React全家桶01

课堂目标□

2. 能设计并实现自己的组件
3. 掌握redux及实现

知识要点

2. 设计并实现表单控件
3. 实现弹窗类组件
4. 掌握redux及实现

资源

1. ant design
2. redux
3. redux github

知识点

使用第三方组件

不必npm run eject，直接安装：`npm install antd --save`

范例：试用 ant-design组件库

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

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

配置按需加载

安装react-app-rewired取代react-scripts，可以扩展webpack的配置，类似vue.config.js。

由于新的 react-app-rewired@2.x 版本的关系，你还需要安装 customize-cra。

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

`npm install react-app-rewired customize-cra babel-plugin-import -D`

```
//根目录创建config-overrides.js
const { override, fixBabelImports } = require("customize-cra");

module.exports = override(
  fixBabelImports("import", {//antd按需加载
    libraryName: "antd",
    libraryDirectory: "es",
    style: "css"
  })
);

//修改package.json
  "scripts": {
    "start": "react-app-rewired start",
    "build": "react-app-rewired build",
```

```
    "test": "react-app-rewired test",
    "eject": "react-app-rewired eject"
  },
```

支持装饰器配置

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

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

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

```
//按需加载之后页面如下：
import React, { Component } from "react";
import { Button } from "antd";
import "antd/dist/antd.css"

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

表单组件设计与实现

antd表单试用

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

const FormItem = Form.Item;

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

@Form.create()
class FormPageDecorators extends Component {
  handleSubmit = () => {
    /*  const { getFieldsValue, getFieldValue } = this.props.form;
    console.log("submit", getFieldsValue()); */

    const { validateFields } = this.props.form;
    validateFields((err, values) => {
```

```
      if (err) {
        console.log("err", err);
      } else {
        console.log("submit", values);
      }
    });
  };
  render() {
    const { getFieldDecorator } = this.props.form;
    // console.log(this.props.form);
    return (
      <div>
        <h1>FormPageDecorators</h1>
        <Form>
          <FormItem label="姓名">
            {getFieldDecorator("name", { rules: [nameRules] })(
              <Input prefix={<Icon type="user" />} />,
            )}
          </FormItem>
          <FormItem label="密码">
            {getFieldDecorator("password", { rules: [passwordRules] })(
              <Input type="password" prefix={<Icon type="lock" />} />,
            )}
          </FormItem>
          <FormItem>
            <Button type="primary" onClick={this.handleSubmit}>
              提交
            </Button>
          </FormItem>
        </Form>
      </div>
    );
  }
}
export default FormPageDecorators;
// export default Form.create()(FormPageDecorators);
```

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 "../../components/kFormCreate";

const nameRules = { required: true, message: "please input your name!" };
const passwordRules = {
```

```
  required: true,
  message: "please input your password!",
};
class MyFormPage extends Component {
  handleSubmit = () => {
    const { getFieldValue } = this.props;
    const res = {
      name: getFieldValue("name"),
      password: getFieldValue("password"),
    };
    console.log("hah", res);
  };
  handleSubmit2 = () => {
    // 加入校验
    const { validateFields } = this.props;
    validateFields((err, values) => {
      if (err) {
        console.log("validateFields", err);
      } else {
        console.log("submit", values);
      }
    });
  };
  render() {
    const { getFieldDecorator } = this.props;
    return (
      <div>
        <h1>MyFormPage</h1>
        <div>
          {getFieldDecorator("name", { rules: [nameRules] })(
            <input type="text" />,
          )}
          {getFieldDecorator("password", [passwordRules])(
            <input type="password" />,
          )}
        </div>
        <button onClick={this.handleSubmit2}>submit</button>
      </div>
    );
  }
}

export default kFormCreate(MyFormPage);
```

- 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.options = {}; //各字段选项
      this.state = {}; //各字段值
    }
```

```
    handleChange = e => {
      let { name, value } = e.target;
      this.setState({ [name]: value });
    };
    getFieldValue = field => {
      return this.state[field];
    };
    validateFields = callback => {
      const res = { ...this.state };
      const err = [];
      for (let i in this.options) {
        if (res[i] === undefined) {
          err.push({ [i]: "error" });
        }
      }
      if (err.length > 0) {
        callback(err, res);
      } else {
        callback(undefined, res);
      }
    };
    getFieldDecorator = (field, option) => {
      this.options[field] = option;
      return InputCmp => (
        <div>
          {
          React.cloneElement(InputCmp, {
            name: field,
            value: this.state[field] || "", //控件值
            onChange: this.handleChange, //控件change事件处理
          })}
        </div>
      );
    };
    render() {
      return (
        <div className="border">
          <Cmp
            {...this.props}
            getFieldDecorator={this.getFieldDecorator}
            getFieldValue={this.getFieldValue}
            validateFields={this.validateFields}
          />
        </div>
      );
    }
  };
}
```

```
//用useState实现kFormCreate
import React, { useState } from "react";
const kFormCreate = Cmp => props => {
  const [state, setState] = useState({});
  const options = {};
  const handleChange = event => {
    setState({ ...state, [event.target.name]: event.target.value });
```

```
  };
  const getFieldDecorator = (field, option) => {
    options[field] = option;
    return InpurtCmp => {
      return (
        <>
          {React.cloneElement(InpurtCmp, {
            name: field,
            value: state[field] || "",
            onChange: handleChange,
          })}
        </>
      );
    };
  };
  const getFieldsValue = () => {
    return { ...state };
  };
  const getFieldValue = field => {
    return state[field];
  };
  const validateFields = callback => {
    const res = { ...state };
    const err = [];
    for (let item in options) {
      if (res[item] === undefined) {
        err.push({ [item]: "error" });
      }
    }
    if (err.length) {
      callback(err, res);
    } else {
      callback(undefined, res);
    }
  };
  return (
    <div className="border">
      <Cmp
        {...props}
        getFieldDecorator={getFieldDecorator}
        getFieldsValue={getFieldsValue}
        getFieldValue={getFieldValue}
        validateFields={validateFields}
      />
    </div>
  );
};
export default kFormCreate;
```

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
<div class="foo">
  <div> ... </div>
  {
    needDialog &&
    <Dialog>
       <header>Any Header</header>
       <section>Any content</section>
    </Dialog>
   }
</div>
```

具体实现

方案1：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";
import "./index.scss";

export default class Diallog 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,
    );
  }
}
```

```
// Diallog/index.scss
```

```css
.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;
}
```

作业：用createPortal和hooks实现Dialog

方案2：unstable_renderSubtreeIntoContainer

在v16之前，实现“传送门”，要用到react中两个秘而不宣的React API

```jsx
export class Dialog2 extends React.Component {
  render() {
    return null;
  }

  componentDidMount() {
    const doc = window.document;
    this.node = doc.createElement("div");
    doc.body.appendChild(this.node);

    this.createPortal(this.props);
  }

  componentDidUpdate() {
    this.createPortal(this.props);
  }

  componentWillUnmount() {
    unmountComponentAtNode(this.node);
    window.document.body.removeChild(this.node);
  }

  createPortal(props) {
    unstable_renderSubtreeIntoContainer(
      this, //当前组件
      <div className="dialog">{props.children}</div>, // 塞进传送门的JSX
      this.node // 传送门另一端的DOM node
    );
  }
}
```

Reducer

什么是reducer

reducer 就是一个纯函数，接收旧的 state 和 action，返回新的 state。

```
;(previousState, action) => newState
```

之所以将这样的函数称之为 reducer，是因为这种函数与被传入 `Array.prototype.reduce(reducer, ?initialValue)` 里的回调函数属于相同的类型。保持 reducer 纯净非常重要。**永远不要**在 reducer 里做这些操作：

- 修改传入参数；
- 执行有副作用的操作，如 API 请求和路由跳转；
- 调用非纯函数，如 `Date.now()` 或 `Math.random()`。

什么是reduce

此例来自https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;

// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer));
// expected output: 10

// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5));
// expected output: 15
```

思考：有如下函数， 聚合成一个函数，并把第一个函数的返回值传递给下一个函数，如何处理。

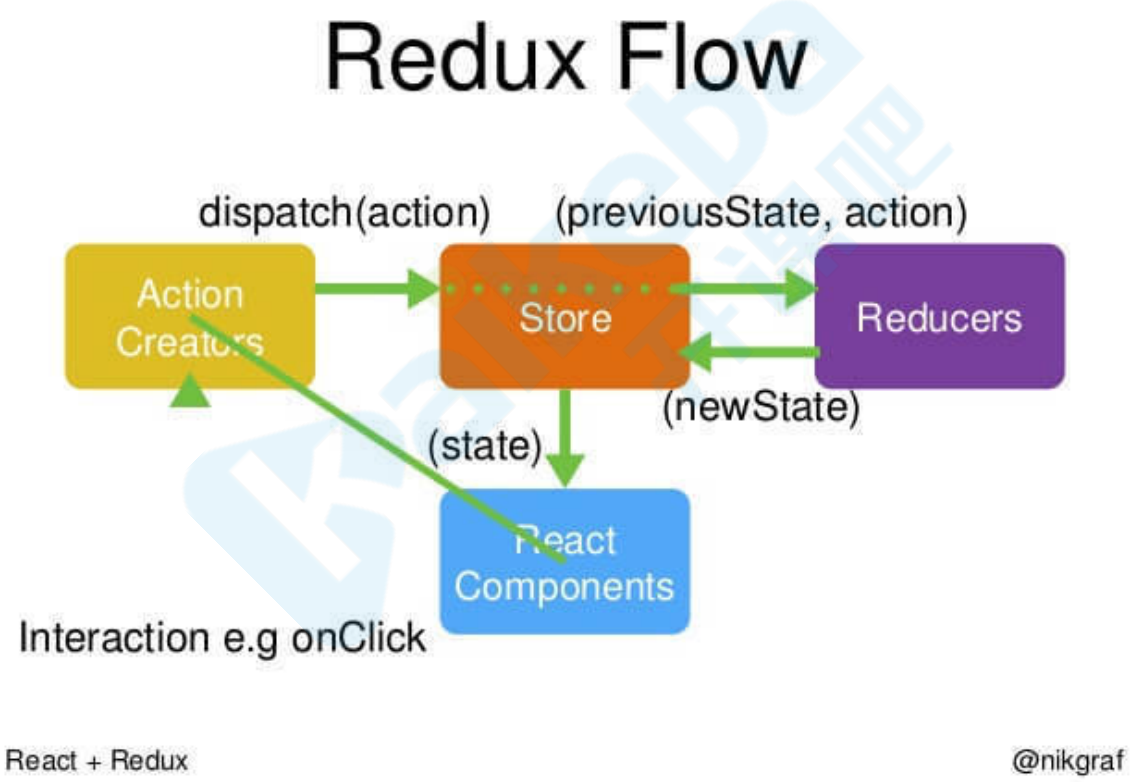
```
function f1(arg) {
  console.log("f1", arg);
  return arg;
}
function f2(arg) {
  console.log("f2", arg);
  return arg;
}
function f3(arg) {
  console.log("f3", arg);
  return arg;
}
```

方法：

```
function compose(...funcs) {
  if (funcs.length === 0) {
    return arg => arg
  }
  if (funcs.length === 1) {
    return funcs[0]
  }
  return funcs.reduce((a, b) => (...args) => a(b(...args)))
}
console.log(compose(f1, f2, f3)("omg"));
```

Redux 上手

Redux是JavaScript应用的状态容器。它保证程序行为一致性且易于测试。



安装redux

`npm install redux --save`

redux上手

redux~~较难上手，是因为上来就有太多的概念需要学习~~，用一个累加器举例

1. 需要一个store来存储数据
2. store里的reducer初始化state并定义state修改规则
3. 通过dispatch一个action来提交对数据的修改
4. action提交到reducer函数里，根据传入的action的type，返回新的state

创建store，src/store/ReduxStore.js

```
import {createStore} from 'redux'

const counterReducer = (state = 0, action) => {
    switch (action.type) {
      case 'add':
        return state + 1
      case 'minus':
        return state - 1
      default:
        return state
    }
  }
const store = createStore(counterReducer)

export default store
```

创建ReduxPage

```
import React, { Component } from "react";
import store from "../store/ReduxStore";

export default class ReduxPage extends Component {
  componentDidMount() {
    store.subscribe(() => {
      console.log("subscribe");
      this.forceUpdate();
      //this.setState({});
    });
  }
  add = () => {
    store.dispatch({ type: "add" });
  };
  minus = () => {
    store.dispatch({ type: "minus" });
  };
  stayStatic = () => {
    store.dispatch({ type: "others" });
  };
  render() {
    console.log("store", store);
    return (
      <div>
        <h1>ReduxPage</h1>
        <p>{store.getState()}</p>
        <button onClick={this.add}>add</button>
        <button onClick={this.minus}>minus</button>
        <button onClick={this.stayStatic}>static</button>
      </div>
    );
  }
}
```

> 如果点击按钮不能更新，因为没有订阅(subscribe)状态变更

还可以在src/index.js的render里订阅状态变更

```
import store from './store/ReduxStore'
const render = ()=>{

  ReactDom.render(
    <App/>,
    document.querySelector('#root')
  )
}
render()

store.subscribe(render)
```

检查点

1. createStore 创建store
2. reducer 初始化、修改状态函数
3. getState 获取状态值
4. dispatch 提交更新
5. subscribe 变更订阅

Redux拓展

核心实现

- 存储状态state
- 获取状态getState
- 更新状态dispatch
- 变更订阅subscribe

kRedux.js

```
export function createStore(reducer, enhancer){
    if (enhancer) {
        return enhancer(createStore)(reducer)
    }
    // 保存状态
    let currentState = undefined;
    // 回调函数
    let currentListeners = [];

    function getState(){
        return currentState
    }
    function subscribe(listener){
        currentListeners.push(listener)
    }
    function dispatch(action){
        currentState = reducer(currentState, action)
        currentListeners.forEach(v=>v())
```

```
        return action
    }
    dispatch({type:'@@OOO/KKB-REDUX'})
    return { getState, subscribe, dispatch}
}
```

store/MyReduxStore.js

```
const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case "add":
      return state + 1;
    case "minus":
      return state - 1;
    default:
      return state;
  }
};
const store = createStore(counterReducer);

export default store;
```

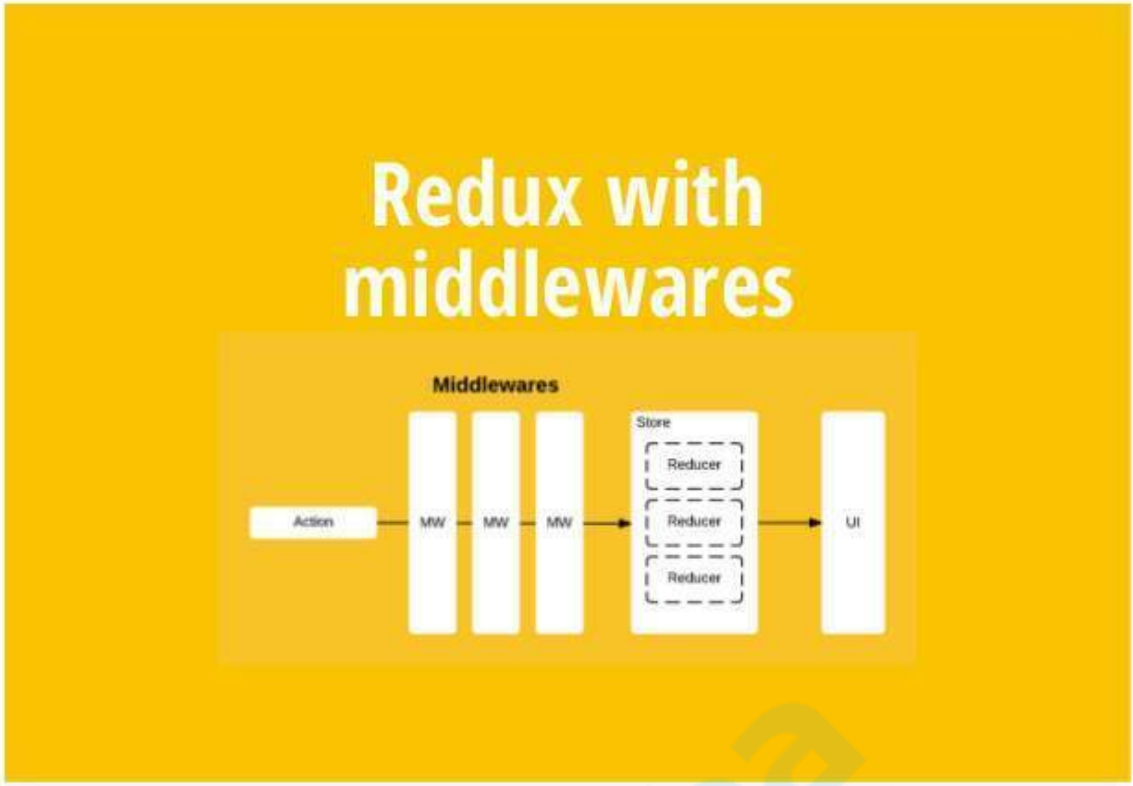
页面可以用原来的ReduxPage.js测试以上代码

异步

Redux只是个纯粹的状态管理器，默认只支持同步，实现异步任务 比如延迟，网络请求，需要中间件的支持，比如我们试用最简单的redux-thunk和redux-logger 。

中间件就是一个函数，对 `store.dispatch` 方法进行改造，在发出 Action 和执行 Reducer 这两步之间，添加了其他功能。

```
npm install redux-thunk redux-logger --save
```



应用中间件，store.js

```
import { createStore, applyMiddleware } from "redux";
import logger from "redux-logger";
import thunk from "redux-thunk";
import counterReducer from './counterReducer'

const store = createStore(counterReducer, applyMiddleware(logger, thunk));
```

使用异步操作时的变化，ReactReduxPage.js

```
const mapDispatchToProps = {
  add: () => {
    return { type: "add" };
  },
  minus: () => {
    return { type: "minus" };
  },
  asyAdd: () => dispatch => {
    setTimeout(() => {
      // 异步结束后，手动执行dispatch
        dispatch({ type: "add" });
    }, 1000);
  },
};
```

中间件实现

核心任务是实现函数序列执行。

//把下面加入kRedux.js

```
export function applyMiddleware(...middlewares){
   // 返回强化以后函数
    return createStore => (...args) => {
        const store = createStore(...args)
        let dispatch = store.dispatch

        const midApi = {
            getState:store.getState,
            dispatch:(...args)=>dispatch(...args)
        }
    // 使中间件可以获取状态值、派发action
        const middlewareChain = middlewares.map(middleware =>
middleware(midApi))
        // compose可以middlewareChain函数数组合并成一个函数
    dispatch = compose(...middlewareChain)(store.dispatch)
        return {
            ...store,
            dispatch
        }
    }
}
export function compose(...funcs) {
  if (funcs.length === 0) {
    return arg => arg
  }
  if (funcs.length === 1) {
    return funcs[0]
  }
  return funcs.reduce((a, b) => (...args) => a(b(...args)))
}
```

redux-logger原理

把下面加入MyReduxStore.js

```
function logger() {
  return dispatch => action => {
    // 中间件任务
    console.log(action.type + "执行了！");
    return dispatch(action);
  };
}

const store = createStore(counterReducer, applyMiddleware(logger));
```

redux-thunk原理

thunk增加了处理函数型action的能力，把下面加入MyReduxStore.js

```
function thunk({ getState }) {
  return dispatch => action => {
    if (typeof action === "function") {
      return action(dispatch, getState);
    } else {
      return dispatch(action);
    }
  };
}
const store = createStore(counterReducer, applyMiddleware(thunk,logger));
```

用以下页面测试：

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

export default class ReduxPage extends Component {
  componentDidMount() {
    store.subscribe(() => {
      //state变化，执行当前的回调
      this.forceUpdate();
    });
  }
  add = () => {
    store.dispatch({ type: "add" });
  };
  asyAdd = () => {
    store.dispatch(dispatch => {
      setTimeout(() => {
        dispatch({ type: "add" });
      }, 1000);
    });
  };

  render() {
    //console.log("store", store);
    return (
      <div>
        <h3>ReduxPage</h3>
        <p>{store.getState()}</p>
        <button onClick={this.add}>add</button>
        <button onClick={this.asyAdd}>asyAdd</button>
      </div>
    );
  }
}
```

作业

2. 用createPortal和hooks实现Dialog

```
import React, { useEffect } from "react";
import { createPortal } from "react-dom";

export default function Dialog() {
  const doc = window.document;
  const node = doc.createElement("div");
  doc.body.appendChild(node);
  useEffect(() => {
    return () => {
      window.document.body.removeChild(node);
    };
  }, []);
  return createPortal(
    <div className="dialog">
      <h1>dialog</h1>
    </div>,
    node,
    // window.document.body
  );
}
```

下节课内容

React全家桶02：掌握react-redux使用及实现、掌握router使用及实现。