React组件化

课堂目标

掌握组件化开发中多种实现技术

1. 掌握高阶组件HOC
2. 了解组件化概念，能设计并实现自己需要的组件
3. 掌握弹窗类实现，掌握传送门使用

资源

1. create-react-app
2. HOC
3. ant design

回顾预习录播

React基础预习课程

1. React入门
2. JSX语法
3. 正确使用setState
4. 生命周期
5. 组件
6. 组件复合
7. redux
8. react-redux
9. react-router
10. PureComponent

进阶篇

1. 认识Hook
2. 自定义Hook与Hook使用规则
3. Hook API之useMemo与useCallback

知识点

快速开始

(https://www.html.cn/create-react-app/docs/getting-started/)

```
npx create-react-app lesson1

cd lesson1

npm start
```

组件化优点

1. 增强代码重用性，提高开发效率
2. 简化调试步骤，提升整个项目的可维护性
3. 便于协同开发
4. 注意点：降低耦合性

高阶组件-HOC

为了提高组件复用率，可测试性，就要保证组件功能单一性；但是若要满足复杂需求就要扩展功能单一的组件，在React里就有了HOC（Higher-Order Components）的概念。

定义：**高阶组件是参数为组件，返回值为新组件的函数。**

基本使用

```
// HocPage.js
import React, {Component} from "react";

// hoc：是一个函数，接收一个组件，返回另外一个组件
//这里大写开头的Cmp是指function或者class组件
const foo = Cmp => props => {
  return (
    <div className="border">
      <Cmp {...props} />
    </div>
  );
};

// const foo = Cmp => {
//   return props => {
//     return (
//       <div className="border">
//         <Cmp {...props} />
//       </div>
//     );
//   };
// };

function Child(props) {
  return <div> Child {props.name}</div>;
}

const Foo = foo(Child);
export default class HocPage extends Component {
  render() {
    return (
```

```
      <div>
        <h3>HocPage</h3>
        <Foo name="msg" />
      </div>
    );
  }
}
```

链式调用

```
// HocPage.js
import React, {Component} from "react";

// hoc: 是一个函数，接收一个组件，返回另外一个组件
//这里大写开头的Cmp是指function或者class组件
const foo = Cmp => props => {
  return (
    <div className="border">
      <Cmp {...props} />
    </div>
  );
};
const foo2 = Cmp => props => {
  return (
    <div className="greenBorder">
      <Cmp {...props} />
    </div>
  );
};

// const foo = Cmp => {
```

```
//   return props => {
//     return (
//       <div className="border">
//         <Cmp {...props} />
//       </div>
//     );
//   };
// };

function Child(props) {
  return <div> Child {props.name}</div>;
}

const Foo = foo2(foo(foo(Child)));
export default class HocPage extends Component {
  render() {
    return (
      <div>
        <h3>HocPage</h3>
        <Foo name="msg" />
      </div>
    );
  }
}
```

装饰器写法

高阶组件本身是对装饰器模式的应用，自然可以利用ES7中出现的装饰器语法来更优雅的书写代码。

`npm install -D @babel/plugin-proposal-decorators`

更新config-overrides.js

```
//配置完成后记得重启下
const { addDecoratorsLegacy } = require("customize-cra");

module.exports = override(
  ...,
  addDecoratorsLegacy()//配置装饰器
);
```

如果vscode对装饰器有warning，vscode设置里加上

`javascript.implicitProjectConfig.experimentalDecorators": true`

```
//HocPage.js
//...
// !装饰器只能用在class上
// 执行顺序从下往上
@foo2
@foo
@foo
class Child extends Component {
  render() {
    return <div> Child {this.props.name}</div>;
  }
}

// const Foo = foo2(foo(foo(Child)));
export default class HocPage extends Component {
  render() {
```

```
    return (
      <div>
        <h3>HocPage</h3>
        {/* <Foo name="msg" /> */}
        <Child />
      </div>
    );
  }
}
```

组件是将 props 转换为 UI，而高阶组件是将组件转换为另一个组件。

HOC 在 React 的第三方库中很常见，例如 React-Redux 的 connect，我们下节课就会学到。

使用HOC的注意事项

高阶组件（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。

- **不要在 render 方法中使用 HOC**

  React 的 diff 算法（称为协调）使用组件标识来确定它是应该更新现有子树还是将其丢弃并挂载新子树。 如果从 `render` 返回的组件与前一个渲染中的组件相同（`===`），则 React 通过将子树与新子树进行区分来递归更新子树。 如果它们不相等，则完全卸载前一个子树。

```
render() {
  // 每次调用 render 函数都会创建一个新的
EnhancedComponent
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // 这将导致子树每次渲染都会进行卸载，和重新挂载的操作!
  return <EnhancedComponent />;
}
```

这不仅仅是性能问题 - 重新挂载组件会导致该组件及其所有子组件的状态丢失。

表单组件设计与实现

antd表单使用

用state实现用户名密码登录

//FormPage.js

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

export default class FormPage extends Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "",
      password: ""
    };
  }
  submit = () => {
```

```
    console.log("submit", this.state);
  };
  render() {
    const {name, password} = this.state;
    return (
      <div>
        <h3>FormPage</h3>
        <Form>
          <Form.Item label="姓名">
            <Input
              placeholder="please input ur name"
              prefix={<Icon type="user" />}
              value={name}
              onChange={e => {
                this.setState({name:
e.target.value});
              }}
            />
          </Form.Item>
          <Form.Item label="密码">
            <Input
              type="password"
              placeholder="please input ur password"
              prefix={<Icon type="lock" />}
              value={password}
              onChange={e => {
                this.setState({password:
e.target.value});
              }}
            />
          </Form.Item>
          <Button type="primary" onClick=
{this.submit}>
```

```
            提交
          </Button>
        </Form>
      </div>
    );
  }
}
```

用Form.create()的方式实现：

- getFieldDecorator： 用于和表单进行双向绑定
- getFieldsValue：获取一组输入控件的值，如不传入参数，则获取全部组件的值
- getFieldValue： 获取一个输入控件的值
- validateFields：校验并获取一组输入域的值与 Error，若 fieldNames 参数为空，则校验全部组件

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

//校验规则
const nameRules = {required: true, message: "please
input ur name"};
const passwordRules = {required: true, message:
"please input ur password"};

@Form.create()
class FormPage2 extends Component {
  submit = () => {
    const {getFieldsValue, getFieldValue,
validateFields} = this.props.form;
```

```
    validateFields((err, values) => {
      if (err) {
        console.log("err", err); //sy-log
      } else {
        console.log("success", values); //sy-log
      }
    });
    // console.log("submit", getFieldsValue(),
getFieldValue("name"));
  };
  render() {
    console.log("props", this.props.form);

    const {getFieldDecorator} = this.props.form;

    return (
      <div>
        <h3>FormPage2</h3>
        <Form>
          <Form.Item label="姓名">
            {getFieldDecorator("name", {rules:
[nameRules]})(
              <Input
                placeholder="please input ur name"
                prefix={<Icon type="user" />}
              />
            )}
          </Form.Item>
          <Form.Item label="密码">
            {getFieldDecorator("password", {rules:
[passwordRules]})(
              <Input
                type="password"
```

```
                placeholder="please input ur
password"
                prefix={<Icon type="lock" />}
              />
            )}
          </Form.Item>
          <Button type="primary" onClick=
{this.submit}>
            提交
          </Button>
        </Form>
      </div>
    );
  }
}

export default FormPage2;
```

表单组件设计思路

- 表单组件要求实现**数据收集**、**校验**、**提交**等特性，可通过高阶组件扩展
- 高阶组件给表单组件传递一个input组件**包装函数**接管其输入事件并统一管理表单数据
- 高阶组件给表单组件传递一个**校验函数**使其具备数据校验功能

表单组件实现

- 表单基本结构，创建MyFormPage.js

```
import React, {Component} from "react";
import kFormCreate from
"../conponents/kFormCreate";

//校验规则
const nameRules = {required: true, message:
"please input ur name"};
const passwordRules = {required: true, message:
"please input ur password"};

export default kFormCreate(
  class MyFormPage extends Component {
    submit = () => {
      const {getFieldsValue, getFieldValue,
validateFields} = this.props;
      validateFields((err, values) => {
        if (err) {
          console.log("err", err); //sy-log
        } else {
          console.log("success", values); //sy-
log
        }
      });
      // console.log("submit", getFieldsValue(),
getFieldValue("name")); //sy-log
    };
    render() {
      console.log("props", this.props); //sy-log
      const {getFieldDecorator} = this.props;
      return (
        <div>
          <h3>MyFormPage</h3>
```

```
          {getFieldDecorator("name", {rules:
[nameRules]})(
            <input type="text"
placeholder="please input ur name" />
          )}
          {getFieldDecorator("password", {rules:
[passwordRules]})(
            <input type="password"
placeholder="please input ur password" />
          )}
          <button onClick={this.submit}>提交
</button>
        </div>
      );
    }
  }
);
```

- 高阶组件kFormCreate：扩展现有表单，./components/kFormCreate.js

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

export default function kFormCreate(Cmp) {
  return class extends Component {
    constructor(props) {
      super(props);
      this.state = {};
      this.options = {};
    }
    handleChange = e => {
      let {name, value} = e.target;
```

```
      this.setState({[name]: value});
    };
    getFieldDecorator = (field, option) => {
      this.options[field] = option;
      return InputCmp =>
        React.cloneElement(InputCmp, {
          name: field,
          value: this.state[field] || "",
          onChange: this.handleChange //控件change
事件处理
        });
    };
    getFieldsValue = () => {
      return {...this.state};
    };
    getFieldValue = field => {
      return this.state[field];
    };
    validateFields = callback => {
      let errors = {};
      const state = {...this.state};
      for (let field in this.options) {
        if (state[field] === undefined) {
          errors[field] = "error";
        }
        console.log("item", field); //sy-log
      }
      if (JSON.stringify(errors) === "{}") {
        // 没有错误信息
        callback(undefined, state);
      } else {
        // 有错误信息，返回
        callback(errors, state);
```

```
      }
    };
    render() {
      return (
        <div className="border">
          <Cmp
            {...this.props}
            getFieldDecorator=
{this.getFieldDecorator}
            getFieldsValue={this.getFieldsValue}
            getFieldValue={this.getFieldValue}
            validateFields={this.validateFields}
          />
        </div>
      );
    }
  };
}
```

弹窗类组件设计与实现

设计思路

弹窗类组件的要求弹窗内容在A处声明，却在B处展示。react中相当于弹窗内容看起来被render到一个组件里面去，实际改变的是网页上另一处的DOM结构，这个显然不符合正常逻辑。但是通过使用框架提供的特定API创建组件实例并指定挂载目标仍可完成任务。

```
// 常见用法如下：Dialog在当前组件声明，但是却在body中另一个
div中显示
import React, {Component} from "react";
```

```
import Dialog from "../conponents/Dialog";

export default class DialogPage extends Component {
  constructor(props) {
    super(props);
    this.state = {
      showDialog: false
    };
  }
  render() {
    const {showDialog} = this.state;
    return (
      <div>
        <h3>DialogPage</h3>
        <button
          onClick={() =>
            this.setState({
              showDialog: !showDialog
            })
          }>
          toggle
        </button>
        {showDialog && <Dialog />}
      </div>
    );
  }
}
```

具体实现: Portal

传送门，react v16之后出现的portal可以实现内容传送功能。

范例：Dialog组件

```
// Diallog.js
import React, { Component } from "react";
import { createPortal } from "react-dom";

export default class Dialog extends Component {
  constructor(props) {
    super(props);
    const doc = window.document;
    this.node = doc.createElement("div");
    doc.body.appendChild(this.node);
  }
  componentWillUnmount() {
    window.document.body.removeChild(this.node);
  }
  render() {
    const { hideDialog } = this.props;
    return createPortal(
      <div className="dialog">
        {this.props.children}
        {typeof hideDialog === "function" && (
          <button onClick={hideDialog}>关掉弹窗
</button>
        )}
      </div>,
      this.node,
    );
  }
}
```

```
.dialog {
  position: absolute;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  line-height: 30px;
  width: 400px;
  height: 300px;
  transform: translate(50%, 50%);
  border: solid 1px gray;
  text-align: center;
}
```

> 总结一下：
>
> 1. Dialog什么都不给自己画，render返回一个null就够了；
> 2. 它做得事情是通过调用createPortal把要画的东西画在DOM树上另一个角落。

回顾

作业

1. 丰富表单组件实现，如加上错误提示
2. 看redux和react-redux的预习视频

下节课内容

lesson2：掌握Context、实现redux与react-redux。