React 小书

基本环境

1. 安装node

   > 修改node镜像

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

2. 全局安装脚手架

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

3. 通过脚手架命令新建工程

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

使用JSX描述UI信息

JSX 原理

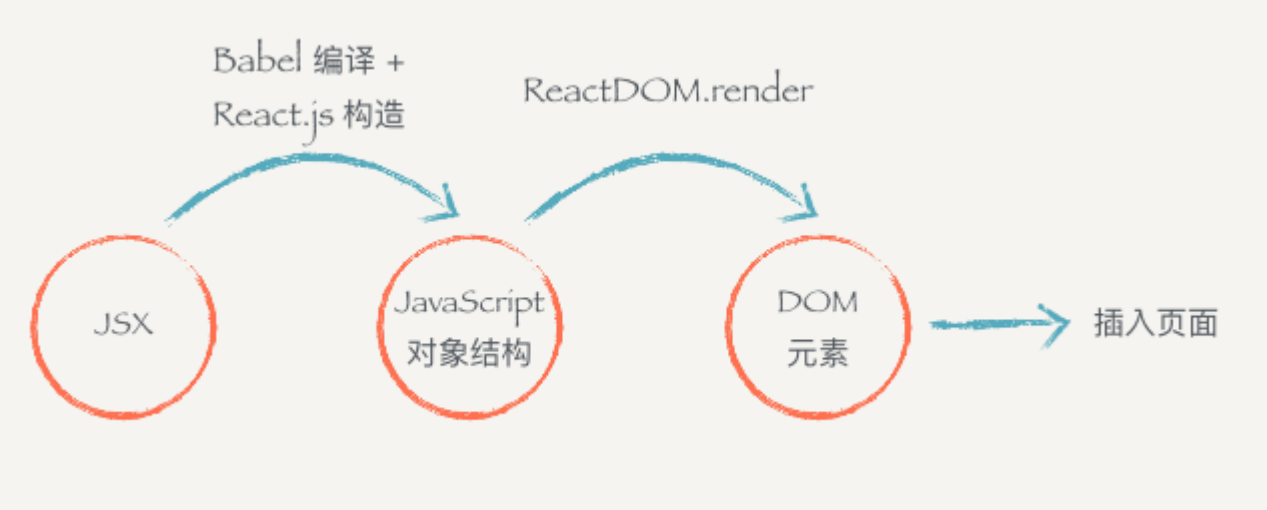
```
<div class='box' id='content'>
  <div class='title'>Hello</div>
  <button>Click</button>
</div>
```

> 每个DOM元素的结构都可以用javaScript的对象来表示，一个DOM元素包含的信息其实包含三个
>
> 1. 标签名
> 2. 属性
> 3. 子元素

用JavaScript 来表示

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

```
      attrs: null,
      children: ['Click']
    }
  ]
}
```



> ，为什么不直接从 JSX 直接渲染构造 DOM 结构，而是要经过中间这么一层呢？
>
> 第一个原因是，当我们拿到一个表示 UI 的结构和信息的对象以后，不一定会把元素渲染到浏览器的普通页面上，我们有可能把这个结构渲染到 canvas 上，或者是手机 App 上。所以这也是为什么会要把 `react-dom` 单独抽离出来的原因，可以想象有一个叫 `react-canvas` 可以帮我们把 UI 渲染到 canvas 上，或者是有一个叫 `react-app` 可以帮我们把它转换成原生的 App（实际上这玩意叫 `ReactNative`）。
>
> 第二个原因是，有了这样一个对象。当数据变化，需要更新组件的时候，就可以用比较快的算法操作这个 JavaScript 对象，而不用直接操作页面上的 DOM，这样可以尽量少的减少浏览器重排，极大地优化性能。这个在以后的章节中我们会提到。

组件的render 方法

- 编写React.js组件的时候，需要继承React的Component
- 一个组建磊必须要实现一个render 方法，这个方法必须要返回一个JSX元素

> 必须是一个根节点

表达式插入

> 注意，直接使用 `class` 在 React.js 的元素上添加类名如 `<div class="xxx">` 这种方式是不合法的。因为 `class` 是 JavaScript 的关键字，所以 React.js 中定义了一种新的方式：`className` 来帮助我们给元素添加类名。
>
> 还有一个特例就是 `for` 属性，例如 `<label for='male'>Male</label>`，因为 `for` 也是 JavaScript 的关键字，所以在 JSX 用 `htmlFor` 替代，即 `<label htmlFor='male'>Male</label>`。而其他的 HTML 属性例如 `style` 、 `data-*` 等就可以像普通的 HTML 属性那样直接添加上去。

条件返回

```
{
          isGoodWord ? <span>在条件渲染中我是真的返回</span> : <span>在条件渲染中我是假的返回</span>
}
```

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

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

JSX 变量

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

```
render () {
  let workKey = '我是表达式插入的(换不换行都不影响我)'
  const isGoodWord = true
  const goodWord = <strong> is good{ workKey }</strong>
  const badWord = <span> is not good</span>
  return (
    <div>
      <h1>
        React 小书
        {isGoodWord ? goodWord : badWord}
      </h1>
    </div>
  )
}
```

事件监听

> 在 React.js 不需要手动调用浏览器原生的 `addEventListener` 进行事件监听。React.js 帮我们封装好了一系列的 `on*` 的属性

> *这些 on* 的事件监听只能用在普通的 HTML 的标签上，而不能用在组件标签上*。也就是说，`<Header onClick={…} />` 这样的写法不会有什么效果的。这一点要注意，但是有办法可以做到这样的绑定，以后我们会提及。现在只要记住一点就可以了：这些 `on*` 的事件监听只能用在普通的 HTML 的标签上，而不能用在组件标签上。

event 对象

> 事件监听函数会被自动传入一个 `event` 对象

关于事件中的this

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

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

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

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

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

  render () {
    return (
      // 如果需要传递参数的话
      <h1 onClick={this.handleClickOnTitle.bind(this, 'Hello')}>React 小书</h1>
    )
  }
}
```

组件的 state和setState

setState 接受函数参数

> 你调用 `setState` 的时候，*React.js 并不会马上修改 state*。而是把这个对象放到一个更新队列里面，稍后才会从队列当中把新的状态提取出来合并到 `state` 当中，然后再触发组件更新。

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

> 两次打印都是false， 即使我们中间已经setState过一次了
>
> 如果你想在setState之后使用新的state来做后续运算就做不到

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

最终的结果是NaN ,setState 不能立即进行修改

setState的第二种使用方式

> 可以接受一个函数作为参数， React会把上一个setState的结果传入这个函数，可以使用此结果进行运算、操作、然后返回一个对象作为更新state的对象

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

setState 合并

> 多次进行setState不会带来性能问题

进行了三次setState， 但是实际上组件只会重新渲染一次，而不是三次

因为在React.js内部会把JavaScript事件循环中的消息队列同一个消息中的setState都进行合并以后再重新渲染组件

配置组件的props

向组件内部传入函数作为参数

```
class Index extends Component {
  render () {
    return (
      <div>
        <LikeButton
          wordings={{likedText: '已赞', unlikedText: '赞'}}
          onClick={() => console.log('Click on like button!')}/>
      </div>
    )
  }
}
```

> 可以通过this.props.onClick方法获取到这个传进来的函数

修改一下组件onClick时触发的函数为

```
...
  handleClickOnLikeButton () {
    this.setState({
      isLiked: !this.state.isLiked
    })
    // 在每次点击组件时触发handleClickOnLikeButton 判断是否传入onClick方法
    // 如果传入了 则进行调用
    if (this.props.onClick) {
      this.props.onClick()
    }
  }
...
```

默认配置 defaultProps

```
class LikeButton extends Component {
  // 加上了如下代码
  static defaultProps = {
    likedText: '取消',
    unlikedText: '点赞'
  }

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

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

```
  }

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

propTypes 和 组件参数验证

安装

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

使用

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

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

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

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

函数式组件验证

```
import React from 'react'
import PropTypes from 'prop-types'
import Comment from './Comment'

function CommentList ({ comments = [], onDeleteComment }) {
  return (
    <div>
      {
        comments.map((item, index) => {
          return <Comment comment = { item } key = {index} />
        } )
      }
    </div>
  )
}

CommentList.propTypes  = {
  comments: PropTypes.array.isRequired,
  onDeleteComment: PropTypes.func.isRequired
}

export default CommentList
```

```
const Text = ({ children }) =>
  <p>{children}</p>
Text.propTypes = { children: React.PropTypes.string };
Text.defaultProps = { children: 'Hello World!' };
```

props 不可变

> props 一旦传进来就不能改变

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

但这并不意味着由props决定的显示形态不能被修改 组件的使用者可以主动的通过重新渲染的方式 把新的props传入组件当中

> 通过父组件主动重新渲染的方式来传入新的props，从而达到更新的效果

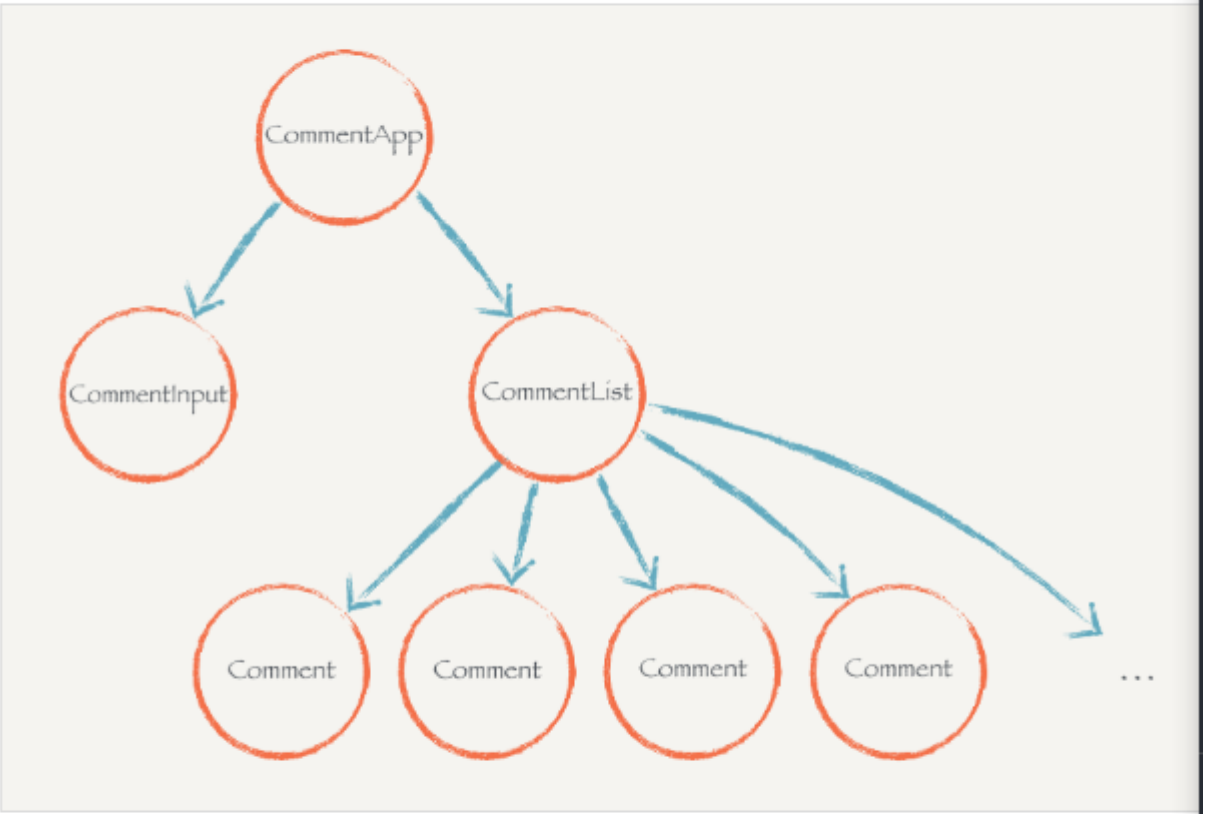
```
class Index extends Component {
  constructor () {
    super()
    this.state = {
      likedText: '已赞',
      unlikedText: '赞'
    }
  }
```

```
9
10    handleClickOnChange () {
11      this.setState({
12        likedText: '取消',
13        unlikedText: '点赞'
14      })
15    }
16
17    render () {
18      return (
19        <div>
20        // 把state中的属性传入子组件
21          <LikeButton
22            likedText={this.state.likedText}
23            unlikedText={this.state.unlikedText} />
24          <div>
25            // 当点击父组件的button时 修改state中的属性 从而修改了子组件中传入的
26            // props
27            <button onClick={this.handleClickOnChange.bind(this)}>
28              修改 wordings
29            </button>
30          </div>
31        </div>
32      )
33    }
34  }
```

给父组件传递数据

向父组件传递数据

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



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

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

state 和 props 的总结

state

> 设置了state的叫做有状态组件

- state 的主要作用是用于组件保存、控制、修改**自己**的可变状态
- 可以认为state是一个局部的，只能被组件自身控制的
- state中状态可以通过this.setState方法进行更新，setState会导致组件的重新渲染

props

> 没有state的组件叫做无状态组件

- props的蛀牙偶哦用时让使用该组件的父组件可以传入参数来配置该组件，他是外部传进来的配置参数
- 租金啊内部无法控制也无法修改，除非外部组件竹筒传入新的props，否则组件的props永远保持不变

列表渲染数据

```
function CommentList ({ comments = [] }) {
  return (
    <div>
      {
        comments.map((item, index) => {
          return <Comment comment = { item } key = {index} />
        } )
      }
    </div>
  )
}

export default CommentList
```

渲染存放JSX元素的数组

挂载阶段的组件生命周期（一）

> React.js 组件渲染，并且构建DOM元素然后塞入页面的过程称为组件的挂载

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

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

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

更新阶段的组件生命周期

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

ref 和 React.js 中的DOM操作

```
class AutoFocusInput extends Component {
  componentDidMount () {
    this.input.focus()
  }

  render () {
    return (
    // 元素在页面挂载完以后，就会调用此函数 传递DOM实列给Input 然后自动获取focus()
      <input ref={(input) => this.input = input} />
    )
  }
}

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

dangerouslySetHTML 和 style 属性

dangerouslySetHTML （动态的插入HTML元素）

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

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

style

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

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

在 React.js 中你需要把 CSS 属性变成一个对象再传给元素：

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

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

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

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

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

组件的命名方法和摆放顺序

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

组件的内容编写顺序

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

高阶组件

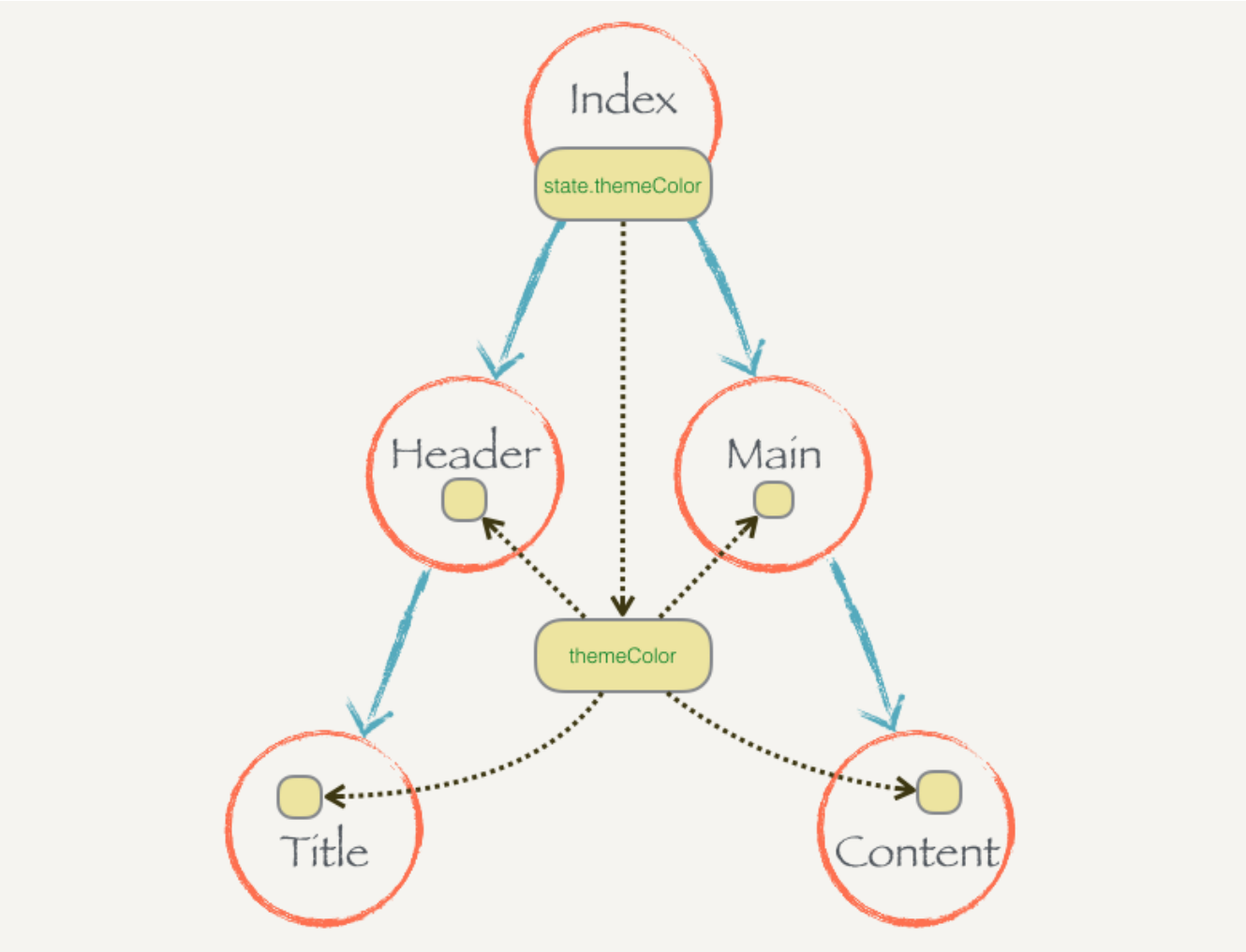
> 高阶组件就是一个函数，传给他一个组件，它返回一个新的组件

> 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

```
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
}
```

React 的 context



`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。

```
class Index extends Component {
  // 验证 getChildContext 返回的对象。
  // 【必写】
  static childContextTypes = {
    themeColor: PropTypes.string
  }

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

  // 【必写】
  getChildContext () {
```

```
    return { themeColor: this.state.themeColor }
  }

  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 {
  // 【使用】
  // 【必写】
  static contextTypes = {
    themeColor: PropTypes.string
  }
  render () {
    return (
      // 获取到context
      <h1 style={{ color: this.context.themeColor }}>
        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')
)
```

动手实现Redux

一 优雅地修改共享状态

- Redux

  > Redux 是一种架构模式，他不关注你到底使用什么库，你可以把他应用到React和Vue

- React-redux

  > 把Redux 这种架构模式和React.js结合起来的一个库，就是Redux

```
// 设置一些状态
const appState = {
  title: {
    text: 'React.js 小书',
    color: 'red',
  },
  content: {
    text: 'React.js 小书内容',
    color: 'blue'
  }
}

// 渲染函数
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
}

// 必须通过此函数进行状态的修改
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
  }
}

// 首次进行渲染
renderApp(appState)

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)
```

抽离store和监控数据变化

抽离store

```

// 设置一些状态
const appState = {
  title: {
    text: 'React.js 小书',
    color: 'red',
  },
  content: {
    text: 'React.js 小书内容',
    color: 'blue'
  }
}

// 渲染函数
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
}

// 必须通过此函数进行状态的修改
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
  }
}

// 返回 state 和 dispatch 合集
function createStore (state, stateChanger) {
  // 新建一个函数集合
  const listeners = []
  // 调用此函数 传入一个函数
  const subscribe = (listener) => {
    console.log(listener)
    listeners.push(listener)
  }
  const getState = () => state
  const dispatch = (action) => {
    stateChanger(state, action)
    // 循环调用函数数组中的函数
    listeners.forEach((listener) => listener())
  }
  return { getState, dispatch, subscribe }
}

// 创建一个state 和 dispath 集合
const store = createStore(appState, stateChanger)
// 传入一个函数去调用渲染函数
store.subscribe(() => renderApp(store.getState()))

// 首次进行渲染
renderApp(store.getState())

// 接下来不需要再进行渲染
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

```

纯函数

> *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。*
>
> - 函数的返回结果只依赖于它的参数
> - 函数执行过程里面没有副作用

函数的返回结果只依赖于它的参数

```
const a = 1
const foo = (b) => a + b
foo(2) // => 3
```

foo 就不是一个纯函数，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

```
const a = 1
const foo = (x, b) => x + b
foo(1, 2) // => 3
```

foo 现在的返回结果只依赖于它的参数x和b，他是纯函数

函数执行过程中没有副作用

> 一个函数执行过程产生了外部可观察的变化，那么就是说这个函数是有副作用的

```
const a = 1
const foo = (obj, b) => {
  return obj.x + b
}
const counter = { x: 1 }
foo(counter, 2) // => 3
counter.x // => 1
```

计算前后counter不会发生任何变化，它就是纯的

```
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
```

计算前后的counter产生了变化，所以它产生了副作用，他不是纯的

> 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。

动手实现React-redux

一 初始化工程

Smart 组件 vs Dumb 组件

- Dumb

  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就是根据props进行渲染

  - 不依赖React。js 和 Dumb 组件以外的内容，不依赖Redux 不 依赖React-redux
- Smart

  > 专门进行数据相关的应用处理，和Ajax 打交道，然后把数据通过props传递给Dumb

划分Smart 和 Dumb 组件

列如一个组件Header.js

此组件依赖了react-redux

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

class Header extends Component {
  static propTypes = {
    themeColor: PropTypes.string
  }

  render () {
    return (
      <h1 style={{ color: this.props.themeColor }}>React.js 小书</h1>
    )
  }
}

const mapStateToProps = (state) => {
  return {
    themeColor: state.themeColor
  }
}
Header = connect(mapStateToProps)(Header)

export default Header
```

1. 在src新建两个目录

```
src/
  components/
  containers/
```

> *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。

2. 新增 src/components/Header.js：

```
// 新增 src/components/Header.js:

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

export default class Header extends Component {
  static propTypes = {
    themeColor: PropTypes.string
  }

  render () {
    return (
      <h1 style={{ color: this.props.themeColor }}>React.js 小书</h1>
    )
  }
}
```

3. 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)
```