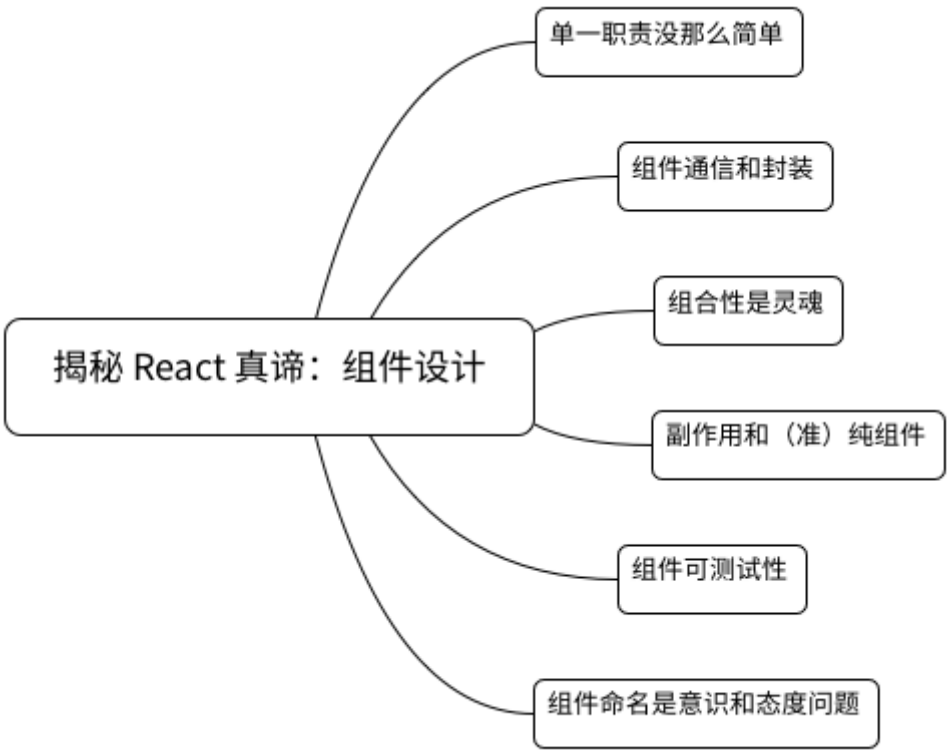
揭秘 React 真谛：组件设计

组件不是 React 特有的概念，但是 React 将组件化的思想发扬光大，可谓用到了极致。良好的组件设计会是良好的应用开发基础，这一讲就让我们谈一谈组件设计的奥秘。

相关知识点如下：



我们将以 React 组件为例，但是其中的设计思想具有共性，不管是其他框架还是原生 Web component 都将适用。

单一职责没那么简单

单一职责我们并不陌生，原则上讲，组件只应该做一件事情。但是对于应用来说，全部组件都拆散，只有单一职责并没有必要，反而增加了编写的繁琐程度。那什么时候需要拆分组件，保证单一职责呢？我认为如果一个功能集合有可能发生变化，那么就需要最大程度地保证单一职责。

单一职责带来的最大好处就是在修改组件时，能够做到全在掌控下，不必担心对其他组件造成影响。举个例子：我们的组件需要通过网络请求获取数据并展示数据内容，这样一来潜在的功能集合改变就有：

请求 API 地址发生变化

请求返回数据格式变化

开发者想更换网络请求第三方库，比如 jQuery.ajax 改成 axios

更改请求数据逻辑

再看一个例子：我们需要一个 table 组件，渲染一个 list，那么潜在更改的可能有：

限制一次性渲染的 item 个数（只渲染前 10 个，剩下的懒加载）

当数据列表为空时显示 「This list is empty」

任何渲染逻辑的更改

这个图很好地说明了问题：

我们来实际看一个场景：

```
import axios from 'axios'

class Weather extends Component {
 constructor(props) {
   super(props)
```

```
    this.state = { temperature: 'N/A', windSpeed: 'N/A' }
  }

  componentDidMount() {
    axios.get('http://weather.com/api').then(response => {
      const { current } = response.data
      this.setState({
        temperature: current.temperature,
        windSpeed: current.windSpeed
      })
    })
  }

  render() {
    const { temperature, windSpeed } = this.state
    return (




Temperature: {temperature} °C


Wind: {windSpeed} km/h




    )
  }
}
```

这个组件很容易理解，并且看上去没什么大问题，但是并不符合单一职责。比如这个 Weather 组件将数据获取与渲染逻辑耦合在一起，如果数据请求有变化，就需要在 componentDidMount 生命周期中进行改动；如果展示天气的逻辑有变化，render 方法又需要变动。

如果我们将这个组件拆分成：WeatherFetch 和 WeatherInfo 两个组件，这两个组件各自只做一件事情，保持单一职责：

```
import axios from 'axios'
import WeatherInfo from './weatherInfo'

class WeatherFetch extends Component {
 constructor(props) {
   super(props)
   this.state = { temperature: 'N/A', windSpeed: 'N/A' }
 }

 componentDidMount() {
   axios.get('http://weather.com/api').then(response => {
     const { current } = response.data
     this.setState({
       temperature: current.temperature,
       windSpeed: current.windSpeed
       })
     })
 }

 render() {
   const { temperature, windSpeed } = this.state
   return (

   )
 }
}
```

另一个文件中：

```
const WeatherInfo = ({ temperature, windSpeed }) =>
 (
```

```
Temperature: {temperature} °C


Wind: {windSpeed} km/h



 )
```

如果我们想进行重构，使用 async/await 代替 Promise，只需要直接更改 WeatherFetch 组件：

```
class WeatherFetch extends Component {
 // ...

 async componentDidMount() {
   const response = await
axios.get('http://weather.com/api')
   const { current } = response.data

   this.setState({
     temperature: current.temperature,
     windSpeed: current.windSpeed
     })
   })
 }

 // ...
}
```

而不会对 WeatherInfo 组件有任何影响。

或者显示风速的逻辑从 Wind: 0 km/h 改为文字描述 Wind: 风平浪静，也只需要改动 WeatherInfo：

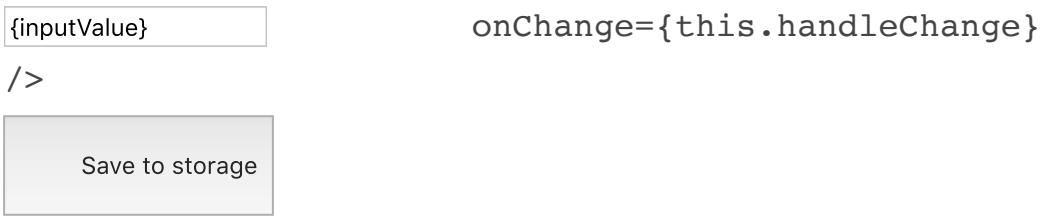
```
const WeatherInfo = ({ temperature, windSpeed }) => {
 const windInfo = windSpeed === 0 ? 'calm' : `${windSpeed}
km/h`
 return (




Temperature: {temperature} °C


Wind: {windSpeed} km/h



 )
}
```

这只是一个简单的例子，在真实项目中，保持组件的单一职责将会非常重要，甚至我们可以使用 HoC 强制组件的单一职责性。

来思考这样的代码：

```
class PersistentForm extends Component {
 constructor(props) {
   super(props)
   this.state = { inputValue:
localStorage.getItem('inputValue') }
   this.handleChange = this.handleChange.bind(this)
   this.handleClick = this.handleClick.bind(this)
 }

 handleChange(event) {
   this.setState({
     inputValue: event.target.value
   })
 }
```

```
 handleClick() {
   localStorage.setItem('inputValue',
this.state.inputValue)
 }

 render() {
   const { inputValue } = this.state
   return (
```




```
   )
 }
}
```

这是一个持久化存储的表单，我们将表单字段内容存储在 localStorage 中，这样不管是刷新页面还是重新进入页面，都会保存上一次点击提交时的内容。可惜 PersistentForm 组件也是包含了两部分职责：存储内容和渲染内容。

这次我们的重构不再是简单的拆分组件，而是使用 HoC 来完成职责单一的实现：

```
class PersistentForm extends Component {
 constructor(props) {
   super(props)
   this.state = { inputValue: props.initialValue }
   this.handleChange = this.handleChange.bind(this)
   this.handleClick = this.handleClick.bind(this)
 }
```

```
  handleChange(event) {
    this.setState({
      inputValue: event.target.value
    })
  }

  handleClick() {
    this.props.saveValue(this.state.inputValue)
  }

  render() {
    const { inputValue } = this.state
    return (
```


```
onChange={this.handleChange}
/>

    )
  }
}
```

我们只是改变了两行代码，初始 state 不再直接读取 localStorage，而是由 this.props.initialValue 提供；handleClick 逻辑调用 this.props.saveValue，而不再直接操作 localStorage，this.props.saveValue 将会由 withPersistence 这个 HoC 提供：

```
function withPersistence(storageKey, storage) {
  return function(WrappedComponent) {
    return class PersistentComponent extends Component {
      constructor(props) {
        super(props)
        this.state = { initialValue:
```

```
storage.getItem(storageKey) }
    }

    render() {
       return (
                       initialValue=
{this.state.initialValue}
           saveValue={this.saveValue}
           {...this.props}
         />
       );
    }

    saveValue(value) {
      storage.setItem(storageKey, value)
    }
  }
 }
}
```

使用方式：

```
const LocalStoragePersistentForm
 = withPersistence('key', localStorage)(PersistentForm)
```

这种方式是组件单一职责和组件复用的结合体现，其他组件当然也可以使用这个 HoC:

```
const LocalStorageMyOtherForm
 = withPersistence('key', localStorage)(MyOtherForm)
```

存储和渲染职责解耦，我们便可以随时切换存储方式，比如切换为 sessionStorage 代替 localStorage：

```
const SessionStoragePersistentForm
 = withPersistence('key', sessionStorage)(PersistentForm)
```

组件通信和封装

另一个和组件职责单一相关的话题是组件的封装，封装又涉及到组件间的通信。因为我们知道，组件再封装，还是要和其他组件去交互通信的，那么当我们说封装时在说些什么呢？

组件关联有紧耦合和松耦合之分，紧耦合是指两个或多个组件之间需要了解彼此的组件内设计，这样的情况是我们不想看到的，这破坏了组件的独立性，「牵一发动全身」。这么看来，松耦合带来的好处是很直接的：

一处组件的改动完全独立，不影响其他组件

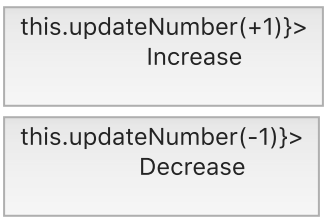
更好的复用设计

更好的可测试性

我们直接来看场景代码，一个简单的计数器足以说明问题：

```
class App extends Component {
 constructor(props) {
   super(props)
   this.state = { number: 0 }
 }

 render() {
   return (


       {this.state.number}



   )
 }
}
```

```
class Controls extends Component {
 updateNumber(toAdd) {
   this.props.parent.setState(prevState => ({
     number: prevState.number + toAdd
   }))
 }

 render() {
   return (
```




```
   )
 }
}
```

这样的组件实现问题很明显：App 组件不具有封装性，它将实例传给 Controls 组件，Controls 组件可以直接更改 App state 的内容。事实上，我们并不是不允许 Controls 组件修改 App 组件，只是 Controls 组件直接调用 App 组件的 setState 方法是不被建议的，因为 Controls 组件如果要调用 App 的 setState，就得需要知道 App 组件 state 的结构，需要感知 this.props.parent.state.number 等详情。

同时上述代码也不利于测试，这个我们将在后面进行说明。那么该如何重构呢？我们应该用更加「含蓄」或者更加「粗暴、直接地」方式修改 number 值。秉承封装性：「只有组件自己知道自己的 state 结构」，将 updateNumber 迁移至 App 组件内：

```
class App extends Component {
 constructor(props) {
   super(props)
   this.state = { number: 0 }
```

```
  }

  updateNumber(toAdd) {
    this.setState(prevState => ({
      number: prevState.number + toAdd
    }))
  }

  render() {
    return (


        {this.state.number}
                  onIncrease={() => this.updateNumber(+1)}
          onDecrease={() => this.updateNumber(-1)}
        />


    )
  }
}


const Controls = ({ onIncrease, onDecrease }) => {
  return (


      Increase
      Decrease


  )
}
```

这样一来，Controls 组件就不需要再知道 App 组件的内部情况，实现了更好的复用性和可测试性，App 组件因此也具有了更好的封装性。

组合性是灵魂

如果说组件单一职责确定了如何拆分组件，封装性明确了组件如何组织，那么组合性就完成了整个应用的拼接。

React 具有天生的组合基因：

对应声明式代码：

```
const app = (
```

```
)
```

如果两个组件 Composed1 和 Composed2 具有相同的逻辑，我们可以使用组合性进行拆分重组：

```
const instance1 = (

     // Composed1 逻辑
     // 重复逻辑

)
const instance2 = (

     // 重复逻辑
     // Composed2 逻辑

)
```

重复逻辑提取为 Common 组件：

```
const instance1 = (




)
const instance2 = (




)
```

另外一个典型应用就是 render prop 模式，这个我们前面已经介绍过，这里给出一个很简单的示例，具体不再展开：

```
const ByDevice = ({ children: { mobile, other } }) =>
{
 return Utils.isMobile() ? mobile : other
}


   {{
     mobile:

Mobile detected!
,
     other:
Not a mobile device

   }}
```

副作用和（准）纯组件

纯函数和非纯函数概念大家并不陌生，简单来说，通过函数参数能够唯一确定函数返回值的函数，我们称之为纯函数，反之就是有副作用的非纯函数。纯/非纯函数延伸到组件中，就是纯/非纯组件。

在理想主义者眼中，最好的情况是应用组件全部由纯组件组成，这样对于组件的调试和强健性非常重要。但这只能是理想情况，在真实环境中，我们需要发送网络请求以获取数据（副作用，因为数据不固定，需要从网络获取），进行条件渲染等操作，如何最大限度地保证纯组件或者（准）纯组件呢？我们先来下一个定义：

（准）纯组件是渲染数据全部来自于 props，但是会产生副作用的组件

从非纯组件中提取纯组件部分，是一个很常见有效的做法。

```
const globalConfig = {
 siteName: 'Animals in Zoo'
```

```
}

const Header = ({ children }) => {
 const heading =
   globalConfig.siteName ?

{globalConfig.siteName}

: null
 return (


      {heading}
      {children}


 );
}
```

这个组件是典型的非纯组件，因为它依赖全局变量 siteName，可能渲染出：

Animals in Zoo

Some content

或者：

Some content

在编写测试用例时，还需要考虑 globalConfig.siteName，使得逻辑更加复杂：

```
import assert from 'assert'
import { shallow } from 'enzyme'
import { globalConfig } from './config'
import Header from './Header'

describe('

', function() {
 it('should render the heading', function() {
   const wrapper = shallow(

Some content

   )
   assert(wrapper.contains(
```

Animals in Zoo

```
))
 })

 it('should not render the heading', function() {
   // 改动全局变量
   globalConfig.siteName = null
   const wrapper = shallow(

Some content

   )
   assert(appWithHeading.find('h1').length === 0)
 })
})
```

在测试 Header 组件时，多了一种 case 不说，我们还需要手动改写全局变量的值。

一个常用的优化方式是使全局变量作为 Header 的 props 出现，而不再是一个外部变量，那么函数式组件 Header 就完全依赖其参数：

```
const Header = ({ children, siteName }) => {
 const heading = siteName ?

{siteName}

: null;
 return (

      {heading}
      {children}

 );
}

Header.defaultProps = {
 siteName: globalConfig.siteName
}
```

这样一来 Header 就成了纯组件，测试用例便可以简化为：

```
import assert from 'assert'
import { shallow } from 'enzyme'
import { Header } from './Header';

describe('

', function() {
 it('should render the heading', function() {
   const wrapper = shallow(

Some content
```

```
    )
    assert(wrapper.contains(

Animals in Zoo

))
 });

 it('should not render the heading', function() {
    const wrapper = shallow(

Some content

    )
    assert(appWithHeading.find('h1').length === 0)
 })
})
```

且不需再手动改动变量的值，以完成测试逻辑。

另一个重构非纯组件的典型案例就是针对有网络请求的副作用情况，重放我们在组件单一职责中的代码：

```
import axios from 'axios'
import WeatherInfo from './weatherInfo'

class WeatherFetch extends Component {
 constructor(props) {
    super(props)
    this.state = { temperature: 'N/A', windSpeed: 'N/A'
}
 }

 componentDidMount() {
    axios.get('http://weather.com/api').then(response
=> {
```

```
      const { current } = response.data
      this.setState({
        temperature: current.temperature,
        windSpeed: current.windSpeed
        })
      })
  }

  render() {
    const { temperature, windSpeed } = this.state
    return (

    )
  }
}
```

从表面上看，WeatherFetch 组件不得不「非纯」，因为网络请求不可避免，但是我们可以将请求的主体逻辑分离出组件，而组件只负责调用请求，这样的操作我称之为「（准）纯组件」：

```
import { connect } from 'react-redux'
import { fetch } from './action'

export class WeatherFetch extends Component {
  render() {
    const { temperature, windSpeed } = this.props
    return (

    )
  }

  componentDidMount() {
    this.props.fetch()
  }
}

function mapStateToProps(state) {
```

```
 return {
   temperature: state.temperate,
   windSpeed: state.windSpeed
 }
}

export default connect(mapStateToProps, { fetch })
```

我们使用 Redux 来完成，这样一来 WeatherFetch 组件至少可以保证「相同的 props，会渲染相同的结果」。测试也就变得可行：

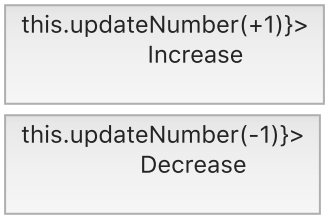
```
import assert from 'assert'
import { shallow, mount } from 'enzyme'
import { spy } from 'sinon'

import { WeatherFetch } from './WeatherFetch';
import WeatherInfo from './WeatherInfo'

describe('', function() {
 it('should render the weather info', function() {
   function noop() {}
   const wrapper = shallow(

   )
   assert(wrapper.contains(

   ))
 });

 it('should fetch weather when mounted', function() {
   const fetchSpy = spy()
   const wrapper = mount(

   )
   assert(fetchSpy.calledOnce)
 })
})
```

组件可测试性

我们一直在提「可测试性」，上面也出现了测试用例代码，我认为是否具有测试意识，是区别高级和一般程序员的考证之一。

还记得我们在上面提到的 Controls 组件吗？最初实现：

```
class App extends Component {
 constructor(props) {
   super(props)
   this.state = { number: 0 }
 }

 render() {
   return (



       {this.state.number}




   )
 }
}

class Controls extends Component {
 updateNumber(toAdd) {
   this.props.parent.setState(prevState => ({
     number: prevState.number + toAdd
   }))
 }

 render() {
   return (
```




```
    )
  }
}
```

因为 Controls 组件的行为完全依赖其父组件，因此为了测试，我们需要临时构造一个父组件 Temp 来完成：

```
class Temp extends Component {
  constructor(props) {
    super(props)
    this.state = { number: 0 }
  }
  render() {
    return null
  }
}

describe('', function() {
  it('should update parent state', function() {
    const parent = shallow()
    const wrapper = shallow()

    assert(parent.state('number') === 0)

    wrapper.find('button').at(0).simulate('click')
    assert(parent.state('number') === 1)

    wrapper.find('button').at(1).simulate('click')
    assert(parent.state('number') === 0)
```

```
 });
});
```

测试编写的非常痛苦，而经过我们重构之后，变的就非常简单了：

```
class App extends Component {
 constructor(props) {
   super(props)
   this.state = { number: 0 }
 }

 updateNumber(toAdd) {
   this.setState(prevState => ({
     number: prevState.number + toAdd
   }))
 }

 render() {
   return (


       {this.state.number}
                                        onIncrease={()  =>
this.updateNumber(+1)}
         onDecrease={() => this.updateNumber(-1)}
       />


   )
 }
}


const Controls = ({ onIncrease, onDecrease }) => {
 return (
```

```
        Increase
        Decrease


 )
}

describe('', function() {
  it('should execute callback on buttons click',
function() {
   const increase = sinon.spy()
   const descrease = sinon.spy()
   const wrapper = shallow(

   )

   wrapper.find('button').at(0).simulate('click')
   assert(increase.calledOnce)
   wrapper.find('button').at(1).simulate('click')
   assert(descrease.calledOnce)
 })
})
```

有的开发者觉得「测试不重要」，因此也不用关心组件编写的可测试性。其实我认为，之所以会有程序员认为「测试不重要」，是因为他不具有看待项目的更高视野和角度，也没有编写稳定可靠组件库或其他库的经验。我们要端正态度，想要进阶，就要从态度入手，从掌握一门测试用例的使用入手。

组件命名是意识和态度问题

我为什么要把组件命名放在最后一部分呢？因为组件命名太简单了，任何一个开发者只要有意识，能用心，都能完成很好的命名；同时组件命名又太重要了，良好的组件命名就是「行走着的注释」。但意识是一个很虚的概念，有的程序员也许天生就不具备，有的程序员即便具备了，也懒得去琢磨。这里，我不赘述太多道理，读者只需观察两段代码即可，其中第一段，我加了大量的注释辅助：

```
// 返回一组 game 信息
// data 是一个数组，包含了所有 game 信息
function Games({ data }) {
 // 选出前 10 条 games
 const data1 = data.slice(0, 10)
 // list 是包含了 10 条 games 的 Game 组件集合
 const list = data1.map(function(v) {
   // v 代码当前 game
   return
 })
 return

{list}


}

  data=[{ id: 1, name: 'Mario' }, { id: 2, name: 'Doom'
}]
/>
```

第二段代码不需要一行注释：

```
const GAMES_LIMIT = 10

const GamesList = ({ items }) => {
 const itemsSlice = items.slice(0, GAMES_LIMIT)
 const games = itemsSlice.map(gameItem =>

 )
 return

{games}


}
```

```
 items=[{ id: 1, name: 'Mario' }, { id: 2, name: 'Doom'
}]
/>
```

组件设计功力，其实一个命名就能看出来；在做 code review 时，一个命名也能出卖你的深浅。

总结

本讲我们剖析了组件设计的基本原则，在原则范畴内，展现了组件的灵活性，并将组件复用性融汇在课程中。其实不光 React 组件如此，任何框架的组件也都是如此，超脱于组件范畴之外，API 设计也应用着相同的原则。这是编程最本质的思想，甚至从某种程度上，在编程之外，原子组建成大千世界的哲学道理都是异曲同工的。

点击查看下一节

揭秘 React 真谛：数据状态管理