**登录和注册**

**1、注册**

手机号、验证码、登录密码、确认密码、同意用户协议、点击注册

（1）验证码

获取手机验证码

v-model绑定手机号输入框，点击发送验证码按钮，向服务器发送请求获取验证码：

封装获取验证码的请求：



正常情况下，发送请求后，后台服务器自动将验证码发送到用户对应的手机上，但此处直接将验证码返回，需要我们自己commit修改：

|  |
| --- |
| import { reqGetCode } from '@/api'  // 登录、注册模块  const state = {    code: '',  }  const mutations = {    GETCODE(state, code) {      state.code = code    },  }  const actions = {    // 获取验证码    async getCode({ commit }, phone) {      let res = await reqGetCode(phone)      if (res.code === 200) {        commit('GETCODE', res.data)        return 'ok'      } else {        return Promise.reject(new Error('failed'))      }    },  }  const getters = {}  export default {    state,    mutations,    actions,    getters,  } |

点击发送验证码按钮的回调函数：

|  |
| --- |
| async getCode() {        try {          const { phone } = this          phone && (await this.$store.dispatch('getCode', phone))          this.code = this.$store.state.user.code        } catch (error) {          console.log(error.message)        }      }, |

（2）设置密码

v-model绑定密码

（3）点击注册按钮，发送请求进行注册

封装请求函数

|  |
| --- |
| // 用户注册  export function reqRegisterUser(data) {    return requests({      url: '/user/passport/register',      data: data,      method: 'post',    })  } |

store注册实现：

|  |
| --- |
| import { reqGetCode, reqRegisterUser } from '@/api'  // 登录、注册模块  const state = {    code: '',  }  const mutations = {    GETCODE(state, code) {      state.code = code    },  }  const actions = {    // 获取验证码    async getCode({ commit }, phone) {      let res = await reqGetCode(phone)      if (res.code === 200) {        commit('GETCODE', res.data)        return 'ok'      } else {        return Promise.reject(new Error('failed'))      }    },    // 用户注册    async userRegister({ commit }, user) {      let res = await reqRegisterUser(user)      if (res.code === 200) {        return 'ok'      } else {        Promise.reject(new Error('failed'))      }    },  }  const getters = {}  export default {    state,    mutations,    actions,    getters,  } |

点击注册按钮派发actions进行用户注册，注册成功则跳转到登陆页面：

|  |
| --- |
| // 用户注册      async userRegister() {        try {          const { phone, code, password, passwordCheck } = this          if (phone && code && password === passwordCheck) {            await this.$store.dispatch('userRegister', { phone, code, password })          }          // 注册成功则跳转到登陆页面进行登录          this.$router.push('/login')        } catch (error) {          alert(error.message)        }      }, |

**2、登录（token）**

用户登录请求接口封装：

|  |
| --- |
| // 用户登录  export function reqUserLogin(data) {    return requests({      url: '/user/passport/login',      method: 'post',      data,    })  } |

登录之前要进行注册，通过数据库存储用户信息，登陆成功为了区分用户，服务器下发了一个token（令牌），是每个用户唯一的身份标识。

仓库store：

|  |
| --- |
| import { reqGetCode, reqRegisterUser, reqUserLogin } from '@/api'  // 登录、注册模块  const state = {    code: '',    token: '',  }  const mutations = {    GETCODE(state, code) {      state.code = code    },    USERLOGIN(state, token) {      state.token = token    },  }  const actions = {    // 获取验证码    async getCode({ commit }, phone) {      let res = await reqGetCode(phone)      if (res.code === 200) {        commit('GETCODE', res.data)        return 'ok'      } else {        return Promise.reject(new Error('failed'))      }    },    // 用户注册    async userRegister({ commit }, user) {      let res = await reqRegisterUser(user)      if (res.code === 200) {        return 'ok'      } else {        Promise.reject(new Error('failed'))      }    },    // 用户登录    async userLogin({ commit }, user) {      let res = await reqUserLogin(user)      // 请求成功，服务器下发token，我们要进行本地存储      if (res.code === 200) {        commit('USERLOGIN', res.data.token)        return 'ok'      } else {        Promise.reject(new Error('failed'))      }    },  }  const getters = {}  export default {    state,    mutations,    actions,    getters,  } |

点击登录按钮进行登录，登录成功跳转到首页：

|  |
| --- |
| async login() {        try {          const { phone, password } = this          if (phone && password) {            await this.$store.dispatch('userLogin', { phone, password })          }          // 登录成功跳转到首页          this.$router.push('/home')        } catch (error) {          alert(error.message)        }      }, |

这里需要注意：登录成功后，是将token存到了vuex仓库中，但是vuex仓库存储数据不是持久化的，页面一刷新数据就丢失了。

**跳转到首页后，实现头部显示：用户名和密码**

通过获取的token向服务器获取用户的基本信息：

（1）封装请求api

|  |
| --- |
| // 获取用户的信息：通过token获取，通过请求头  export function reqGetUserInfo() {    return requests({      url: '/user/passport/auth/getUserInfo',      method: 'get',    })  } |

在请求拦截器中，判断是否存在token，若有则通过请求头的方式发送给服务器以便验证：

|  |
| --- |
| // 请求拦截器：在发请求之前，请求拦截器可以检测到并在请求发送之前处理一些业务  requests.interceptors.request.use((config) => {    // 在请求拦截器中判断游客身份信息    if (store.state.detail.uuid\_token) {      // 给请求头添加字段：和后台商量好了      config.headers.userTempId = store.state.detail.uuid\_token    }    // 判断需要携带token发给服务器    if (store.state.user.token) {      config.headers.token = store.state.user.token    }    // 进度条开始    nprogress.start()    // config：配置对象，有一个属性headers请求头很重要    return config  }) |

仓库三连环：

|  |
| --- |
| const actions = {    // 获取验证码    async getCode({ commit }, phone) {      let res = await reqGetCode(phone)      if (res.code === 200) {        commit('GETCODE', res.data)        return 'ok'      } else {        return Promise.reject(new Error('failed'))      }    },    // 用户注册    async userRegister({ commit }, user) {      let res = await reqRegisterUser(user)      if (res.code === 200) {        return 'ok'      } else {        Promise.reject(new Error('failed'))      }    },    // 用户登录    async userLogin({ commit }, user) {      let res = await reqUserLogin(user)      // 请求成功，服务器下发token，我们要进行本地存储      if (res.code === 200) {        commit('USERLOGIN', res.data.token)        return 'ok'      } else {        Promise.reject(new Error('failed'))      }    },    // 获取用户信息    async getUserInfo({ commit }) {      let res = await reqGetUserInfo()      console.log('用户信息', res.data)    },  } |

跳转到首页后，在首页的mounted钩子函数中请求用户信息：

|  |
| --- |
| mounted() {      // 派发actions，获取floor组件的数据      this.$store.dispatch('getFloorList')      // 获取用户信息在首页展示      this.$store.dispatch('getUserInfo')    }, |

仓库中的数据不是持久存储，一刷新就消失。一刷新首页就获取不到用户信息了。实现持久化存储token。

封装函数实现本地存储：

|  |
| --- |
| // 用户信息的token  export const setToken = (token) => {    localStorage.setItem('TOKEN', token)  }  export const getToken = () => {    return localStorage.getItem('TOKEN')  } |

用户登录的actions中将其存储在本地：

|  |
| --- |
| // 用户登录    async userLogin({ commit }, user) {      let res = await reqUserLogin(user)      // 请求成功，服务器下发token，我们要进行本地存储      if (res.code === 200) {        commit('USERLOGIN', res.data.token)        // 持久化存储token        // localStorage.setItem('TOKEN', res.data.token)        setToken(res.data.token)        return 'ok'      } else {        return Promise.reject(new Error('failed'))      }    }, |

将仓库token的默认值设置为本地存储的token值：

|  |
| --- |
| const state = {    code: '',    // token: localStorage.getItem('TOKEN'),    token: getToken(),    user: {},  } |

退出登录实现：

|  |
| --- |
| // 用户信息的token  export const setToken = (token) => {    localStorage.setItem('TOKEN', token)  }  export const getToken = () => {    return localStorage.getItem('TOKEN')  }  export const removeToken = () => {    localStorage.removeItem('TOKEN')  } |

actions：

|  |
| --- |
| // 退出登录    async userLogout({ commit }) {      // 通知服务器清除token      let res = await reqLogoutUser()      if (res.code === 200) {        // 提交mutations去修改state中的数据        commit('CLEARTOKEN')        return 'ok'      } else {        return Promise.reject(new Error('failed'))      }    }, |

mutations：

|  |
| --- |
| // 清除本地数据    CLEARTOKEN(state) {      state.token = ''      state.user = {}      removeToken()    }, |

点击退出登录：

|  |
| --- |
| async logout() {        try {          this.$store.dispatch('userLogout')          // 跳转到首页          this.$router.push('/home')        } catch (error) {          alert(error.message)        }      }, |

**导航守卫：**

在首页mounted中派发获取用户信息更改为在路由导航守卫中派发：

|  |
| --- |
| // 配置路由  let router = new VueRouter({    // 配置路由    routes: routes,    // 控制的是滚动行为：即在路由跳转时，从一个路由跳转到另一个路由时，控制跳转成功后页面滚动条的位置    scrollBehavior(to, from, savedPosition) {      return { y: 0 }    },  })  // 全局守卫：前置守卫：在路由跳转之前进行判断  router.beforeEach(async (to, from, next) => {    // to：要跳转的路由信息    // form：来自哪个路由    // next：放行函数，next('/login')放行到指定的路由，next(false):终端当前的导航    // 用户已登录，获取token    let token = store.state.user.token    let userName = store.state.user.user.name    if (token) {      if (to.path == '/login') {        next('/home')      } else {        // 判断仓库的用户信息        if (userName) {          next()        } else {          // 没有用户信息，派发actions让仓库存储用户信息再跳转          try {            await store.dispatch('getUserInfo')            next()          } catch (error) {            // token失效了，获取不到用户信息，重新登录            // 清除token            await store.dispatch('userLogout')            next('/login')          }        }      }    } else {      // 未登录      next()    }  })  export default router |

点击提交订单按钮，向服务器发送请求支付订单：

不利用vuex获取请求数据，直接在组件中使用请求接口获取数据或提交数据，同时保证所有组件都能使用封装的接口，而不用每个组件单独引用：

|  |
| --- |
| // 统一接收api文件夹里面的全部请求函数  import \* as API from '@/api'  new Vue({    render: (h) => h(App),    // 注册路由    // router，此时组件拥有$route,$router属性    router,    // 注册仓库：此时组件实例的身上会多一个$store属性    store,    beforeCreate() {      // 配置全局事件总线      Vue.prototype.$bus = this      // 保证所有组件实例都可以使用      Vue.prototype.$API = API    },  }).$mount('#app') |

提交订单并发送请求：

|  |
| --- |
| // 提交订单      async submitOrder() {        let { tradeNo } = this.orderInfo        let data = {          consignee: this.userDefaultAddress.consignee,          consigneeTel: this.userDefaultAddress.phoneNum,          deliveryAddress: this.userDefaultAddress.fullAddress,          paymentWay: 'ONLINE',          orderComment: this.msg,          orderDetailList: this.orderInfo.detailArrayList,        }        let res = await this.$API.reqSubmitOrder(tradeNo, data)        if (res.code === 200) {          this.orderId = res.data  **在路由跳转时传递query参数（订单编号）给Pay支付页面**          this.$router.push('/pay?orderId=' + this.orderId)          return 'ok'        } else {          return Promise.reject(new Error('failed'))        }      }, |

在支付页面收到query参数以及获取支付信息：

**别在生命周期钩子函数中使用async：**

|  |
| --- |
| mounted() {      this.getPayInfo()    },    methods: {      // 获取支付信息      async getPayInfo() {        let res = await this.$API.reqPayInfo(this.orderId)        // 在组件中存储支付信息        if (res.code === 200) {          this.payInfo = res.data          return 'ok'        } else {          return new Error('failed')        }      },  } |

使用ElementUI完善支付页面：

按需加载：

安装element-ui

npm i --save element-ui

安装按需加载插件：

npm i babel-plugin-component -D

配置文件：

babel.config.js

|  |
| --- |
| module.exports = {    presets: ['@vue/cli-plugin-babel/preset'],    plugins: [      [        'component',        {          libraryName: 'element-ui',          styleLibraryName: 'theme-chalk',        },      ],    ],  } |

main.js中引入element-ui：

|  |
| --- |
| import { Button, MessageBox } from 'element-ui'  // 注册全局组件  Vue.component(Button.name, Button)  // 挂载在组件原型  Vue.prototype.$msgbox = MessageBox  Vue.prototype.$alert = MessageBox.alert |

ElementUI弹出框使用：

|  |
| --- |
| <a class="btn" @click="open">立即支付</a>  methods: {      // 支付弹出框      open() {        this.$alert('<strong>这是 <i>HTML</i> 片段</strong>', '微信支付', {          dangerouslyUseHTMLString: true,          // 中间布局          center: true,          // 是否显示取消按钮          showCancelButton: true,          // 取消关闭按钮          showClose: false,          // 取消按钮文本          cancelButtonText: '支付遇到问题',          // 确定按钮文本          confirmButtonText: '已支付',        })      },    }, |

生成二维码：在npm官网上搜索插件：qrcode

|  |
| --- |
| npm i –-save qrcode |

在要使用的组件中直接引入：

Pay.vue

|  |
| --- |
| import QRCode from 'qrcode'  export default {    name: 'Pay',    data() {      return {        payInfo: {},        timer: null,        // 支付状态        code: '',      }    },    computed: {      orderId() {        return this.$route.query.orderId      },    },    mounted() {      this.getPayInfo()    },    methods: {      // 获取支付信息      async getPayInfo() {        let res = await this.$API.reqPayInfo(this.orderId)        // 在组件中存储支付信息        if (res.code === 200) {          this.payInfo = res.data          return 'ok'        } else {          return new Error('failed')        }      },      // 支付弹出框      async open() {        // 生成二维码(地址)        let url = await QRCode.toDataURL(this.payInfo.codeUrl)        this.$alert(`<img src=${url} />`, '微信支付', {          dangerouslyUseHTMLString: true,          // 中间布局          center: true,          // 是否显示取消按钮          showCancelButton: true,          // 取消关闭按钮          showClose: false,          // 取消按钮文本          cancelButtonText: '支付遇到问题',          // 确定按钮文本          confirmButtonText: '已支付',          // 关闭弹出框的配置          beforeClose: (action, instance, done) => {            // action区分：取消、确定按钮            // instance：当前组件实例            // done：关闭弹出框的方法            if (action === 'cancel') {              alert('请连续管理员！')              // 清除定时器              clearInterval(this.timer)              this.timer = null              done()            } else {              // 判断是否支付              if (this.code === 200) {                clearInterval(this.timer)                this.timer = null                done()                this.$router.push('/paysuccess')              }            }          },        })        // 支付成功：路由的跳转        if (!this.timer) {          // 开启定时器          this.timer = setInterval(async () => {            // 发请求获取用户支付状态            let res = await this.$API.reqPayStatus(this.orderId)            if (res.code === 200) {              // 支付成功              clearInterval(this.timer)              this.timer = null              // 保存支付成功的code              this.code = res.code              // 关闭弹出框              this.$msgbox.close()              // 跳转路由              this.$router.push('/paysuccess')            }          }, 1000)        }        // 支付失败：提示信息      },    },  } |

**全局守卫：**

|  |
| --- |
| // 全局守卫：前置守卫：在路由跳转之前进行判断  router.beforeEach(async (to, from, next) => {    // to：要跳转的路由信息    // form：来自哪个路由    // next：放行函数，next('/login')放行到指定的路由，next(false):终端当前的导航    // 用户已登录，获取token    let token = store.state.user.token    let userName = store.state.user.user.name    if (token) {      if (to.path == '/login') {        next('/home')      } else {        // 判断仓库的用户信息        if (userName) {          next()        } else {          // 没有用户信息，派发actions让仓库存储用户信息再跳转          try {            await store.dispatch('getUserInfo')            next()          } catch (error) {            // token失效了，获取不到用户信息，重新登录            // 清除token            await store.dispatch('userLogout')            next('/login')          }        }      }    } else {      // 未登录      if (        to.path.indexOf('/trade') !== -1 ||        to.path.indexOf('/pay') !== -1 ||        to.path.indexOf('/center') !== -1      ) {        alert('请先登录！')        next('/login?redirect=' + to.path)      } else {        next()      }    }  }) |

**路由独享守卫：**

**在路由规则中设置：route.js**

|  |
| --- |
| {      path: '/pay',      component: Pay,      meta: { show: true },      beforeEnter: (to, from, next) => {        if (from.path === '/trade') {          next()        } else {          next(false)        }      },    }, |

**组件内守卫：**

|  |
| --- |
| // 组件内守卫，这里不能获取组件实例，因为还未跳转到该组件    // 不能访问this，在跳转到该组件前调用    beforeRouteEnter(to, from, next) {      if (from === '/pay') {        next()      } else {        next(false)      }    },    // 在当前路由改变，但是该组件被复用时调用    // 对于一个带有动态参数的路径：/foo/:id，在/foo/1和/foo/2之间跳转的时候    // 由于会渲染同样的foo组件，所以组件实例会复用，此时就会调用这个钩子函数    // 可以访问this组件实例    beforeRouteLeave(to, from, next) {      next()    }, |

**图片懒加载**

<https://www.npmjs.com/package/vue-lazyload>

安装：

|  |
| --- |
| npm i vue-lazyload |

使用：

main.js

|  |
| --- |
| import Vue from 'vue'  import App from './App.vue'  import VueLazyload from 'vue-lazyload'    Vue.use(VueLazyload)    // or with options  Vue.use(VueLazyload, {    preLoad: 1.3,    error: 'dist/error.png',    loading: 'dist/loading.gif',    attempt: 1  })    new Vue({    el: 'body',    components: {      App    }  }) |

template：自定义指令

|  |
| --- |
| <ul>      <li v-for="img in list">        <img v-lazy="img.src" />      </li>  </ul> |

use v-lazy-container work with raw HTML

|  |
| --- |
| <div v-lazy-container="{ selector: 'img' }">      <img data-src="//domain.com/img1.jpg" />      <img data-src="//domain.com/img2.jpg" />      <img data-src="//domain.com/img3.jpg" />    </div> |

具体案例使用：

（1）安装

（2）在main.js中使用插件：

|  |
| --- |
| // 引入插件  import VueLazyload from 'vue-lazyload'  // 引入图片  import load from '@/assets/images/itemlike05.png'  // 使用插件：Vue.use()实际上是调用了插件的install方法  Vue.use(VueLazyload, {    // 懒加载默认的图片    loading: load,  }) |

相当于在全局新增了v-lazy指令：

|  |
| --- |
| <img v-lazy="good.defaultImg" /> |

**自定义插件**

Vue.js 的插件应该暴露一个 install 方法。这个方法的第一个参数是 Vue 构造器，第二个参数是一个可选的选项对象：

|  |
| --- |
| MyPlugin.install = function (Vue, options) {    // 1. 添加全局方法或 property    Vue.myGlobalMethod = function () {      // 逻辑...    }    // 2. 添加全局资源    Vue.directive('my-directive', {      bind (el, binding, vnode, oldVnode) {        // 逻辑...      }      ...    })    // 3. 注入组件选项    Vue.mixin({      created: function () {        // 逻辑...      }      ...    })    // 4. 添加实例方法    Vue.prototype.$myMethod = function (methodOptions) {      // 逻辑...    }  } |

自定义一个指令，使得使用指令的元素的innerHTML全部为大写：

（1）定义插件：

|  |
| --- |
| // Vue插件一定是向外暴露一个对象，且有install方法  // Vue.js 的插件应该暴露一个 install 方法。  // 这个方法的第一个参数是 Vue 构造器，第二个参数是一个可选的选项对象：  let myPlugins = {}  myPlugins.install = function (Vue, options) {    // Vue.prototype.$bus：任何组件都可以使用    // Vue.directive    // Vue.component    // Vue.filter    Vue.directive(options.name, (element, params) => {      element.innerHTML = params.value.toUpperCase()    })  }  export default myPlugins |

main.js中使用：

|  |
| --- |
| // 引入自定义插件  import myPlugins from './plugins/myPlugins'  Vue.use(myPlugins, {    name: 'upper',  }) |

在模板中使用：

|  |
| --- |
| <h1 v-upper="msg"></h1>  data() {      return {        msg: 'msg',      }    }, |

**表单验证模块实现：**

vee-validate的基本使用：

（1）插件的安装和使用：

|  |
| --- |
| npm i vee-validate |

（2）将引入使用其封装到一个单独的文件，防止main.js文件混乱：

|  |
| --- |
| // vee-validate插件：表单验证区域  import Vue from 'vue'  import VeeValidate from 'vee-validate'  Vue.use(VeeValidate) |

main.js中引入：

|  |
| --- |
| // 引入vee-validate表单校验插件  import '@/plugins/validate' |

validate.js

|  |
| --- |
| // vee-validate插件：表单验证区域  import Vue from 'vue'  import VeeValidate from 'vee-validate'  // 中文提示信息  import zh\_CN from 'vee-validate/dist/locale/zh\_CN'  Vue.use(VeeValidate)  // 表单验证  VeeValidate.Validator.localize('zh\_CN', {    messages: {      ...zh\_CN.messages,      // 修改内置规则的message，让确认密码和密码相同      is: (field) => `${field}必须与密码相同`,    },    attributes: {      phone: '手机号',      code: '验证码',      password: '密码',      password1: '确认密码',      agree: '协议',    },  })  // 自定义规则  VeeValidate.Validator.extend('agree', {    validate: (value) => {      return value    },    getMessage: (field) => field + '必须同意',  }) |

在组件中使用：

|  |
| --- |
| <div class="content">          <label>手机号:</label>          <input            placeholder="请输入你的手机号"            v-model.number="phone"            name="phone"            v-validate="{ required: true, regex: /^1\d{10}$/ }"            :class="{ invalid: errors.has('phone') }"          />          <span class="error-msg">{{ errors.first('phone') }}</span>        </div>        <div class="content">          <label>验证码:</label>          <input            type="text"            placeholder="请输入验证码"            v-model="code"            name="code"            v-validate="{ required: true, regex: /^\d{6}$/ }"            :class="{ invalid: errors.has('code') }"          />          <button style="width: 100px; height: 40px" @click="getCode">            点击获取验证码          </button>          <span class="error-msg">{{ errors.first('code') }}</span>        </div>        <div class="content">          <label>登录密码:</label>          <input            type="password"            placeholder="请输入你的登录密码"            v-model="password"            name="password"            v-validate="{ required: true, regex: /^[0-9a-zA-Z]{8,20}$/ }"            :class="{ invalid: errors.has('password') }"          />          <span class="error-msg">{{ errors.first('password') }}</span>        </div>        <div class="content">          <label>确认密码:</label>          <input            type="password"            placeholder="请输入确认密码"            v-model="passwordCheck"            name="password1"            v-validate="{ required: true, is: password }"            :class="{ invalid: errors.has('password1') }"          />          <span class="error-msg">{{ errors.first('password1') }}</span>        </div>        <div class="controls">          <input            name="agree"            type="checkbox"            :checked="agree"            v-validate="{ required: true, agree: true }"            :class="{ invalid: errors.has('agree') }"          />          <span>同意协议并注册《尚品汇用户协议》</span>          <span class="error-msg">{{ errors.first('agree') }}</span>        </div>        <div class="btn">          <button @click="userRegister">完成注册</button>        </div> |

在注册请求之前判断全部表单是否验证成功：

|  |
| --- |
| // 用户注册      async userRegister() {        // 全部的表单验证成功        const success = await this.$validator.validateAll()        if (success) {          try {            const { phone, code, password, passwordCheck } = this            await this.$store.dispatch('userRegister', {              phone,              code,              password,            })            // 注册成功则跳转到登陆页面进行登录            this.$router.push('/login')          } catch (error) {            alert(error.message)          }        }      }, |

**路由懒加载**

好处：**当打包构建应用时，JavaScript 包会变得非常大，影响页面加载。如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应组件，这样就会更加高效。**

|  |
| --- |
| const foo = () => {    return import('@/pages/Home')  } |

在路由规则中使用：

|  |
| --- |
| {      path: '/home',      component: foo,      meta: { show: true },    }, |

实现路由的懒加载

简写：直接在路由规则中使用函数的形式加载对应的组件

|  |
| --- |
| {      path: '/addcartsuccess',      name: 'addcartsuccess',      component: () => import('@/pages/AddCartSuccess'),      meta: { show: true },    }, |

简单的完整版写法：

|  |
| --- |
| const foo = () => {    return import('@/pages/Home')  }  /\*\*   \* 当打包构建应用时，JavaScript 包会变得非常大，影响页面加载。   \* 如果我们能把不同路由对应的组件分割成不同的代码块   \* 然后当路由被访问的时候才加载对应组件，这样就会更加高效。   \*/  export default [    {      path: '/trade',      component: () => import('@/pages/Trade'),      meta: { show: true },      // 路由独享守卫      beforeEnter: (to, from, next) => {        if (from.path === '/shopcart') {          next()        } else {          // 终端当前导航，重置到from的url          next(false)        }      },    },  } |

**打包上线**

**（1）处理map文件**

项目打包之后，代码都是经过压缩加密的，如果运行时报错，输出的错误信息无法准确得知是哪里的代码出现了错误。

有了map之后就可以像未加密得代码一样，准确的输出是哪一行哪一列有错；所以该文件如果项目不需要就可以去掉；

vue.config.js配置

productionSourceMap：false

|  |
| --- |
| module.exports = {    // 处理map文件    productionSourceMap: false,  ...  } |

**（2）购买云服务器：**