**Vue**

环境准备：

1.安装node.js

检查 node –v, npm –v

2.淘宝镜像：npm config set registry https://registry.npm.taobao.org

3. 安装vue-cli3.0, 目的使用vue的命令

npm install –g @vue/cli

检查： vue --version

停止当前的运行 ： Ctrl+C> 确定要终止当前操作吗（Y/N）,确定终止的话 ：Y 回车

package.json 对项目的描述

创建cli2.0 vue init webpack 项目名称

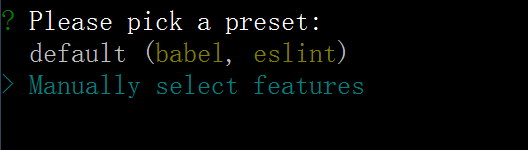
1.创建项目 cli3.0

1>vue ui 回车

自动打开一个网址 => 项目仪表盘

2>vue create 项目名称

项目名称英文，下划线，可以使用数字 比如 vue create vue\_demo



default: 默认

manually: 手动去配置项目

上下箭头选择配置，空格键切换选中状态

Babel: es6转es5

TypeScript: TS语言 angular框架

Progress……:

Router: 路由

Vuex: 集成式的状态

Css: 预处理器 sass less

Linter: 校验代码格式

Unit test: 测试

E2E test: 端对端测试

In dedicated config files: 生成配置文件

In package.json: 集成到package.json文件中

npm run serve: 启动项目

Vue-cli:搭建项目的脚手架，进行快速开发的完整系统

官网 🡪生态系统 ->vue/cli

vue基础

Vue：构建用户界面的 渐进式 框架

安装：

1.CDN <script *src*="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>

2.直接下载到本地

3.cli 构建

使用：Vue 一个构造函数,接受配置对象,配置对象中有配置参数(固定的) new Vue({})

配置参数： el:{} element 告诉Vue实例管理的模板区域

date:{} 一个对象 数据，数据里面存的是key或value键值对的形式

methods:{} 配置参数方法 (函数)

var vm = new **Vue**({ *//配置对象、选项对象*

el: '#zpp', *//元素*

*//渲染dom, 必须有一个根标签*

template:`

<div>

<h3>标题</h3>

<p>{{massage}}</p>

<p>{{btn}}</p>

</div>

`,

data: {massage:'hello' },*//数据*

computed: { **btn**(){return 666}}, *//计算属性的值是函数的返回值*

watch: {}, *//监听属性*

methods: {}, *//Vue声明中的方法函数*

*// 生命周期钩子*

**beforeCreate**() {},

**created**() {},

**beforeMount**() {}, *//挂载之前*

**mounted**() {}, *//挂载完成*

**beforeUpdate**() {}, *//更新数据前*

**updated**() {}, *//更新完成*

**beforeDestroy**() { }, *//销毁*

**destroyed**() {}, *//销毁完成*

})

computed:{} 计算属性

watch:{} 监听属性

计算属性 computed (配置属性：Vue的属性)

计算属性名称是函数名称,属性值是函数的返回值,必须写return

使用 {{}} 把计算属性的结果显示在页面上

计算属性会被存入到**缓存**中(多次执行，只计算一次),相比

methods节省性能

执行：在初始化和相关数据改变时执行

监听属性 watch (配置属性)

属性发生变化时，回调函数自动调用，在函数内部计算

vm.$watch() 另一种写法，写在Vue实例外

computed:{ 基于依赖进行缓存(即this.f和this.l)，只有依赖

**idn**(){ 改变时，才会重新计算

**console**.**log**('执行值')

return this.f + this.l

}

},

watch: {

**f**(newVal,oldVal){

*// newVal 新值 oldVal 旧值(一般不写，只写一个value代表新值)*

this.idn2 = this.newVal+this.oldVal

},

**l**(value){

*// newVal 新值 oldVal 旧值*

this.idn2 = this.f+value

}

},

挂在实例的方法 1.配置对象的el参数中 =>el: '#app' 写在Vue实例内

2.通过实例的方法 **$mount**() => vm.**$mount**('#app') 写在Vue实例外

vm.**$destroy**() *//完全销毁一个实例*

Vue优点：1.遵循MVVM模式

2.代码简介，灵活，运行效率高

3.不操作dom,只关注数据

**MVVM模式**： M：model 模型，数据

V：view视图层，模板

VM：ViewModel 视图模型层，中间桥梁，实例化的Vue对象

\*模板语法

插值：1.文本 “Mustache”语法 (双大括号)小胡子语法 {{}}

2.指令，都是v-开头,自定义属性

v-html更新HTML，将HTML元素解析出来

v-text解析文本数据,输出纯文本

**v-if** 判断，根据表达式的真假值显示元素，为真时v-if="true"，渲染。为假时v-if="false"，不渲染

**v-else** v-if的else块,必须跟在v-if 或 v-else-if后，不能单独使用

**v-show** 判断，根据表达式的真假值显示元素，为真时v-show="true"，渲染。为假时v-show="false"，不渲染

v-if与v-show的区别：

v-if 控制元素的添加和删除，DOM操作 消耗性能

v-show控制元素的css的display属性 频繁切换v-show

**v-once** 绑定事件，只触发一次

v-on: 绑定事件

语法：v-on:事件类型 = "函数名"

简写 @click="add"

1. add:function(){} 2> add(){}

不传参数，函数名可以不加参数，如果需要传参数，函数名必须家小括号，并把参数放在括号内

参数： $event vue提供，事件详情

事件修饰符：

.prevent 阻止默认事件 *@click.prevent*="fn"

.stop 阻止冒泡

按键修饰符：

.event 键盘事件(键名或键值)

keyup.键值 keyup.键名

1. *v-on:keyup.13*="ev($event)" 2>*v-on:keyup.evter*="ev($event)"

<div *id*="app">

{{ message.toUpperCase() }} *<!--js表达式,数字，-->*

<p *v-html*="name"></p>

<span *v-html*="age"></span>

<p *v-text*="t"></p>

<button *@click*="add3('点击',$event)">点击4</button>

<!--函数的两个参数皆为可选，无顺序要求 $event vue提供，事件详情-->

</div>

<script>

var app = new **Vue**({

el: '#app', *//* **id名或类名或标签名(一般是id名，唯一性)**

data: { *//数据(插入值)*

message: 'Hello Vue!',

name:"<b>html标签</b>",

age:56,

t:"<b>html标签</b>",

}，

methods:{

**add**:function () { *//两种写法*

**console**.**log**(1)

},

**add3**(str,event){ //对应上边函数的两个参数

**console**.**log**('击')

**console**.**log**(str)

**console**.**log**(event)

}

}

})

</script>

v-for 列表渲染指令，主要用来遍历数组或对象

语法：v-for="item(参数) in object(需要遍历的对象)"

遍历数组：只写一个参数时，是数组的每一项

两个参数：数组项 索引

遍历对象：一个参数：属性值

两个参数：属性值 属性名

三个参数：属性值 属性名 索引



上面的 key 可以引用到下面



v-bind 给元素绑定属性

语法： v-bind:属性名='属性值'

简写： :属性名='属性值'

v-bind:class='变量'

简写 :class='变量' 改变class类

v-bind:style=''

简写 :style='' 改变样式

class和style设置时可以通过数组形式设置多个 => :class="[obj1,obj2]" :style="[obj1,obj2]"

<div *id*="app1">

<p *:class*='siz'>不安良</p>

<span *v-bind:class*={psiz:false,pclass:true}>对象</span>

*<!-- 两种写法 -->*

<span *:class*={psiz:isa,pclass:isb}>对象</span>

*<!-- font-size => fontSize 50px => 50 + 'px' -->*

<span *v-bind:style*="{color:' red',fontSize: 50+'px'}">style类</span>

*<!-- 两种写法 -->*

<span *v-bind:style*="{color:scla,fontSize: n+'px'}">style类</span>

<button *@click*='pcli'> 变换</button>

</div>

<p *:style*="[obj,obj2]">**数组形式**</p>

<script>

new **Vue**({

el: '#app1',

data: {

obj:{

background:"red"

},

obj1:{

fontSize: "50px"

}

}}

v-model: 给表单控件实现双向数据绑定

语法 ： v-model= ''

表单控件：文本框、密码框、单选按钮、复选框、文本域

v-model会忽略表单元素的初始值，将Vue实力上的数据作为来源，所以要在Vue实例中的data里设置**初始值**

**生命周期钩子(生命周期函数、生命周期事件)**

从Vue实例创建，运行，销毁，其间伴随着各种各样的事件，这些事件叫做生命周期函数

分三类：

1.初始化(一次)

\*beforeCreate 实例刚从内存中创建出来，此时还没有初始化data和methods属性，实例还没有挂在元素上

created 数据、方法都已经完成初始化，但是实例还没有挂在元素上

beforeMount 实例已经挂在模板上

\*mounted 所有初始化准备工作已完成，数据渲染了，实例挂载了 【操作DOM，发送请求】

2.更新(多次)

beforeUpdate 只有在实例上的数据更改时才执行

updated

3.销毁(一次)

\*beforeDestroy 【清空定时器、变量防止内存泄漏】

destroyed

**双向数据绑定**

Object.defineProperty 给一个对象添加属性或修改已有属性

语法：Object.defineProperty(obj,attr,descriptor)

obj:要修改或添加的对象

attr:操作的属性

descriptor:描述符，配置项 {}

数据描述符：

value:'属性值' 默认undefined

writable:属性是否可以修改, 默认 false

enumerable:可枚举的(遍历)，默认false

configurable:属性是否可以被删除，默认 false

访问其描述符： 不能与value同时存在，用return代替(用返回值作为属性值)

get: gettter 获得属性值的方法

set: setter 监听属性值变化的方法

【注】当使用了getter或setter方法，不允许使用writable和value这两个属性

双向数据绑定的原理：把一个普通的javascript对象传给vue实例的data选项，vue将遍历此对象的所有属性，通过Object.defineProperty()把全部属性转为访问器的方式（getter/setter），vue对数据进行劫持进而追踪依赖，在属性被访问和修改时通知变化

<input *type*="text" *id*="txt">

<script>

var txt = document.**getElementById**('txt');

obj\_t={};

**Object**.**defineProperty**(obj\_t,'name',{

**get**(){

return n\_name 用return代替(用返回值作为属性值)

},

**set**(value){

**console**.**log**(value);

n\_name = value;

*//数据层改变，传给页面*

txt.value = value;

}

})

txt.**oninput** = function () {

obj\_t.name = txt.value;

}

</script>

**双向数据绑定必须在表单里面使用。**

**ref**

<input *type*="text" ***ref***="userinfo" />

*//获取ref定义的dom节点* this.$refs.userinfo 即表单内输入的数据

**console**.**log**(this.$refs.userinfo);

this.$refs.box.style.background='red';

**alert**(this.$refs.userinfo.value);

**e.srcElement dom节点**

<button *data-aid*='123' @*click*="**eventFn**($event)">事件对象</button>

methods:{

**eventFn**(e){

**console**.**log**(e);

*//* **e.srcElement dom节点**

e.srcElement.style.background='red';

**console**.**log**(e.srcElement.dataset.aid); */\*获取自定义属性的值\*/*

}

}

<style *lang*="scss" *scoped*>

css 局部作用域 scoped

插槽

<slot></slot> 元素 在组件中使用

作用： 默认情况下，在组件元素中写一些内容是不起作用的，在组件中使用 slot 标签，组件中写的内容才生效

如果在组件(slot)标签中写内容，会显示slot标签中的内容

如果在组件(slot)标签中未写内容，会显示组件标签中的内容 (优先)

默认插槽

具名插槽 : 起名字的插槽 顺序：按slot顺序

给 slot 标签添加name属性，name属性值作为slot的属性值， slot属性给要添加的标签

VueCLi

组件：可复用的功能界面

组件化开发优点:

提高开发效率

方便重复使用

简化调试步骤

便于协同开发

提升整个项目的可维护性

组件：页面上可复用的局部功能界面，整个页面是组件的容器，每个组件都有自己的逻辑(可复用的Vue实例)

组件构成要素：

html结构

css样式

逻辑、数据

组建的**注册**方式：

全局注册 Vue.component('组件名称','{}') 写在newVue的前面

命名规则：1.大驼峰命名 ----- 使用时用分隔符表示

2.断线分隔符、烤串命名法 => a-b-c

使用组件：把组件名称当做自定义html元素

局部注册 new Vue({}) 实例里面 components:{'组件名称':{}}

组件的使用

1. 引入组件

在 script 标签中

import Home from './components/Home.vue';

export default {components: {Home}}

2、挂载组件、注册组件

在 DOM 中

<v-home></v-home>

3、在模板中使用

*根组件*  App.vue

<!-- 模板 写HTML-->

<template> <!--template 盛放盒子、不渲染-->

<div *id*="app"> *<!-- 所有的内容要被根节点包含起来 -->*

<h3>我是根组件标题</h3>

<!-- 使用组件 自闭和 -->

<Header />

<List :*list*="list"/>

<Footer/>

</div>

</template>

<script> *// js逻辑*

// js 使用

// import 引入

// Header 变量 、自定义名

// 路径： ./同级元素

import Header from './components/Header.vue'

import List from './components/List.vue'

import Footer from './components/Footer.vue'

export default { */*/默认导出组件

components: { //注册组件，映射成标签

Header, 或 'v-header':Header,*/\*前面的组件名称不能和html标签一样\*/*

List,

Footer

},

**data**() {

return {

list:["项目一","项目二","项目三"]

}

},

}

</script>

父组件给子组件传值

1.父组件调用子组件的时候 绑定动态属性

<**v-header** :title="title"></v-header>

1. 在子组件里面通过 **props***接收父组件传过来的数据*

*prop两种写法1> [*"title"*]*

1. *{*title:**String,**

it:{

it:**String,**

default:默认值,

required:true, //定义该 prop 是否是必填项

validator:function (value) { return value >= 0 }

//自定义验证函数会将该 prop 的值作为唯一的参数代入

}

}

子组件向父组件传值

子组件调用$emit()方法

这个接收两个参数，

第一个参数： 自定义事件名称，必写

第二个参数： 传递的值，可选

父组件监听子组件的自定义事件 v-on

v-on:自定义事件名称="自定义函数名" / @自定义事件名称="自定义函数名"

这个回调函数接收一个参数，参数保留了子组件传入的数据

**自定义函数名**(value){

**console**.**log**(value)

}

v-on:click: DOM提供的事件

父组件主动获取子组件的数据和方法：

1.调用子组件的时候定义一个ref

<v-header ref="header"></v-header>

2.在父组件里面通过

this.$refs.header.属性

this.$refs.header.方法

子组件主动获取父组件的数据和方法：

this.$parent.数据

this.$parent.方法

非父子组件传值

1、新建一个js文件 然后引入vue 实例化vue 最后暴露这个实例

VueEvent.js文件

import Vue from 'vue';

var VueEvent = new **Vue**()

export default VueEvent;

2、在要广播的地方引入刚才定义的实例

import VueEvent from '../model/VueEvent.js';

3、通过 VueEmit.**$emit**('名称','数据')

4、在接收收数据的地方通过 $om接收广播的数据

VueEmit.**$on**('名称',function(){})

**路由**

路由 ：指向。 实现单页面应用 SPA : Single Page Application

映射的关系：组件和路径，路径指向组件

vue-Router是vue的核心插件

安装：

npm install vue-router/npm install vue-router --save(保存)

创建路由组件，组件放在views文件夹中

配置路由信息(关键) => 在js文件中

使用路由

0、安装路由

1、配置路由组件

2、 a、 引入vue和vue-router

b、引入相关组件

c、声明使用路由插件 Vue.**use**(VueRouter)

d、创建路由实例，并且把配置的路由作为参数传递进去，并且导出路由以便被使用的文件导入

e、把路由注入到 main.js中以便整个项目都可以使用路由

3、<router-link>写路由</router-link> 标签，<router-view></router-view> 标签 路由出口

$route 路由组件的属性，保存当前路由的数据信息

<router-link></router-link>

专门写路由的标签 ,

有一个 to 属性，属性值是路由 path 值

标签最后渲染成 a 标签 to 属性渲染成 herf 属性 ,当签选中的路由，a 标签会增加一个类名 .router-link-active

<router-view></router-view>

路由出口，显示组件的地方

占位符，写在哪，显示在哪

<router-link *to*="/home">HOME</router-link>

<router-link *to*="/about">ABOUT</router-link>

<router-view></router-view> 只写标签(写一次)

*<!--命名路由 l另一种写法-->*

<router-link :*to*="{name:'child1',params:{num:123}}">路由--1</router-link>

*<!-- params 传一个对象 -->*

<router-link :*to*="{name:'child2'}">路由--2</router-link>

*// 配置路由*  **router.js中**

*// vue-router依赖vue*

*//引入vue*

import Vue from 'vue'

*// 引入vue-router插件*

import VueRouter from 'vue-router'

*// 引入相关组件*

import Home from './views/Home.vue'

import About from './views/About.vue'

*// 嵌套的子路由*

import child1 from './views/child1.vue'

import child2 from './views/child2.vue'

*// 插件被vue使用的配置，只要使用就要引入*

Vue.**use**(VueRouter)

*// 配置路由信息，每条路由都有参数*

*/\**

path 路径(自定义，一般 "/...") component 对应的组件 name命名路由(自定义)

route 每条路由 routes 路由的集合 router 管理者

*\*/*

const routes = [

{

path: '/home', *//路径(自定义) / 表示根*

component: Home, *//对应的组件*

},

{

path: '/about', *//路径*

component:About

},

*//* 设置默认显示(默认跳转路由)

{

path: '\*',

redirect:'/home' //重定向

}

]

*//创建路由实例*

const router = new **VueRouter**({ *//VueRouter 构造函数，new创建一个实例，参数是一个对象*

*// routes:routes*

routes,

mode:"hash", *//*默认的路由哈希模式

})

*// 导出实例 main.js接收*

export default router

*// 简写*

export default new **VueRouter**({

[

{

path: '/home', *//路径(自定义)*

component: Home *//对应的组件*

},

{

path: '/about', *//路径*

component: About,

children: [ *//***表示子路由**

{

*//path: '/child1:num', //不用 "/" :num 占位符，传值用*

path: '/child1/:num', *//路径*

name:"child1", *//命名路由(自定义)*

component: child1

}

]

})

*<!--* 声明式 导航*-->*

<router-link *to*="/home">Home</router-link>&nbsp;

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

<br>

*<!--* 编程式 导航*-->*

*<!-- 第三个路由组件 -->*

<button @*click*="addRouter">点击</button>

<router-view></router-view>

export default {

name: "app",

methods: {

**addRouter**() {

**console**.**log**(1);

*// 跳转到第三个路由组件*

this.$router.**push**({ path: "/three" });

}

},

编程式导航 router 的实例方法，通过编写代码来实现

$router 路由管理者

方法：

push 增加一条路由

push({push:"/..."})

replace(location, onComplete?, onAbort?)

替换掉当前的 history 记录。

go(n)

在history 记录中向前或者后退多少步，类似 window.history.go(n)。

**父子路由传值：**

params 父传子

1.传参不暴露在地址栏中

1>router.js 设置占位符

2>父路由组件设置 params:{占位符：要传的数据}

3>子路由组件接收 {{$router.parmar.占位符}}

2.传参暴露在地址栏中

query 把参数拼在 url? 后

1>不需要设置占位符

2>父路由组件设置 query:{id:要传的数据}

3>子路由组件接收 {{$router.query.id}}

router-view

利用父子组件传值的方式

路由模式 ：

**hash** (URL中 # 后的部分) 默认 通过改变 # 后面的 hash 值来显示不同的组件，**不会向后端发送请求**(页面刷新)

**history** 去掉hash的 # ，缺点：**URL路径要完全覆盖**(前后端请求的URL要完全一致)，需要后端去处理

运行原理

点击 router-link的时候，router-link身上有to属性，to属性值会去router.js找匹配的path值，path值，path值对应的组件名称就是要显示的组件，最后通过router-view显示出组件

**单页面应用 SPA**  在 vue 中主要通过路由实现的，通过切换路由可以实现路由之间的跳转，这种跳转数局部刷新，不像 a 标签刷新整个页面

请求数据的模板 -- ajax

1. vue-resource 官方提供的 vue的一个插件

使用vue-resource请求数据的步骤

1>需要安装vue-resource模块， 注意加上 --save

npm install vue-resource --save / cnpm install vue-resource --save

2>main.js引入 vue-resource

import VueResource from 'vue-resource';

3>main.js Vue.**use**(VueResource);

4>在组件里面直接使用

this.$http.**get**(地址 URL).**then**(function(res){

*// 成功*

},function(err){

*// 失败*

})

1. axios

axios 的使用

1>安装 cnpm install axios --save

2>哪里用哪里引入axios

Axios.**get**(api).**then**((response)=>{ 返回Promise

this.list=response.data.result;

}).**catch**((error)=>{

**console**.**log**(error);

})

1. fetch-jsonp

### Vuex

Vuex是vue的**状态管理工具**，采用集中存储所有组件的状态，并以相应的规则保证状态以一种可以预测的方式发生变化

vuex的实现原理

state:存放数据

getters：类似于computed 放计算属性

actions：处理异步的函数，通过 组件触发，this.$store.dispatch(‘自定义事件名称’，值)

mutations：只能处理同步代码，由actions触发，commit(‘自定义事件名称’，值)或者通过组件 this.$store.commit（‘自定义事件名称’，值）

module：模块，功能较大的项目可以建立多个store模块，每个模块都有自己的 state,getters,mutations,

actions

Vuex的核心属性：

**state：**存储数据状态；可以通过this.$store.state访问；对应vue里的data；存放数据的方式是响应式的，vue 组件可以从store中读取数据，如果数据变化，组件也会发生相应的更新。

**getters：**就相当于store的计算属性，他的返回值会根据他的依赖被缓存起来，且只有当它的依赖发生改变的时候才 会被重新计算。

**mutations**：更改vuex的store的唯一方法就是提交mutation

**actions**：包含任意异步操作，通过提交mutation间接改变状态。

**modules**：将store分割成模块，每个模块都具备上边四种方法，甚至子模块

### 为什么用vuex?

当组件多层嵌套的时候，父子组件传递数据很麻烦。

增加代码的可读性和可维护性

数据是响应式的，减少数据的派发操作；



数据流的起点

vue辅助调试工具

提交

改变

提供

发出请求

后端数据接口

**vuex数据传递流程**

辅助函数 ：mapState , mapGetters, mapActions,mapMutations

vuex的 state 、getter 使用辅助函数映射过来的数据要放在 computed 中

vuex的 actions 、mutations 使用辅助函数映射过来的数据要放在 methods 中

mapState:映射 state 中的数据到 this.data上

写法： 1> ...mapState([]) 每一项都是字符串 参数：数组[]

2> ...mapState({ 传对象的时候可以起别名 参数:对象{}

别名：箭头函数的返回值就是别名的值，state作为参数的返回值

别名：state => state.count

key:value 原名value是字符串

})

mapGetters:映射 getters 中的数据到 this.data上

写法： 同上

mapActions:映射 actions 中的数据到 methods上

写法： 同上

mapMutations：映射 mutations 中的数据到 methods上

写法: 同上

vue生命周期总共分为8个阶段: 创建前/后，载入前/后，更新前/后， 销毁前/后。

* **beforeCreate** （创建前）vue实例的挂载元素$el和数据对象 data都是undefined, 还未初始化
* **created** (创建后) 完成了 data数据初始化, el还未初始化
* **beforeMount** (载入前) vue实例的$el和data都初始化了, 相关的render函数首次被调用。实例已完成以下的配置：编译模板，把data里面的数据和模板生成html。注意此时还没有挂载html到页面上。
* **mounted** (载入后) 在el 被新创建的 vm.$el替换，并挂载到实例上去之后调用。实例已完成以下的配置：用上面编译好的html内容替换el属性指向的DOM对象。完成模板中的html渲染到html页面中。此过程中进行ajax交互
* **beforeUpdate** (更新前) 在数据更新之前调用，发生在虚拟DOM重新渲染和打补丁之前。可以在该钩子中进一步地更改状态，不会触发附加的重渲染过程。
* **updated** （更新后） 在由于数据更改导致的虚拟DOM重新渲染和打补丁之后调用。调用时，组件DOM已经更新，所以可以执行依赖于DOM的操作。然而在大多数情况下，应该避免在此期间更改状态，因为这可能会导致更新无限循环。该钩子在服务器端渲染期间不被调用。
* **beforeDestroy** (销毁前） 在实例销毁之前调用。实例仍然完全可用。
* **destroyed** (销毁后） 在实例销毁之后调用。调用后，所有的事件监听器会被移除，所有的子实例也会被销毁。该钩子在服务器端渲染期间不被调用。