Vue

> **Progressive JavaScript Framework**
>
> **渐进式** JavaScript 框架，
>
> 是一个 MVVM 框架，适合于以 **数据操作** 为主的项目（web,app）
>
> 特征：易用；灵活；高效
>
> [官网](https://vuejs.org/)
>
> [中文官网](https://cn.vuejs.org/)

项目中使用 Vue.js 有两种方法

(1) 直接使用 script 引入 vue.js 文件

(2) 使用 Vue-CLI 脚手架工具

MVVM

网页的组成：

HTML：提供网页的内容

CSS：为内容提供样式

JS：为内容提供行为/数据

上述三部分分为三个层次

Model：**模型**，指网页中的数据，一般用 JS 中的变量来表示 // data: { ... }

View：**视图**，指网页中的呈现效果，一般用 HTML标签 / CSS 来表示 // div#app

VM(View Model)：**视图模型/控制器**，替代之前的 DOM 操作，把视图和模型 "**绑定**" // new Vue(...);

数据绑定

`{{ }}`：双括号语法，也叫大胡子语法（Mustache）；用于选择特定的 Model 数据绑定 到View视图中

—— 今后只要 Model 数据发生了改变，所有用到该数据的 `{{}}` 都会自动更新。

> `{{ 表达式 }}` 其中的表达式可以是：
>
> - **Model 变量名**，如 `{{ userName }}`
> - **算术 / 比较 / 逻辑 / 三目运算**，如 `{{ price*count }}`
> - **方法调用**，如 `{{ userName.toUpperCase() }}`
> - **属性访问**，如 `{{ book.price }}`

> - **数组下标访问**，如 `{{ list[2] }}`
>
> 注意：不能出现逻辑结构 (if / for 等)

```
var obj = {
  data: { msg: "hello"},
  get msg(){ return this.data.msg; },
  set msg(val){ return this.data.msg = val; }
}
```

指令(directive)

Vue.js 提供的增强型HTML视图 —— 指令 (directive)

> Vue.js为View视图提供了增强型的属性：v-xxx，可以增强HTML的能力
>
> ——称为指令。

(1) `v-text`

- **预期**：`string`
- **详细**：

  更新元素的 `textContent` 。

  如果要更新部分的 `textContent` ，需要使用 `{{ Mustache }}` 插值
- **示例**：

```
<span v-text="msg"></span>
<!-- 和下面的一样 -->
<span>{{msg}}</span>
```

(2) `v-html`

- **预期**：`string`
- **详细**：

  更新元素的 `innerHTML`

  注意： **内容按普通 HTML 插入 - 不会作为 Vue 模版进行编译。**

  如果试图使用 `v-html` 组合模版，可以重新考虑是否通过使用组件来替代

> 在网站上动态渲染任意 HTML 时非常危险的，因为容易导致 XSS攻击
>
> 只在可信内容上使用 `v-html` ，**永不** 用在用户提交的内容上

> 在 **单文件组件** 中， `scoped` 的样式不会应用在 `v-html` 内部，因为那部分HTML没有被 Vue 的模版编译器处理。如果你希望针对 `v-html` 的内容，设置带作用域的CSS，你可以替换为 **CSS Module** 或用一个额外的全局 `<style>` 元素手动设置类似 BEM 的作用域策略

- **示例**：

```
<div v-html="html"></div>
```

(3) `v-show`

- **预期**：`any`
- **用法**：

  根据表达式之真假值，切换元素的 `display` CSS属性

  当条件变化时该指令触发过渡效果

> 注意，`v-show` 不支持 `<template>` 元素，也不支持 `v-else`

(4) `v-if`

- **预期**：`any`
- **用法**：

  根据表达式的值得真假条件渲染条件。在切换元素及它的数据绑定 / 组件被销毁并重建。如果元素是 `<template` ，将提出它的内容作为条件块

  当条件变化时该指令触发过渡效果

> 当和 `v-if` 一起使用时， `v-for` 的优先级比 `v-if` 更高

(5) `v-else`

- **不需要表达式**
- **限制**：前一兄弟元素必须有 `v-if` 和 `v-else-if`
- **用法**：

  为 `v-if` 或者 `v-else-if` 添加 "else 块"

```
<div v-if="Math.random() > .5">
  Now you see me
</div>
<div v-else>
  Now you don't
</div>
```

(6) `v-else-if`

- **类型**：`any`
- **限制**：前一兄弟元素必须有 `v-if` 或 `v-else-if`

- **用法**：

  表示 `v-if` 的 "else if 块" ，可以链式调用

```
<div v-if="type === 'A'">
  A
</div>
<div v-if="type === 'B'">
  B
</div>
<div v-if="type === 'c'">
  C
</div>
<div v-else>
  Not A/B/C
</div>
```

(7) `v-for`

- **预期**：`Array | Object | number | string`
- **用法**：

  基于源数据多次渲染元素或模板块。此指令之值，必须使用特定语法 `alias in expression` ，为当前遍历的元素提供别名：

```
<div v-for="item in items">
  {{ item.text }}
</div>
```

  另外也可以为数组索引指定别名 (或者用于对象的键)：

```
<div v-for="(item, index) in items"></div>
<div v-for="(val, key) in object"></div>
<div v-for="(val, key, index) in object"></div>
```

  `v-for` 默认行为试着不改变整体，而是替换元素。迫使其重新排序的元素，你需要提供一个 `key` 的特殊属性：

```
<div v-for="item in items" :key="item.id">
  {{ item.text }}
</div>
```

(8) `v-on`

- **缩写**：`@`
- **类型**：`Function | Inline Statement`

- **参数**：`event`
- **修饰符**：
  - `.stop` - 调用 `event.stopPropagation()`。
  - `.prevent` - 调用 `event.preventDefault()`。
  - `.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
- **用法**：

  绑定事件监听器。事件类型由参数指定。表达式可以是一个方法的名字或一个内联语句，如果没有修饰符也可以省略。

  用在普通元素上时，只能监听**原生 DOM 事件**。用在自定义元素组件上时，也可以监听子组件触发的**自定义事件**。

  在监听原生 DOM 事件时，方法以事件为唯一的参数。如果使用内联语句，语句可以访问一个 `$event` 属性：`v-on:click="handle('ok', $event)"`。

  从 `2.4.0` 开始，`v-on` 同样支持不带参数绑定一个事件/监听器键值对的对象。注意当使用对象语法时，是不支持任何修饰器的。

- **示例**：

```
<!-- 方法处理器 -->
<button v-on:click="doThis"></button>

<!-- 内联语句 -->
<button v-on:click="doThat('hello', $event)"></button>

<!-- 缩写 -->
<button @click="doThis"></button>

<!-- 停止冒泡 -->
<button @click.stop="doThis"></button>

<!-- 阻止默认行为 -->
<button @click.prevent="doThis"></button>

<!-- 阻止默认行为，没有表达式 -->
<form @submit.prevent></form>

<!--  串联修饰符 -->
<button @click.stop.prevent="doThis"></button>

<!-- 键修饰符，键别名 -->
<input @keyup.enter="onEnter">

```

```
<!-- 键修饰符，键代码 -->
<input @keyup.13="onEnter">

<!-- 点击回调只会触发一次 -->
<button v-on:click.once="doThis"></button>

<!-- 对象语法 (2.4.0+) -->
<button v-on="{ mousedown: doThis, mouseup: doThat }"></button>
```

在子组件上监听自定义事件 (当子组件触发“my-event”时将调用事件处理器)：

```
<my-component @my-event="handleThis"></my-component>

<!-- 内联语句 -->
<my-component @my-event="handleThis(123, $event)"></my-component>

<!-- 组件中的原生事件 -->
<my-component @click.native="onClick"></my-component>
```

(9) `v-bind`

- **缩写**：`:`
- **预期**：`any (with argument) | Object (without argument)`
- **参数**：`attrOrProp (optional)`
- **修饰符**：
  - `.prop` - 被用于绑定 DOM 属性 (property)。(差别在哪里？)
  - `.camel` - (2.1.0+) 将 kebab-case 特性名转换为 camelCase. (从 2.1.0 开始支持)
  - `.sync` (2.3.0+) 语法糖，会扩展成一个更新父组件绑定值的 `v-on` 侦听器
- **用法**：

  动态地绑定一个或多个特性，或一个组件 prop 到表达式。

  在绑定 `class` 或 `style` 特性时，支持其它类型的值，如数组或对象。可以通过下面的教程链接查看详情。

  在绑定 prop 时，prop 必须在子组件中声明。可以用修饰符指定不同的绑定类型。

  没有参数时，可以绑定到一个包含键值对的对象。注意此时 `class` 和 `style` 绑定不支持数组和对象。
- **示例**：

```
<!-- 绑定一个属性 -->
<img v-bind:src="imageSrc">

<!-- 缩写 -->
<img :src="imageSrc">

<!-- 内联字符串拼接 -->
<img :src="'/path/to/images/' + fileName">

<!-- class 绑定 -->
```

```
<div :class="{ red: isRed }"></div>
<div :class="[classA, classB]"></div>
<div :class="[classA, { classB: isB, classC: isC }]">

<!-- style 绑定 -->
<div :style="{ fontSize: size + 'px' }"></div>
<div :style="[styleObjectA, styleObjectB]"></div>

<!-- 绑定一个有属性的对象 -->
<div v-bind="{ id: someProp, 'other-attr': otherProp }"></div>

<!-- 通过 prop 修饰符绑定 DOM 属性 -->
<div v-bind:text-content.prop="text"></div>

<!-- prop 绑定。“prop”必须在 my-component 中声明。-->
<my-component :prop="someThing"></my-component>

<!-- 通过 $props 将父组件的 props 一起传给子组件 -->
<child-component v-bind="$props"></child-component>

<!-- XLink -->
<svg><a :xlink:special="foo"></a></svg>
```

`.camel` 修饰符允许在使用 DOM 模板时将 `v-bind` 属性名称驼峰化，例如 SVG 的 `viewBox` 属性：

```
<svg :view-box.camel="viewBox"></svg>
```

在使用字符串模板或通过 `vue-loader` / `vueify` 编译时，无需使用 `.camel`。

(10) `v-model`

- **类型**：随表单控件类型不同而不同
  - 限制：
  - `<input>`
  - `<select>`
  - `<textarea>`
  - components
- **修饰符**：
  - `.lazy` - 取代 `input` 监听 `change` 事件
  - `.number` - 输入字符串转为数字
  - `.trim` - 输入首尾空格过滤
- **用法**：

  在表单控件或者组件上创建双向绑定

(11) `v-pre`

- **不需要表达式**

- **用法**：

  跳过这个元素和它的子元素的编译过程；可以用来显示原始 Mustache 标签；跳过大量没有指令的节点会加快编译。

- **示例**：

```
<span v-pre>{{ this will not be compiled }}</span>
```

(12) `v-cloak`

- **不需要表达式**
- **用法**：

  这个指令保持在元素上直到关联实例结束编译。

  和 CSS 规则如 `[v-cloak] {display: none}` (需要自己定义这条规则）一起用时，

  这个指令可以隐藏未编译的 Mustache 标签直到实例准备完毕

- **示例**：

```
[v-cloak]{
  display: none;
}
```

```
<div v-cloak>
  {{ message }}
</div>
```

  不会显示，直到编译结束。

(12) `v-once`

- **不需要表达式**
- **用法**：

  只渲染元素和组件 **一次** ；随后的重新渲染，元素 / 组件及其所有的子节点将被视为静态内容并跳过

  这可以用于优化更新性能

- **示例**：

```
<!-- 单个元素 -->
<span v-once>This will never change: {{msg}}</span>
<!-- 有子元素 -->
<div v-once>
  <h1>comment</h1>
  <p>{{msg}}</p>
</div>
<!-- 组件 -->
<my-component v-once :comment="msg"></my-component>
<!-- `v-for` 指令-->
<ul>
  <li v-for="i in list" v-once>{{i}}</li>
</ul>
```

对 class 和 style 的属性绑定

- **方式1：把 class 和 style 作为普通字符串进行绑定**

```
<span :class="unameClassString"></span>
unameClassString: 'label  label-danger'
-------------------------------------------------
<span :style="unameStyleString"></span>
unameStyleString: 'color:red; font-size: 2em'
```

- **方式2：把 class 和 style 绑定为对象**

```
<span :class="unameClassObject"></span>
unameClassObject: {
  label: true,
  'label-danger': false
  'label-success': true
}
-------------------------------------------------
<span :style="unameStyleObject"></span>
unameStyleObject: {
  color:  'red',
  'font-size':  '2em'
}
```

HTML编写规范

(1) HTML标签名

H5官方标签写法：<productlist></productlist>

自定义组件推荐写法：<product-list></product-list>

不推荐：<productList></productList>

(2) HTML标签属性名

H5官方属性写法：<p title="" href="">

自定义组件属性推荐写法：<xz-header my-title="">

不推荐：<xz-header myTitle="">

(3)HTML标签事件名

H5官方事件写法：<p onmouseover="">

自定义组件事件推荐写法：<xz-header onmyevent="">

不推荐：<xz-footer onMyEvent="">

Vue.js中组件的模板写法

写法1： 直接写在组件的 template 属性(string)中

```
{
  template: `<div>...<div>`
}
```

写法2：写在一个独立的 <template> 元素中，后在 template 属性中引用

```
<template id="tplMyComponent">
  <div>....</div>
</template>
--------------------------------
{
  template: '#tplMyComponent'
}
```

> <template>元素是HTML5标准提供的标签，浏览器默认display:none；
>
> 其作用就是为后面的JS提供一段可用反复使用的HTML片段

计算属性

问题描述： 在 View 中多次调用函数以获取计算后的值：

```
<div> 总计：{{ getSum()}} </div>
<div> 总计：{{ getSum()}} </div>
<div> 总计：{{ getSum()}} </div>
```

每次 {{ }} 绑定都会重新调用一次该函数，而不会复用刚刚得到的结果，影响页面渲染效率

解决方法：计算属性（Computed Property）

计算属性的本质是一个声明在 computed 中的函数，使用时像使用普通模型属性一样，不要加 `()`

Vue.js 会缓存计算属性的计算结果，只要关联的模型数据没有改变，不论使用多少次，

该属性访问的都是对应的缓存

`computed`: {

sum(){}

}

自定义指令

Vue.js 中，除了预定义的 13 个指令外，还允许用户 **自定义扩展指令**

```
<input v-focus>
// 注册一个全局自定义指令 `v-focus`
Vue.directive('focus',{
  // 当被绑定的元素插入到 DOM 中时
  inserted: function(el){
    // 聚焦元素
    el.focus();
  }
});
```

钩子函数(Hook Function)

> 在一个对象的不同生命时期的某个时刻，会自动调用的函数
>
> —— 称为 “生命周期钩子函数”

- `bind` - 当自定义指令绑定到元素上时
- `inserted` - 当绑定了自定义指令的元素插入到 DOM 时
- `update` - 元素绑定的数据发生了更新时
- `componentUpdate` - 元素的子组件发生更新时
- `unbind` - 元素与自定义指令 **解除绑定** 时

自定义过滤器

> Filter：可用于接受一些原始数据，加以处理，返回另一种格式的数据
>
> Vue.js 官方没有提供预定义过滤器

创建：

```
Vue.filter("fileterName", function(val){
  ...
  return ...;
})
```

使用：

```
<div>
  {{ data | filtername }}
</div>
```

组件(Component)

> 组件是可复用的 Vue 实例，没有 `el` 属性，其他属性和 `new Vue()` 基本一样
>
> Component：组件，指页面中的独立区域（有专有HTML，CSS，数据），
>
> 这样的多个组件聚合而成一个大型的页面，在代码层面上，一个组件就是一个可以反复使用的自定义标签

```
<body>
  <tb-header></tb-header>
  <tb-banner></tb-banner>
  <tb-youhaohuo></tb-youhaohuo>
  <tb-aiguangjia></tb-aiguangjia>
  ...
</body>
```

```
Vue.component('xz-buy-counter', {
  data(){
    return {
      num: 0
    }
  },
  methods: {
    add(){
      this.num ++;
    },
    reduce(){
      this.num --;
```

```
    }
  },
  watch: {},
  template: `
  <div>
    <button @click="reduce" class="btn btn-primay">-</button>
  <span>{{ num }}</span>
    <button @click="add" class="btn btn-primay">+</button>
  </div>
`
});
```

组件的生命周期

> Vue/组件的实例的生命周期

生命周期名称	描述	钩子函数
create	组件创建完成，尚未挂载到 DOM 树	beforeCreate(){} created(){}
mount	组件挂载到 DOM 树	beforeMount(){} mounted(){}
update	组件所绑定的 Model 数据发生改变，需要更新视图	beforeUpdate(){} updated(){}
destroy	调用了 `$destroy` 方法，组件要被销毁	beforeDestroy(){} destroyed(){}

1532478826300

异步请求

(1) 使用原生 XHR 对象——麻烦

(2) 使用 jQuery 的封装函数 —— 大材小用

(3) 使用官方提供的 VueResource 插件 —— 官方废弃

(4) 使用第三方工具 Axios —— 本身与Vue没任何关系

Axios 是一个基于 promise 的 HTTP 库，可以用在浏览器和 node.js 中。

> 使用Axios的步骤：
>
> (1) 提前搭建好可供访问的服务器端 API：
>
> http://127.0.0.1/lwh/VUEJS_DAY04/project/data/product_list.php?pno=2
>
> (2) 在HTML页面中引入JS文件

> `<scriptsrc="js/axios.min.js">\</script>`
>
> (3) 调用Axios，发起异步请求
>
> `axios.get(url).then(function(res){ }).catch(function(err){})`

> ES6 新特性：Promise解决回调地狱问题
>
> var p = new Promise(fn, fn){ }
>
> p.then(function(){ }).catch(function(){ })

父子组件间的通信 ☆

> 父组件 => 子组件
>
> Props Down
>
> 原理：
>
> 给子组件声明自定义属性，父组件使用子组件时为自定义属性绑定数据

```
// 父：
{
  template: `<my-photo :signiture="myName" :age="myAge" />`,
  data(){return {
    myName: '呵呵',
    myAge: '18'
  }},
  components: {
    myPhoto
  }
}
// 子:
var myPhoto = {
  template: `<h4>{{ signiture }}</h4>`,
  props: ['signiture', 'age'] // 子组件自定义属性
}
```

> 子组件 => 父组件
>
> Events Up
>
> 原理：
>
> 子组件触发自定义事件（事件中携带自己的数据）；
>
> 父组件提供函数处理该事件，得到子组件传递而来的数据

```
//父：
  {
   template: `<my-modify  @changename="fun"/>`,
     //注意：自定义事件监听函数不能写()
```

```
    components: {
        myModify
    },
    methods:{
       fun(data){  //获得子组件通过事件传递的数据：data }
    }
}
//子：
var myModify = {
    template: `....`,
    data(){ return { userInput: 'ABC' }}
    ...
       this.$emit('changename', this.userInput);
   //子组件某种情况下发射自定义事件，并携带数据
    ...
}
```

> 小技巧：父/子组件间传递的数据——简便方法：**$refs 和 $parent**
>
> 组件模板内的任何一个子元素都可以声明ref属性

```
<template id="tplMyParent">
  <div>
      <h3  ref="c1"></h3>
      <my-child  ref="c2"></my-child>
  </div>
</template>
// ----------------------------------------
var myParent = {
    methods: {
        show(){  this.$refs —— 其中包含着每个具有ref属性的子元素     }
    }
}
```

> 注意：此方法让父组件和子组件可以直接访问对方的全部的属性和方法，可能产生“过分暴露”问题

兄弟组件间的通信

> 说明：一般项目中，大多出现的是父子组件间传递数据；如果真的需要兄弟组件间传递数据，一般是先传递给父组件再传递兄弟组件。也可以使用“消息总线 (Message Bus) ”方式简化此过程。
>
> 使用Bus机制，任何一个组件都可以在总线上发射事件，任何一个特定的兄弟组件都可以处理相关事件。

// 创建一个消息总线对象——全局的Vue示例(不带Options)

var bus = newVue( );

// 组件1内发射事件

bus.$emit('事件名', 数据)

// 组件2内处理事件

bus.$on('事件名', function(data){ data就是事件数据 })

动画钩子(transition)

Vue 在插入、更新或者移除 DOM 时，提供多种不同方式的应用过渡效果。包括以下工具：

(1)在 CSS 过渡和动画中自动应用 class

(2)可以配合使用第三方 CSS 动画库，如 Animate.css

(3)在过渡钩子函数中使用 JavaScript 直接操作 DOM

(4)可以配合使用第三方 JavaScript 动画库，如 Velocity.js

使用方法：

```
<transition  name="动画名">
  <any>此处的ANY可以是任意HTML元素或自定义组件元素</any>
</transtion>
------------------------------------------------------------
<style>
/*transition元素为指定元素提供了六个动画钩子class*/
  .动画名-enter-active{   }        // 进入动画进行中
  .动画名-enter {   }              // 进入动画开始时的状态
  .动画名-enter-to{  }             // 进入动画完成时的状态
  .动画名-leave-active{   }        // 离开动画进行中
  .动画名-leave{   }               // 离开动画开始时的状态
  .动画名-leave-to{   }            // 离开动画完成时的状态
</style>
```

SPA 应用

> SPA : Single Page Application 单页面应用

多页面应用	单页面应用
每个项目中有多个独立的完整的HTML页面	整个项目中只有一个页面具有完整的HTML结构， 其他 '页面'只是一段HTML片段而已（如几个div）
页面跳转： 删除当前DOM，下载并挂载另一个DOM	页面跳转： 把当前DOM树某个div删除，下载并挂载另一个div片段 不涉及DOM树重建
如果两个页面中都需要某个资源，需要下载两次	如果两个"页面"中都需要某个资源，无需多次下载
无法实现"页面切换过场动画"	很容易实现"页面切换过场动画"

SPA 应用的原理：

(1) 定义一个 路由词典(routes)

```
[
  { path: '/login', template: 'login.html' },
  { path: '/main', template: 'main.html'},
  ...
]
```

(2) 客户端浏览器加载整个项目唯一完整的页面： index.html

```
<html>
  <head>
    ...
    <style></style>
    <script></script>
    ...
  </head>
  <body>
    <router-view></router-view>
  </body>
</html>
```

(3) 客户端浏览器通过地址栏中的地址来决定需要哪个路由页面

ex: `http://127.0.0.1/index.html#/main`

使用 js 解析 location.hash，查找路由词典，异步加载对应的路由页面挂载到 router-view 缺口中

Vue-Router

> Vue.js 核心内容中没有提供 SPA 功能，而是用一个官方扩展插件 Vue-Router 实现了此功能

插件 Vue-Router 使用

(1) 引入必需的 js 文件

`<script src='vue.js'></script>`

`<script src="vue-router.js"></script>`

(2) 创建必需的 “页面” 组件

```
var Login = {
  template: '<div>这里是登录页面</div>'
}
var Main = {
  template: '<div>这里是主页面</div>'
}
```

(3) 创建路由词典，为每个组件指派一个路由地址

```
var routes = [
  { path: '/', component: Login}, // 默认路由地址
  { path: '/login', component: Login },
  { path: '/main', component: Main }
];
```

(4) 创建一个路由器对象，包含该路由词典

```
var router = new VueRouter({ routes })
```

(5) 将路由器对象指派给根组件

```
new Vue({
  router
})
```

(6) 在唯一完整的 HTML 页面中添加 “路由插槽”

```
<router-view></router-view>
```

> 小知识：如何引入一个Vue.js的插件：
>
> Vue.use(插件名称);
>
> new Vue({})

如何在两个路由地址间跳转

HTML 标签实现跳转

```
<div id="app">
  <router-link to='/main'>Go to Main</router-link>
  <router-link to="/login">Go to Login</router-link>
</div>
```

JS 实现跳转

```
// 给按钮绑定事件，实现跳转
this.$router.push('/main')
this.$router.back()/forward()
this.$router.go(-2)
```

嵌套路由(Nested Routes)：

嵌套路由：在某个路由组件中再嵌套一个 `<router-view>` ，以切换多个不同的局部组件

```
var routes = [
  {
    path: '/main', component: Main, children:[
      {path:'/main/stat',component: Stat},
      {path:'/main/plist',component: ProductList},
    ]
  }
]
```

注意：Main组件的模板中必须存在“路由插槽”：`<router-view />`

Vue.js 的两种使用方法

(1) 直接 script 引入 vue.js -- 适合于在某一个页面中使用 vue

(2) 使用 Vue-CLI 脚手架工具创建空白项目模板 —— 适合于在整个项目中全部采用 Vue

CLI : Command Line Interface 命令行界面，即一个命令行工具
Scaffolding 脚手架，用于创建最终项目的一个骨架工具

- Vue-CLI 是官方提供的命令行版的脚手架工具，用于快速的创建一个基本 SPA 应用，
  用户可以基于该项目脚手架继续开发。
  这个项目脚手架包含：Vue.js、Webpack、Babel、ESLint、Node/Express ...
- 注意：**这个项目的空模板其实是一个完整的 web 服务器！**其运行不依赖于任何其他 web 服务器

Vue-CLI 使用步骤

(1) 下载 Vue-CLI 工具到当前计算机中一个可以在全局运行命令的目录下

```
npm install @vue/cli
```

默认下载目录 C:\Users\vip班\AppData\Roaming\npm

(2) 运行 Vue-CLI 工具，创建一个空白脚手架项目

```
vue create xz_admin_v2
```

创建一个空白项目，自带脚手架文件，也会自动的下载所有依赖的NPM包

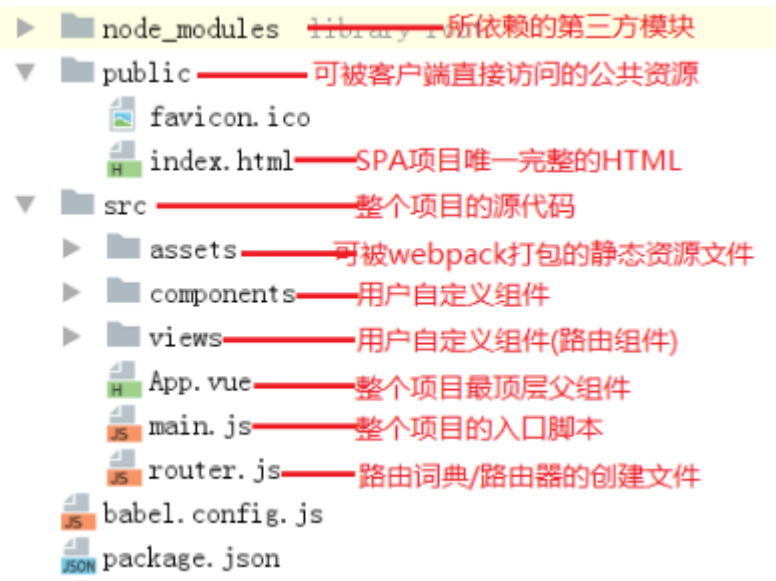
(3) 进入空白项目，添加需要的组件，运行该项目

```
npm run serve
```

(4) 使用客户端浏览器访问上述项目

http://127.0.0.1:8080

CLI 项目基本目录结构



提示：Vue-CLI创建的脚手架项目使用方法

先创建所需的路由组件，配置路由地址；

再详细划分每个路由组件内部的子组件及数据

> Vue.js脚手架项目经典错误：

```
Failed to compile.
./src/assets/img/favicon.ico 1:0
Module parse failed: Unexpectedcharacter '' (1:0)
You may need an appropriate loader tohandle this file type.
(Source code omitted for this binaryfile)
```

原因：Webpack默认的file-loader加载器只能打包jpg/png/gif，不能打包.ico！

解决：修改Webpack的配置文件，让其加载.ico图片。

在项目根目录下创建vue.config.js，内容如下（无需记忆，查询手册即可）：

```
module.exports = { /*对webpack进行配置*/
  configureWebpack:{
  module: {
       rules:[{
         test: /\.ico$/,
         use: 'file-loader'
       }]
    }
  }
}
```

> Node.js中的模块系统 vs ES6中的模块系统

Node.js 中的模块	ES6 模块
module.exports = { }	export default { }
const m = require('xx')	import m from 'xx'
exports和require可以在任意地方编写，如for/if/function内部	export和import只能在最顶层编写，不能放在其他结构内部

Vue.js 单文件组件中引入外部 CSS 和 JS

```
...
<script>
  import '../assets/css/base.css'
  import '../assets/js/jquery.js'  // 通用JS，不涉及具体DOM操作

export default {
  mounted(){
    require('../asssets/css/index.css')
  }
}
<script>
...
```

Webpack 资源打包时的特点

- **只会对源代码中出现的静态资源进行打包**

  `<img src="../assets/img/1.jpg">`

  `import "../assets/css/base.css"`

- **不会对动态生成的资源地址打包**

  `<img :src="imgSrc">`

  - 解决：`<img :src="require('../assets/' + imgSrc)">`