redux从入门到深入

尚硅谷前端研究院

# 第1章：redux使用

## 1.1. redux理解

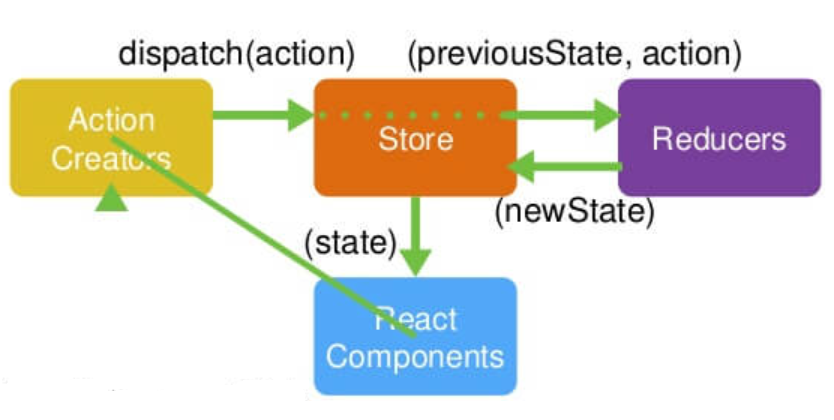
### 1.1.1. 学习文档

1. 英文文档: <https://redux.js.org/>
2. 中文文档: <https://cn.redux.js.org/>
3. Github: <https://github.com/reactjs/redux>

### 1.1.2. redux是什么?

1. redux是一个独立专门用于做状态管理的JS库(不是react插件库)
2. 它可以用在react, angular, vue等项目中, 但基本与react配合使用
3. 作用: 集中式管理react应用中多个组件共享的状态

### 1.1.3. redux工作流程



### 1.1.4. 什么情况下需要使用redux

1. 总体原则: 能不用就不用, 如果不用比较吃力才考虑使用
2. 某个组件的状态，需要共享
3. 某个状态需要在任何地方都可以拿到
4. 一个组件需要改变全局状态
5. 一个组件需要改变另一个组件的状态

## 1.2. redux的核心API

### 1.2.1. createStore()

1. 作用:

创建包含指定reducer的store对象

1. 编码:

import {createStore} from 'redux'

import reducer from './reducer'

const store = createStore(reducer)

### 1.2.2. store对象

1. 作用:

redux库最核心的管理对象

1. 它内部维护着:

state

reducer

1. 核心方法:

getState()

dispatch(action)

subscribe(listener)

1. 编码:

store.getState()

store.dispatch({type:'INCREMENT', number})

store.subscribe(render)

### 1.2.3. applyMiddleware()

1. 作用:

应用上基于redux的中间件(插件库)

1. 编码:

import {createStore, applyMiddleware} from 'redux'

import thunk from 'redux-thunk' // redux异步中间件

const store = createStore(

counter,

applyMiddleware(thunk) // 应用上异步中间件

)

### 1.2.4. combineReducers()

1. 作用:

合并多个reducer函数

1. 编码:

export default combineReducers({

user,

chatUser,

chat

})

## 1.3. redux的三个核心概念

### 1.3.1. action

1. 标识要执行行为的对象
2. 包含2个方面的属性
   1. type: 标识属性, 值为字符串, 唯一, 必要属性
   2. xxx: 数据属性, 值类型任意, 可选属性
3. 例子:

const action = {

type: 'INCREMENT',

data: 2

}

1. Action Creator(创建Action的工厂函数)

const increment = (number) => ({type: 'INCREMENT', data: number})

### 1.3.2. reducer

1. 根据老的state和action, 产生新的state的纯函数
2. 样例

export default function counter(state = 0, action) {

switch (action.type) {

case 'INCREMENT':

return state + action.data

case 'DECREMENT':

return state - action.data

default:

return state

}

}

1. 注意
   1. 返回一个新的状态
   2. 不要修改原来的状态

### 1.3.3. store

1. 将state,action与reducer联系在一起的对象
2. 如何得到此对象?

import {createStore} from 'redux'

import reducer from './reducer'

const store = createStore(reducer)

1. 此对象的功能?

getState(): 得到state

dispatch(action): 分发action, 触发reducer调用, 产生新的state

subscribe(listener): 注册监听, 当产生了新的state时, 自动调用

## 1.4. 使用redux编写应用

### 1.4.1. 效果



### 1.4.2. 下载依赖包

npm install --save redux

### 1.4.3. redux/action-types.js

|  |
| --- |
| */\* action对象的type常量名称模块  \*/* **export const** INCREMENT = **'increment' export const** DECREMENT = **'decrement'** |

### 1.4.4. redux/actions.js

|  |
| --- |
| */\* action creator模块 包含n个action creator函数  \*/* **import** {  INCREMENT,  DECREMENT } **from './action-types'** */\* 增加的action  \*/* **export const** *increment* = (number) => ({**type**: INCREMENT, number})  */\* 减少的action  \*/* **export const** *decrement* = (number) => ({**type**: DECREMENT, number}) |

### 1.4.5. redux/reducer.js

|  |
| --- |
| */\* reducer函数: 根据旧的state和指定的action处理返回新的state  \*/* **import** {  INCREMENT,  DECREMENT } **from './action-types'  export default function** count(state = 0, action) {  ***console***.log(**'count()'**, state, action)  **switch** (action.**type**) {  **case** INCREMENT:  **return** state + action.**number  case** DECREMENT:  **return** state - action.**number  default**:  **return** state  } } |

### 1.4.5. redux/store.js

|  |
| --- |
| */\* redux最核心的管理对象store  \*/* **import** {createStore} **from 'redux' import** *reducer* **from './reducer'  export default** createStore(*reducer*) |

### 1.4.6. App.jsx

|  |
| --- |
| **import** React, {Component} **from 'react' import** PropTypes **from 'prop-types' import** {*increment*, *decrement*} **from './redux/actions'** */\* 应用组件  \*/* **export default class** App **extends** Component {   **static** *propTypes* = {  **store**: PropTypes.object.isRequired  }   constructor (props) {  **super**(props)  **this**.**numberRef** = React.createRef()  }   *increment* = () => {  **const** number = **this**.**numberRef**.**current**.**value**\*1  **this**.**props**.**store**.dispatch(*increment*(number))  }   *decrement* = () => {  **const** number = **this**.**numberRef**.**current**.**value**\*1  **this**.**props**.**store**.dispatch(*decrement*(number))  }   *incrementIfOdd* = () => {  **const** number = **this**.**numberRef**.**current**.**value**\*1  **const** count = **this**.**props**.**store**.getState()  **if** (count%2 === 1) {  **this**.**props**.**store**.dispatch(*increment*(number))  }  }   *incrementAsync* = () => {  **const** number = **this**.**numberRef**.**current**.**value**\*1  *setTimeout*(() => {  **this**.**props**.**store**.dispatch(*increment*(number))  }, 1000)  }   render() {  **const** count = **this**.**props**.**store**.getState()  **return** (  <**div**>  <**p**>click {count}</**p**>  <**div**>  <**select ref=**{**this**.**numberRef**}>  <**option value="1"**>1</**option**>  <**option value="2"**>2</**option**>  <**option value="3"**>3</**option**>  </**select**>**&nbsp;** <**button onClick=**{**this**.*increment*}>+</**button**>**&nbsp;** <**button onClick=**{**this**.*decrement*}>-</**button**>**&nbsp;** <**button onClick=**{**this**.*incrementIfOdd*}>increment if odd</**button**>**&nbsp;** <**button onClick=**{**this**.*incrementAsync*}>increment async</**button**>  </**div**>  </**div**>  )  } } |

### 1.4.7. index.js

|  |
| --- |
| */\* 入口js  \*/* **import** React **from 'react' import** ReactDOM **from 'react-dom'  import** App **from './App' import** store **from './redux/store'** *// 引入store  // 将store传递给App组件* ReactDOM.render(<**App store=**{store}/>, **document**.getElementById(**'root'**))  *// 通过store订阅state改变的监听 ==> 一旦store中的state改变了立即调用回调函数* store.subscribe(() => {  ReactDOM.render(<**App store=**{store}/>, **document**.getElementById(**'root'**)) }) |

### 1.4.8. 问题

1. redux与react组件的代码耦合度太高
2. 编码不够简洁

## 1.5. react-redux

### 1.5.1. 理解

1. 一个react插件库
2. 专门用来简化react应用中使用redux

### 1.5.2. React-Redux将所有组件分成两大类

1. UI组件
   1. 只负责 UI 的呈现，不带有任何业务逻辑
   2. 通过props接收数据(一般数据和函数)
   3. 不使用任何 Redux 的 API
   4. 一般保存在components文件夹下
2. 容器组件
   1. 负责管理数据和业务逻辑，不负责UI的呈现
   2. 使用 Redux 的 API
   3. 一般保存在containers文件夹下

### 1.5.3. 相关API

1. Provider

// 让所有组件都可以得到state数据

<Provider store={store}>  
    <App />  
  </Provider>

1. connect()

// 用于包装 UI 组件生成容器组件  
  connect(  
    mapStateToprops,  
    mapDispatchToProps  
  )(Counter)

1. mapStateToprops()

// 函数: 将state数据转换为UI组件的标签属性  
  function mapStateToProps (state) {

return {

count: state

}

}

1. mapDispatchToProps

// 函数: 将分发action的函数转换为UI组件的标签属性

function mapDispatchToProps(dispatch) {

return {

increment: (number) => dispatch(increment(number)),

decrement: (number) => dispatch(decrement(number)),

}

}

// 对象: 简洁语法, 可以直接指定包含多个action方法

const mapDispatchToProps = {

increment,

decrement

}

### 1.5.4. 使用react-redux

1. 下载依赖包

npm install --save react-redux

1. redux/action-types.js

不变

1. redux/actions.js

不变

1. redux/reducers.js

不变

1. redux/store.js

不变

1. components/Counter.jsx

|  |
| --- |
| **import** React, {Component} **from 'react' import** PropTypes **from 'prop-types'** */\* 应用组件  \*/* **export default class** Counter **extends** Component {   **static** *propTypes* = {  **count**: PropTypes.number.isRequired,  **increment**: PropTypes.func.isRequired,  **decrement**: PropTypes.func.isRequired,  }   constructor (props) {  **super**(props)  **this**.**numberRef** = React.createRef()  }   *increment* = () => {  **const** number = **this**.**numberRef**.**current**.**value**\*1  **this**.**props**.**increment**(number)  }   *decrement* = () => {  **const** number = **this**.**numberRef**.**current**.**value**\*1  **this**.**props**.**decrement**(number)  }   *incrementIfOdd* = () => {  **const** number = **this**.**numberRef**.**current**.**value**\*1  **const** count = **this**.**props**.**count  if** (count%2 === 1) {  **this**.**props**.**increment**(number)  }  }   *incrementAsync* = () => {  **const** number = **this**.**numberRef**.**current**.**value**\*1  *setTimeout*(() => {  **this**.**props**.**increment**(number)  }, 1000)  }   render() {  **const** count = **this**.**props**.**count  return** (  <**div**>  <**p**>click {count}</**p**>  <**div**>  <**select ref=**{**this**.**numberRef**}>  <**option value="1"**>1</**option**>  <**option value="2"**>2</**option**>  <**option value="3"**>3</**option**>  </**select**>**&nbsp;** <**button onClick=**{**this**.*increment*}>+</**button**>**&nbsp;** <**button onClick=**{**this**.*decrement*}>-</**button**>**&nbsp;** <**button onClick=**{**this**.*incrementIfOdd*}>increment if odd</**button**>**&nbsp;** <**button onClick=**{**this**.*incrementAsync*}>increment async</**button**>  </**div**>  </**div**>  )  } } |

1. containters/App.jsx

|  |
| --- |
| */\* 包装UI组件的容器组件 通过connect()生成  \*/* **import** React **from 'react' import** {**connect**} **from 'react-redux'  import** Counter **from '../components/Counter' import** {*increment*, *decrement*} **from '../redux/actions'** */\*function mapStateToProps (state) {  return {  count: state  } }  function mapDispatchToProps(dispatch) {  return {  increment: (number) => dispatch(increment(number)),  decrement: (number) => dispatch(decrement(number)),  } }  const mapDispatchToProps = {  increment,  decrement }  export default connect(  mapStateToProps,  mapDispatchToProps )(Counter)\*/* **export default connect**(  state => ({**count**: state}),  {*increment*, *decrement*} )(Counter) |

1. index.js

|  |
| --- |
| */\* 入口js  \*/* **import** React **from 'react' import** ReactDOM **from 'react-dom' import** {**Provider**} **from 'react-redux'  import** App **from './containers/App' import** store **from './redux/store'** *// 引入store  // 将store传递给Provider组件* ReactDOM.render((  <**Provider store=**{store}>  <**App**/>  </**Provider**> ), **document**.getElementById(**'root'**)) |

### 1.5.5. 问题

1. redux默认是不能进行异步处理的,
2. 应用中又需要在redux中执行异步任务(ajax, 定时器)

## 1.6. redux异步编程

### 1.6.1. 下载redux插件(异步中间件)

npm install --save redux-thunk

### 1.6.2. redux/store.js

|  |
| --- |
| */\* redux最核心的管理对象store  \*/* **import** {createStore, *applyMiddleware*} **from 'redux' import** thunk **from 'redux-thunk' import** *reducer* **from './reducer'  export default** createStore(*reducer*, *applyMiddleware*(thunk)) |

### 1.6.3. redux/actions.js

|  |
| --- |
| */\* 异步增加的异步action  \*/* **export const** *incrementAsync* = **function** (number) {  *// 返回一个带dispatch参数的函数* **return** dispatch => {  *// 执行异步操作  setTimeout*(() => {  *// 有了结果后, 分发同步action* dispatch(*increment*(number))  }, 1000)  } } |

### 1.6.4. components/Counter.jsx

|  |
| --- |
| **static** *propTypes* = {  **count**: PropTypes.number.isRequired,  **increment**: PropTypes.func.isRequired,  **decrement**: PropTypes.func.isRequired,  **incrementAsync**: PropTypes.func.isRequired, }  *incrementAsync* = () => {  **const** number = **this**.**numberRef**.**current**.**value**\*1  **this**.**props**.**incrementAsync**(number) } |

### 1.6.5. containers/App.jsx

|  |
| --- |
| **import** {*increment*, *decrement, incrementAsync}* **from '../redux/actions'**  *// 向外暴露连接App组件的包装组件* **export default** connect(  state => ({**count**: state}),  {*increment*, *decrement, incrementAsync}* )(Counter) |

## 1.7. 使用上redux调试工具

### 1.7.1. 安装chrome浏览器插件



注意: 如果安装不上, 需要翻墙在线安装

如果所有都做好了, 有调试界面但应用异常, 将redux卸载后安装其3.7.2的版本

yarn remove redux / yarn add redux@3.7.2 // 调试工具与浏览器版本问题

### 1.7.2. 下载工具依赖包

npm install --save-dev redux-devtools-extension

### 1.7.3. 编码

|  |
| --- |
| */\* redux最核心的管理对象store  \*/* **import** {createStore, *applyMiddleware*} **from 'redux' import** thunk **from 'redux-thunk' import** {**composeWithDevTools**} **from 'redux-devtools-extension'  import** *reducer* **from './reducer'  export default** createStore(*reducer*, **composeWithDevTools**(*applyMiddleware*(thunk))) |

# 第2章：自定义redux与react-redux

## 2.1. 自定义redux库

### 2.1.1. redux语法功能分析

1. redux库向外暴露下面几个函数

createStore(): 接收的参数为reducer函数, 返回为store对象

combineReducers(): 接收包含n个reducer方法的对象, 返回一个新的reducer函数

applyMiddleware() // 暂不实现

1. store对象的内部结构

getState(): 返回值为内部保存的state数据

dispatch(): 参数为action对象

subscribe(): 参数为监听内部state更新的回调函数

### 2.1.2. 自定义整体结构

|  |
| --- |
| */\* 创建并返回一个store对象  \*/* **export function** *createStore*(reducer) {   */\*  返回当前state值  \*/* **function** *getState*() {   }   */\*  分发指定的action:  \*/* **function** *dispatch*(action) {   }   */\*  订阅state变化的监听  \*/* **function** *subscribe*(lisener) {   }   *// 返回包含3个方法的store对象* **return** {  *getState*,  *dispatch*,  *subscribe* } }  */\* 合并多个reducer函数, 返回一个总的reducer函数  \*/* **export function** *combineReducers*(reducers) {  *// 返回一个新的reducer函数  // 函数接收的是总的state和指定的action* **return** (state = {}, action) => {    } } |

### 2.1.3. 实现createStore函数

|  |
| --- |
| */\* 创建并返回一个store对象  \*/* **export function** *createStore*(reducer) {  **let** state = reducer(undefined, **'@@redux/init'**)  **const** listeners = []   */\*  返回当前state值  \*/* **function** *getState*() {  **return** state  }   */\*  分发指定的action:  1). 调用reducer()得到新的state数据  2). 保存新的state  3). 调用所有监听的回调函数 => 通知组件更新  \*/* **function** *dispatch*(action) {   *// 调用reducer函数得到最新的state值* **const** newState = reducer(state, action)  *// 保存state* state = newState  *// 调用listeners中所有的监听回调* listeners.forEach(listener => listener())  }   */\*  订阅state变化的监听  \*/* **function** *subscribe*(lisener) {  listeners.push(lisener)  }   *// 返回包含3个方法的store对象* **return** {  *getState*,  *dispatch*,  *subscribe* } } |

### 2.1.4. 实现combineReducers函数

|  |
| --- |
| */\* 合并多个reducer函数, 返回一个总的reducer函数  \*/* **export function** *combineReducers*(reducers) {  *// 返回一个新的reducer函数  // 函数接收的是总的state和指定的action* **return** (state = {}, action) => {  *// 遍历调用所有的reducer, 并得到其返回的新状态值, 并封装成对象作为总的新state对象* **const** newState = Object.keys(reducers).reduce((preState, key) => {  preState[key] = reducers[key](state[key], action)  **return** preState  }, {})  *// 返回新的状态对象* **return** newState  } } |

## 2.2. 自定义react-redux库

### 2.2.1. react-redux语法功能分析

1. react-redux向外暴露了2个API

a. Provider组件类

b. connect函数

1. Provider组件

接收store属性

让所有容器组件都可以看到store, 从而通过store读取/更新状态

1. connect函数

接收2个参数: mapStateToProps和mapDispatchToProps

mapStateToProps: 为一个函数, 用来指定向UI组件传递哪些一般属性

mapDispatchToProps: 为一个函数或对象, 用来指定向UI组件传递哪些函数属性

connect()执行的返回值为一个高阶组件: 包装UI组件, 返回一个新的容器组件

容器组件会向UI传入前面指定的一般/函数类型属性

### 2.2.2. 组件context的理解和使用

1. context的理解

相对于props, context可以非常方便直接的将数据传递给任何后代组件,

而不用像props那样逐层传递, 相当于提供了一个全局数据

在应用开发中不太建议使用context, 但react-redux库用它来共享store

1. context的使用

|  |
| --- |
| <**div id="test"**></**div**>  <**script type="text/javascript" src="./js/react.development.js"**></**script**> <**script type="text/javascript" src="./js/react-dom.development.js"**></**script**> <**script type="text/javascript" src="./js/prop-types.js"**></**script**> <**script type="text/javascript" src="./js/babel.min.js"**></**script**>  <**script type="text/babel"**>  **class** A **extends *React***.**Component** {   **state** = {  **color**: **'red'** }   */\*  声明向后代组件传递的context中的数据  \*/* **static** *childContextTypes* = {  **color**: ***PropTypes***.**string** }   */\*  给后代组件返回包含指定数据的context对象  \*/* getChildContext() {  **return** {**color**: **this**.**state**.**color**};  }   render () {  **return** (  <**div**>  <**h2**>A组件</**h2**>  <**B** />  </**div**>  )  }  }   **class** B **extends *React***.**Component** {   render () {  **return** (  <**div**>  <**h2**>B组件</**h2**>  <**C** />  </**div**>  )  }  }   **class** C **extends *React***.**Component** {   **constructor** (props, context) {  **super**(props)   ***console***.log(**'C'**, context.**color**)  }   */\*  声明接收context中的数据  \*/* **static** *contextTypes* = {  **color**: ***PropTypes***.**string** }   render () {  **return** (  <**div**>  <**h2 style=**{{**color**: **this**.**context**.**color**}}>C组件</**h2**>  </**div**>  )  }  }   ***ReactDOM***.render(<**A** />, **document**.getElementById(**'test'**)) </**script**> |

### 2.2.3. 自定义整体结构

|  |
| --- |
| */\* 用来为所有容器组件提供store的组件类  \*/* **export class** Provider **extends** React.PureComponent {   **static** *propTypes* = {  **store**: PropTypes.object.isRequired  }   render() {  *// 原样渲染<Provider>的所有子节点* **return this**.**props**.**children** } }  */\* 用来包装UI组件生成容器组件的高阶函数  \*/* **export function** *connect*(mapStateToProps, mapDispatchToProps) {  *// 返回一个高阶组件: 接收UI组件, 返回容器组件* **return** (UIComponent) => {   **return class** ConnectComponent **extends** React.PureComponent {   render() {  **return** <**UIComponent** />  }  }  } } |

### 2.2.4. 实现Provider组件

|  |
| --- |
| */\* 用来为所有容器组件提供store的组件类  \*/*  **export class** Provider **extends** React.PureComponent {   **static** *propTypes* = {  **store**: PropTypes.object.isRequired  }   */\*  声明向后代组件传递的context中的数据  \*/* **static** *childContextTypes* = {  **store**: PropTypes.object.isRequired  }   */\*  给后代组件返回包含指定数据的context对象  \*/* getChildContext () {  **return** {  **store**: **this**.**props**.**store** }  }   render() {  *// 原样渲染<Provider>的所有子节点* **return this**.**props**.**children** } } |

### 2.2.5. 实现connect函数

|  |
| --- |
| */\* 用来包装UI组件生成容器组件的高阶函数  \*/* **export function** *connect*(mapStateToProps = () => {}, mapDispatchToProps = {}) {  *// 返回一个高阶组件: 接收UI组件, 返回容器组件* **return** (UIComponent) => {   **return class** ConnectComponent **extends** React.PureComponent {   *// 声明接收全局store* **static** *contextTypes* = {  **store**: PropTypes.object.isRequired  }   constructor(props, context) {  **super**(props)   *// 从context中得到store* **const** store = context.**store** *// 调用第一个参数函数, 得到包含所有需要传递的一般属性的对象* **const** stateProps = mapStateToProps(store.getState())   **let** dispathProps  *// 如果第二个参数是函数, 调用它得到包含所有需要传递的函数属性的对象* **if** (**typeof** mapDispatchToProps === **'function'**) {  dispathProps = mapDispatchToProps(store.**dispatch**)  } **else** { *// 第二个参数是对象  // 遍历对象中所有方法, 生成包含所有需要传递的函数属性的对象* dispathProps = Object.keys(mapDispatchToProps).reduce((pre, key) => {   **const** actionCreator = mapDispatchToProps[key]  pre[key] = **function** (...args) {  store.dispatch(actionCreator(...args))  }   **return** pre  }, {})  }   *// 将包含一般属性的对象初始化为容器组件的状态* **this**.**state** = {  ...stateProps  }   *// 将包含函数属性的对象保存到容器组件上* **this**.**dispathProps** = dispathProps   *// 订阅store状态变化的监听* store.subscribe(() => {  *// 一旦store中的state有变化, 更新容器组件状态, 从而导致UI组件重新* **this**.setState(mapStateToProps(store.getState()))  })  }   render() {   *// 返回UI组件标签, 传递所有准备好的一般属性和函数属性* **return** <**UIComponent** {...**this**.**state**} {...**this**.**dispathProps**}/>  }  }  } } |