

复习

本节内容

课堂目标

使用ant-Design

普通方式使用

高级配置

聪明组件VS傻瓜组件

组件组合而非继承

高阶组件

高阶组件链式调用

高阶组件装饰器写法

高阶组件应用

权限控制

页面复用

组件通信Context

何时使用Context

封装antd的form表单

复习

1. react基本用法
2. JSX
3. 状态State+setState({}) / setState(prev=>({}))
4. 组件的两种形式class ,function
5. 组件交互props
6. 生命周期
7. 受控组件和非受控组件

本节内容

课堂目标

使用ant-Design

[官方网站](#)

普通方式使用

下载

```
1 npm install antd --save
```

修改 `src/App.js`，引入 antd 的按钮组件。

```
1 import React, { Component } from 'react';
2 import Button from 'antd/es/button';
3 import './App.css';
4 import 'antd/dist/antd.css'
5
6 class App extends Component {
7   render() {
8     return (
9       <div className="App">
10         <Button type="primary">Button</Button>
11       </div>
12     );
13   }
14 }
15
16 export default App;
```

修改 `src/App.css`，在文件顶部引入 `antd/dist/antd.css`。

高级配置

上面的配置不太友好,在实际开发中会带来很多问题.例如上面的加载样式是加载了全部的风格

此时我们需要使用 `react-app-rewired` (一个对 `create-react-app` 进行自定义配置的社区解决方案).引入 `react-app-rewired` 并修改 `package.json` 里的启动配置。由于新的 [react-app-rewired@2.x](#) 版本的关系, 你还需要安装 [customize-cra](#)。

```
1 npm i react-app-rewired customize-cra babel-plugin-import --save-dev
```

修改`package.json`的启动文件

```
1 "scripts": {  
2   "start": "react-app-rewired start",  
3   "build": "react-app-rewired build",  
4   "test": "react-app-rewired test",  
5   "eject": "react-app-rewired eject"  
6 },
```

然后在项目根目录创建一个 `config-overrides.js` 用于修改默认配置

`babel-plugin-import` 是一个用于按需加载组件代码和样式的 `babel` 插件

```
1  const { override, fixBabelImports } =  
    require('customize-cra');  
2  module.exports = override(  
3      fixBabelImports('import', {  
4          libraryName: 'antd',  
5          libraryDirectory: 'es',  
6          style: 'css',  
7      })),  
8  );
```

修改App.js

```
1  // import Button from 'antd/es/button';  
2  // import 'antd/dist/antd.css'  
3  
4  import { Button } from 'antd';
```

运行

```
1  npm start
```

聪明组件VS傻瓜组件

基本原则:聪明组件(容器组件)负责数据获取,傻瓜组件(展示组件)负责根据props显示信息内容

优势:

1. 逻辑和内容展示分离
2. 重用性高
3. 复用性高
4. 易于测试

在 `CommentList.js`

```

1 import React, {
2   Component
3 } from 'react';
4
5 function Comment({ comment }) {
6   console.log('render');
7   return (
8     <div>
9       <p>{comment.id}</p>
10      <p>{comment.content}</p>
11      <p>{comment.author}</p>
12    </div>
13  )
14 }
15
16 class CommentList extends Component {
17   constructor(props) {
18     super(props);
19     this.state = {
20       comments: []
21     }
22   }
23   componentDidMount() {
24     // 获取数据
25     setTimeout(() => {
26       this.setState({
27         comments: [
28           {
29             id: 1,
30             content: 'react非常好',
31             author: 'facebook'
32           },
33           {
34             id: 2,

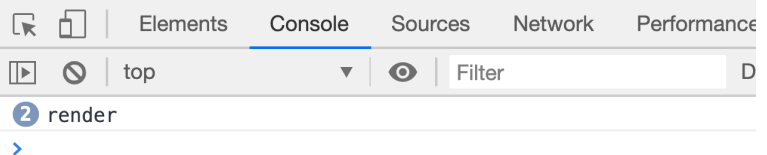
```

```

35         content: 'vue比你好',
36         author: '尤雨溪'
37     }
38 ]
39 })
40 }, 1000);
41 }
42
43 render() {
44     return (
45         <div>
46             {
47                 this.state.comments.map((item, i)
=> (
48                     <Comment comment={item} key=
{item.id} />
49                 ))
50             }
51
52         </div>
53     );
54 }
55 }
56
57 export default CommentList;

```

1
react非常好
facebook
2
vue比你好
尤雨溪



一个展示性的组件打印了两次render,这是因为数据有两条,渲染两次
我现在做一个轮训,每1秒让数据更新一次

```
1 // 每隔1秒钟,就开始更新一次
2 setInterval(() => {
3   this.setState({
4     comments: [
5       {
6         id: 1,
7         content: 'react非常好',
8         author: 'facebook'
9       },
10      {
11        id: 2,
12        content: 'vue比你',
13        author: '尤雨溪'
14      }
15    ]
16  })
17 }, 1000);
```

效果显示:



会发现,每次轮训,render都会被打印,也就是说傻瓜式组件被重新渲染了

思考:数据没有发生变化,我需要更新么?哪怕react再聪明,它们之间也会diff算法对比,是很消耗性能的

解决方法1:使用shouldComponentUpdate

将comment扩展成类组件,定义以下方法

```

1 shouldComponentUpdate(nextProps){
2     if(nextProps.comment.id === this.props.comment.id){
3         // return false表示不更新
4         return false;
5     }else{
6         return true;
7     }
8 }

```

解决方法2:使用React.PureComponent

React.PureComponent 与 shouldComponentUpdate() 很相似。两者的区别在于 React.PureComponent 并未实现 shouldComponentUpdate(), 而 React.PureComponent 中以浅层比较prop和state的方式来实现该函数

如果对象中包含复杂的数据结构, 则有可能因为无法检查深层的差别, 产生错误的比对结果。仅在你的 props 和 state 较为简单时, 才使用 React.PureComponent

修改 CommentList.js

```

1 render() {
2     return (
3         <div>
4             {/* {
5                 this.state.comments.map((item, i)
6                     <Comment id={item.id} content=
7                     {item.content} author={item.author} key={item.id} />
8                     ))
9                 } */}
10             {/*或者可以这样写*/}

```



```

11         {
12             this.state.comments.map((item, i) => (
13                 <Comment {...item} key={item.id} />
14             ))
15         }
16     </div>
17 );
18 }

```

修改 `Comment.js`

```

1 class Comment extends PureComponent{
2     constructor(props) {
3         super(props);
4     }
5     render() {
6         console.log('render');
7         return (
8             <div>
9                 <p>{this.props.id}</p>
10                <p>{this.props.content}</p>
11                <p>{this.props.author}</p>
12            </div>
13        )
14    }
15 }

```

解决方法3：使用 `React.memo`

`React.memo` 为[高阶组件](#)。它与 `React.PureComponent` 非常相似，但它适用于函数组件，但不适用于 class 组件。

```

1  const Comment = React.memo(({id,content,author})=>{
2      return (
3          <div>
4              <p>{id}</p>
5              <p>{content}</p>
6              <p>{author}</p>
7          </div>
8      )
9  })

```

修改 `CommentList.js`

```

1  class CommentList extends Component{
2      render() {
3          return (
4              <div>
5                  {
6                      this.state.comments.map((item, i)
=> (
7                          <Comment id={item.id} content=
{item.content} author={item.author} key={item.id} />
8                      ))
9                  }
10             </div>
11         );
12     }
13 }

```

组件组合而非继承

官方的原话是这样说的：

React 有十分强大的组合模式。我们推荐使用组合而非继承来实现组件间的代码重用。

/components/Compound.js

```
1 import React, { Component } from 'react';
2 import {
3   Button
4 } from "antd";
5 function Dialog(props) {
6   return (
7     <div style={{ border: `3px solid ${props.color
8 || 'blue'}` }}>
9       /*等价于: vue的匿名插槽*/
10      {props.children}
11      /*具名插槽*/
12      <div>
13        {props.btn}
14      </div>
15    </div>
16  )
17 }
18 // 将任意的组件作为子组件传递
19 function WelcomeDialog() {
20   const confirmBtn = <Button type='primary' onClick=
21   (() => { alert('react真的好') })>确定</Button>
22   return (
23     <Dialog color='green' btn={confirmBtn}>
24       <h3>welcome</h3>
25       <p>
26         欢迎光临
27       </p>
28     </Dialog>
29   )
30 }
31 class Compound extends Component {
32   render() {
```

```

31         return (
32             <div>
33                 <WelcomeDialog />
34             </div>
35         );
36     }
37 }
38
39 export default Compound;

```

高阶组件

组件设计的目的：保证组件功能的单一性

```

1  // 高阶组件
2  本质是一个函数
3  函数接收一个一个组件，返回一个新的组件 则Comment为高阶组件
4  好比是：我给你一个赛亚人，你给我一个超级赛亚人
5
6  // 高阶函数
7  定义：接收的参数是函数或者返回值是函数
8  常见的：数组遍历相关的方法 、定时器、Promise /高阶组件
9  作用：实现一个更加强大，动态的功能

```

高阶组件 (higher-ordercomponent) (HOC) 是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API 的一部分，它是一种基于 React 的组合特性而形成的设计模式。

具体而言，高阶组件是参数为组件，返回值为新组件的函数

/components/Hoc.js

```

1  const HightOrderCom = (Comp) => {
2      // 返回值为新组件
3      const NewComponent = function (props) {
4          return <Comp name='react' content='高级组件的使
用' {...props}></Comp>
5      }
6      return NewComponent;
7  }
8

```

上面的 `HightOrderCom` 组件，其实就是代理了 `Comp` 只是多传递了 `name` 和 `content` 参数

```

1  import React, { Component } from 'react'
2
3  function MyNormalCom(props) {
4      return (
5          <div>
6              <p>课程名字:{props.name}</p>
7              <h3>课程内容:{props.content}</h3>
8          </div>
9      )
10 }
11
12 // 高级组件
13 const HighOrderCom = (Comp) => {
14     return (props)=>{
15         return <Comp name='react' content='高阶组件的使用'
16             {...props}></Comp>
17     }
18 }
19 export default HighOrderCom(MyNormalCom);

```

重写高阶组件内部的生命周期

```
1  const HightOrderCom = (Comp) => {
2    // 返回值为新组件
3    return class extends Component {
4      constructor(props) {
5        super(props)
6      }
7      componentDidMount(){
8        console.log('发起ajax请求');
9      }
10     }
11     render() {
12       return (
13         <Comp name='react' content='高级组件的使
14 用' {...this.props}></Comp>
15       );
16     }
17   }
```

高阶组件链式调用

```
1  // 打印日志的高阶组件
2  const WithLog = (Comp)=>{
3    console.log(Comp.name + '渲染了');
4    const MyComponent = function(props) {
5      return <Comp {...props}></Comp>
6    }
7    return MyComponent;
8  }
```

调用:

```
1 | const HOC =  
   HightOrderCom(WithLog(WithLog(MyNormalCom)));
```



高阶组件装饰器写法

上面链式写法非常蛋疼，逻辑也比较绕，ES7中有一个优秀的语法:装饰器，专门用于处理这种问题

```
1 | cnpm install --save-dev babel-plugin-transform-  
   decorators-legacy @babel/plugin-proposal-decorators
```

配置修改:

```
1 | const {  
2 |   override,  
3 |   fixBabelImports, //按需加载配置函数  
4 |   addBabelPlugins //babel插件配置函数  
5 | } = require('customize-cra');  
6 | module.exports = override(  
7 |   fixBabelImports('import', {  
8 |     libraryName: 'antd',  
9 |     libraryDirectory: 'es',  
10 |    style: 'css',  
11 |  }),  
12 |   addBabelPlugins( // 支持装饰器  
13 |     [  
14 |       '@babel/plugin-proposal-decorators',  
15 |       {  
16 |         legacy: true  
17 |       }  
18 |     ]  
19 |   )  
20 | );
```

```
18     ]
19   )
20 );
```

```
1  const HightOrderCom = (Comp) => {
2    // 返回值为新组件
3    return class extends Component {
4      constructor(props) {
5        super(props)
6      }
7      componentDidMount() {
8        console.log('发起ajax请求');
9      }
10     }
11     render() {
12       return (
13         <Comp name='react' content='高级组件的使
14 用' {...this.props}></Comp>
15       );
16     }
17   }
18   // 打印日志的高阶组件
19   const WithLog = (Comp) => {
20     console.log(Comp.name + '渲染了');
21     const MyComponent = function (props) {
22       return <Comp {...props}></Comp>
23     }
24     return MyComponent;
25   }
26   @withLog
27   @HighOrderCom
28   @withLog
29   class Hoc extends Component {
```



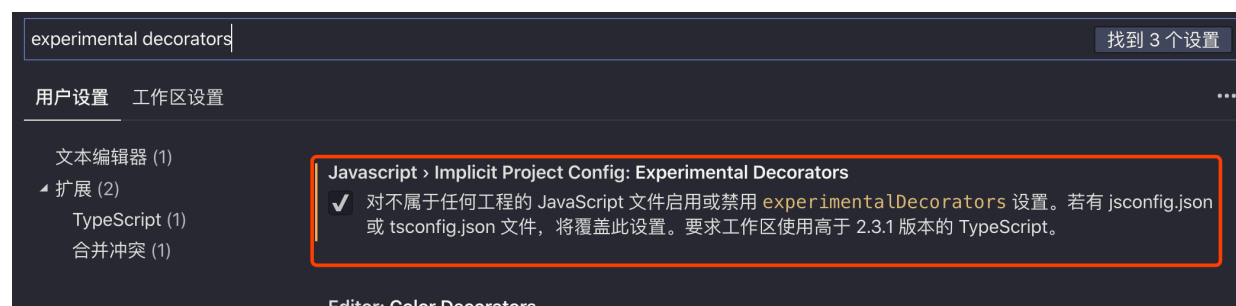
```

30     constructor(props){
31         super(props);
32     }
33     render() {
34         return (
35             <div>
36                 <p>课程名字:{this.props.name}</p>
37                 <h3>课程内容:{this.props.content}</h3>
38             </div>
39         );
40     }
41 }
42 }
43
44 export default Hoc;

```

解决vscode中红色警告

在编辑器中左下角找到齿轮按钮，点击按钮找到，找到设置。你会出现两种界面： 1、 在编辑界面输入experimental decorators，讲如图选项打钩即可



高阶组件应用

权限控制

利用高阶组件的 条件渲染 特性可以对页面进行权限控制

/components/HocApp.js

```
1 import React, { Component } from 'react'
2 // 权限控制
3 // 不谈场景的技术就是在耍流氓
4 export const withAdminAuth = (role) => (WrappedComp) =>
5 {
6   return class extends Component {
7     constructor(props) {
8       super(props);
9       this.state = {
10         isAdmin: false
11       }
12     }
13     componentDidMount() {
14       setTimeout(() => {
15         this.setState({
16           isAdmin: role === 'Admin'
17         })
18       }, 1000);
19     }
20
21     render() {
22       if (this.state.isAdmin) {
23         return (
24           <WrappedComp {...this.props}></WrappedComp>
25         );
26       } else {
27         return (
28           <div>您没有权限查看该页面，请联系管理员</div>
29         )
30       }
31     }
32   }
33 }
```

然后是两个页面：

/components/PageA.js

```
1 import React, { Component } from 'react'
2 import { withAdminAuth } from "../HocApp";
3 class PageA extends Component {
4   componentDidMount() {
5     setTimeout(() => {
6       this.setState({
7         isAdmin: true
8       })
9     }, 1000);
10  }
11  render() {
12    return (
13      <div>
14        <h2>我是页面A</h2>
15      </div>
16    )
17  }
18 }
19 export default withAdminAuth('Admin')(PageA)
```

/components/PageB.js

```
1 import React, { Component } from 'react'
2 import { withAdminAuth } from "../HocApp";
3 class PageB extends Component {
4   render() {
5     return (
6       <div>
7         <h2>我是页面B</h2>
8       </div>
9     )
10  }
11 }
12 export default withAdminAuth()(PageB);
13
```

页面A有权限访问， 页面B无权限访问

页面复用

组件通信Context

Context 提供了一个无需为每层组件手动添加 props，就能在组件树间进行数据传递的方法。

在一个典型的 React 应用中，数据是通过 props 属性自上而下（由父及子）进行传递的，但这种做法对于某些类型的属性而言是极其繁琐的（例如：地区偏好，UI 主题），这些属性是应用程序中许多组件都需要的。Context 提供了一种在组件之间共享此类值的方式，而不必显式地通过组件树的逐层传递 props。

何时使用Context

Context设计的目的是为了共享那些 **全局** 的数据，例如当前认证的用户、主题等。举个例子：

```
1 import React, { Component } from 'react';
2 // Context可以让我们传遍每一个组件，就能将值深入组件树中
3 // 创建一个ThemeContext，默认值为light
4 const ThemeContext = React.createContext('lighter');
5
6 class ThemeButton extends Component {
7     // 指定 contextType 读取当前的 theme context。
8     // React 会往上找到最近的 theme Provider，然后使用它
    的值。
9     // 在这个例子中，当前的 theme 值为 “dark”。
10    // 第一种渲染的方式(下面通过this.context渲染数
    据):static contextType = ThemeContext;
11    render() {
12        return (
13            // <button type={this.context}></button>
14            <ThemeContext.Consumer>
15                {/* 2.基于函数去渲染 value等价于
    this.context */}
16                {
17                    value => <button theme=
    {value.theme}>{value.name}</button>
18                }
19                </ThemeContext.Consumer>
20            );
21        }
22    }
23    function Toolbar(props) {
24        // Toolbar 组件接受一个额外的“theme”属性，然后传递给
    ThemedButton 组件。
25        // 如果应用中每一个单独的按钮都需要知道 theme 的值，这
    会是件很麻烦的事，
26        // 因为必须将这个值层层传递所有组件。
27        return (
28            <div>
```

```

29         <ThemeButton></ThemeButton>
30     </div>
31 )
32
33 }
34 //传递的数据
35 const store = {
36     theme: 'dark',
37     name: '按钮'
38 }
39 class ContextSimple extends Component {
40     // 使用Provider将当前的value='dark'深入到组件树中，无
    论多深，任何组件都能读取这个值
41     render() {
42         return (
43             <div>
44                 <ThemeContext.Provider value={store}>
45                     <Toolbar></Toolbar>
46                 </ThemeContext.Provider>
47             </div>
48         );
49     }
50 }
51
52 export default ContextSimple;

```

高阶组件装饰器写法

```

1  import React, { Component } from 'react';
2  const ThemeContext = React.createContext('lighter');
3  const withConsumer = Comp => {
4      return class extends Component {
5          constructor(props) {
6              super(props);

```

```

7
8     }
9     render() {
10         return (
11             <ThemeContext.Consumer>
12                 {
13                     value => <Comp {...this.props}
value={value}></Comp>
14                 }
15             </ThemeContext.Consumer>
16         );
17     }
18 }
19 }
20 @withConsumer
21 class ThemeButton extends Component {
22     constructor(props) {
23         super(props);
24
25     }
26     render() {
27         return (
28             <button theme={this.props.value.theme}>
{this.props.value.name}</button>
29         );
30     }
31 }
32 function Toolbar(props) {
33     return (
34         <div>
35             <ThemeButton></ThemeButton>
36         </div>
37     )
38 }
39 //传递的数据

```

```

40 const store = {
41     theme: 'dark',
42     name: '按钮'
43 }
44
45 const withProvider = Comp => {
46     return function (props) {
47         return (
48             <ThemeContext.Provider value={store}>
49                 <Comp {...props} />
50             </ThemeContext.Provider>
51         )
52     }
53 }
54 @withProvider
55 class ContextSimple extends Component {
56     render() {
57         return (
58             <div>
59                 <Toolbar></Toolbar>
60             </div>
61         );
62     }
63 }
64
65 export default ContextSimple;

```

封装antd的form表单

打开[antd官网](#)

```

1 import React from 'react'
2 import { Form, Icon, Input, Button } from 'antd';
3

```



```

4 function hasErrors(fieldsError) {
5     return Object.keys(fieldsError).some(field =>
fieldsError[field]);
6 }
7
8 class HorizontalLoginForm extends React.Component {
9     componentDidMount() {
10         // To disabled submit button at the beginning.
11         this.props.form.validateFields();
12     }
13     handleSubmit = e => {
14         e.preventDefault();
15         this.props.form.validateFields((err, values) =>
{
16             if (!err) {
17                 console.log('Received values of form:
', values);
18             }
19             });
20     };
21
22     render() {
23         const { getFieldDecorator, getFieldsError,
getFieldError, isFieldTouched } = this.props.form;
24
25         // Only show error after a field is touched.
26         const usernameError = isFieldTouched('username') &&
getFieldError('username');
27         const passwordError = isFieldTouched('password') &&
getFieldError('password');
28         return (
29             <Form layout="inline" onSubmit=
{this.handleSubmit}>
30                 <Form.Item validateStatus={usernameError ?
'error' : ''} help={usernameError || ''}>

```

```

31         {getFieldDecorator('username', {
32             rules: [{ required: true, message:
33 'Please input your username!' }]},
34         ))(
35             <Input
36                 prefix={<Icon type="user"
37 style={{ color: 'rgba(0,0,0,.25)' }} />}
38                 placeholder="Username"
39             />,
40         )}
41     </Form.Item>
42     <Form.Item validateStatus={passwordError ?
43 'error' : ''} help={passwordError || ''}>
44         {getFieldDecorator('password', {
45             rules: [{ required: true, message:
46 'Please input your Password!' }]},
47         ))(
48             <Input
49                 prefix={<Icon type="lock"
50 style={{ color: 'rgba(0,0,0,.25)' }} />}
51                 type="password"
52                 placeholder="Password"
53             />,
54         )}
55     </Form.Item>
56     <Form.Item>
57         <Button type="primary"
58             htmlType="submit" disabled=
59 {hasErrors(getFieldsError())}>
60             Log in
61         </Button>
62     </Form.Item>
63 </Form>
64 );
65 }

```

```
59 | }
60
61 const WrappedHorizontalLoginForm = Form.create({ name:
    'horizontal_login' })(HorizontalLoginForm);
62
63 ReactDOM.render(<WrappedHorizontalLoginForm />,
    mountNode);
```

组件功能分析

- 每个input输入框被触发后开始做非空校验并提示错误
- 表单提交时做表单项校验，全部校验成功则提示登录，否则提示校验失败
- 表单项有前置图标

组件封装思路

1. 需要一个高阶函数HOC, `MFormCreate`，主要负责包装用户表单，增加数据管理能力；它需要扩展4个功能：`getFieldDecorator`, `getFieldsError`, `getFieldError`, `isFieldTouched`。获取字段包装器方法 `getFieldDecorator` 的返回值是个高级函数，接收一个Input组件作为参数，返回一个新组件。这就是让一个普通的表单项，变成了带有扩展功能的表单项(例如：增加该项的校验规则)
2. `FormItem`组件，负责校验及错误信息的展示，需要保存两个属性，校验状态和错误信息，当前校验通过时错误信息为空
3. `Input`组件，展示型组件，增加输入框前置icon
4. 导出 `MFormCreate` 装饰后的 `MForm` 组件，`MForm` 组件负责样式布局以及提交控制

组件封装步骤

1. 完成一个基础的组件 `MForm`, 页面展示
2. 编写高阶组件 `MFormCreate` 对 `MForm` 进行扩展，让 `MForm` 组件拥有管理数据的能力。

1. 保存字段选项设置 `this.options = {}`; 这里不需要保存为 `state`, 因为我们不希望字段选项变化而让组件重新渲染
 2. 保存各字段的值 `this.state = {}`
 3. 定义方法 `getFieldDecorator()`, 第一个参数传递配置项, 第二个参数传入 `Input` 组件; 第一个参数包括: 当前校验项、校验规则 `'username',{rules:[require:true,message:'请输入用户名']}`
 4. 在 `MFormCreate` 中, 克隆一份 `Input` 组件, 并且定义 `Input` 的 `onChange` 事件。首先这里需要把已经存在的 `jsx` 克隆一份, 并修改它的属性, 直接修改属性是不允许的; 这里在更高级别定义 `onChange` 事件, 控制元素的值, 这样当组件发生变化时, 就不用进行组件之间的来回通信。数据变化交给容器型组件去做, 低层级的组件只负责展示即可。
3. 增加提交校验功能
 4. 增加 `FormItem` 组件, 在表单项触发后做实时校验并提示错误信息

MForm.js

```
1 import React, { Component } from 'react';
2
3 class MForm extends Component {
4   render() {
5     return (
6       <div>
7         <input type="text" />
8         <input type="password"/>
9         <input type="submit" value='登录'/>
10      </div>
11    );
12  }
13 }
14
15 export default MForm;
```

使用高阶组件MFormCreate对MForm组件进行扩展； 通过表单项组件FormItem展示校验错误信息

```
1 import React, { Component } from 'react';
2 // 包装用户表单，增加数据管理能力以及校验功能
3 const MFormCreate = function (Comp) {
4   return class extends Component {
5     constructor(props) {
6       super(props);
7       this.state = {}; // 保存各字段的值
8       this.options = {}; // 保存字段选项设置 不希望
      它的变化让组件渲染
9
10    }
11    // 处理表单项输入事件
12    handlerChange = (e) => {
13      const { name, value } = e.target;
14      console.log(name, value);
15      this.setState({
16        [name]: value
17      }, () => {
18        // 用户在页面中已经输入完成，接下来校验
19      })
20
21    }
22    getFieldDecorator = (fieldName, option) => {
23      // 设置字段选项配置
24      this.options[fieldName] = option;
25
26      return (InputComp) => {
27        return <div>
28          {/* 给当前的InputComp 定制name,
          value和onChange属性 */}
```

```

29         {
30             React.cloneElement(InputComp, {
31                 name: fieldName, //控件name
32                 value:
33                 this.state[fieldName] || '', //控件值
34                 onChange:
35                 this.handlerChange, //change事件处理
36             })
37         }
38     }
39     render() {
40         return (
41             <Comp {...this.props} name='张三'
42             getFieldDecorator={this.getFieldDecorator} />
43         )
44     }
45 }
46 }
47 @MFormCreate
48 class MForm extends Component {
49
50     render() {
51         const { getFieldDecorator } = this.props;
52
53         return (
54             <div>
55                 {
56                     getFieldDecorator('username', {
57                         rules: [
58                             {
59                                 required: true,

```

```

60         message: "用户名是必填
项"
61     }
62 ]
63 })(<input type='text' />)
64 }
65 {
66     getFieldDecorator('pwd', {
67         rules: [
68             {
69                 required: true,
70                 message: "密码是必填项"
71             }
72         ]
73     })(<input type='password' />)
74 }
75     <input type="submit" value='登录' />
76 </div>
77 );
78 }
79 }
80
81 export default MForm;

```

表单项输入完成后的校验

MFormCreate高阶组件中

```

1  const MFormCreate = function (Comp) {
2      return class extends Component {
3          constructor(props) {
4              super(props);
5              this.state = {}; //保存各字段的值
6              this.options = {}; //保存字段选项设置
7

```

```

8      }
9      // 处理表单项输入事件
10     handlerChange = (e) => {
11         const { name, value } = e.target;
12         // console.log(name, value);
13         this.setState({
14             [name]: value
15         }, () => {
16             // 用户在页面中已经输入完成，接下来表单项
17             // 校验
18             this.validateField(name)
19         })
20     }
21     // 表单项校验
22     validateField = (fieldName) => {
23         const { rules } = this.options[fieldName];
24         const ret = rules.some(rule => {
25             if (rule.required) {
26                 // 如果输入框值为空
27                 if (!this.state[fieldName]) {
28                     this.setState({
29                         [fieldName + 'Message']:
30                         rule.message
31                     })
32                     return true; //校验失败，返回
33                     true
34                 }
35             }
36         })
37         // console.log(ret);
38         if (!ret) {
39             this.setState({
40                 [fieldName + 'Message']: ''
41             })

```



```

40         }
41         return !ret; //校验成功 返回false
42     }
43     getFieldDecorator = (fieldName, option) => {
44         // 设置字段选项配置
45         this.options[fieldName] = option;
46
47         return (InputComp) => {
48             return <div>
49                 { /*....*/ }
50
51                 { /*验证显示*/ }
52                 {
53                     this.state[fieldName +
54 "Message"] && (
55                         <p style={{ color: 'red'
56 }}>{this.state[fieldName + 'Message']}
57                     )
58                 }
59             </div>
60         }
61     }
62     render() {
63         return (
64             <Comp {...this.props} name='张三'
65             getFieldDecorator={this.getFieldDecorator} />
66         )
67     }
68 }
69 }

```

点击提交按钮校验

```

1  const MFormCreate = function (Comp) {
2      return class extends Component {
3          constructor(props) {
4              super(props);
5              this.state = {}; //保存各字段的值
6              this.options = {}; //保存字段选项设置
7
8          }
9          // 处理表单项输入事件
10         handlerChange = (e) => {
11             const { name, value } = e.target;
12             // console.log(name, value);
13             this.setState({
14                 [name]: value
15             }, () => {
16                 // 用户在页面中已经输入完成，接下来表单项
17                 // 校验
18                 this.validateField(name)
19             })
20         }
21         // 表单项校验
22         validateField = (fieldName) => {
23             const { rules } = this.options[fieldName];
24             const ret = rules.some(rule => {
25                 if (rule.required) {
26                     // 如果输入框值为空
27                     if (!this.state[fieldName]) {
28                         this.setState({
29                             [fieldName + 'Message']:
30                             rule.message
31                         })
32                     }
33                     return true; //校验失败，返回
34                 }
35             })
36             return true;
37         }
38     }
39 }

```

```

32         }
33     }
34 })
35 // console.log(ret);
36 if (!ret) {
37     this.setState({
38         [fieldName + 'Message']: ''
39     })
40 }
41 return !ret; //校验成功 返回false
42 }
43 validate = (cb) => {
44     const rets =
Object.keys(this.options).map(fieldName =>
this.validateField(fieldName))
45     // 如果校验结果的数组中全部为true,则校验成功
46     const ret = rets.every(v=>v===true);
47     cb(ret);
48 }
49 getFieldDecorator = (fieldName, option) => {
50     // 设置字段选项配置
51     this.options[fieldName] = option;
52
53     return (InputComp) => {
54         return <div>
55             {/* 给当前的InputComp 定制name,
value和onChange属性 */}
56             {
57                 React.cloneElement(InputComp,
58 {
59                 name: fieldName, //控件
name
59                 value:
this.state[fieldName] || '', //控件值

```

```

60         onChange:
this.handlerChange, //change事件处理
61     })
62
63     }
64     {
65         this.state[fieldName +
"Message"] && (
66             <p style={{ color: 'red'
}}>{this.state[fieldName + 'Message']}

```

败

```

89         this.props.validate((isValid) => {
90             console.log(isValid);
91             if(isValid){
92                 alert('验证成功');
93             }else{
94                 alert('验证失败');
95             }
96         })
97     }
98     render() {
99         const { getFieldDecorator } = this.props;
100         return (
101             <div>
102                 { /* ..... */ }
103                 <input type="submit" value='登录'
onClick={this.handlerSubmit} />
104             </div>
105         );
106     }
107 }

```

最后封装FormItem和Input组件

```

1  import React, { Component } from 'react';
2  import { Icon } from 'antd'
3
4  // 包装用户表单，增加数据管理能力以及校验功能
5  const MFormCreate = function (Comp) {
6      return class extends Component {
7          constructor(props) {
8              super(props);
9              this.state = {}; //保存各字段的值
10             this.options = {}; //保存字段选项设置
11

```

```

12     }
13     // 处理表单项输入事件
14     handlerChange = (e) => {
15         const { name, value } = e.target;
16         // console.log(name, value);
17         this.setState({
18             [name]: value
19         }, () => {
20             // 用户在页面中已经输入完成，接下来表单项
21             // 校验
22             this.validateField(name)
23         })
24     }
25     // 表单项校验
26     validateField = (fieldName) => {
27         const { rules } = this.options[fieldName];
28         const ret = rules.some(rule => {
29             if (rule.required) {
30                 // 如果输入框值为空
31                 if (!this.state[fieldName]) {
32                     this.setState({
33                         [fieldName + 'Message']:
34                         rule.message
35                     })
36                     return true; //校验失败，返回
37                     true
38                 }
39             }
40         })
41         // console.log(ret);
42         if (!ret) {
43             this.setState({
44                 [fieldName + 'Message']: ''
45             })

```

```

44         }
45         return !ret; //校验成功 返回false
46     }
47     validate = (cb) => {
48         const rets =
Object.keys(this.options).map(fieldName =>
this.validateField(fieldName))
49         // 如果校验结果的数组中全部为true,则校验成功
50         const ret = rets.every(v => v === true);
51         cb(ret);
52     }
53     getFieldDecorator = (fieldName, option) => {
54         // 设置字段选项配置
55         this.options[fieldName] = option;
56
57         return (InputComp) => {
58             return <div>
59                 {/* 给当前的InputComp 定制name,
value和onChange属性 */}
60                 {
61                     React.cloneElement(InputComp,
62 {
63                         name: fieldName, //控件
name
64                         value:
this.state[fieldName] || '', //控件值
65                         onChange:
this.handlerChange, //change事件处理
66                         onFocus: this.handlerFocus
67                     })
68                 }
69             </div>
70         }
71     }

```

```
72      // 控件获取焦点事件
73      handlerFocus = (e) => {
74          const field = e.target.name;
75          console.log(field);
76
77          this.setState({
78              [field + 'Focus']: true
79          })
80      }
81      // 判断控件是否被点击过
82      isFieldTouched = field => !!this.state[field +
'Focus']
83
84      // 获取控件错误提示信息
85      getFieldError = field => this.state[field +
'Message'];
86      render() {
87          return (
88              <Comp
89                  {...this.props}
90                  getFieldDecorator=
{this.getFieldDecorator}
91                  validate={this.validate}
92                  isFieldTouched=
{this.isFieldTouched}
93                  getFieldError={this.getFieldError}
94              />
95          )
96      }
97
98  }
99  }
100  // 创建FormItem组件
101  class FormItem extends Component {
102      render() {
```



```
103         return (
104             <div className='formItem'>
105                 {this.props.children}
106                 {
107                     this.props.validateStatus ===
108                     'error' && (<p style={ { color: 'red' } }>{
109                         this.props.help}</p>)
110                 }
111             </div>
112         );
113     }
114 }
115 // 创建Input组件
116 class Input extends Component {
117     render() {
118         return (
119             <div>
120                 { /* 前缀图标 */ }
121                 {this.props.prefix}
122                 <input {...this.props} />
123             </div>
124         );
125     }
126 }
127
128
129 @MFormCreate
130 class MForm extends Component {
131     constructor(props) {
132         super(props);
133     }
134     handlerSubmit = () => {
```

```

135 // isValid为true表示校验成功，为false表示校验失
败
136 this.props.validate((isValid) => {
137     console.log(isValid);
138     if (isValid) {
139         alert('验证成功');
140     } else {
141         alert('验证失败');
142     }
143 })
144 }
145 render() {
146     const { getFieldDecorator, isFieldTouched,
147     getFieldError } = this.props;
148     const usernameError =
149     isFieldTouched('username') &&
150     getFieldError('username');
151     const pwdError = isFieldTouched('pwd') &&
152     getFieldError('pwd');
153
154     return (
155         <div>
156             <FormItem validateStatus=
157             {usernameError ? 'error' : ''} help={usernameError ||
158             ''}>
159                 {
160                     getFieldDecorator('username',
161                     {
162                         rules: [
163                             {
164                                 required: true,
165                                 message: "用户名是
166                                 必填项",
167                             }
168                         ]
169                     }
170                 )
171             }
172         )
173     )
174 }
175 }
176 
```

```

161         })(<Input type='text' prefix=
{<Icon type='user' /> />)
162     }
163     </FormItem>
164     <FormItem validateStatus={pwdError ?
'error' : ''} help={pwdError || ''}>
165         {
166             getFieldDecorator('pwd', {
167                 rules: [
168                     {
169                         required: true,
170                         message: "密码是必
填项"
171                     }
172                 ]
173             })(<Input type='password'
prefix={<Icon type='lock' /> />)
174         }
175     </FormItem>
176     <input type="submit" value='登录'
onClick={this.handlerSubmit} />
177     </div>
178 );
179 }
180 }
181
182 export default MForm;
183

```

总结

- react的组件是自上而下的扩展，将扩展的能力由上往下传递下去，Input组件在合适的时间就可以调用传递下来的值。
- react开发组件的原则是：把逻辑控制往上层提，低层级的组件尽

量做成傻瓜组件，不接触业务逻辑。