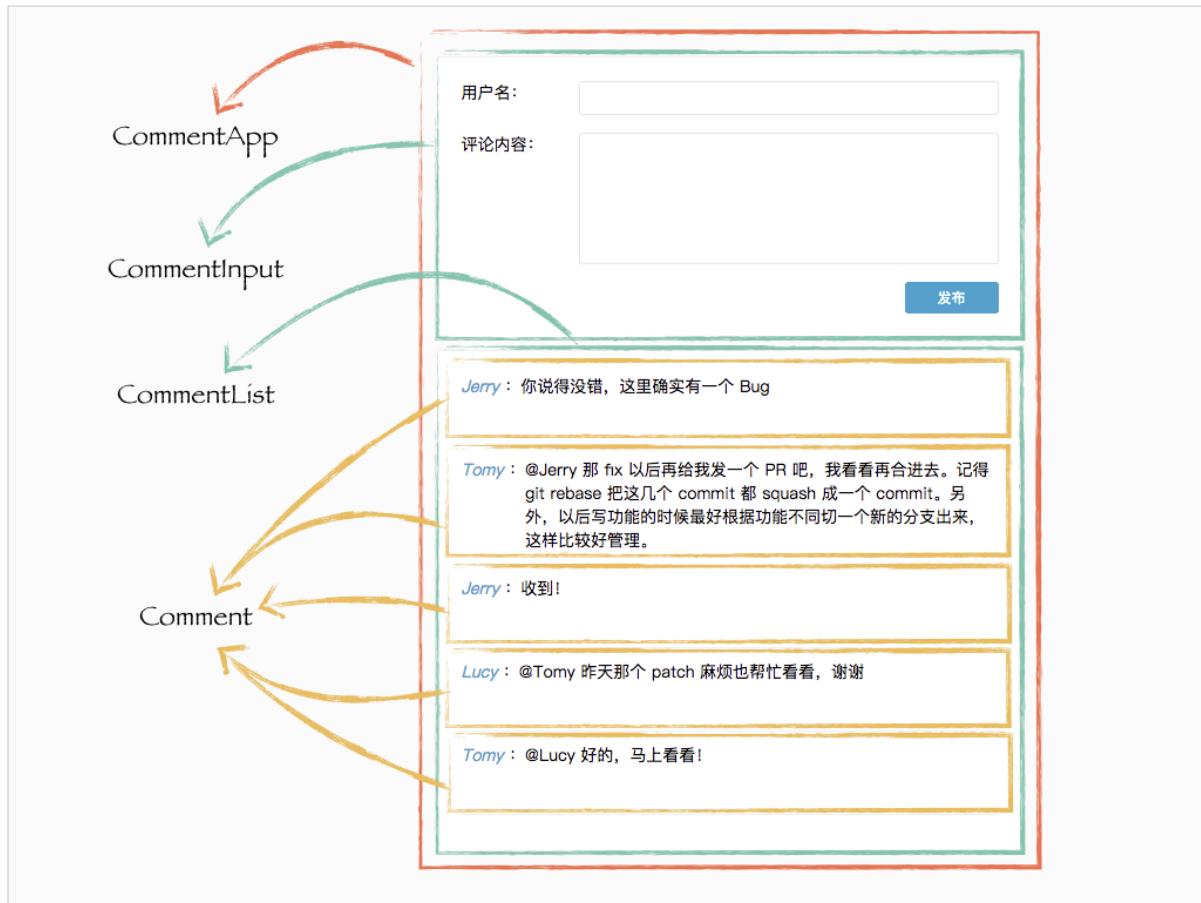
接下来会带大家一起来学习如何分析、编写这个功能。在这个过程中会补充一些之前没有提及的知识点，虽然这些知识点之前没有单独拿出来讲解，但是这些知识点也很关键。

## 组件划分

React.js 中一切都是组件，用 React.js 构建的功能其实也就是由各种组件组合而成。所以拿到一个需求以后，我们要做的第一件事情就是理解需求、分析需求、划分这个需求由哪些组件构成。

组件的划分没有特别明确的标准。划分组件的目的性是为了代码可复用性、可维护性。只要某个部分有可能复用到别的地方，你都可以把它抽离出来当成一个组件；或者把某一部分抽离出来对代码的组织和管理会带来帮助，你也可以毫不犹豫地把它抽离出来。

对于上面这个评论功能，可以粗略地划分成以下几部分：



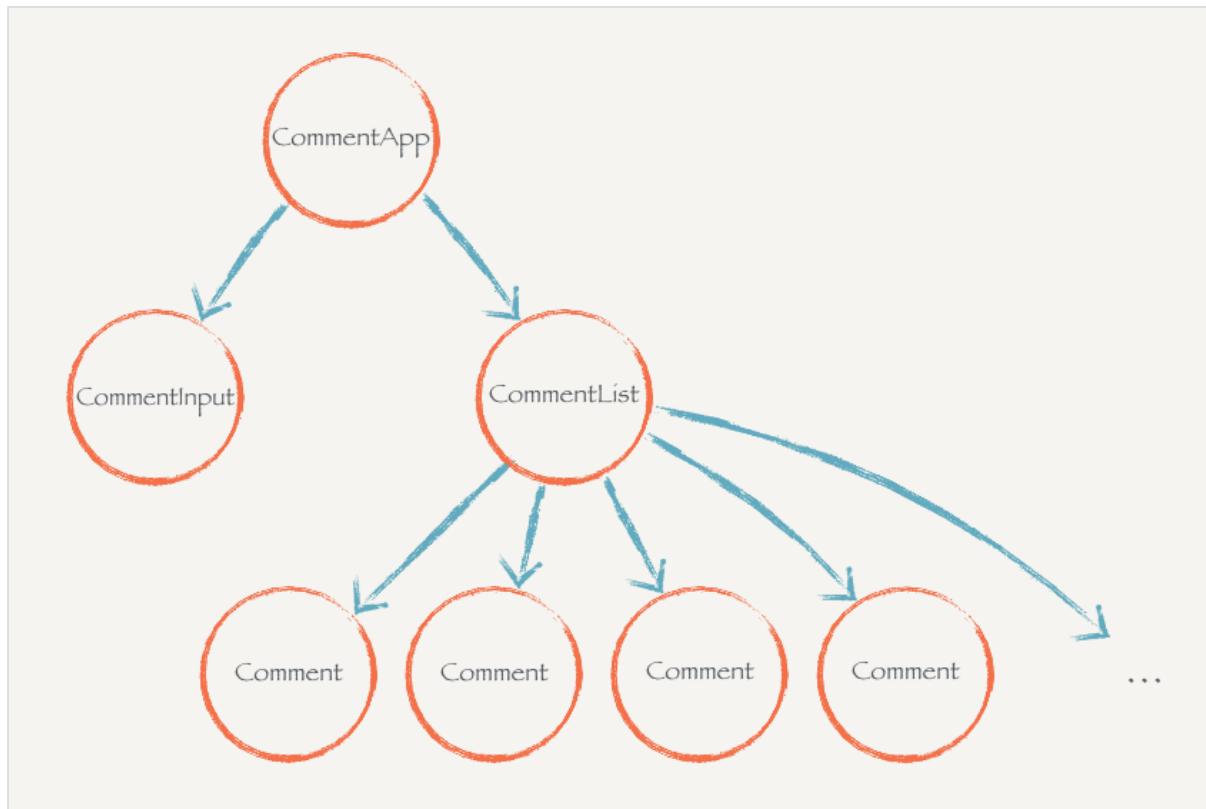
`CommentApp`：评论功能的整体用一个叫 `CommentApp` 的组件包含起来。`CommentApp` 包含上部和下部两部分。

`CommentInput`：上面部分是负责用户输入可操作的输入区域，包括输入评论的用户名、评论内容和发布按钮，这一部分功能划分到一个单独的组件 `CommentInput` 中。

`CommentList`：下面部分是评论列表，用一个叫 `CommentList` 的组件负责列表的展示。

`Comment`：每个评论列表项由独立的组件 `Comment` 负责显示，这个组件被 `CommentList` 所使用。

所以这个评论功能划分成四种组件，`CommentApp`、`CommentInput`、`CommentList`、`Comment`。用组件树表示：



现在就可以尝试编写代码了。

## 组件实现

在写代码之前，我们先用 `create-react-app` 构建一个新的工程目录。所有的评论功能在这个工程内完成：

```
create-react-app comment-app
```

然后在工程目录下的 `src/` 目录下新建四个文件，每个文件对应的是上述的四个组件。

```
src/
  CommentApp.js
  CommentInput.js
  CommentList.js
  Comment.js
  ...
```

你可以注意到，这里的文件名的开头是大写字母。我们遵循一个原则：如果一个文件导出的是一个类，那么这个文件名就用大写开头。四个组件类文件导出都是类，所以都是大写字母开头。

我们先铺垫一些基础代码，让组件之间的关系清晰起来。遵循“自顶而下，逐步求精”的原则，我们从组件的顶层开始，再一步步往下构建组件树。先修改

CommentApp.js 如下：

```
import React, { Component } from 'react'
import CommentInput from './CommentInput'
import CommentList from './CommentList'

class CommentApp extends Component {
  render() {
    return (
      <div>
        <CommentInput />
        <CommentList />
      </div>
    )
  }
}

export default CommentApp
```

CommentApp 现在暂时还很简单，文件顶部引入了 CommentInput 和 CommentList。然后按照上面的需求，应用在了 CommentApp 返回的 JSX 结构中，上面是用户输入区域，下面是评论列表。

现在来修改 CommentInput.js 中的内容：

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

class CommentInput extends Component {
  render() {
    return (
      <div>CommentInput</div>
    )
  }
}

export default CommentInput
```

这里暂时让它只简单返回 <div> 结构，同样地修改 CommentList.js：

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

class CommentList extends Component {
  render() {
    return (
      <div>CommentList</div>
    )
  }
}
```

```
export default CommentList
```

现在可以把这个简单的结构渲染到页面上看看什么效果，修改 `src/index.js`：

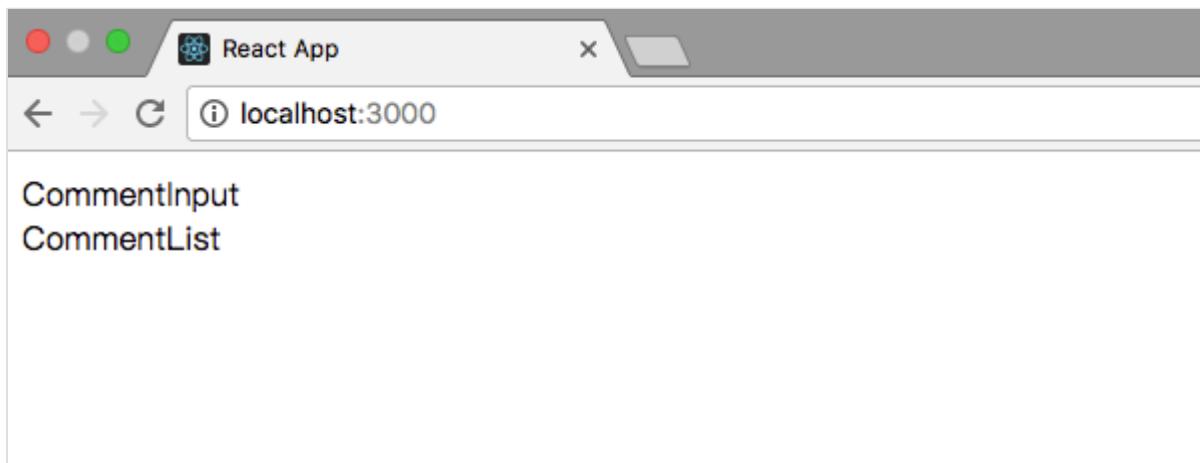
```
import React from 'react'
import ReactDOM from 'react-dom'
import CommentApp from './CommentApp'
import './index.css'

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

然后进入工程目录启动工程：

```
npm run start
```

在浏览器中可以看到，基本的结构已经渲染到了页面上了：



## 添加样式

现在想让这个结构在浏览器中居中显示，我们就要给 `CommentApp` 里面的 `<div>` 添加样式。修改 `CommentApp` 中的 `render` 方法，给它添加一个 `wrapper` 类名：

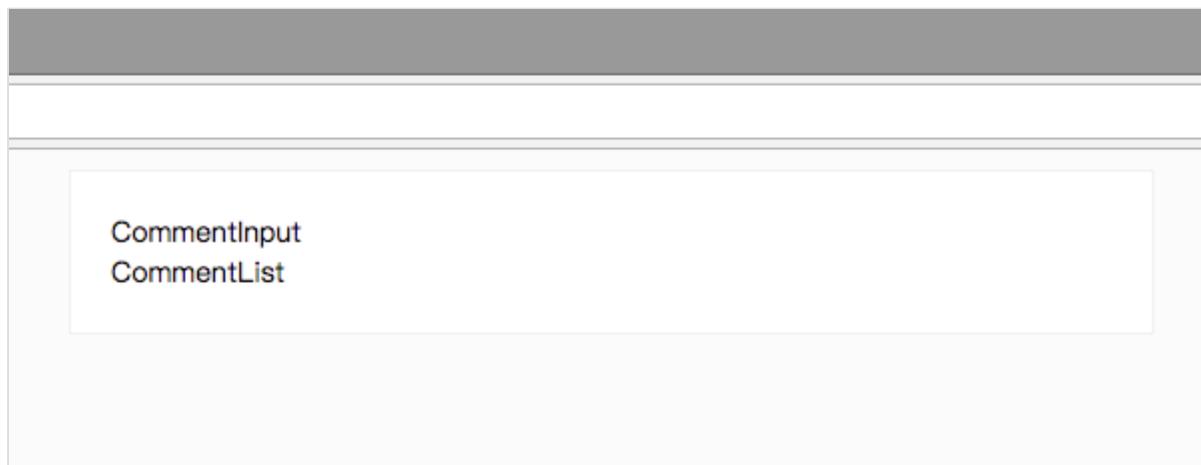
```
...
class CommentApp extends Component {
  render() {
    return (
      <div className='wrapper'>
        <CommentInput />
        <CommentList />
      </div>
    )
  }
}
```

```
)  
}  
}  
...  
}
```

然后在 `index.css` 文件中添加样式：

```
.wrapper {  
  width: 500px;  
  margin: 10px auto;  
  font-size: 14px;  
  background-color: #fff;  
  border: 1px solid #f1f1f1;  
  padding: 20px;  
}
```

在浏览器中可以看到样式生效了：



评论功能案例的所有样式都是通过这种方式进行添加。由于我们专注点在于 React.js，本案例后续不会在样式上过于纠缠。这里写好了一个样式文件（[index.css](#)）提供给大家，可以复制到 `index.css` 当中。后续只需要在元素上加上类名就可以了。

如何在 React.js 中使用样式有很多种方式，也是一个比较大的话题，有很多种不同的方式也有很多不同的争论，这个话题后续有机会会重点讲解。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：15. 实战分析：评论功能（二）](#)

[上一节：13. 渲染列表数据](#)



React.js 小书

[<-- 返回首页](#)

## 15. 实战分析：评论功能（二）

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson15>
- 转载请注明出处，保留原文链接和作者信息。

[上一节](#)我们构建了基本的代码框架，现在开始完善其他的内容。

### 处理用户输入

我们从 `ComponentInput` 组件开始，学习 React.js 是如何处理用户输入的。首先修改 `ComponentInput.js`，完善 `ComponentInput` 的 `render` 函数中的 HTML 结构：

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

class CommentInput extends Component {
  render () {
    return (
      <div className='comment-input'>
        <div className='comment-field'>
          <span className='comment-field-name'>用户名：</span>
          <div className='comment-field-input'>
            <input />
          </div>
        </div>
        <div className='comment-field'>
          <span className='comment-field-name'>评论内容：</span>
          <div className='comment-field-input'>
            <textarea />
          </div>
        </div>
        <div className='comment-field-button'>
          <button>
            发布
          </button>
        </div>
      </div>
    )
  }
}

export default CommentInput
```

在浏览器中可以看到 `ComponentInput` 的结构和样式都已经生效：

用户名:

评论内容:

**发布**

[CommentList](#)

因为还没有加入处理逻辑，所以你输入内容，然后点击发布是不会有什么效果的。用户可输入内容一个是用户名 (`username`)，一个是评论内容 (`content`)，我们在组件的构造函数中初始化一个 `state` 来保存这两个状态：

```
...
class CommentInput extends Component {
  constructor () {
    super()
    this.state = {
      username: '',
      content: ''
    }
  }
  ...
}
```

然后给输入框设置 `value` 属性，让它们的 `value` 值等于 `this.state` 里面相应的值：

```
...
<div className='comment-field'>
  <span className='comment-field-name'>用户名:</span>
  <div className='comment-field-input'>
    <input value={this.state.username} />
  </div>
</div>
<div className='comment-field'>
  <span className='comment-field-name'>评论内容:</span>
```

```

<div className='comment-field-input'>
  <textarea value={this.state.content} />
</div>
</div>
...

```

可以看到接受用户名输入的 `<input />` 和接受用户评论内容的 `<textarea />` 的 `value` 值分别由 `state.username` 和 `state.content` 控制。这时候你到浏览器里面去输入内容看看，你会发现你什么都输入不了。

这是为什么呢？React.js 认为所有的状态都应该由 React.js 的 state 控制，只要类似于 `<input />`、`<textarea />`、`<select />` 这样的输入控件被设置了 `value` 值，那么它们的值永远以被设置的值为准。值不变，`value` 就不会变化。

例如，上面设置了 `<input />` 的 `value` 为 `this.state.username`，`username` 在 `constructor` 中被初始化为空字符串。即使用户在输入框里面尝试输入内容了，还是没有改变 `this.state.username` 是空字符串的事实。

所以应该怎么做才能把用户内容输入更新到输入框当中呢？在 React.js 当中必须要用 `setState` 才能更新组件的内容，所以我们需要做的就是：监听输入框的 `onChange` 事件，然后获取到用户输入的内容，再通过 `setState` 的方式更新 `state` 中的 `username`，这样 `input` 的内容才会更新。

```

...
<div className='comment-field-input'>
  <input
    value={this.state.username}
    onChange={this.handleUsernameChange.bind(this)} />
</div>
...

```

上面的代码给 `input` 加上了 `onChange` 事件监听，绑定到 `this.handleUsernameChange` 方法中，该方法实现如下：

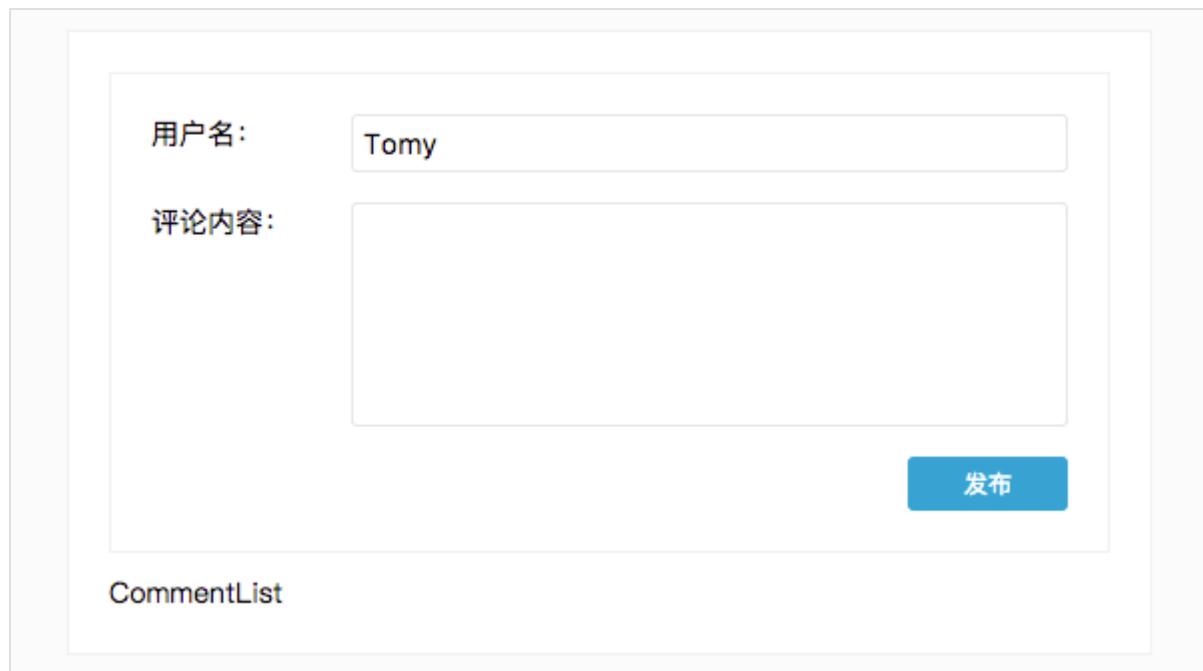
```

...
handleUsernameChange (event) {
  this.setState({
    username: event.target.value
  })
}
...

```

在这个方法中，我们通过 `event.target.value` 获取 `<input />` 中用户输入的内容，然后通过 `setState` 把它设置到 `state.username` 当中，这时候组件的内容就会更

新，`input` 的 `value` 值就会得到更新并显示到输入框内。这时候输入已经没有问题了：



类似于 `<input />`、`<select />`、`<textarea>` 这些元素的 `value` 值被 React.js 所控制、渲染的组件，在 React.js 当中被称为受控组件（Controlled Component）。对于用户可输入的控件，一般都可以让它们成为受控组件，这是 React.js 所推崇的做法。另外还有非受控组件，这里暂时不提及。

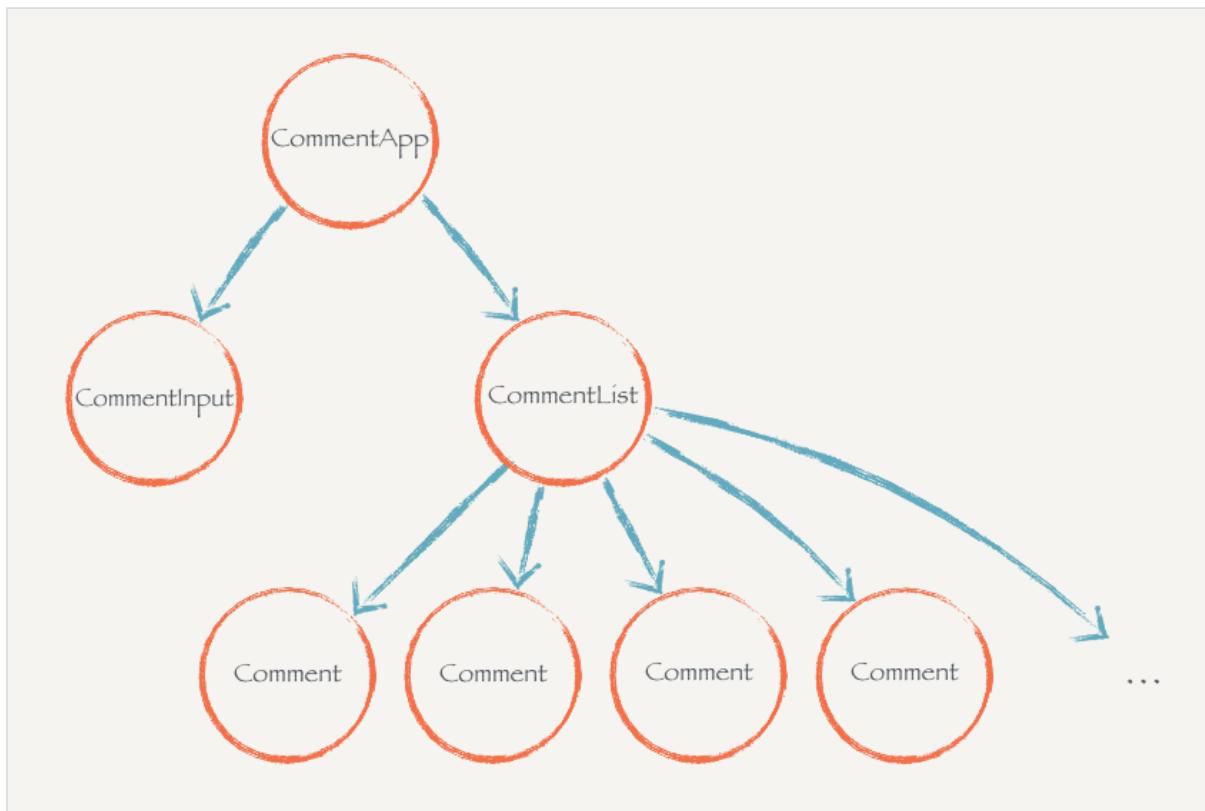
同样地，让 `<textarea />` 成为受控组件：

```
...
handleContentChange (event) {
  this.setState({
    content: event.target.value
  })
}

...
<div className='comment-field'>
  <span className='comment-field-name'>评论内容：</span>
  <div className='comment-field-input'>
    <textarea
      value={this.state.content}
      onChange={this.handleContentChange.bind(this)} />
  </div>
</div>
...
```

## 向父组件传递数据

当用户在 `CommentInput` 里面输入完内容以后，点击发布，内容其实是需要显示到 `CommentList` 组件当中的。但这两个组件明显是单独的、分离的组件。我们再回顾一下之前是怎么划分组件的：



可以看到，`CommentApp` 组件将 `CommentInput` 和 `CommentList` 组合起来，它是它们俩的父组件，可以充当桥接两个子组件的桥梁。所以当用户点击发布按钮的时候，我们就将 `CommentInput` 的 `state` 当中最新的评论数据传递给父组件 `CommentApp`，然后让父组件把这个数据传递给 `CommentList` 进行渲染。

`CommentInput` 如何向 `CommentApp` 传递的数据？父组件 `CommentApp` 只需要通过 `props` 给子组件 `CommentInput` 传入一个回调函数。当用户点击发布按钮的时候，`CommentInput` 调用 `props` 中的回调函数并且将 `state` 传入该函数即可。

先给发布按钮添加事件：

```

...
<div className='comment-field-button'>
  <button
    onClick={this.handleSubmit.bind(this)}>
    发布
  </button>
</div>
...
  
```

用户点击按钮的时候会调用 `this.handleSubmit` 方法：

```

...
handleSubmit () {
  if (this.props.onSubmit) {
    const { username, content } = this.state
    this.props.onSubmit({username, content})
  }
  this.setState({ content: '' })
}
...

```

`handleSubmit` 方法会判断 `props` 中是否传入了 `onSubmit` 属性。有的话就调用该函数，并且把用户输入的用户名和评论数据传入该函数。然后再通过 `setState` 清空用户输入的评论内容（但为了用户体验，保留输入的用户名）。

修改 `CommentApp.js`，让它可以通过传入回调来获取到新增评论数据：

```

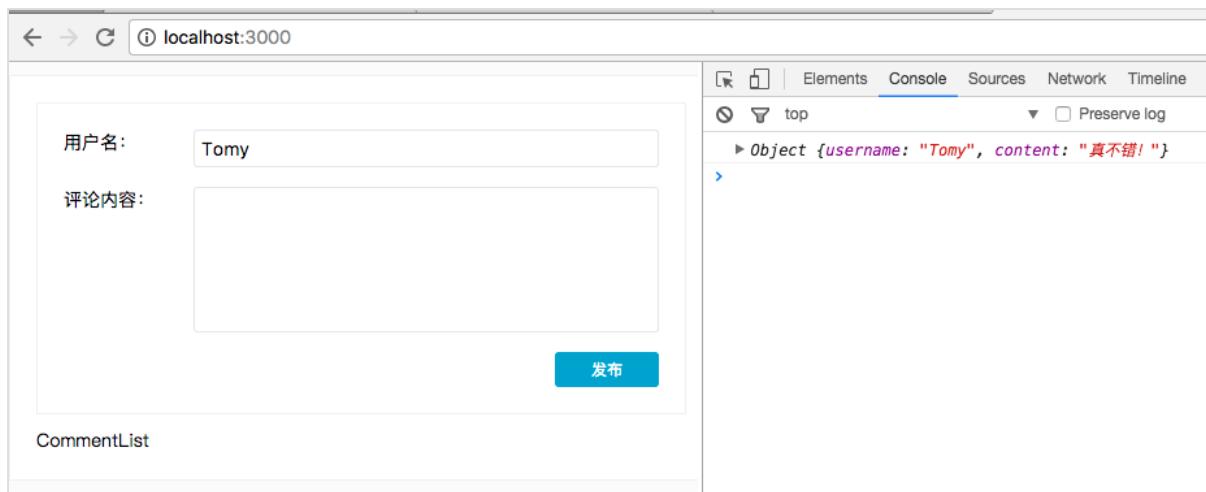
class CommentApp extends Component {
  handleSubmitComment (comment) {
    console.log(comment)
  }

  render() {
    return (
      <div className='wrapper'>
        <CommentInput
          onSubmit={this.handleSubmitComment.bind(this)} />
        <CommentList />
      </div>
    )
  }
}

```

在 `CommentApp` 中给 `CommentInput` 传入一个 `onSubmit` 属性，这个属性值是 `CommentApp` 自己的一个方法 `handleSubmitComment`。这样 `CommentInput` 就可以调用 `this.props.onSubmit(...)` 把数据传给 `CommentApp`。

现在在 `CommentInput` 中输入完评论内容以后点击发布，就可以看到 `CommentApp` 在控制台打印的数据：



这样就顺利地把数据传递给了父组件，接下来我们开始处理评论列表相关的逻辑。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：16. 实战分析：评论功能（三）](#)

[上一节：14. 实战分析：评论功能（一）](#)

React.js 小书

[<-- 返回首页](#)

## 16. 实战分析：评论功能（三）

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson16>
- 转载请注明出处，保留原文链接和作者信息。

接下来的代码比较顺理成章了。修改 `CommentList` 可以让它可以显示评论列表：

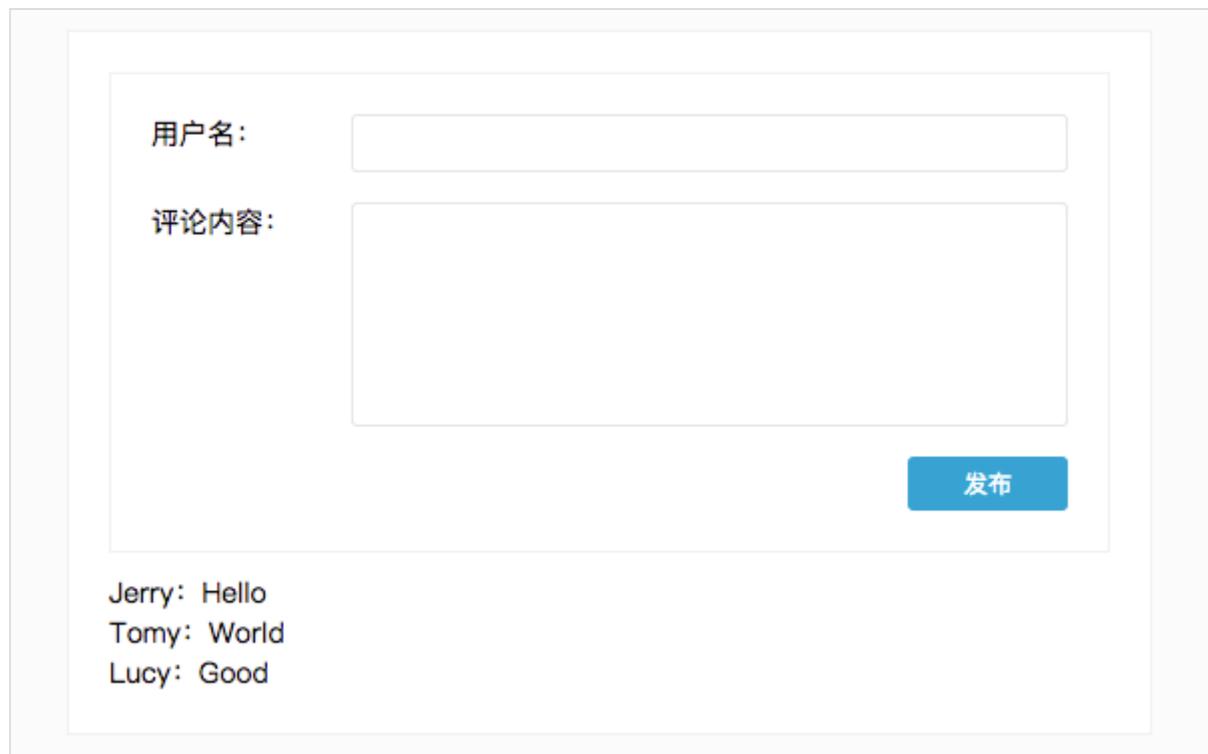
```
// CommentList.js
import React, { Component } from 'react'

class CommentList extends Component {
  render() {
    const comments = [
      {username: 'Jerry', content: 'Hello'},
      {username: 'Tomy', content: 'World'},
      {username: 'Lucy', content: 'Good'}
    ]

    return (
      <div>{comments.map((comment, i) => {
        return (
          <div key={i}>
            {comment.username}: {comment.content}
          </div>
        )
      })}</div>
    )
  }
}

export default CommentList
```

这里的代码没有什么新鲜的内容，只不过是建立了一个 `comments` 的数组来存放一些测试数据的内容，方便我们后续测试。然后把 `comments` 的数据渲染到页面上，这跟我们之前讲解的章节的内容一样—使用 `map` 构建一个存放 `JSX` 的数组。就可以在浏览器看到效果：



修改 `Comment.js` 让它来负责具体每条评论内容的渲染：

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

class Comment extends Component {
  render () {
    return (
      <div className='comment'>
        <div className='comment-user'>
          <span>{this.props.comment.username}</span>:
        </div>
        <p>{this.props.comment.content}</p>
      </div>
    )
  }
}

export default Comment
```

这个组件可能是我们案例里面最简单的组件了，它只负责每条评论的具体显示。你只需要给它的 `props` 中传入一个 `comment` 对象，它就会把该对象中的 `username` 和 `content` 渲染到页面上。

马上把 `Comment` 应用到 `CommentList` 当中，修改 `CommentList.js` 代码：

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

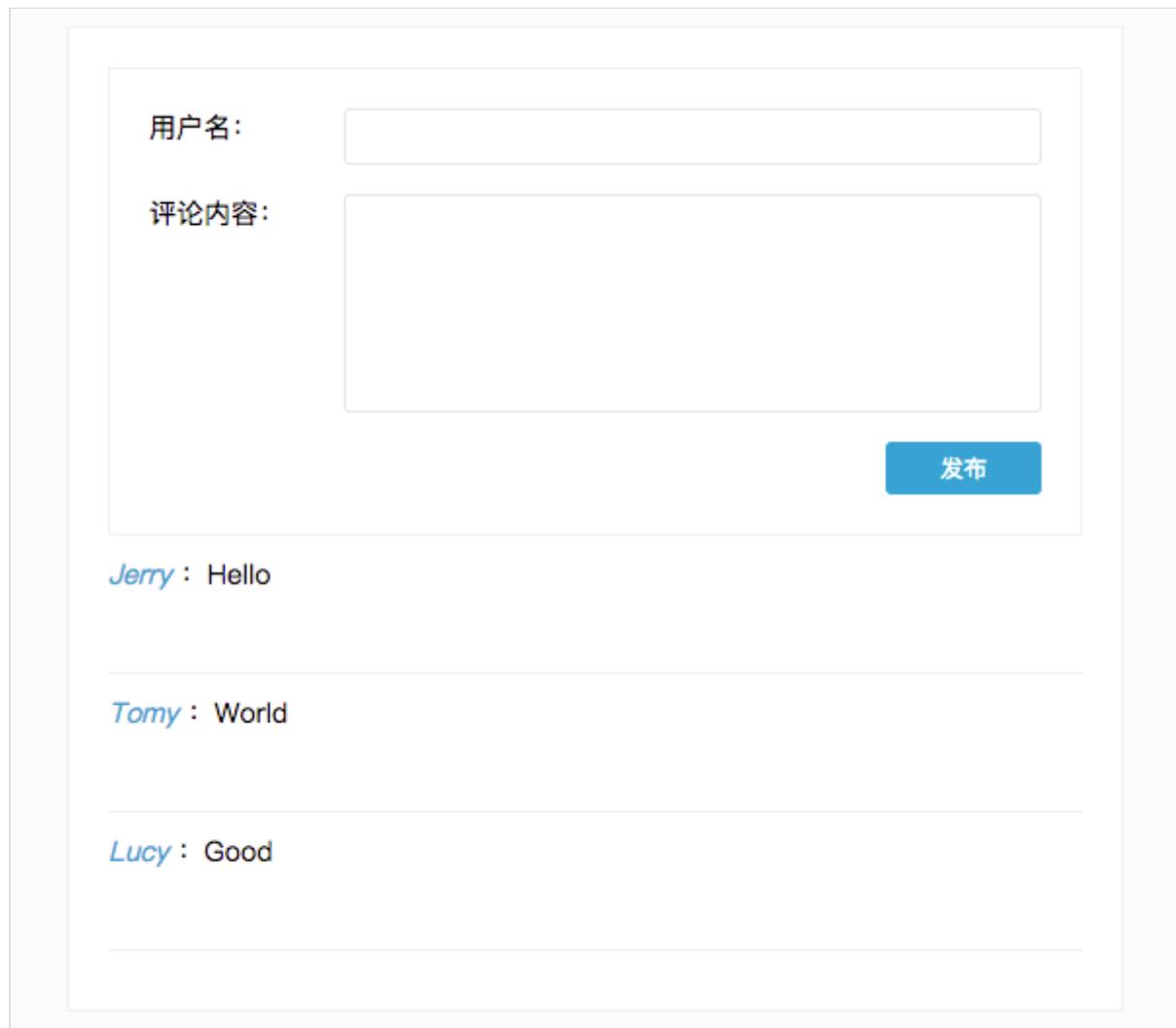
class CommentList extends Component {
```

```
render() {
  const comments = [
    {username: 'Jerry', content: 'Hello'},
    {username: 'Tomy', content: 'World'},
    {username: 'Lucy', content: 'Good'}
  ]

  return (
    <div>
      {comments.map((comment, i) => <Comment comment={comment} key={i} />)}
    </div>
  )
}

export default CommentList
```

可以看到测试数据显示到了页面上：



之前我们说过 `CommentList` 的数据应该是由父组件 `CommentApp` 传进来的，现在我们删除测试数据，改成从 `props` 获取评论数据：

```

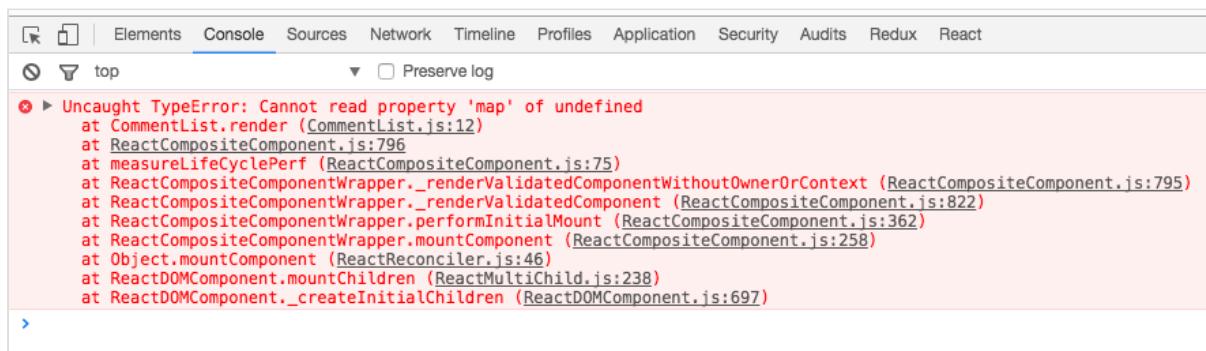
import React, { Component } from 'react'
import Comment from './Comment'

class CommentList extends Component {
  render() {
    return (
      <div>
        {this.props.comments.map((comment, i) =>
          <Comment comment={comment} key={i} />
        )}
      </div>
    )
  }
}

export default CommentList

```

这时候可以看到浏览器报错了：



这是因为 `CommentApp` 使用 `CommentList` 的时候并没有传入 `comments`。我们给 `CommentList` 加上 `defaultProps` 防止 `comments` 不传入的情况：

```

class CommentList extends Component {
  static defaultProps = {
    comments: []
  }
  ...
}

```

这时候代码就不报错了。但是 `CommentInput` 给 `CommentApp` 传递的评论数据并没有传递给 `CommentList`，所以现在发表评论时没有反应的。

我们在 `CommentApp` 的 `state` 中初始化一个数组，来保存所有的评论数据，并且通过 `props` 把它传递给 `CommentList`。修改 `CommentApp.js`：

```

import React, { Component } from 'react'
import CommentInput from './CommentInput'
import CommentList from './CommentList'

```

```

class CommentApp extends Component {
  constructor () {
    super()
    this.state = {
      comments: []
    }
  }

  handleSubmitComment (comment) {
    console.log(comment)
  }

  render() {
    return (
      <div className='wrapper'>
        <CommentInput onSubmit={this.handleSubmitComment.bind(this)} />
        <CommentList comments={this.state.comments}/>
      </div>
    )
  }
}

export default CommentApp

```

接下来，修改 `handleSubmitComment`：每当用户发布评论的时候，就把评论数据插入 `this.state.comments` 中，然后通过 `setState` 把数据更新到页面上：

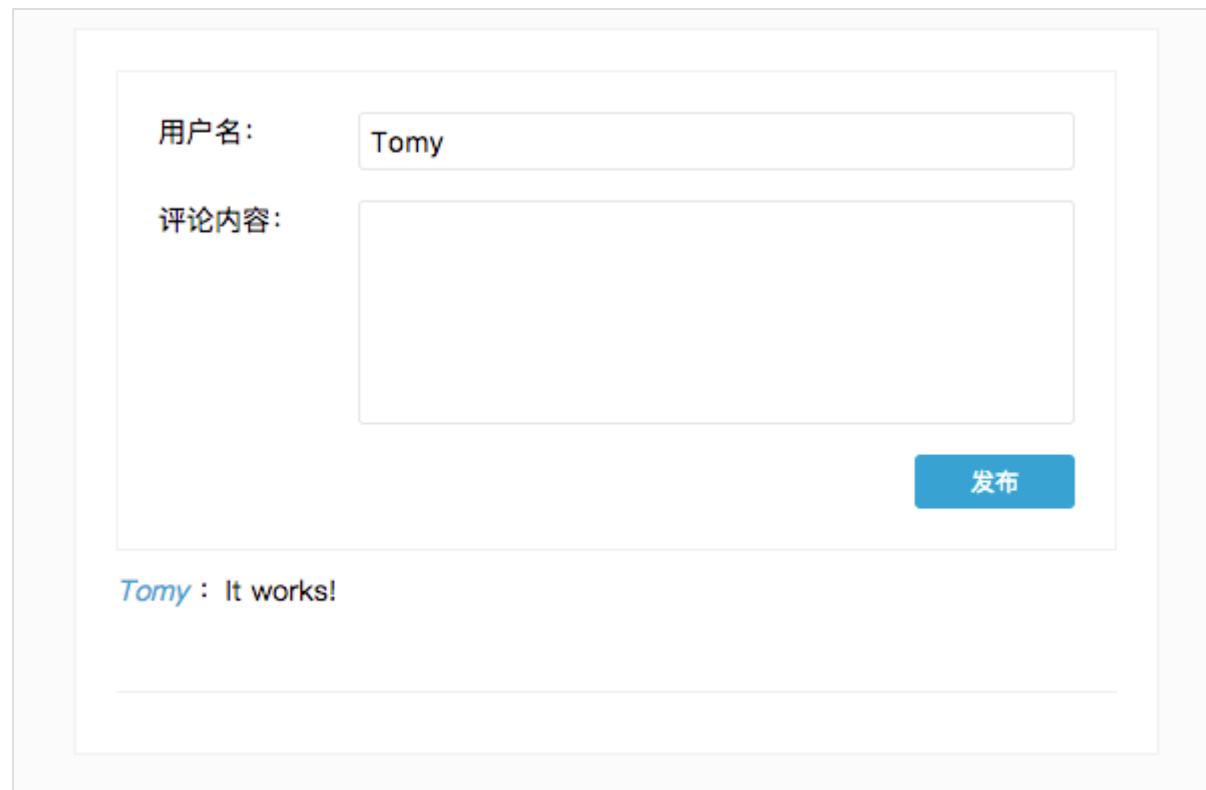
```

...
handleSubmitComment (comment) {
  this.state.comments.push(comment)
  this.setState({
    comments: this.state.comments
  })
}
...

```

小提示：这里的代码直接往 `state.comments` 数组里面插入数据其实违反了 React.js 的 `state` 不可直接修改的原则。但其实这个原则是为了 `shouldComponentUpdate` 的优化和变化的跟踪，而这种目的在使用 React-redux 的时候其实会自然而然达到，我们很少直接手动地优化，这时候这个原则就会显得有点鸡肋。所以这里为了降低大家的理解成本就不强制使用这个原则，有兴趣的朋友可以参考：[Tutorial: Intro To React - React](#)。

现在代码应该是可以按照需求正常运作了，输入用户名和评论内容，然后点击发布：



为了让代码的健壮性更强，给 `handleSubmitComment` 加入简单的数据检查：

```
...
handleSubmitComment (comment) {
  if (!comment) return
  if (!comment.username) return alert('请输入用户名')
  if (!comment.content) return alert('请输入评论内容')
  this.state.comments.push(comment)
  this.setState({
    comments: this.state.comments
  })
}
...
...
```

到这里，我们的第一个实战案例—评论功能已经完成了！完整的案例代码可以在这里 [comment-app](#) 找到， [在线演示](#) 体验。

## 总结

在这个案例里面，我们除了复习了之前所学过的内容以外还学习了新的知识点。包括：

1. 实现功能之前先理解、分析需求，划分组件。并且掌握划分组件的基本原则—可复用性、可维护性。
2. 受控组件的概念，React.js 中的 `<input />`、`<textarea />`、`<select />` 等元素的 `value` 值如果是受到 React.js 的控制，那么就是受控组件。
3. 组件之间使用 `props` 通过父元素传递数据的技巧。

当然，在真实的项目当中，这个案例很多地方是可以优化的。包括组件可复用性方面（有没有发现其实 `CommentInput` 中有重复的代码？）、应用的状态管理方面。但在这里为了给大家总结和演示，实现到这个程度也就足够了。

到此为止，React.js 小书的第一阶段已经结束，你可以利用这些知识点来构建简单的功能模块了。但是在实际项目如果要构建比较系统和完善的功能，还需要更多的 React.js 的知识还有关于前端开发的一些认知来协助我们。接下来我们会开启新的一个阶段来学习更多关于 React.js 的知识，以及如何更加灵活和熟练地使用它们。让我们进入第二阶段吧！

---

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

---

[下一节：17. 前端应用状态管理 — 状态提升](#)

[上一节：15. 实战分析：评论功能（二）](#)

React.js 小书

[<-- 返回首页](#)

## 17. 前端应用状态管理 — 状态提升

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson17>
- 转载请注明出处，保留原文链接和作者信息。

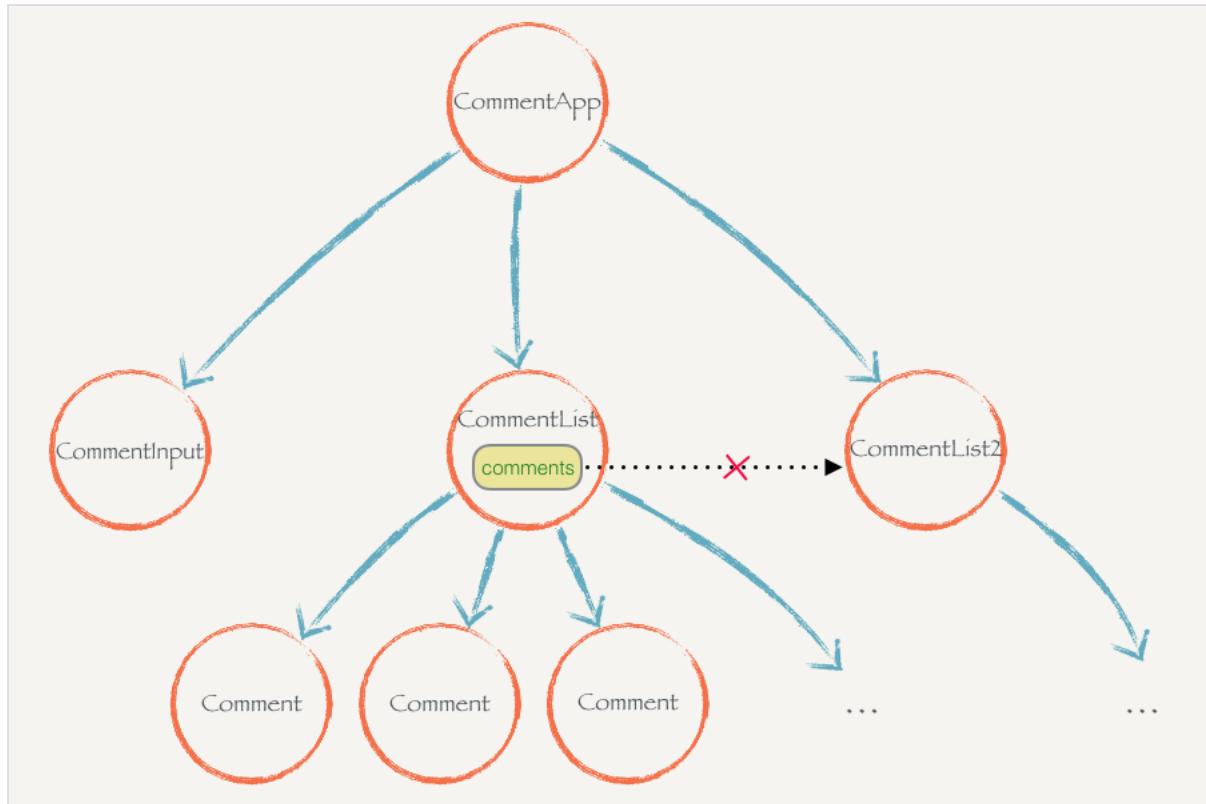
上一个评论功能的案例中，可能会有些同学会对一个地方感到疑惑：`CommentList` 中显示的评论列表数据为什么要通过父组件 `CommentApp` 用 `props` 传进来？为什么不直接存放在 `CommentList` 的 `state` 当中？例如这样做也是可以的：

```
class CommentList extends Component {
  constructor () {
    this.state = { comments: [] }
  }

  addComment (comment) {
    this.state.comments.push(comment)
    this.setState({
      comments: this.state.comments
    })
  }

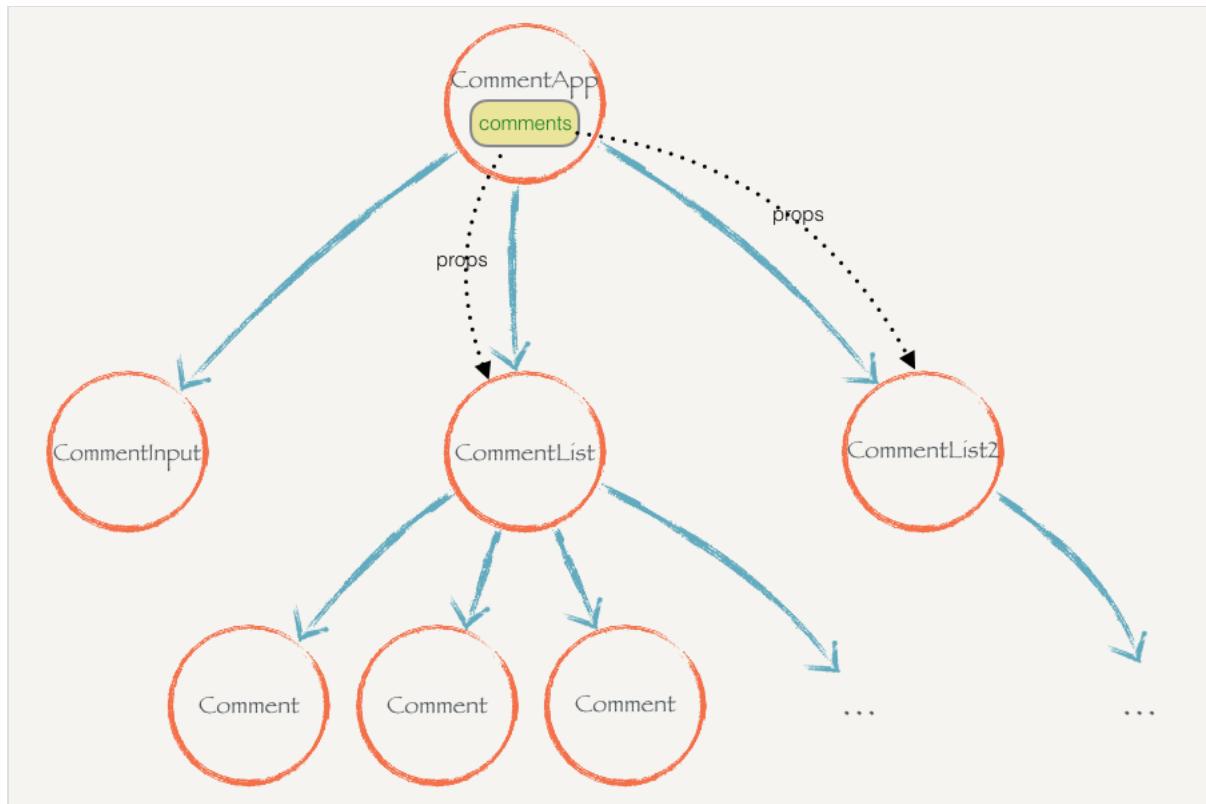
  render() {
    return (
      <div>
        {this.state.comments.map((comment, i) =>
          <Comment comment={comment} key={i} />
        )}
      </div>
    )
  }
}
```

如果把这个 `comments` 放到 `CommentList` 当中，当有别的组件也依赖这个 `comments` 数据或者有别的组件会影响这个数据，那么就带来问题了。举一个数据依赖的例子：例如，现在我们有另外一个和 `CommentList` 同级的 `CommentList2`，也是需要显示同样的评论列表数据。



`CommentList2` 和 `CommentList` 并列为 `CommentApp` 的子组件，它也需要依赖 `comments` 显示评论列表。但是因为 `comments` 数据在 `CommentList` 中，它没办法访问到。

遇到这种情况，我们将这种组件之间共享的状态交给组件最近的公共父节点保管，然后通过 `props` 把状态传递给子组件，这样就可以在组件之间共享数据了。

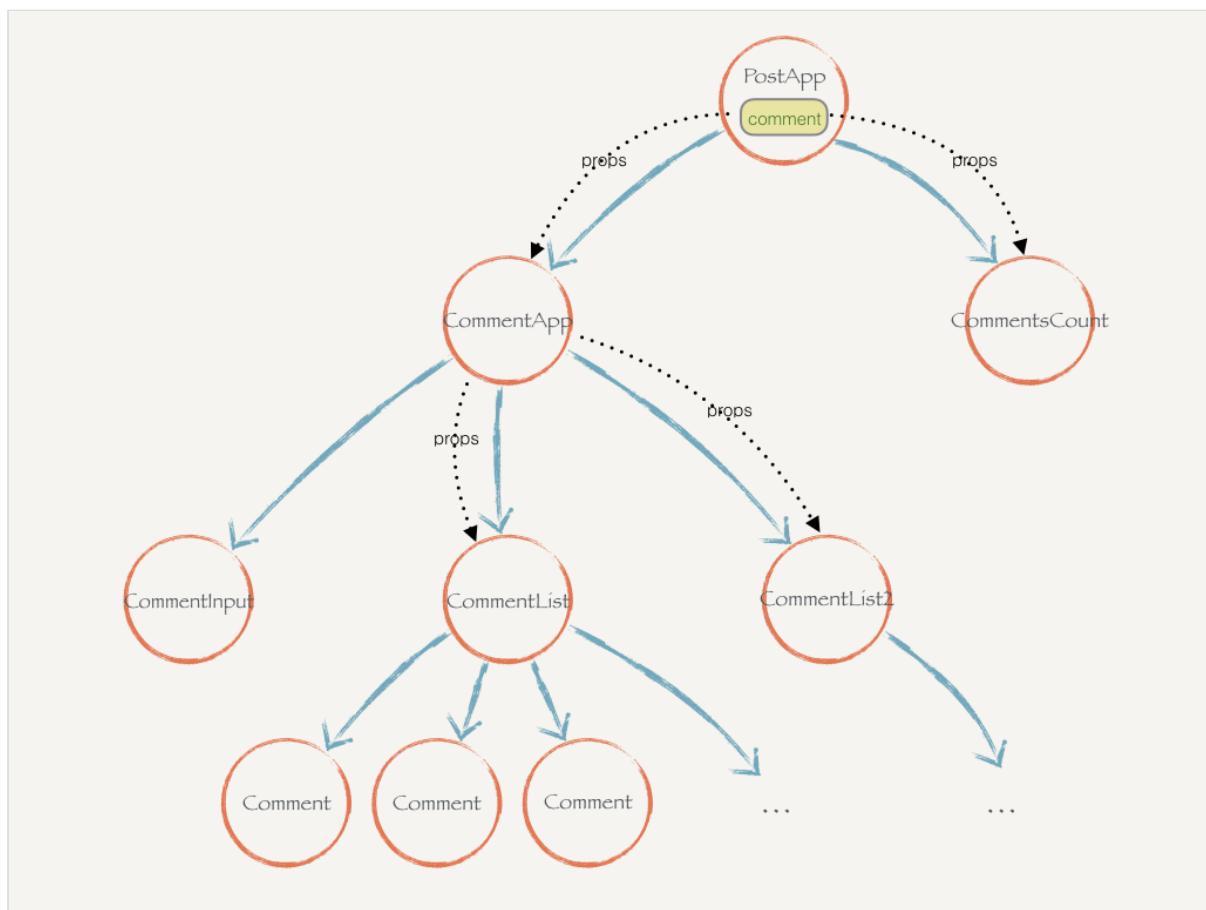


在我们的例子当中，如果把 `comments` 交给父组件 `CommentApp`，那么 `CommentList` 和 `CommentList2` 都可以通过 `props` 获取到 `comments`，React.js 把这种行为叫做“状态提升”。

但是这个 `CommentList2` 是我们临时加上去的，在实际案例当中并没有涉及到这种组件之间依赖 `comments` 的情况，为什么还需要把 `comments` 提升到 `CommentApp`？那是因为有个组件会影响到 `comments`，那就是 `CommentInput`。`CommentInput` 产生的新的评论数据是会插入 `comments` 当中的，所以我们遇到这种情况也会把状态提升到父组件。

总结一下：当某个状态被多个组件依赖或者影响的时候，就把该状态提升到这些组件的最近公共父组件中去管理，用 `props` 传递数据或者函数来管理这种依赖或者影响的行为。

我们来看看状态提升更多的例子，假设现在我们的父组件 `CommentApp` 只是属于更大的组件树 `PostApp` 的一部分：



而这个更大的组件树的另外的子树的 `CommentsCount` 组件也需要依赖 `comments` 来显示评论数，那我们就只能把 `comments` 继续提升到这些依赖组件的最近公共父组件 `PostApp` 当中。

现在继续让我们的例子极端起来。假设现在 `PostApp` 只是另外一个更大的父组件 `Index` 的子树。而 `Index` 的某个子树的有一个按钮组件可以一键清空所有

`comments` (也就是说，这个按钮组件可以影响到这个数据) , 我们只能继续  
`commenets` 提升到 `Index` 当中。

你会发现这种无限制的提升不是一个好的解决方案。一旦发生了提升，你就需要修改原来保存这个状态的组件的代码，也要把整个数据传递路径经过的组件都修改一遍，好让数据能够一层层地传递下去。这样对代码的组织管理维护带来很大的问题。到这里你可以抽象一下问题：

| 如何更好的管理这种被多个组件所依赖或影响的状态？

你可以看到 `React.js` 并没有提供好的解决方案来管理这种组件之间的共享状态。在实际项目当中状态提升并不是一个好的解决方案，所以我们后续会引入 `Redux` 这样的状态管理工具来帮助我们来管理这种共享状态，但是在讲解到 `Redux` 之前，我们暂时采取状态提升的方式来进行管理。

对于不会被多个组件依赖和影响的状态（例如某种下拉菜单的展开和收起状态），一般来说只需要保存在组件内部即可，不需要做提升或者特殊的管理。

## 课后练习

- [百分比换算器](#)

| 因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

下一节：18. 挂载阶段的组件生命周期（一）

上一节：16. 实战分析：评论功能（三）

React.js 小书

[← 返回首页](#)

## 18. 挂载阶段的组件生命周期（一）

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson18>
- 转载请注明出处，保留原文链接和作者信息。

我们在[讲解 JSX 的章节](#)中提到，下面的代码：

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

会编译成：

```
ReactDOM.render(  
  React.createElement(Header, null),  
  document.getElementById('root')  
)
```

其实我们把 `Header` 组件传给了 `React.createElement` 函数，又把函数返回结果传给了 `ReactDOM.render`。我们可以简单猜想一下它们会干什么事情：

```
// React.createElement 中实例化一个 Header  
const header = new Header(props, children)  
// React.createElement 中调用 header.render 方法渲染组件的内容  
const headerJsxObject = header.render()  
  
// ReactDOM 用渲染后的 JavaScript 对象来构建真正的 DOM 元素  
const headerDOM = createDOMFromObject(headerJsxObject)  
// ReactDOM 把 DOM 元素塞到页面上  
document.getElementById('root').appendChild(headerDOM)
```

上面过程其实很简单，看代码就能理解。

我们把 **React.js 将组件渲染，并且构造 DOM 元素然后塞入页面的过程称为组件的挂载**（这个定义请好好记住）。其实 `React.js` 内部对待每个组件都有这么一个过程，也就是初始化组件 -> 挂载到页面上的过程。所以你可以理解一个组件的方法调用是这么一个过程：

```
-> constructor()
-> render()
// 然后构造 DOM 元素插入页面
```

这当然是很好理解的。React.js 为了让我们能够更好的掌控组件的挂载过程，往上面插入了两个方法：

```
-> constructor()
-> componentWillMount()
-> render()
// 然后构造 DOM 元素插入页面
-> componentDidMount()
```

`componentWillMount` 和 `componentDidMount` 都是可以像 `render` 方法一样自定义在组件的内部。挂载的时候，React.js 会在组件的 `render` 之前调用 `componentWillMount`，在 DOM 元素塞入页面以后调用 `componentDidMount`。

我们给 `Header` 组件加上这两个方法，并且打一些 Log：

```
class Header extends Component {
  constructor () {
    super()
    console.log('construct')
  }

  componentWillMount () {
    console.log('component will mount')
  }

  componentDidMount () {
    console.log('component did mount')
  }

  render () {
    console.log('render')
    return (
      <div>
        <h1 className='title'>React 小书</h1>
      </div>
    )
  }
}
```

在控制台你可以看到依次输出：

```

construct
component will mount
render
component did mount

```

可以看到，React.js 确实按照我们上面所说的那样调用了定义的两个方法 `componentWillMount` 和 `componentDidMount`。

机灵的同学可以想到，一个组件可以插入页面，当然也可以从页面中删除。

```

-> constructor()
-> componentWillMount()
-> render()
// 然后构造 DOM 元素插入页面
-> componentDidMount()
// ...
// 从页面中删除

```

React.js 也控制了这个组件的删除过程。在组件删除之前 React.js 会调用组件定义的 `componentWillUnmount`：

```

-> constructor()
-> componentWillMount()
-> render()
// 然后构造 DOM 元素插入页面
-> componentDidMount()
// ...
// 即将从页面中删除
-> componentWillUnmount()
// 从页面中删除

```

看看什么情况下会把组件从页面中删除，继续使用上面例子的代码，我们再定义一个 `Index` 组件：

```

class Index extends Component {
  constructor() {
    super()
    this.state = {
      isShowHeader: true
    }
  }

  handleShowOrHide () {
    this.setState({
      isShowHeader: !this.state.isShowHeader
    })
  }
}

```

```

render () {
  return (
    <div>
      {this.state.isShowHeader ? <Header /> : null}
      <button onClick={this.handleShowOrHide.bind(this)}>
        显示或者隐藏标题
      </button>
    </div>
  )
}

ReactDOM.render(
  <Index />,
  document.getElementById('root')
)

```

`Index` 组件使用了 `Header` 组件，并且有一个按钮，可以控制 `Header` 的显示或者隐藏。下面这行代码：

```

...a
{this.state.isShowHeader ? <Header /> : null}
...

```

相当于 `state.isShowHeader` 为 `true` 的时候把 `Header` 插入页面，`false` 的时候把 `Header` 从页面上删除。这时候我们给 `Header` 添加 `componentWillUnmount` 方法：

```

...
componentWillUnmount() {
  console.log('component will unmount')
}
...

```

这时候点击页面上的按钮，你会看到页面的标题隐藏了，并且控制台打印出来下图的最后一行，说明 `componentWillUnmount` 确实被 `React.js` 所调用了：

construct
component will mount
render
component did mount
component will unmount

你可以多次点击按钮，随着按钮的显示和隐藏，上面的内容会按顺序重复地打印出来，可以体会一下这几个方法的调用过程和顺序。

## 总结

React.js 将组件渲染，并且构造 DOM 元素然后塞入页面的过程称为组件的挂载。这一节我们学习了 React.js 控制组件在页面上挂载和删除过程里面几个方法：

- `componentWillMount`：组件挂载开始之前，也就是在组件调用 `render` 方法之前调用。
- `componentDidMount`：组件挂载完成以后，也就是 DOM 元素已经插入页面后调用。
- `componentWillUnmount`：组件对应的 DOM 元素从页面中删除之前调用。

但这一节并没有讲这几个方法到底在实际项目当中有什么作用，下一节我们通过例子来讲解一下这几个方法的用途。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：19. 挂载阶段的组件生命周期（二）](#)

[上一节：17. 前端应用状态管理 — 状态提升](#)

React.js 小书

[<-- 返回首页](#)

## 19. 挂载阶段的组件生命周期（二）

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson19>
- 转载请注明出处，保留原文链接和作者信息。

这一节我们来讨论一下对于一个组件来说，`constructor`、`componentWillMount`、`componentDidMount`、`componentWillUnmount` 这几个方法在一个组件的出生到死亡的过程里面起了什么样的作用。

一般来说，所有关于组件自身的状态的初始化工作都会放在 `constructor` 里面去做。你会发现本书所有组件的 `state` 的初始化工作都是放在 `constructor` 里面的。假设我们现在在做一个时钟应用：

现在的时间是  
下午3:35:58

我们会在 `constructor` 里面初始化 `state.date`，当然现在页面还是静态的，等下一会让时间动起来。

```
class Clock extends Component {
  constructor () {
    super()
    this.state = {
      date: new Date()
    }
  }

  render () {
    return (
      <div>
        <h1>
          <p>现在的时间是</p>
```

```

        {this.state.date.toLocaleTimeString()}
    </h1>
</div>
)
}
}

```

一些组件启动的动作，包括像 Ajax 数据的拉取操作、一些定时器的启动等，就可以放在 `componentWillMount` 里面进行，例如 Ajax：

```

...
componentWillMount () {
    ajax.get('http://json-api.com/user', (userData) => {
        this.setState({ userData })
    })
}
...

```

当然在我们这个例子里面是定时器的启动，我们给 `Clock` 启动定时器：

```

class Clock extends Component {
    constructor () {
        super()
        this.state = {
            date: new Date()
        }
    }

    componentWillMount () {
        this.timer = setInterval(() => {
            this.setState({ date: new Date() })
        }, 1000)
    }
    ...
}

```

我们在 `componentWillMount` 中用 `setInterval` 启动了一个定时器：每隔 1 秒更新中的 `state.date`，这样页面就可以动起来了。我们用一个 `Index` 把它用起来，并且插入页面：

```

class Index extends Component {
    render () {
        return (
            <div>
                <Clock />
            </div>
        )
    }
}

```

```

        }
    }

ReactDOM.render(
    <Index />,
    document.getElementById('root')
)

```

像上一节那样，我们修改这个 `Index` 让这个时钟可以隐藏或者显示：

```

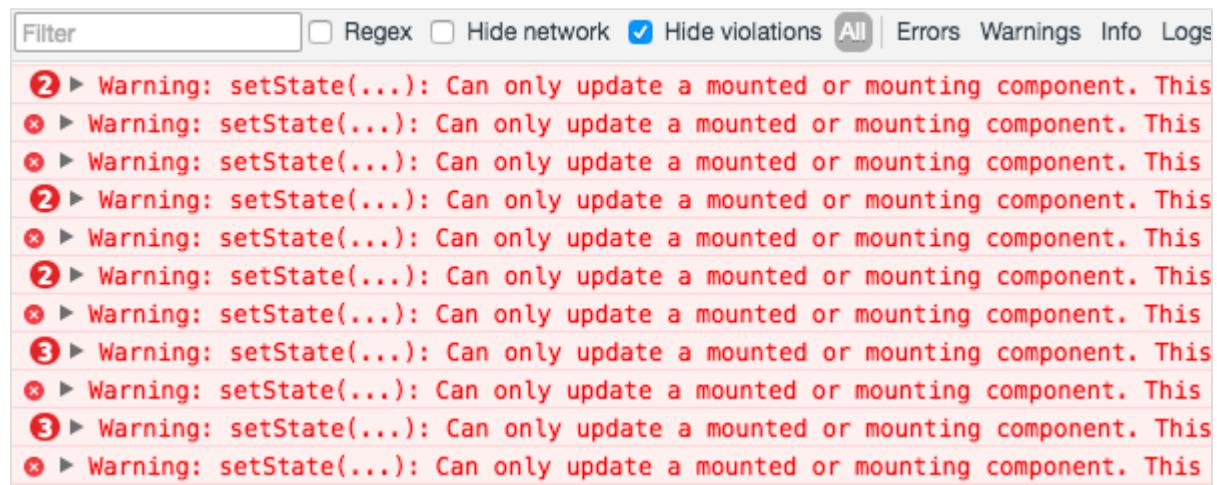
class Index extends Component {
    constructor () {
        super()
        this.state = { isShowClock: true }
    }

    handleShowOrHide () {
        this.setState({
            isShowClock: !this.state.isShowClock
        })
    }

    render () {
        return (
            <div>
                {this.state.isShowClock ? <Clock /> : null }
                <button onClick={this.handleShowOrHide.bind(this)}>
                    显示或隐藏时钟
                </button>
            </div>
        )
    }
}

```

现在页面上有个按钮可以显示或者隐藏时钟。你试一下显示或者隐藏时钟，虽然页面上看起来功能都正常，在控制台你会发现报错了：



这是因为，**当时钟隐藏的时候，我们并没有清除定时器**。时钟隐藏的时候，定时器的回调函数还在不停地尝试 `setState`，由于 `setState` 只能在已经挂载或者正在挂载的组件上调用，所以 `React.js` 开始疯狂报错。

多次的隐藏和显示会让 `React.js` 重新构造和销毁 `Clock` 组件，每次构造都会重新构建一个定时器。而销毁组件的时候没有清除定时器，所以你看到报错会越来越多。而且因为 `JavaScript` 的闭包特性，这样会导致严重的内存泄漏。

这时候 `componentWillUnmount` 就可以派上用场了，它的作用就是在组件销毁的时候，做这种清场的工作。例如清除该组件的定时器和其他的数据清理工作。我们给 `Clock` 添加 `componentWillUnmount`，在组件销毁的时候清除该组件的定时器：

```
...
componentWillUnmount () {
  clearInterval(this.timer)
}
...
```

这时候就没有错误了。

## 总结

我们一般会把组件的 `state` 的初始化工作放在 `constructor` 里面去做；在 `componentWillMount` 进行组件的启动工作，例如 `Ajax` 数据拉取、定时器的启动；组件从页面上销毁的时候，有时候需要一些数据的清理，例如定时器的清理，就会放在 `componentWillUnmount` 里面去做。

说一下本节没有提到的 `componentDidMount`。一般来说，有些组件的启动工作是依赖 `DOM` 的，例如动画的启动，而在 `componentWillMount` 的时候组件还没挂载完成，所以没法进行这些启动工作，这时候就可以把这些操作放在 `componentDidMount` 当中。

`componentDidMount` 的具体使用我们会在接下来的章节当中结合 `DOM` 来讲。

## 课后练习

- [React.js 加载、刷新数据](#)

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：20. 更新阶段的组件生命周期](#)

[上一节：18. 挂载阶段的组件生命周期（一）](#)



React.js 小书

[<-- 返回首页](#)

## 20. 更新阶段的组件生命周期

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson20>
- 转载请注明出处，保留原文链接和作者信息。

从之前的章节我们了解到，组件的挂载指的是将组件渲染并且构造 DOM 元素然后插入页面的过程。**这是一个从无到有的过程**，React.js 提供一些生命周期函数可以给我们在这个过程中做一些操作。

除了挂载阶段，还有一种“更新阶段”。说白了就是 `setState` 导致 React.js 重新渲染组件并且把组件的变化应用到 DOM 元素上的过程，**这是一个组件的变化过程**。而 React.js 也提供了一系列的生命周期函数可以让我们在这个组件更新的过程执行一些操作。

这些生命周期在深入项目开发阶段是非常重要的。而要完全理解更新阶段的组件生命周期是一个需要经验和知识积累的过程，你需要对 Virtual-DOM 策略有比较深入理解才能完全掌握，但这超出了本书的目的。**本书的目的是为了让大家快速掌握 React.js 核心的概念，快速上手项目进行实战**。所以对于组件更新阶段的组件生命周期，我们简单提及并且提供一些资料给大家。

这里为了知识的完整，补充关于更新阶段的组件生命周期：

- `shouldComponentUpdate(nextProps, nextState)`：你可以通过这个方法控制组件是否重新渲染。如果返回 `false` 组件就不会重新渲染。这个生命周期在 React.js 性能优化上非常有用。
- `componentWillReceiveProps(nextProps)`：组件从父组件接收到新的 `props` 之前调用。
- `componentWillUpdate()`：组件开始重新渲染之前调用。
- `componentDidUpdate()`：组件重新渲染并且把更改变更到真实的 DOM 以后调用。

大家对这更新阶段的生命周期比较感兴趣的话可以查看[官网文档](#)。

但这里建议大家可以先简单了解 React.js 组件是有更新阶段的，并且有这么几个更新阶段的生命周期即可。然后在深入项目实战的时候逐渐地掌握理解他们，现在并不需要对他们放过多的精力。

有朋友对 Virtual-DOM 策略比较感兴趣的话，可以参考这篇博客：[深度剖析：如何实现一个 Virtual DOM 算法](#)。对深入理解 React.js 核心算法有一定帮助。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

---

下一节：21. ref 和 React.js 中的 DOM 操作

上一节：19. 挂载阶段的组件生命周期（二）

React.js 小书

[<-- 返回首页](#)

## 21. ref 和 React.js 中的 DOM 操作

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson21>
- 转载请注明出处，保留原文链接和作者信息。

在 React.js 当中你基本不需要和 DOM 直接打交道。React.js 提供了一系列的 `on*` 方法帮助我们进行事件监听，所以 React.js 当中不需要直接调用 `addEventListener` 的 DOM API；以前我们通过手动 DOM 操作进行页面更新（例如借助 jQuery），而在 React.js 当中可以直接通过 `setState` 的方式重新渲染组件，渲染的时候可以把新的 `props` 传递给子组件，从而达到页面更新的效果。

React.js 这种重新渲染的机制帮助我们免除了绝大部分的 DOM 更新操作，也让类似于 jQuery 这种以封装 DOM 操作为主的第三方的库从我们的开发工具链中删除。

但是 React.js 并不能完全满足所有 DOM 操作需求，有些时候我们还是需要和 DOM 打交道。比如说你想进入页面以后自动 `focus` 到某个输入框，你需要调用 `input.focus()` 的 DOM API，比如说你想动态获取某个 DOM 元素的尺寸来做后续的动画，等等。

React.js 当中提供了 `ref` 属性来帮助我们获取已经挂载的元素的 DOM 节点，你可以给某个 JSX 元素加上 `ref` 属性：

```
class AutoFocusInput extends Component {  
  componentDidMount () {  
    this.input.focus()  
  }  
  
  render () {  
    return (  
      <input ref={(input) => this.input = input} />  
    )  
  }  
}  
  
ReactDOM.render(  
  <AutoFocusInput />,  
  document.getElementById('root')  
)
```

可以看到我们给 `input` 元素加了一个 `ref` 属性，这个属性值是一个函数。当 `input` 元素在页面上挂载完成以后，React.js 就会调用这个函数，并且把这个挂载以后的 DOM 节点传给这个函数。在函数中我们把这个 DOM 元素设置为组件实例的一个属性，这样以后我们就可以通过 `this.input` 获取到这个 DOM 元素。

然后我们就可以在 `componentDidMount` 中使用这个 DOM 元素，并且调用 `this.input.focus()` 的 DOM API。整体就达到了页面加载完成就自动 `focus` 到输入框的功能（大家可以注意到我们用上了 `componentDidMount` 这个组件生命周期）。

我们可以给任意代表 HTML 元素标签加上 `ref` 从而获取到它 DOM 元素然后调用 DOM API。但是记住一个原则：**能不用 `ref` 就不用**。特别是要避免用 `ref` 来做 React.js 本来就可以帮助你做到的页面自动更新的操作和事件监听。多余的 DOM 操作其实是代码里面的“噪音”，不利于我们理解和维护。

顺带一提的是，其实可以给组件标签也加上 `ref`，例如：

```
<Clock ref={(clock) => this.clock = clock} />
```

这样你获取到的是这个 `Clock` 组件在 React.js 内部初始化的实例。但这并不是什么常用的做法，而且也并不建议这么做，所以这里就简单提及，有兴趣的朋友可以自己学习探索。

## 课后练习

- [获取文本的高度](#)

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：22. props.children 和容器类组件](#)

[上一节：20. 更新阶段的组件生命周期](#)

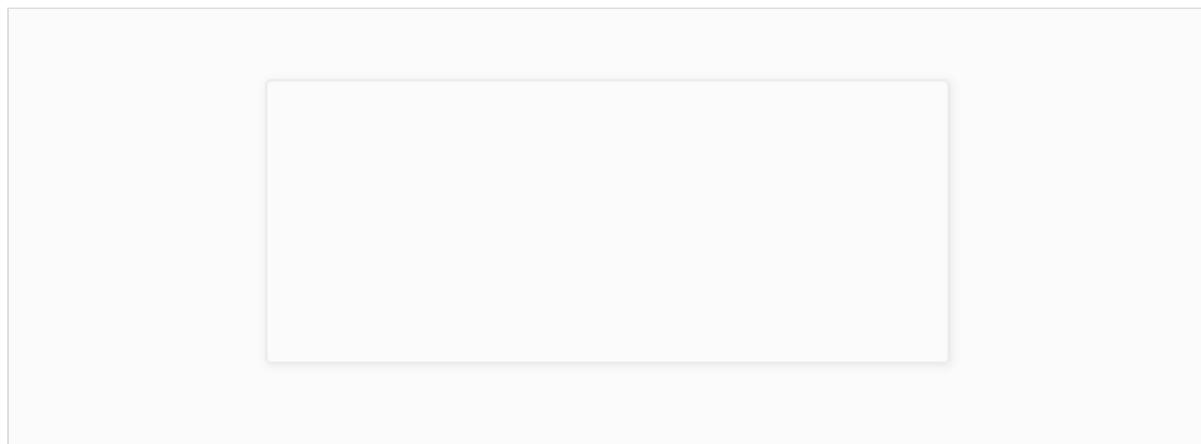
React.js 小书

[<-- 返回首页](#)

## 22. props.children 和容器类组件

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson22>
- 转载请注明出处，保留原文链接和作者信息。

有一类组件，充当了容器的作用，它定义了一种外层结构形式，然后你可以往里面塞任意的内容。这种结构在实际当中非常常见，例如这种带卡片组件：



组件本身是一个不带任何内容的方形的容器，我可以在用这个组件的时候给它传入任意内容：



基于我们目前的知识储备，可以迅速写出这样的代码：

```
class Card extends Component {  
  render () {  
    return (  
      <div className='card'>
```

```

        <div className='card-content'>
            {this.props.content}
        </div>
    </div>
)
}
}

ReactDOM.render(
<Card content={
    <div>
        <h2>React.js 小书</h2>
        <div>开源、免费、专业、简单</div>
        订阅: <input />
    </div>
} />,
document.getElementById('root')
)

```

我们通过给 `Card` 组件传入一个 `content` 属性，这个属性可以传入任意的 JSX 结构。然后在 `Card` 内部会通过 `{this.props.content}` 把内容渲染到页面上。

这样明显太丑了，如果 `Card` 除了 `content` 以外还能传入其他属性的话，那么这些 JSX 和其他属性就会混在一起。很不好维护，如果能像下面的代码那样使用 `Card` 那想必也是极好的：

```

ReactDOM.render(
<Card>
    <h2>React.js 小书</h2>
    <div>开源、免费、专业、简单</div>
    订阅: <input />
</Card>,
document.getElementById('root')
)

```

如果组件标签也能像普通的 HTML 标签那样编写内嵌的结构，那么就方便很多了。实际上，React.js 默认就支持这种写法，所有嵌套在组件中的 JSX 结构都可以在组件内部通过 `props.children` 获取到：

```

class Card extends Component {
    render () {
        return (
            <div className='card'>
                <div className='card-content'>
                    {this.props.children}
                </div>
            </div>
        )
    }
}

```

```

        }
    }
}

```

把 `props.children` 打印出来，你可以看到它其实是个数组：

```

▼ Array[4] ⓘ
▶ 0: Object
▶ 1: Object
▶ 2: "订阅："
▶ 3: Object
  length: 4
▶ __proto__: Array[0]

```

React.js 就是把我们嵌套的 JSX 元素一个个都放到数组当中，然后通过 `props.children` 传给了 `Card`。

由于 JSX 会把插入表达式里面数组中的 JSX 一个个罗列下来显示。所以其实就相当于在 `Card` 中嵌套了什么 JSX 结构，都会显示在 `Card` 的类名为 `card-content` 的 `div` 元素当中。

这种嵌套的内容成为了 `props.children` 数组的机制使得我们编写组件变得非常的灵活，我们甚至可以在组件内部把数组中的 JSX 元素安置在不同的地方：

```

class Layout extends Component {
  render () {
    return (
      <div className='two-cols-layout'>
        <div className='sidebar'>
          {this.props.children[0]}
        </div>
        <div className='main'>
          {this.props.children[1]}
        </div>
      </div>
    )
  }
}

```

这是一个两列布局组件，嵌套的 JSX 的第一个结构会成为侧边栏，第二个结构会成为内容栏，其余的结构都会被忽略。这样通过这个布局组件，就可以在各个地方高度复用我们的布局。

## 总结

使用自定义组件的时候，可以在其中嵌套 JSX 结构。嵌套的结构在组件内部都可以通过 `props.children` 获取到，这种组件编写方式在编写容器类型的组件当中非常有用。而在实际的 React.js 项目当中，我们几乎每天都需要用这种方式来编写组件。

## 课后练习

### \*黑色边框的容器组件

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

下一节：23. dangerouslySetHTML 和 style 属性

上一节：21. ref 和 React.js 中的 DOM 操作

React.js 小书

[<-- 返回首页](#)

## 23. dangerouslySetHTML 和 style 属性

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson23>
- 转载请注明出处，保留原文链接和作者信息。

这一节我们来补充两个之前没有提到的属性，但是在 React.js 组件开发中也非常常用，但是它们也很简单。

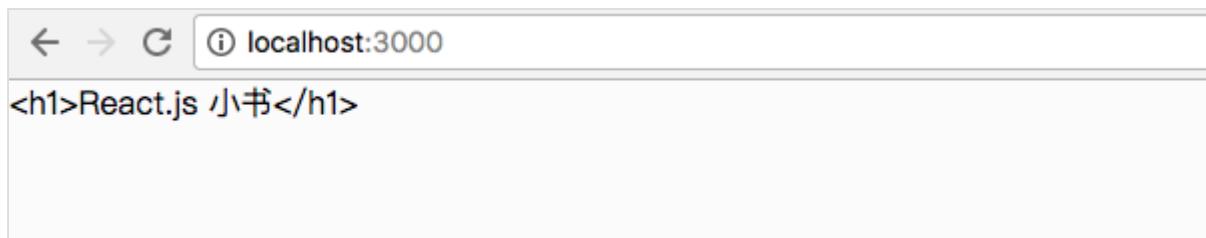
### dangerouslySetHTML

出于安全考虑的原因（xss 攻击），在 React.js 当中所有的表达式插入的内容都会被自动转义，就相当于 jQuery 里面的 `text(...)` 函数一样，任何的 HTML 格式都会被转义掉：

```
class Editor extends Component {
  constructor() {
    super()
    this.state = {
      content: '<h1>React.js 小书</h1>'
    }
  }

  render () {
    return (
      <div className='editor-wrapper'>
        {this.state.content}
      </div>
    )
  }
}
```

假设上面是一个富文本编辑器组件，富文本编辑器的内容是动态的 HTML 内容，用 `this.state.content` 来保存。我希望在编辑器内部显示这个动态 HTML 结构，但是因为 React.js 的转义特性，页面上会显示：



表达式插入并不会把一个 `<h1>` 渲染到页面，而是把它的文本形式渲染了。那要怎么才能做到设置动态 HTML 结构的效果呢？React.js 提供了一个属性 `dangerouslySetInnerHTML`，可以让我们设置动态设置元素的 `innerHTML`：

```
...
  render () {
    return (
      <div
        className='editor-wrapper'
        dangerouslySetInnerHTML={{__html: this.state.content}} />
    )
  }
...

```

需要给 `dangerouslySetInnerHTML` 传入一个对象，这个对象的 `__html` 属性值就相当于元素的 `innerHTML`，这样我们就可以动态渲染元素的 `innerHTML` 结构了。

有写朋友会觉得很奇怪，为什么要把一件这么简单的事情搞得这么复杂，名字又长，还要传入一个奇怪的对象。那是因为设置 `innerHTML` 可能会导致跨站脚本攻击（XSS），所以 React.js 团队认为把事情搞复杂可以防止（警示）大家滥用这个属性。这个属性不必要的情况就不要使用。

## style

React.js 中的元素的 `style` 属性的用法和 DOM 里面的 `style` 不大一样，普通的 HTML 中的：

```
<h1 style='font-size: 12px; color: red;'>React.js 小书</h1>
```

在 React.js 中你需要把 CSS 属性变成一个对象再传给元素：

```
<h1 style={{fontSize: '12px', color: 'red'}}>React.js 小书</h1>
```

`style` 接受一个对象，这个对象里面是这个元素的 CSS 属性键值对，原来 CSS 属性中带 `-` 的元素都必须要去掉 `-` 换成驼峰命名，如 `font-size` 换成 `fontSize`，`text-align` 换成 `textAlign`。

用对象作为 `style` 方便我们动态设置元素的样式。我们可以用 `props` 或者 `state` 中的数据生成样式对象再传给元素，然后用 `setState` 就可以修改样式，非常灵活：

```
<h1 style={{fontSize: '12px', color: this.state.color}}>React.js 小书</h1>
```

只要简单地 `setState({color: 'blue'})` 就可以修改元素的颜色成蓝色。

## 课后练习

### \*覆盖默认样式

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：24. PropTypes 和组件参数验证](#)

[上一节：22. props.children 和容器类组件](#)

React.js 小书

[<-- 返回首页](#)

## 24. PropTypes 和组件参数验证

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson24>
- 转载请注明出处，保留原文链接和作者信息。

我们来了到了一个非常尴尬的章节，很多初学的朋友可能对这一章的知识点不屑一顾，觉得用不用对程序功能也没什么影响。但其实这一章节的知识在你构建多人协作、大型的应用程序的时候也是非常重要的，不可忽视。

都说 JavaScript 是一门灵活的语言 — 这就是像是说“你是个好人”一样，凡事都有背后没有说出来的话。JavaScript 的灵活性体现在弱类型、高阶函数等语言特性上。而语言的弱类型一般来说确实让我们写代码很爽，但是也很容易出 bug。

变量没有固定类型可以随意赋值，在我们构建大型应用程序的时候并不是什么好的事情。你写下了 `let a = {}`，如果这是个共享的状态并且在某个地方把 `a = 3`，那么 `a.xxx` 就会让程序崩溃了。而这种非常隐晦但是低级的错误在强类型的语言例如 C/C++、Java 中是不可能发生的，这些代码连编译都不可能通过，也别妄图运行。

大型应用程序的构建其实更适合用强类型的语言来构建，它有更多的规则，可以帮助我们在编写代码阶段、编译阶段规避掉很多问题，让我们的应用程序更加的安全。JavaScript 早就脱离了玩具语言的领域并且投入到大型的应用程序的生产活动中，因为它的弱类型，常常意味着不是很安全。所以近年来出现了类似 TypeScript 和 Flow 等技术，来弥补 JavaScript 这方面的缺陷。

React.js 的组件其实是为了构建大型应用程序而生。但是因为 JavaScript 这样的特性，你在编写了一个组件以后，根本不知道别人会怎么使用你的组件，往里传什么乱七八糟的参数，例如评论组件：

```
class Comment extends Component {
  const { comment } = this.props
  render () {
    return (
      <div className='comment'>
        <div className='comment-user'>
          <span>{comment.username} </span>:
        </div>
        <p>{comment.content}</p>
      </div>
    )
  }
}
```

```

    }
}

```

但是别人往里面传一个数字你拿他一点办法都没有：

```
<Comment comment={1} />
```

JavaScript 在这种情况下是不会报任何错误的，但是页面就是显示不正常，然后我们踏上了漫漫 debug 的路程。这里的例子还是过于简单，容易 debug，但是对于比较复杂的成因和情况是比较难处理的。

于是 React.js 就提供了一种机制，让你可以给组件的配置参数加上类型验证，就用上述的评论组件例子，你可以配置 `Comment` 只能接受对象类型的 `comment` 参数，你传个数字进来组件就强制报错。我们这里先安装一个 React 提供的第三方库 `prop-types`：

```
npm install --save prop-types
```

它可以帮助我们验证 `props` 的参数类型，例如：

```

import React, { Component } from 'react'
import PropTypes from 'prop-types'

class Comment extends Component {
  static propTypes = {
    comment: PropTypes.object
  }

  render () {
    const { comment } = this.props
    return (
      <div className='comment'>
        <div className='comment-user'>
          <span>{comment.username}</span>:
        </div>
        <p>{comment.content}</p>
      </div>
    )
  }
}

```

注意我们在文件头部引入了 `PropTypes`，并且给 `Comment` 组件类添加了类属性 `propTypes`，里面的内容的意思就是你传入的 `comment` 类型必须为 `object` (对象)。

这时候如果再往里面传入数字，浏览器就会报错：

```
✖ ▶ Warning: Failed prop type: Invalid prop `comment` of type `number` supplied to `Comment`, expected `object`.
  in Comment (at index.js:376)
```

出错信息明确告诉我们：你给 `Comment` 组件传了一个数字类型的 `comment`，而它应该是 `object`。你就清晰知道问题出在哪里了。

虽然 `propTypes` 帮我们指定了参数类型，但是并没有说这个参数一定要传入，事实上，这些参数默认都是可选的。可选参数我们可以通过配置 `defaultProps`，让它在不传入的时候有默认值。但是我们这里并没有配置 `defaultProps`，所以如果直接用 `<Comment />` 而不传入任何参数的话，`comment` 就会是 `undefined`，`comment.username` 会导致程序报错：

```
✖ ▶ Uncaught TypeError: Cannot read property 'username' of undefined
  at Comment.render (index.js:108)
  at ReactCompositeComponent.js:796
  at measureLifeCyclePerf (ReactCompositeComponent.js:75)
  at ReactCompositeComponentWrapper._renderValidatedComponentWithoutOwner
  at ReactCompositeComponentWrapper._renderValidatedComponent (ReactCompo
  at ReactCompositeComponentWrapper.performInitialMount (ReactCompositeCo
```

这个出错信息并不够友好。我们可以通过 `isRequired` 关键字来强制组件某个参数必须传入：

```
...
static propTypes = {
  comment: PropTypes.object.isRequired
}
...
```

那么会获得一个更加友好的出错信息，查错会更方便：

```
✖ ▶ Warning: Failed prop type: The prop `comment` is marked as required in `Comment`, but its value is `undefined`.
  in Comment (at index.js:376)
```

React.js 提供的 `PropTypes` 提供了一系列的数据类型可以用来配置组件的参数：

```
PropTypes.array
PropTypes.bool
PropTypes.func
PropTypes.number
PropTypes.object
PropTypes.string
PropTypes.node
PropTypes.element
...
```

更多类型及其用法可以参看官方文档：[Typechecking With PropTypes – React](#)。

组件参数验证在构建大型的组件库的时候相当有用，可以帮助我们迅速定位这种类型错误，让我们组件开发更加规范。另外也起到了一个说明文档的作用，如果大家都约定都写 `propTypes`，那你在使用别人写的组件的时候，只要看到组件的 `propTypes` 就清晰地知道这个组件到底能够接受什么参数，什么参数是可选的，什么参数是必选的。

## 总结

通过 `PropTypes` 给组件的参数做类型限制，可以在帮助我们迅速定位错误，这在构建大型应用程序的时候特别有用；另外，给组件加上 `propTypes`，也让组件的开发、使用更加规范清晰。

这里建议大家写组件的时候尽量都写 `propTypes`，有时候有点麻烦，但是是值得的。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：25. 实战分析：评论功能（四）](#)

[上一节：23. dangerouslySetHTML 和 style 属性](#)

React.js 小书

[← 返回首页](#)

## 25. 实战分析：评论功能（四）

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson25>
- 转载请注明出处，保留原文链接和作者信息。

(本文未审核)

目前为止，第二阶段知识已经基本介绍完，我们已经具备了项目上手实战必备的 React.js 知识，现在可以把这些知识应用起来。接下来是实战环节，我们会继续上一阶段的例子，把评论功能做得更加复杂一点。

我们在上一阶段的评论功能基础上加上以下功能需求：

1. 页面加载完成自动聚焦到评论输入框。
2. 把用户名持久化，存放到浏览器的 LocalStorage 中。页面加载时会把用户名加载出来显示到输入框，用户就不需要重新输入用户名了。
3. 把已经发布的评论持久化，存放到浏览器的 LocalStorage 中。页面加载时会把已经保存的评论加载出来，显示到页面的评论列表上。
4. 评论显示发布日期，如“1 秒前”，“30 分钟前”，并且会每隔 5 秒更新发布日期。
5. 评论可以被删除。
6. 类似 Markdown 的行内代码块显示功能，用户输入的用 `` 包含起来的内容都会被处理成用 `<code>` 元素包含。例如输入 `console.log` 就会处理成 `<code>console.log</code>` 再显示到页面上。

The screenshot shows a comment submission form at the top and a list of comments below it.

**Comment Submission Form:**

- 用户名: Tomy
- 评论内容: 所有 `export default class` 都要写类名字。
- 发布 (Post) button

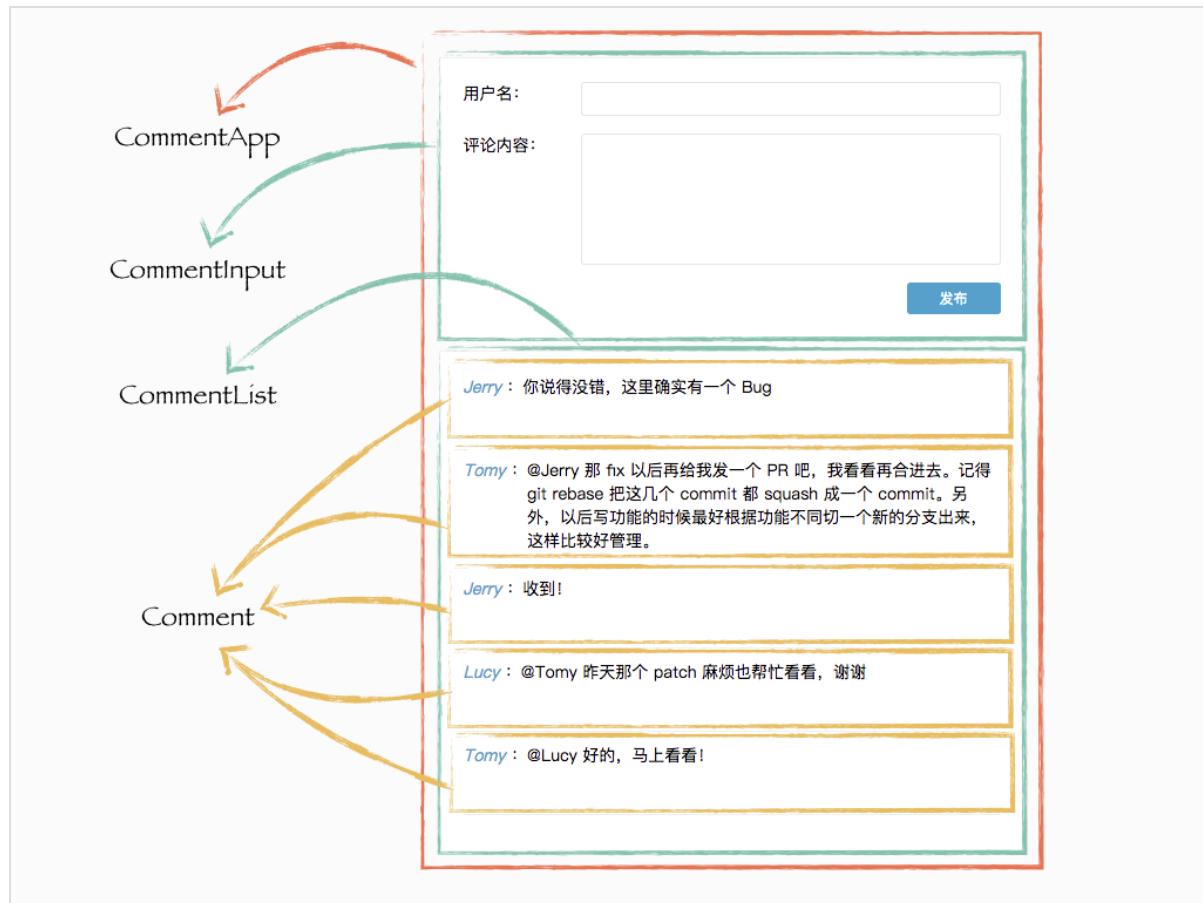
**Comment Feed:**

- Tomy:** 所有的 `console.log` 都可以删掉  
24 分钟前
- Jerry:** 收到!  
24 分钟前
- Lucy:** 好的!  
6 秒前

### 在线演示地址。

大家可以在原来的第一阶段代码的基础上进行修改，第一、二阶段评论功能代码可以在这里找到：[react-naive-book-examples](#)。可以直接使用最新的样式文件 [index.css](#) 覆盖原来的 index.css。

接下来可以分析如何利用第二阶段的知识来构建这些功能，在这个过程里面可能会穿插一些小技巧，希望对大家有用。我们回顾一下这个页面的组成：



我们之前把页面分成了四种不同的组件：分别是 `CommentApp`、`CommentInput`、`CommentList`、`Comment`。我们开始修改这个组件，把上面的需求逐个完成。

## 自动聚焦到评论框

这个功能是很简单的，我们需要获取 `textarea` 的 DOM 元素然后调用 `focus()` API 就可以了。我们给输入框元素加上 `ref` 以便获取到 DOM 元素，修改 `src/CommentInput.js` 文件：

```
...
<textarea
  ref={(textarea) => this.textarea = textarea}
  value={this.state.content}
  onChange={this.handleContentChange.bind(this)} />
...
```

组件挂载完以后完成以后就可以调用 `this.textarea.focus()`，给 `CommentInput` 组件加上 `ComponentDidMount` 生命周期：

```
class CommentInput extends Component {
  static propTypes = {
    onSubmit: PropTypes.func
  }
}
```

```

constructor () {
  super()
  this.state = {
    username: '',
    content: ''
  }
}

componentDidMount () {
  this.textarea.focus()
}
...

```

这个功能就完成了。现在体验还不是很好，接下来我们把用户名持久化一下，体验就会好很多。

大家可以注意到我们给原来的 `props.onSubmit` 参数加了组件参数验证，在这次实战案例中，我们都会给评论功能的组件加上 `propTypes` 进行参数验证，接下来就不累述。

## 持久化用户名

用户输入用户名，然后我们把用户名保存到浏览器的 `LocalStorage` 当中，当页面加载的时候再从 `LocalStorage` 把之前保存的用户名显示到用户名输入框当中。这样用户就不用每次都输入用户名了，并且评论框是自动聚焦的，用户的输入体验就好很多。

我们监听用户名输入框失去焦点的事件 `onBlur`：

```

...
<input
  value={this.state.username}
  onBlur={this.handleUsernameBlur.bind(this)}
  onChange={this.handleUsernameChange.bind(this)} />
...

```

在 `handleUsernameBlur` 中我们把用户的输入内容保存到 `LocalStorage` 当中：

```

class CommentInput extends Component {
  constructor () {
    super()
    this.state = {
      username: '',
      content: ''
    }
  }
}

```

```

componentDidMount () {
  this.textarea.focus()
}

_saveUsername (username) {
  localStorage.setItem('username', username)
}

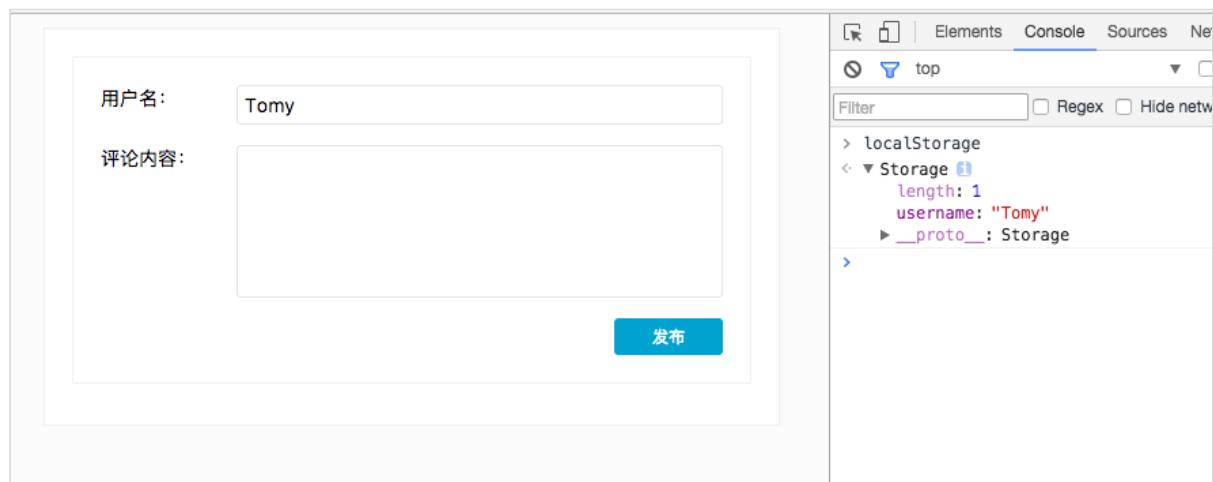
handleUsernameBlur (event) {
  this._saveUsername(event.target.value)
}

...

```

在 `handleUsernameBlur` 中我们把用户输入的内容传给了 `_saveUsername` 私有方法（所有私有方法都以 `_` 开头）。`_saveUsername` 会设置 `LocalStorage` 中的 `username` 字段，用户名就持久化了。这样就相当于每当用户输入完用户名以后（输入框失去焦点的时候），都会把用户名自动保存一次。

输入用户名，然后到浏览器里里面看看是否保存了：



然后我们组件挂载的时候把用户名加载出来。这是一种数据加载操作，我们说过，不依赖 DOM 操作的组件启动的操作都可以放在 `componentWillMount` 中进行，所以给 `CommentInput` 添加 `componentWillMount` 的组件生命周期：

```

...
componentWillMount () {
  this._loadUsername()
}

_loadUsername () {
  const username = localStorage.getItem('username')
  if (username) {
    this.setState({ username })
  }
}

```

```

_saveUsername (username) {
  localStorage.setItem('username', username)
}
...

```

`componentWillMount` 会调用 `_loadUsername` 私有方法，`_loadUsername` 会从 `LocalStorage` 加载用户名并且 `setState` 到组件的 `state.username` 中。那么组件在渲染的时候 (`render` 方法) 挂载的时候就可以用上用户名了。

这样体验就好多了，刷新页面，不需要输入用户名，并且自动聚焦到了输入框。我们 1、2 需求都已经完成。

## 小贴士

这里插入一些小贴示，大家可以注意到我们组件的命名和方法的摆放顺序其实有一定的讲究，这里可以简单分享一下个人的习惯，仅供参考。

组件的私有方法都用 `_` 开头，所有事件监听的方法都用 `handle` 开头。把事件监听方法传给组件的时候，属性名用 `on` 开头。例如：

```
<CommentInput
  onSubmit={this.handleSubmitComment.bind(this)} />
```

这样统一规范处理事件命名会给我们带来语义化组件的好处，监听 (`on`) `CommentInput` 的 `Submit` 事件，并且交给 `this` 去处理 (`handle`)。这种规范在多人协作的时候也会非常方便。

另外，组件的内容编写顺序如下：

1. `static` 开头的类属性，如 `defaultProps`、`propTypes`。
2. 构造函数，`constructor`。
3. `getter/setter` (还不了解的同学可以暂时忽略)。
4. 组件生命周期。
5. `_` 开头的私有方法。
6. 事件监听方法，`handle*`。
7. `render*` 开头的方法，有时候 `render()` 方法里面的内容会分开到不同函数里面进行，这些函数都以 `render*` 开头。
8. `render()` 方法。

如果所有的组件都按这种顺序来编写，那么维护起来就会方便很多，多人协作的时候别人理解代码也会一目了然。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

---

[下一节：26. 实战分析：评论功能（五）](#)

[上一节：24. PropTypes 和组件参数验证](#)

React.js 小书

[<-- 返回首页](#)

## 26. 实战分析：评论功能（五）

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson26>
- 转载请注明出处，保留原文链接和作者信息。

(本文未审核)

### 持久化评论

同样地，可以通过类似于用户名持久化的方式对评论列表内容进行持久化，让用户发布的评论在刷新页面以后依然可以存在。修改 `src/CommentApp.js`：

```
class CommentApp extends Component {  
  constructor () {  
    super()  
    this.state = {  
      comments: []  
    }  
  }  
  
  componentWillMount () {  
    this._loadComments()  
  }  
  
  _loadComments () {  
    let comments = localStorage.getItem('comments')  
    if (comments) {  
      comments = JSON.parse(comments)  
      this.setState({ comments })  
    }  
  }  
  
  _saveComments (comments) {  
    localStorage.setItem('comments', JSON.stringify(comments))  
  }  
  
  handleSubmitComment (comment) {  
    if (!comment) return  
    if (!comment.username) return alert('请输入用户名')  
    if (!comment.content) return alert('请输入评论内容')  
    const comments = this.state.comments  
    comments.push(comment)
```

```

    this.setState({ comments })
    this._saveComments(comments)
}
...

```

我们增加了 `_loadComments` 和 `_saveComments` 分别用于加载和保存评论列表数据。用户每次提交评论都会把评论列表数据保存一次，所以我们在 `handleSubmitComment` 调用 `_saveComments` 方法；而在 `componentWillMount` 中调用 `_loadComments` 方法，在组件开始挂载的时候把评论列表数据加载出来 `setState` 到 `this.state` 当中，组件就可以渲染从 `LocalStorage` 从加载出来的评论列表数据了。

现在发布评论，然后刷新可以看到我们的评论并不会像以前一样消失。非常的不错，持久化评论的功能也完成了。

## 显示评论发布时间

现在我们给每条评论都加上发布的日期，并且在评论列表项上显示已经发表了多久，例如“1 秒前”、“30分钟前”，并且会每隔 5 秒进行更新。修改 `src/CommentInput.js` 当用户点击发布按钮的时候，传出去的评论数据带上评论发布的时间戳：

```

...
handleSubmit () {
  if (this.props.onSubmit) {
    this.props.onSubmit({
      username: this.state.username,
      content: this.state.content,
      createdTime: +new Date()
    })
  }
  this.setState({ content: '' })
}
...

```

在评论列表项上显示评论，修改 `src/comment.js`：

```

class Comment extends Component {
  static propTypes = {
    comment: PropTypes.object.isRequired
  }

  constructor () {
    super()
    this.state = { timeString: '' }
  }

  componentWillMount () {
    this._updateTimeString()
  }
}

```

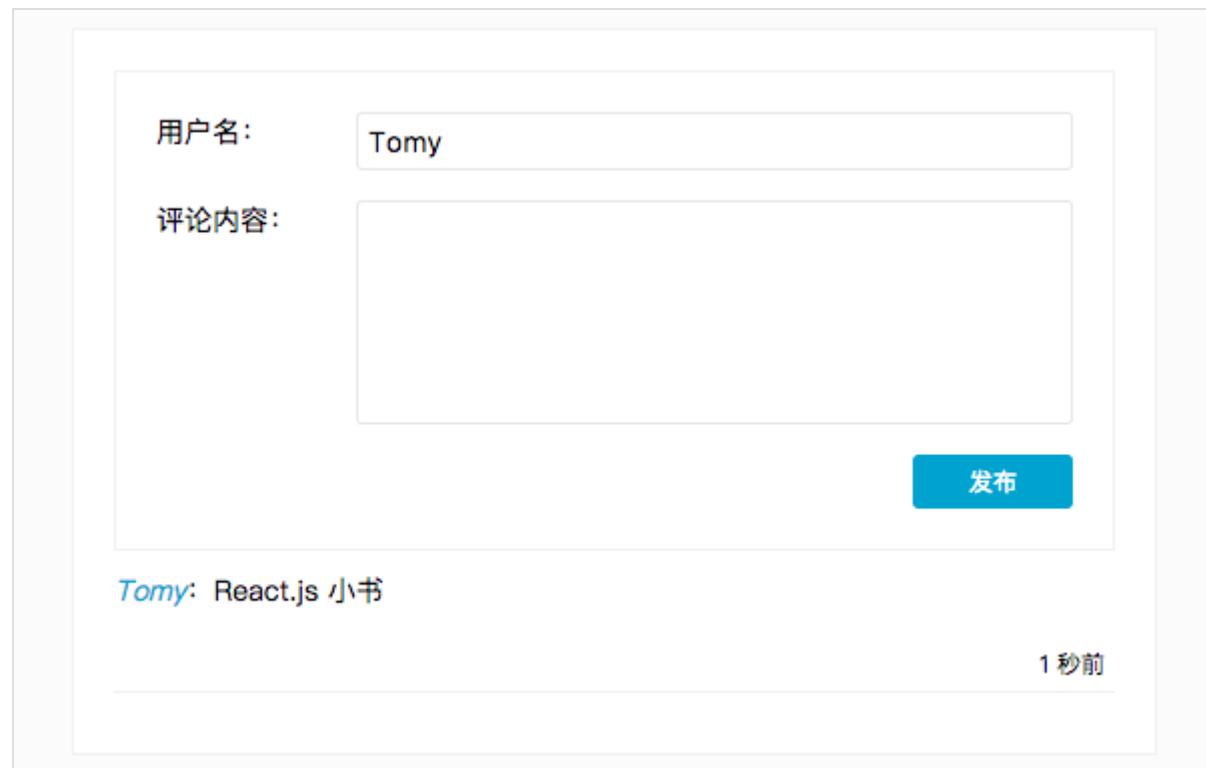
```
}

_updateTimeString () {
  const comment = this.props.comment
  const duration = (+Date.now() - comment.createdTime) / 1000
  this.setState({
    timeString: duration > 60
      ? `${Math.round(duration / 60)} 分钟前`
      : `${Math.round(Math.max(duration, 1))} 秒前`
  })
}

render () {
  return (
    <div className='comment'>
      <div className='comment-user'>
        <span>{this.props.comment.username}</span>
      </div>
      <p>{this.props.comment.content}</p>
      <span className='comment-createdtime'>
        {this.state.timeString}
      </span>
    </div>
  )
}
}
```

每个 `Comment` 组件实例会保存一个 `timeString` 状态，用于该评论显示发布了多久。`_updateTimeString` 这个私有方法会根据 `props.comment` 里面的 `createdTime` 来更新这个 `timeString`：计算当前时间和评论发布时间的时间差，如果已经发布 60 秒以上就显示分钟，否则就显示秒。然后 `componentWillMount` 会在组件挂载阶段调用 `_updateTimeString` 更新一下这个字符串，`render()` 方法就把这个显示时间差的字符串渲染到一个 `<span>` 上。

再看看页面显示：



这时候的时间是不会自动更新的。除非你手动刷新页面，否则永远显示“1 秒前”。我们可以把 `componentWillMount` 中启动一个定时器，每隔 5 秒调用一下 `_updateTimeString`，让它去通过 `setState` 更新 `timeString`：

```
...
componentWillMount () {
  this._updateTimeString()
  this._timer = setInterval(
    this._updateTimeString.bind(this),
    5000
  )
}
...
```

这样就可以做到评论的发布时间自动刷新了，到这里前 4 个需求都已经完成了。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

下一节：[27. 实战分析：评论功能（六）](#)

上一节：[25. 实战分析：评论功能（四）](#)



React.js 小书

[<-- 返回首页](#)

## 27. 实战分析：评论功能（六）

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson27>
- 转载请注明出处，保留原文链接和作者信息。

(本文未审核)

### 删除评论

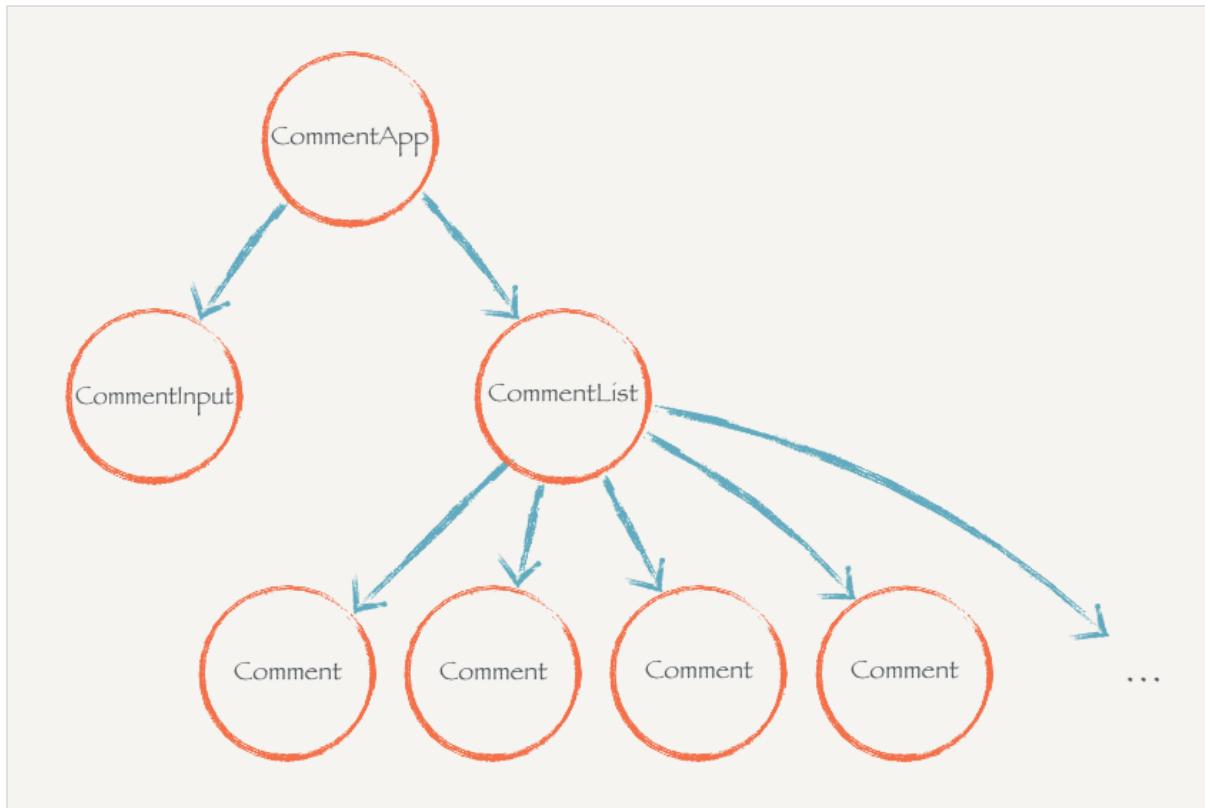
现在发布评论，评论不会消失，评论越来越多并不是什么好事。所以我们给评论组件加上删除评论的功能，这样就可以删除不想要的评论了。修改 `src/Comment.js` 的 `render` 方法，新增一个删除按钮：

```
...
render () {
  const { comment } = this.props
  return (
    <div className='comment'>
      <div className='comment-user'>
        <span className='comment-username'>
          {comment.username}
        </span>:
      </div>
      <p>{comment.content}</p>
      <span className='comment-createdtime'>
        {this.state.timeString}
      </span>
      <span className='comment-delete'>
        删除
      </span>
    </div>
  )
}
...
```

我们在后面加了一个删除按钮，因为 `index.css` 定义了样式，所以鼠标放到特定的评论上才会显示删除按钮，让用户体验好一些。

我们知道评论列表数据是放在 `CommentApp` 当中的，而这个删除按钮是在 `Comment` 当中的，现在我们要做的事情是用户点击某条评论的删除按钮，然后在 `CommentApp` 中

把相应的数据删除。但是 `CommentApp` 和 `Comment` 的关系是这样的：



`Comment` 和 `CommentApp` 之间隔了一个 `CommentList`，`Comment` 无法直接跟 `CommentApp` 打交道，只能通过 `CommentList` 来转发这种删除评论的消息。修改 `Comment` 组件，让它可以把删除的消息传递到上一层：

```

class Comment extends Component {
  static propTypes = {
    comment: PropTypes.object.isRequired,
    onDeleteComment: PropTypes.func,
    index: PropTypes.number
  }
  ...
  handleDeleteComment () {
    if (this.props.onDeleteComment) {
      this.props.onDeleteComment(this.props.index)
    }
  }
  render () {
    ...
    <span
      onClick={this.handleDeleteComment.bind(this)}
      className='comment-delete'>
      删除
    </span>
  </div>
}
  
```

```
}
```

现在在使用 `Comment` 的时候，可以传入 `onDeleteComment` 和 `index` 两个参数。

`index` 用来标志这个评论在列表的下标，这样点击删除按钮的时候我们才能知道你点击的是哪个评论，才能知道怎么从列表数据中删除。用户点击删除会调用 `handleDeleteComment`，它会调用从上层传入的 `props.onDeleteComment` 函数告知上一层组件删除的消息，并且把评论下标传出去。现在修改 `src/CommentList.js` 让它把这两个参数传进来：

```
class CommentList extends Component {
  static propTypes = {
    comments: PropTypes.array,
    onDeleteComment: PropTypes.func
  }

  static defaultProps = {
    comments: []
  }

  handleDeleteComment (index) {
    if (this.props.onDeleteComment) {
      this.props.onDeleteComment(index)
    }
  }

  render() {
    return (
      <div>
        {this.props.comments.map((comment, i) =>
          <Comment
            comment={comment}
            key={i}
            index={i}
            onDeleteComment={this.handleDeleteComment.bind(this)} />
        )}
      </div>
    )
  }
}
```

当用户点击按钮的时候，`Comment` 组件会调用 `props.onDeleteComment`，也就是 `CommentList` 的 `handleDeleteComment` 方法。而 `handleDeleteComment` 会调用 `CommentList` 所接受的配置参数中的 `props.onDeleteComment`，并且把下标传出去。

也就是说，我们可以在 `CommentApp` 给 `CommentList` 传入一个 `onDeleteComment` 的配置参数来接受这个删除评论的消息，修改 `CommentApp.js`：

```

...
    handleDeleteComment (index) {
      console.log(index)
    }

    render() {
      return (
        <div className='wrapper'>
          <CommentInput onSubmit={this.handleSubmitComment.bind(this)} />
          <CommentList
            comments={this.state.comments}
            onDeleteComment={this.handleDeleteComment.bind(this)} />
        </div>
      )
    }
  }
...

```

现在点击删除按钮，可以在控制台看到评论对应的下标打印了出来。其实这就是这么一个过程：`CommentList` 把下标 `index` 传给 `Comment`。点击删除按钮的时候，`Comment` 把 `index` 传给了 `CommentList`，`CommentList` 再把它传给 `CommentApp`。现在可以在 `CommentApp` 中删除评论了：

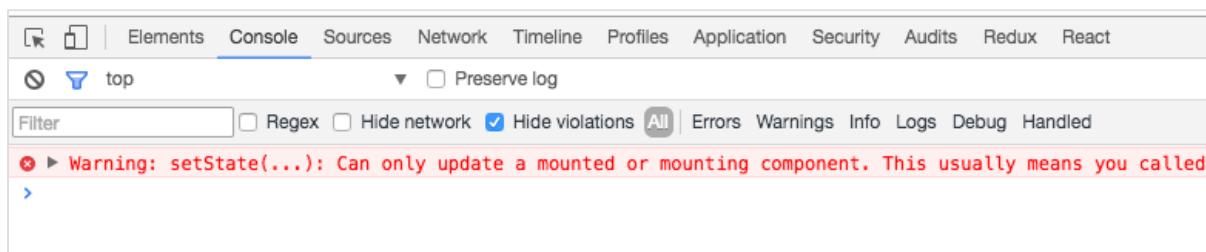
```

...
  handleDeleteComment (index) {
    const comments = this.state.comments
    comments.splice(index, 1)
    this.setState({ comments })
    this._saveComments(comments)
  }
...

```

我们通过 `comments.splice` 删除特定下标的评论，并且通过 `setState` 重新渲染整个评论列表；当然了，还需要把最新的评论列表数据更新到 `LocalStorage` 中，所以我们在删除、更新以后调用了 `_saveComments` 方法把数据同步到 `LocalStorage` 中。

现在就可以愉快地删除评论了。但是，你删除评论以后 5 秒钟后就会在控制台中看到报错了：



这是因为我们忘了清除评论的定时器，修改 `src/Comment.js`，新增生命周期

`commentWillUnmount` 在评论组件销毁的时候清除定时器：

```
...
componentWillUnmount () {
  clearInterval(this._timer)
}
...
```

这才算完成了第 5 个需求。

## 显示代码块

用户在的输入内容中任何以 `` 包含的内容都会用 `<code>` 包含起来显示到页面上。

`<code>` 这是一个 HTML 结构，需要往页面动态插入 HTML 结构我们只能用

`dangerouslySetInnerHTML` 了，修改 `src/Comment.js`，把原来 `render()` 函数中的：

```
<p>{comment.content}</p>
```

修改成：

```
<p dangerouslySetInnerHTML={{  
  __html: this._getProcessedContent(comment.content)  
}} />
```

我们把经过 `this._getProcessedContent` 处理的评论内容以 HTML 的方式插入到 `<p>` 元素中。`this._getProcessedContent` 要把 `` 包含的内容用 `<code>` 包裹起来，一个正则表达式就可以做到了：

```
...
_getProcessedContent (content) {
  return content
  .replace(/\`([\S\s]+?)\`/g, '<code>$1</code>')
}
...
```

但是这样做会有严重的 XSS 漏洞，用户可以输入任意的 HTML 标签，用 `<script>` 执行任意的 JavaScript 代码。所以在替换代码之前，我们要手动地把这些 HTML 标签进行转义：

```
...
_getProcessedContent (content) {
  return content
  .replace(/\`([\S\s]+?)\`/g, '<code>$1</code>')
}
```

```
.replace(/&/g, "&amp;")
.replace(/</g, "&lt;")
.replace(/>/g, "&gt;")
.replace(/"/g, "&quot;")
.replace(/\'/g, "&#039;")
.replace(`([\S\s]+?)`/g, '<code>$1</code>')
}
```

...

前 5 个 `replace` 实际上是把类似于 `<`、`>` 这种内容替换转义一下，防止用户输入 HTML 标签。最后一行代码才是实现功能的代码。

这时候在评论框中输入：

这是代码块 `console.log`，这是 `<h1>正常内容</h1>`。

然后点击发布，看看效果：



Tomy: 这是代码块 `console.log`，这是 `<h1>正常内容</h1>`。

1 秒前

我们安全地完成了第 6 个需求。到目前为止，第二阶段的实战已经全部完成，你可以[在这里](#)找到完整的代码。

## 总结

到这里第二阶段已经全部结束，我们已经掌握了全部 React.js 实战需要的入门知识。接下来我们会学习两个相对比较高级的 React.js 的概念，然后进入 React-redux 的世界，让它们配合 React.js 来构建更成熟的前端页面。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：28. 高阶组件（Higher-Order Components）](#)

[上一节：26. 实战分析：评论功能（五）](#)

React.js 小书

[← 返回首页](#)

## 28. 高阶组件 (Higher-Order Components)

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson28>
- 转载请注明出处，保留原文链接和作者信息。

(本文未审核)

有时候人们很喜欢造一些名字很吓人的名词，让人一听这个名词就觉得自己不可能学会，从而让人望而却步。但是其实这些名词背后所代表的东西其实很简单。

我不能说高阶组件就是这么一个东西。但是它是一个概念上很简单，但却非常常用、实用的东西，被大量 React.js 相关的第三方库频繁地使用。在前端的业务开发当中，你不掌握高阶组件其实也可以完成项目的开发，但是如果你能够灵活地使用高阶组件，可以让你代码更加优雅，复用性、灵活性更强。它是一个加分项，而且加的分还不少。

本章节可能有部分内容理解起来会有难度，如果你觉得无法完全理解本节内容。可以先简单理解高阶组件的概念和作用即可，其他内容选择性地跳过。

了解高阶组件对我们理解各种 React.js 第三方库的原理很有帮助。

### 什么是高阶组件

高阶组件就是一个函数，传给它一个组件，它返回一个新的组件。

```
const NewComponent = higherOrderComponent(OldComponent)
```

重要的事情再重复一次，高阶组件是一个函数（而不是组件），它接受一个组件作为参数，返回一个新的组件。这个新的组件会使用你传给它的组件作为子组件，我们看看一个很简单的高阶组件：

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

export default (WrappedComponent) => {
  class NewComponent extends Component {
    // 可以做很多自定义逻辑
    render () {
      return <WrappedComponent />
    }
}
```

```

    }
    return NewComponent
}

```

现在看来好像什么用都没有，它就是简单的构建了一个新的组件类 `NewComponent`，然后把传进去的 `WrappedComponent` 渲染出来。但是我们可以给 `NewComponent` 做一些数据启动工作：

```

import React, { Component } from 'react'

export default (WrappedComponent, name) => {
  class NewComponent extends Component {
    constructor () {
      super()
      this.state = { data: null }
    }

    componentWillMount () {
      let data = localStorage.getItem(name)
      this.setState({ data })
    }

    render () {
      return <WrappedComponent data={this.state.data} />
    }
  }
  return NewComponent
}

```

现在 `NewComponent` 会根据第二个参数 `name` 在挂载阶段从 `LocalStorage` 加载数据，并且 `setState` 到自己的 `state.data` 中，而渲染的时候将 `state.data` 通过 `props.data` 传给 `WrappedComponent`。

这个高阶组件有什么用呢？假设上面的代码是在 `src/wrapWithData.js` 文件中的，我们可以在别的地方这么用它：

```

import wrapWithData from './wrapWithData'

class InputWithUserName extends Component {
  render () {
    return <input value={this.props.data} />
  }
}

InputWithUserName = wrapWithData(InputWithUserName, 'username')
export default InputWithUserName

```

假如 `InputWithUserName` 的功能需求是挂载的时候从 `LocalStorage` 里面加载 `username` 字段作为 `<input />` 的 `value` 值，现在有了 `wrapWithData`，我们可以很容易地做到这件事情。

只需要定义一个非常简单的 `InputWithUserName`，它会把 `props.data` 作为 `<input />` 的 `value` 值。然把这个组件和 `'username'` 传给 `wrapWithData`，`wrapWithData` 会返回一个新的组件，我们用这个新的组件覆盖原来的 `InputWithUserName`，然后再导出去模块。

别人用这个组件的时候实际是用了被加工过的组件：

```
import InputWithUserName from './InputWithUserName'

class Index extends Component {
  render () {
    return (
      <div>
        用户名: <InputWithUserName />
      </div>
    )
  }
}
```

根据 `wrapWithData` 的代码我们可以知道，这个新的组件挂载的时候会先去 `LocalStorage` 加载数据，渲染的时候再通过 `props.data` 传给真正的 `InputWithUserName`。

如果现在我们需要另外一个文本输入框组件，它也需要 `LocalStorage` 加载 `'content'` 字段的数据。我们只需要定义一个新的 `TextareaWithContent`：

```
import wrapWithData from './wrapWithData'

class TextareaWithContent extends Component {
  render () {
    return <textarea value={this.props.data} />
  }
}

TextareaWithContent = wrapWithData(TextareaWithContent, 'content')
export default TextareaWithContent
```

写起来非常轻松，我们根本不需要重复写从 `LocalStorage` 加载数据字段的逻辑，直接用 `wrapWithData` 包装一下就可以了。

我们来回顾一下到底发生了什么事情，对于 `InputWithUserName` 和 `TextareaWithContent` 这两个组件来说，它们的需求有着这么一个相同的逻辑：“挂载

阶段从 `LocalStorage` 中加载特定字段数据”。

如果按照之前的做法，我们需要给它们两个都加上 `componentWillMount` 生命周期，然后在里面调用 `LocalStorage`。要是有第三个组件也有这样的加载逻辑，我又得写一遍这样的逻辑。但有了 `wrapWithData` 高阶组件，我们把这样的逻辑用一个组件包裹了起来，并且通过给高阶组件传入 `name` 来达到不同字段的数据加载。充分复用了逻辑代码。

到这里，高阶组件的作用其实不言而喻，**其实就是为了组件之间的代码复用**。组件可能有着某些相同的逻辑，把这些逻辑抽离出来，放到高阶组件中进行复用。**高阶组件内部的包装组件和被包装组件之间通过 `props` 传递数据**。

## 高阶组件的灵活性

代码复用的方法、形式有很多种，你可以用类继承来做到代码复用，也可以分离模块的方式。但是高阶组件这种方式很有意思，也很灵活。学过设计模式的同学其实应该能反应过来，它其实就是设计模式里面的装饰者模式。它通过组合的方式达到很高的灵活程度。

假设现在我们需求变化了，现在要的是通过 `Ajax` 加载数据而不是从 `LocalStorage` 加载数据。我们只需要新建一个 `wrapWithAjaxData` 高阶组件：

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

export default (WrappedComponent, name) => {
  class NewComponent extends Component {
    constructor () {
      super()
      this.state = { data: null }
    }

    componentWillMount () {
      ajax.get('/data/' + name, (data) => {
        this.setState({ data })
      })
    }

    render () {
      return <WrappedComponent data={this.state.data} />
    }
  }
  return NewComponent
}
```

其实就改了一下 `wrapWithData` 的 `componentWillMount` 中的逻辑，改成了从服务器加载数据。现在只需要把 `InputWithUserName` 稍微改一下：

```

import wrapWithAjaxData from './wrapWithAjaxData'

class InputWithUserName extends Component {
  render () {
    return <input value={this.props.data} />
  }
}

InputWithUserName = wrapWithAjaxData(InputWithUserName, 'username')
export default InputWithUserName

```

只要改一下包装的高阶组件就可以达到需要的效果。而且我们并没有改动 `InputWithUserName` 组件内部的任何逻辑，也没有改动 `Index` 的任何逻辑，只是改动了中间的高阶组件函数。

(以下内容为选读内容，有兴趣的同学可以继续往下读，否则也可以直接跳到文末的总结部分。)

## 多层高阶组件（选读）

假如现在需求有变化了：我们需要先从 `LocalStorage` 中加载数据，再用这个数据去服务器取数据。我们改一下（或者新建一个）`wrapWithAjaxData` 高阶组件，修改其中的 `componentWillMount`：

```

...
componentWillMount () {
  ajax.get('/data/' + this.props.data, (data) => {
    this.setState({ data })
  })
}
...

```

它会用传进来的 `props.data` 去服务器取数据。这时候修改 `InputWithUserName`：

```

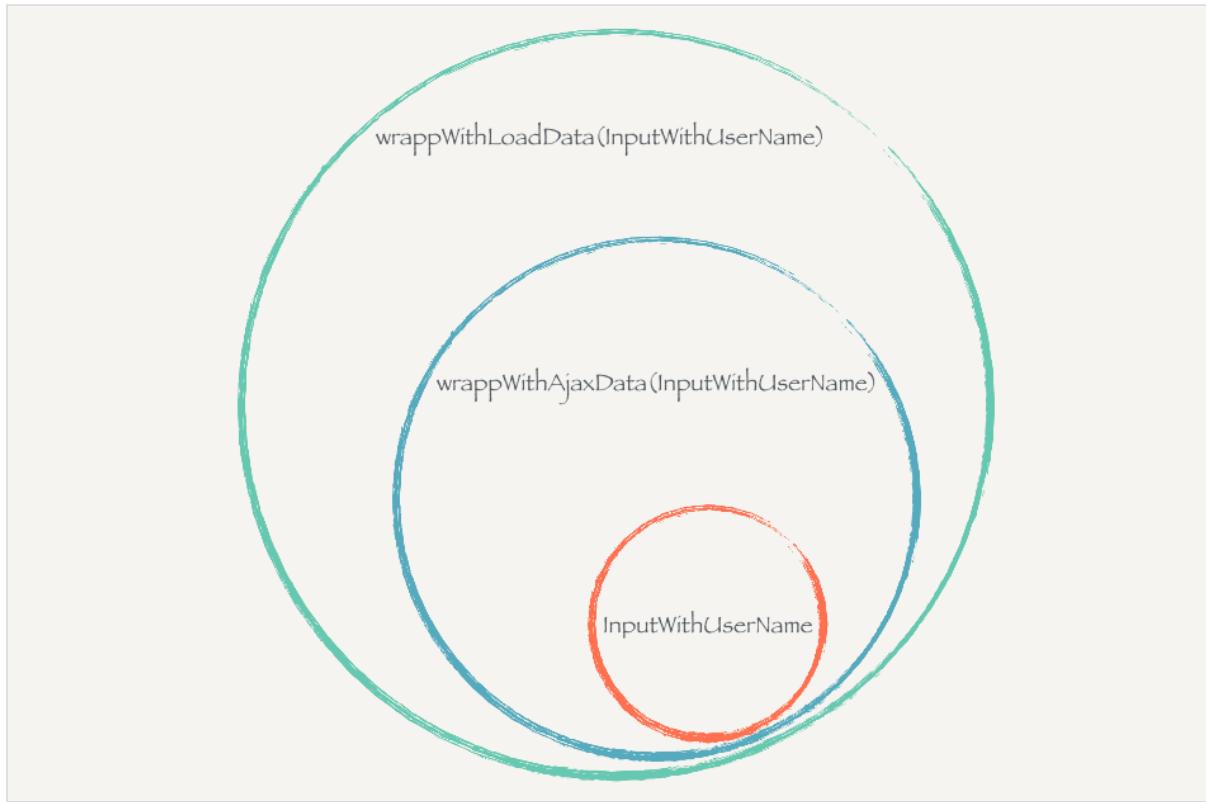
import wrapWithLoadData from './wrapWithLoadData'
import wrapWithAjaxData from './wrapWithAjaxData'

class InputWithUserName extends Component {
  render () {
    return <input value={this.props.data} />
  }
}

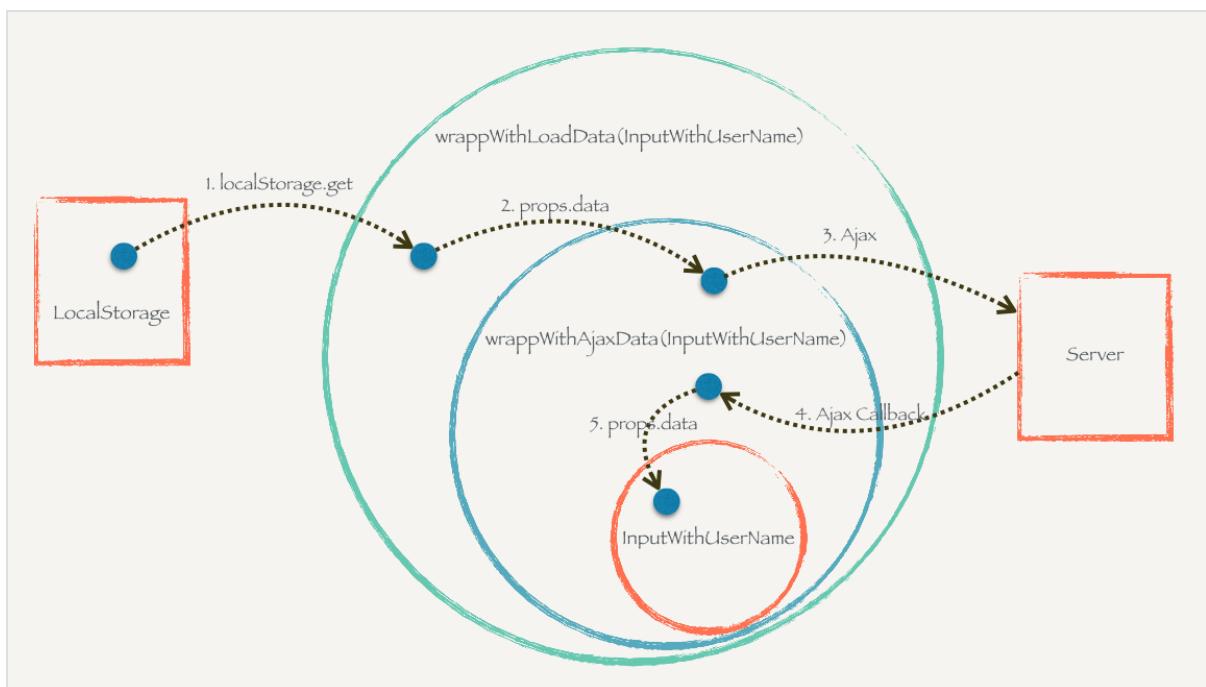
InputWithUserName = wrapWithAjaxData(InputWithUserName)
InputWithUserName = wrapWithLoadData(InputWithUserName, 'username')
export default InputWithUserName

```

大家可以看到，我们给 `InputWithUserName` 应用了两种高阶组件：先用 `wrapWithAjaxData` 包裹 `InputWithUserName`，再用 `wrapWithLoadData` 包含上次包裹的结果。它们的关系就如下图的三个圆圈：



实际上最终得到的组件会先去 `LocalStorage` 取数据，然后通过 `props.data` 传给下一层组件，下一层用这个 `props.data` 通过 Ajax 去服务端取数据，然后再通过 `props.data` 把数据传给下一层，也就是 `InputWithUserName`。大家可以体会一下下图尖头代表的数据流向：



## 用高阶组件改造评论功能（选读）

大家对这种在挂载阶段从 `LocalStorage` 加载数据的模式都很熟悉，在上一阶段的实战中，`CommentInput` 和 `CommentApp` 都用了这种方式加载、保存数据。实际上我们可以构建一个高阶组件把它们的相同的逻辑抽离出来，构建一个高阶组件

`wrapWithLoadData`：

```
export default (WrappedComponent, name) => {
  class LocalStorageActions extends Component {
    constructor () {
      super()
      this.state = { data: null }
    }

    componentWillMount () {
      let data = localStorage.getItem(name)
      try {
        // 尝试把它解析成 JSON 对象
        this.setState({ data: JSON.parse(data) })
      } catch (e) {
        // 如果出错了就当普通字符串读取
        this.setState({ data })
      }
    }

    saveData (data) {
      try {
        // 尝试把它解析成 JSON 字符串
        localStorage.setItem(name, JSON.stringify(data))
      } catch (e) {
        // 如果出错了就当普通字符串保存
        localStorage.setItem(name, `${data}`)
      }
    }

    render () {
      return (
        <WrappedComponent
          data={this.state.data}
          saveData={this.saveData.bind(this)}
          // 这里的意思是把其他的参数原封不动地传递给被包装的组件
          {...this.props} />
      )
    }
  }
  return LocalStorageActions
}
```

`CommentApp` 可以这样使用：

```

class CommentApp extends Component {
  static propTypes = {
    data: PropTypes.any,
    saveData: PropTypes.func.isRequired
  }

  constructor (props) {
    super(props)
    this.state = { comments: props.data }
  }

  handleSubmitComment (comment) {
    if (!comment) return
    if (!comment.username) return alert('请输入用户名')
    if (!comment.content) return alert('请输入评论内容')
    const comments = this.state.comments
    comments.push(comment)
    this.setState({ comments })
    this.props.saveData(comments)
  }

  handleDeleteComment (index) {
    const comments = this.state.comments
    comments.splice(index, 1)
    this.setState({ comments })
    this.props.saveData(comments)
  }

  render() {
    return (
      <div className='wrapper'>
        <CommentInput onSubmit={this.handleSubmitComment.bind(this)} />
        <CommentList
          comments={this.state.comments}
          onDeleteComment={this.handleDeleteComment.bind(this)} />
      </div>
    )
  }
}

CommentApp = wrapWithData(CommentApp, 'comments')
export default CommentApp

```

同样地可以在 `CommentInput` 中使用 `wrapWithData`，这里就不贴代码了。有兴趣的同学可以查看[高阶组件重构的 CommentApp 版本](#)。

## 总结

高阶组件就是一个函数，传给它一个组件，它返回一个新的组件。新的组件使用传入的组件作为子组件。

高阶组件的作用是用于代码复用，可以把组件之间可复用的代码、逻辑抽离到高阶组件当中。新的组件和传入的组件通过 `props` 传递信息。

高阶组件有助于提高我们代码的灵活性，逻辑的复用性。灵活和熟练地掌握高阶组件的用法需要经验的积累还有长时间的思考和练习，如果你觉得本章节的内容无法完全消化和掌握也没有关系，可以先简单了解高阶组件的定义、形式和作用即可。

## 课后练习

### \*[React.js 加载、刷新数据 - 高阶组件](#)

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：29. React.js 的 context](#)

[上一节：27. 实战分析：评论功能（六）](#)

React.js 小书

[<-- 返回首页](#)

## 29. React.js 的 context

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson29>
- 转载请注明出处，保留原文链接和作者信息。

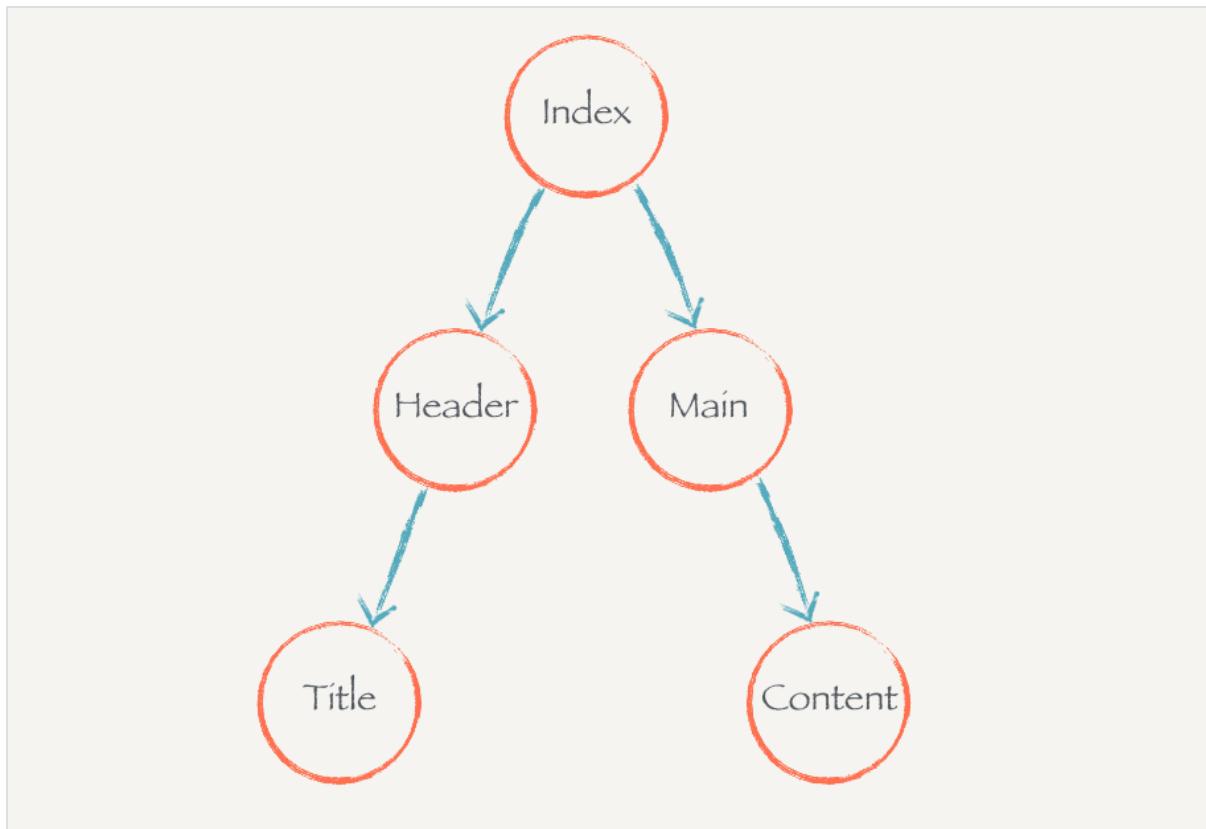
(本文未审核)

这一节我们来介绍一个你可能永远用不上的 React.js 特性 — context。但是了解它对于了解接下来要讲解的 React-redux 很有好处，所以大家可以简单了解一下它的概念和作用。

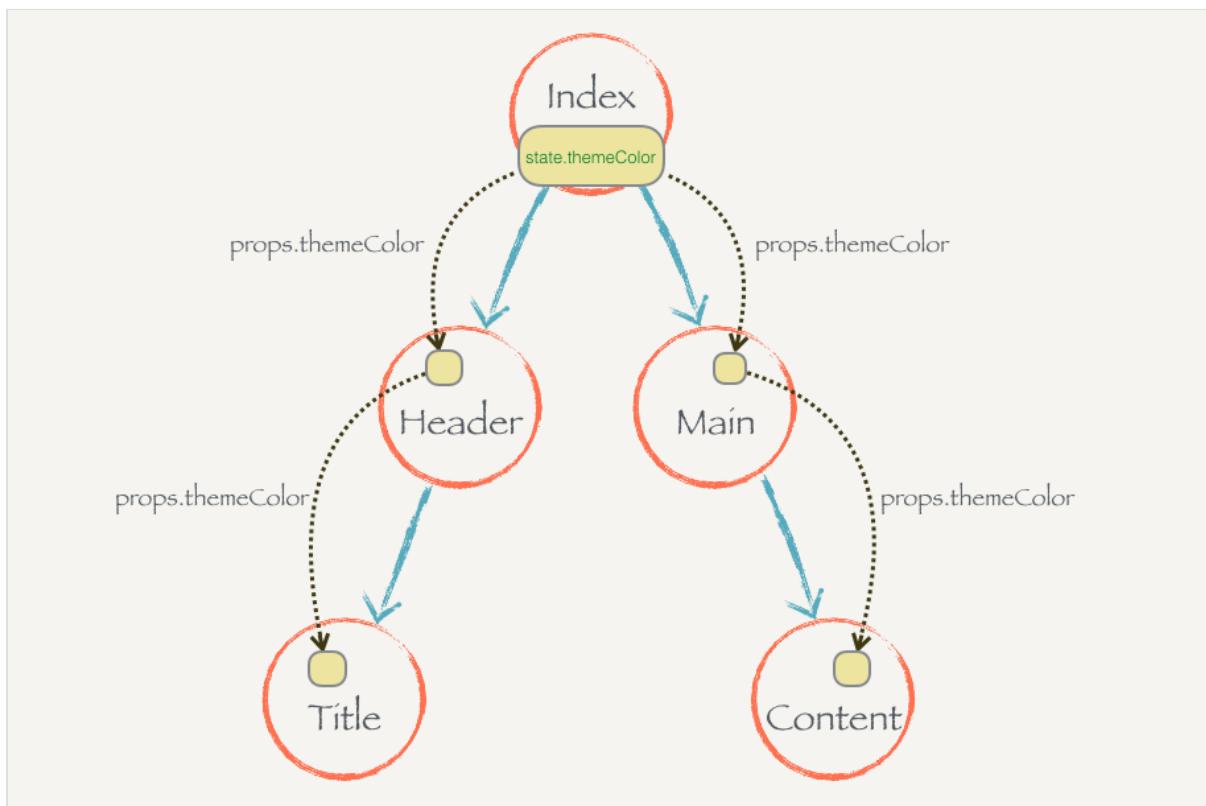
在过去很长一段时间里面，React.js 的 context 一直被视为一个不稳定的、危险的、可能会被去掉的特性而不被官网文档所记载。但是全世界的第三方库都在使用这个特性，直到了 React.js 的 v0.14.1 版本，context 才被官方文档所记录。

除非你觉得自己的 React.js 水平已经比较炉火纯青了，否则你永远不要使用 context。就像你学 JavaScript 的时候，总是会被提醒不要用全局变量一样，React.js 的 context 其实像就是组件树上某颗子树的全局变量。

想象一下我们有这么一棵组件树：



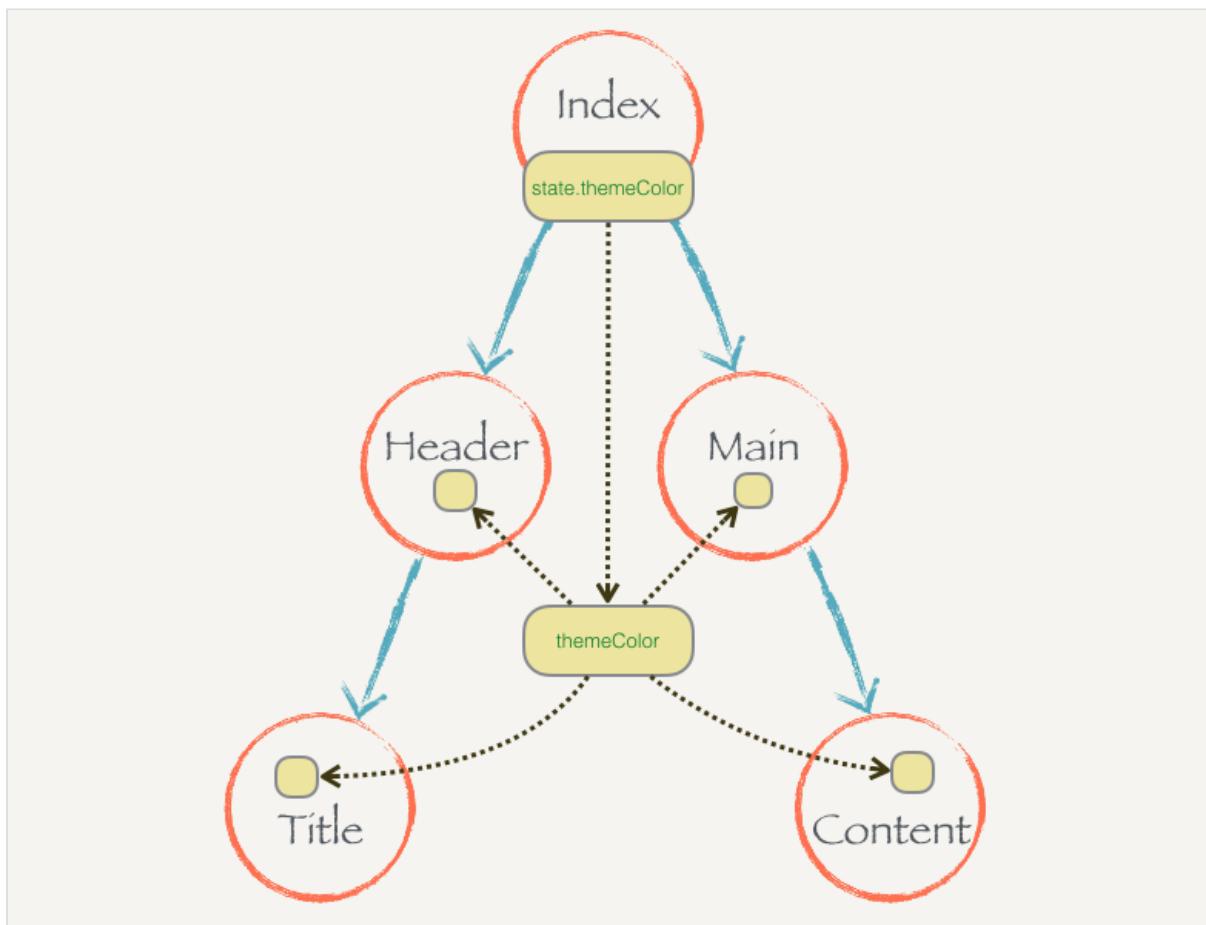
假设现在这个组件树代表的应用是用户可以自主换主题色的，每个子组件会根据主题色的不同调整自己的字体颜色或者背景颜色。“主题色”这个玩意是所有组件共享的状态，根据我们在 [前端应用状态管理 — 状态提升](#) 中所提到的，需要把这个状态提升到根节点的 `Index` 上，然后把这个状态通过 `props` 一层层传递下去：



假设原来主题色是绿色，那么 `Index` 上保存的就是 `this.state = { themeColor: 'green' }`。如果要改变主题色，可以直接通过 `this.setState({ themeColor: 'red' })` 来进行。这样整颗组件树就会重新渲染，子组件也就可以根据重新传进来的 `props.themeColor` 来调整自己的颜色。

但这里的问题也是非常明显的，我们需要把 `themeColor` 这个状态一层层手动地从组件树顶层往下传，每层都需要写 `props.themeColor`。如果我们的组件树很层次很深的话，这样维护起来简直是灾难。

如果这颗组件树能够全局共享这个状态就好了，我们要的时候就去取这个状态，不用手动地传：



就像这样，`Index` 把 `state.themeColor` 放到某个地方，这个地方是每个 `Index` 的子组件都可以访问到的。当某个子组件需要的时候就直接去那个地方拿就好了，而不需要一层层地通过 `props` 来获取。不管组件树的层次有多深，任何一个组件都可以直接到这个公共的地方提取 `themeColor` 状态。

React.js 的 `context` 就是这么一个东西，某个组件只要往自己的 `context` 里面放了某些状态，这个组件之下的所有子组件都直接访问这个状态而不需要通过中间组件的传递。一个组件的 `context` 只有它的子组件能够访问，它的父组件是不能访问到的，你可以理解每个组件的 `context` 就是瀑布的源头，只能往下流不能往上飞。

我们看看 React.js 的 context 代码怎么写，我们先把整体的组件树搭建起来，这里不涉及到 context 相关的内容：

```
class Index extends Component {
  render () {
    return (
      <div>
        <Header />
        <Main />
      </div>
    )
  }
}

class Header extends Component {
  render () {
    return (
      <div>
        <h2>This is header</h2>
        <Title />
      </div>
    )
  }
}

class Main extends Component {
  render () {
    return (
      <div>
        <h2>This is main</h2>
        <Content />
      </div>
    )
  }
}

class Title extends Component {
  render () {
    return (
      <h1>React.js 小书标题</h1>
    )
  }
}

class Content extends Component {
  render () {
    return (
      <div>
        <h2>React.js 小书内容</h2>
      </div>
    )
  }
}
```

```

        }
    }

ReactDOM.render(
    <Index />,
    document.getElementById('root')
)

```

代码很长但是很简单，这里就不解释了。

现在我们修改 `Index`，让它往自己的 `context` 里面放一个 `themeColor`：

```

class Index extends Component {
    static childContextTypes = {
        themeColor: PropTypes.string
    }

    constructor () {
        super()
        this.state = { themeColor: 'red' }
    }

    getChildContext () {
        return { themeColor: this.state.themeColor }
    }

    render () {
        return (
            <div>
                <Header />
                <Main />
            </div>
        )
    }
}

```

构造函数里面的内容其实就很好理解，就是往 `state` 里面初始化一个 `themeColor` 状态。`getChildContext` 这个方法就是设置 `context` 的过程，它返回的对象就是 `context`（也就是上图中处于中间的方块），所有的子组件都可以访问到这个对象。我们用 `this.state.themeColor` 来设置了 `context` 里面的 `themeColor`。

还有一个看起来很可怕的 `childContextTypes`，它的作用其实 `propTypes` 验证组件 `props` 参数的作用类似。不过它是验证 `getChildContext` 返回的对象。为什么要验证 `context`，因为 `context` 是一个危险的特性，按照 React.js 团队的想法就是，把危险的事情搞复杂一些，提高使用门槛人们就不会去用了。如果你要给组件设置 `context`，那么 `childContextTypes` 是必写的。

现在我们已经完成了 `Index` 往 `context` 里面放置状态的工作了，接下来我们要看看子组件怎么获取这个状态，修改 `Index` 的孙子组件 `Title`：

```
class Title extends Component {
  static contextTypes = {
    themeColor: PropTypes.string
  }

  render () {
    return (
      <h1 style={{ color: this.context.themeColor }}>React.js 小书标题</h1>
    )
  }
}
```

子组件要获取 `context` 里面的内容的话，就必须写 `contextTypes` 来声明和验证你需要获取的状态的类型，它也是必写的，如果你不写就无法获取 `context` 里面的状态。`Title` 想获取 `themeColor`，它是一个字符串，我们就在 `contextTypes` 里面进行声明。

声明以后我们就可以通过 `this.context.themeColor` 获取到在 `Index` 放置的值为 `red` 的 `themeColor`，然后设置 `h1` 的样式，所以你会看到页面上的字体是红色的：



如果我们要改颜色，只需要在 `Index` 里面 `setState` 就可以了，子组件会重新渲染，渲染的时候会重新取 `context` 的内容，例如我们给 `Index` 调整一下颜色：

```
...
componentWillMount () {
  this.setState({ themeColor: 'green' })
}
...
```

那么 `Title` 里面的字体就会显示绿色。我们可以如法炮制孙子组件 `Content`，或者任意的 `Index` 下面的子组件。让它们可以不经过中间 `props` 的传递就可以获取到由 `Index` 设定的 `context` 内容。

## 总结

一个组件可以通过 `getChildContext` 方法返回一个对象，这个对象就是子树的 `context`，提供 `context` 的组件必须提供 `childContextTypes` 作为 `context` 的声明和验证。

如果一个组件设置了 `context`，那么它的子组件都可以直接访问到里面的内容，它就像这个组件为根的子树的全局变量。任意深度的子组件都可以通过 `contextTypes` 来声明你想要的 `context` 里面的哪些状态，然后可以通过 `this.context` 访问到那些状态。

`context` 打破了组件和组件之间通过 `props` 传递数据的规范，极大地增强了组件之间的耦合性。而且，就如全局变量一样，**context 里面的数据能被随意接触就能被随意修改**，每个组件都能够改 `context` 里面的内容会导致程序的运行不可预料。

但是这种机制对于前端应用状态管理来说是很有帮助的，因为毕竟很多状态都会在组件之间进行共享，`context` 会给我们带来很大的方便。一些第三方的前端应用状态管理的库（例如 Redux）就是充分地利用了这种机制给我们提供便利的状态管理服务。但我们一般不需要手动写 `context`，也不要用它，只需要用好这些第三方的应用状态管理库就行了。

## 课后练习

### \*高阶组件 + context

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

下一节：30. 动手实现 Redux（一）：优雅地修改共享状态

上一节：28. 高阶组件（Higher-Order Components）

React.js 小书

[<-- 返回首页](#)

## 30. 动手实现 Redux (一) : 优雅地修改共享状态

- 作者: [胡子大哈](#)
- 原文链接: <http://huziketang.com/books/react/lesson30>
- 转载请注明出处, 保留原文链接和作者信息。

(本文未审核)

从这节起我们开始学习 Redux, 一种新型的前端“架构模式”。经常和 React.js 一并提出, 你要用 React.js 基本都要伴随着 Redux 和 React.js 结合的库 React-redux。

要注意的是, Redux 和 React-redux 并不是同一个东西。Redux 是一种架构模式 (Flux 架构的一种变种), 它不关注你到底用什么库, 你可以把它应用到 React 和 Vue, 甚至跟 jQuery 结合都没有问题。而 React-redux 就是把 Redux 这种架构模式和 React.js 结合起来的一个库, 就是 Redux 架构在 React.js 中的体现。

如果把 Redux 的用法重新介绍一遍那么这本书的价值就不大了, 我大可把官网的 Reducers、Actions、Store 的用法、API、关系重复一遍, 画几个图, 说两句很玄乎的话。但是这样对大家理解和使用 Redux 都没什么好处, 本书初衷还是跟开头所说的一样: 希望大家对问题的根源有所了解, 了解这些工具到底解决什么问题, 怎么解决的。

现在让我们忘掉 React.js、Redux 这些词, 从一个例子的代码 + 问题开始推演。

用 `create-react-app` 新建一个项目 `make-redux`, 修改 `public/index.html` 里面的 `body` 结构为:

```
<body>
  <div id='title'></div>
  <div id='content'></div>
</body>
```

删除 `src/index.js` 里面所有的代码, 添加下面代码, 代表我们应用的状态:

```
const appState = {
  title: {
    text: 'React.js 小书',
    color: 'red',
```

```

    },
    content: {
      text: 'React.js 小书内容',
      color: 'blue'
    }
}

```

我们新增几个渲染函数，它会把上面状态的数据渲染到页面上：

```

function renderApp (appState) {
  renderTitle(appState.title)
  renderContent(appState.content)
}

function renderTitle (title) {
  const titleDOM = document.getElementById('title')
  titleDOM.innerHTML = title.text
  titleDOM.style.color = title.color
}

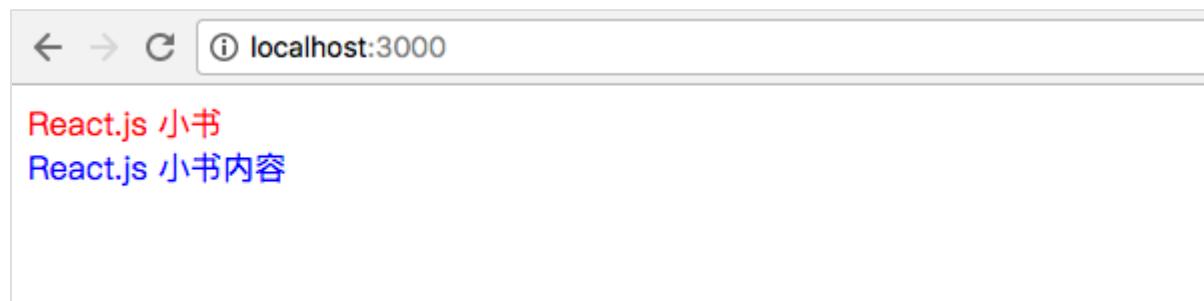
function renderContent (content) {
  const contentDOM = document.getElementById('content')
  contentDOM.innerHTML = content.text
  contentDOM.style.color = content.color
}

```

很简单，`renderApp` 会调用 `renderTitle` 和 `renderContent`，而这两者会把 `appState` 里面的数据通过原始的 DOM 操作更新到页面上，调用：

```
renderApp(appState)
```

你会在页面上看到：



这是一个很简单的 App，但是它存在一个重大的隐患，我们渲染数据的时候，使用的是一个共享状态 `appState`，**每个人都可以修改它**。如果我在渲染之前做了一系列其他操作：

```

loadDataFromServer()
doSomethingUnexpected()

```

```
doSomethingMore()
// ...
renderApp(appState)
```

`renderApp(appState)` 之前执行了一大堆函数操作，你根本不知道它们会对 `appState` 做什么事情，`renderApp(appState)` 的结果根本没法得到保障。一个可以被不同模块任意修改共享的数据状态就是魔鬼，一旦数据可以任意修改，**所有对共享状态的操作都是不可预料的**（某个模块 `appState.title = null` 你一点意见都没有），出现问题的时候 `debug` 起来就非常困难，这就是老生常谈的尽量避免全局变量。

你可能会说我看一下它们函数的实现就知道了它们修改了什么，在我们这个例子里面还算比较简单，但是真实项目当中的函数调用和数据初始化操作非常复杂，深层次的函数调用修改了状态是很难调试的。

但不同的模块（组件）之间确实需要共享数据，这些模块（组件）还可能需要修改这些共享数据，就像上一节的“主题色”状态（`themeColor`）。这里的矛盾就是：“**模块（组件）之间需要共享数据**”，和“**数据可能被任意修改导致不可预料的结果**”之间的矛盾。

让我们来想办法解决这个问题，我们可以学习 React.js 团队的做法，把事情搞复杂一些，提高数据修改的门槛：模块（组件）之间可以共享数据，也可以改数据。但是我们约定，这个数据并不能直接改，你只能执行某些我允许的某些修改，而且你修改的必须**大张旗鼓**地告诉我。

我们定义一个函数，叫 `dispatch`，它专门负责数据的修改：

```
function dispatch (action) {
  switch (action.type) {
    case 'UPDATE_TITLE_TEXT':
      appState.title.text = action.text
      break
    case 'UPDATE_TITLE_COLOR':
      appState.title.color = action.color
      break
    default:
      break
  }
}
```

**所有对数据的操作必须通过 `dispatch` 函数**。它接受一个参数 `action`，这个 `action` 是一个普通的 JavaScript 对象，里面必须包含一个 `type` 字段来声明你到底想干什么。`dispatch` 在 `switch` 里面会识别这个 `type` 字段，能够识别出来的操作才会执行对 `appState` 的修改。

上面的 `dispatch` 它只能识别两种操作，一种是 `UPDATE_TITLE_TEXT` 它会用 `action` 的 `text` 字段去更新 `appState.title.text`；一种是 `UPDATE_TITLE_COLOR`，它会用 `action` 的 `color` 字段去更新 `appState.title.color`。可以看到，`action` 里面除了 `type` 字段是必须的以外，其他字段都是可以自定义的。

任何的模块如果想要修改 `appState.title.text`，必须大张旗鼓地调用 `dispatch`：

```
dispatch({ type: 'UPDATE_TITLE_TEXT', text: '《React.js 小书》' }) // 修改标题文本
dispatch({ type: 'UPDATE_TITLE_COLOR', color: 'blue' }) // 修改标题颜色
```

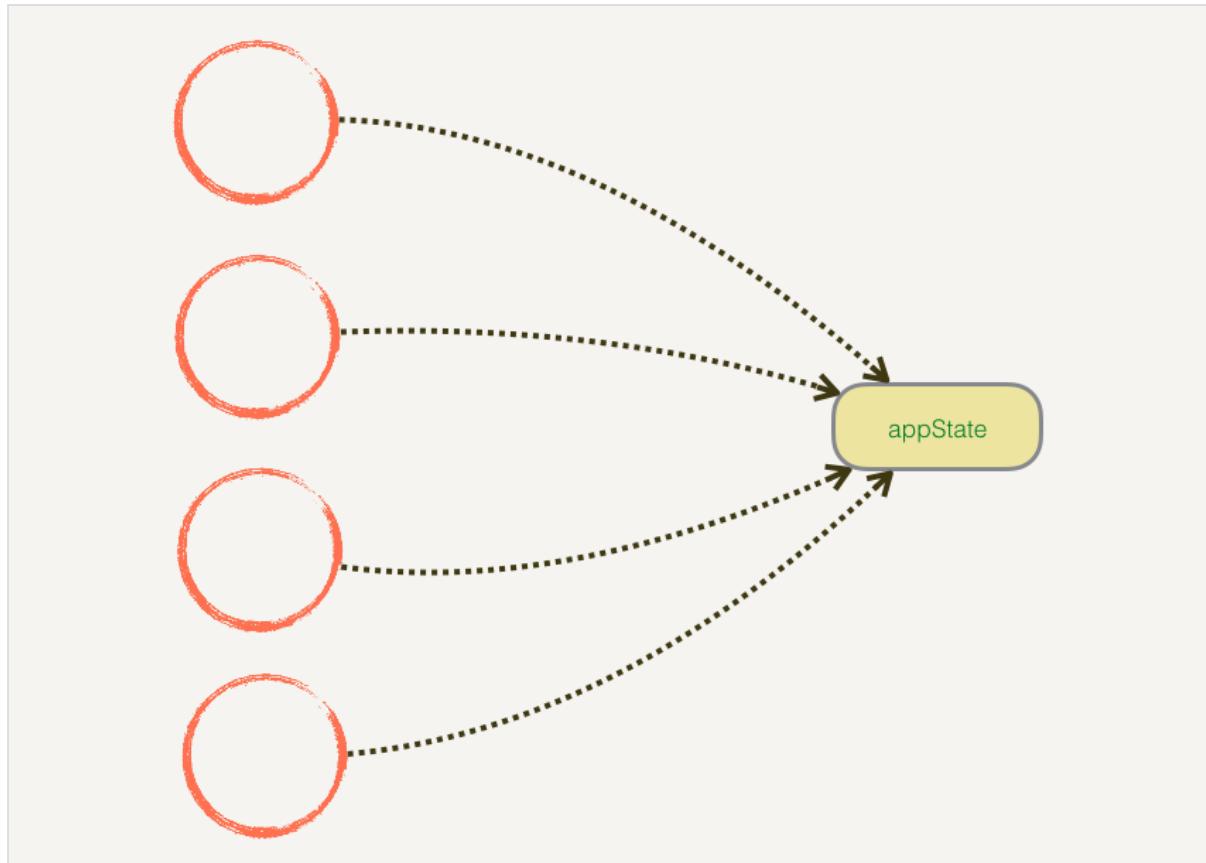
我们来看看有什么好处：

```
loadDataFromServer() // => 里面可能通过 dispatch 修改标题文本
doSomethingUnexpected()
doSomthingMore() // => 里面可能通过 dispatch 修改标题颜色
// ...
renderApp(appState)
```

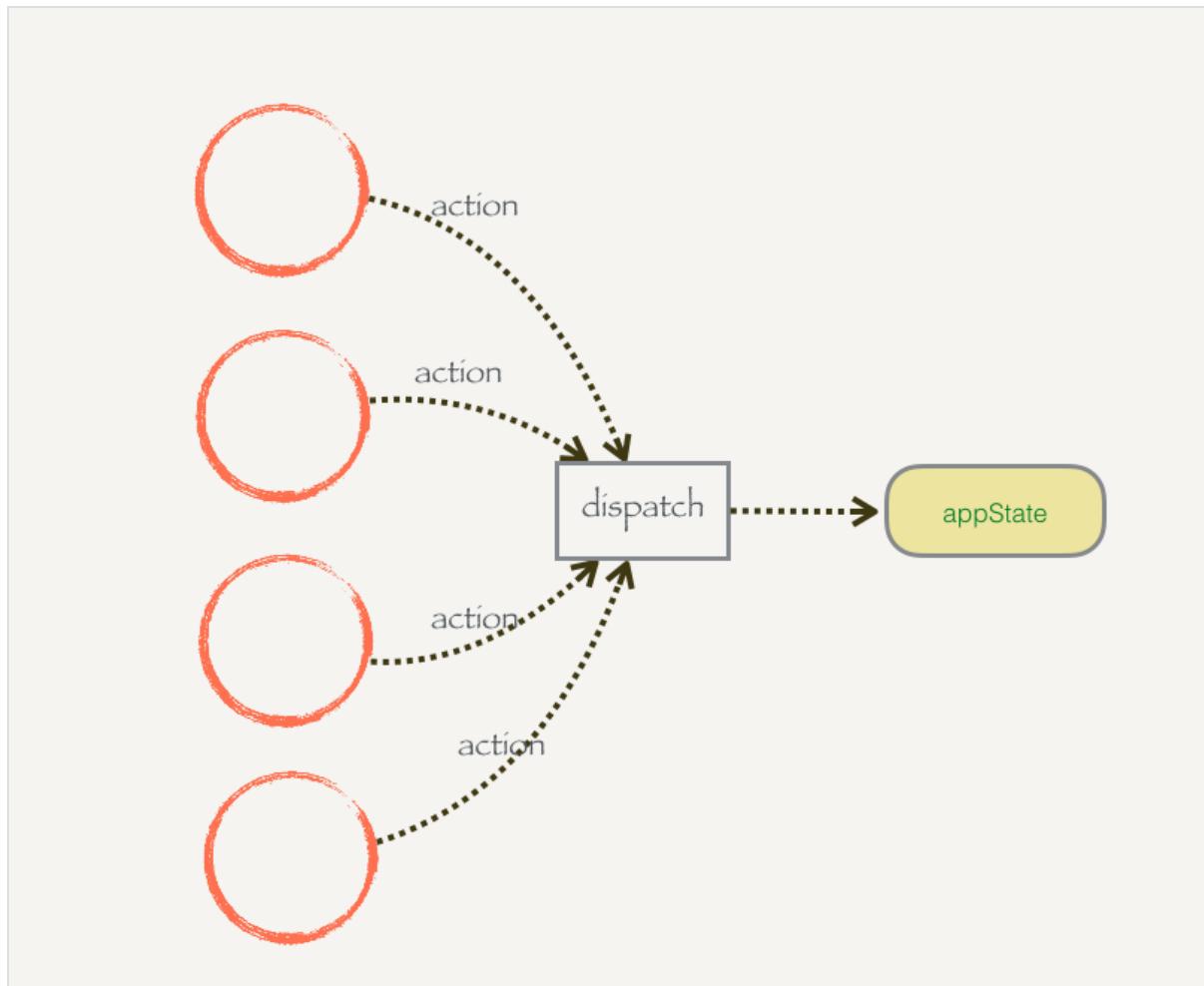
我们不需要担心 `renderApp(appState)` 之前的那堆函数操作会干什么奇奇怪怪得事情，因为我们规定不能直接修改 `appState`，它们对 `appState` 的修改必须只能通过 `dispatch`。而我们看看 `dispatch` 的实现可以知道，你只能修改 `title.text` 和 `title.color`。

如果某个函数修改了 `title.text` 但是我并不想要它这么干，我需要 `debug` 出来是哪个函数修改了，我只需要在 `dispatch` 的 `switch` 的第一个 `case` 内部打个断点就可以调试出来了。

原来模块（组件）修改共享数据是直接改的：



我们很难把控每一根指向 `appState` 的箭头，`appState` 里面的东西就无法把控。但现在我们必须通过一个“中间人” — `dispatch`，所有的数据修改必须通过它，并且你必须用 `action` 来大声告诉它要修改什么，只有它允许的才能修改：



我们再也不用担心共享数据状态的修改的问题，我们只要把控了 `dispatch`，所有的对 `appState` 的修改就无所遁形，毕竟只有一根箭头指向 `appState` 了。

本节完整的代码如下：

```
let appState = {
  title: {
    text: 'React.js 小书',
    color: 'red',
  },
  content: {
    text: 'React.js 小书内容',
    color: 'blue'
  }
}

function dispatch (action) {
  switch (action.type) {
    case 'UPDATE_TITLE_TEXT':
      appState.title.text = action.text
      break
    case 'UPDATE_TITLE_COLOR':
      appState.title.color = action.color
      break
  }
}
```

```
default:  
    break  
}  
}  
  
function renderApp (appState) {  
    renderTitle(appState.title)  
    renderContent(appState.content)  
}  
  
function renderTitle (title) {  
    const titleDOM = document.getElementById('title')  
    titleDOM.innerHTML = title.text  
    titleDOM.style.color = title.color  
}  
  
function renderContent (content) {  
    const contentDOM = document.getElementById('content')  
    contentDOM.innerHTML = content.text  
    contentDOM.style.color = content.color  
}  
  
renderApp(appState) // 首次渲染页面  
dispatch({ type: 'UPDATE_TITLE_TEXT', text: '《React.js 小书》' }) // 修改标题文本  
dispatch({ type: 'UPDATE_TITLE_COLOR', color: 'blue' }) // 修改标题颜色  
renderApp(appState) // 把新的数据渲染到页面上
```

下一节我们会把这种 `dispatch` 的模式抽离出来，让它变得更加通用。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

下一节：31. 动手实现 Redux (二) : 抽离 store 和监控数据变化

上一节：29. React.js 的 context

React.js 小书

[<-- 返回首页](#)

## 31. 动手实现 Redux (二) : 抽离 store 和监控数据变化

- 作者: [胡子大哈](#)
- 原文链接: <http://huziketang.com/books/react/lesson31>
- 转载请注明出处, 保留原文链接和作者信息。

(本文未审核)

### 抽离出 store

[上一节](#) 的我们有了 `appState` 和 `dispatch`:

```
let appState = {
  title: {
    text: 'React.js 小书',
    color: 'red',
  },
  content: {
    text: 'React.js 小书内容',
    color: 'blue'
  }
}

function dispatch (action) {
  switch (action.type) {
    case 'UPDATE_TITLE_TEXT':
      appState.title.text = action.text
      break
    case 'UPDATE_TITLE_COLOR':
      appState.title.color = action.color
      break
    default:
      break
  }
}
```

现在我们把它们集中到一个地方, 给这个地方起个名字叫做 `store`, 然后构建一个函数 `createStore`, 用来专门生产这种 `state` 和 `dispatch` 的集合, 这样别的 App 也可以用这种模式了:

```
function createStore (state, stateChanger) {
  const getState = () => state
  const dispatch = (action) => stateChanger(state, action)
  return { getState, dispatch }
}
```

`createStore` 接受两个参数，一个是表示应用程序状态的 `state`；另外一个是 `stateChanger`，它来描述应用程序状态会根据 `action` 发生什么变化，其实就是相当于本节开头的 `dispatch` 代码里面的内容。

`createStore` 会返回一个对象，这个对象包含两个方法 `getState` 和 `dispatch`。  
`getState` 用于获取 `state` 数据，其实就是简单地把 `state` 参数返回。

`dispatch` 用于修改数据，和以前一样会接受 `action`，然后它会把 `state` 和 `action` 一并传给 `stateChanger`，那么 `stateChanger` 就可以根据 `action` 来修改 `state` 了。

现在有了 `createStore`，我们可以这么修改原来的代码，保留原来所有的渲染函数不变，修改数据生成的方式：

```
let appState = {
  title: {
    text: 'React.js 小书',
    color: 'red',
  },
  content: {
    text: 'React.js 小书内容',
    color: 'blue'
  }
}

function stateChanger (state, action) {
  switch (action.type) {
    case 'UPDATE_TITLE_TEXT':
      state.title.text = action.text
      break
    case 'UPDATE_TITLE_COLOR':
      state.title.color = action.color
      break
    default:
      break
  }
}

const store = createStore(appState, stateChanger)

renderApp(store.getState()) // 首次渲染页面
store.dispatch({ type: 'UPDATE_TITLE_TEXT', text: '《React.js 小书》' }) // 修改标
```

```
store.dispatch({ type: 'UPDATE_TITLE_COLOR', color: 'blue' }) // 修改标题颜色
renderApp(store.getState()) // 把新的数据渲染到页面上
```

针对每个不同的 App，我们可以给 `createStore` 传入初始的数据 `appState`，和一个描述数据变化的函数 `stateChanger`，然后生成一个 `store`。需要修改数据的时候通过 `store.dispatch`，需要获取数据的时候通过 `store.getState`。

## 监控数据变化

上面的代码有一个问题，我们每次通过 `dispatch` 修改数据的时候，其实只是数据发生了变化，如果我们不手动调用 `renderApp`，页面上的内容是不会发生变化的。但是我们总不能每次 `dispatch` 的时候都手动调用一下 `renderApp`，我们肯定希望数据变化的时候程序能够智能一点地自动重新渲染数据，而不是手动调用。

你说这好办，往 `dispatch` 里面加 `renderApp` 就好了，但是这样 `createStore` 就不够通用了。我们希望用一种通用的方式“监听”数据变化，然后重新渲染页面，这里要用到观察者模式。修改 `createStore`：

```
function createStore (state, stateChanger) {
  const listeners = []
  const subscribe = (listener) => listeners.push(listener)
  const getState = () => state
  const dispatch = (action) => {
    stateChanger(state, action)
    listeners.forEach((listener) => listener())
  }
  return { getState, dispatch, subscribe }
}
```

我们在 `createStore` 里面定义了一个数组 `listeners`，还有一个新的方法 `subscribe`，可以通过 `store.subscribe(listener)` 的方式给 `subscribe` 传入一个监听函数，这个函数会被 `push` 到数组当中。

我们修改了 `dispatch`，每次当它被调用的时候，除了会调用 `stateChanger` 进行数据的修改，还会遍历 `listeners` 数组里面的函数，然后一个个地去调用。相当于我们可以通过 `subscribe` 传入数据变化的监听函数，每当 `dispatch` 的时候，监听函数就会被调用，这样我们就可以在每当数据变化时候进行重新渲染：

```
const store = createStore(appState, stateChanger)
store.subscribe(() => renderApp(store.getState()))

renderApp(store.getState()) // 首次渲染页面
store.dispatch({ type: 'UPDATE_TITLE_TEXT', text: '《React.js 小书》' }) // 修改标题文本
store.dispatch({ type: 'UPDATE_TITLE_COLOR', color: 'blue' }) // 修改标题颜色
// ...后面不管如何 store.dispatch，都不需要重新调用 renderApp
```

对观察者模式不熟悉的朋友可能会在这里晕头转向，建议了解一下这个设计模式的相关资料，然后进行练习：[实现一个 Event Emitter](#) 再进行阅读。

我们只需要 `subscribe` 一次，后面不管如何 `dispatch` 进行修改数据，`renderApp` 函数都会被重新调用，页面就会被重新渲染。这样的订阅模式还有好处就是，以后我们还可以拿同一块数据来渲染别的页面，这时 `dispatch` 导致的变化也会让每个页面都重新渲染：

```
const store = createStore(appState, stateChanger)
store.subscribe(() => renderApp(store.getState()))
store.subscribe(() => renderApp2(store.getState()))
store.subscribe(() => renderApp3(store.getState()))
...
...
```

本节的完整代码：

```
function createStore (state, stateChanger) {
  const listeners = []
  const subscribe = (listener) => listeners.push(listener)
  const getState = () => state
  const dispatch = (action) => {
    stateChanger(state, action)
    listeners.forEach((listener) => listener())
  }
  return { getState, dispatch, subscribe }
}

function renderApp (appState) {
  renderTitle(appState.title)
  renderContent(appState.content)
}

function renderTitle (title) {
  const titleDOM = document.getElementById('title')
  titleDOM.innerHTML = title.text
  titleDOM.style.color = title.color
}

function renderContent (content) {
  const contentDOM = document.getElementById('content')
  contentDOM.innerHTML = content.text
  contentDOM.style.color = content.color
}

let appState = {
  title: {
    text: 'React.js 小书',
    color: 'red',
  }
}
```

```

    },
    content: {
      text: 'React.js 小书内容',
      color: 'blue'
    }
}

function stateChanger (state, action) {
  switch (action.type) {
    case 'UPDATE_TITLE_TEXT':
      state.title.text = action.text
      break
    case 'UPDATE_TITLE_COLOR':
      state.title.color = action.color
      break
    default:
      break
  }
}

const store = createStore(appState, stateChanger)
store.subscribe(() => renderApp(store.getState())) // 监听数据变化

renderApp(store.getState()) // 首次渲染页面
store.dispatch({ type: 'UPDATE_TITLE_TEXT', text: '《React.js 小书》' }) // 修改标题文本
store.dispatch({ type: 'UPDATE_TITLE_COLOR', color: 'blue' }) // 修改标题颜色

```

## 总结

现在我们有了一个比较通用的 `createStore`，它可以产生一种我们新定义的数据类型 `store`，通过 `store.getState` 我们获取共享状态，而且我们约定只能通过 `store.dispatch` 修改共享状态。`store` 也允许我们通过 `store.subscribe` 监听数据状态被修改了，并且进行后续的例如重新渲染页面的操作。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：32. 动手实现 Redux \(三\) : 纯函数 \(Pure Function\) 简介](#)

[上一节：30. 动手实现 Redux \(一\) : 优雅地修改共享状态](#)

React.js 小书

[<-- 返回首页](#)

## 32. 动手实现 Redux (三) : 纯函数 (Pure Function) 简介

- 作者: [胡子大哈](#)
- 原文链接: <http://huziketang.com/books/react/lesson32>
- 转载请注明出处, 保留原文链接和作者信息。

(本文未审核)

我们接下来会继续优化我们的 `createStore` 的模式, 让它使我们的应用程序获得更好的性能。

但在开始之前, 我们先用一节的课程来介绍一下一个函数式编程里面非常重要的概念 — 纯函数 (Pure Function)。

简单来说, 一个函数的返回结果只依赖于它的参数, 并且在执行过程里面没有副作用, 我们把这个函数叫做纯函数。这么说肯定比较抽象, 我们把它掰开来看:

- 函数的返回结果只依赖于它的参数。
- 函数执行过程里面没有副作用。

### 函数的返回结果只依赖于它的参数

```
const a = 1
const foo = (b) => a + b
foo(2) // => 3
```

`foo` 函数不是一个纯函数, 因为它返回的结果依赖于外部变量 `a`, 我们在不知道 `a` 的值的情况下, 并不能保证 `foo(2)` 的返回值是 3。虽然 `foo` 函数的代码实现并没有变化, 传入的参数也没有变化, 但它的返回值却是不可预料的, 现在 `foo(2)` 是 3, 可能过了一会就是 4 了, 因为 `a` 可能发生了变化变成了 2。

```
const a = 1
const foo = (x, b) => x + b
foo(1, 2) // => 3
```

现在 `foo` 的返回结果只依赖于它的参数 `x` 和 `b`, `foo(1, 2)` 永远是 3。今天是 3, 明天也是 3, 在服务器跑是 3, 在客户端跑也 3, 不管你外部发生了什么变化,

`foo(1, 2)` 永远是 3。只要 `foo` 代码不改变，你传入的参数是确定的，那么 `foo(1, 2)` 的值永远是可预料的。

这就是纯函数的第一个条件：一个函数的返回结果只依赖于它的参数。

## 函数执行过程没有副作用

一个函数执行过程对产生了外部可观察的变化那么就说这个函数是有副作用的。

我们修改一下 `foo`：

```
const a = 1
const foo = (obj, b) => {
  return obj.x + b
}
const counter = { x: 1 }
foo(counter, 2) //=> 3
counter.x //=> 1
```

我们把原来的 `x` 换成了 `obj`，我现在可以往里面传一个对象进行计算，计算的过程里面并不会对传入的对象进行修改，计算前后的 `counter` 不会发生任何变化，计算前是 1，计算后也是 1，它现在是纯的。但是我再稍微修改一下它：

```
const a = 1
const foo = (obj, b) => {
  obj.x = 2
  return obj.x + b
}
const counter = { x: 1 }
foo(counter, 2) //=> 4
counter.x //=> 2
```

现在情况发生了变化，我在 `foo` 内部加了一句 `obj.x = 2`，计算前 `counter.x` 是 1，但是计算以后 `counter.x` 是 2。`foo` 函数的执行对外部的 `counter` 产生了影响，它产生了副作用，因为它修改了外部传进来的对象，现在它是不纯的。

但是你在函数内部构建的变量，然后进行数据的修改不是副作用：

```
const foo = (b) => {
  const obj = { x: 1 }
  obj.x = 2
  return obj.x + b
}
```

虽然 `foo` 函数内部修改了 `obj`, 但是 `obj` 是内部变量, 外部程序根本观察不到, 修改 `obj` 并不会产生外部可观察的变化, 这个函数是没有副作用的, 因此它是一个纯函数。

除了修改外部的变量, 一个函数在执行过程中还有很多方式产生外部可观察的变化, 比如说调用 `DOM API` 修改页面, 或者你发送了 `Ajax` 请求, 还有调用 `window.reload` 刷新浏览器, 甚至是 `console.log` 往控制台打印数据也是副作用。

纯函数很严格, 也就是说你几乎除了计算数据以外什么都不能干, 计算的时候还不能依赖除了函数参数以外的数据。

## 总结

一个函数的返回结果只依赖于它的参数, 并且在执行过程里面没有副作用, 我们就把这个函数叫做纯函数。

为什么要煞费苦心地构建纯函数? 因为纯函数非常“靠谱”, 执行一个纯函数你不用担心它会干什么坏事, 它不会产生不可预料的行为, 也不会对外部产生影响。不管何时何地, 你给它什么它就会乖乖地吐出什么。如果你的应用程序大多数函数都是由纯函数组成, 那么你的程序测试、调试起来会非常方便。

因为第三方评论工具有问题, 对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖, 我会回答大家的疑问。

---

[下一节: 33. 动手实现 Redux \(四\) : 共享结构的对象提高性能](#)

[上一节: 31. 动手实现 Redux \(二\) : 抽离 store 和监控数据变化](#)

React.js 小书

[<-- 返回首页](#)

## 33. 动手实现 Redux (四) : 共享结构的对象提高性能

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson33>
- 转载请注明出处，保留原文链接和作者信息。

(本文未审核)

接下来两节某些地方可能会稍微有一点点抽象，但是我会尽可能用简单的方式进行讲解。如果你觉得理解起来有点困难，可以把这几节多读多理解几遍，其实我们一路走来都是符合“逻辑”的，都是发现问题、思考问题、优化代码的过程。所以最好能够用心留意、思考我们每一个提出来的问题。

细心的朋友可以发现，其实我们之前的例子当中是有比较严重的**性能问题**的。我们在每个渲染函数的开头打一些 Log 看看：

```
function renderApp (appState) {
  console.log('render app...')
  renderTitle(appState.title)
  renderContent(appState.content)
}

function renderTitle (title) {
  console.log('render title...')
  const titleDOM = document.getElementById('title')
  titleDOM.innerHTML = title.text
  titleDOM.style.color = title.color
}

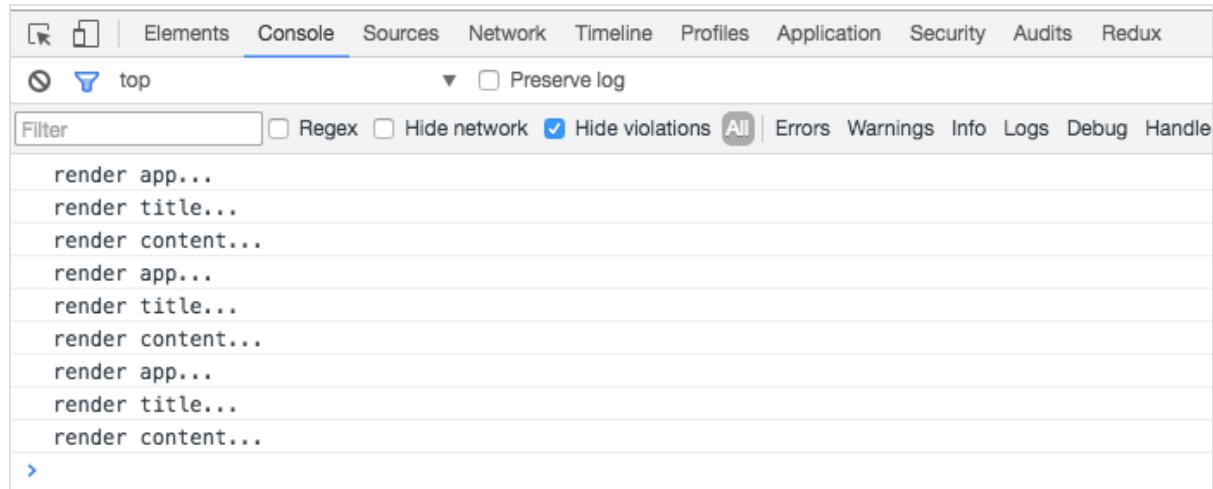
function renderContent (content) {
  console.log('render content...')
  const contentDOM = document.getElementById('content')
  contentDOM.innerHTML = content.text
  contentDOM.style.color = content.color
}
```

依旧执行一次初始化渲染，和两次更新，这里代码保持不变：

```
const store = createStore(appState, stateChanger)
store.subscribe(() => renderApp(store.getState())) // 监听数据变化
```

```
renderApp(store.getState()) // 首次渲染页面
store.dispatch({ type: 'UPDATE_TITLE_TEXT', text: '《React.js 小书》' }) // 修改标题文本
store.dispatch({ type: 'UPDATE_TITLE_COLOR', color: 'blue' }) // 修改标题颜色
```

可以在控制台看到：



前三个毫无疑问是第一次渲染打印出来的。中间三个是第一次 `store.dispatch` 导致的，最后三个是第二次 `store.dispatch` 导致的。可以看到问题就是，每当更新数据就重新渲染整个 App，但其实我们两次更新都没有动到 `appState` 里面的 `content` 字段的对象，而动的是 `title` 字段。其实并不需要重新 `renderContent`，它是一个多余的更新操作，现在我们需要优化它。

这里提出的解决方案是，在每个渲染函数执行渲染操作之前先做个判断，判断传入的新数据和旧的数据是不是相同，相同的话就不渲染了。

```
function renderApp (newAppState, oldAppState = {}) { // 防止 oldAppState 没有传入,
  if (newAppState === oldAppState) return // 数据没有变化就不渲染了
  console.log('render app...')
  renderTitle(newAppState.title, oldAppState.title)
  renderContent(newAppState.content, oldAppState.content)
}

function renderTitle (newTitle, oldTitle = {}) {
  if (newTitle === oldTitle) return // 数据没有变化就不渲染了
  console.log('render title...')
  const titleDOM = document.getElementById('title')
  titleDOM.innerHTML = newTitle.text
  titleDOM.style.color = newTitle.color
}

function renderContent (newContent, oldContent = {}) {
  if (newContent === oldContent) return // 数据没有变化就不渲染了
  console.log('render content...')
  const contentDOM = document.getElementById('content')
  contentDOM.innerHTML = newContent.text
```

```
    contentDOM.style.color = newContent.color
}
```

然后我们用一个 `oldState` 变量保存旧的应用状态，在需要重新渲染的时候把新旧数据传进去：

```
const store = createStore(appState, stateChanger)
let oldState = store.getState() // 缓存旧的 state
store.subscribe(() => {
  const newState = store.getState() // 数据可能变化，获取新的 state
  renderApp(newState, oldState) // 把新旧的 state 传进去渲染
  oldState = newState // 渲染完以后，新的 newState 变成了旧的 oldState，等待下一次
})
...
...
```

希望到这里没有把大家忽悠到，上面的代码根本不会达到我们的效果。看看我们的 `stateChanger`：

```
function stateChanger (state, action) {
  switch (action.type) {
    case 'UPDATE_TITLE_TEXT':
      state.title.text = action.text
      break
    case 'UPDATE_TITLE_COLOR':
      state.title.color = action.color
      break
    default:
      break
  }
}
```

即使你修改了 `state.title.text`，但是 `state` 还是原来那个 `state`，`state.title` 还是原来的 `state.title`，这些引用指向的还是原来的对象，只是对象内的内容发生了改变。所以即使你在每个渲染函数开头加了那个判断又什么用？这就像是下面的代码那样自欺欺人：

```
let appState = {
  title: {
    text: 'React.js 小书',
    color: 'red',
  },
  content: {
    text: 'React.js 小书内容',
    color: 'blue'
  }
}
const oldState = appState
```

```
appState.title.text = '《React.js 小书》'
oldState !== appState // false, 其实两个引用指向的是同一个对象，我们却希望它们不同
```

但是，我们接下来就要让这种事情变成可能。

## 共享结构的对象

希望大家都知道这种 ES6 的语法：

```
const obj = { a: 1, b: 2}
const obj2 = { ...obj } // => { a: 1, b: 2 }
```

`const obj2 = { ...obj }` 其实就是新建一个对象 `obj2`，然后把 `obj` 所有的属性都复制到 `obj2` 里面，相当于对象的浅复制。上面的 `obj` 里面的内容和 `obj2` 是完全一样的，但是却是两个不同的对象。除了浅复制对象，还可以覆盖、拓展对象属性：

```
const obj = { a: 1, b: 2}
const obj2 = { ...obj, b: 3, c: 4} // => { a: 1, b: 3, c: 4 }, 覆盖了 b，新增了 c
```

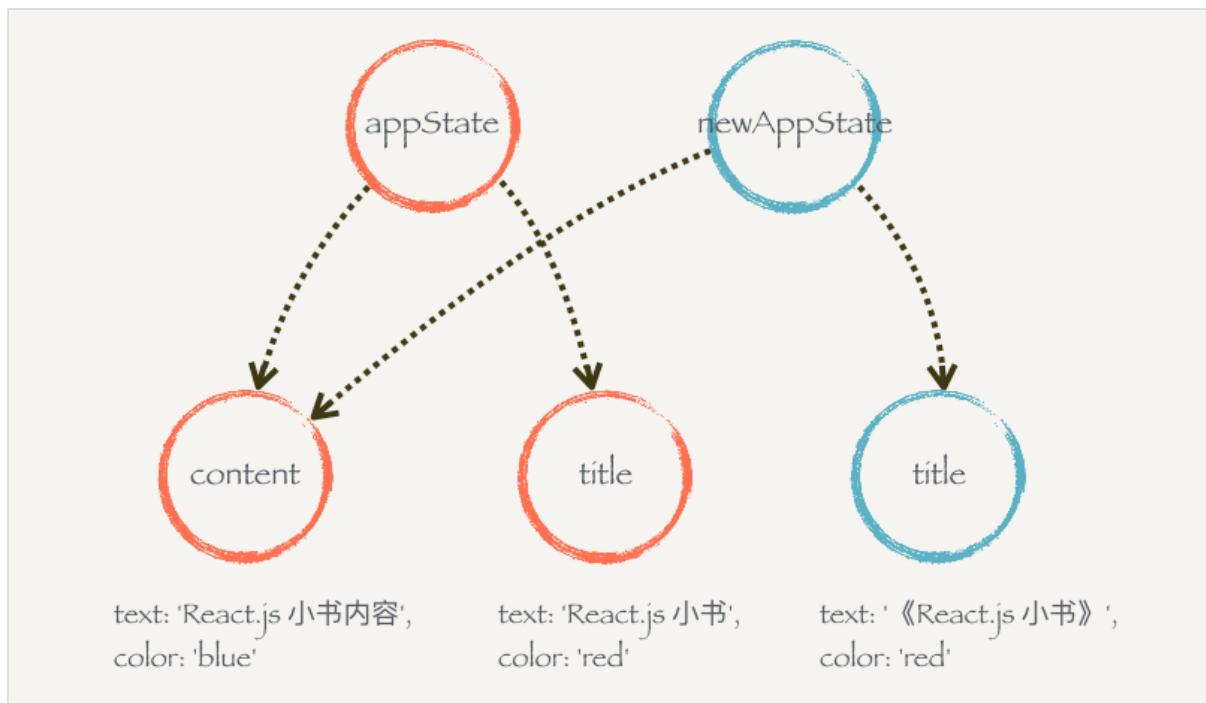
我们可以把这种特性应用在 `state` 的更新上，我们禁止直接修改原来的对象，一旦你要修改某些东西，你就得把修改路径上的所有对象复制一遍，例如，我们不写下面的修改代码：

```
appState.title.text = '《React.js 小书》'
```

取而代之的是，我们新建一个 `appState`，新建 `appState.title`，新建 `appState.title.text`：

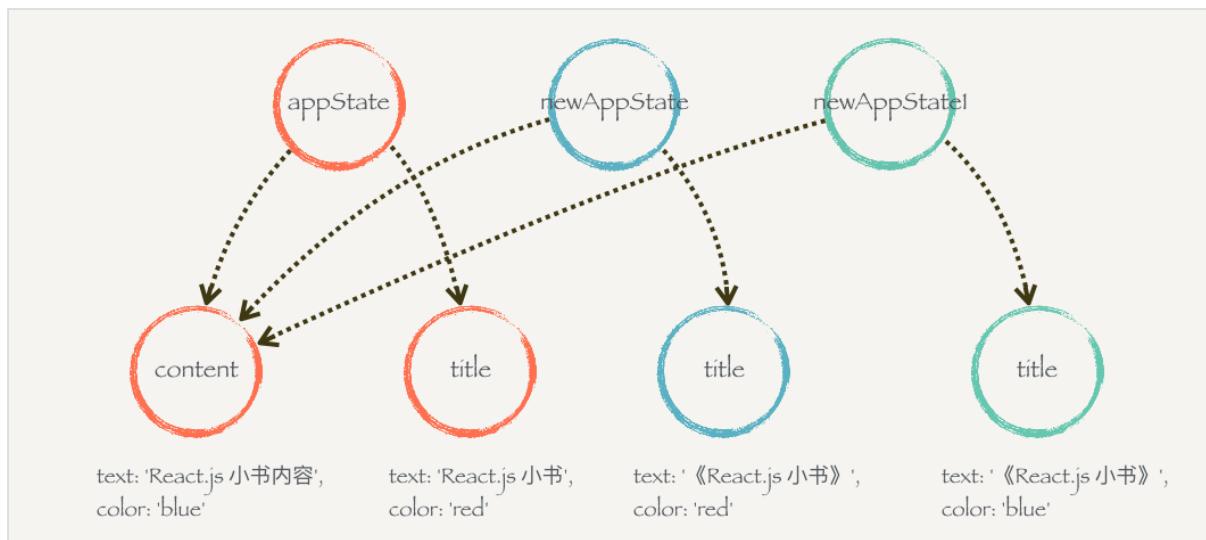
```
let newAppState = { // 新建一个 newAppState
  ...appState, // 复制 appState 里面的内容
  title: { // 用一个新的对象覆盖原来的 title 属性
    ...appState.title, // 复制原来 title 对象里面的内容
    text: '《React.js 小书》' // 覆盖 text 属性
  }
}
```

如果我们用一个树状的结构来表示对象结构的话：



`appState` 和 `newAppState` 其实是两个不同的对象，因为对象浅复制的缘故，其实它们里面的属性 `content` 指向的是同一个对象；但是因为 `title` 被一个新的对象覆盖了，所以它们的 `title` 属性指向的对象是不同的。同样地，修改 `appState.title.color`：

```
let newAppState1 = { // 新建一个 newAppState1
  ...newAppState, // 复制 newAppState1 里面的内容
  title: { // 用一个新的对象覆盖原来的 title 属性
    ...newAppState.title, // 复制原来 title 对象里面的内容
    color: "blue" // 覆盖 color 属性
  }
}
```



我们每次修改某些数据的时候，都不会碰原来的数据，而是把需要修改数据路径上的对象都 copy 一个出来。这样有什么好处？看看我们的目的达到了：

```
appState !== newState // true, 两个对象引用不同, 数据变化了, 重新渲染
appState.title !== newState.title // true, 两个对象引用不同, 数据变化了, 重新渲染
appState.content !== newState.content // false, 两个对象引用相同, 数据没有变化, 不
```

修改数据的时候就把修改路径都复制一遍, 但是保持其他内容不变, 最后的所有对象具有某些不变共享的结构 (例如上面三个对象都共享 `content` 对象)。大多数情况下我们可以保持 50% 以上的内容具有共享结构, 这种操作具有非常优良的特性, 我们可以用它来优化上面的渲染性能。

## 优化性能

我们修改 `stateChanger`, 让它修改数据的时候, 并不会直接修改原来的数据 `state`, 而是产生上述的共享结构的对象:

```
function stateChanger (state, action) {
  switch (action.type) {
    case 'UPDATE_TITLE_TEXT':
      return { // 构建新的对象并且返回
        ...state,
        title: {
          ...state.title,
          text: action.text
        }
      }
    case 'UPDATE_TITLE_COLOR':
      return { // 构建新的对象并且返回
        ...state,
        title: {
          ...state.title,
          color: action.color
        }
      }
    default:
      return state // 没有修改, 返回原来的对象
  }
}
```

代码稍微比原来长了一点, 但是是值得的。每次需要修改的时候都会产生新的对象, 并且返回。而如果没有修改 (在 `default` 语句中) 则返回原来的 `state` 对象。

因为 `stateChanger` 不会修改原来对象了, 而是返回对象, 所以我们需要修改一下 `createStore`。让它用每次 `stateChanger(state, action)` 的调用结果覆盖原来的 `state`:

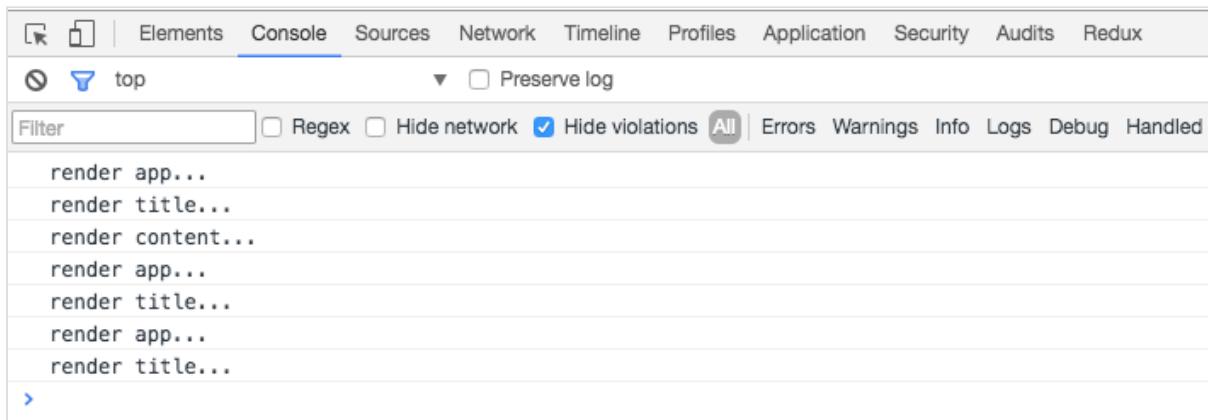
```
function createStore (state, stateChanger) {
  const listeners = []
```

```

const subscribe = (listener) => listeners.push(listener)
const getState = () => state
const dispatch = (action) => {
  state = stateChanger(state, action) // 覆盖原对象
  listeners.forEach((listener) => listener())
}
return { getState, dispatch, subscribe }
}

```

保持上面的渲染函数开头的对象判断不变，再看看控制台：



前三个是首次渲染。后面的 `store.dispatch` 导致的重新渲染都没有关于 `content` 的 Log 了。因为产生共享结构的对象，新旧对象的 `content` 引用指向的对象是一样的，所以触发了 `renderContent` 函数开头的：

```

...
if (newContent === oldContent) return
...

```

我们成功地把不必要的页面渲染优化掉了，问题解决。另外，并不需要担心每次修改都新建共享结构对象会有性能、内存问题，因为构建对象的成本非常低，而且我们最多保存两个对象引用 (`oldState` 和 `newState`)，其余旧的对象都会被垃圾回收掉。

本节完整代码：

```

function createStore (state, stateChanger) {
  const listeners = []
  const subscribe = (listener) => listeners.push(listener)
  const getState = () => state
  const dispatch = (action) => {
    state = stateChanger(state, action) // 覆盖原对象
    listeners.forEach((listener) => listener())
  }
  return { getState, dispatch, subscribe }
}

```

```
function renderApp (newAppState, oldAppState = {}) { // 防止 oldAppState 没有传入,
  if (newAppState === oldAppState) return // 数据没有变化就不渲染了
  console.log('render app...')
  renderTitle(newAppState.title, oldAppState.title)
  renderContent(newAppState.content, oldAppState.content)
}

function renderTitle (newTitle, oldTitle = {}) {
  if (newTitle === oldTitle) return // 数据没有变化就不渲染了
  console.log('render title...')
  const titleDOM = document.getElementById('title')
  titleDOM.innerHTML = newTitle.text
  titleDOM.style.color = newTitle.color
}

function renderContent (newContent, oldContent = {}) {
  if (newContent === oldContent) return // 数据没有变化就不渲染了
  console.log('render content...')
  const contentDOM = document.getElementById('content')
  contentDOM.innerHTML = newContent.text
  contentDOM.style.color = newContent.color
}

let appState = {
  title: {
    text: 'React.js 小书',
    color: 'red',
  },
  content: {
    text: 'React.js 小书内容',
    color: 'blue'
  }
}

function stateChanger (state, action) {
  switch (action.type) {
    case 'UPDATE_TITLE_TEXT':
      return { // 构建新的对象并且返回
        ...state,
        title: {
          ...state.title,
          text: action.text
        }
      }
    case 'UPDATE_TITLE_COLOR':
      return { // 构建新的对象并且返回
        ...state,
        title: {
          ...state.title,
          color: action.color
        }
      }
  }
}
```

```
default:  
    return state // 没有修改, 返回原来的对象  
}  
}  
  
const store = createStore(appState, stateChanger)  
let oldState = store.getState() // 缓存旧的 state  
store.subscribe(() => {  
    const newState = store.getState() // 数据可能变化, 获取新的 state  
    renderApp(newState, oldState) // 把新旧的 state 传进去渲染  
    oldState = newState // 渲染完以后, 新的 newState 变成了旧的 oldState, 等待下一次  
})  
  
renderApp(store.getState()) // 首次渲染页面  
store.dispatch({ type: 'UPDATE_TITLE_TEXT', text: '《React.js 小书》' }) // 修改标  
store.dispatch({ type: 'UPDATE_TITLE_COLOR', color: 'blue' }) // 修改标题颜色
```

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：34. 动手实现 Redux（五）：不要问为什么的 reducer](#)

[上一节：32. 动手实现 Redux（三）：纯函数（Pure Function）简介](#)

React.js 小书

[<-- 返回首页](#)

## 34. 动手实现 Redux（五）：不要问为什么的 reducer

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson34>
- 转载请注明出处，保留原文链接和作者信息。

(本文未审核)

经过了这么多节的优化，我们有了一个很通用的 `createStore`：

```
function createStore (state, stateChanger) {  
  const listeners = []  
  const subscribe = (listener) => listeners.push(listener)  
  const getState = () => state  
  const dispatch = (action) => {  
    state = stateChanger(state, action) // 覆盖原对象  
    listeners.forEach((listener) => listener())  
  }  
  return { getState, dispatch, subscribe }  
}
```

它的使用方式是：

```
let appState = {  
  title: {  
    text: 'React.js 小书',  
    color: 'red',  
  },  
  content: {  
    text: 'React.js 小书内容',  
    color: 'blue'  
  }  
}  
  
function stateChanger (state, action) {  
  switch (action.type) {  
    case 'UPDATE_TITLE_TEXT':  
      return {  
        ...state,  
        title: {  
          ...state.title,  
        }  
      }  
    default:  
      return state  
  }  
}
```

```

        text: action.text
    }
}

case 'UPDATE_TITLE_COLOR':
    return {
        ...state,
        title: {
            ...state.title,
            color: action.color
        }
    }
default:
    return state
}
}

const store = createStore(appState, stateChanger)
...

```

我们再优化一下，其实 `appState` 和 `stateChanger` 可以合并到一起去：

```

function stateChanger (state, action) {
    if (!state) {
        return {
            title: {
                text: 'React.js 小书',
                color: 'red',
            },
            content: {
                text: 'React.js 小书内容',
                color: 'blue'
            }
        }
    }
    switch (action.type) {
        case 'UPDATE_TITLE_TEXT':
            return {
                ...state,
                title: {
                    ...state.title,
                    text: action.text
                }
            }
        case 'UPDATE_TITLE_COLOR':
            return {
                ...state,
                title: {
                    ...state.title,
                    color: action.color
                }
            }
    }
}

```

```

    default:
      return state
  }
}

```

`stateChanger` 现在既充当了获取初始化数据的功能，也充当了生成更新数据的功能。如果有传入 `state` 就生成更新数据，否则就是初始化数据。这样我们可以优化 `createStore` 成一个参数，因为 `state` 和 `stateChanger` 合并到一起了：

```

function createStore (stateChanger) {
  let state = null
  const listeners = []
  const subscribe = (listener) => listeners.push(listener)
  const getState = () => state
  const dispatch = (action) => {
    state = stateChanger(state, action)
    listeners.forEach((listener) => listener())
  }
  dispatch({}) // 初始化 state
  return { getState, dispatch, subscribe }
}

```

`createStore` 内部的 `state` 不再通过参数传入，而是一个局部变量 `let state = null`。`createStore` 的最后会手动调用一次 `dispatch({})`，`dispatch` 内部会调用 `stateChanger`，这时候的 `state` 是 `null`，所以这次的 `dispatch` 其实就是初始化数据了。`createStore` 内部第一次的 `dispatch` 导致 `state` 初始化完成，后续外部的 `dispatch` 就是修改数据的行为了。

我们给 `stateChanger` 这个玩意起一个通用的名字：reducer，不要问为什么，它就是个名字而已，修改 `createStore` 的参数名字：

```

function createStore (reducer) {
  let state = null
  const listeners = []
  const subscribe = (listener) => listeners.push(listener)
  const getState = () => state
  const dispatch = (action) => {
    state = reducer(state, action)
    listeners.forEach((listener) => listener())
  }
  dispatch({}) // 初始化 state
  return { getState, dispatch, subscribe }
}

```

这是一个最终形态的 `createStore`，它接受的参数叫 `reducer`，`reducer` 是一个函数，细心的朋友会发现，它其实是一个纯函数（Pure Function）。

## reducer

`createStore` 接受一个叫 `reducer` 的函数作为参数, 这个函数规定是一个纯函数, 它接受两个参数, 一个是 `state`, 一个是 `action`。

如果没有传入 `state` 或者 `state` 是 `null`, 那么它就会返回一个初始化的数据。如果有传入 `state` 的话, 就会根据 `action` 来“修改”数据, 但其实它没有、也规定不能修改 `state`, 而是要通过上节所说的把修改路径的对象都复制一遍, 然后产生一个新的对象返回。如果它不能识别你的 `action`, 它就不会产生新的数据, 而是 (在 `default` 内部) 把 `state` 原封不动地返回。

`reducer` 是不允许有副作用的。你不能在里面操作 DOM, 也不能发 Ajax 请求, 更不能直接修改 `state`, 它要做的仅仅是 — 初始化和计算新的 `state`。

现在我们可以用这个 `createStore` 来构建不同的 `store` 了, 只要给它传入符合上述的定义的 `reducer` 即可:

```
function themeReducer (state, action) {
  if (!state) return {
    themeName: 'Red Theme',
    themeColor: 'red'
  }
  switch (action.type) {
    case 'UPDATE_THEME_NAME':
      return { ...state, themeName: action.themeName }
    case 'UPDATE_THEME_COLOR':
      return { ...state, themeColor: action.themeColor }
    default:
      return state
  }
}

const store = createStore(themeReducer)
...
```

因为第三方评论工具有问题, 对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖, 我会回答大家的疑问。

[下一节: 35. 动手实现 Redux \(六\) : Redux 总结](#)

[上一节: 33. 动手实现 Redux \(四\) : 共享结构的对象提高性能](#)



React.js 小书

[<-- 返回首页](#)

## 35. 动手实现 Redux (六) : Redux 总结

- 作者: [胡子大哈](#)
- 原文链接: <http://huziketang.com/books/react/lesson35>
- 转载请注明出处, 保留原文链接和作者信息。

(本文未审核)

不知不觉地, 到这里大家不仅仅已经掌握了 Redux, 而且还自己动手写了一个 Redux。我们从一个非常原始的代码开始, 不停地在发现问题、解决问题、优化代码的过程中进行推演, 最后把 Redux 模式自己总结出来了。这就是所谓的 Redux 模式, 我们再来看看一下这几节我们到底干了什么事情。

我们从一个简单的例子的代码中发现了共享的状态如果可以被任意修改的话, 那么程序的行为将非常不可预料, 所以我们提高了修改数据的门槛: 你必须通过 `dispatch` 执行某些允许的修改操作, 而且必须大张旗鼓的在 `action` 里面声明。

这种模式挺好用的, 我们就把它抽象出来一个 `createStore`, 它可以产生 `store`, 里面包含 `getState` 和 `dispatch` 函数, 方便我们使用。

后来发现每次修改数据都需要手动重新渲染非常麻烦, 我们希望自动重新渲染视图。所以来加入了订阅者模式, 可以通过 `store.subscribe` 订阅数据修改事件, 每次数据更新的时候自动重新渲染视图。

接下来我们发现了原来的“重新渲染视图”有比较严重的性能问题, 我们引入了“共享结构的对象”来帮我们解决问题, 这样就可以在每个渲染函数的开头进行简单的判断避免没有被修改过的数据重新渲染。

我们优化了 `stateChanger` 为 `reducer`, 定义了 `reducer` 只能是纯函数, 功能就是负责初始 `state`, 和根据 `state` 和 `action` 计算具有共享结构的新 `state`。

`createStore` 现在可以直接拿来用了, 套路就是:

```
// 定一个 reducer
function reducer (state, action) {
  /* 初始化 state 和 switch case */
}

// 生成 store
const store = createStore(reducer)
```

```
// 监听数据变化重新渲染页面  
store.subscribe(() => renderApp(store.getState()))  
  
// 首次渲染页面  
renderApp(store.getState())  
  
// 后面可以随意 dispatch 了，页面自动更新  
store.dispatch(...)
```

现在的代码跟 React.js 一点关系都没有，接下来我们要把 React.js 和 Redux 结合起来，用 Redux 模式帮助管理 React.js 的应用状态。

## 课后练习

- 
- [实现 Users Reducer](#)

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

下一节：36. 动手实现 React-redux (一) : 初始化工程

上一节：34. 动手实现 Redux (五) : 不要问为什么的 reducer

React.js 小书

[<-- 返回首页](#)

## 36. 动手实现 React-redux (一) : 初始化工程

- 作者: [胡子大哈](#)
- 原文链接: <http://huziketang.com/books/react/lesson36>
- 转载请注明出处, 保留原文链接和作者信息。

(本文未审核)

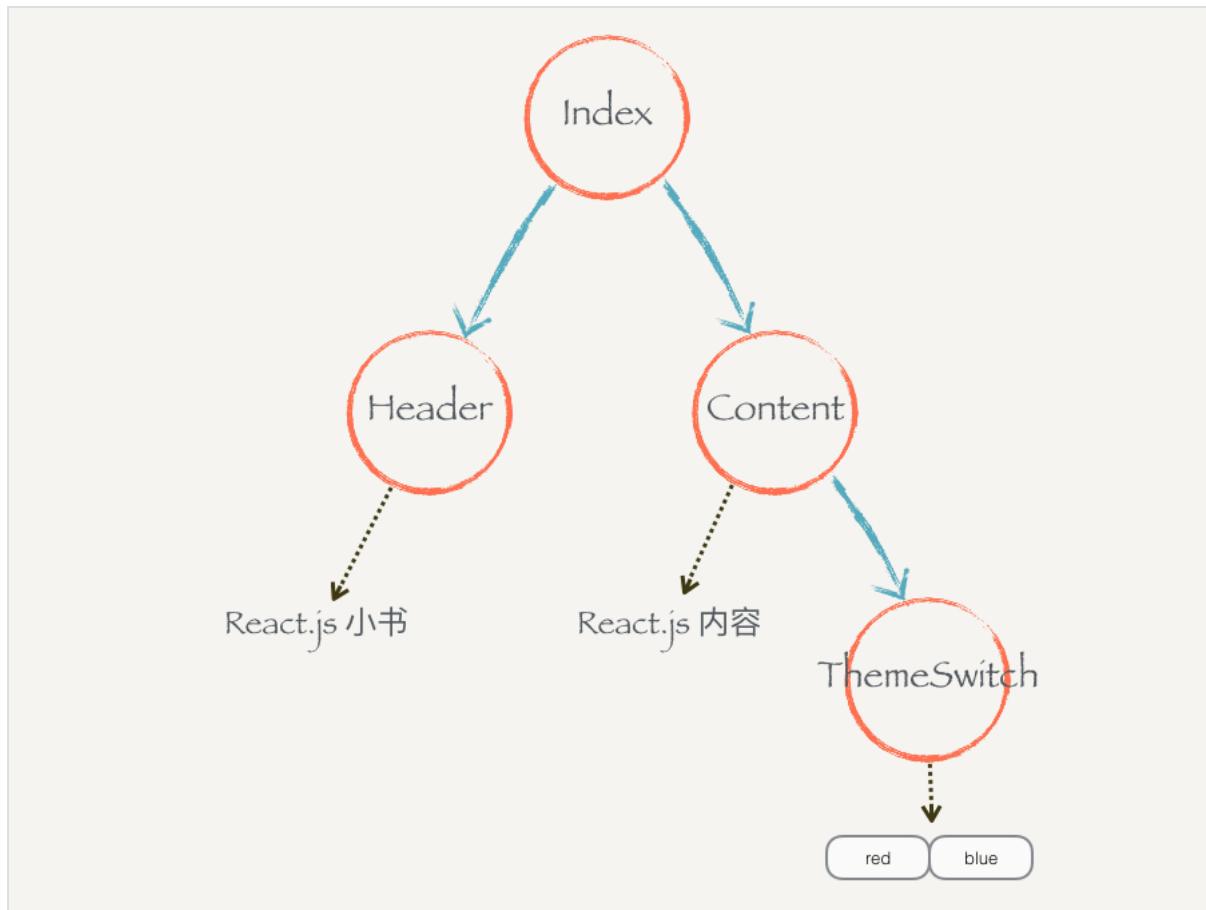
可以看到 Redux 并不复杂, 它那些看起来匪夷所思的设定其实都是为了解决特定的问题而存在的, 我们把问题想清楚以后就不难理解它的那些奇怪的设定了。这节开始我们来看看如何把 Redux 和 React.js 结合起来, 你会发现其实它们也并不复杂。

回顾一下, 我们在 [前端应用状态管理 — 状态提升](#) 中提过, 前端中应用的状态存在的问题: 一个状态可能被多个组件依赖或者影响, 而 React.js 并没有提供好的解决方案, 我们只能把状态提升到依赖或者影响这个状态的所有组件的公共父组件上, 我们把这种行为叫做状态提升。但是需求不停变化, 共享状态没完没了地提升也不是办法。

后来我们在 [React.js 的 context](#) 中提出, 我们可用把共享状态放到父组件的 context 上, 这个父组件下所有的组件都可以从 context 中直接获取到状态而不需要一层层地进行传递了。但是直接从 context 里面存放、获取数据增强了组件的耦合性; 并且所有组件都可以修改 context 里面的状态就像谁都可以修改共享状态一样, 导致程序运行的不可预料。

既然这样, 为什么不把 context 和 store 结合起来? 毕竟 store 的数据不是谁都能修改, 而是约定只能通过 `dispatch` 来进行修改, 这样的话每个组件既可以去 context 里面获取 store 从而获取状态, 又不用担心它们乱改数据了。

听起来不错, 我们动手试一下。我们还是拿“主题色”这个例子做讲解, 假设我们现在需要做下面这样的组件树:



`Header` 和 `Content` 的组件的文本内容会随着主题色的变化而变化，而 `Content` 下的子组件 `ThemeSwitch` 有两个按钮，可以切换红色和蓝色两种主题，按钮的颜色也会随着主题色的变化而变化。

用 `create-react-app` 新建一个工程，然后安装一个 React 提供的第三方库 `prop-types`：

```
npm install --save prop-types
```

安装好后在 `src/` 目录下新增三个文件：`Header.js`、`Content.js`、`ThemeSwitch.js`。

修改 `src/Header.js`：

```

import React, { Component } from 'react'
import PropTypes from 'prop-types'

class Header extends Component {
  render () {
    return (
      <h1>React.js 小书</h1>
    )
  }
}
  
```

```
export default Header
```

修改 `src/ThemeSwitch.js`:

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'

class ThemeSwitch extends Component {
  render () {
    return (
      <div>
        <button>Red</button>
        <button>Blue</button>
      </div>
    )
  }
}

export default ThemeSwitch
```

修改 `src/Content.js`:

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'
import ThemeSwitch from './ThemeSwitch'

class Content extends Component {
  render () {
    return (
      <div>
        <p>React.js 小书内容</p>
        <ThemeSwitch />
      </div>
    )
  }
}

export default Content
```

修改 `src/index.js`:

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'
import ReactDOM from 'react-dom'
import Header from './Header'
import Content from './Content'
import './index.css'
```

```
class Index extends Component {  
  render () {  
    return (  
      <div>  
        <Header />  
        <Content />  
      </div>  
    )  
  }  
}  
  
ReactDOM.render(  
  <Index />,  
  document.getElementById('root')  
)
```

这样我们就简单地把整个组件树搭建起来了，用 `npm start` 启动工程，然后可以看到页面上显示：



当然现在文本都没有颜色，而且点击按钮也不会有什么反应，我们还没有加入表示主题色的状态和相关的业务逻辑，下一节我们就把相关的逻辑加进去。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

下一节：37. 动手实现 React-redux (二) : 结合 context 和 store

上一节：35. 动手实现 Redux (六) : Redux 总结

React.js 小书

[<-- 返回首页](#)

## 37. 动手实现 React-redux (二) : 结合 context 和 store

- 作者: [胡子大哈](#)
- 原文链接: <http://huziketang.com/books/react/lesson37>
- 转载请注明出处, 保留原文链接和作者信息。

(本文未审核)

既然要把 store 和 context 结合起来, 我们就先构建 store。在 `src/index.js` 加入之前创建的 `createStore` 函数, 并且构建一个 `themeReducer` 来生成一个 `store`:

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'
import ReactDOM from 'react-dom'
import Header from './Header'
import Content from './Content'
import './index.css'

function createStore (reducer) {
  let state = null
  const listeners = []
  const subscribe = (listener) => listeners.push(listener)
  const getState = () => state
  const dispatch = (action) => {
    state = reducer(state, action)
    listeners.forEach((listener) => listener())
  }
  dispatch({}) // 初始化 state
  return { getState, dispatch, subscribe }
}

const themeReducer = (state, action) => {
  if (!state) return {
    themeColor: 'red'
  }
  switch (action.type) {
    case 'CHANGE_COLOR':
      return { ...state, themeColor: action.themeColor }
    default:
      return state
  }
}
```

```
const store = createStore(themeReducer)
```

```
...
```

`themeReducer` 定义了一个表示主题色的状态 `themeColor`，并且规定了一种操作 `CHANGE_COLOR`，只能通过这种操作修改颜色。现在我们把 `store` 放到 `Index` 的 `context` 里面，这样每个子组件都可以获取到 `store` 了，修改 `src/index.js` 里面的 `Index`：

```
class Index extends Component {
  static childContextTypes = {
    store: PropTypes.object
  }

  getChildContext () {
    return { store }
  }

  render () {
    return (
      <div>
        <Header />
        <Content />
      </div>
    )
  }
}
```

如果有些同学已经忘记了 `context` 的用法，可以参考之前的章节：[React.js 的 context](#)。

然后修改 `src/Header.js`，让它从 `Index` 的 `context` 里面获取 `store`，并且获取里面的 `themeColor` 状态来设置自己的颜色：

```
class Header extends Component {
  static contextTypes = {
    store: PropTypes.object
  }

  constructor () {
    super()
    this.state = { themeColor: '' }
  }

  componentWillMount () {
    this._updateThemeColor()
  }
```

```

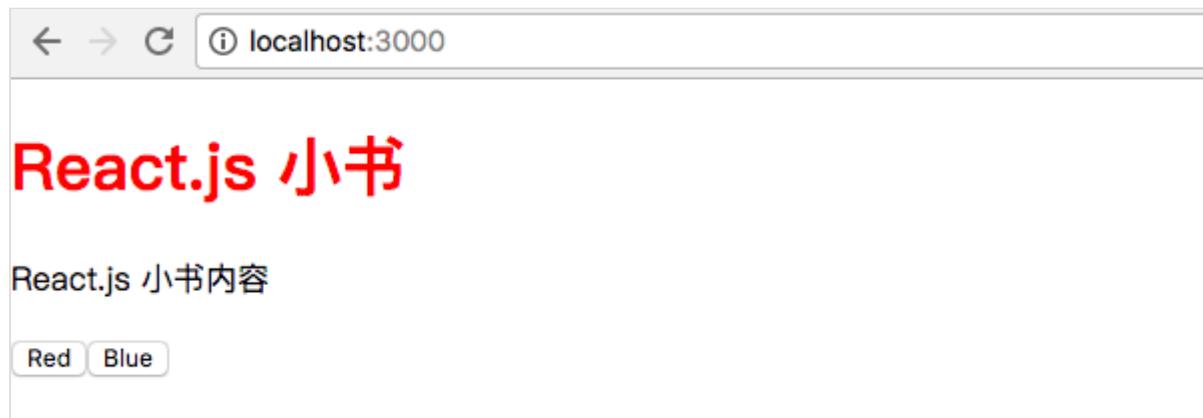
_updateThemeColor () {
  const { store } = this.context
  const state = store.getState()
  this.setState({ themeColor: state.themeColor })
}

render () {
  return (
    <h1 style={{ color: this.state.themeColor }}>React.js 小书</h1>
  )
}
}

```

其实也很简单，我们在 `constructor` 里面初始化了组件自己的 `themeColor` 状态。然后在生命周期中 `componentWillMount` 调用 `_updateThemeColor`，`_updateThemeColor` 会从 `context` 里面把 `store` 取出来，然后通过 `store.getState()` 获取状态对象，并且用里面的 `themeColor` 字段设置组件的 `state.themeColor`。

然后在 `render` 函数里面获取了 `state.themeColor` 来设置标题的样式，页面上就会显示：



如法炮制 `Content.js`：

```

class Content extends Component {
  static contextTypes = {
    store: PropTypes.object
  }

  constructor () {
    super()
    this.state = { themeColor: '' }
  }

  componentWillMount () {
    this._updateThemeColor()
  }

  _updateThemeColor () {

```

```

    const { store } = this.context
    const state = store.getState()
    this.setState({ themeColor: state.themeColor })
}

render () {
  return (
    <div>
      <p style={{ color: this.state.themeColor }}>React.js 小书内容</p>
      <ThemeSwitch />
    </div>
  )
}
}

```

还有 `src/ThemeSwitch.js`:

```

class ThemeSwitch extends Component {
  static contextTypes = {
    store: PropTypes.object
  }

  constructor () {
    super()
    this.state = { themeColor: '' }
  }

  componentWillMount () {
    this._updateThemeColor()
  }

  _updateThemeColor () {
    const { store } = this.context
    const state = store.getState()
    this.setState({ themeColor: state.themeColor })
  }

  render () {
    return (
      <div>
        <button style={{ color: this.state.themeColor }}>Red</button>
        <button style={{ color: this.state.themeColor }}>Blue</button>
      </div>
    )
  }
}

```

这时候，主题已经完全生效了，整个页面都是红色的：



当然现在点按钮还是没什么效果，我们接下来给按钮添加事件。其实也很简单，监听 `onClick` 事件然后 `store.dispatch` 一个 `action` 就好了，修改 `src/ThemeSwitch.js`：

```
class ThemeSwitch extends Component {
  static contextTypes = {
    store: PropTypes.object
  }

  constructor () {
    super()
    this.state = { themeColor: '' }
  }

  componentWillMount () {
    this._updateThemeColor()
  }

  _updateThemeColor () {
    const { store } = this.context
    const state = store.getState()
    this.setState({ themeColor: state.themeColor })
  }

  // dispatch action 去改变颜色
  handleSwitchColor (color) {
    const { store } = this.context
    store.dispatch({
      type: 'CHANGE_COLOR',
      themeColor: color
    })
  }

  render () {
    return (
      <div>
        <button
          style={{ color: this.state.themeColor }}
          onClick={this.handleSwitchColor.bind(this, 'red')}>Red</button>
    
```

```

    <button
      style={{ color: this.state.themeColor }}
      onClick={this.handleSwitchColor.bind(this, 'blue')}>Blue</button>
  </div>
)
}
}

```

我们给两个按钮都加上了 `onClick` 事件监听，并绑定到了 `handleSwitchColor` 方法上，两个按钮分别给这个方法传入不同的颜色 `red` 和 `blue`，`handleSwitchColor` 会根据传入的颜色 `store.dispatch` 一个 `action` 去修改颜色。

当然你现在点击按钮还是没有反应的。因为点击按钮的时候，只是更新 `store` 里面的 `state`，而并没有在 `store.state` 更新以后去重新渲染数据，我们其实就是忘了 `store.subscribe` 了。

给 `Header.js`、`Content.js`、`ThemeSwitch.js` 的 `componentWillMount` 生命周期都加上监听数据变化重新渲染的代码：

```

...
componentWillMount () {
  const { store } = this.context
  this._updateThemeColor()
  store.subscribe(() => this._updateThemeColor())
}
...

```

通过 `store.subscribe`，在数据变化的时候重新调用 `_updateThemeColor`，而 `_updateThemeColor` 会去 `store` 里面取最新的 `themeColor` 然后通过 `setState` 重新渲染组件，这时候组件就更新了。现在可以自由切换主题色了：



我们顺利地把 `store` 和 `context` 结合起来，这是 Redux 和 React.js 的第一次胜利会师，当然还有很多需要优化的地方。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

下一节：38. 动手实现 React-redux (三) : connect 和 mapStateToProps

上一节：36. 动手实现 React-redux (一) : 初始化工程

React.js 小书

[<-- 返回首页](#)

## 38. 动手实现 React-redux (三) : connect 和 mapStateToProps

- 作者: [胡子大哈](#)
- 原文链接: <http://huziketang.com/books/react/lesson38>
- 转载请注明出处, 保留原文链接和作者信息。

(本文未审核)

我们来观察一下刚写下的这几个组件, 可以轻易地发现它们有两个重大的问题:

- **有大量重复的逻辑:** 它们基本的逻辑都是, 取出 context, 取出里面的 store, 然后用里面的状态设置自己的状态, 这些代码逻辑其实都是相同的。
- **对 context 依赖性过强:** 这些组件都要依赖 context 来取数据, 使得这个组件复用性基本为零。想一下, 如果别人需要用到里面的 `ThemeSwitch` 组件, 但是他们的组件树并没有 context 也没有 store, 他们没法用这个组件了。

对于第一个问题, 我们在 [高阶组件](#) 的章节说过, 可以把一些可复用的逻辑放在高阶组件当中, 高阶组件包装的新组件和原来组件之间通过 `props` 传递信息, 减少代码的重复程度。

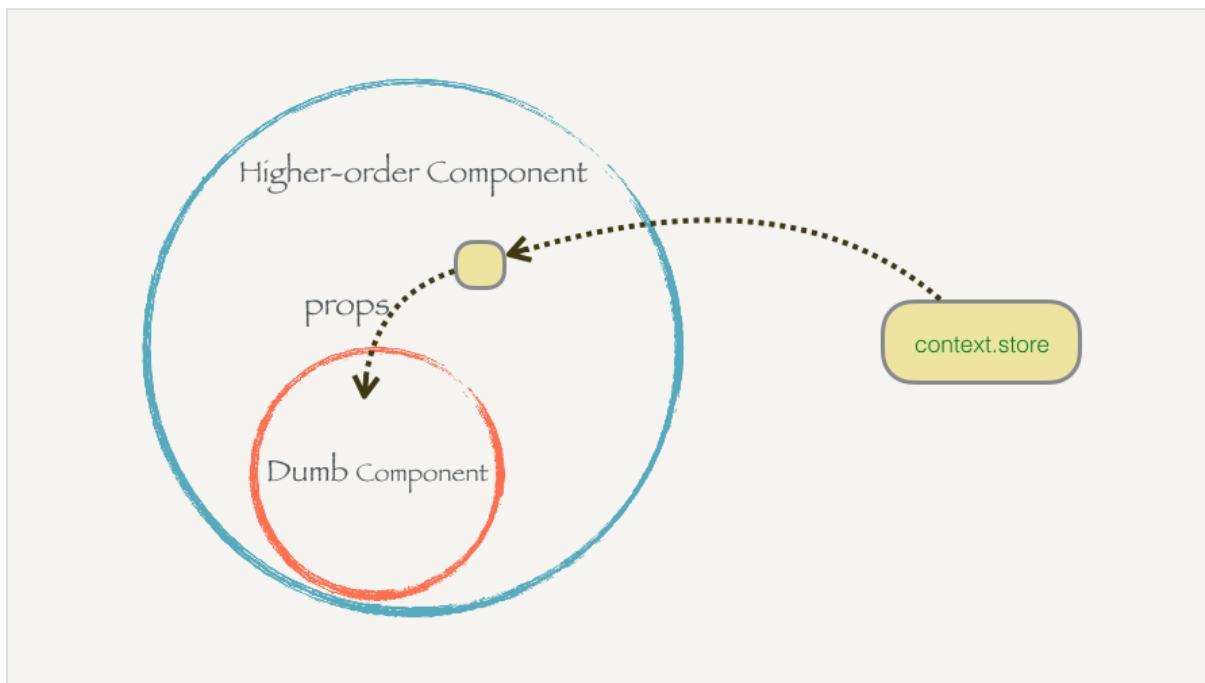
对于第二个问题, 我们得弄清楚一件事情, 到底什么样的组件才叫复用性强的组件。如果一个组件对外界的依赖过于强, 那么这个组件的移植性会很差, 就像这些严重依赖 `context` 的组件一样。

如果一个组件的渲染只依赖于外界传进去的 `props` 和自己的 `state`, 而并不依赖于其他的外界的任何数据, 也就是说像纯函数一样, 给它什么, 它就吐出(渲染)什么出来。这种组件的复用性是最强的, 别人使用的时候根本不用担心任何事情, 只要看看 `PropTypes` 它能接受什么参数, 然后把参数传进去控制它就行了。

我们把这种组件叫做 `Pure Component`, 因为它就像纯函数一样, 可预测性非常强, 对参数 (`props`) 以外的数据零依赖, 也不产生副作用。这种组件也叫 `Dumb Component`, 因为它们呆呆的, 让它干啥就干啥。写组件的时候尽量写 `Dumb Component` 会提高我们的组件的可复用性。

到这里思路慢慢地变得清晰了, 我们需要高阶组件帮助我们从 `context` 取数据, 我们也需要写 `Dumb` 组件帮助我们提高组件的复用性。所以我们尽量多地写 `Dumb` 组件,

然后用高阶组件把它们包装一层，高阶组件和 context 打交道，把里面数据取出来通过 `props` 传给 Dumb 组件。



我们把这个高阶组件起名字叫 `connect`，因为它把 Dumb 组件和 context 连接 (`connect`) 起来了：

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'

export connect = (WrappedComponent) => {
  class Connect extends Component {
    static contextTypes = {
      store: PropTypes.object
    }

    // TODO: 如何从 store 取数据?

    render () {
      return <WrappedComponent />
    }
  }

  return Connect
}
```

`connect` 函数接受一个组件 `WrappedComponent` 作为参数，把这个组件包含在一个新的组件 `Connect` 里面，`Connect` 会去 `context` 里面取出 `store`。现在要把 `store` 里面的数据取出来通过 `props` 传给 `WrappedComponent`。

但是每个传进去的组件需要 store 里面的数据都不一样的，所以除了给高阶组件传入 Dumb 组件以外，还需要告诉高级组件我们需要什么数据，高阶组件才能正确地去取数据。为了解决这个问题，我们可以给高阶组件传入类似下面这样的函数：

```
const mapStateToProps = (state) => {
  return {
    themeColor: state.themeColor,
    themeName: state.themeName,
    fullName: `${state.firstName} ${state.lastName}`
    ...
  }
}
```

这个函数会接受 `store.getState()` 的结果作为参数，然后返回一个对象，这个对象是根据 `state` 生成的。`mapStateToProps` 相当于告知了 `Connect` 应该如何去 `store` 里面取数据，然后可以把这个函数的返回结果传给被包装的组件：

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'

export const connect = (mapStateToProps) => (WrappedComponent) => {
  class Connect extends Component {
    static contextTypes = {
      store: PropTypes.object
    }

    render () {
      const { store } = this.context
      let stateProps = mapStateToProps(store.getState())
      // {...stateProps} 意思是把这个对象里面的属性全部通过 'props' 方式传递进去
      return <WrappedComponent {...stateProps} />
    }
  }

  return Connect
}
```

`connect` 现在是接受一个参数 `mapStateToProps`，然后返回一个函数，这个返回的函数才是高阶组件。它会接受一个组件作为参数，然后用 `Connect` 把组件包装以后再返回。`connect` 的用法是：

```
...
const mapStateToProps = (state) => {
  return {
    themeColor: state.themeColor
  }
}
```

```
Header = connect(mapStateToProps)(Header)
```

```
...
```

有些朋友可能会问为什么不直接 `const connect = (mapStateToProps, WrappedComponent)`，而是要额外返回一个函数。这是因为 React-redux 就是这么设计的，而个人观点认为这是一个 React-redux 设计上的缺陷，这里有机会会在关于函数编程的章节再给大家科普，这里暂时不深究了。

我们把上面 `connect` 的函数代码单独分离到一个模块当中，在 `src/` 目录下新建一个 `react-redux.js`，把上面的 `connect` 函数的代码复制进去，然后就可以在 `src/Header.js` 里面使用了：

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'
import { connect } from './react-redux'

class Header extends Component {
  static propTypes = {
    themeColor: PropTypes.string
  }

  render () {
    return (
      <h1 style={{ color: this.props.themeColor }}>React.js 小书</h1>
    )
  }
}

const mapStateToProps = (state) => {
  return {
    themeColor: state.themeColor
  }
}
Header = connect(mapStateToProps)(Header)

export default Header
```

可以看到 `Header` 删掉了大部分关于 `context` 的代码，它除了 `props` 什么也不依赖，它是一个 Pure Component，然后通过 `connect` 取得数据。我们不需要知道 `connect` 是怎么和 `context` 打交道的，只要传一个 `mapStateToProps` 告诉它应该怎么取数据就可以了。同样的方式修改 `src/Content.js`：

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'
import ThemeSwitch from './ThemeSwitch'
import { connect } from './react-redux'
```

```

class Content extends Component {
  static propTypes = {
    themeColor: PropTypes.string
  }

  render () {
    return (
      <div>
        <p style={{ color: this.props.themeColor }}>React.js 小书内容</p>
        <ThemeSwitch />
      </div>
    )
  }
}

const mapStateToProps = (state) => {
  return {
    themeColor: state.themeColor
  }
}
Content = connect(mapStateToProps)(Content)

export default Content

```

`connect` 还没有监听数据变化然后重新渲染，所以现在点击按钮只有按钮会变颜色。

我们给 `connect` 的高阶组件增加监听数据变化重新渲染的逻辑，稍微重构一下

`connect` :

```

export const connect = (mapStateToProps) => (WrappedComponent) => {
  class Connect extends Component {
    static contextTypes = {
      store: PropTypes.object
    }

    constructor () {
      super()
      this.state = { allProps: {} }
    }

    componentWillMount () {
      const { store } = this.context
      this._updateProps()
      store.subscribe(() => this._updateProps())
    }

    _updateProps () {
      const { store } = this.context
      let stateProps = mapStateToProps(store.getState(), this.props) // 额外传入 p
      this.setState({
        allProps: { // 整合普通的 props 和从 state 生成的 props

```

```

    ...stateProps,
    ...this.props
  }
})
}

render () {
  return <WrappedComponent {...this.state.allProps} />
}
}

return Connect
}

```

我们在 `Connect` 组件的 `constructor` 里面初始化了 `state.allProps`，它是一个对象，用来保存需要传给被包装组件的所有的参数。生命周期 `componentWillMount` 会调用调用 `_updateProps` 进行初始化，然后通过 `store.subscribe` 监听数据变化重新调用 `_updateProps`。

为了让 `connect` 返回新组件和被包装的组件使用参数保持一致，我们会把所有传给 `Connect` 的 `props` 原封不动地传给 `WrappedComponent`。所以在 `_updateProps` 里面会把 `stateProps` 和 `this.props` 合并到 `this.state.allProps` 里面，再通过 `render` 方法把所有参数都传给 `WrappedComponent`。

`mapStateToProps` 也发生点变化，它现在可以接受两个参数了，我们会把传给 `Connect` 组件的 `props` 参数也传给它，那么它生成的对象配置性就更强了，我们可以根据 `store` 里面的 `state` 和外界传入的 `props` 生成我们想传给被包装组件的参数。

现在已经很不错了，`Header.js` 和 `Content.js` 的代码都大大减少了，并且这两个组件 `connect` 之前都是 Dumb 组件。接下来会继续重构 `ThemeSwitch`。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：39. 动手实现 React-redux \(四\) : mapDispatchToProps](#)

[上一节：37. 动手实现 React-redux \(二\) : 结合 context 和 store](#)

React.js 小书

[<-- 返回首页](#)

## 39. 动手实现 React-redux (四) : mapDispatchToProps

- 作者: [胡子大哈](#)
- 原文链接: <http://huziketang.com/books/react/lesson39>
- 转载请注明出处, 保留原文链接和作者信息。

(本文未审核)

在重构 `ThemeSwitch` 的时候我们发现, `ThemeSwitch` 除了需要 `store` 里面的数据以外, 还需要 `store` 来 `dispatch`:

```
...
// dispatch action 去改变颜色
handleSwitchColor (color) {
  const { store } = this.context
  store.dispatch({
    type: 'CHANGE_COLOR',
    themeColor: color
  })
}
...
```

目前版本的 `connect` 是达不到这个效果的, 我们需要改进它。

想一下, 既然可以通过给 `connect` 函数传入 `mapStateToProps` 来告诉它如何获取、整合状态, 我们也可以想到, 可以给它传入另外一个参数来告诉它我们的组件需要如何触发 `dispatch`。我们把这个参数叫 `mapDispatchToProps`:

```
const mapDispatchToProps = (dispatch) => {
  return {
    onSwitchColor: (color) => {
      dispatch({ type: 'CHANGE_COLOR', themeColor: color })
    }
  }
}
```

和 `mapStateToProps` 一样, 它返回一个对象, 这个对象内容会同样被 `connect` 当作是 `props` 参数传给被包装的组件。不一样的是, 这个函数不是接受 `state` 作为参

数，而是 `dispatch`，你可以在返回的对象内部定义一些函数，这些函数会用到 `dispatch` 来触发特定的 `action`。

调整 `connect` 让它能接受这样的 `mapDispatchToProps`：

```
export const connect = (mapStateToProps, mapDispatchToProps) => (WrappedComponent)
  class Connect extends Component {
    static contextTypes = {
      store: PropTypes.object
    }

    constructor () {
      super()
      this.state = {
        allProps: {}
      }
    }

    componentWillMount () {
      const { store } = this.context
      this._updateProps()
      store.subscribe(() => this._updateProps())
    }

    _updateProps () {
      const { store } = this.context
      let stateProps = mapStateToProps
        ? mapStateToProps(store.getState(), this.props)
        : {} // 防止 mapStateToProps 没有传入
      let dispatchProps = mapDispatchToProps
        ? mapDispatchToProps(store.dispatch, this.props)
        : {} // 防止 mapDispatchToProps 没有传入
      this.setState({
        allProps: {
          ...stateProps,
          ...dispatchProps,
          ...this.props
        }
      })
    }

    render () {
      return <WrappedComponent {...this.state.allProps} />
    }
  }
  return Connect
}
```

在 `_updateProps` 内部，我们把 `store.dispatch` 作为参数传给 `mapDispatchToProps`，它会返回一个对象 `dispatchProps`。接着把 `stateProps`、`dispatchProps`、`this.props`

三者合并到 `this.state.allProps` 里面去，这三者的内容都会在 `render` 函数内全部传给被包装的组件。

另外，我们稍微调整了一下，在调用 `mapStateToProps` 和 `mapDispatchToProps` 之前做判断，让这两个参数都是可以缺省的，这样即使不传这两个参数程序也不会报错。

这时候我们就可以重构 `ThemeSwitch`，让它摆脱 `store.dispatch`：

```

import React, { Component } from 'react'
import PropTypes from 'prop-types'
import { connect } from './react-redux'

class ThemeSwitch extends Component {
  static propTypes = {
    themeColor: PropTypes.string,
    onSwitchColor: PropTypes.func
  }

  handleSwitchColor (color) {
    if (this.props.onSwitchColor) {
      this.props.onSwitchColor(color)
    }
  }

  render () {
    return (
      <div>
        <button
          style={{ color: this.props.themeColor }}
          onClick={this.handleSwitchColor.bind(this, 'red')}>Red</button>
        <button
          style={{ color: this.props.themeColor }}
          onClick={this.handleSwitchColor.bind(this, 'blue')}>Blue</button>
      </div>
    )
  }
}

const mapStateToProps = (state) => {
  return {
    themeColor: state.themeColor
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    onSwitchColor: (color) => {
      dispatch({ type: 'CHANGE_COLOR', themeColor: color })
    }
  }
}

ThemeSwitch = connect(mapStateToProps, mapDispatchToProps)(ThemeSwitch)

```

```
export default ThemeSwitch
```

光看 `ThemeSwitch` 内部，是非常清爽干净的，只依赖外界传进来的 `themeColor` 和 `onSwitchColor`。但是 `ThemeSwitch` 内部并不知道这两个参数其实都是我们去 `store` 里面取的，它是 Dumb 的。这时候这三个组件的重构都已经完成了，代码大大减少、不依赖 `context`，并且功能和原来一样。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：40. 动手实现 React-redux \(五\) : Provider](#)

[上一节：38. 动手实现 React-redux \(三\) : connect 和 mapStateToProps](#)

React.js 小书

[<-- 返回首页](#)

## 40. 动手实现 React-redux（五）：Provider

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson40>
- 转载请注明出处，保留原文链接和作者信息。

(本文未审核)

我们要把 context 相关的代码从所有业务组件中清除出去，现在的代码里面还有一个地方是被污染的。那就是 `src/index.js` 里面的 `Index`：

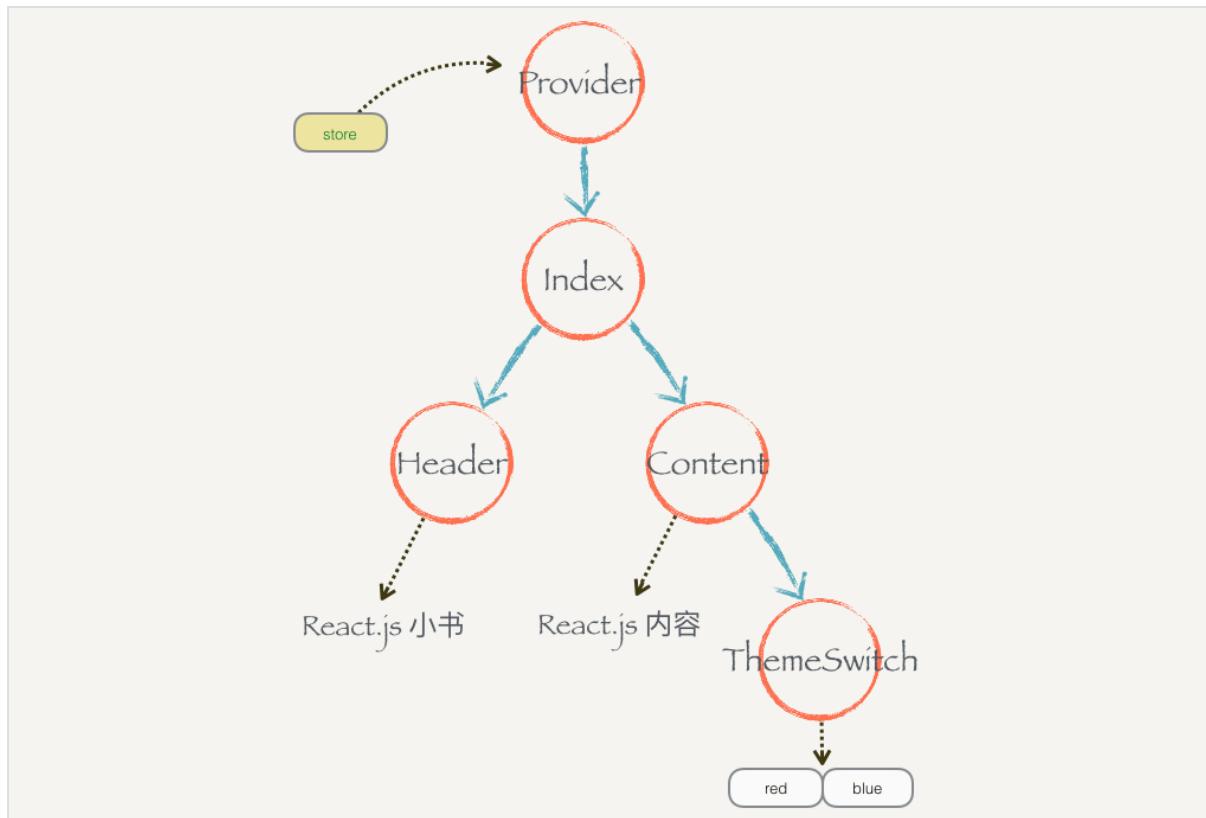
```
...
class Index extends Component {
  static childContextTypes = {
    store: PropTypes.object
  }

  getChildContext () {
    return { store }
  }

  render () {
    return (
      <div>
        <Header />
        <Content />
      </div>
    )
  }
}
...
```

其实它要用 context 就是因为要把 `store` 存放到里面，好让子组件 `connect` 的时候能够取到 `store`。我们可以额外构建一个组件来做这种脏活，然后让这个组件成为组件树的根节点，那么它的子组件都可以获取到 context 了。

我们把这个组件叫 `Provider`，因为它提供 (provide) 了 `store`：



在 `src/react-redux.js` 新增代码：

```
export class Provider extends Component {
  static propTypes = {
    store: PropTypes.object,
    children: PropTypes.any
  }

  static childContextTypes = {
    store: PropTypes.object
  }

  getChildContext () {
    return {
      store: this.props.store
    }
  }

  render () {
    return (
      <div>{this.props.children}</div>
    )
  }
}
```

`Provider` 做的事情也很简单，它就是一个容器组件，会把嵌套的内容原封不动作为自己的子组件渲染出来。它还会把外界传给它的 `props.store` 放到 `context`，这样子组件 `connect` 的时候都可以获取到。

可以用它来重构我们的 `src/index.js` :

```
...
// 头部引入 Provider
import { Provider } from './react-redux'
...

// 删除 Index 里面所有关于 context 的代码
class Index extends Component {
  render () {
    return (
      <div>
        <Header />
        <Content />
      </div>
    )
  }
}

// 把 Provider 作为组件树的根节点
ReactDOM.render(
  <Provider store={store}>
    <Index />
  </Provider>,
  document.getElementById('root')
)
```

这样我们就把所有关于 `context` 的代码从组件里面删除了。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：41. 动手实现 React-redux \(六\) : React-redux 总结](#)

[上一节：39. 动手实现 React-redux \(四\) : mapDispatchToProps](#)

React.js 小书

[<-- 返回首页](#)

## 41. 动手实现 React-redux (六) : React-redux 总结

- 作者: [胡子大哈](#)
- 原文链接: <http://huziketang.com/books/react/lesson41>
- 转载请注明出处, 保留原文链接和作者信息。

(本文未审核)

到这里大家已经掌握了 React-redux 的基本用法和概念, 并且自己动手实现了一个 React-redux, 我们回顾一下这几节都干了什么事情。

React.js 除了状态提升以外并没有更好的办法帮我们解决组件之间共享状态的问题, 而使用 context 全局变量让程序不可预测。通过 Redux 的章节, 我们知道 store 里面的内容是不可以随意修改的, 而是通过 dispatch 才能变更里面的 state。所以我们尝试把 store 和 context 结合起来使用, 可以兼顾组件之间共享状态问题和共享状态可能被任意修改的问题。

第一个版本的 store 和 context 结合有诸多缺陷, 有大量的重复逻辑和对 context 的依赖性过强。我们尝试通过构建一个高阶组件 `connect` 函数的方式, 把所有的重复逻辑和对 context 的依赖放在里面 `connect` 函数里面, 而其他组件保持 Pure (Dumb) 的状态, 让 `connect` 跟 context 打交道, 然后通过 `props` 把参数传给普通的组件。

而每个组件需要的数据和需要触发的 action 都不一样, 所以调整 `connect`, 让它可以接受两个参数 `mapStateToProps` 和 `mapDispatchToProps`, 分别用于告诉 `connect` 这个组件需要什么数据和需要触发什么 action。

最后为了把所有关于 context 的代码完全从我们业务逻辑里面清除掉, 我们构建了一个 `Provider` 组件。`Provider` 作为所有组件树的根节点, 外界可以通过 `props` 给它提供 `store`, 它会把 `store` 放到自己的 context 里面, 好让子组件 `connect` 的时候都能够获取到。

这几节的成果就是 `react-redux.js` 这个文件里面的两个内容: `connect` 函数和 `Provider` 容器组件。这就是 React-redux 的基本内容, 当然它是一个残疾版本的 React-redux, 很多地方需要完善。例如上几节提到的性能问题, 现在不相关的数据变化的时候其实所有组件都会重新渲染的, 这个性能优化留给读者做练习。

通过这种方式大家不仅仅知道了 React-redux 的基础概念和用法，而且还知道这些概念到底是解决什么问题，为什么 React-redux 这么奇怪，为什么要 connect，为什么要 mapStateToProps 和 mapDispatchToProps，什么是 Provider，我们通过解决一个个问题就知道它们到底为什么要这么设计的了。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

---

[下一节：42. 使用真正的 Redux 和 React-redux](#)

[上一节：40. 动手实现 React-redux \(五\) : Provider](#)

React.js 小书

[--> 返回首页](#)

## 42. 使用真正的 Redux 和 React-redux

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson42>
- 转载请注明出处，保留原文链接和作者信息。

(本文未审核)

现在 `make-react-redux` 工程代码中的 Redux 和 React-redux 都是我们自己写的，现在让我们来使用真正的官方版本的 Redux 和 React-redux。

在工程目录下使用 `npm` 安装 Redux 和 React-redux 模块：

```
npm install redux react-redux --save
```

把 `src/` 目录下 `Header.js`、`ThemeSwitch.js`、`Content.js` 的模块导入中的：

```
import { connect } from './react-redux'
```

改成：

```
import { connect } from 'react-redux'
```

也就是本来从本地 `./react-redux` 导入的 `connect` 改成从第三方 `react-redux` 模块中导入。

修改 `src/index.js`，把前面部分的代码调整为：

```
import React, { Component } from 'react'
import ReactDOM from 'react-dom'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import Header from './Header'
import Content from './Content'
import './index.css'

const themeReducer = (state, action) => {
  if (!state) return {
    themeColor: 'red'
```

```

    }
    switch (action.type) {
      case 'CHANGE_COLOR':
        return { ...state, themeColor: action.themeColor }
      default:
        return state
    }
}

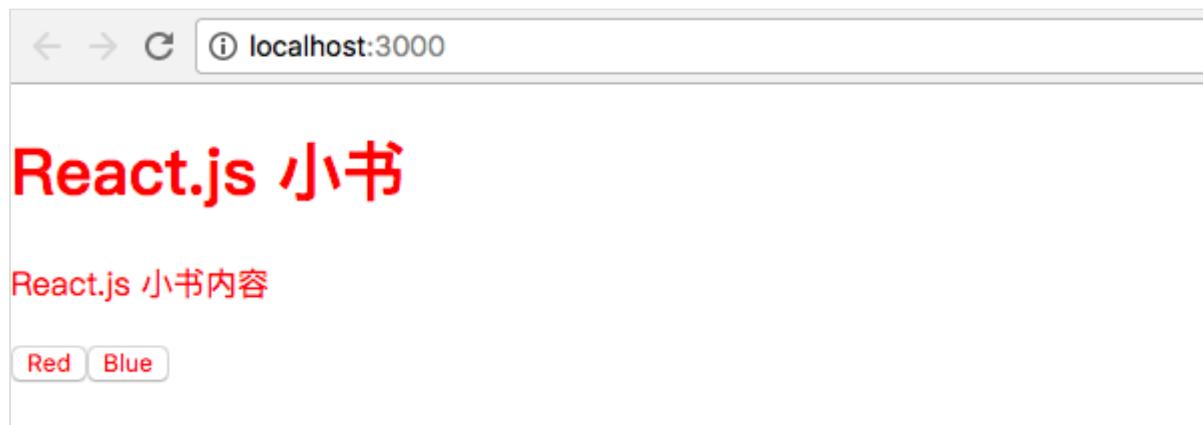
const store = createStore(themeReducer)

...

```

我们删除了自己写的 `createStore`，改成使用第三方模块 `redux` 的 `createStore`；  
`Provider` 本来从本地的 `./react-redux` 引入，改成从第三方 `react-redux` 模块中引入。其余代码保持不变。

接着删除 `src/react-redux.js`，它的已经用处不大了。最后启动工程 `npm start`：



可以看到我们原来的业务代码其实都没有太多的改动，实际上我们实现的 `redux` 和 `react-redux` 和官方版本在该场景的用法上是兼容的。接下来的章节我们都会使用官方版本的 `redux` 和 `react-redux`。

## 课后练习

- [React-redux 实现用户列表的显示、增加、删除](#)

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

下一节：43. Smart 组件 vs Dumb 组件

上一节：41. 动手实现 React-redux（六）：React-redux 总结



React.js 小书

[<-- 返回首页](#)

## 43. Smart 组件 vs Dumb 组件

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson43>
- 转载请注明出处，保留原文链接和作者信息。

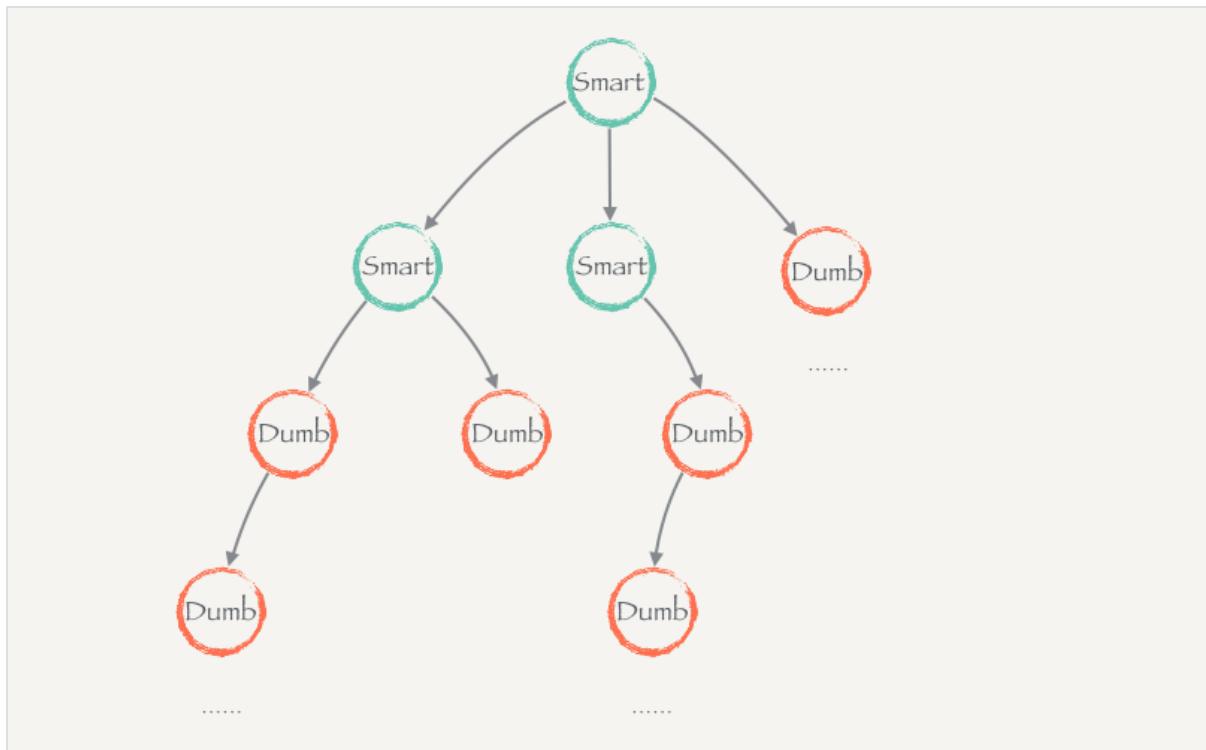
(本文未审核)

大家已经知道，只会接受 `props` 并且渲染确定结果的组件我们把它叫做 Dumb 组件，这种组件只关心一件事情 — 根据 `props` 进行渲染。

Dumb 组件最好不要依赖除了 React.js 和 Dumb 组件以外的内容。它们不要依赖 Redux 不要依赖 React-redux。这样的组件的可复用性是最好的，其他人可以安心地使用而不用怕会引入什么奇奇怪怪的东西。

当我们拿到一个需求开始划分组件的时候，要认真考虑每个被划分成组件的单元到底会不会被复用。如果这个组件可能会在多处被使用到，那么我们就把它做成 Dumb 组件。

我们可能拆分了一堆 Dumb 组件出来。但是单纯靠 Dumb 是没有办法构建应用程序的，因为它们实在太“笨”了，对数据的力量一无所知。所以还有一种组件，它们非常聪明（smart），城府很深精通算计，我们叫它们 Smart 组件。它们专门做数据相关的应用逻辑，和各种数据打交道、和 Ajax 打交道，然后把数据通过 `props` 传递给 Dumb，它们带领着 Dumb 组件完成了复杂的应用程序逻辑。



Smart 组件不用考虑太多复用性问题，它们就是用来执行特定应用逻辑的。Smart 组件可能组合了 Smart 组件和 Dumb 组件；但是 Dumb 组件尽量不要依赖 Smart 组件。因为 Dumb 组件目的之一是为了复用，一旦它引用了 Smart 组件就相当于带入了一堆应用逻辑，导致它无法复用，所以尽量不要干这种事情。一旦一个可复用的 Dumb 组件之下引用了一个 Smart 组件，就相当于污染了这个 Dumb 组件树。如果一个组件是 Dumb 的，那么它的子组件们都应该是 Dumb 的才对。

## 划分 Smart 和 Dumb 组件

知道了组件有这两种分类以后，我们来重新审视一下之前的 `make-react-redux` 工程里面的组件，例如 `src/Header.js`：

```

import React, { Component } from 'react'
import PropTypes from 'prop-types'
import { connect } from 'react-redux'

class Header extends Component {
  static propTypes = {
    themeColor: PropTypes.string
  }

  render () {
    return (
      <h1 style={{ color: this.props.themeColor }}>React.js 小书</h1>
    )
  }
}

const mapStateToProps = (state) => {

```

```

    return {
      themeColor: state.themeColor
    }
}
Header = connect(mapStateToProps)(Header)

export default Header

```

这个组件到底是 Smart 还是 Dumb 组件？这个文件其实依赖了 react-redux，别人使用的时候其实会带上这个依赖，所以这个组件不能叫 Dumb 组件。但是你观察一下，这个组件在 connect 之前它却是 Dumb 的，就是因为 connect 了导致它和 context 扯上了关系，导致它变 Smart 了，也使得这个组件没有了很好的复用性。

为了解决这个问题，我们把 Smart 和 Dumb 组件分开到两个不同的目录，不再在 Dumb 组件内部进行 connect，在 src/ 目录下新建两个文件夹 components/ 和 containers/：

```

src/
  components/
  containers/

```

我们规定：所有的 Dumb 组件都放在 components/ 目录下，所有的 Smart 的组件都放在 containers/ 目录下，这是一种约定俗成的规则。

删除 src/Header.js，新增 src/components/Header.js：

```

import React, { Component } from 'react'
import PropTypes from 'prop-types'

export default class Header extends Component {
  static propTypes = {
    themeColor: PropTypes.string
  }

  render () {
    return (
      <h1 style={{ color: this.props.themeColor }}>React.js 小书</h1>
    )
  }
}

```

现在 src/components/Header.js 毫无疑问是一个 Dumb 组件，它除了依赖 React.js 什么都不依赖。我们新建 src/container/Header.js，这是一个与之对应的 Smart 组件：

```

import { connect } from 'react-redux'
import Header from '../components/Header'

const mapStateToProps = (state) => {
  return {
    themeColor: state.themeColor
  }
}
export default connect(mapStateToProps)(Header)

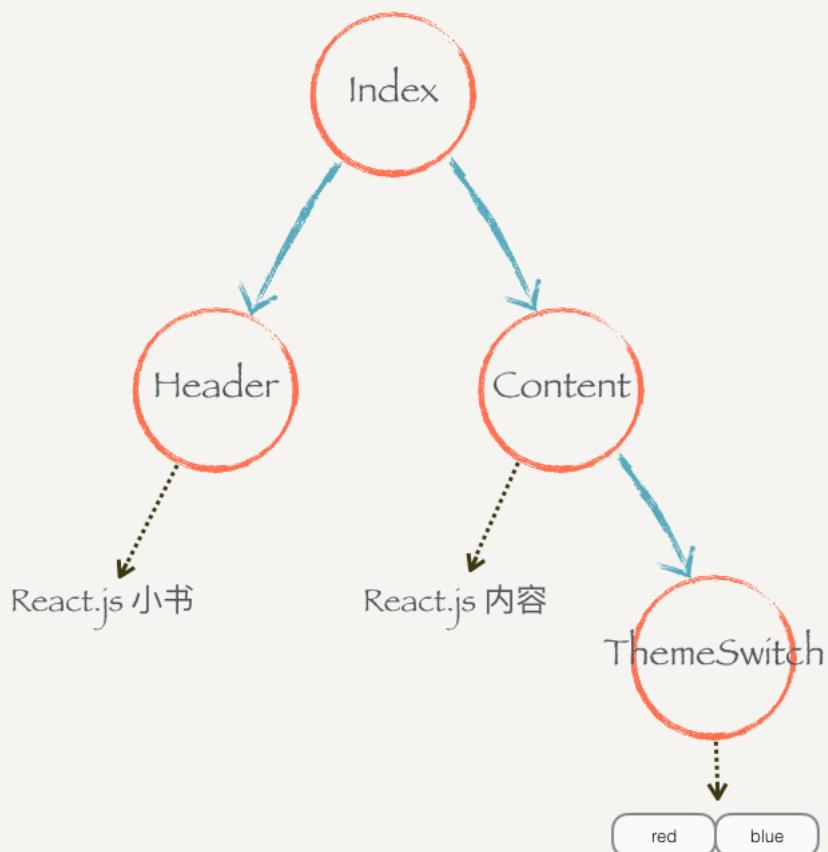
```

它会从导入 Dumb 的 `Header.js` 组件，进行 `connect` 一番变成 Smart 组件，然后把它导出模块。

这样我们就把 Dumb 组件抽离出来了，现在 `src/components/Header.js` 可复用性非常强，别的同事可以随意用它。而 `src/containers/Header.js` 则是跟业务相关的，我们只用在特定的应用场景下。我们可以继续用这种方式来重构其他组件。

## 组件划分原则

接下来的情况就有点意思了，可以趁机给大家讲解一下组件划分的一些原则。我们看看这个应用原来的组件树：



对于 `Content` 这个组件，可以看到它是依赖 `ThemeSwitch` 组件的，这就需要好好思考一下了。我们分两种情况来讨论：`Content` 不复用和可复用。

## Content 不复用

如果产品场景并没有要求说 Content 需要复用，它只是在特定业务需要而已。那么没有必要把 Content 做成 Dumb 组件了，就让它成为一个 Smart 组件。因为 Smart 组件是可以使用 Smart 组件的，所以 Content 可以使用 Dumb 的 ThemeSwitch 组件 connect 的结果。

新建一个 `src/components/ThemeSwitch.js`：

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'

export default class ThemeSwitch extends Component {
  static propTypes = {
    themeColor: PropTypes.string,
    onSwitchColor: PropTypes.func
  }

  handleSwitchColor (color) {
    if (this.props.onSwitchColor) {
      this.props.onSwitchColor(color)
    }
  }

  render () {
    return (
      <div>
        <button
          style={{ color: this.props.themeColor }}
          onClick={this.handleSwitchColor.bind(this, 'red')}>Red</button>
        <button
          style={{ color: this.props.themeColor }}
          onClick={this.handleSwitchColor.bind(this, 'blue')}>Blue</button>
      </div>
    )
  }
}
```

这是一个 Dumb 的 ThemeSwitch。新建一个 `src/containers/ThemeSwitch.js`：

```
import { connect } from 'react-redux'
import ThemeSwitch from '../components/ThemeSwitch'

const mapStateToProps = (state) => {
  return {
    themeColor: state.themeColor
  }
}
const mapDispatchToProps = (dispatch) => {
  return {
```

```

    onSwitchColor: (color) => {
      dispatch({ type: 'CHANGE_COLOR', themeColor: color })
    }
  }
}

export default connect(mapStateToProps, mapDispatchToProps)(ThemeSwitch)

```

这是一个 Smart 的 `ThemeSwitch`。然后用一个 Smart 的 `Content` 去使用它，新建 `src/containers/Content.js`：

```

import React, { Component } from 'react'
import PropTypes from 'prop-types'
import ThemeSwitch from './ThemeSwitch'
import { connect } from 'react-redux'

class Content extends Component {
  static propTypes = {
    themeColor: PropTypes.string
  }

  render () {
    return (
      <div>
        <p style={{ color: this.props.themeColor }}>React.js 小书内容</p>
        <ThemeSwitch />
      </div>
    )
  }
}

const mapStateToProps = (state) => {
  return {
    themeColor: state.themeColor
  }
}
export default connect(mapStateToProps)(Content)

```

删除 `src/ThemeSwitch.js` 和 `src/Content.js`，在 `src/index.js` 中直接使用 Smart 组件：

```

...
import Header from './containers/Header'
import Content from './containers/Content'
...

```

这样就把这种业务场景下的 Smart 和 Dumb 组件分离开来了：

```

src
├── components
│   ├── Header.js
│   └── ThemeSwitch.js
└── containers
    ├── Content.js
    ├── Header.js
    └── ThemeSwitch.js
index.js

```

### Content 可复用

如果产品场景要求 `Content` 可能会被复用，那么 `Content` 就要是 Dumb 的。那么 `Content` 之下的子组件 `ThemeSwitch` 就一定要是 Dumb，否则 `Content` 就没法复用了。这就意味着 `ThemeSwitch` 不能 `connect`，即使你 `connect` 了，`Content` 也不能使用你 `connect` 的结果，因为 `connect` 的结果是个 Smart 组件。

这时候 `ThemeSwitch` 的数据、`onSwitchColor` 函数只能通过它的父组件传进来，而不是通过 `connect` 获得。所以只能让 `Content` 组件去 `connect`，然后让它把数据、函数传给 `ThemeSwitch`。

这种场景下的改造留给大家做练习，最后的结果应该是：

```

src
├── components
│   ├── Header.js
│   └── Content.js
│       └── ThemeSwitch.js
└── containers
    ├── Header.js
    └── Content.js
index.js

```

可以看到对复用性的需求不同，会导致我们划分组件的方式不同。

## 总结

根据是否需要高度的复用性，把组件划分为 Dumb 和 Smart 组件，约定俗成地把它们分别放到 `components` 和 `containers` 目录下。

Dumb 基本只做一件事情 — 根据 `props` 进行渲染。而 Smart 则是负责应用的逻辑、数据，把所有相关的 Dumb (Smart) 组件组合起来，通过 `props` 控制它们。

Smart 组件可以使用 Smart、Dumb 组件；而 Dumb 组件最好只使用 Dumb 组件，否则它的复用性就会丧失。

要根据应用场景不同划分组件，如果一个组件并不需要太强的复用性，直接让它成为 Smart 即可；否则就让它成为 Dumb 组件。

还有一点要注意，Smart 组件并不意味着完全不能复用，Smart 组件的复用性是依赖场景的，在特定的应用场景下是当然是可以复用 Smart 的。而 Dumb 则是可以跨应用场景复用，Smart 和 Dumb 都可以复用，只是程度、场景不一样。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：44. 实战分析：评论功能（七）](#)

[上一节：42. 使用真正的 Redux 和 React-redux](#)

React.js 小书

[<-- 返回首页](#)

## 44. 实战分析：评论功能（七）

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson44>
- 转载请注明出处，保留原文链接和作者信息。

(本文未审核)

从本节开始，我们开始用 Redux、React-redux 来重构第二阶段的评论功能。产品需求跟之前一样，但是会用 Redux、React-redux 来帮助管理应用状态，而不是“状态提升”。让整个应用更加接近真实的工程。

大家可以在第二阶段的代码上进行修改 [comment-app2](#)（非高阶组件版本）。如果已经忘了第二阶段评论功能的同学可以先简单回顾一下它的功能需求，[实战分析：评论功能（四）](#)。第一、二、三阶段的实战代码都可以在这里找到：[react-naive-book-examples](#)。

我们首先安装好依赖，现在 comment-app2 需要依赖 Redux、React-redux 了，进入工程目录执行命令安装依赖：

```
npm install redux react-redux --save
```

然后我们二话不说先在 `src` 下建立三个空目录：`components`、`containers`、`reducers`。

### 构建评论的 reducer

我们之前的 reducer 都是直接写在 `src/index.js` 文件里面，这是一个不好的做法。因为随着应用越来越复杂，可能需要更多的 reducer 来帮助我们管理应用（这里后面的章节会有所提及）。所以最好还是把所有 reducer 抽出来放在一个目录下 `src/reducers`。

对于评论功能其实还是比较简单的，回顾一下我们在[状态提升](#)章节里面不断提升的状态是什么？其实评论功能的组件之间共享的状态只有 `comments`。我们可以直接只在 `src/reducers` 新建一个 reducer `comments.js` 来对它进行管理。

思考一下评论功能对于评论有什么操作？想清楚我们才能写好 reducer，因为 reducer 就是用来描述数据的形态和相应的变更。**新增和删除评论**这两个操作是最明

显的，大家应该都能够轻易想到。还有一个，我们的评论功能其实会从 `LocalStorage` 读取数据，读取数据以后其实需要保存到应用状态中。所以我们还有一个**初始化评论**的操作。所以目前能想到的就是三个操作：

```
// action types
const INIT_COMMENTS = 'INIT_COMMENTS'
const ADD_COMMENT = 'ADD_COMMENT'
const DELETE_COMMENT = 'DELETE_COMMENT'
```

我们用三个常量来存储 `action.type` 的类型，这样以后我们修改起来就会更方便一些。根据这三个操作编写 `reducer`：

```
// reducer
export default function (state, action) {
  if (!state) {
    state = { comments: [] }
  }
  switch (action.type) {
    case INIT_COMMENTS:
      // 初始化评论
      return { comments: action.comments }
    case ADD_COMMENT:
      // 新增评论
      return {
        comments: [...state.comments, action.comment]
      }
    case DELETE_COMMENT:
      // 删除评论
      return {
        comments: [
          ...state.comments.slice(0, action.commentIndex),
          ...state.comments.slice(action.commentIndex + 1)
        ]
      }
    default:
      return state
  }
}
```

我们只存储了一个 `comments` 的状态，初始化为空数组。当遇到 `INIT_COMMENTS` 的 `action` 的时候，会新建一个对象，然后用 `action.comments` 覆盖里面的 `comments` 属性。这就是初始化评论操作。

同样新建评论操作 `ADD_COMMENT` 也会新建一个对象，然后新建一个数组，接着把原来 `state.comments` 里面的内容全部拷贝到新的数组当中，最后在新的数组后面追加 `action.comment`。这样就相当新的数组会比原来的多一条评论。（这里不要担心数组拷贝的性能问题，`[...state.comments]` 是浅拷贝，它们拷贝的都是对象引用而已。）

对于删除评论，其实我们需要做的是新建一个删除了特定下标的内容的数组。我们知道数组 `slice(from, to)` 会根据你传进去的下标拷贝特定范围的内容放到新数组里面。所以我们可以利用 `slice` 把原来评论数组中 `action.commentIndex` 下标之前的内容拷贝到一个数组当中，把 `action.commentIndex` 坐标之后到内容拷贝到另外一个数组当中。然后把两个数组合并起，就相当于“删除”了 `action.commentIndex` 的评论了。

这样就写好了评论相关的 reducer。

### action creators

之前我们使用 `dispatch` 的时候，都是直接手动构建对象：

```
dispatch({ type: 'INIT_COMMENTS', comments })
```

每次都要写 `type` 其实挺麻烦的，而且还要去记忆 action type 的名字也是一种负担。我们可以把 action 封装到一种函数里面，让它们去帮助我们去构建这种 action，我们把它叫做 action creators。

```
// action creators
export const initComments = (comments) => {
  return { type: INIT_COMMENTS, comments }
}

export const addComment = (comment) => {
  return { type: ADD_COMMENT, comment }
}

export const deleteComment = (commentIndex) => {
  return { type: DELETE_COMMENT, commentIndex }
}
```

所谓 action creators 其实就是返回 action 的函数，这样我们 `dispatch` 的时候只需要传入数据就可以了：

```
dispatch(initComments(comments))
```

action creators 还有额外好处就是可以帮助我们对传入的数据做统一的处理；而且有了 action creators，代码测试起来会更方便一些。这些内容大家可以后续在实际项目当中进行体会。

整个 `src/reducers/comments.js` 的代码就是：

```

// action types
const INIT_COMMENTS = 'INIT_COMMENTS'
const ADD_COMMENT = 'ADD_COMMENT'
const DELETE_COMMENT = 'DELETE_COMMENT'

// reducer
export default function (state, action) {
  if (!state) {
    state = { comments: [] }
  }
  switch (action.type) {
    case INIT_COMMENTS:
      // 初始化评论
      return { comments: action.comments }
    case ADD_COMMENT:
      // 新增评论
      return {
        comments: [...state.comments, action.comment]
      }
    case DELETE_COMMENT:
      // 删除评论
      return {
        comments: [
          ...state.comments.slice(0, action.commentIndex),
          ...state.comments.slice(action.commentIndex + 1)
        ]
      }
    default:
      return state
  }
}

// action creators
export const initComments = (comments) => {
  return { type: INIT_COMMENTS, comments }
}

export const addComment = (comment) => {
  return { type: ADD_COMMENT, comment }
}

export const deleteComment = (commentIndex) => {
  return { type: DELETE_COMMENT, commentIndex }
}

```

有些朋友可能会发现我们的 reducer 跟网上其他的 reducer 的例子不大一样。有些人喜欢把 action 单独切出去一个目录 `actions`，让 action 和 reducer 分开。个人观点觉得这种做法可能有点过度优化了，其实多数情况下特定的 action 只会影响特定的 reducer，直接放到一起可以更加清晰地知道这个 action 其实只是会影响到什么样的 reducer。而分开会给我们维

护和理解代码带来额外不必要的负担，这有种矫枉过正的意味。但是这里没有放之四海皆准的规则，大家可以多参考、多尝试，找到适合项目需求的方案。

个人写 reducer 文件的习惯，仅供参考：

1. 定义 action types
2. 编写 reducer
3. 跟这个 reducer 相关的 action creators

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：45. 实战分析：评论功能（八）](#)

[上一节：43. Smart 组件 vs Dumb 组件](#)

React.js 小书

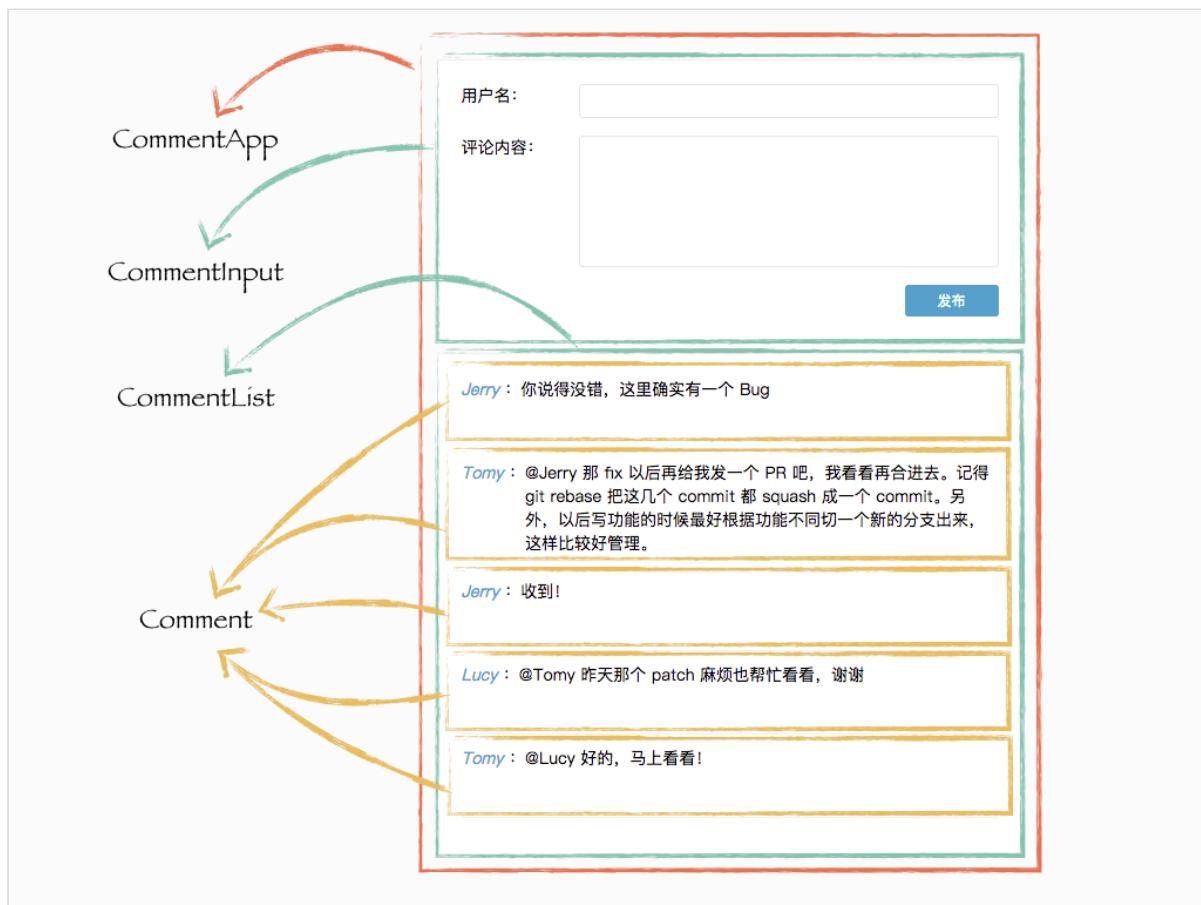
<-- 返回首页

## 45. 实战分析：评论功能（八）

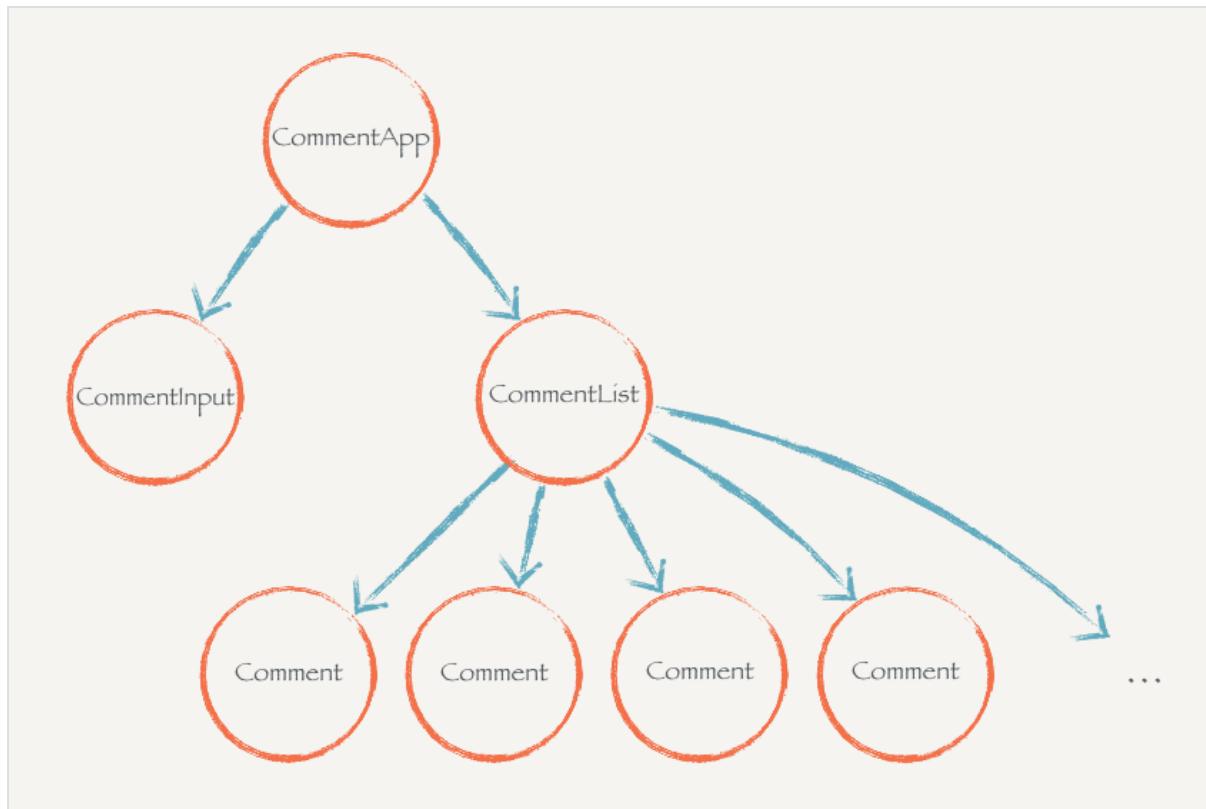
- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson45>
- 转载请注明出处，保留原文链接和作者信息。

(本文未审核)

接下可以重构组件部分的内容了。先回顾一下之前我们是怎么划分组件的：



组件树：



这样划分方式当然是没错的。但是在组件的实现上有些问题，我们之前并没有太多地考虑复用性问题。所以现在可以看看 `comment-app2` 的 `CommentInput` 组件，你会发现它里面有一些 `LocalStorage` 操作：

```

...
_loadUsername () {
  const username = localStorage.getItem('username')
  if (username) {
    this.setState({ username })
  }
}

_saveUsername (username) {
  localStorage.setItem('username', username)
}

handleUsernameBlur (event) {
  this._saveUsername(event.target.value)
}

handleUsernameChange (event) {
  this.setState({
    username: event.target.value
  })
}
...
  
```

它是一个依赖 `LocalStorage` 数据的 `Smart` 组件。如果别的地方想使用这个组件，但是数据却不是从 `LocalStorage` 里面取的，而是从服务器取的，那么这个组件就无法复用了。

所以现在需要从复用性角度重新思考如何实现和组织这些组件。假定在目前的场景下，`CommentInput`、`CommentList`、`Comment` 组件都是需要复用的，我们就要把它们做成 `Dumb` 组件。

幸运的是，我们发现其实 `CommentList` 和 `Comment` 本来就是 `Dumb` 组件，直接把它们俩移动到 `components` 目录下即可。而 `CommentInput` 就需要好好重构一下了。我们把它里面和 `LocalStorage` 操作相关的代码全部删除，让它从 `props` 获取数据，变成一个 `Dumb` 组件，然后移动到 `src/components/CommentInput.js` 文件内：

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'

export default class CommentInput extends Component {
  static propTypes = {
    username: PropTypes.any,
    onSubmit: PropTypes.func,
    onUserNameInputBlur: PropTypes.func
  }

  static defaultProps = {
    username: ''
  }

  constructor (props) {
    super(props)
    this.state = {
      username: props.username, // 从 props 上取 username 字段
      content: ''
    }
  }

  componentDidMount () {
    this.textarea.focus()
  }

  handleUsernameBlur (event) {
    if (this.props.onUserNameInputBlur) {
      this.props.onUserNameInputBlur(event.target.value)
    }
  }

  handleUsernameChange (event) {
    this.setState({
      username: event.target.value
    })
  }
}
```

```

    }

    handleContentChange (event) {
        this.setState({
            content: event.target.value
        })
    }

    handleSubmit () {
        if (this.props.onSubmit) {
            this.props.onSubmit({
                username: this.state.username,
                content: this.state.content,
                createdTime: +new Date()
            })
        }
        this.setState({ content: '' })
    }

    render () {
        // render 方法保持不变
        // ...
    }
}

```

其实改动不多。原来 `CommentInput` 需要从 `LocalStorage` 中获取 `username` 字段，现在让它从 `props` 里面去取；而原来用户名的输入框 `blur` 的时候需要保存 `username` 到 `LocalStorage` 的行为也通过 `props.onUserNameInputBlur` 传递到上层去做。现在 `CommentInput` 是一个 Dumb 组件了，它的所有渲染操作都只依赖于 `props` 来完成。

现在三个 Dumb 组件 `CommentInput`、`CommentList`、`Comment` 都已经就位了。但是单靠 Dumb 组件是没办法完成应用逻辑的，所以接下来我们要构建 Smart 组件来带领它们完成任务。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

[下一节：46. 实战分析：评论功能（九）](#)

[上一节：44. 实战分析：评论功能（七）](#)



React.js 小书

[<-- 返回首页](#)

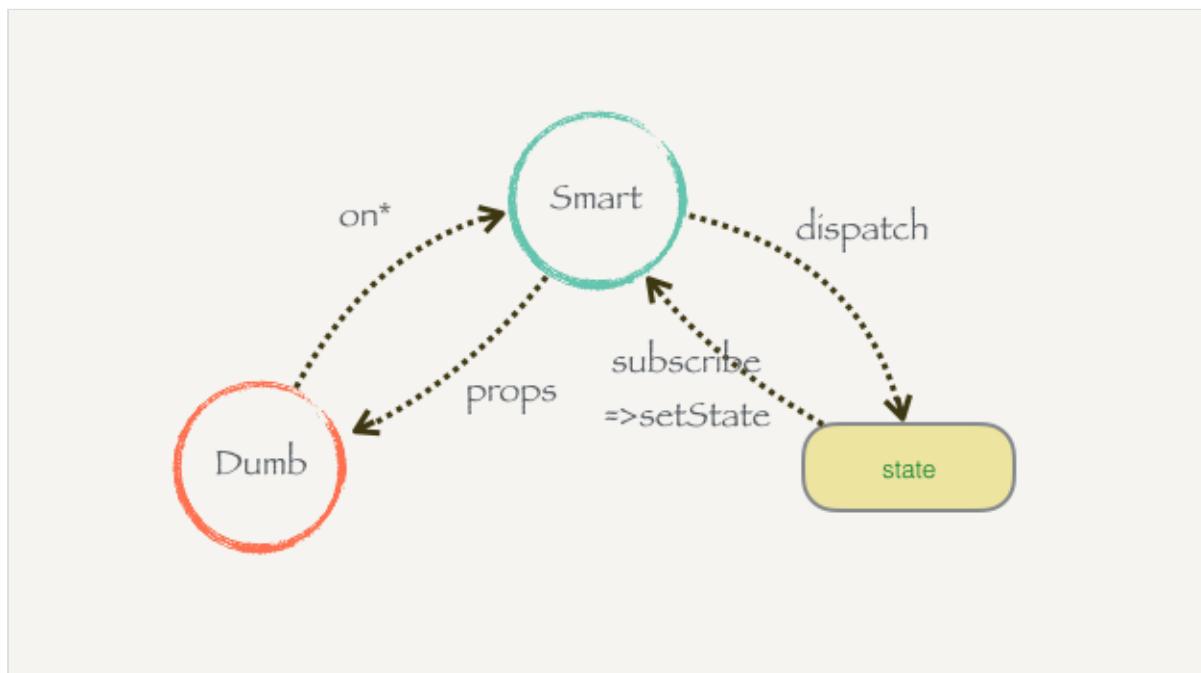
## 46. 实战分析：评论功能（九）

- 作者：[胡子大哈](#)
- 原文链接：<http://huziketang.com/books/react/lesson46>
- 转载请注明出处，保留原文链接和作者信息。

(本文未审核)

现在我们有三个 Dumb 组件，一个控制评论的 reducer。我们还缺什么？需要有人去 LocalStorage 加载数据，去控制新增、删除评论，去把数据保存到 LocalStorage 里面。之前这些逻辑我们都是零散地放在各个组件里面的（主要是 CommentApp 组件），那是因为当时我们还没对 Dumb 和 Smart 组件类型划分的认知，状态和视图之间也没有这么泾渭分明。

而现在我们知道，这些逻辑是应该放在 Smart 组件里面的：



了解 MVC、MVP 架构模式的同学应该可以类比过去，Dumb 组件就是 View（负责渲染），Smart 组件就是 Controller（Presenter），State 其实就有点类似 Model。其实不能完全类比过去，它们还是有不少差别的。但是本质上兜兜转转还是把东西分成了三层，所以说前端很喜欢炒别人早就玩烂的概念，这话果然不假。废话不多说，我们现在就把这些应用逻辑抽离到 Smart 组件里面。

Smart CommentList

对于 `CommentList` 组件，可以看到它接受两个参数：`comments` 和 `onDeleteComment`。说明需要一个 Smart 组件来负责把 `comments` 数据传给它，并且还得响应它删除评论的请求。我们新建一个 Smart 组件 `src/containers/CommentList.js` 来干这些事情：

```

import React, { Component } from 'react'
import PropTypes from 'prop-types'
import { connect } from 'react-redux'
import CommentList from '../components/CommentList'
import { initComments, deleteComment } from '../reducers/comments'

// CommentListContainer
// 一个 Smart 组件，负责评论列表数据的加载、初始化、删除评论
// 沟通 CommentList 和 state
class CommentListContainer extends Component {
  static propTypes = {
    comments: PropTypes.array,
    initComments: PropTypes.func,
    onDeleteComment: PropTypes.func
  }

  componentWillMount () {
    // componentWillMount 生命周期中初始化评论
    this._loadComments()
  }

  _loadComments () {
    // 从 LocalStorage 中加载评论
    let comments = localStorage.getItem('comments')
    comments = comments ? JSON.parse(comments) : []
    // this.props.initComments 是 connect 传进来的
    // 可以帮我们把数据初始化到 state 里面去
    this.props.initComments(comments)
  }

  handleDeleteComment (index) {
    const { comments } = this.props
    // props 是不能变的，所以这里新建一个删除了特定下标的评论列表
    const newComments = [
      ...comments.slice(0, index),
      ...comments.slice(index + 1)
    ]
    // 保存最新的评论列表到 LocalStorage
    localStorage.setItem('comments', JSON.stringify(newComments))
    if (this.props.onDeleteComment) {
      // this.props.onDeleteComment 是 connect 传进来的
      // 会 dispatch 一个 action 去删除评论
      this.props.onDeleteComment(index)
    }
  }

  render () {

```

```

    return (
      <CommentList
        comments={this.props.comments}
        onDeleteComment={this.handleDeleteComment.bind(this)} />
    )
  }
}

// 评论列表从 state.comments 中获取
const mapStateToProps = (state) => {
  return {
    comments: state.comments
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    // 提供给 CommentListContainer
    // 当从 LocalStorage 加载评论列表以后就会通过这个方法
    // 把评论列表初始化到 state 当中
    initComments: (comments) => {
      dispatch(initComments(comments))
    },
    // 删除评论
    onDeleteComment: (commentIndex) => {
      dispatch(deleteComment(commentIndex))
    }
  }
}

// 将 CommentListContainer connect 到 store
// 会把 comments、initComments、onDeleteComment 传给 CommentListContainer
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(CommentListContainer)

```

代码有点长，大家通过注释应该了解这个组件的基本逻辑。有一点要额外说明的是，我们一开始传给 `CommentListContainer` 的 `props.comments` 其实是 `reducer` 里面初始化的空的 `comments` 数组，因为还没有从 `LocalStorage` 里面取数据。

而 `CommentListContainer` 内部从 `LocalStorage` 加载 `comments` 数据，然后调用 `this.props.initComments(comments)` 会导致 `dispatch`，从而使得真正从 `LocalStorage` 加载的 `comments` 初始化到 `state` 里面去。

因为 `dispatch` 了导致 `connect` 里面的 `Connect` 包装组件去 `state` 里面取最新的 `comments` 然后重新渲染，这时候 `CommentListContainer` 才获得了有数据的 `props.comments`。

这里的逻辑有点绕，大家可以回顾一下我们之前实现的 `react-redux.js` 来体会一下。

## Smart CommentInput

对于 `CommentInput` 组件，我们可以看到它有三个参数：`username`、`onSubmit`、`onUserNameInputBlur`。我们需要一个 Smart 的组件来管理用户名在 `LocalStorage` 的加载、保存；用户还可能点击“发布”按钮，所以还需要处理评论发布的逻辑。我们新建一个 Smart 组件 `src/containers/CommentInput.js` 来干这些事情：

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'
import { connect } from 'react-redux'
import CommentInput from '../components/CommentInput'
import { addComment } from '../reducers/comments'

// CommentInputContainer
// 负责用户名的加载、保存，评论的发布
class CommentInputContainer extends Component {
  static propTypes = {
    comments: PropTypes.array,
    onSubmit: PropTypes.func
  }

  constructor () {
    super()
    this.state = { username: '' }
  }

  componentWillMount () {
    // componentWillMount 生命周期中初始化用户名
    this._loadUsername()
  }

  _loadUsername () {
    // 从 LocalStorage 加载 username
    // 然后可以在 render 方法中传给 CommentInput
    const username = localStorage.getItem('username')
    if (username) {
      this.setState({ username })
    }
  }

  _saveUsername (username) {
    // 看看 render 方法的 onUserNameInputBlur
    // 这个方法会在用户名输入框 blur 的时候被调用，保存用户名
    localStorage.setItem('username', username)
  }

  handleSubmitComment (comment) {
```

```

// 评论数据的验证
if (!comment) return
if (!comment.username) return alert('请输入用户名')
if (!comment.content) return alert('请输入评论内容')
// 新增评论保存到 LocalStorage 中
const { comments } = this.props
const newComments = [...comments, comment]
localStorage.setItem('comments', JSON.stringify(newComments))
// this.props.onSubmit 是 connect 传进来的
// 会 dispatch 一个 action 去新增评论
if (this.props.onSubmit) {
  this.props.onSubmit(comment)
}
}

render () {
  return (
    <CommentInput
      username={this.state.username}
      onUserNameInputBlur={this._saveUsername.bind(this)}
      onSubmit={this.handleSubmitComment.bind(this)} />
  )
}
}

const mapStateToProps = (state) => {
  return {
    comments: state.comments
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    onSubmit: (comment) => {
      dispatch(addComment(comment))
    }
  }
}

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(CommentInputContainer)

```

同样地，对代码的解释都放在了注释当中。这样就构建了一个 Smart 的 `CommentInput`。

## Smart CommentApp

接下来的事情都是很简单，我们用 `CommentApp` 把这两个 Smart 的组件组合起来，把 `src/CommentApp.js` 移动到 `src/containers/CommentApp.js`，把里面的内容替换为：

```
import React, { Component } from 'react'
import CommentInput from './CommentInput'
import CommentList from './CommentList'

export default class CommentApp extends Component {
  render() {
    return (
      <div className='wrapper'>
        <CommentInput />
        <CommentList />
      </div>
    )
  }
}
```

原本很复杂的 `CommentApp` 现在变得异常简单，因为它的逻辑都分离到了两个 Smart 组件里面去了。原来的 `CommentApp` 确实承载了太多它不应该承担的责任。分离这些逻辑对我们代码的维护和管理也会带来好处。

最后一步，修改 `src/index.js`：

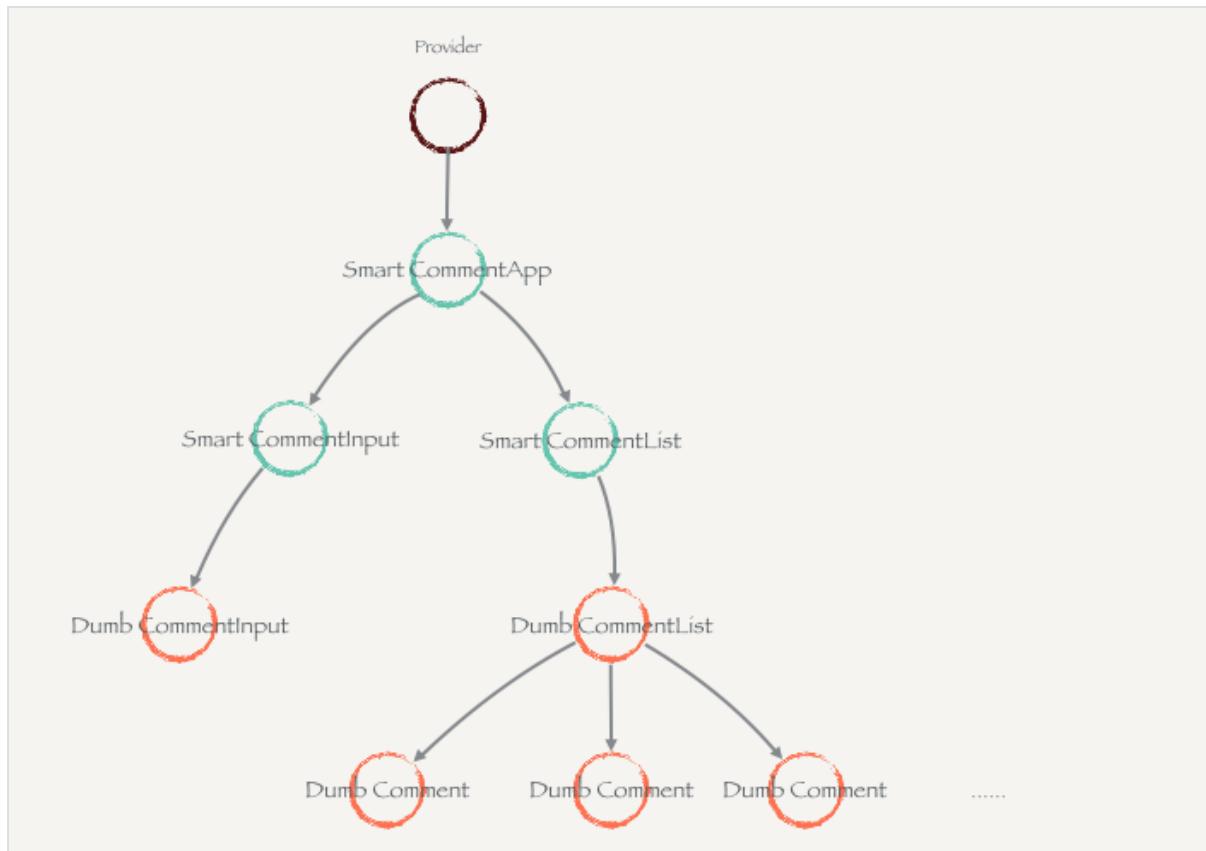
```
import React from 'react'
import ReactDOM from 'react-dom'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import CommentApp from './containers/CommentApp'
import commentsReducer from './reducers/comments'
import './index.css'

const store = createStore(commentsReducer)

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

通过 `commentsReducer` 构建一个 `store`，然后让 `Provider` 把它传递下去，这样我们就完成了最后的重构。

我们最后的组件树是这样的：



文件目录：

```

src
├── components
│   ├── Comment.js
│   ├── CommentInput.js
│   └── CommentList.js
├── containers
│   ├── CommentApp.js
│   ├── CommentInput.js
│   └── CommentList.js
└── reducers
    └── comments.js
└── index.css
└── index.js
  
```

所有代码可以在这里找到：[comment-app3](#)。

因为第三方评论工具有问题，对本章节有任何疑问的朋友可以移步到 [React.js 小书的论坛](#) 发帖，我会回答大家的疑问。

上一节：45. 实战分析：评论功能（八）  
本《React.js小书》的PDF版本

是由 若川 <https://lxchuan12.cn>

使用node 库 puppeteer爬虫生成， 仅供学习交流，严禁用于商业用途。

文章 前端使用puppeteer 爬虫生成《React.js 小书》PDF并合并：

<https://juejin.im/post/5b86732451882542af1c8082>

项目源代码地址：<https://github.com/lxchuan12/learn-nodejs/tree/master/src/puppeteer/reactMiniBook.js>