渐进式框架(逐渐递进),虚拟dom,diff算法

插值语法:用于标签体内容,<h1>{{ name }}</h1>

指令语法:用于标签属性,绑定事件等,<a :href="url">点击</a>

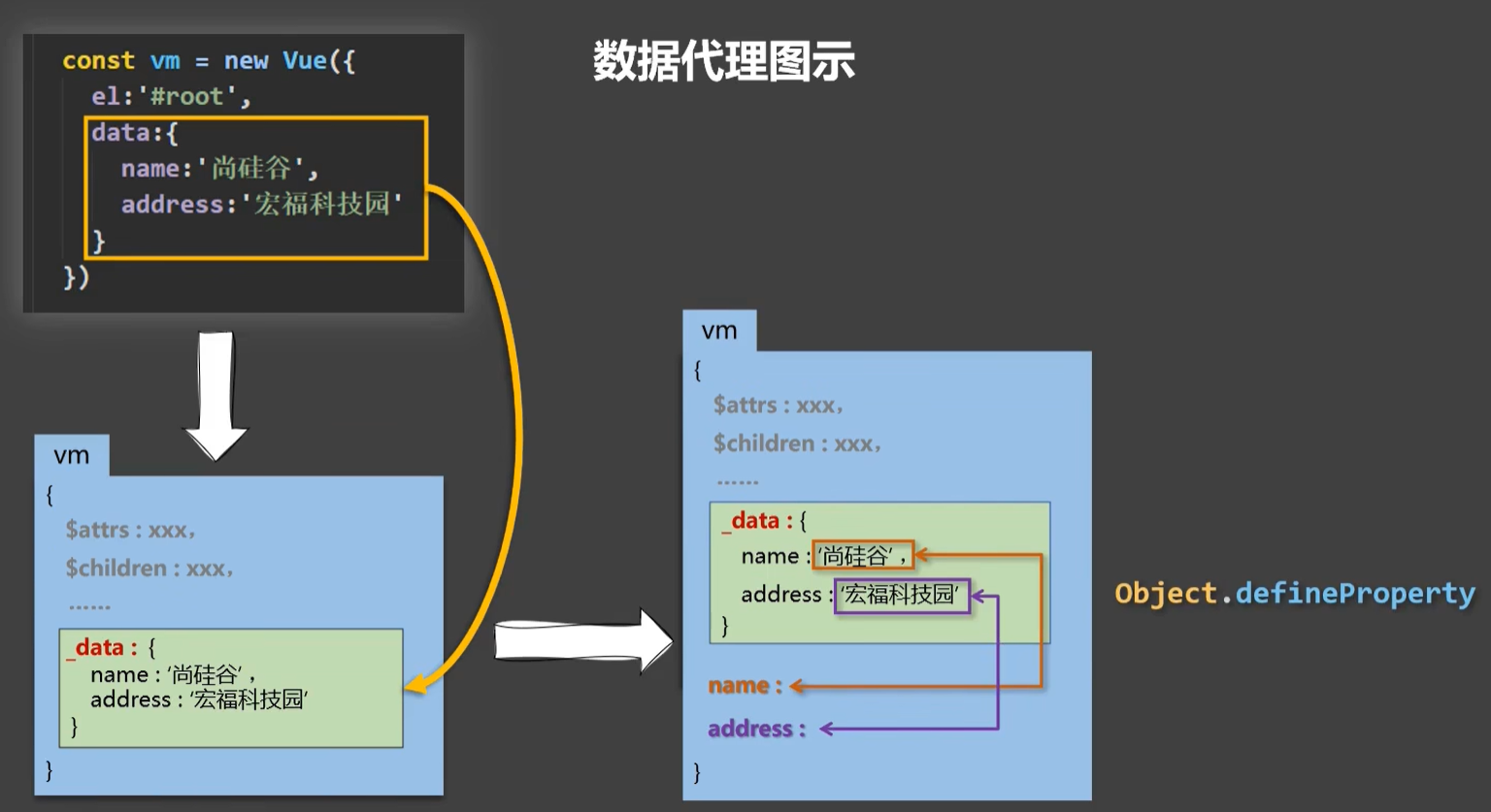
ref:相当于html中的id this.$refs.ref值

scoped:样式只限当前组件使用(默认不写是所有组件的样式会汇总到一起)

重要知识点：被Vue管理的函数写成普通函数,不被Vue管理的函数(定时器的回调函数、ajax的回调函数、Promise的回调函数)写成箭头函数,这样this的指向才是vm或组件实例对象。

数据代理：





Object.defineProperty(obj,'age',{

get(){

return '';

},

set(){

}

})

vue通过数据代理来简化代码写法:{{\_data.name}} --> {{name}}

const vm = new Vue({

el:'#root',

data:{

name:'111'

}

})

data中的数据实际保存在\_data中,data中的每1个属性都有其对应的getter和setter,vm.name实际上使用数据代理的方式使用name的getter获取\_data中的name

计算属性: 对data中的属性进行进一步的加工处理形成新的属性,属性名可包含get()和set()2个函数

computed:{

computedName:{

get(){

return '';

},

set(value){

}

}

}

只有读取没有写时可简写

computed:{

computedName(){

return '';

}

}

监视属性:对data中的属性(包含计算属性)进行监视,监测属性值的变化，可获取变化前后的值来做进一步的业务处理。

a为多层级结构

a:{

b:{

b1:'b1',

b2:'b2'

b3:{

bb1:'bb1'

}

},

c:'c'

}

watch:{

name:{

immediate:true, //初始化是让hander调用1下,默认false

deep:true, //深度监视,默认false,用于监测的属性为多层级结构

handler(newValue,oldValue){ //当name发生改变是handler调用

//处理具体业务

}

}

}

写法2：

const vm = new vue({

...

})

vm.$watch('name',{

immediate:true,

deep:true,

handler(newValue,oldValue){

//处理具体业务

}

})

当只有handler时可简写

watch:{

name(newValue,oldValue){ //当name发生改变是handler调用

//处理具体业务

}

}

写法2：

const vm = new vue({

...

})

vm.$watch('name',function(newValue,oldValue){

//处理具体业务

})

不通过在data{}中修改代码新增1一个属性的情况下,还可以通过set()向data{}中添加属性,该属性同样是响应式的。

student为data中已存在的属性名,sex为新增的属性名,男为新增的属性值

重点：不能给vm或vm的根数据对象添加属性,即该新增属性不能属于data中的第1层级

修改数组中的某个响应式元素一定要用如下方法(vue对这些方法进行了包装):push(),pop(),shift(),unshift(),splice(),sort(),reverse()

Vue.set(this.student,'sex','男') 或 this.$set(this.student,'sex','男')

收集表单数据(v-model.number收集的内容会转为有效数字)

input为text,v-model收集的就是用户输入的value值

input为radio,v-model收集是用户输入的value值,要配置value值

input为checkbox

如果没有配置value值,收集的是checked(true/false)

如果配置了value值,v-model值为非数组,收集的是checked(true/false);v-model值为数组,收集的就是value组成的数组

自定义指令(定义时不需要v-,使用时要加上v-;指令名如果为多个单词,定义: 'bind-number',使用: v-bind-number)

<span v-big="n"></span>

<span v-fbind="n"></span>

directives:{

//简单函数写法;调用:1.指令与元素成功绑定时(一上来) 2.指令所在模板被重新解析时

big(element,binding){

element.innerText = binding.value \* 10

},

//完整写法

fbind:{

//指令与元素成功绑定时(一上来)

bind(element,binding){

element.value = binding.value

},

//指令所在元素被插入页面时

inserted(element,binding){

element.focus()

},

//指令所在模板被重新解析时

update(element,binding){

element.value = binding.value

}

}

}

定义全局指令

//完整写法

Vue.directive('fbind'{

//指令与元素成功绑定时(一上来)

bind(element,binding){

element.value = binding.value

},

//指令所在元素被插入页面时

inserted(element,binding){

element.focus()

},

//指令所在模板被重新解析时

update(element,binding){

element.value = binding.value

}

})

//简单函数写法

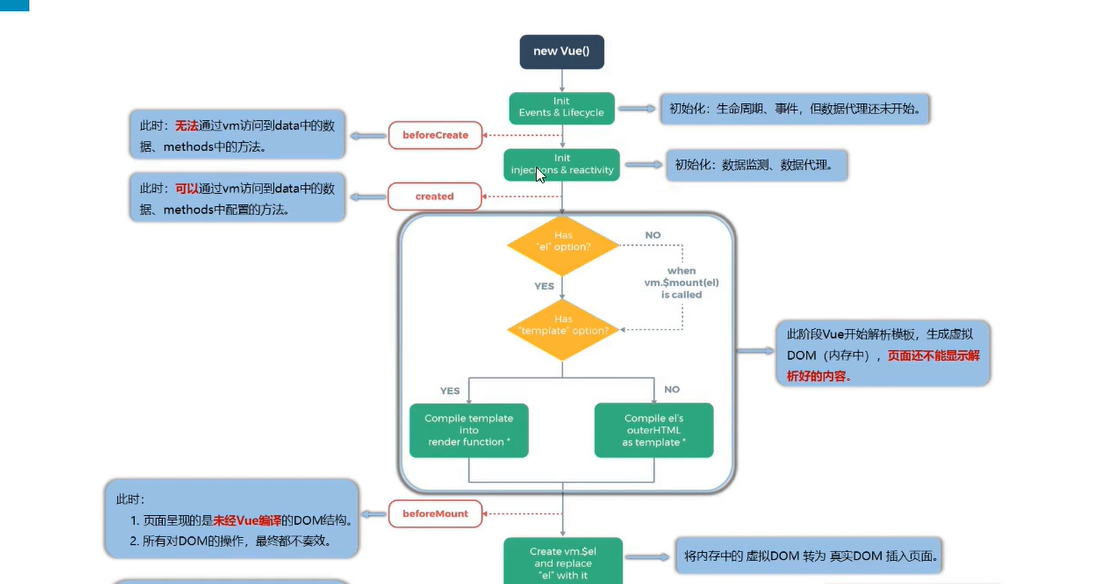
Vue.directive('big',function(element,binding){

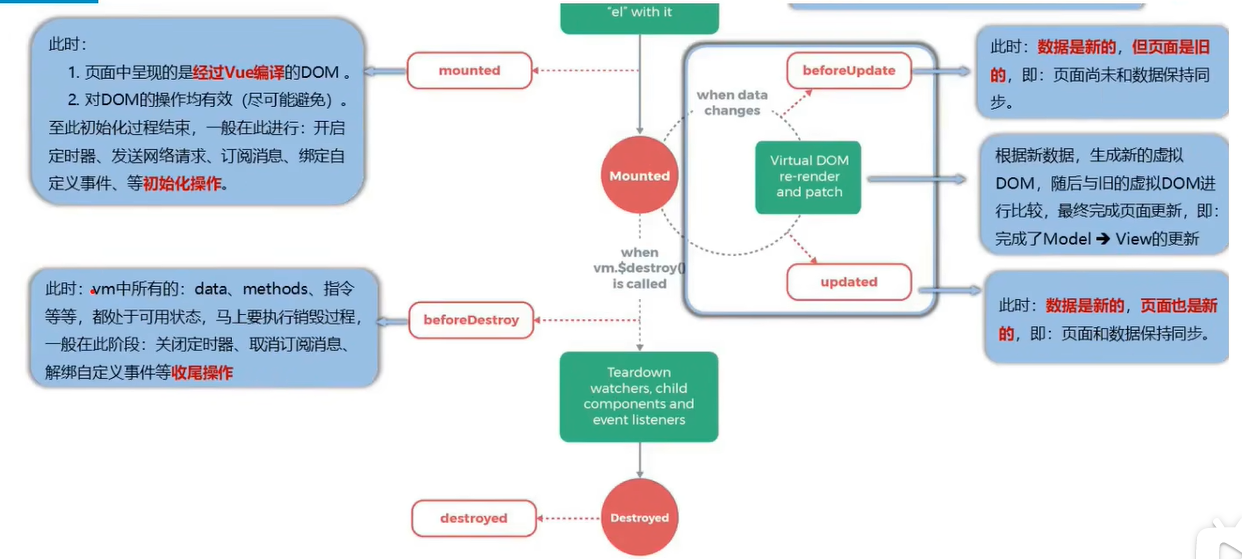
element.innerText = binding.value \* 10

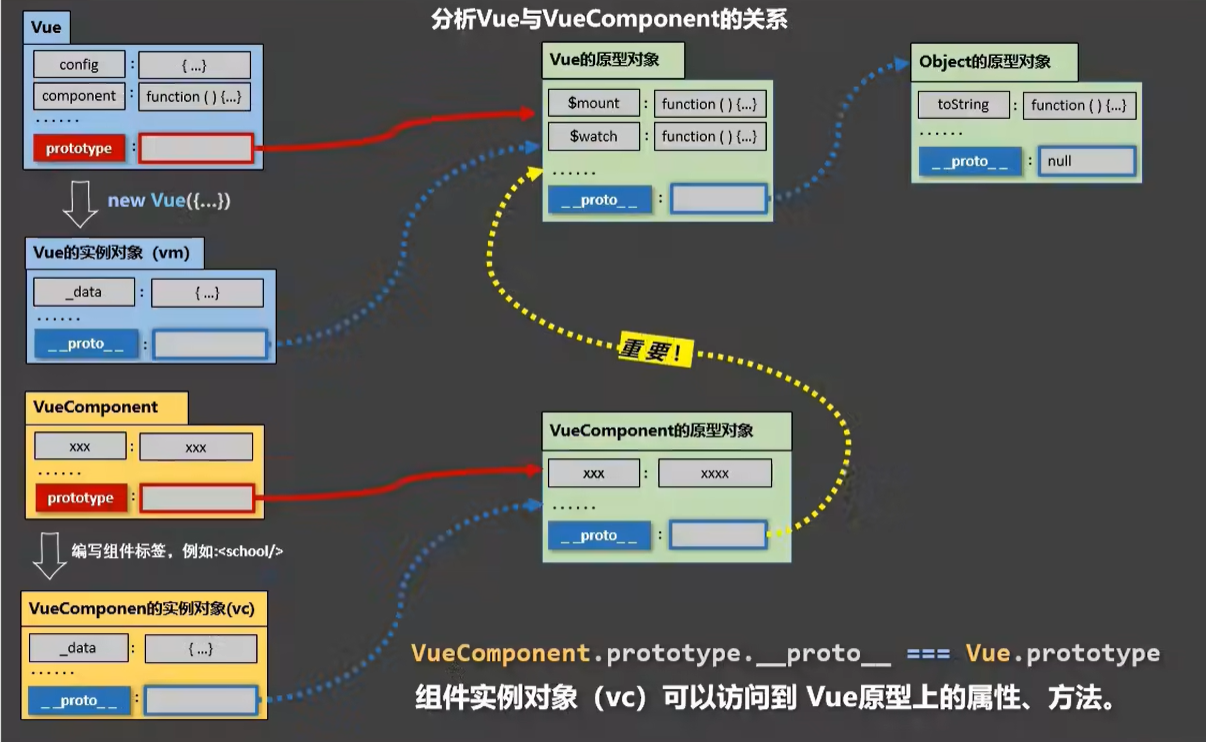
})

vue的生命周期函数









mixin混入:将配置项相同的部分(data,methods...等)提取到js中,再引入使用。如果自己的组件和混入中的内容相同,

数据和方法以自己组件为主,生命周期函数会都使用。

1.局部引入

import {xxx} from '...'

新增配置项

mixins:[xxx]

2.全局引入

在main.js中引入

import {xxx} from '...'

Vue.mixin(xxx)

插件:对Vue功能的增强

定义1个plugins.js

export default{

install(参数1,参数2...){

.....

}

}

在main.js中引入

import plugins from '...'

Vue.use(plugins,参数1,参数2...)

插槽：让父组件爱你可以向子组件指定位置插入html结构内容。

1.默认插槽

父组件:

<Category>

<div>html结构内容<div>

<Category>

子组件:

<div>

<solt>此处填写父组件传入子组件html内容为空时显示内容</solt>

</div>

2.具名插槽(可指定多个插槽,根据名称决定htnl内容放在那个插槽)

父组件:

<Category>

<template solt="center">

<div>html结构内容1<div>

</template>

<template solt="footer">

<div>html结构内容2<div>

</template>

<Category>

子组件:

<div>

<solt name="center">此处填写父组件传入子组件html内容为空时显示内容</solt>

<solt name="footer">此处填写父组件传入子组件html内容为空时显示内容</solt>

</div>

3.作用域插槽；数据在子组件中,html结构和需要的数据在父组件中

父组件:

<Category>

<template scope="mydata">

//生成ul结构

<ul>

<li v-for="(g,index) in mydata.myGames" :key="index">{{g}}</li>

</ul>

</template>

<Category>

<Category>

<template scope="mydata">

//生成h4结构

<h4 v-for="(g,index) in mydata.myGames" :key="index">{{g}}</h4>

</template>

<Category>

子组件:

<div>

<solt :myGames="games">此处填写父组件传入子组件html内容为空时显示内容</solt>

</div>

data(){

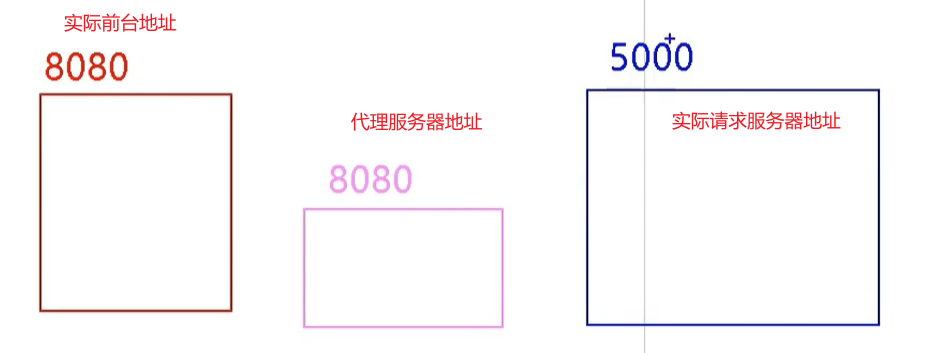
return(){

games:['g1','g2','g3']

}

}

配置代理



方式1: 简单配置,只能配置1个代理,不能灵活控制走不走代理

在vue.config.js中:

module.exports={

...

//开启代理服务器,5000为实际请求的目标服务器,开启的代理服务器为8080

devServer:{

proxy: 'http://localhost:5000'

}

}

请求: axios.get('http://localhost:8080/students')

方式2

在vue.config.js中:

module.exports={

...

//开启代理服务器,5000为实际请求的目标服务器,开启的代理服务器为8080

devServer:{

proxy: {

'/api1':{ //路径匹配规则

target: 'http://localhost:5000', // 请求的目标服务器

pathRewrite:{'^/api1':''}, //路径重写规则

ws:true, //默认为true,用于支持websocket

changeOrigin:true //默认为true,为true时请求与实际请求服务器相同即5000;为false时请求为真实即8080

},

'/api2':{ //路径匹配规则

target: 'http://localhost:5001', // 请求的目标服务器

pathRewrite:{'^/api2':''}, //路径重写规则

ws:true, //默认为true,用于支持websocket

changeOrigin:true //默认为true,为true时请求与实际请求服务器相同即5001;为false时请求为真实即8080

}

}

}

}

请求: axios.get('http://localhost:8080/students')

组件间通信

props:向组件中传值,组件中用prop接收的值可以向data中的属性一样使用,但是值默认不支持修改;

可用于父组件向子组件传值;还可以传递1个方法(函数),可用于子组件向父组件传值

<Student name="李四" sex="女" :age="18" :myFunction="myFunction"/>

在父组件中提前定义好一个方法并传给子组件

methods:{

myFunction(参数){

...

}

}

在子组件中接收并调用

methods:{

zFunction(参数){

this.myFunction(参数);

}

}

组件内接收

方式1:

props:['name','sex','age','myFunction']

方式2:

props:{

name:String,

age:Number,

sex:String

}

方式3:

props:{

name:{

type:String, //类型

required:true //是否必要

},

age:{

type:Number,

default:99 //默认值

},

sex:{

type:String,

required:true

}

}

修改使用:重新定义1个属性,将prop传过来的值赋给myAge,操作和使用myAge

data(){

return{

myAge:this.age

}

}

组件的自定义事件: 适用于子组件向父组件传递数据

父组件

方式1: myFunction为自定义事件名称

<Student @myFunction="getStudentName"/>

methods:{

getStudentName(接收到的参数){

...

}

}

<Student ref="student"/>

methods:{

getStudentName(接收到的参数){

...

}

},

mounted(){

this.$refs.student.$on('myFunction',this.getStudentName) --绑定自定义事件

}

子组件

myFunction为自定义事件名称，$emit用于触发自定义事件

methods:{

sendStudentName(){

this.$emit('myFunction',传递的参数)

}

}

全局事件总线：任意组件间通信

安装全局事件总线:

在main.js中:

new Vue({

el:'#app',

render: h=> h(App),

beforeCreate(){

Vue.prototype.$bus = this //安装

}

})

接收数据组件

methods:{

getStudentName(接收到的参数){

...

}

},

mounted(){

this.$bus.$on('myFunction',this.getStudentName) --绑定自定义事件,myFunction为自定义事件名称

},

beforeDestroy(){

//解绑自定义事件

this.$off('myFunction') //解绑1个自定义事件

this.$off(['myFunction','myFunction2']) //解绑多个自定义事件

this.$off() //解绑所有自定义事件

}

发送数据组件

myFunction为自定义事件名称，$emit用于触发自定义事件

methods:{

sendStudentName(){

this.$bus.$emit('myFunction',传递的参数)

}

}

消息的订阅与发布:借助第3方库实现的任意组件间通信

安装依赖: npm i pubsub-js

接收数据组件:在组件中引入: import pubsub from 'pubsub-js'

methods:{

getStudentName(订阅的消息名称,接收到的参数){

...

}

},

mounted(){

this.pubId = pubsub.subscribe('myFunction',this.getStudentName) --订阅消息,myFunction为消息名称

},

beforeDestroy(){

//取消订阅消息

pubsub.subscribe(this.pubId)

}

发送数据组件:在组件中引入: import pubsub from 'pubsub-js'

myFunction为消息名称

methods:{

sendStudentName(){

pubsub.publish('myFunction',传递的参数)

}

}

nextTick:在Dom更新结束后再执行其指定的回调。

...

todo.isEdit = true

...

this.$nextTick(function(){

this.$refs.inputTitle.focus()

})

上述代码中数据改变了,需要等重新解析模板后,input出现在页面上在执行获取焦点代码才能生效

使用: 当改变数据后,要基于更新后的新DOM进行某些操作时,要在nextTick所指定的回调函数中执行。

vuex:组件间的共享数据。



安装依赖:npm i vuex@3

在main.js中引用并使用

//引入store

//import store from './store'

new Vue({

el:'#app',

render: h => h(App),

store

})

在src下新建store/index.js

//引入Vue

import Vue form 'vue'

//引入Vuex

import Vuex from 'vuex'

//使用vuex

Vue.use(Vuex)

//准备actions-用于响应组件中的动作

//若action中有其他业务逻辑或需发送后台请求,可写在action中

const actions = {

jia(context,value){

context.commit('JIA',value)

},

jiaOdd(context,value){

if(context.state.sum % 2){

context.commit('JIA',value)

}

},

}

//准备mutations-用于操作数据(state)

const mutations = {

JIA(state,value){

state.sum += value

}

}

//准备state-用于存储数据

const state = {

sum:0,

school:'abc'

}

//准备getters-用于将state中的数据进一步加工

const getters = {

bigSum(state){

return state.sum\*10

}

}

//创建并暴露store

export default new Vuex.Store({

actions,

mutations,

state,

getters

})

组件中方法

插值语法获取值: {{$store.state.sum}} {{$store.getters.bigSum}}

methods:{

increment(){

this.$store.dispatch('jia',this.n)

//也可直接commit

this.$store.commit('JIA',this.n)

},

incrementOdd(){

this.$store.dispatch('jiaOdd',this.n)

},

}

简写

import {mapState,mapGetters,mapMutations,mapActions} from 'vuex'

computed:{

he(){

return this.$store.state.sum

},

xuexiao(){

return this.$store.state.school

},

//借助mapState生成计算属性,从state中读取数据(对象写法,计算属性名可与存储数据名不一样)

// ...mapState({he:'sum',xuexiao:'school'})

sum(){

return this.$store.state.sum

},

school(){

return this.$store.state.school

},

//借助mapState生成计算属性,从state中读取数据(数据写法,计算属性名必须与存储数据名一样)

// ...mapState(['sum','school'])

bigHe(){

return this.$store.getters.bigSum

},

//借助mapGetters生成计算属性,从getters中读取数据(对象写法,计算属性名可与存储数据名不一样)

// ...mapGetters({he:'bigHe'})

bigSum(){

return this.$store.getters.bigSum

},

//借助mapGetters生成计算属性,从getters中读取数据(数据写法,计算属性名需与存储数据名一样)

// ...mapGetters(['bigSum'])

},

methods:{

increment(){

//可直接commit

this.$store.commit('JIA',this.n)

},

//借助mapMutations生成对应的方法(对象写法,生成的方法名可与mutations中定义的方法名不一样),使用简写传参时需要在调用方法是传入参数:@click="increment(n)"

// ...mapMutations({increment:'JIA'})

//借助mapMutations生成对应的方法(数组写法,生成的方法名需与mutations中定义的方法名一样),使用简写传参时需要在调用方法是传入参数:@click="JIA(n)"

// ...mapMutations(['JIA'])

incrementOdd(){

this.$store.dispatch('jiaOdd',this.n)

},

//借助mapActions生成对应的方法(对象写法,生成的方法名可与actions中定义的方法名不一样),使用简写传参时需要在调用方法是传入参数:@click="incrementOdd(n)"

// ...mapActions({incrementOdd:'jiaOdd'})

//借助mapActions生成对应的方法(数组写法,生成的方法名需与actions中定义的方法名一样),使用简写传参时需要在调用方法是传入参数:@click="jiaOdd(n)"

// ...mapActions(['jiaOdd'])

}

vuex模块化

在src下新建store/count.js

在src下新建store/person.js

count.js中

//若action中有其他业务逻辑或需发送后台请求,可写在action中

//count功能相关的配置

export default {

namespaced:true, //开启命名空间

actions:{

jia(context,value){

context.commit('JIA',value)

},

jiaOdd(context,value){

if(context.state.sum % 2){

context.commit('JIA',value)

}

},

},

mutations{

JIA(state,value){

state.sum += value

},

},

state{

sum:0,

school:'abc',

},

getters{

bigSum(state){

return state.sum\*10

}

},

}

person.js中

//若action中有其他业务逻辑或需发送后台请求,可写在action中

//person功能相关的配置

export default {

namespaced:true, //开启命名空间

actions:{},

mutations{

ADD\_PERSON(state,value){

state.personList.unshift(value)

}

},

state{

personList:[{id:'001',name:'张三'}]

},

getters{

firstPersonName(state){

return state.personList[0].name

}

},

}

在src下新建store/index.js

//引入Vue

import Vue form 'vue'

//引入Vuex

import Vuex from 'vuex'

import countOptions from './count'

import personOptions from './person'

//使用vuex

Vue.use(Vuex)

//创建并暴露store

export default new Vuex.Store({

modules:{

countAbout:countOptions,

personAbout:personOptions

}

})

count组件中：借助简写方式

import {mapState,mapGetters,mapMutations,mapActions} from 'vuex'

computed:{

//借助mapState生成计算属性,从state中读取数据(对象写法,计算属性名可与存储数据名不一样)

// ...mapState('countAbout',{he:'sum',xuexiao:'school'})

// ...mapState('personAbout',{renliebiao:'personList'})

//借助mapState生成计算属性,从state中读取数据(数据写法,计算属性名必须与存储数据名一样)

// ...mapState('countAbout',['sum','school'])

// ...mapState('personAbout',['personList'])

//借助mapGetters生成计算属性,从getters中读取数据(对象写法,计算属性名可与存储数据名不一样)

// ...mapGetters('countAbout',{he:'bigHe'})

//借助mapGetters生成计算属性,从getters中读取数据(数据写法,计算属性名需与存储数据名一样)

// ...mapGetters('countAbout',['bigSum'])

},

methods:{

//借助mapMutations生成对应的方法(对象写法,生成的方法名可与mutations中定义的方法名不一样),使用简写传参时需要在调用方法是传入参数:@click="increment(n)"

// ...mapMutations('countAbout',{increment:'JIA'})

//借助mapMutations生成对应的方法(数组写法,生成的方法名需与mutations中定义的方法名一样),使用简写传参时需要在调用方法是传入参数:@click="JIA(n)"

// ...mapMutations('countAbout',['JIA'])

//借助mapActions生成对应的方法(对象写法,生成的方法名可与actions中定义的方法名不一样),使用简写传参时需要在调用方法是传入参数:@click="incrementOdd(n)"

// ...mapActions('countAbout',{incrementOdd:'jiaOdd'})

//借助mapActions生成对应的方法(数组写法,生成的方法名需与actions中定义的方法名一样),使用简写传参时需要在调用方法是传入参数:@click="jiaOdd(n)"

// ...mapActions('countAbout',['jiaOdd'])

}

person组件中:原始写法

computed:{

personList(){

return this.$store.state.personAbout.personList

},

firstPersonName(){

return this.$store.getters['personAbout/firstPersonName']

},

sum(){

return this.$store.state.countAbout.sum

}

},

methods:{

add(){

const personObj={id:'002',name:this.name}

this.$store.commit('personAbout/ADD\_PERSON',personObj)

}

}

路由(插件):一组key-value的对应关系。

安装:npm i vue-router@3

引入并应用:

在main.js中

import VueRouter from 'vue-router'

import router from './router.js'

Vue.use(VueRouter)

new Vue({

...

router:router

})

在src下新建router/index.js

import VueRouter from 'vue-router'

import About from '../pages/About'

import Home from '../pages/Home'

import News from '../pages/News'

export default new VueRouter({

routes:[

{

name:'guanyu', //路由命名

path:'/about', //路径

component:About //组件

},

{

name:'jia',

path:'/home',

component:Home,

children:[

{

name:'xinwen',

path:'news', //非1级路由不需要/

component:News

}

]

},

]

})

组件中

router-link类似于a标签跳转

<router-link to="/about">About</router-link>

<router-link to="/home/news">News</router-link>

//展示

<router-view><router-view>

路由传参query方式

//1.字符串写法

<router-link :to="`/home/news?id={m.id}&title=${m.title}`">News</router-link>

//2.对象写法

<router-link :to="{

path:'/home/news',

query:{

id:m.id,

title:m.title

}

}">News

</router-link>

有name可以直接用name

<router-link :to="{

name:'xinwen',

query:{

id:m.id,

title:m.title

}

}">News

</router-link>

接收参数

$route.query.id

$route.query.title

路由传参params方式

name:'xinwen',

path:'news/:id/:title', //非1级路由不需要/,加占位符

component:News

//1.字符串写法

<router-link :to="`/home/news/{m.id}/${m.title}`">News</router-link>

//2.对象写法,不能使用path,只能用name

<router-link :to="{

name:'xinwen',

params:{

id:m.id,

title:m.title

}

}">News

</router-link>

接收参数

$route.params.id

$route.params.title

prop传参

{

name:'xinwen',

path:'news', //非1级路由不需要/

component:News,

//第一种,值为对象,传的值为固定值

props:{id:666,title:'你好'}

//第二种,值为布尔值,true时会把路由的所有params参数以props的形式传给组件,只支持params参数

props:true

//第三种,值为函数

props($route){

return {id:$route.query.id,title:$route.query.title}

//return {id:$route.params.id,title:$route.params.title}

}

}

组件内接收

props:['id','title']

编程式路由:通过js代码实现路由跳转

methods:{

myPush(m){

this.$router.push({

name:'xinwen',

query:{

id:m.id,

title:m.title

}

})

},

myReplace(m){

//路由的跳转模式为replace,即会覆盖前一次路由跳转历史

this.$router.replace({

name:'xinwen',

query:{

id:m.id,

title:m.title

}

})

},

myBack(){

this.$router.back() //路由后退

},

myForward(){

this.$router.forward() //路由前进

},

myGo(){

this.$router.go(1) //路由的前进或后退由参数数值决定,正数前进,负数后退

}

}

路由的2个生命周期函数

//组件被激活

activated(){

...

},

//组件失活

deactivated(){

...

}

路由守卫

在src/router/index.js中

import VueRouter from 'vue-router'

import About from '../pages/About'

import Home from '../pages/Home'

import News from '../pages/News'

const router new VueRouter({

routes:[

{

name:'guanyu', //路由命名

path:'/about', //路径

component:About //组件

},

{

name:'jia',

path:'/home',

component:Home,

children:[

{

name:'xinwen',

path:'news', //非1级路由不需要/

component:News,

meta:{isAuth:false} //meta为原有属性,值可根据具体业务需求添加

}

]

},

]

})

//全局前置路由守卫,初始化和每次路由切换之前被调用

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

if(to.meta.isAuth && ...){

...

next()

}

})

//全局后置路由守卫,初始化和每次路由切换之后被调用

router.afterEach((to,from)=>{

...

})

export default router

独享路由守卫,只有单独的前置路由守卫,没有单独的后置路由守卫

{

name:'xinwen',

path:'news', //非1级路由不需要/

component:News,

meta:{isAuth:false} //meta为原有属性,值可根据具体业务需求添加

beforeEnter:(to,from,next)=>{

...

}

}

组件内的路由守卫

组件内

//通过路由规则进入该组件时被调用

beforeRouteEnter(to,from,next){

...

},

//通过路由规则离开该组件时被调用

beforeRouteLeave(to,from,next){

...

}

路由模式:history(#)和hash(/),默认history

vue前端到后台的传参方式

queryParams: {

vModType: "",

acc: ""

}

listRiskModMag(this.queryParams).then(response => {

});

export function listRiskModMag(query) {

return request({

url: '/zhfxjc/riskmodmag/list',

method: 'get',

params: query

})

}

后台：TbRiskModMag tbRiskModMag

let formData = new FormData();

formData.append("file", this.uploadForm.paramsFile); //文件

formData.append("appName", this.uploadForm.uploadAppName); //版本名称

importFile(formData).then(response => {

})

export function importFile(data){

return request({

url: '/zhfxjc/riskmodmag/importfile',

headers: {

'Content-Type': 'multipart/form-data'

},

method: 'post',

data: data

})

}

后台:@RequestParam("file") MultipartFile file

issueRiskModMag({ "irowids": this.ids, "dInspectionEndDate": this.issueFormData.dInspectionEndDate }).then(response => {

});

export function issueRiskModMag(data) {

return request({

url: '/zhfxjc/riskmodmag/issueRiskModMag',

method: 'post',

data: data

})

}

后台:@RequestBody RiskModMagReq riskModMagReq

const irowids = row.impId || this.ids;

delRiskAccPoolImport(irowids)

export function delRiskAccPoolImport(irowid) {

return request({

url: '/zhfxjc/riskaccpool/' + irowid,

method: 'delete'

})

}

后台:

@DeleteMapping("/{irowid}")

public AjaxResult delete(@PathVariable String irowid) {

let formData = new FormData();

formData.append("irowids", this.ids);

issueCustComplaints(formData).then(response => {

});

export function issueCustComplaints(data) {

return request({

url: '/zhfxjc/custcomplaints/issueCustComplaints',

method: 'post',

data: data

})

}

@RequestParam(value = "irowids", required = true) List<String> irowids

键值对或对象类型数据用get,params:query 传参,后台可用@RequestParam单个或封装对象接收

表单post,data:data 传参(注意传文件),后台可用@RequestParam单个或封装对象接收

键值对或对象类型数据用post,data:data 传参,后台必须要用@RequestBody封装的对象接收

路径传参@PathVariable