Vue 学习

Vue-Router

首先安装router npm install vue-router --save

创建一个router文件夹 在文件夹下面创建一个index.js来编写router 代码如下

这是router 的一个基本配置

import Vue from 'vue'

import Router from 'vue-router'

import HelloWorld from '@/components/HelloWorld'

import Film from '@/views/Film'

import Center from '@/views/Center'

import Cinama from '@/views/Cinama'

Vue.use(Router)

export default new Router({

  routes: [

    {

      path: '/film',

      name: 'Film',

      component: Film

    },

    {

      path: '/center',

      name: 'Center',

      component: Center

    },

    {

      path: '/cinama',

      name: 'cinama',

      component: Cinama

    }

  ]

})

在main.js 中引入router

// The Vue build version to load with the `import` command

// (runtime-only or standalone) has been set in webpack.base.conf with an alias.

import Vue from 'vue'

import App from './App'

import router from './router'

console.log(router)

Vue.config.productionTip = false

var vm = new Vue({

  el: '#app',

  router,

  components: { App },

  template: '<App id="5354"/>'

})

console.log(vm)

在所需要使用路由的页面中要使用 <router-view></router-view>当做路由插槽

动态路由

使用this.$router.push(‘path’) 添加一个路径 进行跳转

在接受页面用this.$route.params.myId 进行参数接收

代码示例:

添加详情路由

 {

      path: '/detail/:id',

      name: 'Detail',

      component: Detail

    },

<template>

<div>

NoPlaying

<ul>

<li v-for="data in dataList" @click="handleClick(data)">

{{data}}

</li>

</ul>

</div>

</template>

<script>

export default {

name: 'NoPlaying',

data() {

return {

dataList: [11111,22222,33333],

isActive: true,

classObj:{

a: true,

b: true

}

}

},

methods: {

handleClick(id){

console.log(id)

// console.log(this.$router.push)

this.$router.push(`/detail/${id}`)

}

}

}

</script>

<template>

<div>

Detail

</div>

</template>

<script>

export default {

data() {

return {

}

},

mounted(){

console.log(this.$route.params.id)

},

methods: {

}

}

</script

命名路由:

this.$router.push({name:'Detail',params: {id}}) 通过这种方式跳转

路由配置 mode 有两种模式 history 去掉路径中的#号 后端需要配置好路由 可能出现路由找不到的情况

Hash 模式 路径中带有# 号 后端不需要做任何配置

路由拦截

全局拦截

router.beforeEach((to, from, next) => {

  console.log(to)

  if(to.path === '/center'){

    console.log("拦截");

  }else {

    next()

  }

})

局部拦截

beforeRouteEnter (to, from, next) {

console.log("盘查", to);

// 在渲染该组件的对应路由被 confirm 前调用

// 不！能！获取组件实例 `this`

// 因为当守卫执行前，组件实例还没被创建

},

状态管理



首先安转 vuex npm install vuex --save

床架 store.js 创建一个修改mutation 文件来对mutataion 的操作类型进行统一的管理

其中action 处理异步操作代码 但是最终操作state 要经过 mutation 来进行处理

Getters 相当于计算属性

为了方便使用 可以引入 mapState 辅助函数来获取state 的值

mapAction 来操作action mapGetters 来访问getters 函数

Store.js 代码

import Vue from 'vue'

import Vuex from 'vuex'

import axios from 'axios'

import {SHOW\_BAR\_OPERATOR, SET\_DATA\_OPERATOR} from '@/type'

Vue.use(Vuex)

export default new Vuex.Store({

    state: {

        isTabarShow: false,

        maoyanList: []

    },

    getters: {

       getMaoYanList(state) {

            return state.maoyanList.filter((item,index) => index < 3);

       }

    },

    mutations: {

        [SHOW\_BAR\_OPERATOR](state, data) {

            console.log("showTabar");

            state.isTabarShow = data;

        },

        [SET\_DATA\_OPERATOR] (state, data){

            console.log("changeData")

            state.maoyanList = data

            console.log(data)

        }

    },

    actions: {

       getData(store){

        axios.get("/ajax/movieOnInfoList?token=").then(res => {

            console.log(res)

            console.log("监听了")

            store.commit(SET\_DATA\_OPERATOR, res.data.movieList)

          })

       }

    }

})

操作类型文件代码

export const SHOW\_BAR\_OPERATOR = 'showBar'

export const SET\_DATA\_OPERATOR = 'setData'

使用样列

<template>

<div id="app">

<ul>

<router-link to="/film" tag="li" activeClass="myActive">film</router-link>

<router-link to="/center" tag="li" activeClass="myActive">center</router-link>

<router-link to="/cinama" tag="li" activeClass="myActive">cinama</router-link>

</ul>

<div>

<router-view></router-view>

</div>

<div v-show="isTabarShow">

isTabarShow

</div>

</div>

</template>

<script>

import Study from './components/Study'

import NoParent from './components/NoParent'

import axios from 'axios'

import { mapState } from 'vuex'

import {SHOW\_BAR\_OPERATOR, SET\_DATA\_OPERATOR} from '@/type'

export default {

name: 'App',

props:{

id: String

},

components: {

},

data() {

return {

}

},

mounted() {

axios.get("/ajax/movieOnInfoList?token=").then(res => {

console.log(res)

})

this.$store.dispatch("getData");

console.log(SHOW\_BAR\_OPERATOR)

console.log(this.$store)

this.$store.commit(SHOW\_BAR\_OPERATOR, true);

console.log(this.$store.state.maoyanList)

console.log("getters", this.$store.getters.getMaoYanList)

},

methods: {

},

computed: {

...mapState(['isTabarShow'])

}

}

</script>

<style>

#app {

font-family: 'Avenir', Helvetica, Arial, sans-serif;

-webkit-font-smoothing: antialiased;

-moz-osx-font-smoothing: grayscale;

text-align: center;

color: #2c3e50;

margin-top: 60px;

}

.myActive{

color: red;

}

.foote-enter-active, .foote-leave-active{

transition: all 1.5s;

}

.foote-enter, .foote-leave-to{

opacity: 0;

transform: translateX(100px)

}

.footbounce-enter-active {

animation: bounce-in .5s

}

.footbounce-leave-active {

animation: bounce-in .5s reverse

}

@keyframes bounce-in {

0% {

opacity: 0;

transform: translateX(100px)

}

100% {

opacity: 1;

transform: translateX(0px)

}

}

</style>

然后使用

<ul>

<router-link to="/film" tag="li" activeClass="myActive">film</router-link>

<router-link to="/center" tag="li" activeClass="myActive">center</router-link>

<router-link to="/cinama" tag="li" activeClass="myActive">cinama</router-link>

</ul>

链接到所需要跳转的路由 tag标签指定要渲染成的标签 activeClass 设置点击的时候高亮显示的样式

Vue的反向代理配置 解决跨域获取数据问题

在config下的index.js 文件中的proxytable 对象下添加:

 proxyTable: {

          '/ajax': {

            target: 'http://m.maoyan.com',

          //  ws: true,

            changeOrigin: true

          },

    },

如这个例子 :

当访问的路径中带有ajax 关键字时，会自动代理到本路径下去

Vue 生命周期

· beforeCreate()方法是在实例还没有别完全创建出来之前执行，这个方法执行的时候，无论是data中的数据还是methods中的方法都没有被初始化

· created()方法是此时已经初始化好了Vue对象，这是最早能操作data中数据和methods中方法的地方

· beforeMount()这是指该模板已经在内存中编辑完成，但是尚未将其渲染到页面中去，在该方法执行期间，页面中的元素还没有经过Vue的渲染，此时的{{msg}}仍然是一个“{{msg}}”

· mounted()，表示内存中的模板已经渲染到页面中去，这个时候数据已经可以经过Vue的编译，该函数是实例创建的最后一个函数，此时实例已经完全创建好了，如果没有别的操作的话，此函数就静静躺在内存中，如果要使用某些插件操作页上的DOM，最早要在mount中操作

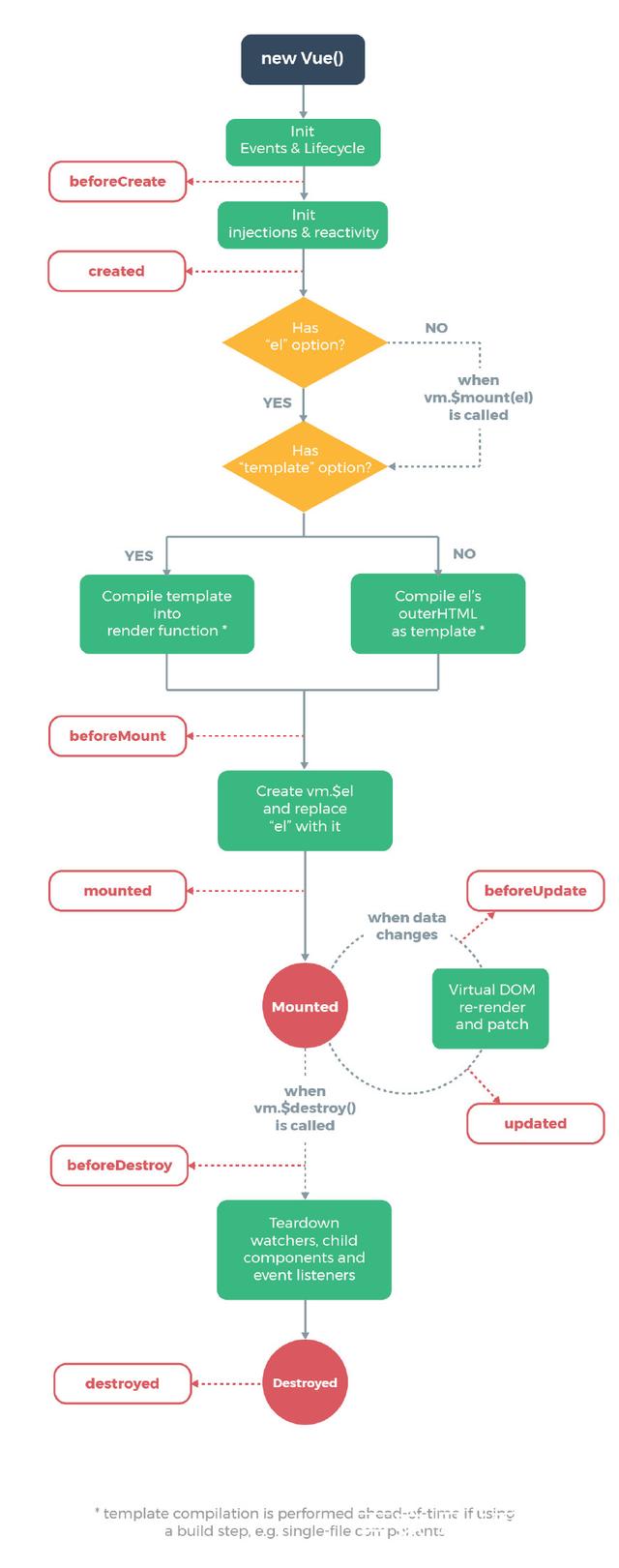
函数就静静躺在内存中，如果要使用某些插件操作页面上的DOM，最早要在mount中操作

这两个事件，会根据data数据的改变选择性的触发0次或者多次

beforeUpdate()事件执行的时候，页面中显示的数据并没有更新，但是内存中的数据已经改变，也就是说此时页面尚未和最新的数据保持同步

update()事件执行的时候，页面中的数据已经和内存中保持一致了，也就是说此时已经是最新的了

### 销毁阶段的生命周期



<html>

    <head>

    </head>

    <script type="text/javascript" src="https://cdn.jsdelivr.net/vue/2.1.3/vue.js"></script>

    <body>

      <div id="app">

          <div v-if="isShow">

                <render></render>

            </div>

           <button @click="isShow = !isShow">改变数据状态</button>

           <div>

            <div v-for="(item, index) in list" :key="index">{{item}}</div>

          </div>

      </div>

       <script>

         var vm = new Vue({

             el: '#app',

             data: {

               message: '',

               list: [1, 1, 1],

               isShow: true

             },

             components:{

               render: {

                   template: '<div>54353453453543</div>',

                   beforeDestroy: function () {

            console.group('beforeDestroy 销毁前状态===============》');

            console.log("%c%s", "color:red","el     : " + this.$el);

            console.log(this.$el);

               console.log("%c%s", "color:red","data   : " + this.$data);

               console.log("%c%s", "color:red","message: " + this.message);

        },

        destroyed: function () {

            console.group('destroyed 销毁完成状态===============》');

            console.log("%c%s", "color:red","el     : " + this.$el);

            console.log(this.$el);

               console.log("%c%s", "color:red","data   : " + this.$data);

               console.log("%c%s", "color:red","message: " + this.message)

        }

               }

             },

       methods: {

          changeState() {

              console.log("改变了数据")

              this.message = "324234"

          }

       },

       beforeCreate: function () {

                console.group('beforeCreate 创建前状态===============》');

               console.log("%c%s", "color:red" , "el     : " + this.$el); //undefined

               console.log("%c%s", "color:red","data   : " + this.$data); //undefined

               console.log("%c%s", "color:red","message: " + this.message)

        },

        created: function () {

            console.group('created 创建完毕状态===============》');

            console.log("%c%s", "color:red","el     : " + this.$el); //undefined

               console.log("%c%s", "color:red","data   : " + this.$data); //已被初始化

               console.log("%c%s", "color:red","message: " + this.message); //已被初始化

               setTimeout(\_ => {

            this.list = [2, 2, 2]

        }, 1000)

        },

        beforeMount: function () {

            console.group('beforeMount 挂载前状态===============》');

            console.log("%c%s", "color:red","el     : " + (this.$el)); //已被初始化

            console.log(this.$el);

               console.log("%c%s", "color:red","data   : " + this.$data); //已被初始化

               console.log("%c%s", "color:red","message: " + this.message); //已被初始化

        },

        // 新的dom 替换掉旧的dom

        mounted: function () {

            console.group('mounted 挂载结束状态===============》');

            console.log("%c%s", "color:red","el     : " + this.$el); //已被初始化

            console.log(this.$el);

               console.log("%c%s", "color:red","data   : " + this.$data); //已被初始化

               console.log("%c%s", "color:red","message: " + this.message); //已被初始化

        },

        beforeUpdate: function () {

            console.group('beforeUpdate 更新前状态===============》');

            console.log("%c%s", "color:red","el     : " + this.$el);

            console.log(this.$el);

               console.log("%c%s", "color:red","data   : " + this.$data);

               console.log("%c%s", "color:red","message: " + this.message);

        },

        updated: function () {

            console.group('updated 更新完成状态===============》');

            console.log("%c%s", "color:red","el     : " + this.$el);

            console.log(this.$el);

               console.log("%c%s", "color:red","data   : " + this.$data);

               console.log("%c%s", "color:red","message: " + this.message);

        },

        beforeDestroy: function () {

            console.group('beforeDestroy 销毁前状态===============》');

            console.log("%c%s", "color:red","el     : " + this.$el);

            console.log(this.$el);

               console.log("%c%s", "color:red","data   : " + this.$data);

               console.log("%c%s", "color:red","message: " + this.message);

        },

        destroyed: function () {

            console.group('destroyed 销毁完成状态===============》');

            console.log("%c%s", "color:red","el     : " + this.$el);

            console.log(this.$el);

               console.log("%c%s", "color:red","data   : " + this.$data);

               console.log("%c%s", "color:red","message: " + this.message)

        }

    })

      </script>

    </body>

</html>

1.class与style绑定比较特殊 有三目运算符绑定 数组形式 对象形式均可绑定 每个属性都有其

get 和set 方法

2.事件修饰符

@click.stop 阻止冒泡行为 .prevent 阻止默认行为 .slef 事件只在自己身上执行 .once

事件执行一次之后就解绑

3. vue父子间的通信

1.1 通过this.$emit() 从子组件向父组件传递信息后者调用父组件的方法

1.2 ref 定义在标签上代表的是当前绑定的标签 绑定在子组件上表示的是当前子组件

可以通过this.$refs.childName 来向子组件通信或者调用子组件的方法

4. 非父子间通信

建立一个bus 事件总线 import Vue from 'vue'

export default new Vue({})

在需要注册事件的地方用bus.$emit()注册事件 在需要监听的地方引入bus 并在mounted中用

<template>

<div class="container">

<input type="text" value="232" ref="myInput"/>

<button @click="handleClick()">发布</button>

</div>

</template>

<script>

import bus from '../bus'

export default {

name: 'hello',

props: {

message: String

},

mounted(){

},

data() {

return {

}

},

methods:{

handleClick(){

console.log(this.$refs.myInput.value)

bus.$emit('change', this.$refs.myInput.value)

}

}

}

</script>

<style scoped>

</style>

this.$on() 监听事件

<template>

<div class="container">

<button @click="changeReverse">改变父组件状态</button>

</div>

</template>

<script>

import bus from '../bus'

export default {

name: 'hello',

props: {

message: String

},

mounted(){

bus.$on('change', (ev) => {

console.log(ev)

console.log('接受到了非父子通信',ev)

})

},

data() {

return {

isActive: true,

classObj:{

a: true,

b: true

},

classArr: ['f','dd'],

styleObj: {

background: 'red',

},

styleArr: [{

background: 'red',

}]

}

},

methods:{

changeReverse(){

console.log("开始通知父组件")

this.$emit("reverse");

}

}

}

</script>

<style scoped>

</style>

5. 动态组件

component 标签上的is 属性 指向组件 当更换展示组件时 is 属性会替换内部组件

可以用 keep-alive 标签包裹着组件 当更换组件的时候会保持上一次组件初始状态

6. transition 动画的使用

name 属性绑定动画名字

然后直接定义动画 .foote-enter-active, .foote-leave-active{

transition: all 1.5s;

}

.foote-enter, .foote-leave-to{

opacity: 0;

transform: translateX(100px)

}

也可以定义动画帧 的方式来定义

.footbounce-enter-active {

animation: bounce-in .5s

}

.footbounce-leave-active {

animation: bounce-in .5s reverse

}

@keyframes bounce-in {

0% {

opacity: 0;

transform: translateX(100px)

}

100% {

opacity: 1;

transform: translateX(0px)

}

}

多元素过渡

加上一个key保证元素渲染的时候重新渲染 mode模式 out-in 先出后进 in-out 先进后出

列表渲染用 <transition-group>标签 name 属性绑定定义的动画 tag 标签指定要渲染成的

标签类型 在每个li 上加上一个key 属性 不能是index 索引值

1. Vue.component('button-counter', {

data: function () {

return {

count: 0

}

},

template: '<button v-on:click="count++">You clicked me {{ count }} times.</button>'

})

注册全局组件 在此组件注册后的任何地方都可以引用它

局部注册组件 new Vue({

el: '#app',

components: {

'component-a': ComponentA,

'component-b': ComponentB

}

})

components 定义的是一个对象 以键值对的形式注册组件

2.props 如果不检查类型 就直接用数组接收 如果要检查类型就直接用对象的形式接收

example:

props: ['title', 'likes', 'isPublished', 'commentIds', 'author']。

props: {

title: String,

likes: Number,

isPublished: Boolean,

commentIds: Array,

author: Object,

callback: Function,

contactsPromise: Promise // or any other constructor

}

动态props 可以用 v-bind:绑定属性传 其接收的时候和静态属性同种方式接收即可

props验证 包括设置默认值等Vue.component('my-component', {

props: {

// 基础的类型检查 (`null` 和 `undefined` 会通过任何类型验证)

propA: Number,

// 多个可能的类型

propB: [String, Number],

// 必填的字符串

propC: {

type: String,

required: true

},

// 带有默认值的数字

propD: {

type: Number,

default: 100

},

// 带有默认值的对象

propE: {

type: Object,

// 对象或数组默认值必须从一个工厂函数获取

default: function () {

return { message: 'hello' }

}

},

// 自定义验证函数

propF: {

validator: function (value) {

// 这个值必须匹配下列字符串中的一个

return ['success', 'warning', 'danger'].indexOf(value) !== -1

}

}

}

})

3.自定义事件

不同于组件和 prop，事件名不会被用作一个 JavaScript 变量名或属性名，所以就没有理由使用

camelCase 或 PascalCase 了。

并且 v-on 事件监听器在 DOM 模板中会被自动转换为全小写 (因为 HTML 是大小写不敏感的)，

所以 v-on:myEvent 将会变成 v-on:myevent——导致 myEvent 不可能被监听到。

4.slot 动态填充内容 感觉很像react 的 this.props.children 效果一样

<navigation-link url="/profile">

Your Profile

</navigation-link>

然后你在 <navigation-link> 的模板中可能会写为：

<a

v-bind:href="url"

class="nav-link"

>

<slot>后背内容</slot>

</a>

插槽里面可以嵌入组件 标签等内容 可以在插槽里放入后备内容 当组件里面没有放入内容时

就可以直接渲染后备内容

一个不带 name 的 <slot> 出口会带有隐含的名字“default”。

React 状态管理实现原理

<html>

    <head>

    </head>

    <body>

      <div>

          <button onclick="store.dispatch({

            type: 'JIAN',

            n: 2

          })">-</button>

          <span id="countDisplay">10</span>

          <button onclick="store.dispatch({

            type: 'JIA',

            n: 2

          })">+</button>

      </div>

       <script>

         const countDisplay = document.querySelector('#countDisplay');

         const countState = {

           count: 5

         }

         const reducer = (state = countState, action) => {

          if(!state) {

              return countState

            }

            switch(action.type){

              case 'JIAN' : return {

                ...state,

                count: state.count - action.n

              }

              case 'JIA':return {

                ...state,

                count: state.count + action.n

              }

              default:

                 return state

                break;

            }

          }

          const createStore = (reducer) => {

            let state = null

            const getState =  () =>{

              return state

            }

            const listeners = [];

            const subcribe = (listener) => {

              listeners.push(listener);

            }

            const dispatch = (action) =>{

               state = reducer(state,action)

               listeners.forEach((listener) => listener())

          }

            return {

              getState,

              dispatch,

              subcribe

            }

          }

          const store = createStore(reducer);

          const renderCount = () =>{

             countDisplay.innerHTML = store.getState().count;

          }

          store.subcribe(renderCount);

          // react 状态管理    个人感觉是最难的部分  在这里记一下笔记

          // 1.新建reduce   这个是用来更改状态的  参数有（state, action）其中state就是状态值  action 就是动作

          // 类型  一般是函数  返回一个对象值   reduce 更改状态值就是根据action.type 来判断是否更改

          // 2 store = createStore 这里要用combineReduces 将reduce封装  然后导出即可

          //然后store.getState()返回的值就是这个reduce

          // 3. 创建action 一般为函数  定义好动作类型  返回一个对象  返回的对象包括type: 字段  reduce更改state值就根据type

          // 字端的值来返回

          // 4.在需要用的地方先this.setState(store.getState())  返回一次状态

          // 然后store.subcribe(this.setState(store.getState())) 订阅  然后每次调用action就可以了

          // 5.这里可以用react-redux  来实现不用每层都传store  只要定义mapState =(state) => {

          //  return {

          //    cartList: state.cart

          // }

          //  }

          //

      </script>

    </body>

</html>

npm install --save react-redux

React-Redux 提供<Provider/>组件，能够使你的整个app访问到Redux store中的数据：

**import** React **from** "react";**import** ReactDOM **from** "react-dom";

**import** { Provider } **from** "react-redux";**import** store **from** "./store";

**import** App **from** "./App";

**const** rootElement = document.getElementById("root");

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

rootElement

);

React-Redux提供一个connect方法能够让你把组件和store连接起来。

通常你可以以下面这种方式调用connect方法：

**import** { connect } **from** "react-redux";**import** { increment, decrement, reset } **from** "./actionCreators";

*// const Counter = ...*

**const** mapStateToProps = (state /\*, ownProps\*/) => {

**return** {

counter: state.counter

};

};

**const** mapDispatchToProps = { increment, decrement, reset };

**export** **default** connect(

mapStateToProps,

mapDispatchToProps

)(Counter);

例子:

建立cartList

import React, {Component} from 'react';

import {connect}  from 'react-redux'

import {increment, decrement,decrementAsync} from '../../stores/action/cart'

class CartList extends Component {

    render() {

        return(

            <table>

                <thead>

                    <tr>

                    <th>标签</th>

                    <th>名称</th>

                    <th>单价</th>

                    <th>数量</th>

                    <th>操作</th>

                    </tr>

                </thead>

                <tbody>

                    {

                        this.props.cartList.map((item) =>{

                            return(

                             <tr key={item.id}>

                               <td>{item.id}</td>

                               <td>{item.title}</td>

                               <td>{item.price}</td>

                               <td>

                                <button onClick={this.props.decrementAsync.bind(this,item.id)}>过一会再减</button>

                                <button onClick={this.props.decrement.bind(this,item.id)}>-</button>

                               {item.amount}

                               <button onClick={this.props.increment.bind(this,item.id)}>+</button>

                               </td>

                            </tr>

                           )

                       })

                    }

                </tbody>

            </table>

        )

    }

}

const mapState = (state) => {

    return {

        cartList: state.cart

    }

}

const mapDispatchToProps = dispacth => {

    return {

        dispatchIncrement: (id) => dispacth(increment(id)),

        dispatchDecrement: (id) => dispacth(decrement(id))

    }

}

export default connect(mapState,{increment, decrement,decrementAsync})(CartList)

建立 action

import actionType from './actionType'

export const increment = (id) =>{

     return {

         type: actionType.CART\_AMOUNT\_INCREMENT,

         payload: {

             id

         }

     }

}

export const decrement = (id) =>{

    return {

        type: actionType.CART\_AMOUNT\_DECREMENT,

        payload: {

            id

        }

    }

}

export const decrementAsync4 = id => dispatch => {

    setTimeout(()=>{

       dispatch(decrement(id));

    },200)

}

export const decrementAsync = (id) =>{

    return (dispatch) => {

        setTimeout(()=>{

            dispatch(decrement(id));

         },200)

    }

}

建立reduce

import actionType from '../action/actionType'

const initState = [{

        id: 1,

        title: 'apple',

        price: 8888,

        amount: 10

    },{

        id: 2,

        title: 'apples',

        price: 6666,

        amount: 12

    }]

export default (state = initState, action) => {

    switch(action.type){

        case actionType.cart.CART\_AMOUNT\_INCREMENT:

        return state.map(item => {

            if(item.id === action.payload.id){

                item.amount += 1;

            }

            return item;

        })

        case actionType.cart.CART\_AMOUNT\_DECREMENT:

        return state.map(item => {

            if(item.id === action.payload.id){

                item.amount -= 1;

            }

            return item;

        })

        default: return state

    }

}

创建store 这里的store 用到了异步action

import {createStore} from 'redux'

import rootReducer from '../reduces'

import {applyMiddleware} from 'redux'

import  thunk from 'redux-thunk'

export default createStore(rootReducer,applyMiddleware(thunk))

**树组件实现**

安装ant-design

npm install antd --save

## 高级配置[#](https://ant.design/docs/react/use-with-create-react-app-cn" \l "%E9%AB%98%E7%BA%A7%E9%85%8D%E7%BD%AE)

这个例子在实际开发中还有一些优化的空间，比如无法进行主题配置，而且上面的例子加载了全部的 antd 组件的样式（gzipped 后一共大约 60kb）。

此时我们需要对 create-react-app 的默认配置进行自定义，这里我们使用 [react-app-rewired](https://github.com/timarney/react-app-rewired) （一个对 create-react-app 进行自定义配置的社区解决方案）。

引入 react-app-rewired 并修改 package.json 里的启动配置。由于新的 [react-app-rewired@2.x](https://github.com/timarney/react-app-rewired" \l "alternatives) 版本的关系，你还需要安装 [customize-cra](https://github.com/arackaf/customize-cra)。

$ yarn add react-app-rewired customize-cra

/\* package.json \*/

"scripts": {- "start": "react-scripts start",+ "start": "react-app-rewired start",- "build": "react-scripts build",+ "build": "react-app-rewired build",- "test": "react-scripts test",+ "test": "react-app-rewired test",

}

然后在项目根目录创建一个 config-overrides.js 用于修改默认配置。

module.exports = function override(config, env) {

// do stuff with the webpack config...

return config;};

### 使用 babel-plugin-import[#](https://ant.design/docs/react/use-with-create-react-app-cn" \l "%E4%BD%BF%E7%94%A8-babel-plugin-import)

注意：antd 默认支持基于 ES module 的 tree shaking，js 代码部分不使用这个插件也会有按需加载的效果。

[babel-plugin-import](https://github.com/ant-design/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](https://ant.design/docs/react/getting-started-cn" \l "%E6%8C%89%E9%9C%80%E5%8A%A0%E8%BD%BD)），现在我们尝试安装它并修改 config-overrides.js文件。

$ yarn add babel-plugin-import

+ const { override, fixBabelImports } = require('customize-cra');

- module.exports = function override(config, env) {- // do stuff with the webpack config...- return config;- };+ module.exports = override(+ fixBabelImports('import', {+ libraryName: 'antd',+ libraryDirectory: 'es',+ style: 'css',+ }),+ );

import './App.css';

import React from 'react'

import { Button } from 'antd';

import { TreeSelect } from 'antd';

import treeData from './data/TreeNodeData'

import dataList from './data/anyouData'

const { SHOW\_PARENT } = TreeSelect;

 class App extends React.Component {

    state = {

        value: ['0-0-0'],

      };

      onChange = value => {

        console.log('onChange ', value);

        this.setState({ value });

      };

      onSearch = value => {

          console.log("onSearch",value)

      }

  render() {

    console.log(dataList)

    const tProps = {

        treeData,

        value: this.state.value,

        onChange: this.onChange,

        treeCheckable: true,

        showCheckedStrategy: SHOW\_PARENT,

        searchPlaceholder: 'Please select',

        treeNodeFilterProp: "value",

        onSearch: this.onSearch,

        allowClear: true,

        multiple: true,

        style: {

          width: '20%',

        },

      };

      return (

          <TreeSelect {...tProps} />

        )

  }

}

export default App;