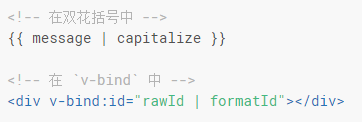
1. Vue
   1. filters:过滤器

Vue.js 允许你自定义过滤器，可被用于一些常见的文本格式化。过滤器可以用在两个地方：双花括号插值和 v-bind 表达式 (后者从 2.1.0+ 开始支持)。过滤器应该被添加在 JavaScript 表达式的尾部，由“管道”符号指示：





message是参数，capitalize是过滤函数.

rawid是参数，formatId是过滤函数.

过滤器可以串联.

* 1. created()

在实例创建完成后被立即调用。在这一步，实例已完成以下的配置：数据观测 (data observer)，属性和方法的运算，watch/event 事件回调。然而，挂载阶段还没开始，$el 属性目前不可见。

* 1. watch

当组件一个属性需要观测其状态变化时，用watch属性.属性a和b发生变化时，立刻调用相应函数.



* 1. vue.config

Vue.config 是一个对象，包含 Vue 的全局配置。可以在启动应用之前修改下列属性：

productionTip 设置为 false 以阻止 vue 在启动时生成生产提示

* 1. vue-devtools
     1. 使chrome插件生效

new Vue({

el: '#app',

router,

store,

template: '<App/>',

components: {App}

})

Vue.config.devtools = true; //要加上这句

ref:https://juejin.im/post/5be1430e518825170b1018f3

* 1. 编译打包概念
     1. npm install 安装依赖包

npm install --registry=https://registry.npm.taobao.org 如果安装的慢，则使用淘宝的镜像.

* + 1. npm run build 编译打包
    2. npm run dev 使用nodejs环境运行工程.
    3. vue-ci工具是vue的创建工程的工具，规范了目录结构，编译打包的配置文件.

ref:vue-cli生成的重要代码详解

* + 1. package.json

里面定义了脚本的位置.

"scripts": {

"dev": "webpack-dev-server --inline --progress --config build/webpack.dev.conf.js",

"start": "npm run dev",

"build": "node build/build.js",

"build:report":"npm\_config\_report=true node build/build.js",

"lint": "eslint --ext .js,.vue src",

"test": "npm run lint"

},

所以 npm run build命令中build就是上述脚本中的“build”，同理npm run dev 也在上述.

定义了依赖，需要查看vue的版本

"dependencies": {

"axios": "0.17.1",

"element-ui": "2.3.4",

"js-cookie": "2.2.0",

"normalize.css": "7.0.0",

"nprogress": "0.2.0",

"vue": "2.5.10",

"vue-router": "3.0.1",

"vuex": "3.0.1"

},

* + 1. src/main.js是主函数入口
    2. build.js

其中

process.env.NODE\_ENV = 'production'

const webpackConfig = require('./webpack.prod.conf')

这两个表示使用production环境的配置文件进行编译.

webpack(webpackConfig, 表示具体编译在webpackConfig中，

* + 1. baseWebpackConfig

webpack.dev.conf.js webpack.prod.conf.js 两个表示开发和生产环境配置，依赖base的配置文件，共性的在base配置中，prod配置中有

const webpackConfig = merge(baseWebpackConfig,

从而把base配置合并进来了.

* 1. process.env

node.js编译发布的环境变量。

在build.js文件中设置process.env.NODE\_ENV = 'production'

* 1. $mount

等价于创建实例中的el表达式，和一个html元素关联起来。

// 创建并挂载到 #app (会替换 #app)

new MyComponent().$mount('#app')

// 同上

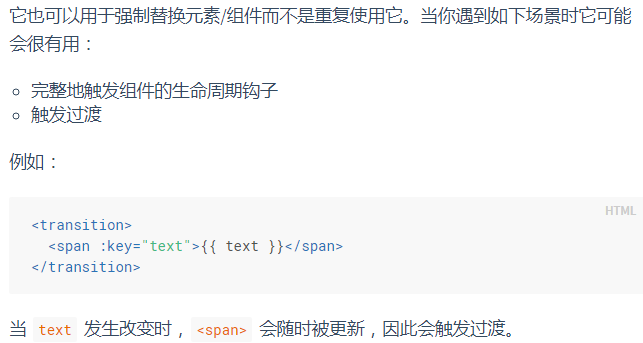
new MyComponent({ el: '#app' })

* 1. 特殊特性

以v-开头的是vue的指令，特殊特性不以v开头.

* + 1. key

有相同父元素的子元素必须有独特的 key。重复的 key 会造成渲染错误。



* + 1. ref和$refs

有的时候你仍可能需要在 JavaScript 里直接访问一个子组件。为了达到这个目的，你可以通过 ref 特性为这个子组件赋予一个 ID 引用。

<base-input ref="usernameInput"></base-input>

现在在你已经定义了这个 ref 的组件里，你可以使用：

this.$refs.usernameInput在js代码中访问这个base-input

* 1. v-bind 和v-on的缩写

v-bind:href缩写为:href,就是冒号开头。

v-on:click缩写为@click。

* 1. v-bind
     1. 通过数据与属性值绑定，数据的true和false操作控件class和style的属性是否具备该属性值。

<div v-bind:class="{ active: isActive }"></div>

上面的语法表示 active 这个 class 存在与否将取决于数据属性 isActive 的 truthiness.

通过compute计算数据的true和false；

<div v-bind:class="classObject"></div>

data: {

isActive: true,

error: null

},

computed: {

classObject: function () {

return {

active: this.isActive && !this.error,

'text-danger': this.error && this.error.type === 'fatal'

}

}

}

computed 的结果classObject绑定了this.isActive && !this.error，所以isActive变化了，classObject会自动更新.

关于'text-danger'，当key中包含特殊字符如横线“-”（不是下划线）， 空格等js不支持的变量时需要用引号。 因为vue会将"{}"中的内容应用到js中处理！

* + 1. 绑定style，直接给样式属性赋值.



* + 1. :model

<child :model="msg"></child>

:model是v-bind:model的缩写,只是给标签<child>绑定了model这个属性,且属性值和msg对象保持一致。

* 1. v-on
     1. $event

原始的 DOM 事件可以用特殊变量 $event 把它传入方法

<button v-on:click="warn('Form cannot be submitted yet.', $event)">

Submit

</button>

methods: {

warn: function (message, event) {

// 现在我们可以访问原生事件对象

if (event) event.preventDefault()

alert(message)

}

}

尽管我们可以在方法中轻松实现这点，但更好的方式是：方法只有纯粹的数据逻辑，而不是去处理 DOM 事件细节。为了解决这个问题，Vue.js 为 v-on 提供了事件修饰符。之前提过，修饰符是由点开头的指令后缀来表示的.

<!-- 阻止单击事件继续传播 -->

<a v-on:click.stop="doThis"></a>

* + 1. 事件修饰符

修饰符如下：

.stop - 调用 event.stopPropagation()。事件不向外界传播，或者冒泡.事件冒泡（dubbed bubbling）：与事件捕获恰恰相反，事件冒泡顺序是由内到外进行事件传播，直到根节点。

.prevent - 调用 event.preventDefault()。取消事件的默认动作，如动作submit，则提交动作被阻止.

.capture - 添加事件侦听器时使用 capture 模式。

.self - 只当事件是从侦听器绑定的元素本身触发时才触发回调。

.{keyCode | keyAlias} - 只当事件是从特定键触发时才触发回调。

.native - 监听组件根元素的原生事件。

.once - 只触发一次回调。

.left - (2.2.0) 只当点击鼠标左键时触发。

.right - (2.2.0) 只当点击鼠标右键时触发。

.middle - (2.2.0) 只当点击鼠标中键时触发。

.passive - (2.3.0) 以 { passive: true } 模式添加侦听器

* + 1. 按键修饰符

<!-- 只有在 `key` 是 `Enter` 时调用 `vm.submit()` -->

<input v-on:keyup.enter="submit">

keyup是按键，enter限制具体按键，enter还替代为如下：

.enter

.tab

.delete (捕获“删除”和“退格”键)

.esc

.space

.up

.down

.left

.right

* + 1. 系统修饰符

.ctrl

.alt

.shift

<!-- Ctrl + Click -->

<div @click.ctrl="doSomething">Do something</div>

* + 1. v-on="$listeners"

vm.$listeners

有了这个 $listeners 属性，你就可以配合 v-on="$listeners" 将所有的事件监听器指向这个组件的某个特定的子元素

* 1. v-model
     1. 原理

v-bind将数据对象绑定到html的标签属性上，但是标签属性的唯一修改来源是依赖数据对象，ui交互不会修改class style等标签属性的.vue的核心就是不直接操作DOM，通过数据对象去操作DOM.但是对于表单输入标签，ui交互会产生新数据，需要反向从标签属性通知数据对象，所以使用v-model进行数据绑定.将表单输入的值传递给数据对象.



v-model编译后，绑定了input事件，input输入的值通过event事件抛出来并赋值给searchText变量.

* + 1. 应用场景

表单输入的双向数据绑定

自定义组件，输入内容的绑定，见《实现v-model参数双向绑定》

* + 1. 各种表单用法

表单 <input>、<textarea> 及 <select> 元素上创建双向数据绑定.

你应该通过 JavaScript 在组件的 data 选项中声明初始值.

v-model 在内部使用不同的属性为不同的输入元素并抛出不同的事件：

text 和 textarea 元素使用 value 属性和 input 事件；

checkbox 和 radio 使用 checked 属性和 change 事件；

select 字段将 value 作为 prop 并将 change 作为事件。

例子

<div id='example-3'>

<input type="checkbox" id="jack" value="Jack" v-model="checkedNames">

<span>Checked names: {{ checkedNames }}</span>

</div>

new Vue({

el: '#example-3',

data: {

checkedNames: [] //数据是多值，如果数据是单值，使用checkedNames: "";

}

})

就是data选项的checkedNames作为值和界面显示绑定。

* + 1. 和v-bind区别

1. v-bind是数据绑定，没有双向绑定效果，但不一定在表单元素上使用，任何有效元素上都可以使用；

2. v-model是双向绑定，基本上只用在表单元素上；

3. 当v-bind和v-model同时用在一个元素上时，它们各自的作用没变，但v-model优先级更高，而且需区分这个元素是单个的还是一组出现的。

* 1. 组件
     1. data必须为对象

Vue.component('button-counter', {

data: function () { //data后面必须是函数，不能是对象（如data:count:0），否则多个组件button-counter的实例会共享count值，点击按钮会显示所有对象点击次数，而不是该按钮的点击次数。

return {

count: 0

}

},

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

})

* + 1. 给组件传参数

Vue.component('blog-post', {

props: ['title'], //通过props属性从外部传递参数。

template: '<h3>{{ title }}</h3>'

})

使用组件

<blog-post title="My journey with Vue"></blog-post>

如果参数是变量，则可以动态传递参数，posts是输入参数数组.post.title相当于上面的"My journey with Vue"



如果参数很多，可以将参数设置为post对象，包含title、content等属性.

* + 1. 组件和外部事件交互

组件内使用$emit(‘eventName’)发射事件。

外部使用v-on：eventName=“事件处理代码，也可以是处理函数名”。

事件的参数如0.1数字，在"$emit('enlarge-text', 0.1)"

在外部可以使用$event表示参数值

* + 1. 实现v-model参数双向绑定

父组件中在子组件上使用v-model，默认会用value的prop来接受父组件v-model绑定的值，然后子组件通过input事件将更新后的值传递给父组件.应该是各种组件选择框，radio等都是抛出自定义的input事件，



* + 1. 注册组件的方式。

1. 全局注册

Vue.component('my-component',{

template: '#myComponent',

data(){

return {}

},

methods: {

}

})

1. 局部注册

const componentA = { //直接定义一个常量

template: '#a',

data(){

return { }

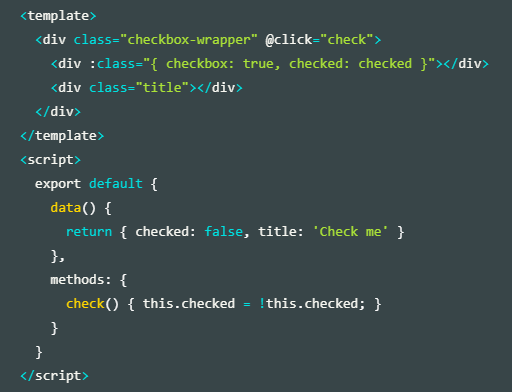
}

//...

}

1. 单文件组件,最常用.

html和js分离，非常清晰.



ref：vue常用创建组件几种方式总结

* + 1. slot插槽

组件定义时，使用<slot>标签表示占位符，使用组件时，组件标签之间的内容自动替换了<slot>的内容.

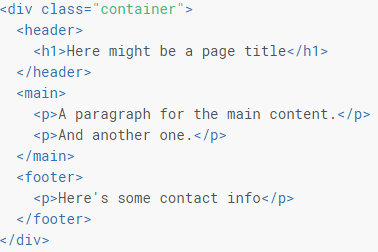
组件定义时实现多个插槽，通过<slot>标签的name属性区分，如图header和footer插槽，一个不带 name 的 <slot> 出口会带有隐含的名字“default”



在向具名插槽提供内容的时候，我们可以在一个 <template> 元素上使用 v-slot 指令，并以 v-slot 的参数的形式提供其名称



渲染结果，没有用v-slot指定的出现的默认插槽位置.



插槽传参数.

* 1. svg-icon

<svg-icon icon-class="user"/>使用icon，需要在main.js中引入import '@/icons' // icon

* 1. vm

vue的文档中经常会使用 vm (ViewModel 的缩写) 这个变量名表示Vue 实例

1. vuex
   1. 全局概念图



注意关键词，component和action之间是用dispatch函数，action和mutations之间用commit函数，state只能是mutation去操作。

* 1. 创建store对象

const store = new Vuex.Store({ //存储对象，

state: {

count: 0

},

mutations: { //修改state函数

increment (state) {

state.count++

}

}

})

// 如果在模块化构建系统中，请确保在开头调用了 Vue.use(Vuex)

* 1. 在组件中用this.$store访问数据

需要将store在根组件中注册

const app = new Vue({

el: '#app',

// 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

store,

components: { Counter },

template: `

<div class="app">

<counter></counter>

</div>

`

})

读数据方式：this.$store.state.count

建议使用getter函数读取，对变量的读取需要封装成getter函数。

* 1. getter

getters: {

doneTodos: state => {

return state.todos.filter(todo => todo.done)

}

}

可以做一些过滤操作，state值没变，则getter后的结果都是缓存的，不用每次计算，和computed一个概念。

* + 1. mapGetters

辅助函数仅仅是将 store 中的 getter 映射到局部计算属性,如下，computed的结果包含doneTodosCount成员变量，值和getter中doneTodosCount()函数返回值一致.



* 1. mutation

const store = new Vuex.Store({

state: {

count: 1

},

mutations: {

increment (state) {//默认带state参数

// 变更状态

state.count++

}

}

})

修改store必须使用mutations域定义的操作，调用统一使用commit函数

store.commit('increment')；

store.commit('increment', 10)；//操作函数带参数情况，默认带state参数。

* 1. action
     1. 支持异步，

actions: {

increment (context) {

context.commit('increment')

}

}

Action 函数接受一个与 store 实例具有相同方法和属性的 context 对象，因此你可以调用 context.commit 提交一个 mutation，或者通过 context.state 和 context.getters 来获取 state 和 getters。

* + 1. 简便写法

actions: {

increment ({ commit }) {

commit('increment')//等价于context.commit('increment')，因为context用{ commit }形式传递了。

}

}

上述increment函数简写如上。用{commit}表示context对象，则调用context的函数时，不需要注明context实例名，直接操作其成员commint即可。

* + 1. 分发dispatch，调用action的方式

// 以载荷形式分发

store.dispatch('incrementAsync', {

amount: 10

})

// 以对象形式分发

store.dispatch({

type: 'incrementAsync',

amount: 10

})

在组件中调用action，this.$store.dispatch('xxx')

* + 1. 异步调用

action执行后，如何知道是否结束，结束后进一步处理，使用promise。

actions: {

actionA ({ commit }) {

return new Promise((resolve, reject) => {

setTimeout(() => {

commit('someMutation')

resolve() //resolve()就是处理结束，外部调用进一步处理就是then这个lamda表达式。

}, 1000)

})

}

}

调用

store.dispatch('actionA').then(() => {

// ...

})

* + 1. promise

const promise = new Promise(function(resolve, reject) {

// ... some code

if (/\* 异步操作成功 \*/){

resolve(value);

} else {

reject(error);

}

});

调用时

promise.then(function(value) {

// success //由上面resolve(value)调用

}, function(error) {

// failure//由上面reject（error）调用

});

* 1. 模块

模块内部有自己的store，getter，mutations，action。其中getter和action的参数多个局部state和根state。

getters: {

sumWithRootCount (state, getters, rootState) {

return state.count + rootState.count

}

}

actions: {

incrementIfOddOnRootSum ({ state, commit, rootState }) {

if ((state.count + rootState.count) % 2 === 1) {

commit('increment')

}

}

}

* 1. 项目典型解构

├── index.html

├── main.js

├── api

│ └── ... # 抽取出API请求

├── components

│ ├── App.vue

│ └── ...

└── store

├── index.js # 我们组装模块并导出 store 的地方

├── actions.js # 根级别的 action

├── mutations.js # 根级别的 mutation

└── modules

├── cart.js # 购物车模块

└── products.js # 产品模块

1. vue router
   1. 嵌套路由

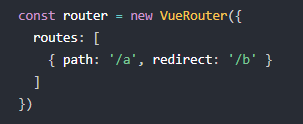


当匹配path时，使用组件Layout，

当进一步匹配到/dashboard时，使用父组件Layout，并且Layout中router-view的内容由dashboard/index这个组件填充.

注意，当路径仅为/时，redirect重定向到了/dashboard，所以还是匹配了children的路径，调用了子组件.

* 1. 重定向

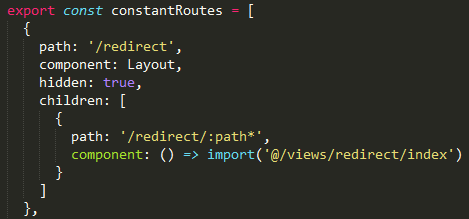


redirect应该在最低一级才有用，如果有children，则运用到children才有效，在父目录下面的redirect没有用.

* 1. 路由懒加载

在路由表中延迟加载组件.

只有匹配路由redirect/:path\*时，才会导入组件定义.



* 1. push

当你点击 <router-link> 时，这个方法会在内部调用，所以说，点击 <router-link :to="..."> 等同于调用 router.push(...)

router.push(location, onComplete?, onAbort?)

// 字符串

router.push('home')

// 对象

router.push({ path: 'home' })

// 命名的路由

router.push({ name: 'user', params: { userId: '123' }})

// 带查询参数，变成 /register?plan=private

router.push({ path: 'register', query: { plan: 'private' }})

注意：如果提供了 path，params 会被忽略

* 1. 命名路由

给路由一个名词，从而通过名词直接调用

const router = new VueRouter({

routes: [

{

path: '/user/:userId',

name: 'user',

component: User

}

]

})

调用时

router.push({ name: 'user', params: { userId: '123' }})

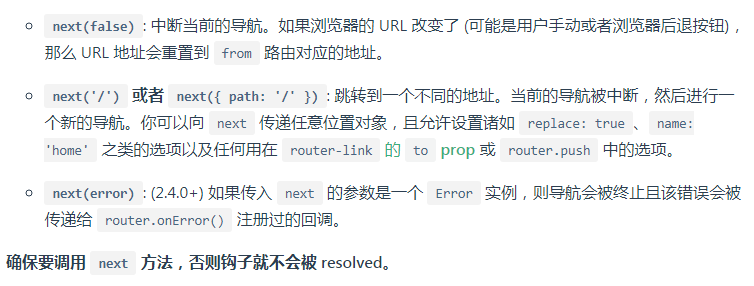
* 1. 判断是否登录进行跳转

在所有路径处理前加钩子，没登录就跳转登录界面

使用 router.beforeEach 注册一个全局前置守卫.







1. Element Ui
   1. this.$message('submit!')

Element\_ui 为 Vue.prototype 添加了全局方法 $message。因此在 vue instance 中可以采用本页面中的方式调用 Message.

只要import ElementUI from 'element-ui'，就是全局导入.

* 1. el-form

在 Form 组件中，每一个表单域由一个 Form-Item 组件构成，表单域中可以放置各种类型的表单控件，包括 Input、Select、Checkbox、Radio、Switch、DatePicker、TimePicker。

* + 1. rules

Form 组件提供了表单验证的功能，只需要通过 rules 属性传入约定的验证规则，并将 Form-Item 的 prop 属性设置为需校验的字段名即可.

* + 1. 行内表单

设置 inline 属性可以让表单域变为行内的表单域

<el-form :inline="true"

* + 1. label-position

通过设置 label-position 属性可以改变表单域标签的位置，可选值为 top、left，当设为 top 时标签会置于表单域的顶部。如果值为 left 或者 right 时，则需要设置 label-width

* + 1. label-width

如果不设置，则标签长度无限长，后面的输入框就换到下一行了.

1. html
   1. span

使用 <span> 来组合行内元素，以便通过样式来格式化它们

主要给子元素提高统一的外围样式.

1. CSS
   1. box-sizing属性

取值为border-box表示，width 和 height 属性包括内容，内边距和边框，但不包括外边距。

最佳实践，取值为border-box，减少margin和padding对尺寸的影像.

取值为content-box表示，width 和 height 属性只包括内容.

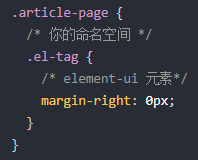
取值为inherit，表示从父元素继承

* 1. 居中对齐
     1. 标签居中 margin: auto 两边都是auto，则左右平均分配
     2. 文字

text-align: center

* 1. 覆盖其他组件的样式

比如覆盖element-ui的样式，现在我们来说说怎么覆盖 element-ui 样式。由于 element-ui 的样式我们是在全局引入的，所以你想在某个页面里面覆盖它的样式就不能加 scoped，但你又想只覆盖这个页面的 element 样式，你就可在它的父级加一个 class，用命名空间来解决问题。



* 1. position属性

Position属性四个值：static、fixed、absolute和relative的区别和用法



* + 1. relative

定位为relative的元素脱离正常的文本流中，但其在文本流中的位置依然存在.

无论父级存在不存在，无论有没有TRBL（top right bottom left），均是以父级的左上角进行定位，但是父级的Padding属性会对其影响。

top和left优先于bottom和right，

* + 1. absolute

定位为absolute的层脱离正常文本流，但与relative的区别是其在正常流中的位置不再存在.

若想把一个定位属性为absolute的元素定位于其父级元素内，只有满足两个条件：

　 第一：设定TRBL

　第二：父级设定Position属性为非static.

记得设置top:0; left:0，否则没有脱离文档流，此时absolute无效，变为了relative.

* 1. display属性
     1. block，inline和inline-block概念和区别





* 1. 选择器

选择器的种类可以分为三种：标签名选择器、类选择器和ID选择器

1.id选择器（#myid）

2.类选择器（.myclassnahme）

3.标签选择器（div,h1,p）

4.相邻选择器（h1+p）

5.子选择器（ul > li）

下一级，直接后代，是后代选择器的特殊情况

6.后代选择器（li a）

7.通配符选择器（\*）

8.属性选择器（a[rel="external"]）

具有属性rel，并且属性值为“external”的

9.伪类选择器（a:hover, li:nth-child）

:hover表示当鼠标悬停在a标签上时

* + 1. 写法

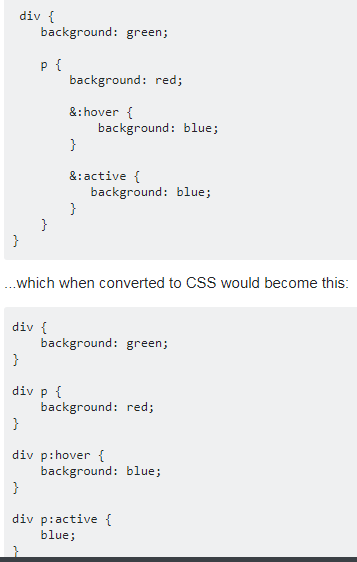
优先级高的在前面，

id选择器少用，

优先使用class选择器

* + 1. 特殊符号

& 代表父选择器.



* 1. scss/sass
     1. @include

类似调用一个函数，使用@mixin将类似的样式封装为函数，函数也是可以带参数的.

@mixin clearfix {

&:after {

content: "";

display: table;

clear: both;

}

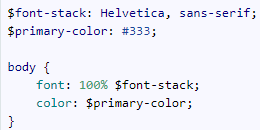
}

调用时

@include clearfix;即可

* + 1. &

使用$符号来创建一个变量.引用时，直接调用该变量值.



1. JS
   1. eslint格式要求
      1. 语法规则在每个项目的.eslintrc.js文件中
      2. 逗号后面加上空格，逗号前不加空格.

optionData: [{

label: 'one',

value: 'oneV'

}, {

label: 'two',

value: 'twoV'

}]

一个对象的格式如上.

* 1. 或操作符|| 与操作符&&

var aa = a1 && a2 && a3; //3个条件连接起来判断真假，都为真 则返回最后一个值。

alert(aa); //返回a3

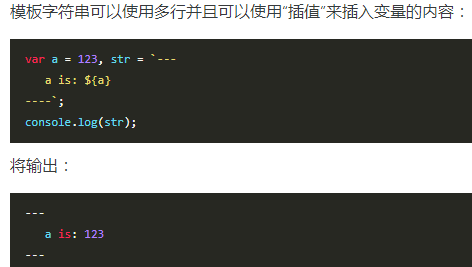
有一个假的返回假的那个

或操作符|| ，从左到右只要发现有一个有意义则整个表达式都有意义，并返回这个有意义的值，如果都是没意义的值，则返回最后那个值。

* 1. 模板字符串

`反单引号，也是用来表示字符串，是es6里面增强功能，

模板字符串` `,中嵌入变量，需要将变量名写在${}之中。



* 1. cookie
     1. domain

cookie在浏览器中存储了一个键值对，当页面当前路径与某一条domain匹配，则向服务器请求都会带上它.

domain表示的是cookie所在的域，默认为请求的地址，如网址为www.test.com/test/test.aspx，那么domain默认为www.test.com。而跨域访问，如域A为t1.test.com，域B为t2.test.com，那么在域A生产一个令域A和域B都能访问的cookie就要将该cookie的domain设置为.test.com；如果要在域A生产一个令域A不能访问而域B能访问的cookie就要将该cookie的domain设置为t2.test.com。

Domain为设置Cookie的有效域(即.com结束，后面具体的文件路径不限制，通过path限制)，Path限制有效路径，如以下几种情况：

1.最大可能的作用域 yourdomain.com所有角落（设置Cookie.Domain = "yourdomain.com"，效果为yourdomain.com域下的所有主机的所有位置都可以

取到这个Cookie)

2.域下的某主机 www.yourdomain.com（设置Cookie.Domain = "www.yourdomain.com"，效果为www主机下所有位置可以取)

3.某主机的某个文件夹 www.yourdomain.com/bbs (Cookie.Domain ="www.yourdomain.com" ; Cookie.Path = "/bbs"，www主机的bbs下可以取)

* + 1. HttpOnly:

告知浏览器不允许通过脚本document.cookie去更改这个值，同样这个值在document.cookie中也不可见。但在http请求张仍然会携带这个cookie。注意这个值虽然在脚本中不可获取，但仍然在浏览器安装目录中以文件形式存在。这项设置通常在服务器端设置。

* + 1. secure:

安全标志，指定后，只有在使用SSL链接时候才能发送到服务器，如果是http链接则不会传递该信息。就算设置了secure 属性也并不代表他人不能看到你机器本地保存的 cookie 信息，所以不要把重要信息放cookie就对了.

* 1. require和module.exports

node模块中，我们采用的是CommonJS规范，使用require引入模块，使用module.exports导出接口。

es6中则引入模块，使用export和import命令。

require的使用非常简单，它相当于module.exports的传送门，module.exports后面的内容是什么，require的结果就是什么，对象、数字、字符串、函数……再把require的结果赋值给某个变量，相当于把require和module.exports进行平行空间的位置重叠。

a．js文件中

const \_import = require('./\_import\_' + process.env.NODE\_ENV)

\_import\_production.js文件

module.exports = file => () => import('@/views/' + file + '.vue')

则

\_import = file => () => import('@/views/' + file + '.vue') //这是两层调用的lamda表达式

import使用方式 \_import('login/index') 等价于import('@/views/ login/index.vue')

比较：

require是运行时，

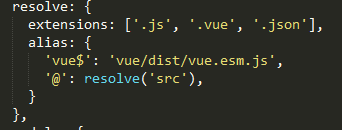
import是编译时处理。

* 1. import一个文件夹

import { Navbar, Sidebar, AppMain } from '@/views/layout/components'

当components表示文件夹时，导出定义在该文件夹下面的index.js文件中，这是nodejs的规范.

@表示起始目录，在webpack.base.conf.js文件中定义，此处表示src目录.



* 1. import()函数

因为import关键词是静态的，为了实现require的动态加载，es6引入了import(xxx)函数，而且是异步加载括号内容xxx.

import()类似于 Node 的require方法，区别主要是前者是异步加载，后者是同步加载。

import()返回一个 Promise 对象。所以可以import().then()接着处理.

* 1. const

创建一个常量，该引用不可指向其他对象。

const cars = ["Saab", "Volvo", "BMW"];

cars = ["Toyota", "Volvo", "Audi"]; // 错误

// 修改元素，修改里面内容是可以的。

cars[0] = "Toyota";

和var关键词互斥，var创建的变量是可变的。

let关键词和const类似，但是let可以创建时不进行初始化，const不行。

* 1. <script>标签
     1. 所有<script>元素都会按照它们在页面中出现的先后顺序依次被解析。在不使用 defer 和async 属性的情况下，只有在解析完前面<script>元素中的代码之后，才会开始解析后面<script>元素中的代码。
     2. 由于浏览器会先解析完不使用 defer 属性的<script>元素中的代码，然后再解析后面的内容，所以一般应该把<script>元素放在页面最后，即主要内容后面， </body>标签前面。
     3. 使用 defer 属性（延迟属性）可以让脚本在文档完全呈现之后再执行。延迟脚本总是按照指定它们的顺序执行。
     4. 使用 async 属性可以表示当前脚本不必等待其他脚本，也不必阻塞文档呈现。不能保证异步脚本按照它们在页面中出现的顺序执行。
  2. 数据类型

对变量或值调用 typeof 运算符将返回下列值之一：

* + 1. undefined - 如果变量是 Undefined 类型的。

var message; // 这个变量声明之后默认取得了 undefined 值

// 下面这个变量并没有声明

// var age

alert(typeof message); // "undefined"

alert(typeof age); // "undefined"

结果表明，对未初始化和未声明的变量执行 typeof 操作符都返回了 undefined 值；

即便未初始化的变量会自动被赋予 undefined 值，但显式地初始化变量依然是明智的选择。如果能够做到这一点，那么当 typeof 操作符返回"undefined"值时，我们就知道被检测的变量还没有被声明，而不是尚未初始化。上面var message;未初始化是不好的习惯，通过typeof无法区分message是未创建该变量还是未创建了未初始化。

* + 1. boolean - 如果变量是 Boolean 类型的
    2. number - 如果变量是 Number 类型的
    3. string - 如果变量是 String 类型的 ，字符串就是基本类型，不是引用类型。字符串对象是Object类型，即通过：str = new string（“abc”）；创建的对象，typeof（str）是object类型。
    4. object - 如果变量是一种引用类型或 Null 类型的，Null是一种类型，只有一个值null，但是typeof把null返回为object表示空对象，也是合理的。
    5. 确定对象的具体类型使用instanceof

判断规则，判断左值是不是右值的实例对象（子类）

如：cat instanceof Animal;

判断如下：

var L = cat.\_\_proto\_\_;

var R = Animal.prototype;

if(L === R)

return true;

用处：判断一个对象是不是数组

var a = [];

console.log(a instanceof Array); // true

特例：Array是对象，但是不同框架可能有多个Array的实现，A实现的实例 instanceof B实现，会返回false，但是A实现的实例是数组对象，所以需要Array.isArray(A实现的实例)函数判断是不是数组对象。

* 1. 语法基础
     1. isFinite

Number.MIN\_VALUE 表示5e-324；

Number.MAX\_VALUE 表示1.7976931348623157e+308；

超过范围，数字用Infinity，该值不能用于计算，所以使用isFinite判断能不能计算。

* + 1. 字符串转数字

有 3 个函数可以把非数值转换为数值： Number()、 parseInt()和 parseFloat()。第一个函数，即转型函数 Number()可以用于任何数据类型（不建议使用），而另两个函数则专门用于把字符串转换成数值。

* + 1. 数字转字符串

使用toString（），null类型转为“null”，undefined类型转换为“undefined”；

* + 1. 字符串比较

按照顺序比较字符串编码，区分大小写，所以使用.toLowerCase()函数转为小写再比较。

如果“23”>3 返回true，因为先将字符串转换为数值；

“23”>“3”返回false，因为字符串中2的编码小于3的编码；

* + 1. 相等判断

== 是先将两个比较对象转换为同一个类型，再比较。类型优先级 Number 》String 》Boolean，两个类型不一致，向高优先级转换。

null和undefined不进行类型转换为同一个类型，两者表示Null类型和Undefined类型，==比较时相等。===比较时不等。

Boolean类型true转换为1，所以与2比较是不想的。

* 1. 函数
     1. arguments

代表函数参数对象。未赋值参数为undefined类型。

* + 1. 函数参数传递

基础数据类型，传递的值。

引用类型，传递的是指针地址，修改对象的成员值，函数内外指针指向同一块内存，该对象均变化。直接给函数参数赋值，则让函数内指针指向新内存区，函数外指针指向的内存区不变。

和java的引用对象在函数内外的关联一致。

* + 1. 没有重载

因为函数的描述不包括参数字段，仅包括函数名，无法重载。

* + 1. 函数声明和表达式

function addSomeNumber(num){

return num + 100;

} 这是函数声明，会扫描所有声明，所以函数声明在执行后面有效

var addSomeNumber = function (num){

return num + 100;

};这是函数表达式，没解析到这里，不会创建函数对象。

* + 1. Callee

function factorial(num){

if (num <=1) {

return 1;

} else {

return num \* arguments.callee(num-1)

}

}

Callee表示拥有这个 arguments 对象的函数

* + 1. Call和apply函数

让方法和对象this解耦合，方法执行时传递不同的对象。

* 1. 作用域

如果初始化变量时没有使用 var 声明，该变量会自动被添加到全局环境；

没有块级作用域

window.color，全局作用域为window，通过这个可以访问全局对象color；

* 1. 引用类型
     1. 创建对象的另一种方式

var person = {

name : "Nicholas",

age : 29

};

* 1. Prototype和类定义

类定义中的函数，作为Prototype的字段，所有实例共享。

类定义中的成员，不能作为Prototype字段，否则A实例修改了，B实例也受到影响。

定义一个类。

function SuperType(name){

this.name = name;

this.colors = ["red", "blue", "green"];

}

SuperType.prototype.sayName = function(){

alert(this.name);

};

function SubType(name, age){

//继承属性

SuperType.call(this, name);

this.age = age;

}

//继承方法

SubType.prototype = new SuperType();

SubType.prototype.constructor = SubType;

SubType.prototype.sayAge = function(){

alert(this.age);

};

上述代码是类SubType的定义，包括基类属性name，colors;基类方法sayName，

派生类属性age。

var instance1 = new SubType("Nicholas", 29);

instance1.colors.push("black");

alert(instance1.colors); //"red,blue,green,black"

instance1.sayName(); //"Nicholas";

instance1.sayAge(); //29

var instance2 = new SubType("Greg", 27);

alert(instance2.colors); //"red,blue,green"

instance2.sayName(); //"Greg";

instance2.sayAge(); //27

* 1. 块级作用域

Js中块不是作用域，一些语句块为了不污染全局空间，可以利用function创建块级作用域

(function(){

//这里是块级作用域，匿名的函数定义并且最后加上（）进行了执行。

})();

* 1. Down is old content=========================================================
  2. $.Ajax()

$.ajaxSetup({}) 可以设置ajax的默认初始值。

$.ajax({}) 选项

Datatype :返回的数据格式，如json、text等

error:function(xhr, message, error) message是jquery的状态码，值为error时表示http错误，具体从xhr的返回状态去判断。

Complete： ajax完成之后就，调用玩success或者error回调之后，会调用该函数。

Anysc：默认为true，即异步，想变为同步，等待返回再结束ajax，则改为true。

报bad request 400 错误

Js中的$.ajax的data的参数类型，与controler中的java参数不一致（data的变量为undefined也会造成类型不一致），或者变量名不一致

* 1. 复制一个对象

使用target = $.extend({},src); 只复制第一层；

target = $.extend(true,{},src);深复制；

jquery复制dom树。

Var element = $(“selector”).clone(); 选中的是所有匹配js变量，包括符合selector的dom对象及上下文，在必须接收dom对象作为参数的场合，需要element[0]来取对应的dom对象。如果一直使用jquery函数则直接对js变量进行操作，当然需要$(element);

* 1. 页面操作

刷新：history.go(0) location.reload() 页面前进history.go()

* 1. 对象属性的访问

成员函数访问自身属性

**var** test ={

name1:"1",

fun:**function**(){

**this**.name1 = "2";//需要加上this关键字

}

}

test.fun();

对象访问自身属性

对象属性可以通过get，set函数访问：

Obj.get(“属性名”)；

遍历所有属性

For(var I in obj){

Obj[i]；i是属性名称

}

判断属性是否在对象中

if (name in layerSet) {

return layerSet[name];

}

综上所述，对象可以当一个map使用

判断是否在数组中，用jquery的$.inArray(id,ary) 如果在，则返回索引，不在返回-1；

注意：id需要和ary的元素类型一致，如果id为Number类型，ary里面为string类型，则无法找到，处理方法：1、自己遍历ary，利用==号可以比较数字和字符串的特点，2、转换其中一个的数据类型。

* 1. $(function(){}

是 $(document).ready(function(){})的缩写，就是文档准备好之后再执行；openlayer3的地图创建必须在该函数中，否则文档未创建好，对文档操作会出现异常行为。

* 1. 函数对象、普通对象 原型对象(prototype) 原型链

通过内置Function()函数创建的为函数对象，表现在两种创建方式：

fun(){}

var f = function(){}

两种，它们内部均调用Function函数创建。

其他方式为普通对象。

函数对象才具备原型对象(prototype)

函数对象有个成员：prototype和\_\_proto\_\_;

prototype主要用于继承使用，如：

fun.prototype.member1 = “1”;

任何对象都有\_\_proto\_\_,new返回的实例的\_\_proto\_\_指向函数对象的 prototype。函数对象的 prototype中有成员\_\_proto\_\_，指向更上一级的prototype，从而形成原型链。函数对象的 prototype中还有成员constructor指向函数对象本身

则fun的实例对象(obj = new fun();此处fun()也称为构造器函数)，此时obj.\_\_proto\_\_即为fun.prototype，则obj可以拥有member1成员。 prototype中的属性只是给实例对象（或者说是子类使用的一组属性），fun本身无法调用(fun.member1不存在)；

* 1. new关键字

例子：

function Animal(name){this.name = name；}

Animal.color = “black”;

Animal.prototype.say = function(){consol.log(this.name);}

cat = new Animal(“cat”);

其中new发生来什么？伪代码如下：

new Animal(“cat”) = {

var obj={};

obj.\_\_proto\_\_ = Animal.prototype;//如果return时返回obj，则cat成

//为Animal的实例对象（即子类）

var result = Animal.call(obj,”cat”);//obj拥有来name属性

return result === “object”? Result:obj;//正常构造器函数均返回obj

}

实例对象cat的原型链：cat.\_\_proto\_\_ = Animal. Prototype

Animal. Prototype.\_\_proto\_\_ = Object. Prototype

Object. Prototype.\_\_proto\_\_ =null

函数对象Animal的原型链：Animal.\_\_proto\_\_ = Function. Prototype

Function. Prototype.\_\_proto\_\_= Object. Prototype

普通对象的原型链：var tmp = {}; var tmp\_obj = new Object();

两者一样， tmp.\_\_proto\_\_ = Object. Prototype

Object. Prototype.\_\_proto\_\_ =null

ref：js核心系列：理解new的运行机制

* 1. 多重继承

多个函数对象可以通过原型链实现多重继承，继承成员函数通过goog.inherits(child,parent),继承父类构造函数中定义的成员通过goog.base()，

ref:google closure--继承模块一:goog.inherits();

自己的实现

var baseClass = function(id){

this.id = id;

}

baseClass.prototype.setid = function(id){this.id = id;}

var derivedClass = function(id,name){

baseClass.call(this,"derived");//让基类成员变为派生类成员

this.name = name;

}

derivedClass.prototype.setName = function(name){this.name= name;}

derivedClass.prototype.\_\_proto\_\_ = baseClass.prototype;

var baseObj = new baseClass("baseObj");

var derivedObj = new derivedClass("derivedObj","derived");

baseObj.setid("success");

derivedObj.setid("hh");

* 1. this

函数中，this总是指向当前函数的所有者对象，只有在运行时才知道其具体指向。

fun(); //此处调用者是window对象。

fun.call(obj); //此处把对象obj放到fun函数上运行，所以所有者为obj。

* 1. 作用域

if while for 不能形成作用域，只有函数才有作用域，即if语句块内变量可以被if外面访问。

* 1. 遍历

$(selecter).each(func(index,element){}) 在遍历页面文档时使用较多。

$.each(data,func(index,element){}) 在遍历数据时使用较多，element参数可以省略，函数体中用this替代。

如果break，使用return false；

如果continue，使用return true；