

React

1 导入导出

- import 和 export 都必须放在顶级域
- import 必须放在script代码最上面

1.1 import 和 require

1.1.1 import

属于加载前置

属于加载前置的机制，因此将其全放在代码顶部，代码逐个解析import获取一个引入的列表，先引入依赖，再去向下执行代码

1.1.2 require

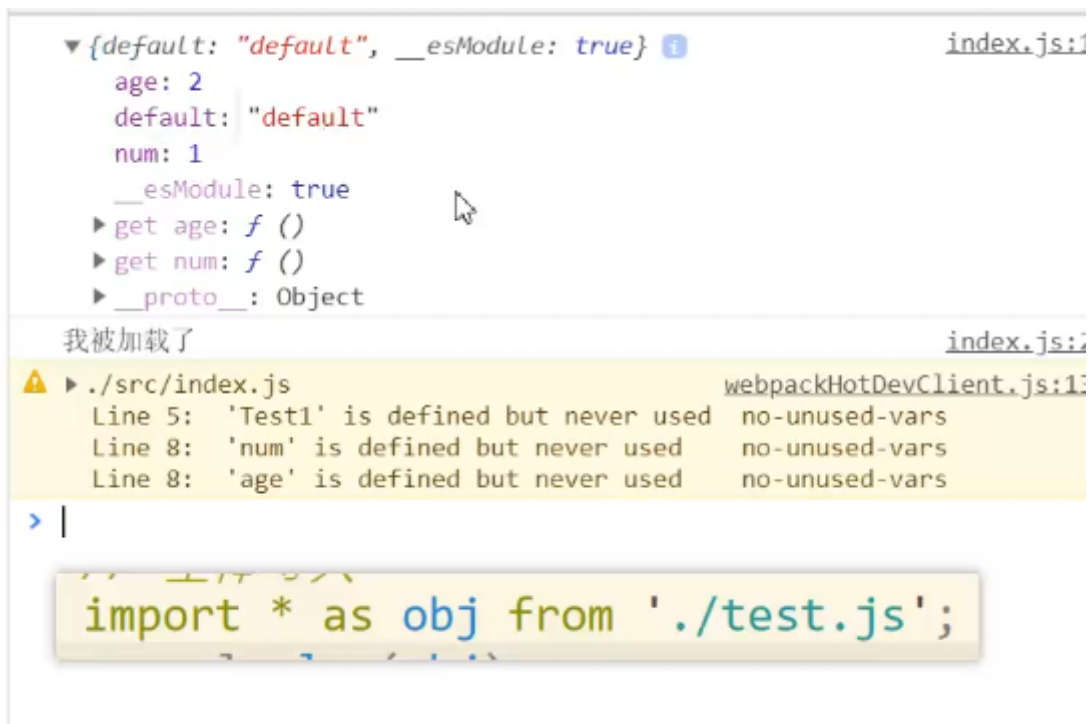
属于加载滞后

代码执行到此行才进行加载

```
1 if (true) {  
2   let querStr = require('querStr')  
3 }
```

1.2 全体导入

全体导入进来 使用 * 号，通过 as 起一个别名



The screenshot shows a code editor with a module object and console logs. The module object is expanded, showing properties like `age`, `default`, `num`, `__esModule`, `get age`, `get num`, and `__proto__`. Below the object, there are console logs indicating that the module was loaded and that certain variables (`Test1`, `num`, `age`) were defined but never used, resulting in warnings.

```
▼ {default: "default", __esModule: true} index.js:1  
  age: 2  
  default: "default"  
  num: 1  
  __esModule: true  
  ▶ get age: f ()  
  ▶ get num: f ()  
  ▶ __proto__: Object  
我被加载了 index.js:1  
⚠ ▶ ./src/index.js webpackHotDevClient.js:1  
  Line 5: 'Test1' is defined but never used no-unused-vars  
  Line 8: 'num' is defined but never used no-unused-vars  
  Line 8: 'age' is defined but never used no-unused-vars  
> |  
import * as obj from './test.js';
```

2 class

```
1  class Obj1 {
2      // 静态属性
3      // 其中不可能有后面生成的数据
4      static staticAge = 999;
5      static staticFn = function () {
6          console.log(this) // 是Obj1 这个构造函数
7          console.log('静态函数')
8      };
9      // 实例属性
10     // 实例可以访问静态属性 因为先有静态
11     myAge = 123;
12     myFn () {
13         console.log('实例的函数', this.myAge)
14     }
15 }
16
17 let o1 = new Obj1()
18
```

2.1 类继承

```
1  class Person {
2      age = 100;
3      constructor (props) {
4          this.age = props.age
5          console.log('先触发父类的构造器')
6      }
7  }
8
9  class Boy extends Person {
10     name = "zhaosi"
11     constructor (props) {
12         this.name = props.name
13         console.log('后触发子类的类构造器')
14     }
15 }
16
17 let boy = new Boy({name: 'zhaosisi', age: 100})
```

3 使用脚手架

3.1 安装

任意目录全局安装

```
1 | npm i -g create-react-app
```

更新npm

```
1 | npm i -g npm to update
```

3.2 使用

- create-react-app 项目名 options 构建项目结构
- cd 项目目录 => npm i 安装依赖

3.3 运行

- npm run start 启动
- npm run build 生成dist

4 基本操作总结

1. 引入React对象
2. 引入ReactDOM对象
3. 操作jsx
 -

4.1 组件体验

```
1  import React, { Component } from 'react'
2
3  class App extends Component {
4    // 初始化组件自己的属性
5    constructor () {
6      super();
7      this.state = {
8        num: 1
9      }
10   }
11   render () {
12     return (
13       // 保证一个根节点
14       <div>
15         我是react
16         <hr />
17         { this.state.num }
18       </div>
19     )
20   }
21 }
22
23 export default App
24
```

4.1.1 双向的输入框数据绑定

4.1.1.1 写法1

```
1  import React, { Component } from 'react'
2
3  class App extends Component {
4    // 初始化组件自己的属性
5    constructor () {
6      super();
```

```

7     this.state = {
8       num: 1
9     }
10  }
11  changeHandler (e) {
12    console.log(e.target.value)
13    this.setState({
14      num: e.target.value
15    })
16  }
17  render () {
18    return (
19      // 保证一个根节点
20      <div>
21        我是react
22        <hr />
23        { this.state.num }
24        <hr/>
25        <input value={ this.state.num } onChange={ (e)=>{
26          this.changeHandler(e)
27        } } />
28      </div>
29    )
30  }
31 }
32
33 export default App

```

4.1.1.2 写法2

```

1  import React, { Component } from 'react'
2
3  export class App2 extends Component {
4    constructor () {
5      super()
6      this.state = {
7        num: 2
8      }
9      // 绑定this
10     this.changeHandler = this.changeHandler.bind(this)
11   }
12   changeHandler (e) {
13     console.log(e.target.value)
14     // 未作处理的情况下 this 为undefined
15     console.log(this)
16     this.setState({
17       num: e.target.value
18     })
19   }
20   render () {
21     return (
22       <div>
23         <span>
24           { this.state.num }
25         <hr />
26         { /* 在{ } 中调用的时候 其实是在顶级域中进行调用 丢死了原有的this */ }

```

```

27     <input type="text" value={ this.state.num } onChange={
this.handleChange } />
28     </span>
29   </div>
30 )
31 }
32 }

```

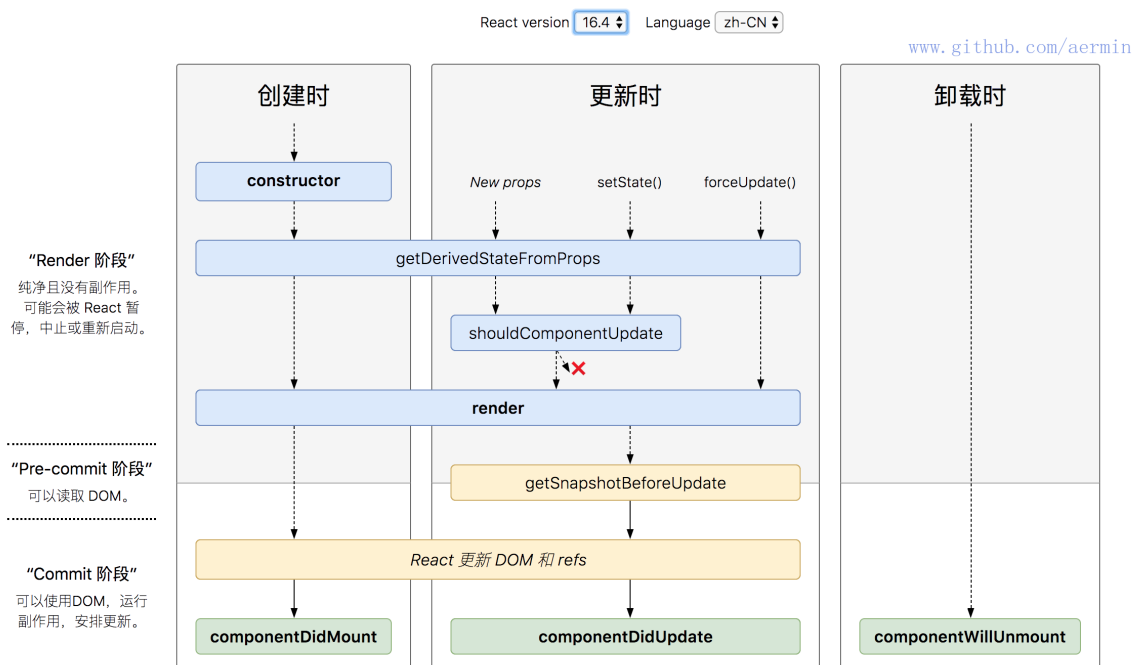
React.Component: 由于使用了 ES6，这里会有些微不同，属性并不会自动绑定到 React 类的实例上。

```

1 import React from 'react';
2 class TodoItem extends React.Component{
3   constructor(props){
4     super(props);
5   }
6   handleClick(){
7     console.log(this); // null
8   }
9   handleFocus(){ // manually bind this
10    console.log(this); // React Component Instance
11  }
12  handleBlur: ()=>{ // use arrow function
13    console.log(this); // React Component Instance
14  }
15  render(){
16    return <input onClick={this.handleClick}
17                      onFocus={this.handleClick.bind(this)}
18                      onBlur={this.handleClick}/>
19  }
20 }

```

4.2 生命周期



```

1 class App extends Component {
2   constructor () {
3     console.log('constructor 1')
4     super()
5     this.state = {
6       num: 1

```

```

7     }
8   }
9   // 不推荐发起网络请求 会引起渲染阻塞
10  componentWillMount () {
11    console.log(' 将要挂载 挂载之前 componentWillMount 2')
12  }
13  render () {
14    console.log('render 3')
15    return (
16      <div>
17        { this.state.num }
18        <button onClick={e=>{
19          this.setState({
20            num: 10
21          })
22        }}>
23          更改数据
24        </button>
25      </div>
26    )
27  }
28  // 发起网络请求 可能会引起二次render
29  componentDidMount () {
30    /*
31     这个方法是建立任何订阅 发起网络请求的好地方
32     但应该在卸载的时候取消订阅
33    */
34    console.log('已经挂载 componentDidMount 4')
35  }
36  // 控制更新
37  shouldComponentUpdate () {
38    console.log('控制更新 shouldComponentUpdate')
39    // 会在更新之前调用 如果返回true则进行更新 否则不进行更新
40    return false
41  }
42  // 将要更新 更新后触发 render
43  componentWillUpdate () {
44    console.log('更新之前 将要更新 componentWillUpdate 更新')
45  }
46  // 参数: 之前的属性 数据 快照
47  // 更改数据之后才触发
48  componentDidUpdate (prevProps, prevState, snapshot) {
49    console.log('已经更新 componentDidUpdate 更新')
50  }
51  // 将要卸载
52  componentWillUnmount () {
53    console.log(' 卸载之前 componentWillUnmount end')
54  }
55 }
56
57 export default App;

```

4.3 组件传值

4.3.1 父组件传值子组件

父组件

```

1  class App extends React.Component {
2    constructor () {
3      super()
4      this.state = {
5        num: 12,
6        name: 'zhaosi'
7      }
8    }
9    render () {
10     let header = (
11       <div>
12         头部
13       </div>
14     )
15     let footer = (
16       <div>
17         底部
18       </div>
19     )
20     return (
21       <div>
22         我是App父组件，以下使用Son组件
23         <hr />
24         {/* 组件的使用必须首字母大写 */}
25         {/* 通过属性传递子组件数据 */}
26         {/* 通过变量传递DOM */}
27         <Son age={ this.state.num } name={ this.state.name }
28           header={ header } footer={ footer } text={ 11 }
29         >
30           <ul>
31             <li>haha</li>
32             <li>kekeke</li>
33             <li>hdwdawd</li>
34           </ul>
35         </Son>
36       </div>
37     )
38   }
39 }

```

子组件

内联样式的编写

prop 属性验证

```

1  import React from 'react'
2  // 引入传值约束的包
3  import PropTypes from 'prop-types'
4
5  class Son extends React.Component {
6
7    // prop 属性的约定
8    static propTypes = {
9      text: PropTypes.string.isRequired || Number
10    }
11

```

```

12   static defaultProps = {
13     text: 'abc'
14   }
15
16   constructor (props) {
17     super(props)
18     this.state = {
19       num: 1
20     }
21   }
22   render () {
23     // 解构赋值 拿到prop中相同属性名的值
24     // 声明一个name和age属性，对this.props中的同名属性进行赋值
25     let {age, name, text} = this.props
26     console.log('props', this.props)
27     return (
28       <div>
29         我是son 子组件
30         <hr />
31         { text }
32         <hr />
33         { age }, {name}
34         <hr />
35         {/* 传入的底部DOM 内联的样式必须放在 {} 中，然后是一个对象 使用驼峰命名 */}
36         <div style={ {backgroundColor: 'red'} }>
37           { this.props.header }
38         </div>
39
40         {/* 必须显示的在子组件中输出 才能显示在父组件中的子组件里slot中的DOM */}
41         { this.props.children }
42
43         {/* 传入的头部DOM */}
44         <div style={ {backgroundColor: 'green'} }>
45           { this.props.header }
46         </div>
47       </div>
48     )
49   }
50 }
51
52 export default Son

```

4.4 混合包装

4.4.1 包装组件

```

1   import React from 'react'
2
3   class Son extends React.Component {
4     constructor () {
5       super()
6       this.state = {
7         num: 1
8       }
9     }
10    componentDidMount () {
11      console.log('son 组件原有的功能')

```



```

12     }
13     render () {
14         return (
15             <React.Fragment>
16                 Son 子组件
17                 <hr />
18                 { this.state.num }
19             </React.Fragment>
20         )
21     }
22 }
23
24 export default class wrap extends React.Component {
25     constructor () {
26         super()
27         this.state = {
28
29         }
30     }
31     componentDidMount () {
32         console.log('包装组件中的功能')
33     }
34     render () {
35         return <Son />
36     }
37 }
38

```

4.4.2 使用包装组件

```

1  import React from 'react';
2  import './App.css';
3  import wrap from './son'
4
5  function App() {
6      return (
7          <div className="App">
8              <wrap />>
9          </div>
10     );
11 }
12
13 export default App;
14

```

4.5 路由

4.5.1 使用

4.5.1.1 安装

- npm

```

1  // 在浏览器中使用
2  npm i react-router-dom

```

- yarn

```
1 | yarn add react-router-dom
```

4.5.1.2 基本使用

<https://reacttraining.com/react-router/web/api/Redirect>

```
1  import React from 'react';
2  import './App.css';
3  // 专门用来给浏览器使用的
4  import {HashRouter, BrowserRouter as Router, Route,
5  NavLink, Switch, Redirect} from 'react-router-dom'
6
7  class User extends React.Component {
8    componentDidMount () {
9      // 获取id
10     // 第一种方式获取
11     console.log(this.props.match.params.id)
12     console.log(this.props)
13   }
14   render () {
15     return (
16       <div>
17         <h1>
18           用户组件
19           <button onClick={ e=> {
20             this.props.history.goBack()
21           } }>
22             后退 前进
23           </button>
24         </h1>
25       </div>
26     )
27   }
28 }
29
30 class Man extends React.Component {
31   render () {
32     return (
33       <div>
34         <h3>男人</h3>
35       </div>
36     )
37   }
38 }
39
40 class Woman extends React.Component {
41   render () {
42     return (
43       <div>
44         <h3>女人</h3>
45       </div>
46     )
47   }
48 }
49
```

```

50 class Home extends React.Component {
51   render () {
52     return (
53       <div>
54         <h1>
55           我是首页
56         </h1>
57         <h2>以下是可变内容</h2>
58         { /* 嵌套路由必须写完整路径 */ }
59         { /* 相同路由造成横向的重复匹配
60            会同时出现 “男人”， “女人”
61            */ }
62         <Switch>
63           { /* 使用Switch 只匹配一个 */ }
64           <Route path="/a/man" component={ Man } />
65           <Route path="/a/man" component={ woman } />
66           <Route path="/a/woman" component={ woman } />
67         </Switch>
68       </div>
69     )
70   }
71 }
72
73
74 class App extends React.Component {
75   render () {
76     let pathObj1 = {
77       pathname: '/user/4',
78     }
79
80     let pathObj2_userq = {
81       pathname: '/userq',
82       search: '?id=11',
83       state: { name: 'zhaosi' },
84       query: { sex: '000' }
85     }
86     return (
87       <div className="App">
88         <h1> 头部 </h1>
89
90         { /* Router 相当于规则和坑 */ }
91         <Router>
92           { /* 因为Router 只能有一个子节点 */ }
93           <React.Fragment>
94             <NavLink to="/a/man" activestyle={{ color: '#4dco60' }}>
95               男
96             </NavLink>
97             <NavLink to="/a/woman" activeClassName="selected">
98               女
99             </NavLink>
100
101             { /* 第一种传参方式 params 直接写路由 */ }
102             <NavLink to="/user/10" activeClassName="selected">
103               去user 字符串传
104             </NavLink>
105             <NavLink to={ pathObj1 } activeClassName="selected">
106               去user 对象方式穿
107             </NavLink>

```

```

108         {/* 第二种方式 query */}
109         <NavLink to="/userq?id=1" activeClassName="selected">
110             去userq 字符串传
111         </NavLink>
112         {/* 在this.props.location 传 */}
113         <NavLink to={ pathObj2_userq }
114             activeClassName="selected">
115             去userq 对象传
116         </NavLink>
117         <Switch>
118             {/* 模糊匹配 只要以/开头的路由都会匹配到
119              如果不加 exact(精确匹配) 路由为 "/a"时会出现两个 "首页"
120             */}
121             <Route path="/" exact component={ Home } />
122             {/* 需要有嵌套路由的 不能使用exact */}
123             <Route path="/a" component={ Home } />
124
125             {/* 传递参数 路由参数 传递字符串 */}
126             <Route path="/user/:id" component={ User } />
127
128             <Route path="/userq" component={ User } />
129             {/* 重定向 */}
130             <Redirect to="/" />
131         </Switch>
132     </React.Fragment>
133 </Router>
134
135     <h1> 底部 </h1>
136 </div>
137 )
138 }
139 }
140
141 export default App;
142

```

- exact

精确匹配 (必须精确匹配到path的字符串)

纵向深入匹配

有嵌套路由的 不能使用exact 进行精确匹配

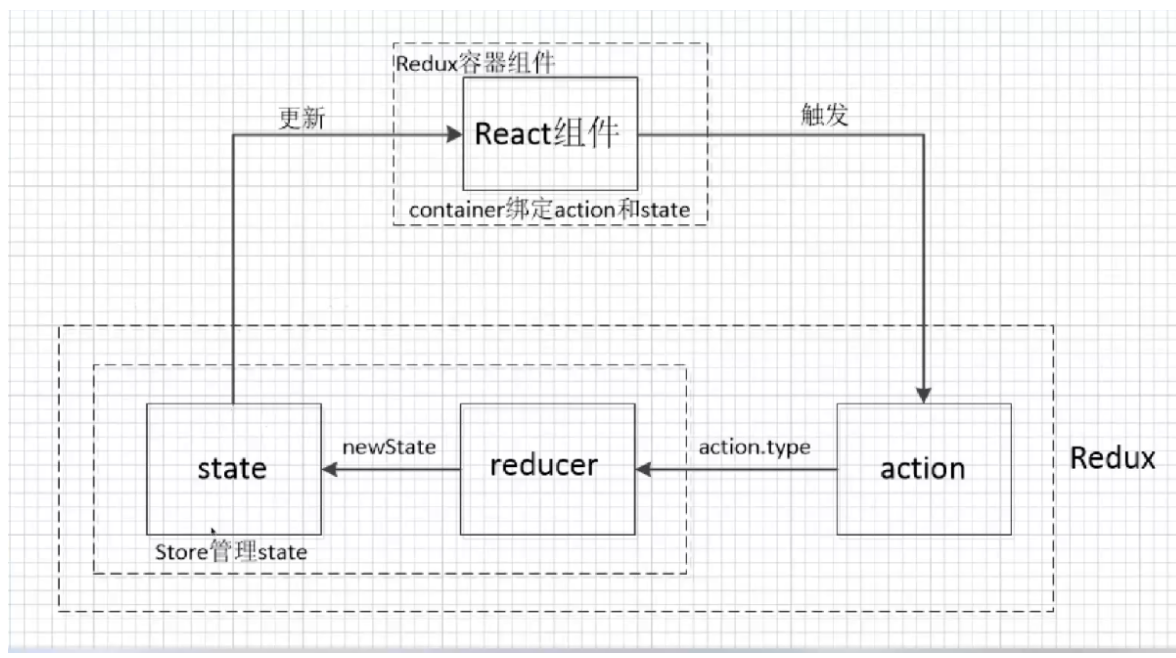
- switch

横向匹配 选择一个

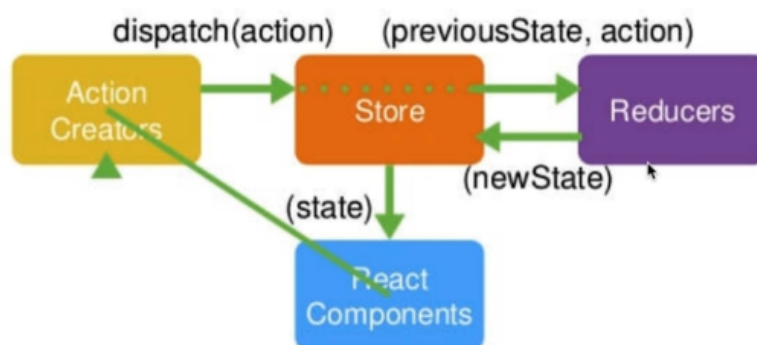
被包裹的

从上到下进行匹配

Redux



Redux Flow



1 安装

```
1 | npm i redux react-redux redux-thunk -S
```

2 使用

1. 新建actions目录

存放发出的actions 指令

- 新建一个user指令集文件 来更新store 中 user 的数据

```
1 | --actions
```

```

2      --user.js
3      // 定义的常量文件
4      import * as userConstants from '../constants/user'
5
6      /*
7       * actions 就是一条对象信息 带着一些描述
8       action 发完 会给到 reducers里
9       */
10
11     /**
12      * 运行login 同时把需要的数据传递进去
13      * 然后其会返回一个对象 对象会带着type的信息 作用始位了USER_LOGIN
14      * type 就是 action 的名字
15      * @param {*} payload 发出action带上的参数
16      */
17     export const login = (payload) => {
18         return {
19             type: userConstants.USER_LOGIN,
20             payload
21         }
22     }
23
24     // 发出指令退出
25     export const logout = (payload) => {
26         return {
27             type: userConstants.USER_LOGOUT,
28             payload
29         }
30     }
31
32     // 发出指令 进行登录
33     export const update = (payload) => {
34         return {
35             type: userConstants.USER_UPDATE,
36             payload
37         }
38     }

```

2. reducers

收到actions 指令 对数据进行处理 产生新的state数据

- 新建user.js 对应指令集发来的指令进行处理

```

1      --reducers
2      --user.js
3      --index.js
4
5      import * as userConstants from '../constants/user'
6
7      /**
8       * 识别发来的action 是干什么的action
9       * @param {*} state
10      * @param {*} action
11      */
12     export default function (state = {}, action) {
13         switch (action.type) {
14             // 在匹配到了一个action 的时候 需要返回一个新的数据

```

```

15     case userConstants.USER_LOGIN:
16         return {
17             ...state,
18             ...action.payload,
19             isLoading: true
20         }
21
22     case userConstants.USER_LOGOUT:
23         return {
24             isLoading: false
25         }
26
27     case userConstants.USER_UPDATE:
28         return {
29             ...state,
30             ...action.payload,
31             isLoading: true
32         }
33
34     // 必须有个默认值
35     default: {
36         // console.log(state)
37         return {
38             ...state
39         }
40     }
41 }
42 }

```

- 新建index.js 合并reducers中的处理指令中的文件

```

1  // 导入用于合并的方法
2  import { combineReducers } from 'redux'
3
4  import user from './user'
5  import city from './city'
6
7  // 存在多个reducers 合并以下
8  export default combineReducers({
9      user,
10     // city, 如果存在此文件, 则进行合并
11 })

```

3. store

存放数据和状态

- 新建index.js

```

1  /**
2   全局只有一个store
3  */
4  import { createStore } from 'redux'
5  // 拿到合并的reducers
6  import reducer from '../reducers'
7
8  /**

```

```

9   * 通过函数去创建store
10  * @param {*} initState
11  */
12  export default function Store (initStroe) {
13    return createStore(reducer, initState)
14  }

```

4. constants

因为actions 需要写一个名字

reducers 需要匹配一个名字

这两个名字会在两个地方用到，重复出现的单独抽出来

- 新建 对应 actions 中user指令集的常量文件

```

1  --constants
2    --user.js
3  /**
4   * 常量定义
5   */
6  export const USER_LOGIN = 'USER_LOGIN'
7  export const USER_LOGOUT = 'USER_LOGOUT'
8  export const USER_UPDATE = 'USER_UPDATE'

```

5. index.js

```

1  // 引入创建好的store方法
2  import Store from './store'
3  // 引入包装的组件
4  import { Provider } from 'react-redux'
5
6  // 初始化一个store
7  // 名字 必须和reducers 中的对应
8  const store = Store({
9    user: {
10      isLoading: false
11    },
12    city: {
13      '024': '西安'
14    }
15  })
16
17  ReactDOM.render(
18    // 传入store
19    <Provider store = { store}>
20      <App />
21    </Provider>,
22    document.getElementById('root')
23  );

```

6. 在组件中使用 列如在App.js 中使用

```

1  import React from 'react';
2  import { connect } from 'react-redux'
3  import logo from './logo.svg';

```



```

4 // 使用action
5 import * as userAction from './actions/user'
6 import './App.css';
7
8 class App extends React.Component {
9   render () {
10     console.log(this.props)
11
12     // 直接通过props 进行获取 因为就是通过props传入的
13     const { user } = this.props
14
15     return (
16       <div className="App">
17         <header className="App-header">
18           <img src={logo} className="App-logo" alt="logo" />
19           <div>
20             {
21               user.isLoading ?
22                 '欢迎您'
23                 :
24                 '未登录'
25             }
26           </div>
27           <button onClick={ e=> {
28             this.props.dispatch(
29               userAction.login({
30                 account: 'test',
31                 password: '123'
32               })
33             )
34           } }>
35             登录
36           </button>
37         </header>
38       </div>
39     );
40   }
41 }
42
43 // 一个页面可能不需要用到这么多的全局数据
44 // 从所有的仓库数据中拿到所需要的数据
45 // function mapStateToProps (state) {
46 //   return {
47 //     state: state.user
48 //   }
49 // }
50
51 /**
52  * 运行后会生成一个新的方法 把APP传入新方法中
53  */
54 // export default connect(mapStateToProps)(App);
55
56 // 这里的命名决定了在props中的取值名字
57 export default connect((state) => {
58   return {
59     user: state.user
60   }
61 })(App)

```

2.1 使用redux-thunk

```
1 | npm i redux-thunk -S
```

```
1 | npm i redux-devtools-extension -D
```

```

/*...*/
import {createStore, applyMiddleware} from 'redux'
import thunk from 'redux-thunk'
import {composeWithDevTools} from 'redux-devtools-extension'

import reducers from './reducers'

// 向外暴露store对象
export default createStore(reducers, composeWithDevTools(applyMiddleware(thunk)))

```

```

1  /*
2    全局只有一个store
3  */
4  // 1、引用 applyMiddleware, compose
5  import { createStore, applyMiddleware, compose } from 'redux'
6
7  // 2、安装引用 -S
8  // 使用redux-thunk+
9  import thunkMiddleware from 'redux-thunk'
10
11 // 拿到合并的reducers
12 import reducer from '../reducers'
13
14 /**
15  * 通过函数去创建store
16  * @param {*} initState
17  */
18 export default function Store (initStroe) {
19   // 3、注册中间件 拦截actions 的发出
20   // 第三个参数做一些扩展的处理
21   return createStore(
22     reducer,
23     initState,
24     compose(applyMiddleware(thunkMiddleware)),
25     window._REDUX_DEVTOOLS_EXTENSION_ &&
26     window._REDUX_DEVTOOLS_EXTENSION_()
27   )
28 }

```

在 action.js 中使用

```

1  import * as userConstants from '../constants/user'
2
3  /*
4   * actions 就是一条对象信息 带着一些描述
5   action 发完 会给到 reducers里

```

```

6  */
7
8  /**
9   * 运行login 同时把需要的数据传递进去
10  * 然后其会返回一个对象 对象会带着type的信息 作用始位了USER_LOGIN
11  * type 就是 action 的名字
12  * @param {*} payload 发出action带上的参数
13  */
14
15  const loginApi = () => {
16    return new Promise ((resolve) => {
17      setTimeout(() => {
18        resolve({
19          isLoading: true,
20          nickname: 'zhaosi',
21          avatar: 'http://baidu.com'
22        })
23      }, 3000)
24    })
25  }
26
27  export const login = (payload) => {
28
29    // 第四版
30    return async (dispatch) => {
31      dispatch(update({
32        loading: true
33      }))
34
35      const res = await loginApi()
36      console.log('res', res)
37
38      dispatch(update({
39        ...res,
40        loading: false
41      }))
42    }
43
44
45    // 第二版
46    // 因为thunk 拦截了 actions , 会传入一个 dispatch 回来
47    // return (dispatch) => {
48    //   setTimeout(
49    //     dispatch(
50    //       update({
51    //         isLoading: true,
52    //         nickname: 'zhaosi',
53    //         avatar: 'http://baidu.com'
54    //       })
55    //     , 3000)
56    //   )
57
58
59    // 第一版
60    // return {
61    //   type: userConstants.USER_LOGIN,
62    //   payload
63    // }

```

```
64 | }
65 |
66 | export const logout = (payload) => {
67 |   return {
68 |     type: userConstants.USER_LOGOUT,
69 |     payload
70 |   }
71 | }
72 |
73 | export const update = (payload) => {
74 |   console.log('update')
75 |   return {
76 |     type: userConstants.USER_UPDATE,
77 |     payload
78 |   }
79 | }
```

dva

1 安装 dva-cli

通过 npm 安装 dva-cli 并确保版本是 0.9.1 或以上。

```
1 | $ npm install dva-cli -g
2 | $ dva -v
3 | dva-cli version 0.9.1
```

2 #创建新应用

安装完 dva-cli 之后，就可以在命令行里访问到 dva 命令（[不能访问?](#)）。现在，你可以通过 dva new 创建新应用。

```
1 | $ dva new dva-quickstart
```

这会创建 dva-quickstart 目录，包含项目初始化目录和文件，并提供开发服务器、构建脚本、数据 mock 服务、代理服务器等功能。

然后我们 cd 进入 dva-quickstart 目录，并启动开发服务器：

```
1 | $ cd dva-quickstart
2 | $ npm start
```

几秒钟后，你会看到以下输出：

```
1 | Compiled successfully!
2 |
3 | The app is running at:
4 |
5 |   http://localhost:8000/
6 |
7 | Note that the development build is not optimized.
8 | To create a production build, use npm run build.
```

在浏览器里打开 <http://localhost:8000>，你会看到 dva 的欢迎界面。

3 #使用 antd

通过 npm 安装 `antd` 和 `babel-plugin-import`。`babel-plugin-import` 是用来按需加载 antd 的脚本和样式的，详见 [repo](#)。

```
1 $ npm install antd babel-plugin-import --save
```

编辑 `.webpackrc`，使 `babel-plugin-import` 插件生效。

```
1 {  
2   + "extraBabelPlugins": [  
3     + ["import", { "libraryName": "antd", "libraryDirectory": "es", "style":  
4       + "css" }]  
5   + ]  
6 }
```

注：dva-cli 基于 roadhog 实现 build 和 dev，更多 `.webpackrc` 的配置详见 [roadhog#配置](#)

4 #定义路由

我们要写个应用来先显示产品列表。首先第一步是创建路由，路由可以想象成是组成应用的不同页面。

新建 route component `routes/Products.js`，内容如下：

```
1 import React from 'react';  
2  
3 const Products = (props) => (  
4   <h2>List of Products</h2>  
5 );  
6  
7 export default Products;
```

添加路由信息到路由表，编辑 `router.js`：

```
1 + import Products from './routes/Products';  
2 ...  
3 + <Route path="/products" exact component={Products} />
```

然后在浏览器里打开 <http://localhost:8000/#/products>，你应该能看到前面定义的 `<h2>` 标签。

5 #编写 UI Component

随着应用的发展，你会需要在多个页面分享 UI 元素 (或在一个页面使用多次)，在 dva 里你可以把这部分抽成 component。

我们来编写一个 `ProductList` component，这样就能在不同的地方显示产品列表了。

新建 `components/ProductList.js` 文件：

```
1 import React from 'react';  
2 import PropTypes from 'prop-types';
```

```

3 import { Table, Popconfirm, Button } from 'antd';
4
5 const ProductList = ({ onDelete, products }) => {
6   const columns = [{
7     title: 'Name',
8     dataIndex: 'name',
9   }, {
10    title: 'Actions',
11    render: (text, record) => {
12      return (
13        <Popconfirm title="Delete?" onConfirm={() => onDelete(record.id)}>
14          <Button>Delete</Button>
15        </Popconfirm>
16      );
17    },
18  ]};
19   return (
20     <Table
21       dataSource={products}
22       columns={columns}
23     />
24   );
25 };
26
27 ProductList.propTypes = {
28   onDelete: PropTypes.func.isRequired,
29   products: PropTypes.array.isRequired,
30 };
31
32 export default ProductList;

```

6 #定义 Model

完成 UI 后，现在开始处理数据和逻辑。

dva 通过 model 的概念把一个领域的模型管理起来，包含同步更新 state 的 reducers，处理异步逻辑的 effects，订阅数据源的 subscriptions。

新建 model `models/products.js`：

```

1 export default {
2   namespace: 'products',
3   state: [],
4   reducers: {
5     'delete'(state, { payload: id }) {
6       return state.filter(item => item.id !== id);
7     },
8   },
9 };

```

这个 model 里：

- `namespace` 表示在全局 state 上的 key
- `state` 是初始值，在这里是空数组
- `reducers` 等同于 redux 里的 reducer，接收 action，同步更新 state

然后别忘记在 `index.js` 里载入他：

```
1 // 3. Model
2 + app.model(require('./models/products').default);
```

7 #connect 起来

到这里，我们已经单独完成了 model 和 component，那么他们如何串联起来呢？

dva 提供了 connect 方法。如果你熟悉 redux，这个 connect 就是 react-redux 的 connect。

编辑 `routes/Products.js`，替换为以下内容：

```
1 import React from 'react';
2 import { connect } from 'dva';
3 import ProductList from '../components/ProductList';
4
5 const Products = ({ dispatch, products }) => {
6   function handleDelete(id) {
7     dispatch({
8       type: 'products/delete',
9       payload: id,
10    });
11  }
12  return (
13    <div>
14      <h2>List of Products</h2>
15      <ProductList onDelete={handleDelete} products={products} />
16    </div>
17  );
18 };
19
20 // export default Products;
21 export default connect(({ products }) => ({
22   products,
23 }))(Products);
```

最后，我们还需要一些初始数据让这个应用 run 起来。编辑 `index.js`：

```
1 - const app = dva();
2 + const app = dva({
3 +   initialState: {
4 +     products: [
5 +       { name: 'dva', id: 1 },
6 +       { name: 'antd', id: 2 },
7 +     ],
8 +   },
9 + });
```

刷新浏览器，应该能看到以下效果：

List of Products

Name	Actions
dva	<button>删除</button>
antd	<button>删除</button>

<1>

8 #构建应用

完成开发并且在开发环境验证之后，就需要部署给我们的用户了。先执行下面的命令：

```
1 | $ npm run build
```

几秒后，输出应该如下：

```
1 | > @ build /private/tmp/myapp
2 | > roadhog build
3 |
4 | Creating an optimized production build...
5 | Compiled successfully.
6 |
7 | File sizes after gzip:
8 |
9 |   82.98 KB  dist/index.js
10 |   270 B    dist/index.css
```

`build` 命令会打包所有的资源，包含 JavaScript, CSS, web fonts, images, html 等。然后你可以在 `dist/` 目录下找到这些文件。