React.js 小书

<-- 返回首页

Smart 组件 vs Dumb 组件

- 作者：胡子大哈
- 原文链接： http://huziketang.com/books/react/lesson43

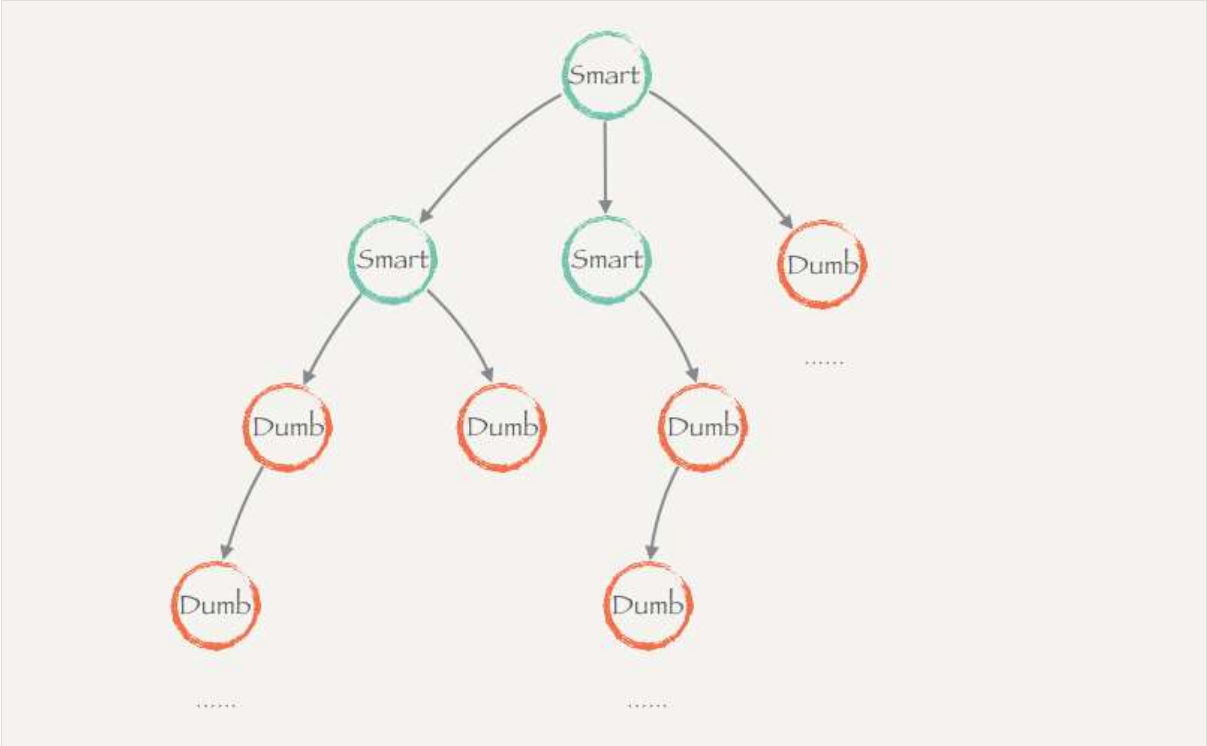
（本文未审核）

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
  }
```

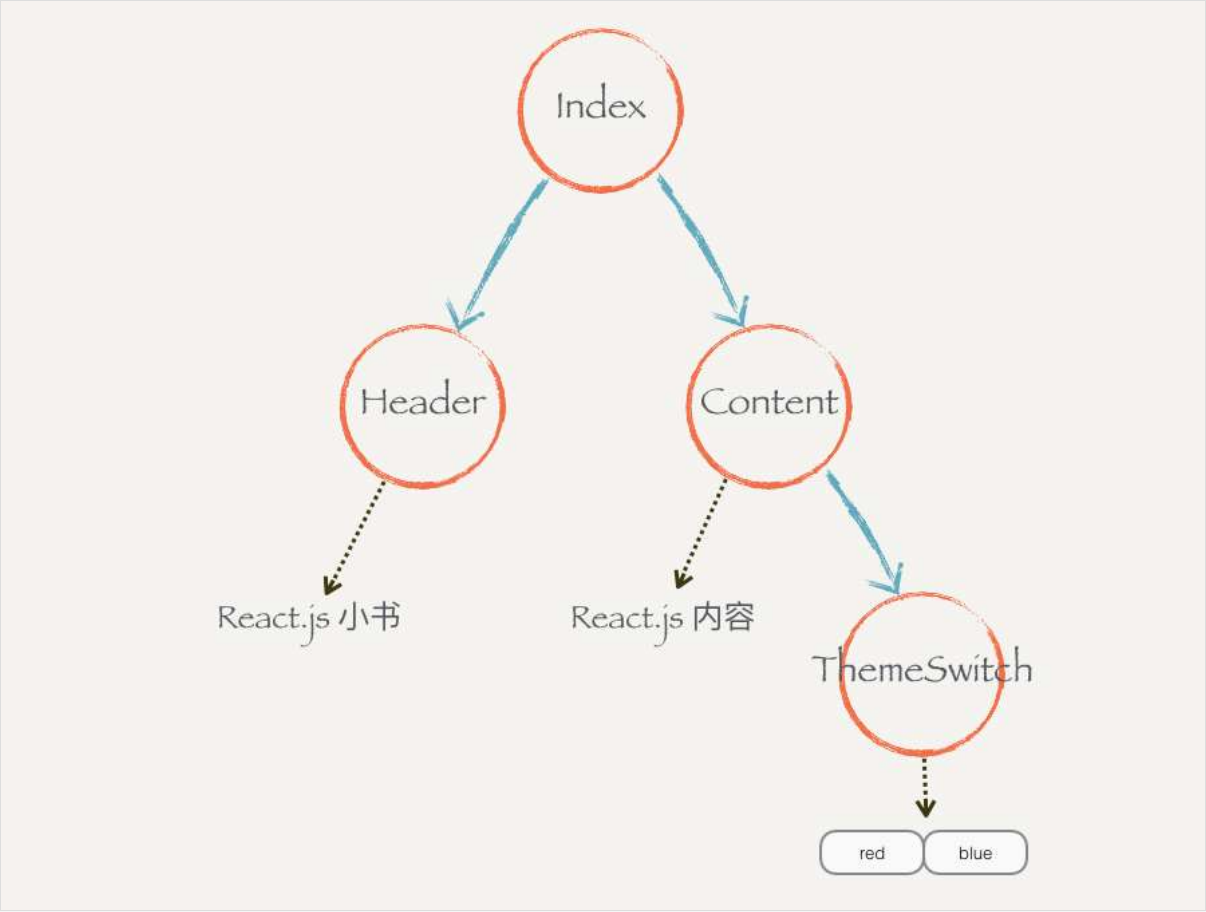
```
}
export default connect(mapStateToProps)(Header)
```

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。

> 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：实战分析：评论功能（七）

上一节：使用真正的 Redux 和 React-redux

如果你觉得小书写得还不错，可以请胡子大哈喝杯茶 :)

赞赏

或者传播一下知识也是一个很好的选择

1 条评论，1 人参与。

★ 7

我有话说...

使用社交帐号登录 发布前先点击左边的按钮登录

最新评论

阳光老男孩 • 4月8日 08:45

contaienrs，文章中段的拼写错误

顶 • 回复 • 分享»

友言?