

Vue 从0到1

slot mixin vue-router 优化 吸顶 中间件 vue优化瓶颈 map 业务场景 key 自定义组件、指令

动态添加路由 map

安装：

- vue init webpack projectName;
- cd projectName
- Cpm install
- Cpm run dev

==>>启动起来了

一. 图片懒加载：

- cnpm install vue-lazyload --save-dev
- 在入口文件main.js 中引入并使用：

Import VueLazyLoad from 'vue-lazyload'

Vue.use(VueLazyLoad)

- 修改图片显示方式为懒加载：将src属性直接改为v-lazy

```
<ul>
  <li v-for="(item, index) in list">
    <img v-lazy="item" >
  </li>
</ul>

<router-view/>

</div>
</template>

<script>
export default {
  name: 'App',
  data(){
    return{
      list:[
        '../static/img/logo.png',
        '../static/img/logo.png',
        '../static/img/logo.png',
        '../static/img/logo.png'
      ]
    }
  }
}
```

二. less安装使用：

npm **install** less less-loader --save //将less和less-loader安装到开发依赖

lang='less'

修改webpack.config.js文件，配置loader加载依赖，让其支持外部的less,在原来的代码上添加

```
1 {  
2  
3   test: /\.less$/,  
4  
5   loader: "style-loader!css-loader!less-loader",  
6  
7 },
```

三. elementUI 安装使用

cnpm install element-ui --save

import ElementUI **from** 'element-ui'

import 'element-ui/lib/theme-chalk/index.css'

Vue.use(ElementUI)

四. 路由懒加载

```
const HelloWorld = ()=>import("@/components/HelloWorld")
const routes = [
  {
    path: '/',
    component: HelloWorld
  }
]

export default new Router({
  routes
  //strict: process.env.NODE_ENV !== 'production',
})
```

五. 组件懒加载

```
<script>
//import headerTop from '../components/top'
const headerTop = () => import('../components/top')
import footerBottom from '../components/bottom'
import childA from '../components/childA'
import childB from '../components/childB'

export default {
  name: "brothers",
  data(){
    return{
    }
  },
  components:{
    headerTop,
    footerBottom,
    childB,
    childA,
  }
}
```

六. Axios 安装配置

cnpm install axios --save
Cpm install qs.js --save
Import axios from './node_modules/axios'
Import qs from 'qs.js'
Vue.prototype.\$axios = axios
main.js中将axios注入到new Vue中

使用和跨域：proxyTable

```
get方式  
this.$axios.get('/user', {  
  数据  
})  
  .then(function (response) {  
  })  
  .catch(function (error) {  
  });
```

vue 中配置跨域访问后台

```
proxyTable: {  
  '/api': {  
    target: 'http://127.0.0.1:18080', // 后台访问地址  
    changeOrigin: true,  
    pathRewrite: {  
      '^/api': ''  
    }  
  }  
},
```

```

post方式
this.$axios.post('/user' , {
  params: {
    数据对象(此处根据情况)
  }
})
.then(function (response) {
})
.catch(function (error) {
});

```

七. key的作用：

vue中列表循环需加:key="唯一标识" 唯一标识可以是item里面id index等，因为vue组件高度复用增加Key可以标识组件的唯一性，为了更好地区别各个组件 key的作用主要是为了高效的更新虚拟DOM的

八. this.\$nextTick (() => {})

下次DOM更新循环结束之后执行延迟回调。场景：在修改数据之后立刻使用这个方法，获取更新后的DOM。如：v-if 设置为true后获取该dom；简单理解就是：数据更新后，自动执行该函数。

Vue生命周期的created () 钩子函数进行的DOM操作一定要放在Vue. nextTick()的回调函数中，原因是在created () 钩子函数执行时候DOM并未进行任何渲染，此时进

行DOM操作无异于徒劳，所以放在nextTick中。与之对应的就是mounted函数，该钩子函数执行时所有的Dom挂载已经完成。所以不需要nextTick。

原理：Vue是异步执行dom更新的，一旦观察到数据变化，Vue就会开启一个队列，然后把在同一个事件循环 (event loop) 当中观察到数据变化的 watcher 推送进这个队列。如果这个watcher被触发多次，只会被推送到队列一次。这种缓冲行为可以有效的去掉重复数据造成的不必要的计算和DOM操作。而在下一个事件循环时，Vue会清空队列，并进行必要的DOM更新。

当你设置 `vm.someData = 'new value'`，DOM 并不会马上更新，而是在异步队列被清除，也就是下一个事件循环开始时执行更新时才会进行必要的DOM更新。如果此时你想要根据更新的 DOM 状态去做某些事情，就会出现问题。。为了在数据变化之后等待 Vue 完成更新 DOM，可以在数据变化之后立即使用 `Vue.nextTick(callback)`。这样回调函数在 DOM 更新完成后就会调用。

九. Vue的生命周期



- beforeCreate 中的 message

undefined;

数据已经和 data 属性 msg 进行了绑定，但此时还没有 el 选项（undefined 还没有挂载）

el, data, data 都是

created 的时候数

重点： created 和 beforeMounted 的过程：

首先判断有没有 el 选项，有则进行下面的编译，没有 el 就停止生命周期，直到 vue 实例调用 `vm.$mount(el)`

如果有 el，再判断有 template 参数，有则把它当作模版编译成 render 函数，没有则把外部的 html 当作模版编译。

Vue对象中还有一个**render**函数，它以**createElement**作为参数，然后做渲染操作，而且可以直接嵌入**JSX**。

优先级：render函数 > template > outer HTML

● **beforeMount** 和 **mounted** 生命周期：

挂载前使用**{{msg}}**占位，这也是虚拟dom的优势，先把坑占了，到时候光放数据；

挂载后，**{{msg}}**被真实的数据替换了。

● **beforeUpdate** 和 **updated**

真实dom在更新前没有改变，更新后改变了

虚拟dom更新前就改变了

● **beforeDestroy** 和 **destroyed**

beforeDestroy都可以用，（**\$destroy**方法被调用的时候就会执行），一般在这里善后：清除计时器、清除非指令绑定的事件等等...）

destroyed就不可以操作了

● 业务场景中的应用：

created:进行ajax请求异步数据的获取、初始化数据

mounted:挂载元素dom节点的获取

nextTick:针对单一事件更新数据后立即操作dom

updated:任何数据的更新，如果要做统一的业务逻辑处理

watch:监听数据变化，并做相应的处理

十. 组件传值

```

<template>
  <div>
    <header-top></header-top>
    <child :parentToChild="parentValue"></child>
    <footer-bottom></footer-bottom>
  </div>
</template>

<script>
import ...
export default {
  name: "father-to-son",
  data(){
    return{
      parentValue: ["p1", "p2", "p3"]
    }
  },

```

```

<ul>
  <li v-for="li in valueFromFather">{{li}}</li>
</ul>
</div>
</template>

<script>
export default {
  name: "child",
  data(){
    return{
      "valueFromFather": [],
      // "valueToFather": "这是传给父组件的值"
    }
  },
  props: ['parentToChild'],
  created() {
    this.valueFromFather = this.parentToChild;
  },

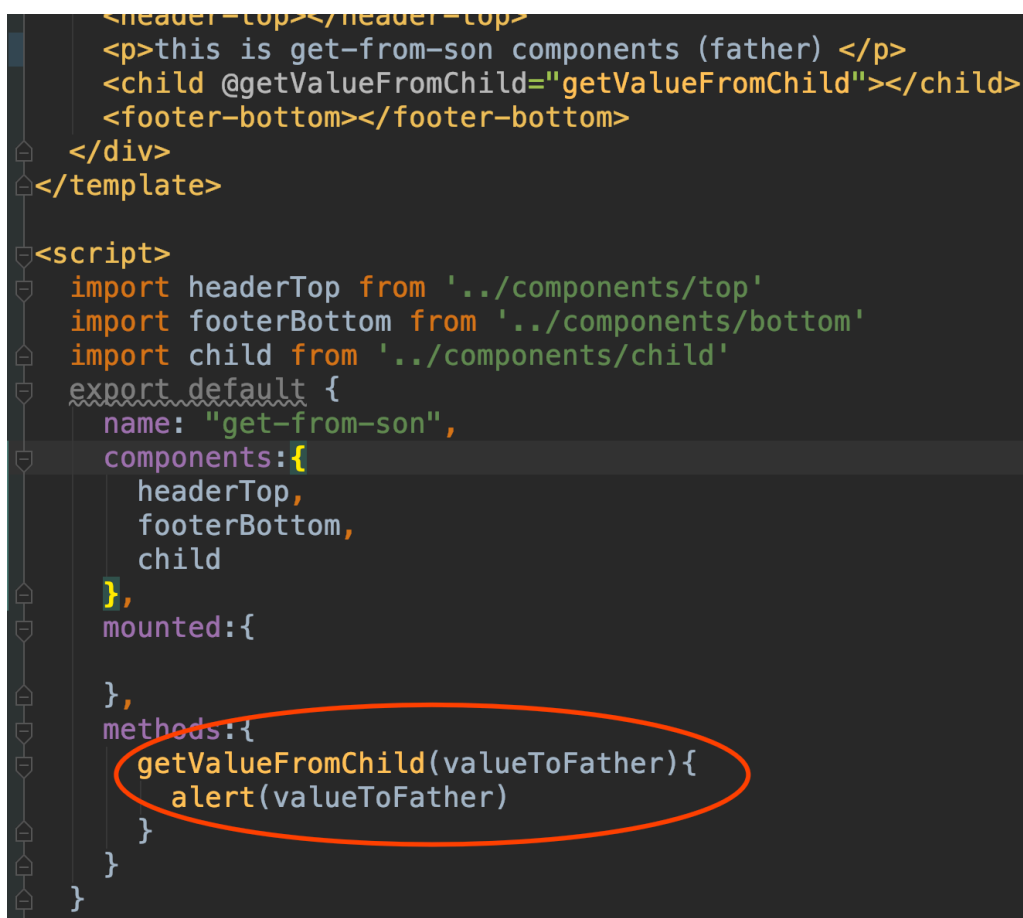
```

● 父子组件传值：

父组件v-bind一个值，在子组件中props接收，如：

- 子组件向父组件传值

通过定义事件，调用\$emit 方法，将值传过去，如：



```
<header-top></header-top>
<p>this is get-from-son components (father) </p>
<child @getValueFromChild="getValueFromChild"></child>
<footer-bottom></footer-bottom>
</div>
</template>

<script>
import headerTop from '../components/top'
import footerBottom from '../components/bottom'
import child from '../components/child'
export default {
  name: "get-from-son",
  components:{
    headerTop,
    footerBottom,
    child
  },
  mounted:{
  },
  methods:{
    getValueFromChild(valueToFather){
      alert(valueToFather)
    }
  }
}
```

- 兄弟组件传值：

通过事件总线的方式，调用eventBus.\$emit方法传值，\$eventBus.on方法接收值：，如：
assets文件夹下创建一个eventBus.js文件,A组件的值传给B组件

```
<div>
  {{textB}}
</div>
</template>

<script>
import EventBus from '../assets/eventBus'
export default {
  name: "childB",
  data(){
    return{
      textB:'this is componentB'
    }
  },
  created() {
    this.btnB()
  },
  methods:{
    btnB(){
      EventBus.$on("myFun", (message)=>{ //这里最好用箭头函数，不然this指向有问题
        this.textB = message
      })
    }
  }
}
```

十一.自定义事件：见子组件向父组件传值

十二.插槽

内容分发，slot元素作为承载内容分发到接口

十三.双向数据绑定

```

ts.vue x childSlot.vue
template>
<ul>
  <li v-for="todo in todos" :key="todo.id">
    <slot :todo="todo"><!--插槽prop 可以在父作用域中v-slot: default="value"一个值来定义我们提供的插槽prop的名字-->
      这是插槽默认值，即后备内容，父组件没有提供内容的时候显示这个，提供来则文本不显示
      //slot作为一个承载内容分发的接口
      //具名插槽指的是一个插槽有一个name值，多个插槽中会一一对应
      //Vue 实现了一套内容分发的 API，将 slot元素作为承载分发内容的出口。
    </slot>
  </li>
</ul>
</template>

script>
export default {
  name:"childSlot",
  props: {
    todos: {
      type: Array
    }
  }
}
/script>

components: {
  childSlot
},
created(){
  this.init();
},
methods:{
  init(){
    let self = this;
    this.$axios.get('../static/mock/slot.json')
      .then(function (response) {
        self.slotData = response.data.res
      })
      .catch(function (error) {
        console.log(error)
      })
  }
}

```

<p>{{message}}</p>

<input type="text" v-model="message" >

十四.动态组件切换

```
<template>
  <div id="dynamic-component">
    <button
      v-for="tab in tabs"
      v-bind:key="tab"
      v-bind:class="['tab-button', { active: currentTab === tab }]"
      v-on:click="changeTab(tab)"
    >{{ tab }}</button>
    <keep-alive>
      <component
        v-bind:is="currentTab"
        class="tab">
      </component>
    </keep-alive>
  </div>
</template>

<script>
import login from '../components/login'
import register from '../components/register'
export default {
  name: "dynamic-component",
  components: {
    login,
    register
  },
  data() {
    return {
      currentTab: 'login',
      tabs: ['login', 'register']
    }
  },
  methods: {
    changeTab(tab) {
      this.currentTab = tab
    }
  }
}
```

十五.keep-alive

vue的内置组件，能在切换过程中将状态保存在内存中，防止重复渲染DOM。

包裹动态组件时，会缓存不活动的组件实例，而不是销毁他们。

- prop:

Include , 匹配的组件被缓存

Exclude 匹配的组件都不被缓存

- Eg1:结合router，缓存部分页面：在router中设置meta信息，meta: {keepAlive:false}

- Eg2:B页面跳转到首页，首页需要缓存；C页面跳转到首页，首页不需要缓存：

首页路由：{path:'/',name:'A',component:A, meta:{keepAlive:true}}

B页面：methods:{},beforeRouterLeave(to, from, next){to.meta.keepAlive = true;}

B页面：methods:{},beforeRouterLeave(to, from, next){to.meta.keepAlive = false;}

- keep-alive生命周期钩子函数 activated deactivated

Activated 组件第一次渲染、每次keep-alive激活时候调用

Deactivated 组件被停用时调用

什么时候获取数据？

引入keep-alive的时候，页面第一次进入，钩子触发顺序是created、mounted、activated，退出出发deactivated，再次进入（前进或后退）时，只触发activated。

keep-alive之后页面模版第一次初始化解析成HTML片段后，再次进入不在重新解析而是读取内存中的数据。即只有当数据变化时才使用VirtualDOM进行diff更新，所以页面进入的数据获取应在在activated中也放一份，数据下载完毕手动操作DOM的部分也应该在activated中执行才生效。

所以，应该在activated中保留一份数据获取的代码，或者不要created部分，直接将created中的代码转移到activated中。

十六.懒加载的最终实现方案

1、路由页面以及路由页面中的组件全都使用懒加载

优点：（1）最大化的实现随用随载

（2）团队开发不会因为沟通问题造成资源的重复浪费

缺点：（1）当一个页面中嵌套多个组件时将发送多次的http请求，可能会造成网页显示过慢且渲染参差不齐的问题

2、路由页面使用懒加载，而路由页面中的组件按需进行懒加载，即如果组件不大且使用不太频繁，直接在路由页面中导入组件，如果组件使用较为频繁使用懒加载

优点：（1）能够减少页面中的http请求，页面显示效果好

缺点：（2）需要团队事先交流，在框架中分别建立懒加载组件与非懒加载组件文件夹

3、路由页面使用懒加载，在不特别影响首页显示延迟的情况下，根页面合理导入复用组件，再结合方案2

优点：（1）合理解决首页延迟显示问题

（2）能够最大化的减少http请求，且做其他路由界面的显示效果最佳

缺点：（1）还是需要团队交流，建立合理区分各种加载方式的组件文件夹

十七.Vuex

- 安装：cnpm install vuex --save

- 配置：Src 目录下新建store文件夹，下有index、mutations、actions三个js文件和modules文件夹

- 其中，index.js中：

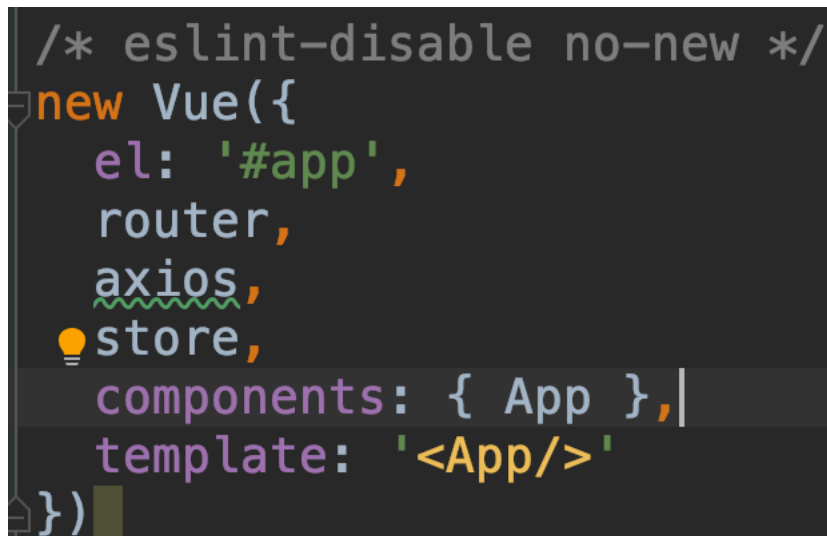

```
import Vue from 'vue'
import Vuex from 'vuex'
Vue.use(Vuex) //使用vuex
const store = new Vuex.Store({
  vuex实例
})
```

```
export default store //导出
```

- 在src 下 Main.js中引入store文件

```
import store from './store'
```

//把 store 对象提供给“store”选项，这可以把 store 的实例注入所有的子组件



```
/* eslint-disable no-new */
new Vue({
  el: '#app',
  router,
  axios,
  store,
  components: { App },
  template: '<App/>'
})
```

核心

● State

Store index.js中，声明一个state变量，赋值一个空对象给它，里面随便定义两个初始属性值。然后在实例化的vuex.store 中传入一个空对象，并将state扔在里面

```
import Vue from 'vue';
import Vuex from 'vuex';
Vue.use(Vuex);
const state={
  isShow:true
};
const store = new Vuex.Store( options: {
  state
});

export default store;
```

此时可以在任意组件中使用is.\$store.state.isShow获取isShow的值了。但这不是理想的方式，官方推荐 **getters**

● getters

和computed一样，实时监听state状态，代码如下：

```
const state={
  isShow:true
};
const getters = { //实时监听state值的变化(最新状态)
  isShow(state) { //方法名随意,主要是来承载变化的showFooter的值
    return state.isShow
  },
};
const store = new Vuex.Store({
  state,
  getters
});
export default store;
```

备注：mapGetters仅仅是将store中的state映射到局部计算属性

有了初始值，需要改变它，这就需要mutations了

- mutations，也是一个对象，里面放改变state的方法。具体的用法就是给里面的方法传入state 或是额外的参数，利用vue的双向数据驱动改变值。

```
const state={
  isShow:true
};
const getters = { //实时监听state值的变化(最新状态)
  isShow(state) { //方法名随意,主要是来承载变化的showFooter的值
    return state.isShow
  },
};
const mutations = {
  changeShow(state){
    state.isShow = !state.isShow
  }
};
const store = new Vuex.Store({
  state,
  getters,
  mutations
});
export default store;
```

备注：辅助函数mapMutations 将methods映射为store.commit,

备注：提交载荷 可提交额外的参数。store.commit(funcName, [item])

但是，mutations只是同步提交，不能异步提交。故引入actions

- actions

actions提交的是mutations，并不是直接变更状态。

```
const state={
  isShow:true
};
const getters={
  isShow(state){
    return state.isShow
  }
};
const mutations={
  showOrHideTemplate(state){
    state.isShow = !state.isShow;
  }
};
const actions={
  showOrHideTemplate(context){
    context.commit("showOrHideTemplate")
  }
};
export default {
  namespaced:true,
  state,
  getters,
  mutations,
  actions
}
```

外部组件调用actions

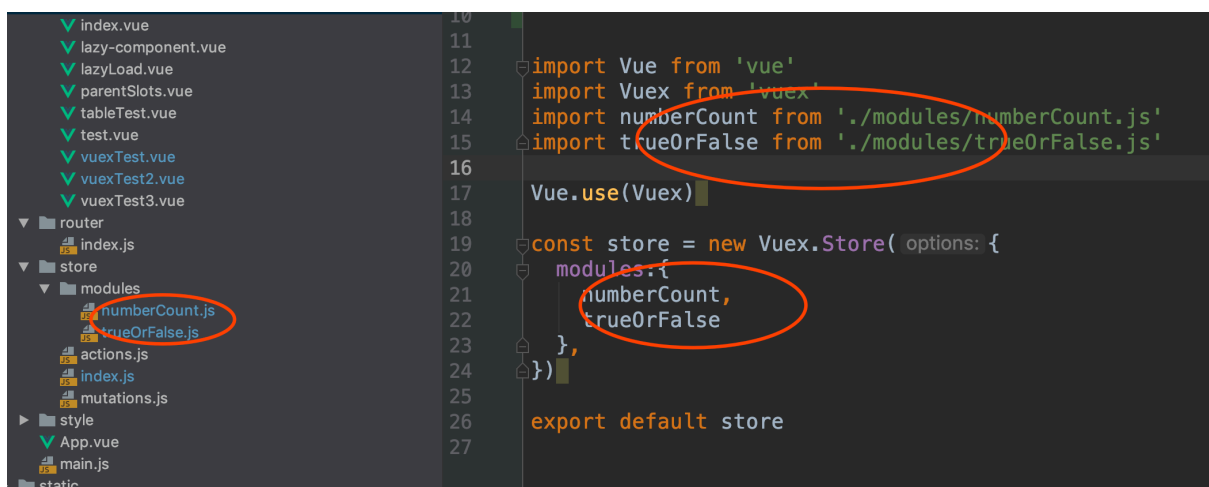
```
methods:{
  showOrHideTemplate(){
    store.dispatch("showOrHideTemplate");
  }
}
```

备注：actions是异步，返回的都是promise，所以可以链式调用then（）

● modules

状态管理中，可能有多种需求，则需要多个模块分别管理

则对应的目录结构和index修改为：



trueOrFalse中的代码：

```
const state={
  isShow:true
};
const getters={
  isShow(state){
    return state.isShow
  }
};
const mutations={
  showOrHideTemplate(state){
    state.isShow = !state.isShow;
  }
};
const actions={
  showOrHideTemplate(context){
    context.commit("showOrHideTemplate")
  }
};
export default {
  namespaced:true,
  state,
  getters,
  mutations,
  actions
}
```

调用时候，需要加上modules名字

```
methods:{
  showOrHideTemplate(){
    store.dispatch("trueOrFalse/showOrHideTemplate");
  }
}
```

2019年8月28日 星期三