lesson28

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`，然后把传进入去的 `WrappedComponent` 渲染出来。但是我们可以给 `NewCompoent` 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/wrapWithLoadData.js` 文件中的，我们可以在别的地方这么用它：

```
import wrapWithLoadData from './wrapWithLoadData'

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

InputWithUserName = wrapWithLoadData(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 `wrapWithLoadData`，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 `wrapWithLoadData`，`wrapWithLoadData` 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 `wrapWithLoadData` 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 wrapWithLoadData from './wrapWithLoadData'

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

TextareaWithContent = wrapWithLoadData(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 `wrapWithLoadData` 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 `wrapWithLoadData` 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
}
```

其实就是改了一下 `wrapWithLoadData` 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）

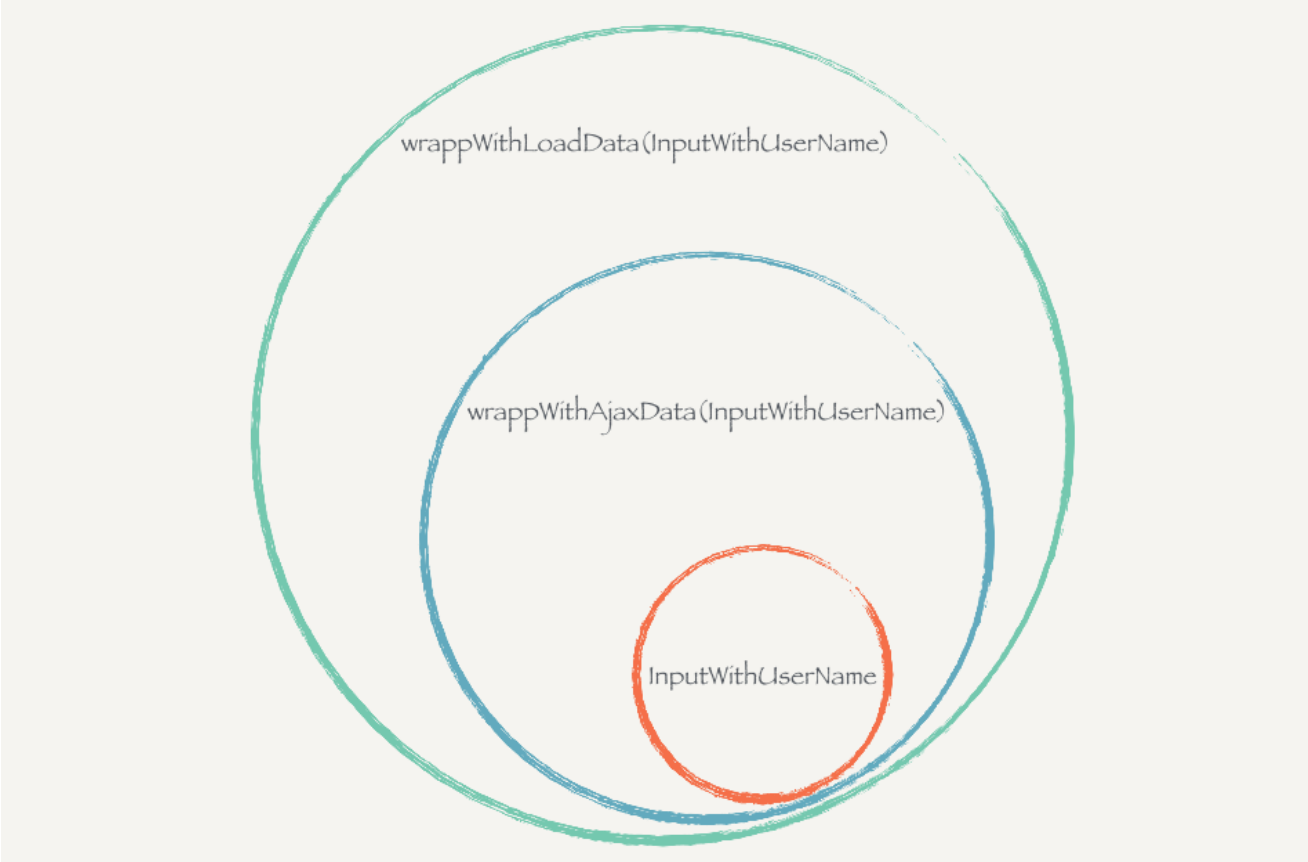
假如现在需求有变化了：我们需要先从 LocalStorage 中加载数据，再用这个数据去服务器取数据。我们改一下（或者新建一个）`wrapWithAjaxData` 高阶组件，修改其中的 `componentWillMount`：

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

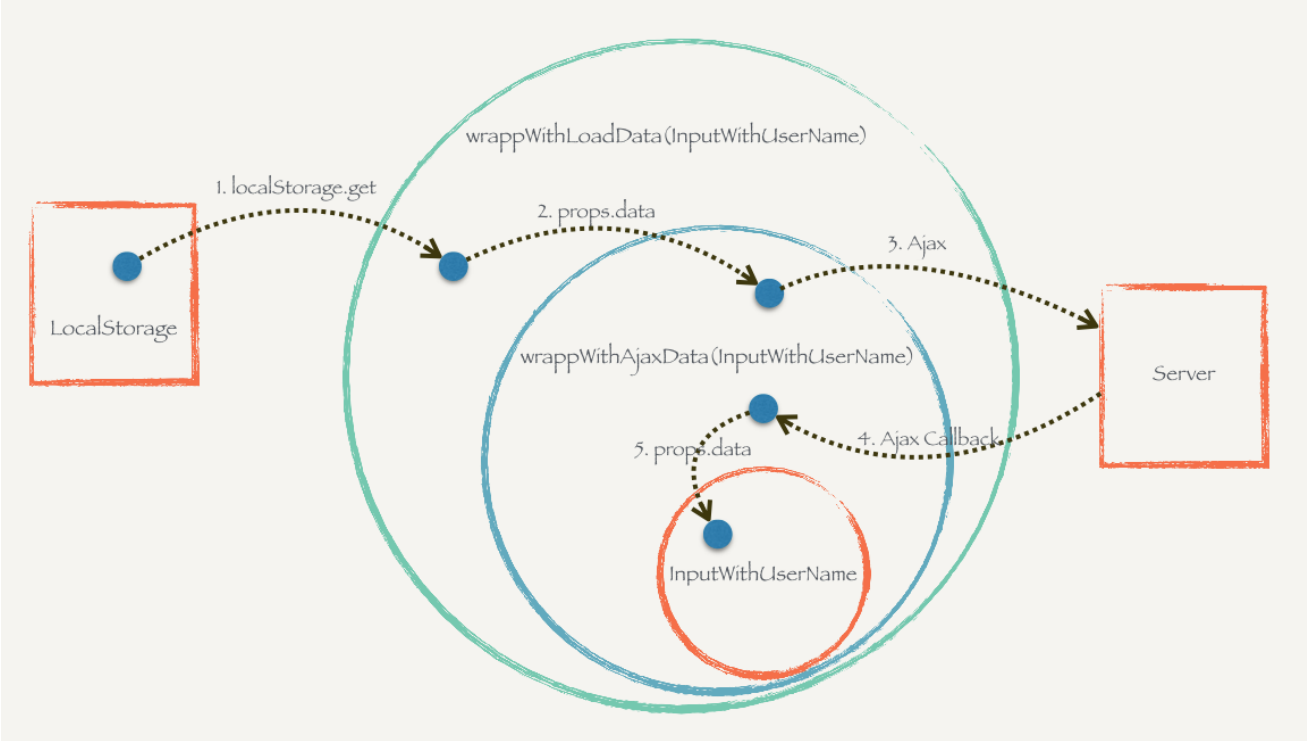
它会用传进来的 `props.data` 去服务器取数据。这时候修改 `InputWithUserName`：

```
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组件之间的数据流向：



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 = wrapWithLoadData(CommentApp, 'comments')
export default CommentApp
```

同样地可以在 `CommentInput` 中使用 `wrapWithLoadData`，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。

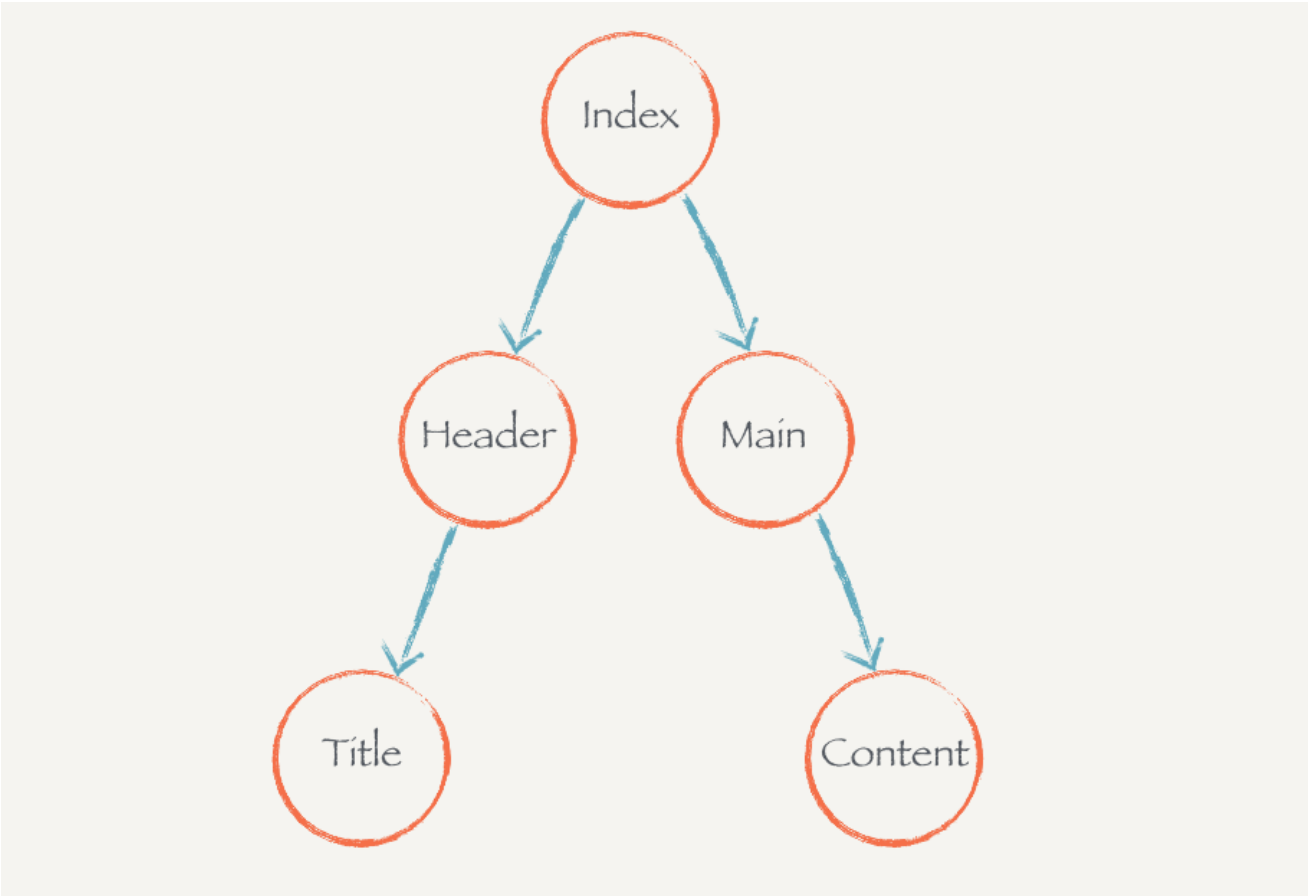
lesson29

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。

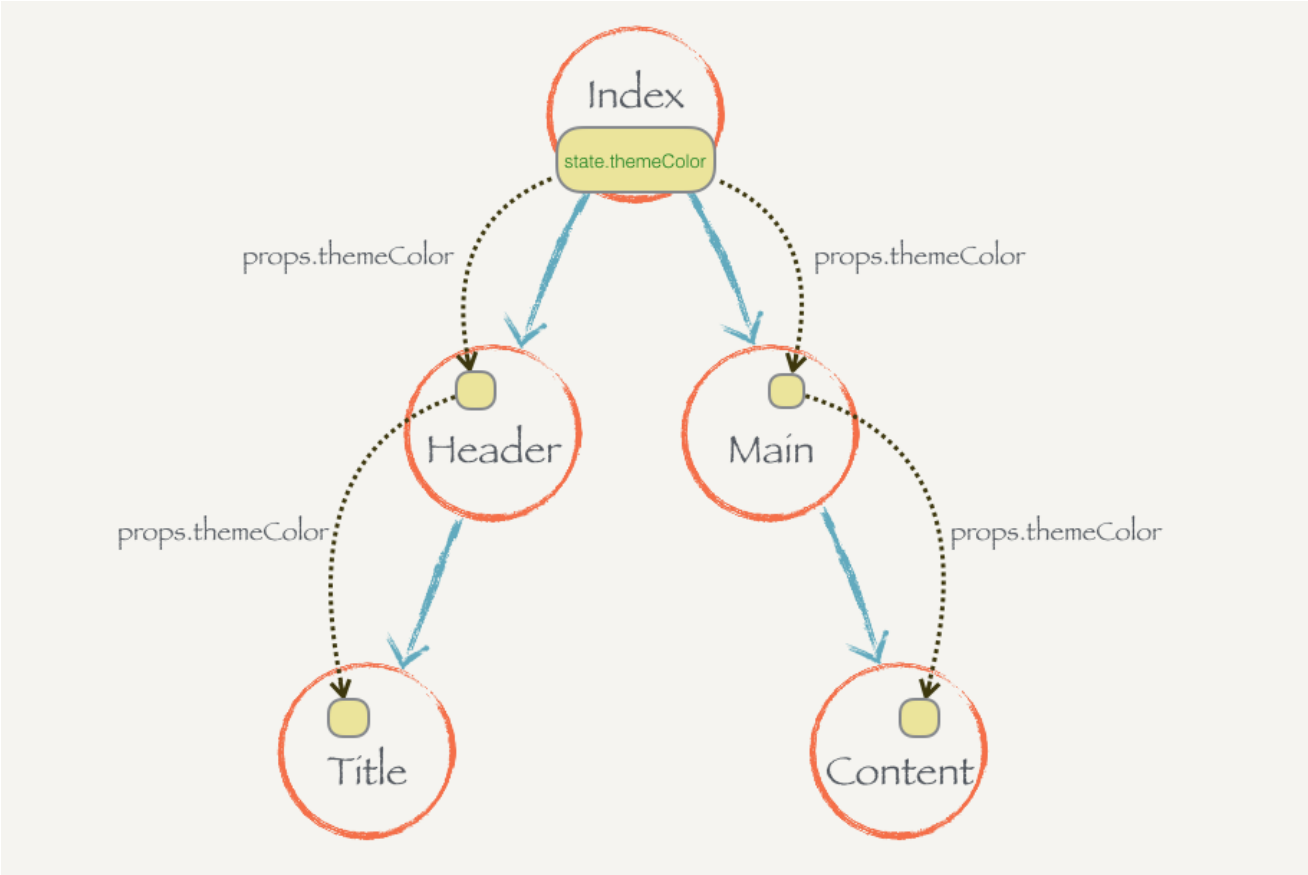
在过去很长一段时间里面，React.js 的 context 一直被视为一个不稳定的、危险的、可能会被去掉的特性而不被官网文档所记载。但是全世界的第三方库都在用使用这个特性，直到了 React.js 的 v0.14.1 版本，context 才被官方文档所记录。

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：



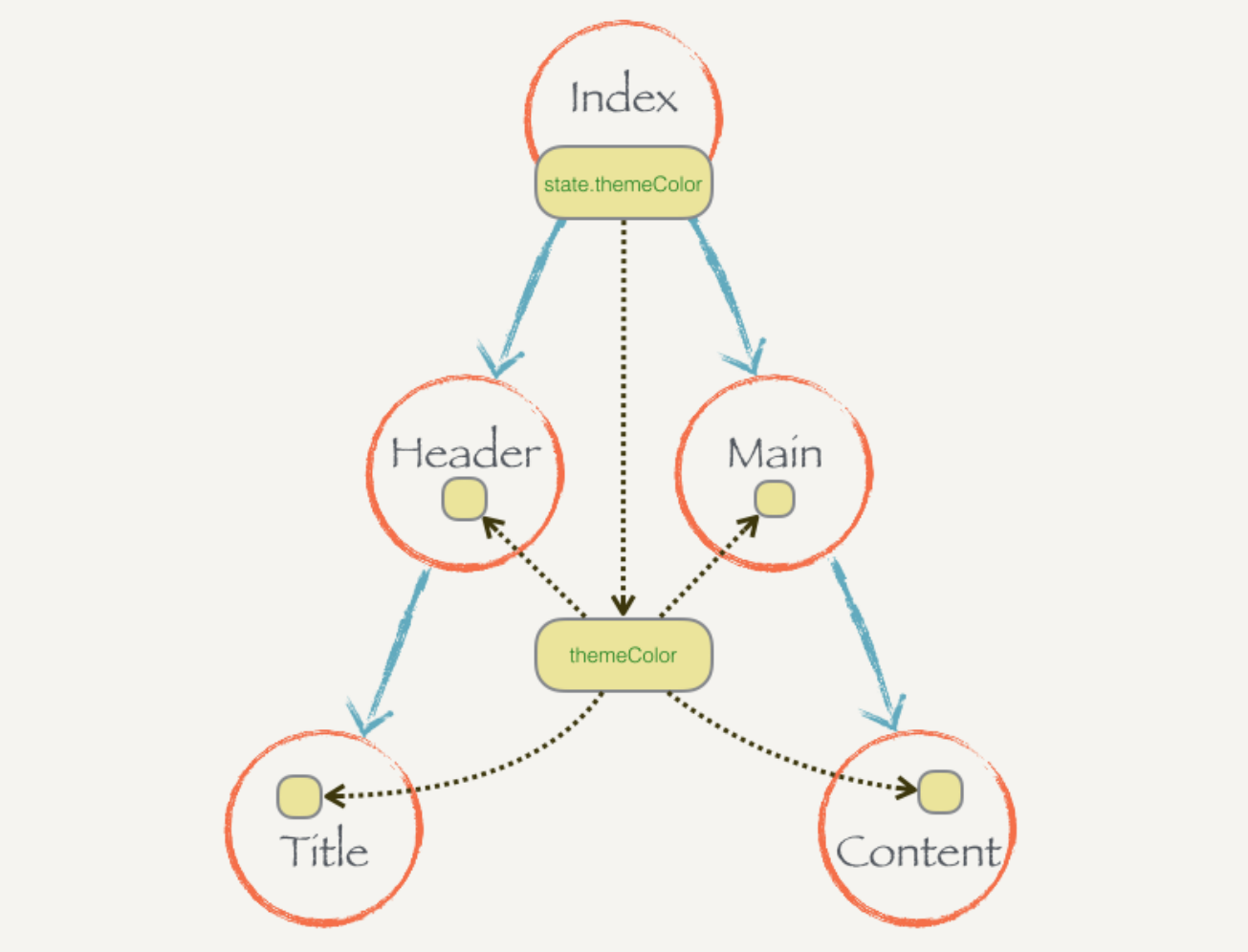
假设现在这个组件树代表的应用是用户可以自主换主题色的，每个子组件会根据主题色的不同调整自己的字体颜色或者背景颜色。“主题色”这个玩意是所有组件共享的状态，根据我们在 前端应用状态管理 —— 状态提升 中所提到的，需要把这个状态提升到根节点的 `Index` 上，然后把这个状态通过 `props` 一层层传递下去：



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 `propsType` 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` 的传递获就可以获取到由 `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。

lesson30

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 `rendeTitle` 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()
doSomthingMore()
// ...
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。

你可能会说我去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 `swtich` 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
```

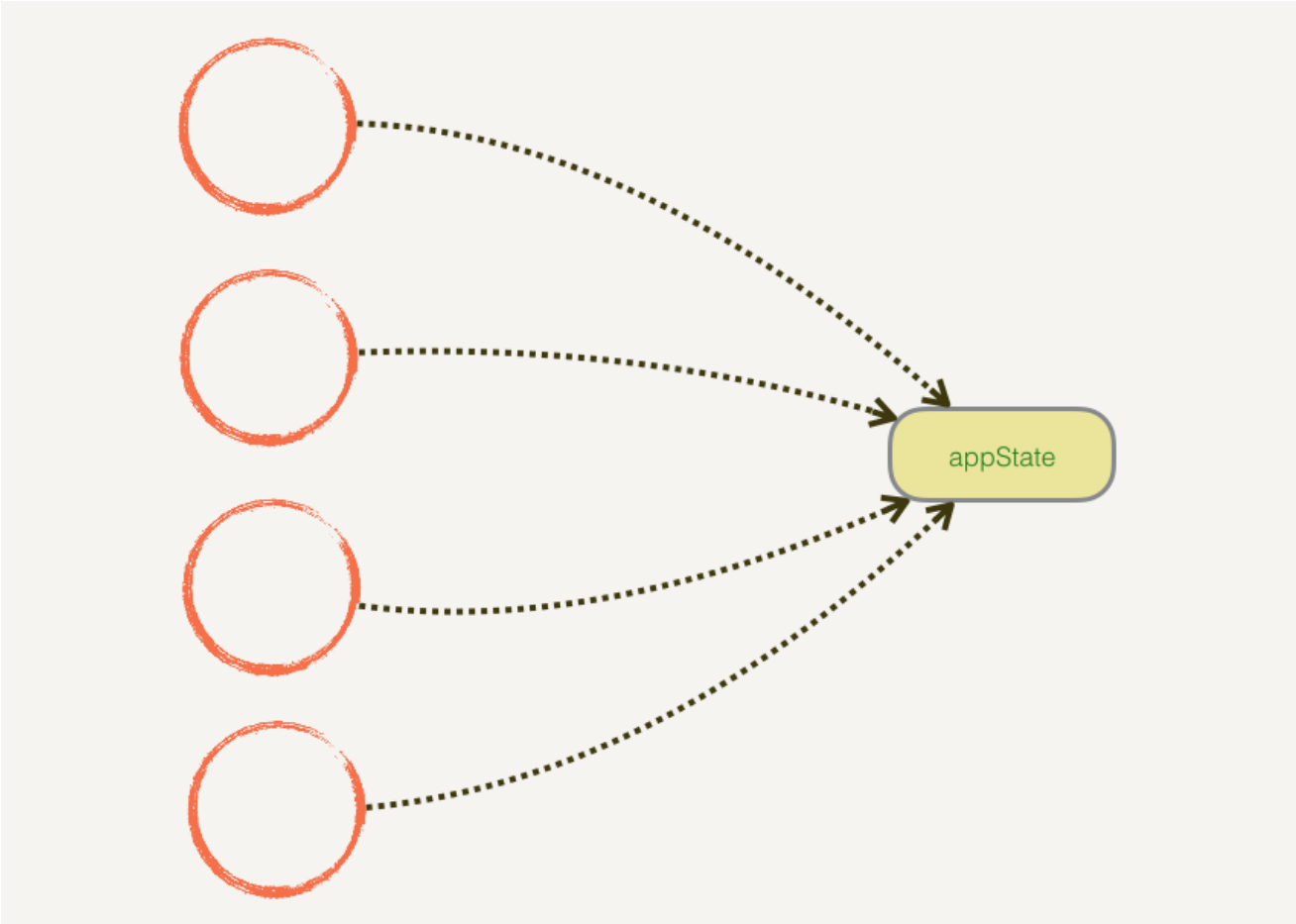
我们来看看有什么好处：

```
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)
```

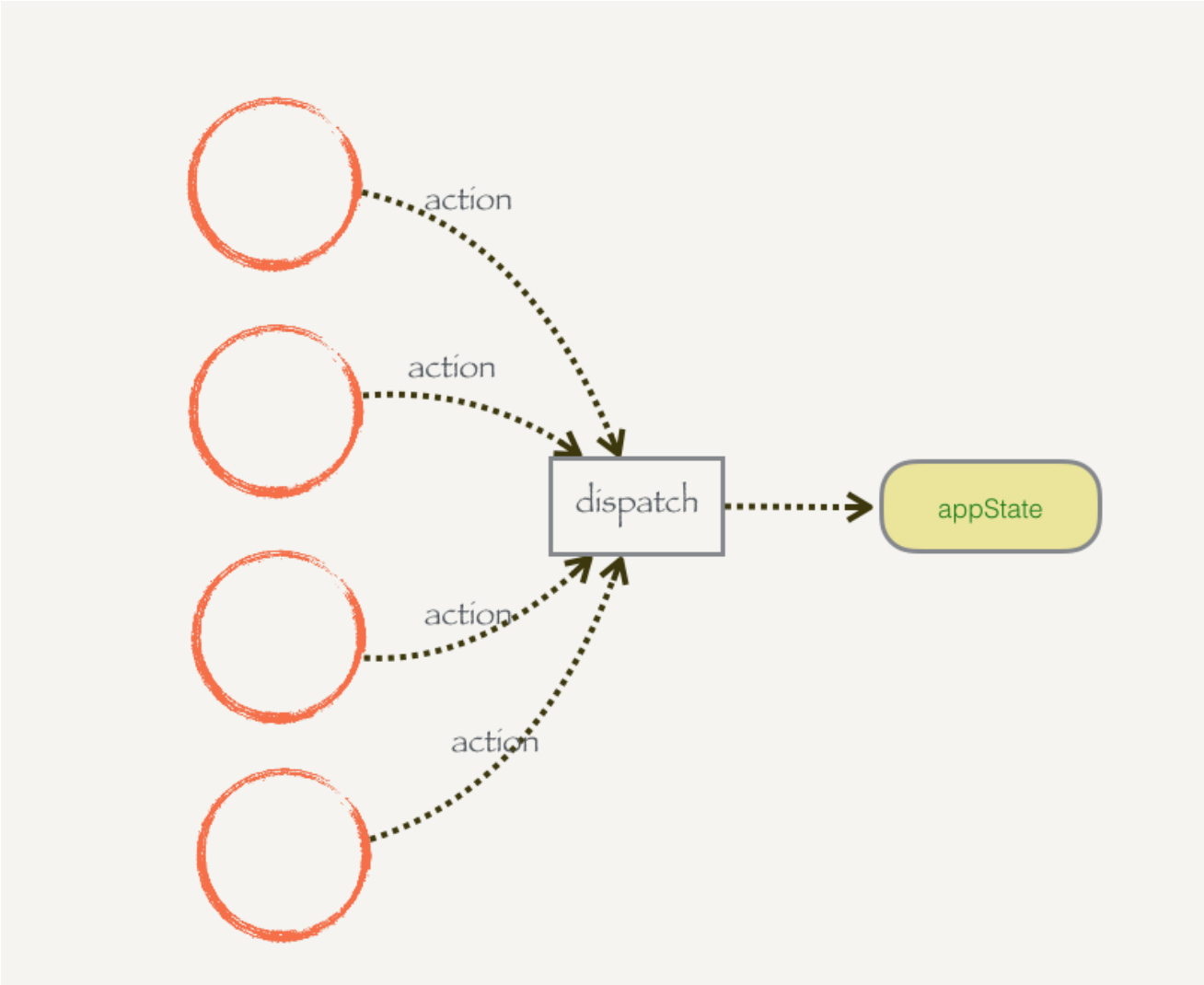
我们不需要担心 `renderApp(appState)` 之前的那堆函数操作会干什么奇奇怪怪得事情，因为我们规定不能直接修改 `appState`，它们对 `appState` 的修改必须只能通过 `dispatch`。而我们看看 `dispatch` 的实现可以知道，你只能修改 `title.text` 和 `title.color`。

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。

lesson31

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》' }) // 修改标题文本
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 EventEmitter 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。

lesson32

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，我们就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：

1. 函数的返回结果只依赖于它的参数。
2. 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。

lesson33

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。

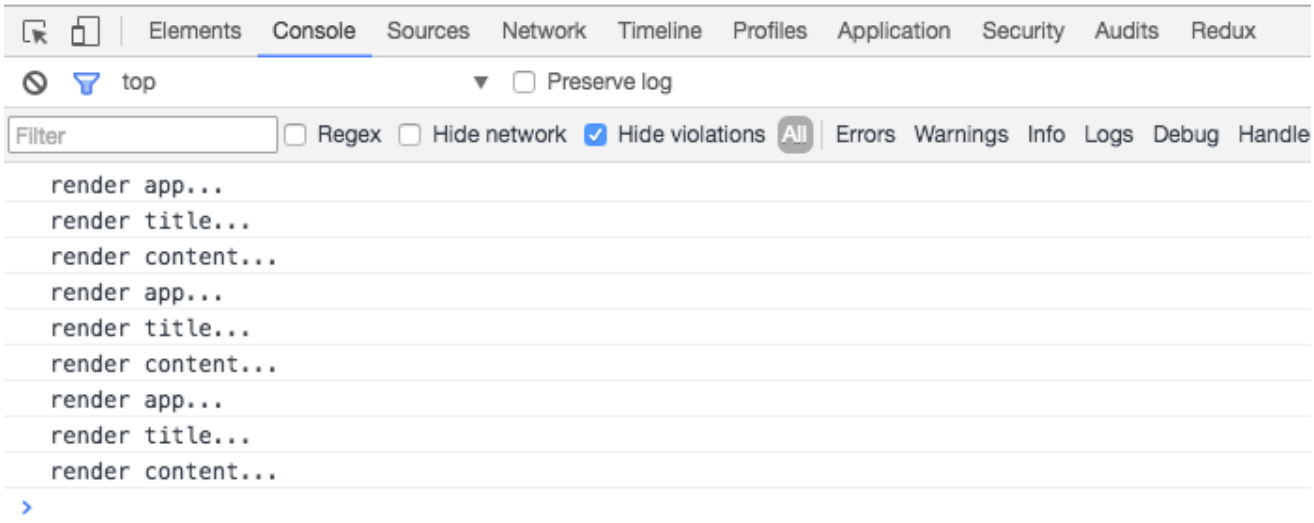
细心的朋友可以发现，其实我们之前的例子当中是有比较严重的*性能问题*的。我们在每个渲染函数的开头打一些 Log 看看：

```
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，所以加了默认参数 oldAppState = {}
  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入去：

```
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，等待下一次数据变化重新渲染
})
...
```

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
```

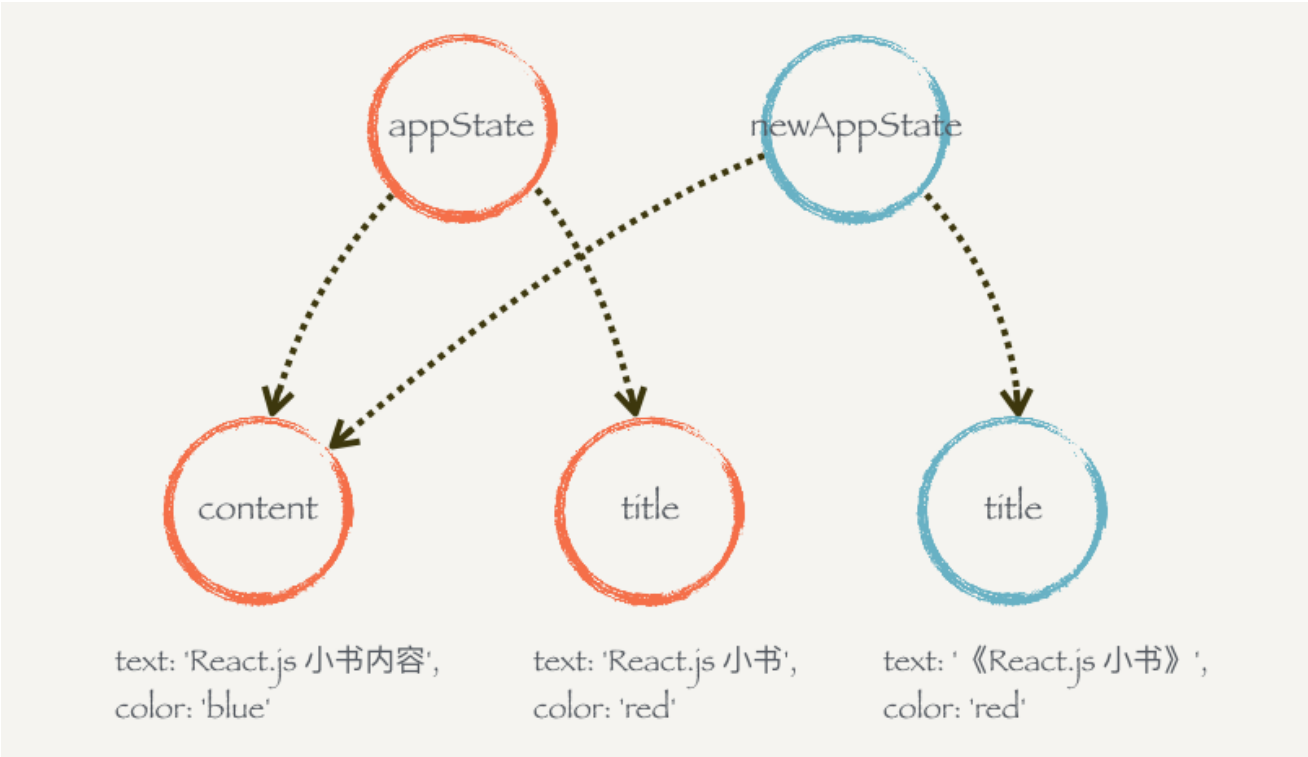
我们可以把这种特性应用在 `state` 的更新上，我们禁止直接修改原来的对象，一旦你要修改某些东西，你就得把修改路径上的所有对象复制一遍，例如，我们不写下面的修改代码：

```
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`：

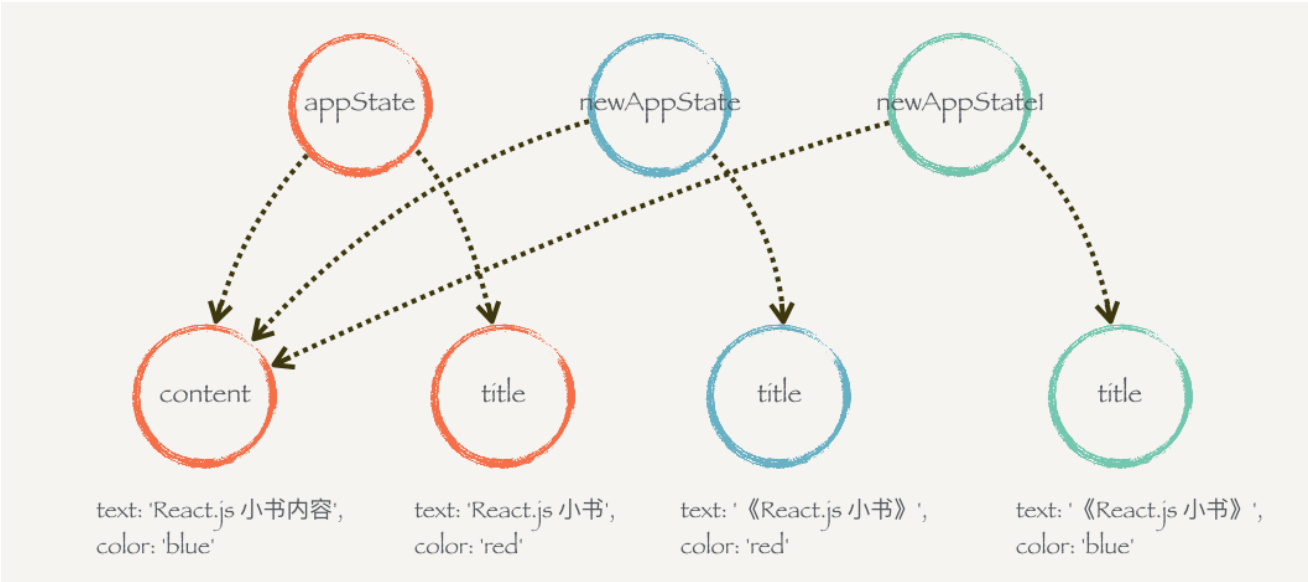
```
let newAppState = { // 新建一个 newAppState
  ...appState, // 复制 appState 里面的内容
  title: { // 用一个新的对象覆盖原来的 title 属性
    ...appState.title, // 复制原来 title 对象里面的内容
    text: '《React.js 小书》' // 覆盖 text 属性
  }
}
```

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 !== newAppState // true，两个对象引用不同，数据变化了，重新渲染
appState.title !== newAppState.title // true，两个对象引用不同，数据变化了，重新渲染
appState.content !== appState.content // false，两个对象引用相同，数据没有变化，不需要重新渲染
```

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：

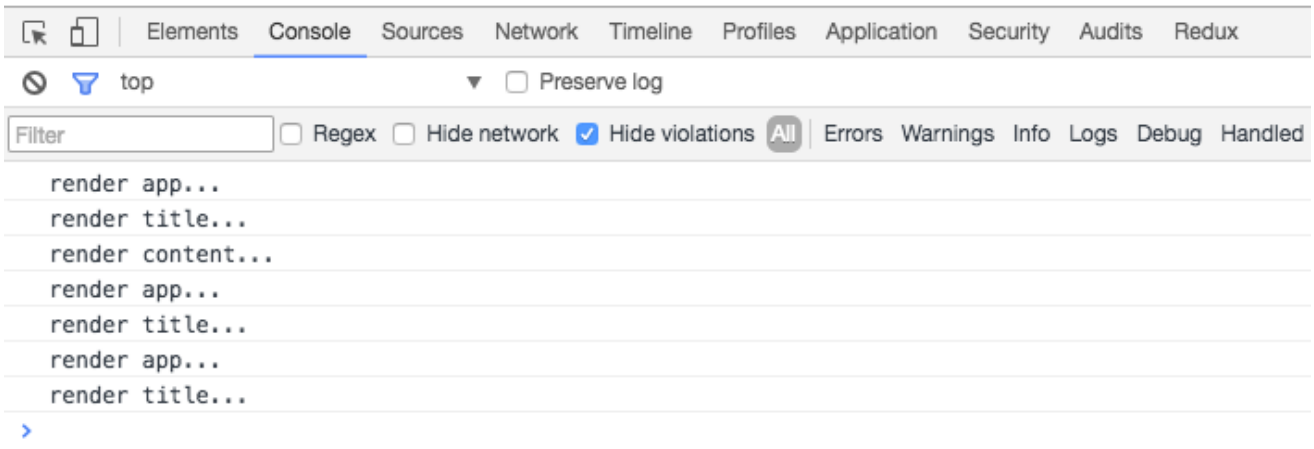
```
function stateChanger (state, action) {
  switch (action.type) {
    case 'UPDATE_TITLE_TEXT':
      return { // 构建新的对象并且返回
        ...state,
        title: {
          ...state.title,
          text: action.text
        }
      }
    case 'UPDATE_TITLE_COLOR':
      return { // 构建新的对象并且返回
        ...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，所以加了默认参数
oldAppState = {}
  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，等待下一次数据变化重新渲染
})

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
```

lesson34

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,
          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 'UPATE_THEME_NAME':
      return { ...state, themeName: action.themeName }
    case 'UPATE_THEME_COLOR':
      return { ...state, themeColor: action.themeColor }
    default:
      return state
  }
}

const store = createStore(themeReducer)
...
```

lesson35

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，我们再来回顾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。所以后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的 `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。

lesson36

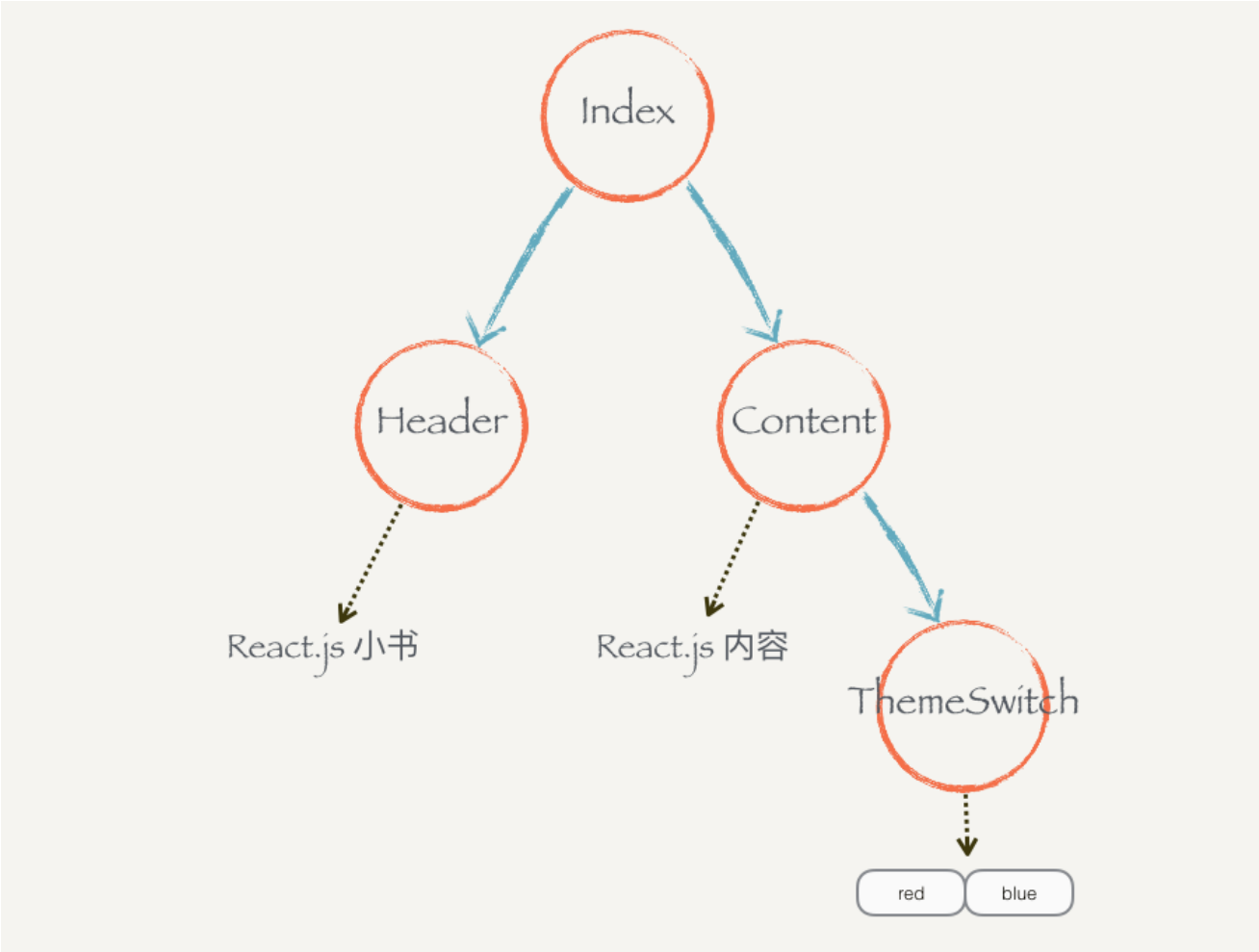
可以看到 Redux 并不复杂，它那些看起来匪夷所思的设定其实都是为了解决特定的问题而存在的，我们把问题想清楚以后就不难理解它的那些奇怪的设定了。这节开始我们来看看如何把 Redux 和 React.js 结合起来，你会发现其实它们也并不复杂。

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, PropTypes } from 'react'

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

export default Header
```

修改 `src/ThemeSwitch.js`：

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

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, PropTypes } from 'react'
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, PropTypes } from 'react'
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。

lesson37

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, PropTypes } from 'react'
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 `CHNAGE_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>
    )
  }
}
```

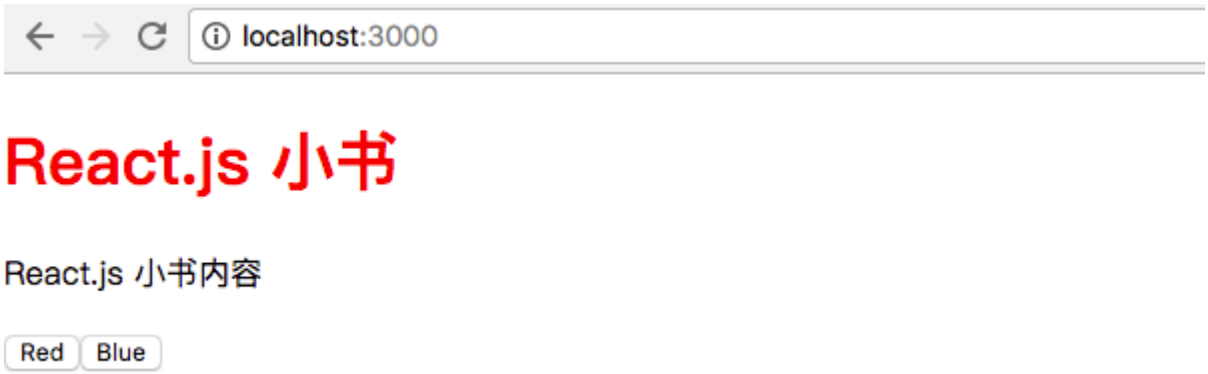
如果有些同学已经忘记了 context 的用法，可以参考之前的章节： React.js 的 context 。

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：



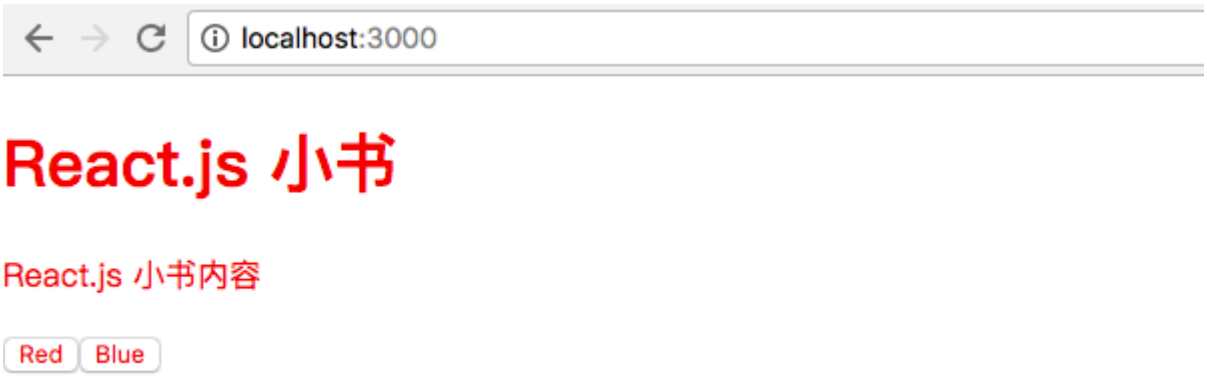
如法炮制 `Content.js`：

```
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：

React.js 小书

React.js 小书内容

Red Blue

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。

lesson38

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：

1. *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。
2. *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。

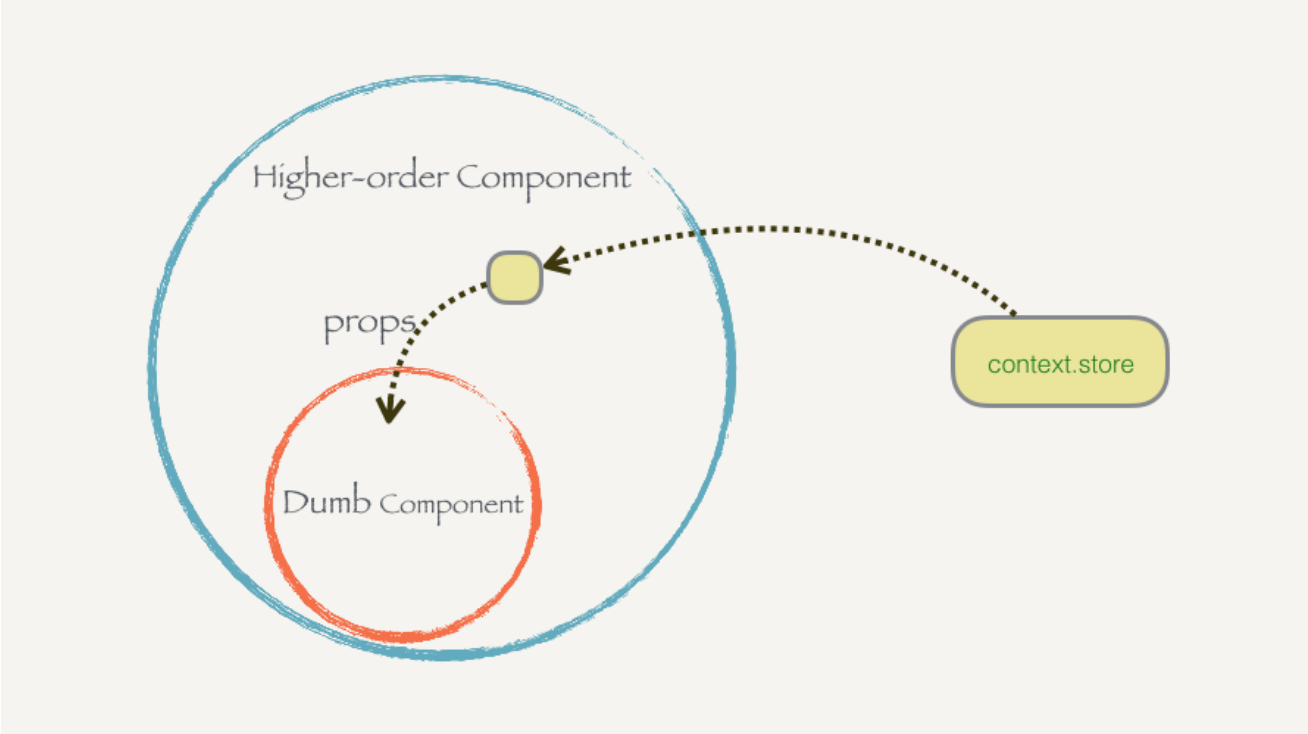
对于第一个问题，我们在 高阶组件 的章节说过，可以把一些可复用的逻辑放在高阶组件当中，高阶组件包装的新组件和原来组件之间通过 `props` 传递信息，减少代码的重复程度。

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 Compoent，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, PropTypes } from 'react'

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。`mapStateTopProps` 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, PropTypes } from 'react'

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, PropTypes } from 'react'
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, PropTypes } from 'react'
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 props，让获取数据
更加灵活方便
      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`。

lesson39

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, PropTypes } from 'react'
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。

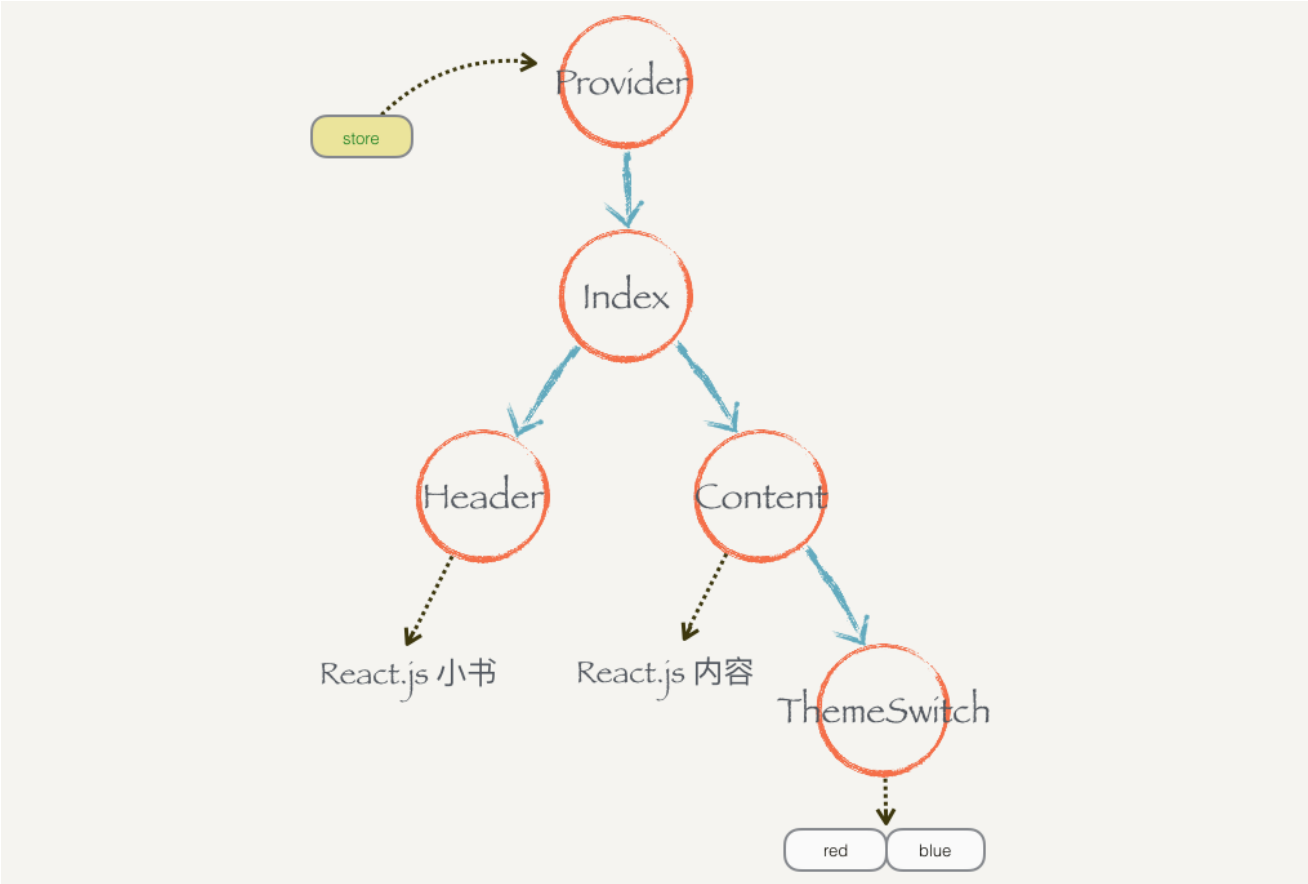
lesson40

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。

lesson41

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。

lesson42

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'
```

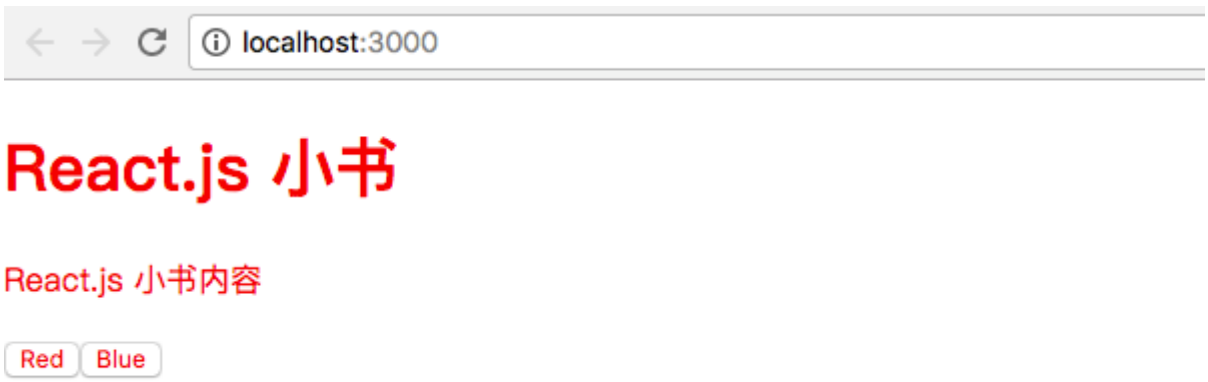
也就是本来从本地 `./react-redux` 导入的 `connect` 改成从第三方 `react-redux` 模块中导入。

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`。

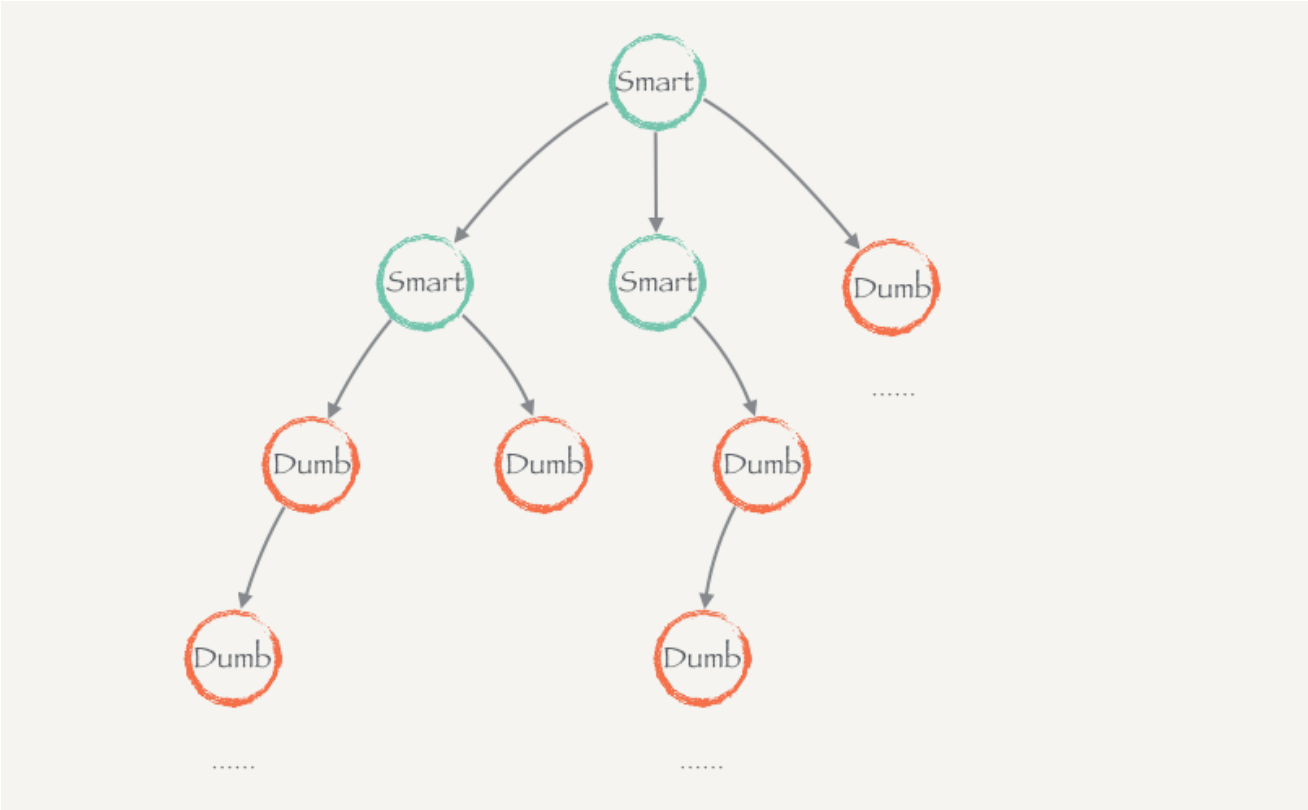
lesson43

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，导致它无法无用，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, PropTypes } from 'react'
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, PropTypes } from 'react'

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：

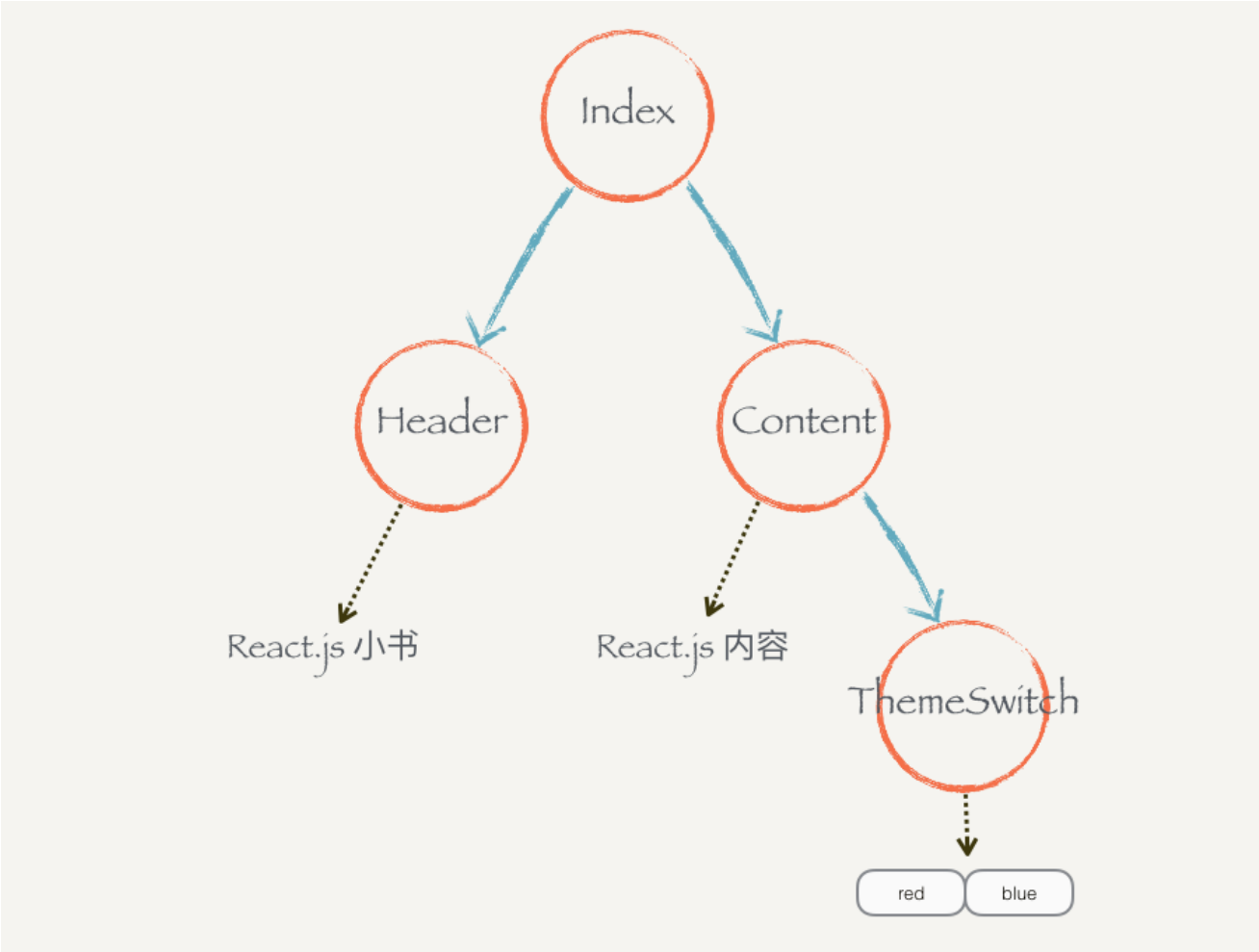
```
import { connect } from 'react-redux'
import Header from '../components/Header'

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, PropTypes } from 'react'

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, PropTypes } from 'react'
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` 的之下的子组件 `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。

lesson44

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 }
}
```

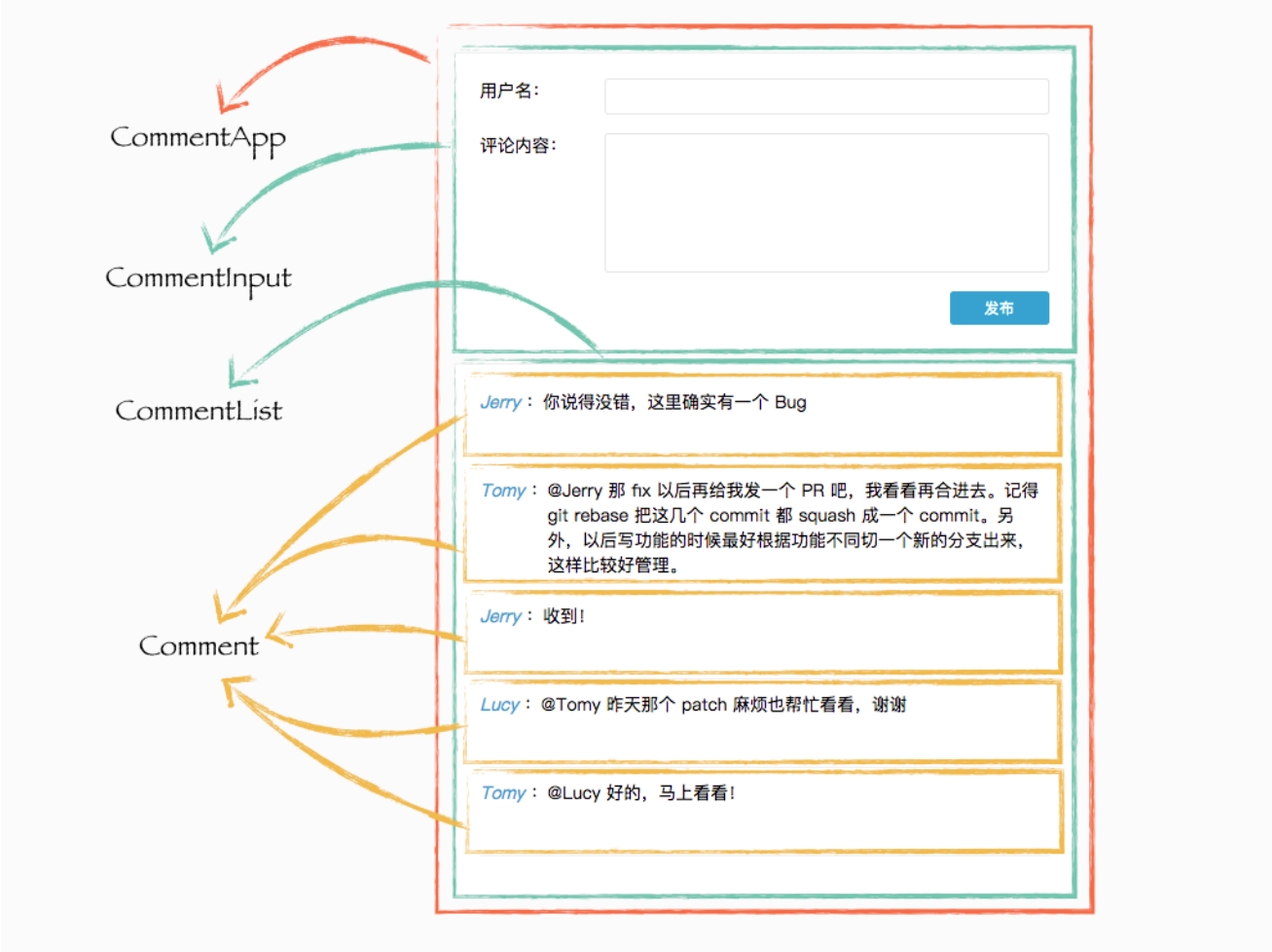
> 有些朋友可能会发现我们的 reducer 跟网上其他的 reducer 的例子不大一样。有些人喜欢把 action 单独切出去一个目录 `actions`，让 action 和 reducer 分开。个人观点觉得这种做法可能有点过度优化了，其实多数情况下特定的 action 只会影响特定的 reducer，直接放到一起可以更加清晰地知道这个 action 其实只是会影响到什么样的 reducer。而分开会给我们维护和理解代码带来额外不必要的负担，这有种矫枉过正的意味。但是这里没有放之四海皆准的规则，大家可以多参考、多尝试，找到适合项目需求的方案。

个人写 reducer 文件的习惯，仅供参考：

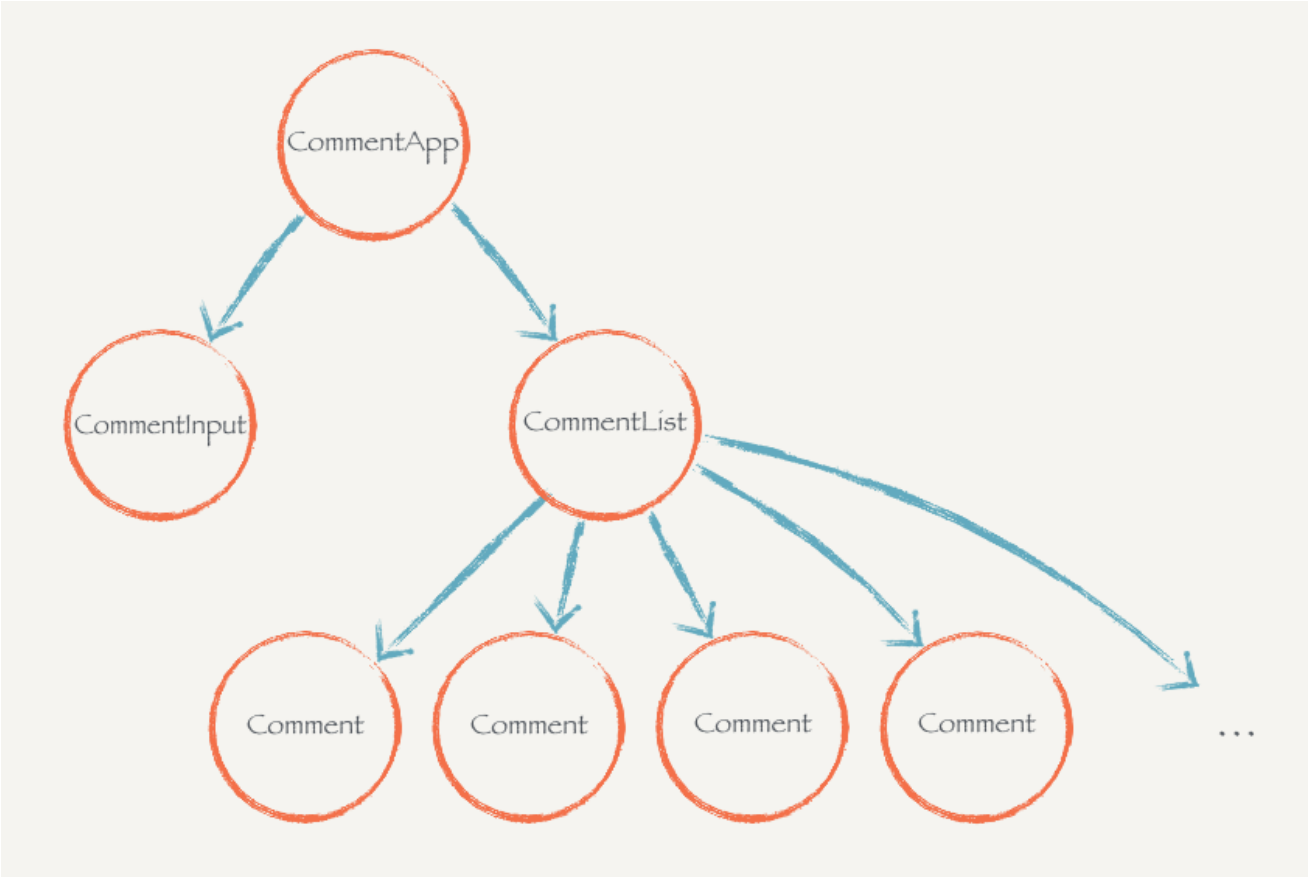
1. 定义 action types
2. 编写 reducer
3. 跟这个 reducer 相关的 action creators

lesson45

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，那么这个组件就没法复用了。

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, PropTypes } from 'react'

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
    // ...
  }
}
```

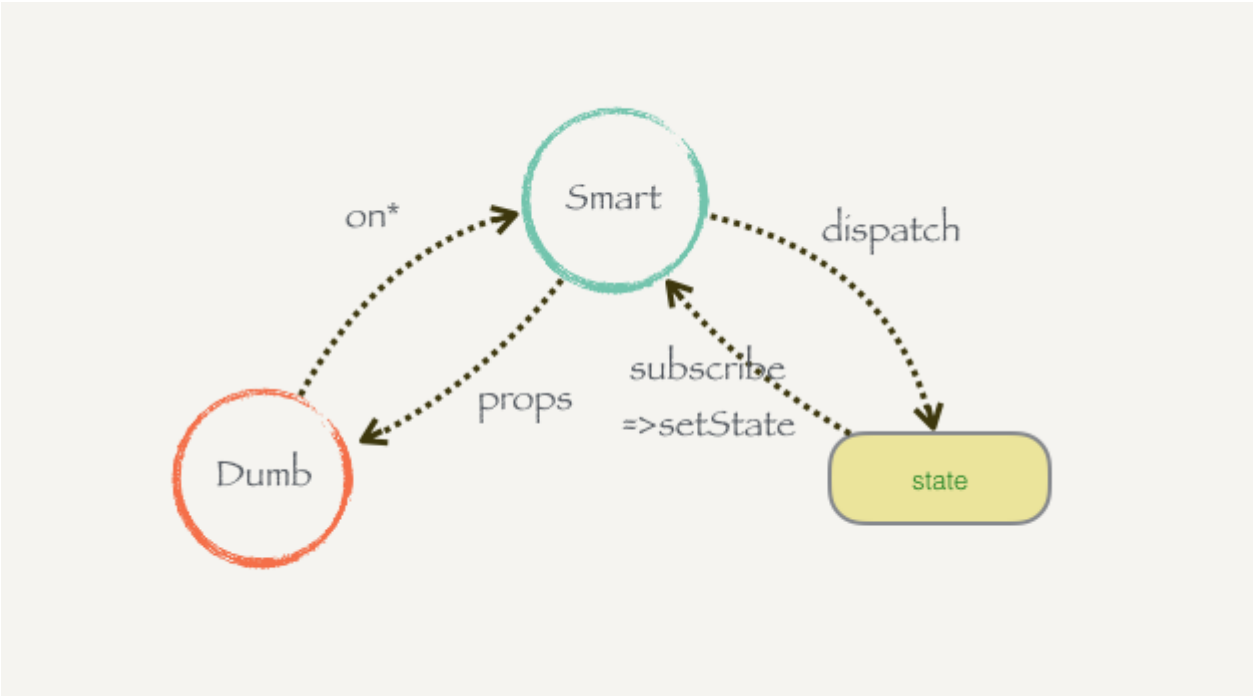
其实改动不多。原来 `CommentInput` 需要从 LocalStorage 中获取 `username` 字段，现在让它从 `props` 里面去取；而原来用户名的输入框 blur 的时候需要保存 `username` 到 LocalStorage 的行为也通过 `props.onUserNameInputBlur` 传递到上层去做。现在 `CommentInput` 是一个 Dumb 组件了，它的所有渲染操作都只依赖于 `props` 来完成。

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。

lesson46

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, PropTypes } from 'react'
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, PropTypes } from 'react'
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 的时候的被调用，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>
    )
  }
}
```

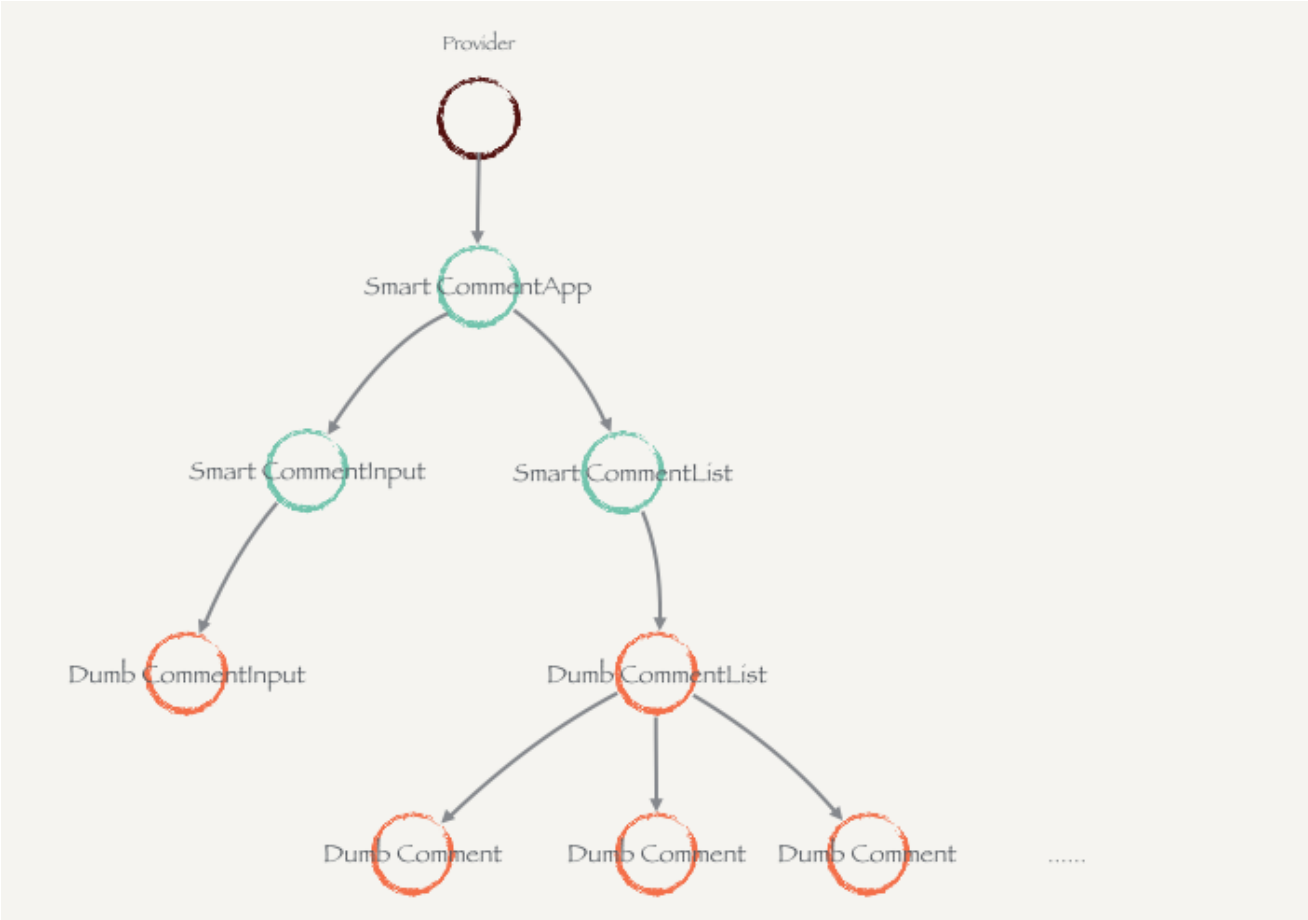
原本很复杂的 `CommentApp` 现在变得异常简单，因为它的逻辑都分离到了两个 Smart 组件里面去了。原来的 `CommentApp` 确实承载了太多它不应该承担的责任。分离这些逻辑对我们代码的维护和管理也会带来好处。

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。