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

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

修改 `src/App.js`，引入 antd 的按钮组件。

```
import React, { Component } from 'react';
import Button from 'antd/es/button';
import './App.css';
import 'antd/dist/antd.css'

class App extends Component {
  render() {
    return (
      <div className="App">
        <Button type="primary">Button</Button>
      </div>
    );
  }
}

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样式

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。

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

修改package.json的启动文件

```
"scripts": {
   "start": "react-app-rewired start",
   "build": "react-app-rewired build",
   "test": "react-app-rewired test",
   "eject": "react-app-rewired eject"
 },
```

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

```
const { override, fixBabelImports } =
require('customize-cra');
module.exports = override(
    fixBabelImports('import', {
        libraryName: 'antd',
        libraryDirectory: 'es',
        style: 'css',
    }),
);
```

修改App.js

```
// import Button from 'antd/es/button';
// import 'antd/dist/antd.css'

import { Button } from 'antd';
```

运行

```
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

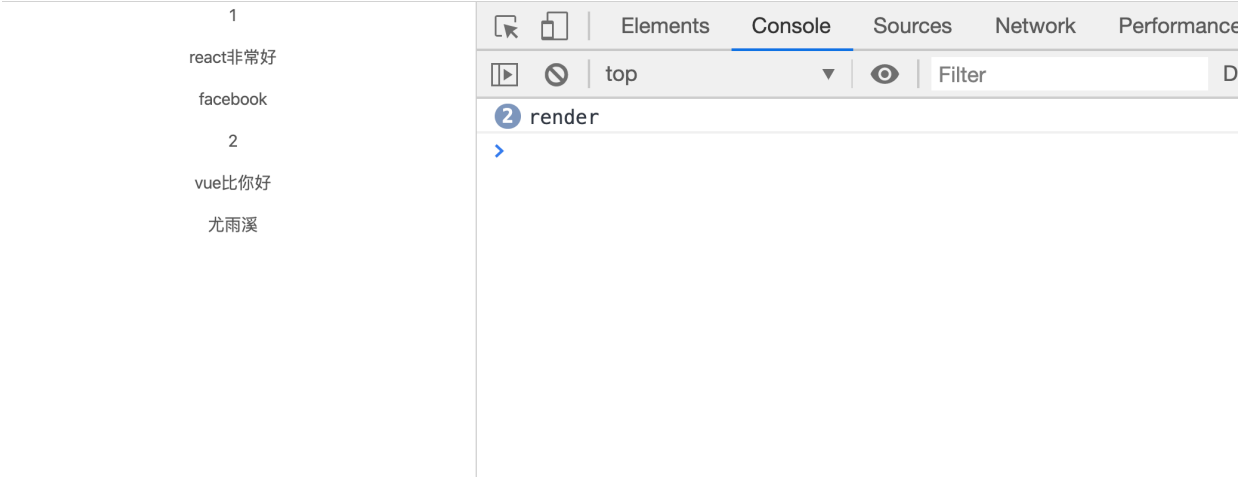
优势:

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`

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

function Comment({ comment }) {
    console.log('render');
    return (
        <div>
            <p>{comment.id}</p>
            <p>{comment.content}</p>
            <p>{comment.author}</p>
        </div>
    )
}

class CommentList extends Component {
    constructor(props) {
        super(props);
        this.state = {
            comments: []
        }
    }
    componentDidMount() {
        // 获取数据
        setTimeout(() => {
            this.setState({
                comments: [
                    {
                        id: 1,
                        content: 'react非常好',
                        author: 'facebook'
                    },
                    {
                        id: 2,
```

```
                        content: 'vue比你好',
                        author: '尤雨溪'
                    }
                ]
            })
        }, 1000);
    }

    render() {
        return (
            <div>
                {
                    this.state.comments.map((item, i) => (
                        <Comment comment={item} key={item.id} />
                    ))
                }

            </div>
        );
    }
}

export default CommentList;
```



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

```
// 每隔1秒钟,就开始更新一次
setInterval(() => {
    this.setState({
        comments: [
            {
                id: 1,
                content: 'react非常好',
                author: 'facebook'
            },
            {
                id: 2,
                content: 'vue比你好',
                author: '尤雨溪'
            }
        ]
    })
}, 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

```
shouldComponentUpdate(nextProps){
    if(nextProps.comment.id === this.props.comment.id){
        // return false表示不更新
        return false;
    }else{
        return true;
    }
}
```

解决方法2:使用React.PureComponent

`React.PureComponent`与`sholdComponentUpdate()`很相似。两者的区别在于`React.PureComponent`并未实现`shonldComponentUpdate()`，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`

```
render() {
    return (
        <div>
            {/* {
                    this.state.comments.map((item, i) => (
                        <Comment id={item.id} content={item.content} author={item.author} key={item.id} />
                    ))
             } */}

            {/*或者可以这样写*}
```

```
            {
                this.state.comments.map((item, i) => (
                    <Comment {...item} key={item.id} />
                ))
            }
        </div>
    );
}
```

修改`Comment.js`

```
class Comment extends PureComponent{
    constructor(props) {
        super(props);
    }
    render() {
        console.log('render');
        return (
            <div>
                <p>{this.props.id}</p>
                <p>{this.props.content}</p>
                <p>{this.props.author}</p>
            </div>
        )
    }
}
```

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。

```
const Comment = React.memo(({id,content,author})=>{
    return (
        <div>
            <p>{id}</p>
            <p>{content}</p>
            <p>{author}</p>
        </div>
    )
})
```

修改`CommentList.js`

```
class CommentList extends Component{
    render() {
        return (
            <div>
                {
                    this.state.comments.map((item, i) => (
                        <Comment id={item.id} content={item.content} author={item.author} key={item.id} />
                    ))
                }
            </div>
        );
    }
}
```

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/Compond.js

```
import React, { Component } from 'react';
import {
    Button
} from "antd";
function Dialog(props) {
    return (
        <div style={{ border: `3px solid ${props.color || 'blue'}` }}>
            {/*等价于：vue的匿名插槽*/}
            {props.children}
            {/*具名插槽*/}
            <div>
                {props.btn}
            </div>
        </div>
    )
}
// 将任意的组件作为子组件传递
function WelcomeDialog() {
    const confirmBtn = <Button type='primary' onClick={() => { alert('react真的好') }}>确定</Button>
          return (
              <Dialog color='green' btn={confirmBtn}>
                  <h3>welcome</h3>
                  <p>
                      欢迎光临
                  </p>
              </Dialog>
          )
}
class Compound extends Component {
    render() {
```

```
        return (
            <div>
                <WelcomeDialog />
            </div>
        );
    }
}

export default Compound;
```

高阶组件

组件设计的目的：保证组件功能的单一性

```
// 高阶组件
  本质是一个函数
  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
  好比是：我给你一个赛亚人，你给我一个超级赛亚人

// 高阶函数
  定义：接收的参数是函数或者返回值是函数
  常见的：数组遍历相关的方法 、定时器、Promise /高阶组件
  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

```
const HightOrderCom = (Comp) => {
    // 返回值为新组件
   const NewComponent =  function (props) {
        return <Comp name='react' content='高级组件的使用' {...props}></Comp>
    }
   return NewComponent;
}

```

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

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

function MyNormalCom(props) {
  return (
    <div>
      <p>课程名字:{props.name}</p>
      <h3>课程内容:{props.content}</h3>
    </div>
  )
}

// 高级组件
const HighOrderCom = (Comp) => {
  return (props)=>{
    return <Comp name='react' content='高阶组件的使用' {...props}></Comp>
  }
}
export default HighOrderCom(MyNormalCom);
```

重写高阶组件内部的生命周期

```
const HightOrderCom = (Comp) => {
    // 返回值为新组件
    return class extends Component {
        constructor(props) {
            super(props)
        }
        componentDidMount(){
            console.log('发起ajax请求');

        }
        render() {
            return (
                <Comp name='react' content='高级组件的使用' {...this.props}></Comp>
            );
        }
    }
}
```

高阶组件链式调用

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

调用:

```
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

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

配置修改:

```
const {
    override,
    fixBabelImports, //按需加载配置函数
    addBabelPlugins //babel插件配置函数
} = require('customize-cra');
module.exports = override(
    fixBabelImports('import', {
        libraryName: 'antd',
        libraryDirectory: 'es',
        style: 'css',
    }),
    addBabelPlugins( // 支持装饰器
        [
            '@babel/plugin-proposal-decorators',
            {
                legacy: true
            }
```

```
        ]
    )
);
```

```
const HightOrderCom = (Comp) => {
    // 返回值为新组件
    return class extends Component {
        constructor(props) {
            super(props)
        }
        componentDidMount() {
            console.log('发起ajax请求');

        }
        render() {
            return (
                <Comp name='react' content='高级组件的使用' {...this.props}></Comp>
            );
        }
    }
}
// 打印日志的高阶组件
const WithLog = (Comp) => {
    console.log(Comp.name + '渲染了');
    const MyComponent = function (props) {
        return <Comp {...props}></Comp>
    }
    return MyComponent;
}
@withLog
@HighOrderCom
@withLog
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

```jsx
import React, { Component } from 'react'
// 权限控制
// 不谈场景的技术就是在耍流氓
export const withAdminAuth = (role) => (WrappedComp) =>
{
  return class extends Component {
    constructor(props) {
      super(props);
      this.state = {
        isAdmin: false
      }
    }

    componentDidMount() {
      setTimeout(() => {
        this.setState({
          isAdmin: role === 'Admin'
        })
      }, 1000);
    }

    render() {
      if (this.state.isAdmin) {
        return (
          <WrappedComp {...this.props}></WrappedComp>
        );
      } else {
        return (
          <div>您没有权限查看该页面，请联系管理员</div>
        )
      }
    }
  }
}
```

```
34
```

然后是两个页面：

/components/PageA.js

```
import React, { Component } from 'react'
import { withAdminAuth } from "./HocApp";
class PageA extends Component {
  componentDidMount() {
    setTimeout(() => {
      this.setState({
        isAdmin: true
      })
    }, 1000);
  }
  render() {
    return (
      <div>
        <h2>我是页面A</h2>
      </div>
    )
  }
}
export default withAdminAuth('Admin')(PageA)
```

/components/PageB.js

```
import React, { Component } from 'react'
import { withAdminAuth } from "./HocApp";
class PageB extends Component {
  render() {
    return (
      <div>
        <h2>我是页面B</h2>
      </div>
    )
  }
}
export default withAdminAuth()(PageB);

```

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：

```jsx
import React, { Component } from 'react';
// Context可以让我们传遍每一个组件，就能将值深入组件树中
// 创建一个ThemeContext，默认值为light
const ThemeContext  = React.createContext('lighter');

class ThemeButton extends Component {
    // 指定 contextType 读取当前的 theme context。
    // React 会往上找到最近的 theme Provider，然后使用它的值。
    // 在这个例子中，当前的 theme 值为 “dark”。
    // 第一种渲染的方式(下面通过this.context渲染数据):static contextType = ThemeContext;
    render() {
        return (
            // <button type={this.context}></button>
            <ThemeContext.Consumer>
                {/* 2.基于函数去渲染 value等价于this.context */}
                {
                    value => <button theme={value.theme}>{value.name}</button>
                }
            </ThemeContext.Consumer>
        );
    }
}
function Toolbar(props) {
    // Toolbar 组件接受一个额外的“theme”属性，然后传递给ThemedButton 组件。
    // 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，
    // 因为必须将这个值层层传递所有组件。
    return (
        <div>
```

```
            <ThemeButton></ThemeButton>
        </div>
    )

}
//传递的数据
const store = {
    theme:'dark',
    name:'按钮'
}
class ContextSimple extends Component {
    // 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
    render() {
        return (
            <div>
                <ThemeContext.Provider value={store}>
                    <Toolbar></Toolbar>
                </ThemeContext.Provider>
            </div>
        );
    }
}

export default ContextSimple;
```

高阶组件装饰器写法

```
import React, { Component } from 'react';
const ThemeContext = React.createContext('lighter');
const withConsumer = Comp => {
    return class extends Component {
        constructor(props) {
            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
```

```
const store = {
    theme: 'dark',
    name: '按钮'
}

const withProvider = Comp => {
    return function (props) {
        return (
            <ThemeContext.Provider value={store}>
                <Comp {...props} />
            </ThemeContext.Provider>
        )
    }
}
@withProvider
class ContextSimple extends Component {
    render() {
        return (
            <div>
                <Toolbar></Toolbar>
            </div>
        );
    }
}

export default ContextSimple;
```

封装antd的form表单

打开antd官网

```
import React from 'react'
import { Form, Icon, Input, Button } from 'antd';

```

```
function hasErrors(fieldsError) {
    return Object.keys(fieldsError).some(field => fieldsError[field]);
}

class HorizontalLoginForm extends React.Component {
    componentDidMount() {
        // To disabled submit button at the beginning.
        this.props.form.validateFields();
    }
    handleSubmit = e => {
        e.preventDefault();
        this.props.form.validateFields((err, values) => {
            if (!err) {
                console.log('Received values of form: ', values);
            }
        });
    };

render() {
    const { getFieldDecorator, getFieldsError, getFieldError, isFieldTouched } = this.props.form;

    // Only show error after a field is touched.
    const usernameError = isFieldTouched('username') && getFieldError('username');
    const passwordError = isFieldTouched('password') && getFieldError('password');
    return (
        <Form layout="inline" onSubmit={this.handleSubmit}>
            <Form.Item validateStatus={usernameError ? 'error' : ''} help={usernameError || ''}>
```

```
                {getFieldDecorator('username', {
                    rules: [{ required: true, message: 'Please input your username!' }],
                })(
                    <Input
                        prefix={<Icon type="user" style={{ color: 'rgba(0,0,0,.25)' }} />}
                        placeholder="Username"
                        />,
                )}
            </Form.Item>
            <Form.Item validateStatus={passwordError ? 'error' : ''} help={passwordError || ''}>
                {getFieldDecorator('password', {
                    rules: [{ required: true, message: 'Please input your Password!' }],
                })(
                    <Input
                        prefix={<Icon type="lock" style={{ color: 'rgba(0,0,0,.25)' }} />}
                        type="password"
                        placeholder="Password"
                        />,
                )}
            </Form.Item>
            <Form.Item>
                <Button type="primary" htmlType="submit" disabled={hasErrors(getFieldsError())}>
                    Log in
                </Button>
            </Form.Item>
        </Form>
    );
}
```

```
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

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

class MForm extends Component {
    render() {
        return (
            <div>
                <input type="text" />
                <input type="password"/>
                <input type="submit" value='登录'/>
            </div>
        );
    }
}

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

```
import React, { Component } from 'react';
// 包装用户表单，增加数据管理能力以及校验功能
const MFormCreate = function (Comp) {
    return class extends Component {
        constructor(props) {
            super(props);
            this.state = {};//保存各字段的值
            this.options = {}; //保存字段选项设置 不希望它的变化让组件渲染

        }
        // 处理表单项输入事件
        handlerChange = (e) => {
            const { name, value } = e.target;
            console.log(name, value);
            this.setState({
                [name]:value
            },()=>{
                // 用户在页面中已经输入完成，接下来校验
            })

        }
        getFieldDecorator = (fieldName, option) => {
            // 设置字段选项配置
            this.options[fieldName] = option;

            return (InputComp) => {
                return <div>
                    {/* 给当前的InputComp 定制name, value和onChange属性  */}

```

```
                    {
                        React.cloneElement(InputComp, {
                            name: fieldName, //控件name
                            value:
this.state[fieldName] || '', //控件值
                            onChange:
this.handlerChange,//change事件处理
                        })
                    }
                </div>
            }
        }
        render() {
            return (
                <Comp {...this.props} name='张三'
getFieldDecorator={this.getFieldDecorator} />
            )
        }

}
}
@MFormCreate
class MForm extends Component {

    render() {
        const { getFieldDecorator } = this.props;

        return (
            <div>
                {
                    getFieldDecorator('username', {
                        rules: [
                            {
                                required: true,
```

```
                                message: "用户名是必填项"
                            }
                        ]
                    })(<input type='text' />)
                }
                {
                    getFieldDecorator('pwd', {
                        rules: [
                            {
                                required: true,
                                message: "密码是必填项"
                            }
                        ]
                    })(<input type='password' />)
                }
                <input type="submit" value='登录' />
            </div>
        );
    }
}

export default MForm;
```

表单项输入完成后的校验

MFormCreate高阶组件中

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

```

```
        }
        // 处理表单项输入事件
        handlerChange = (e) => {
            const { name, value } = e.target;
            // console.log(name, value);
            this.setState({
                [name]: value
            }, () => {
                // 用户在页面中已经输入完成，接下来表单项校验
                this.validateField(name)
            })

        }
        // 表单项校验
        validateField = (fieldName) => {
            const { rules } = this.options[fieldName];
            const ret = rules.some(rule => {
                if (rule.required) {
                    // 如果输入框值为空
                    if (!this.state[fieldName]) {
                        this.setState({
                            [fieldName + 'Message']: rule.message
                        })
                        return true; //校验失败，返回true
                    }
                }
            })
            // console.log(ret);
            if (!ret) {
                this.setState({
                    [fieldName + 'Message']: ''
                })
```

```
            }
            return !ret; //校验成功 返回false
        }
        getFieldDecorator = (fieldName, option) => {
            // 设置字段选项配置
            this.options[fieldName] = option;

            return (InputComp) => {
                return <div>
                    {/*....*/}

                    {/*验证显示*/}
                    {
                        this.state[fieldName + "Message"] && (
                            <p style={{ color: 'red' }}>{this.state[fieldName + 'Message']}</p>
                        )
                    }
                </div>
            }
        }
        render() {
            return (
                <Comp {...this.props} name='张三' getFieldDecorator={this.getFieldDecorator} />
            )
        }

}
}
```

点击提交按钮校验

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

        }
        // 处理表单项输入事件
        handlerChange = (e) => {
            const { name, value } = e.target;
            // console.log(name, value);
            this.setState({
                [name]: value
            }, () => {
                // 用户在页面中已经输入完成，接下来表单项校验
                this.validateField(name)
            })

        }
        // 表单项校验
        validateField = (fieldName) => {
            const { rules } = this.options[fieldName];
            const ret = rules.some(rule => {
                if (rule.required) {
                    // 如果输入框值为空
                    if (!this.state[fieldName]) {
                        this.setState({
                            [fieldName + 'Message']: rule.message
                        })
                        return true; //校验失败，返回 true
```

```
                    }
                }
            })
            // console.log(ret);
            if (!ret) {
                this.setState({
                    [fieldName + 'Message']: ''
                })
            }
            return !ret; //校验成功 返回false
        }
        validate = (cb) => {
            const rets =
Object.keys(this.options).map(fieldName =>
this.validateField(fieldName))
            // 如果校验结果的数组中全部为true,则校验成功
            const ret = rets.every(v=>v===true);
            cb(ret);
        }
        getFieldDecorator = (fieldName, option) => {
            // 设置字段选项配置
            this.options[fieldName] = option;

            return (InputComp) => {
                return <div>
                    {/* 给当前的InputComp 定制name,
value和onChange属性  */}
                    {
                        React.cloneElement(InputComp,
{
                            name: fieldName, //控件
name
                            value:
this.state[fieldName] || '', //控件值
```

```
                                onChange:
this.handlerChange,//change事件处理
                            })

                    }
                    {
                        this.state[fieldName +
"Message"] && (
                            <p style={{ color: 'red'
}}>{this.state[fieldName + 'Message']}</p>
                        )
                    }
                </div>
            }
        }
        render() {
            return (
                <Comp {...this.props} name='张三'
getFieldDecorator={this.getFieldDecorator} validate=
{this.validate} />
            )
        }

    }
}


@MFormCreate
class MForm extends Component {
    constructor(props) {
        super(props);
    }
    handlerSubmit = () => {
        // isValid为true表示校验成功，为flase表示校验失
败
```

```
        this.props.validate((isValid) => {
            console.log(isValid);
            if(isValid){
                alert('验证成功');
            }else{
                alert('验证失败');
            }
        })
    }
    render() {
        const { getFieldDecorator } = this.props;
        return (
            <div>
                {/*....*/}
                <input type="submit" value='登录' onClick={this.handlerSubmit} />
            </div>
        );
    }
}
```

最后封装FormItem和Input组件

```
import React, { Component } from 'react';
import { Icon } from 'antd'

// 包装用户表单，增加数据管理能力以及校验功能
const MFormCreate = function (Comp) {
    return class extends Component {
        constructor(props) {
            super(props);
            this.state = {};//保存各字段的值
            this.options = {}; //保存字段选项设置

```

```
        }
        // 处理表单项输入事件
        handlerChange = (e) => {
            const { name, value } = e.target;
            // console.log(name, value);
            this.setState({
                [name]: value
            }, () => {
                // 用户在页面中已经输入完成，接下来表单项校验
                this.validateField(name)
            })

        }
        // 表单项校验
        validateField = (fieldName) => {
            const { rules } = this.options[fieldName];
            const ret = rules.some(rule => {
                if (rule.required) {
                    // 如果输入框值为空
                    if (!this.state[fieldName]) {
                        this.setState({
                            [fieldName + 'Message']: rule.message
                        })
                        return true; //校验失败，返回true
                    }
                }
            })
            // console.log(ret);
            if (!ret) {
                this.setState({
                    [fieldName + 'Message']: ''
                })
```

```
            }
            return !ret; //校验成功 返回false
        }
        validate = (cb) => {
            const rets =
Object.keys(this.options).map(fieldName =>
this.validateField(fieldName))
            // 如果校验结果的数组中全部为true,则校验成功
            const ret = rets.every(v => v === true);
            cb(ret);
        }
        getFieldDecorator = (fieldName, option) => {
            // 设置字段选项配置
            this.options[fieldName] = option;

            return (InputComp) => {
                return <div>
                    {/* 给当前的InputComp 定制name,
value和onChange属性  */}
                    {
                        React.cloneElement(InputComp,
{
                            name: fieldName, //控件
name
                            value:
this.state[fieldName] || '', //控件值
                            onChange:
this.handlerChange,//change事件处理
                            onFocus: this.handlerFocus
                        })

                    }
                </div>
            }
        }
```

```
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() {
```

```
        return (
            <div className='formItem'>
                {this.props.children}
                {
                   this.props.validateStatus === 'error' && (<p style={ { color: 'red' } }>{this.props.help}</p>)
                }
            </div>
        );
    }
}


// 创建Input组件
class Input extends Component {
    render() {
        return (
            <div>
                {/* 前缀图标 */}
                {this.props.prefix}
                <input {...this.props} />
            </div>
        );
    }
}


@MFormCreate
class MForm extends Component {
    constructor(props) {
        super(props);
    }
    handlerSubmit = () => {
```

```
        // isValid为true表示校验成功，为flase表示校验失败
        this.props.validate((isValid) => {
            console.log(isValid);
            if (isValid) {
                alert('验证成功');
            } else {
                alert('验证失败');
            }
        })
    }
    render() {
        const { getFieldDecorator, isFieldTouched, getFieldError } = this.props;
        const usernameError = isFieldTouched('username') && getFieldError('username');
        const pwdError = isFieldTouched('pwd') && getFieldError('pwd');

        return (
            <div>
                <FormItem validateStatus={usernameError ? 'error' : ''} help={usernameError || ''}>
                    {
                        getFieldDecorator('username', {
                            rules: [
                                {
                                    required: true,
                                    message: "用户名是必填项",
                                }
                            ]
```

```
                        })(<Input type='text' prefix={<Icon type='user' />} />)
                    }
                </FormItem>
                <FormItem validateStatus={pwdError ? 'error' : ''} help={pwdError || ''}>
                    {
                        getFieldDecorator('pwd', {
                            rules: [
                                {
                                    required: true,
                                    message: "密码是必填项"
                                }
                            ]
                        })(<Input type='password' prefix={<Icon type='lock' />} />)
                    }
                </FormItem>
                <input type="submit" value='登录' onClick={this.handlerSubmit} />
            </div>
        );
    }
}

export default MForm;

```

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。