

React全家桶01-redux

React全家桶01-redux

课堂目标

资源

知识点

组件跨层级通信 - Context

Context API

`React.createContext`

`Context.Provider`

`Class.contextType`

`Context.Consumer`

使用Context

消费多个Context

注意事项

总结

Reducer

什么是reducer

什么是reduce

Redux 上手

安装redux

redux上手

检查点

Redux拓展

核心实现

异步

中间件实现

redux-logger原理

redux-thunk原理

回顾

作业

下节课内容

课堂目标

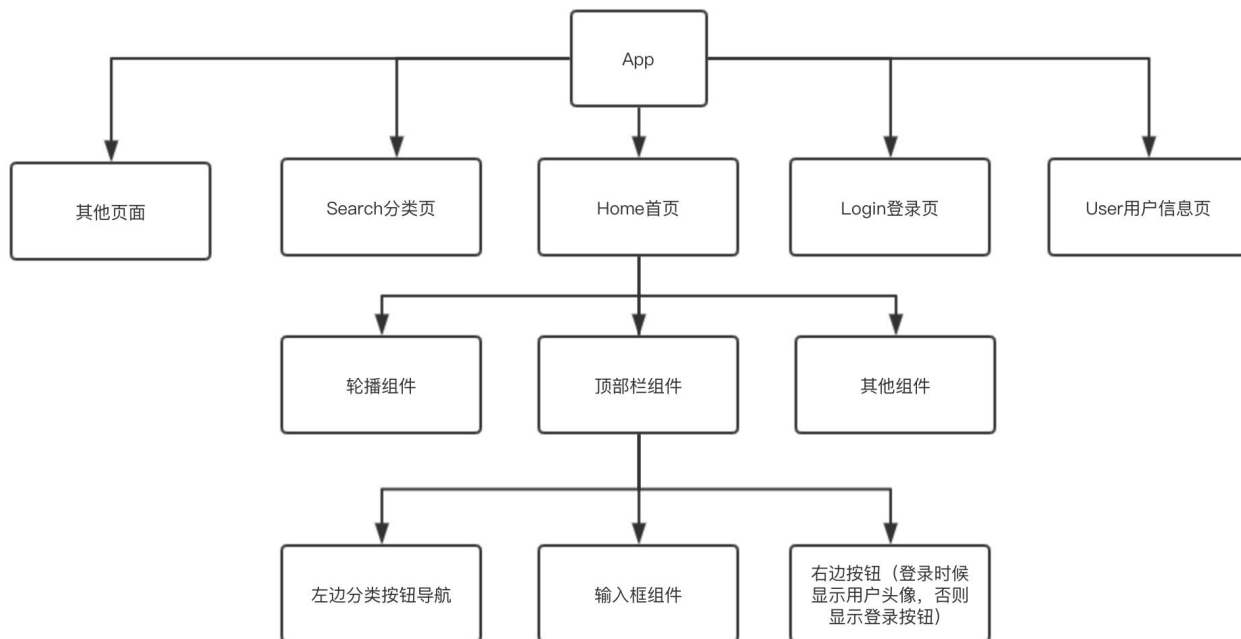
1. 掌握Context
2. 掌握redux使用及实现

资源

1. [Context](#)
2. [redux](#)
3. [redux github](#)

知识点

组件跨层级通信 - Context



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。

React中使用**Context**实现祖代组件向后代组件**跨层级传值**。
Vue中的**provide & inject**来源于Context。

Context API

`React.createContext`

创建一个 Context 对象。当 React 渲染一个订阅了这个 Context 对象的组件，这个组件会从组件树中**离自身最近**的那个**匹配的** `Provider` 中**读取**到当前的 **context 值**。

Context.Provider

Provider 接收一个 value 属性，传递给消费组件，允许消费组件订阅 context 的变化。一个 Provider 可以和多个消费组件有对应关系。多个 Provider 也可以嵌套使用，里层的会覆盖外层的数据。

当 Provider 的 value 值发生变化时，它内部的所有消费组件都会重新渲染。Provider 及其内部 consumer 组件都不受制于 shouldComponentUpdate 函数，因此当 consumer 组件在其祖先组件退出更新的情况下也能更新。

Class.contextType

挂载在 class 上的 contextType 属性会被重赋值为一个由 [React.createContext\(\)](#) 创建的 Context 对象。这能让你使用 this.context 来消费最近 Context 上的那个值。你可以在任何生命周期中访问到它，包括 render 函数中。

你只通过该 API 订阅单一 context。

Context.Consumer

这里，React 组件也可以订阅到 context 变更。这能让你在函数式组件中完成订阅 context。

这个函数接收当前的 context 值，返回一个 React 节点。传递给函数的 value 值等同于往上组件树离这个 context 最近的 Provider 提供的 value 值。如果没有对应的 Provider，value 参数等同于传递给 createContext() 的 defaultValue。

使用Context

可以使用class和function分别制作子孙组件

创建Context => 获取Provider和Consumer => Provider提供值 => Consumer消费值

范例：共享主题色

```
import React, {Component} from "react";
import {ThemeProvider} from "../themeContext";
import ContextTypePage from
"./ContextTypePage";
import ConsumerPage from "../ConsumerPage";

class ContextPage extends Component {
  constructor(props) {
    super(props);
    this.state = {
      theme: {
        themeColor: "red"
      }
    };
  }
}
```

```

changeColor = () => {
  const {themeColor} = this.state.theme;
  this.setState({
    theme: {
      themeColor: themeColor === "red" ?
"green" : "red"
    }
  });
};

render() {
  const {theme} = this.state;
  return (
    <div className="App">
      { /* 组件跨层级通信 */ }
      <button onClick=
{this.changeColor}>change color</button>
      { /* 如果把这里的MyProvider注释掉,
ContextTypePage和ConsumerPage里将取不到theme值, 而
取默认值pink */ }
      <ThemeProvider value={theme}>
        <ContextTypePage />
        <ConsumerPage />
      </ThemeProvider>
    </div>
  );
}
}

export default ContextPage;

```

//themeContext.js

```
import React from "react";

export const ThemeContext =
  React.createContext({themeColor: "pink"});
export const ThemeProvider =
  ThemeContext.Provider;
export const ThemeConsumer =
  ThemeContext.Consumer;
```

// pages/ContextTypePage.js

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

export default class ContextTypePage extends
  Component {
  static contextType = ThemeContext;

  render() {
    console.log("ContextTypePage",
this.context); //sy-log
    const {themeColor} = this.context;
    return (
      <div className="border">
```

```

        <h3 className=
{themeColor}>ContextTypePage</h3>
    </div>
    );
}
}

```

// pages/ConsumerPage.js

```

import React, {Component} from "react";
import {ThemeConsumer} from "../themeContext";

export default class ConsumerPage extends
Component {
  render() {
    return (
      <div className="border">
        <h3>ConsumerPage</h3>
        <ThemeConsumer>{ctx => <HandleTabBar
{...ctx} />}</ThemeConsumer>
      </div>
    );
  }
}

function HandleTabBar({themeColor}) {
  console.log("themeColor", themeColor); //sy-
log

```



```
    return <div className={themeColor}>文本
  </div>;
}
```

消费多个Context

// 补充ContextPage.js

```
<ThemeProvider value={theme}>
  <ContextTypePage />
  <ConsumerPage />

  { /* 多个Context */ }
  <UserProvider value={user}>
    <MultipleContextsPage />
  </UserProvider>
</ThemeProvider>
```

// pages/MultipleContextPage.js

```
import React, {Component} from "react";
import {ThemeConsumer} from "../themeContext";
import {UserConsumer} from "../userContext";

export default class MultipleContextsPage
extends Component {
```

```
render() {  
  return (  
    <div className="border">  
      <h3>MultipleContextsPage</h3>  
      <ThemeConsumer>  
        {theme => (  
          <UserConsumer>  
            {user => <div className=  
{theme.themeColor}>{user.name}</div>}  
          </UserConsumer>  
        )}  
      </ThemeConsumer>  
    </div>  
  );  
}
```

如果两个或者更多的 context 值经常被一起使用，那你可能要考虑一下另外创建你自己的渲染组件，以提供这些值。

注意事项

因为 context 会使用参考标识（reference identity）来决定何时进行渲染，这里可能会有一些陷阱，当 provider 的父组件进行重渲染时，可能会在 consumers 组件中触发意外的渲染。举个例子，当每一次 Provider 重渲染时，以下的代码会重渲染所有下面的 consumers 组件，因为 **value 属性总是**

被赋值为新的对象:

即传入的值是对象的情况下

```
class App extends React.Component {  
  render() {  
    a===a //true  
    return (  
      <Provider value={{something}}  
      'something'}}>  
        <Toolbar />  
      </Provider>  
    );  
  }  
}
```

{a:b}==={a:b} //false
此处可以看出，如果传入的是对象，那么即使传入字面上一模一样的对象，都会被认为是一个新对象，都会重新渲染

为了防止这种情况，将 value 状态提升到父节点的 state

里：此时赋值到 this.state.value，比较点是 this.state.value 的比较，所以每次相等，如果传入的本身就是字符串，则两种方式都可以

```
class App extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      value: {something: 'something'},  
    };  
  }  
  
  render() {  
    return (  
      <Provider value={this.state.value}>  
        <Toolbar />  
      </Provider>  
    );  
  }  
}
```

```
    </Provider>
  );
}
}
```

总结

在React的官方文档中，`Context` 被归类为高级部分 (Advanced)，属于React的高级API，但官方并不建议在稳定版的App中使用Context。

不过，这并非意味着我们不需要关注 `Context`。事实上，很多优秀的React组件都通过Context来完成自己的功能，比如react-redux的`<Provider />`，就是通过Context提供一个全局态的 `store`，路由组件react-router通过Context管理路由状态等等。在React组件开发中，如果用好 `Context`，可以让你的组件变得强大，而且灵活。

函数组件中可以通过`useContext`引入上下文，后面hooks部分介绍。

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"));
```



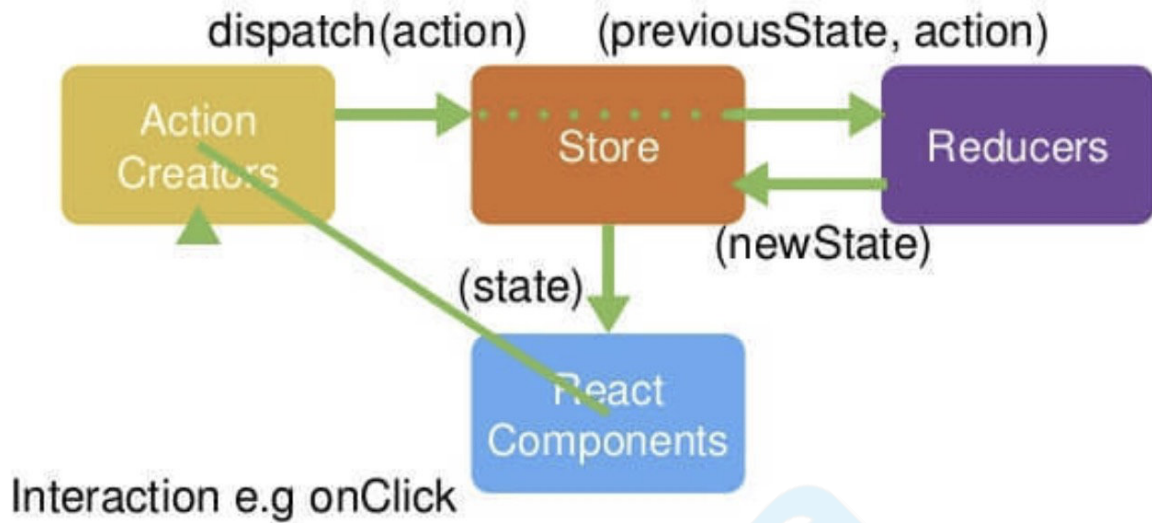
每一个单独的
函数传入的参数

从const reducer = (accumulator, currentValue) => accumulator + currentValue 可以知道reducer的第一个参数是累加器，第二个是每一次添加的内容，所以reduce(a, b)中a是最后一次执行的那个函数，b是每一次执行直至到a的函数

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。

Redux Flow



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/index.js

```
import {createStore} from "redux";

function countReducer(state = 0, action) {
  switch (action.type) {
    case "ADD":
      return state + 1;
    case "MINUS":
      return state - 1;
    default:
      return state;
  }
}

const store = createStore(countReducer);

export default store;
```

创建ReduxPage

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

export default class ReduxPage extends
Component {
  componentDidMount() {
    store.subscribe(() => {
```

```

        this.forceUpdate();
    });
}
add = () => {
    store.dispatch({type: "ADD"});
};
minus = () => {
    store.dispatch({type: "MINUS"});
};
render() {
    console.log("store", store); //sy-log
    return (
        <div>
            <h3>ReduxPage</h3>
            <p>{store.getState()}</p>
            <button onClick={this.add}>add</button>
            <button onClick=
{this.minus}>minus</button>
        </div>
    );
}
}

```

如果点击按钮不能更新，查看是否订阅(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/'
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
  }

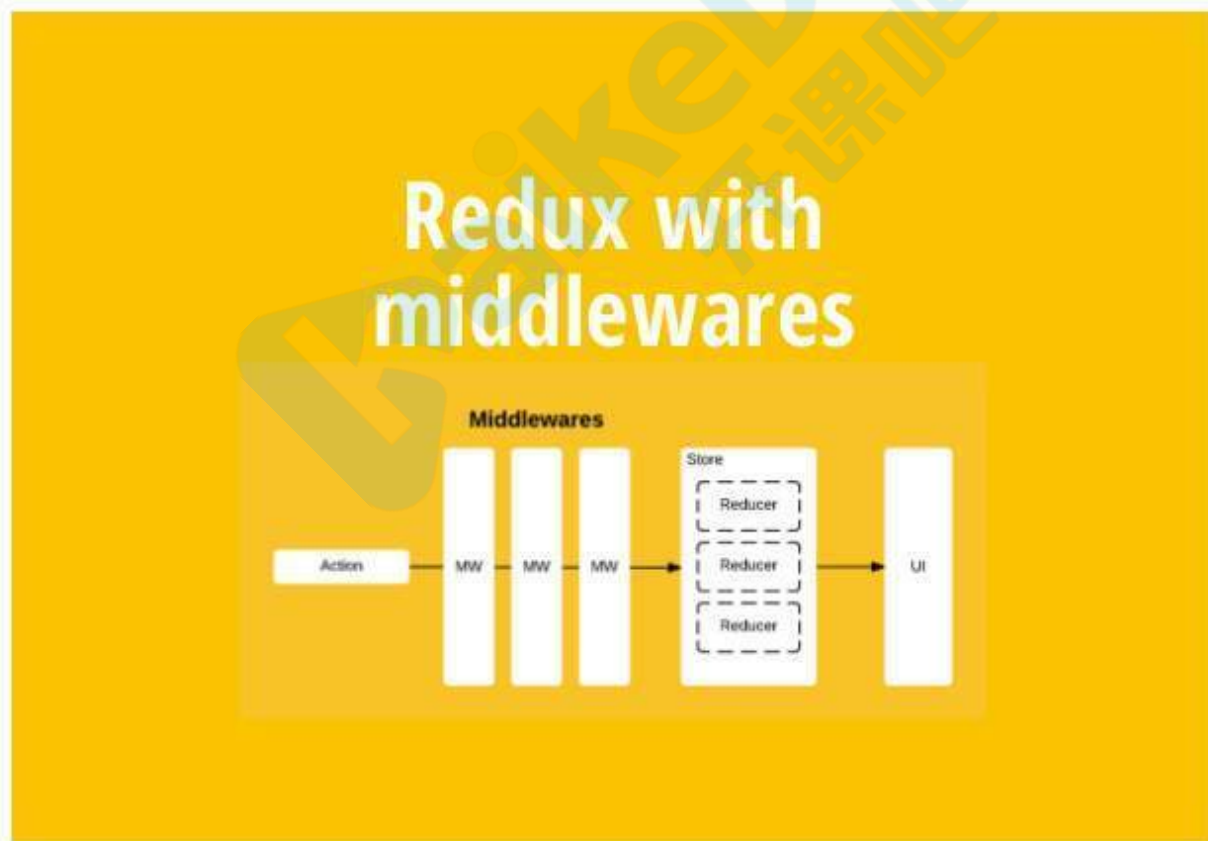
  dispatch({type: '@@OOO/KKB-REDUX'})
  return { getState, subscribe, dispatch}
}
```

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

```
asyAdd = () => {
  store.dispatch(dispatch => {
    setTimeout(() => {
      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函数数组组合成一个函数
  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));
```

回顾

React全家桶01-redux

课堂目标

资源

知识点

组件跨层级通信 - Context

Context API

React.createContext

Context.Provider

Class.contextType

Context.Consumer

使用Context

消费多个Context

注意事项

总结
Reducer
 什么是reducer
 什么是reduce
Redux 上手
 安装redux
 redux上手
 检查点
Redux拓展
 核心实现
异步
中间件实现
 redux-logger原理
 redux-thunk原理
回顾
作业
下节课内容

作业

1. 实现redux、react-redux。
2. 实现输入框的加减法，比如说输入10，加10，用上combineReducer。

下节课内容

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

