Vue

1. **简介：(为什么使用vue)**
2. **与angular相比优点:**
   1. api简单，快速上手，学习成本低；
   2. Angular 使用双向绑定，Vue 也支持双向绑定，不过为单向数据流，数据从父组件单向传给子组件；
   3. Vue.js 中指令和组件分得更清晰。指令只封装 DOM 操作，而组件代表一个自给自足的独立单元 —— 有自己的视图和数据逻辑。在 Angular 中两者有不少相混的地方
   4. Angular的脏检查机制，造成了如果项目庞大之后性能降低问题；每一次更新数据，都导致所有的数据绑定，进行一次遍历；

Vue的数据变化更新是根据依赖追踪的观察系统并且异步列队去更新，数据变化都是独立的触发队列中相应的事件；

* 1. 渲染层创建了虚拟dom（轻量级，2.0），减少了内存开销和加快了渲染速度；
  2. 组件式开发，每个组件都具有生命周期，便于自己状态的维护；实现了代码的高复用性；

1. **与react相比：**

* 共同点：

1. 使用 Virtual DOM
2. 提供了响应式（Reactive）和组件化（Composable）的视图组件
3. 将注意力集中保持在核心库，而将其他功能如路由和全局状态管理交给相关的库

* 优势：
* 性能：

1. vue的更新渲染过程是自动追踪的，而react如果想要提高更新的性能需要通过shouldComponentUpdate钩子函数对数据进行对比；
2. 对于组件的语法，在vue中既可以支持jsx语法，将css和html都写入js文件中，又可以使用template模板进行编译，在vue中推荐使用template模板进行编译，减少学习成本，更加符合前端开发语法；而react仅仅支持jsx语法，jsx的优点：一、开发工具对jsx的支持比其他的vue模板更先进；
3. 对于css的支持：

vue中可以直接在style中写css语法，可以支持媒体查询，等等一切css属性，并且可以通过scoped进行组件作用域的封闭；

react的组件中如果进行css样式的隔离，只能写入js文件中，比较简单的样式可以进行支持，但是如果比较复杂的则无法支持；

* 规模：

共同点：

都提供了路由和状态管理机制；

都有自己的自动化构建工具；

不同点：

react的学习成本比较高，需要掌握jsx语法和es2015；虽然支持前端babel进行编译，但是不适合生产环境；

vue向下扩展可以像jq那样使用在前端项目中，向上扩展可以像react一样进行前端的自动化构建；

* 本地渲染：(跨平台)

reactNative

weex

1. **总结vue：**

优点：

1. MVVM的开发模式，从dom中解脱出来，双向数据绑定；
2. 数据更新采用异步事件机制；
3. 采用单向数据流；
4. 组件式开发；
5. 采用虚拟dom
6. 支持模板和jsx两种开发模式；
7. 可以进行服务端渲染；
8. 扩展性强，既可以向上又可以向下扩展

缺点：

1. 不兼容IE8以下版本
2. 生态圈不繁荣，开源社区不活跃；
3. **安装与使用：**
4. **直接使用CDN引入：**

<script src="https://unpkg.com/vue"></script>

1. **下载到本地：**

https://vuejs.org/js/vue.js

1. **自动化构建工具：**

Vue.js 有一款官方的脚手架生成工具 vue-cli:

* 全局安装

npm install –g vue-cli

* 创建项目

vue init webpack myproject

* 安装依赖包

npm install

* **目录如下：**

build: 用于存放 webpack 相关配置和脚本。

config: 主要存放配置文件，用于区分开发环境、测试环境、线上环境的不同。

src: 项目源码及需要引用的资源文件。

static: 不需要 webpack 处理的静态资源。

test: 用于存放测试文件；

* **启动服务：**

正常开发时，就会运行命令npm run dev，启动一个小型的express 服务。在这个 express 服务中，会使用webpack-dev-middleware 和 webpack-hot-middleware 这两个 中间件，来进行项目的热部署，即每次修改 src 中的文件后，不需要再按浏览器的刷新来更新 代码，启动的 server 服务会自动监听文件的变化并编译，通知浏览器自动刷新。

* **分析packjson文件：**

"scripts": {

"dev": "node build/dev-server.js",

"start": "node build/dev-server.js --open",

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

"e2e": "node test/e2e/runner.js",

"test": "npm run e2e",

"lint": "eslint --ext .js,.vue src test/e2e/specs"

},

dev:为本地开发的启动项；

start：和dev一样；

build:打包线上部署代码

e2e:这个代表端对端测试，确切的来说就是前端的一个自动化测试；这里使用的是java的nightwatch自动化测试工具，使用测试需要进行java的全局安装

1. **实例：**

1、一个 Vue 实例其实正是一个 [MVVM 模式](https://en.wikipedia.org/wiki/Model_View_ViewModel" \t "_blank)中所描述的 ViewModel - 因此在文档中经常会使用 vm 这个变量名。

2、在实例化 Vue 时，需要传入一个**选项对象**，它可以包含数据、模板、挂载元素、方法、生命周期钩子等选项 ；

(1)模板：

三个属性：template、el、replace（2.0）已经删除，规定template中必须有个根节点；

Replace：为true时不会保留原dom节点，直接将原节点替换，如果是false则，保留原节点

(2)数据：

可以对外部数据进行代理，外部传入的数据如果为对象，对此数据进行的浅拷贝，此时我们成为数据的响应式；

对于初始化后，进行的数据添加无效，要想对于初始后添加的数据生效，需要使用$set(),不过不推荐这么，尽量在数据初始化时，就对所有需要的数据进行添加，可以用undefined和null进行占位；

$watch方法对于实例对象的数据进行监听，并在回掉中接受旧数据和新数据；

(3)方法：

Methods对象来定义方法，并且使用v-on指令来监听DOM事件；

自定义事件：$emit、$on、($broadcast、$dispatch在2.0中已经删除)

(4)生命周期：

在vue实例创建时一系列的初始化步骤：

beforeCreate、created、beforeMount、mounted、beforeDestroy、destroyed 、beforeUpdate、updated、activated、deactivated

(5)数据绑定：

传统的web项目，往往是通过后端模板引擎来进行数据和视图的渲染，例如：php的smarty、java的velocity、node的express、也就是传统的服务端渲染，导致前后端语言混杂在一起，造成渲染后期进行视图更改是，只能通过dom；

a、文本插值

使用{{}}大括号，单次赋值：{{\*}} (2.0中使用v-once=””)

b、 属性插值{{}} (2.0 v-bind:id=””)

c、表达式{{}}

（6）指令：

a、参数：v-bind:src=”attr”，给属性绑定值；

或者可以这样写：src=”{{attr}}”

或者可以这样写 :src=“attr”

b、v-on

1. **计算属性：**

三种对于数据变化监听机制：

1. computed：一个属性依赖于多个属性时，推荐使用
2. watch()：多个属性依赖一个属性是，推荐使用
3. Set、get：set对一个属性设置值时，会自动的调用相应的回掉函数，get的回调函数会根据，函数内部依赖的属性的改变而自动改变

1. **条件渲染：**
2. **v-if**
3. **template v-if**

如果在一个判断中要对多个元素进行渲染，则需要配合template标签；

1. **v-else**
2. **v-else-if**

多次进行链式的使用

1. **key管理可复用的元素：**

Vue 会尽可能高效地渲染元素，通常会复用已有元素而不是从头开始渲染

在这里用key进行元素的唯一标识赋值，降低元素的复用性；

1. **v-if与v-show**

v-if 是“真正的”条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建

v-if 有更高的切换开销，而 v-show 有更高的初始渲染开销

1. **v-if与v-for一起使用**

当 v-if 与 v-for 一起使用时，v-for 具有比 v-if 更高的优先级；如果想让v-if优先级更高，则需要在外套一层标签；进行v-if的渲染

1. **列表渲染：**

**A、数组：**

1. **常用写法：**

v-for=”item in items”

v-for=”(item,index) in items”

1. **结合template使用：**

带有 v-for 的 <template> 标签来渲染多个元素块

**B、对象：**

1. **常用写法：**

v-for="value in object"

v-for="(value, key) in object"

v-for="(value, key, index) in object"

**C、整数迭代：**

1. **常用写法：**

v-for="n in 10"

**D、组件：**

1. **常用写法：**

<my-component v-for="item in items" :key="item.id"></my-component>

**E、key：**

当 Vue.js 用 v-for 正在更新已渲染过的元素列表时，它默认用 “就地复用” 策略，默认的模式是高效的，但是只适用于**不依赖子组件状态或临时 DOM 状态。**为了给 Vue 一个提示，以便它能跟踪每个节点的身份，从而重用和重新排序现有元素，你需要为每项提供一个唯一 key 属性

**F、变异方法，触发视图更新：**

Push、pop、shift、unshift、splice、sort、reverse

G、**重塑数组、显示过滤/排序结果**

对于非变异的数组方法，在操作数组时，一般是采用重新赋值的方式，操作做完原数组，再将结果赋给原属性；如：filter、concat、slice

evenNumbers: function () {

return this.numbers.filter(function (number) {

return number % 2 === 0

})

}

1. **方法与事件：**
2. **方法处理器：**

v-on 指令监听 DOM 事件

1. **内联语句处理器：**

方法的调用方式，可以直接出入参数，或$event当前事件传入

1. **事件修饰符：**

.stop:等同于调用event.stopPropagetion()

.prevent:等同于调用event.preventDefault()

.capture:使用capture模式添加事件监听器，使用事件捕获模式

.self:只当事件是从监听元素本身时才会触发回调；

.once:事件将只会触发一次；

1. **按键修饰符：**

KeyCode: （键值）

<input v-on:keyup.13="submit">

按键别名：

enter、tab、delete、esc、space、up、down、left、right

1. **表单控件绑定：**
2. v-model 会忽略所有表单元素的 value、checked、selected 特性的初始值。
3. text、textarea
4. **复选框：**

**单选**：（值为true/false）

<input type="checkbox" id="checkbox" v-model="checked">

**多选**：（值为数组格式，数组中每一项为input的value值）

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

<input type="checkbox" id="john" value="John" v-model="checkedNames">

1. **单选按钮**

值为字符串，当前选中元素的value值；

<input type="radio" id="one" value="One" v-model="picked">

<input type="radio" id="two" value="Two" v-model="picked">

1. **选择列表**

**单选列表：（值为字符串）**

<select v-model="selected">

<option disabled value="">请选择</option>

<option>A</option>

<option>B</option>

<option>C</option>

</select>

**注：如果 v-model 表达初始的值不匹配任何的选项，<select> 元素就会以”未选中”的状态渲染。在 iOS 中，这会使用户无法选择第一个选项，因为这样的情况下，iOS 不会引发 change 事件。因此，像以上提供 disabled 选项是建议的做法**

**多选列表：**

1. 值为一个数组格式
2. 多选时，需要ctrl配合
3. 无实际意义
4. 比单选列表多一个multiple属性

<select v-model="selected" multiple style="width: 50px">

<option>A</option>

<option>B</option>

<option>C</option>

</select>

1. **绑定value：**

**（当单选或复选框选中时，此时的model值一般为boolen值，但是根据业务需求，有时我们需要动态绑定值，用法如下）**

1. **复选框：**

<input type="checkbox" v-model="toggle" v-bind:true-value="a" v-bind:false-value="b">

// 当选中时

vm.toggle === vm.a

// 当没有选中时

vm.toggle === vm.b

1. **单选框：**

<input type="radio" v-model="pick" v-bind:value="a">

// 当选中时

vm.pick === vm.a

1. **修饰符**

[lazy](http://v1-cn.vuejs.org/guide/forms.html" \l "lazy)：在 change 事件中同步

[number](http://v1-cn.vuejs.org/guide/forms.html" \l "number)：自动将用户的输入值转为 Number 类型

trim：自动过滤用户输入的首尾空格

1. **过渡：**

通过 Vue.js 的过渡系统，可以在元素从 DOM 中插入或移除时自动应用过渡效果。Vue.js 会在适当的时机为你触发 CSS 过渡或动画，你也可以提供相应JavaScript 钩子函数在过渡过程中执行自定义的 DOM 操作。

1. **经常使用工具：**

在 CSS 过渡和动画中自动应用 class

可以配合使用第三方 CSS 动画库，如 Animate.css

在过渡钩子函数中使用 JavaScript 直接操作 DOM

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

1. **Vue 提供了 transition 的封装组件，与下面情况一起使用：**

v-if

v-show

动态组件 （介绍见[组件](http://v1-cn.vuejs.org/guide/components.html" \l "动态组件)）

在组件的根节点上，并且被 Vue 实例 DOM 方法（如 vm.$appendTo(el)）触发。

1. **vue对transition组件的处理方式：**
2. 自动嗅探目标元素是否应用了 CSS 过渡或动画，如果是，在恰当的时机添加/删除 CSS 类名。
3. 如果过渡组件提供了 [JavaScript 钩子函数](https://cn.vuejs.org/v2/guide/transitions.html" \l "JavaScript-钩子)，这些钩子函数将在恰当的时机被调用
4. 如果没有找到 JavaScript 钩子并且也没有检测到 CSS 过渡/动画，DOM 操作（插入/删除）在下一帧中立即执行。
5. **类名的添加和切换取决于 transition 特性的值。比如 <transition name="fade">，会有六个 CSS 类名：**
6. 当dom插入时：
7. fade-enter: 定义进入过渡的开始状态。在元素被插入时生效，在下一个帧移除
8. fade-enter-active: 定义过渡的状态。在元素整个过渡过程中作用，在元素被插入时生效，在 transition/animation 完成之后移除。 这个类可以被用来定义过渡的过程时间，延迟和曲线函数。
9. fade-enter-to: **2.1.8版及以上** 定义进入过渡的结束状态。在元素被插入一帧后生效（于此同时 v-enter 被删除），在 transition/animation 完成之后移除。
10. 当dom移除时：
11. fade-leave: 定义离开过渡的开始状态。在离开过渡被触发时生效，在下一个帧移除。
12. fade-leave-active: 定义过渡的状态。在元素整个过渡过程中作用，在离开过渡被触发后立即生效，在 transition/animation 完成之后移除。 这个类可以被用来定义过渡的过程时间，延迟和曲线函数。
13. fade-leave-to: **2.1.8版及以上** 定义离开过渡的结束状态。在离开过渡被触发一帧后生效（于此同时 v-leave 被删除），在 transition/animation 完成之后移除。

注：如果一个动画在dom插入和删除时执行的效果正好相反，则只需要将v-enter和v-leave-to的样式设置成一样就可以；不需要对v-enter-to和v-leave进行设置，对于v-enter-active和v-leave-active可以进行相应的过渡时间和过渡曲线的设置；

1. **通过css的animation属性来实现过渡的效果:**

CSS 动画用法同 CSS 过渡，区别是在动画中 v-enter 类名在节点插入 DOM 后不会立即删除，而是在 animationend 事件触发时删除；

注：对于animation的使用，可以直接在定义一个动画，然后再v-enter-active和v-leave-active中用animation属性定义动画的曲线和时间即可；

1. **自定义过渡类名结合第三方动画库如Animate.css：**

http://cdn.bootcss.com/animate.css/3.5.2/animate.min.css

自定义的类名属性有：

enter-class

enter-active-class

enter-to-class (2.1.8+)

leave-class

leave-active-class

leave-to-class (2.1.8+)

注：他们的优先级高于普通的类名；

使用方法：

将上述属性添加到transition元素上，属性值为第三方库的动画名，可以不书写name属性定义的过渡名称，一般只使用enter-active-class和leave-active-class就可以满足需求书写方式：

注：属性值中一定要加入animated属性值，否则不生效；

如：

enter-active-class="animated bounce"

1. **css样式中与第三方动画库的结合：**

将动画定义到v-enter-active和v-leave-active类名上：

如：

.animations-enter-active,

.animations-leave-active {

animation: bounce 1s;

}

1. **同时使用transitions和animations时显示的声明过渡类型**

Vue 为了知道过渡的完成，dom要显示原有的样式，必须设置相应的事件监听器，它可以是 transitionend 或 animationend ，这取决于给元素应用的 CSS 规则。如果你使用其中任何一种，Vue 能自动识别类型并设置监听。

有时候同时使用transition 和animation ，而执行完毕的时间不统一，会造成后者无法将动画执行完毕，此时可以用type属性在transition元素上进行相应的设置需要 Vue 监听的类型，属性值为transition 或animation

1. **显示定义过渡效果的持续时间**

**应用场景：**

Vue 会等待其在过渡效果的根元素第一个 transitionend 或 animationend 事件触发后，结束过渡效果；一些嵌套的内部元素相比于过渡效果的根元素有延迟的或更长的过渡效果。

**使用方式：**

<transition :duration="1000">...</transition>

<transition :duration="{ enter: 500, leave: 800 }">...</transition>

1. **js中的钩子函数**

beforeEnter、enter、afterEnter、enterCancelled、beforeLeave、leave、afterLeave、leaveCancelled

结合第三方js动画库：<script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"></script>

**动画的书写格式：**

Velocity的参数：第一个为动画挂载的元素，第二个为定义的动画css样式，第三个参数为动画的时间或者回调函数

如：

Velocity（el,{fontSize:’1.5em’},{duration:300,complete:done}）

常用钩子函数为：beforeEnter 、enter、beforeLeave 、leave分别定义，元素进入时的动画和元素离开时的动画；

beforeEnter:定义元素进入时的初始样式；

enter：定义元素进入过程的动画样式(一般第三方库的动画使用在此钩子中)  
 beforeLeave：定义元素离开是的初始样式；

leave：定义元素离开过程的动画样式（一般第三方库的动画使用在此钩子中）

**注：一定注意在enter、leave钩子中一定使用done的方法的调用，否则动画将无效；**

1. **初始渲染的过渡;**

通过appear属性的添加实现初始渲染的过渡

1. **在一个transition元素中对多个标签动画的实现:**

需要给每一个元素添加一个key标签，防止元素的复用(不产生任何动画)

**过渡模式：**

场景：在两个元素的切换过程中，进入元素和离开元素的过渡同时进行，将会导致效果偏差；

解决：

可以在过渡的标签上添加过渡模式：（mode）

in-out: 新元素先进行过渡，完成之后当前元素过渡离开。(一般不用)

out-in: 当前元素先进行过渡，完成之后新元素过渡进入。

1. **多个组件的过渡**

一般使用动态组件的方式实现不同组件间的切换，然后在切换过程中实现过渡效果；

**注：动态组件的切换不存在标签的复用问题**

1. **列表过渡：**
2. **列表的进入和离开过渡**

使用transition-group组件，不同于transition组件，会以一个真实元素呈现，默认为span元素，也可以通过tag特性更换为其他元素；

内部元素必须提供唯一的key值

1. **列表的位移过渡**

v-move特性可以在元素的改变定位的过程中应用，可以通过name属性自定义前缀，也可以通过move-class手动设置

1. **组件**

组件 (Component) 是 Vue.js 最强大的功能之一。组件可以扩展 HTML 元素，封装可重用的代码。在较高层面上，组件是自定义元素，Vue.js 的编译器为它添加特殊功能。

1. 组件注册：

组件构造器：Vue.extend({ })，2.0中可以不使用构造器进行构造，直接写入对象；

规则：对于自定义标签名，Vue.js 不强制要求遵循 [W3C 规则](https://www.w3.org/TR/custom-elements/" \l "concepts" \t "_blank) (小写，并且包含一个短杠)，尽管遵循这个规则比较好

1. 全局注册：

全局注册需要在根实例初始化前注册，这样才能使组件在任意实例中被使用，方式：Vue.component(“组件名”,组件)

1. 局部注册：

组件只能在被注册的组件中使用，而无法在其他组件中使用.

1. 注册语法糖：

全局注册和局部注册(使用components属性，值为对象，组件名和组件定义对象是键值对的关系)；

1. 组件选项：
2. template：注意分析模板，is使用；

<table>

<tr is="my-row"></tr>

</table>

1. el和data两个属性，不可以直接赋值，比如data如果直接赋值，组件使用多次的时候，多个组件将公用一个对象；如果一个改变，则所有都会改变；所以使用function方式，每次return出一个新的对象；el类似；由于el很少直接在组件中挂载，所以可以忽略；
2. props：

组件间的作用域是孤立的，这里跟angular的指令作用域有所不同，angular指令的默认作用域是公用的，不管是在子指令还是父指令中定义的；在vue中，props起到了父组件向下与子组件间的桥梁作用；子组件给父组件发送消息，需要events；

* 驼峰命名：html的属性不区分大小写；
* 动态props：

通过v-bind进行绑定；

如果要进行数值的传递，必须通过动态绑定，否则直接传递过去的是个字符串；

* props的接受：

在子组件中使用props属性进行接受，props的值可以为数组（不可做校验），可以为对象（可以做校验）

* props数据的处理：

不应该在子组件中进行prop值的改变，因为prop的值是随父组件进行更变，如果在组件中直接对prop进行改变，vue则会抛出警告

正确的处理方式：

1. 定义一个局部变量，并用 prop 的值初始化它：（只会在初始化时，进行一次赋值，后期不会动态改变）

props: ['initialCounter'],

data: function () {

return { counter: this.initialCounter }

}

1. 定义一个计算属性，处理prop的值并返回：（进行属性值的动态跟踪）

props: ['size'],

computed: {

normalizedSize: function () {

return this.size.trim().toLowerCase()

}

}

* 绑定类型：

1. 默认时一直跟随父组件值得变化而变化，
2. .sync:双向绑定（取消）；在2.3.0以后,再次以事件监听实现双向数据流；如：

父组件：

<comp :foo="bar" @updata="val => bar = val"></comp>

子组件：

this.$emit('update:foo', newValue)

1. .once:单次绑定，子组件接受一次父组件传递的数据后，单独对这份数据进行维护；（2.0版本后取消此修饰符）

注：如果prop传递的是一个引用类型的数据，如对象或数组，即使单向绑定，子组件也会影响父组件；（传递为单向传递）

* Props验证：

1. 基础类型检测：prop：Number，接受的参数为原生的构造器：String、Number、Boolen、Function、Object、Aarry、Symbol、Null（意味着任何类型都行）
2. 可以为多种类型：prop：[Number，String]
3. 参数必须：prop:{type:Number,required:true} 对象的写法
4. 参数默认：prop {type:Number,default:0} 对象的写法
5. 绑定类型：prop：{twoWay:true} 校验数据绑定类型，如果非双向绑定会抛出一条警告；（这个不用了）
6. 自定义验证函数：prop：{validator：function(val){ return val>1},这样就可以验证一个数值是否大于1；
7. 非Prop属性：
8. 在组件上使用自定义属性，如果不在子组件中使用props进行接受，则此属性会被默认添加到组件的根元素上；
9. 在组件上使用html元素已有的属性，如class、style等，则组件上的属性值，会和组件的根节点的相应属性值进行合并等特性；如type等属性值，会进行覆盖或替换；如：组件中的input的type为number，使用组件时传入的type值为text，则组件内部的input的type值将被替换为text；
10. 组件间通信:
11. 直接访问：
12. $parent:父组件实例
13. $children：包含所有子组件实例
14. $root：组件所在的根实例；

这三个属性都挂载在组件的this上，不推荐这样做，会造成父子组件间的强耦合；

1. 自定义事件监听：
2. 组件实例化时，在组件的标签上使用@进行事件的监听；
3. 自定义事件触发机制：
4. $emit:在实例本身上触发事件》在子组件的钩子函数中去触发事件；
5. 子组件索引：

通过ref指令，绑定一个子组件，可以在父组件中使用this.$refs.属性名 的方式获取子组件；

1. 非父子通信：

思想：建立一个空的vue实例作为中央事件总线，即全局的vue实例；

代码如下：

var bus = new Vue()

// 触发组件 A 中的事件

bus.$emit('id-selected', 1)

// 在组件 B 创建的钩子中监听事件

bus.$on('id-selected', function (id) {

// ...

})

1. 内容分发：
2. 基础用法：

提供一种混合父组件内容与子组件自己模板的方式叫做内容分发；

Slot标签；

1. 使用需求：

当子组件中的dom结构以及展示的内容由父组件决定时

1. 编译作用域：

如果在子组件的模板中使用{{}}解析数据，则模板会解析自己作用域中的数据，而不是解析父组件作用中的数据；

1. 默认slot：

<slot>标签允许有一个匿名slot，不需要name值，作为找不到匹配的内容片段的回退插槽，如果没有默认的slot，这些找不到匹配的内容片段将被忽略；

1. 作用域插槽：

在父组件的slot模板中使用子组件中数据；如：

//子组件

<div class="child">

<slot text="hello from child"></slot>

</div>

//父组件

<div class="parent">

<child>

<template scope="props">

<span>hello from parent</span>

<span>{{ props.text }}</span>

</template>

</child>

</div>

在父级中，具有特殊属性 scope 的 <template> 元素必须存在，表示它是作用域插槽的模板。scope 的值对应一个临时变量名，此变量接收从子组件中传递的 props 对象；

在slot中可以动态绑定属性值，绑定的属性需要在父组件的scope中进行显示声明；

1. 动态组件：

1、基础用法：

component 标签上 is 属性决定了当前采用的子组件，:is 是 v-bind:is 的缩写，绑定了父 组件中 data 的 currentView 属性。顶部的 ul 则起到导航的作用，点击即可修改 currentView 值，也就修改 component 标签中使用的子组件类型，需要注意的事，currentView 的值需要 和父组件实例中的 components 属性的 key 相对应。（起到路由的作用）

2、keep-alive

在component 标签外层包裹一个 keep-alive 标签，可以将切换出去的组件保留在内存中，避免重新 渲染。

1. 杂项：
2. 异步组件：

局部定义组件写法：

new Vue({

components: {

//使用webpack+es6

'my-component': () => import('./my-async-component')

//使用webpack+es5

'my-component': require(['./my-async-component'], resolve)

}

})

使用webpack + es5的写法：

1. 当注册组件 (或者 props) 时，可以使用 kebab-case，camelCase，或 PascalCase。但是使用组件时只能用kebab-case这种写法；html不识别大小写；
2. 当组件中包含**大量**静态内容时，可以考虑使用 v-once 将渲染结果缓存起来

Vue.component('terms-of-service', {

template: '\

<div v-once>\

<h1>Terms of Service</h1>\

... a lot of static content ...\

</div>\

'

})

1. X-Templates

**另一种定义模版的方式是在 JavaScript 标签里使用 text/x-template 类型**

<script type="text/x-template" id="hello-world-template">

<p>Hello hello hello</p>

</script>

1. **指令**
2. 内置指令：
3. V-bind
4. V-model
5. V-if/ v-else-if/v-else/v-show
6. V-for
7. V-on
8. V-text
9. V-html
10. v-pre 指令相对简单，就是跳过编译这个元素和子元素，显示原始的 {{}}Mustache 标 签，用来减少编译时间。
11. v-cloak 指令相当于在元素上添加了一个 [v-cloak] 的属性，直到关联的实例结束编译。 官方推荐可以和 css 规则 [v-cloak]{ display :none } 一起使用，可以隐藏未编译的 Mustache 标签直到实例准备完毕
12. v-once 指令是 Vue.js 2.0 中新增的内置指令，用于标明元素或组件只渲染一次，即使随 后发生绑定数据的变化或更新，该元素或组件及包含的子元素都不会再次被编译和渲染。
13. 自定义指令基础：
14. 注册：通过 Vue.directive(id, definition) 方法注册一个全局自定义指令，接收参数 id 和定义对象。id 是指令的唯一标识，定义对象则是指令的相关属性及钩子函数。

也可以通过在组件的 directives 选项注册一个局部的自定 义指令

1. 指令的定义对象：

主要包含钩子函数：

bind: 只被调用一次，在指令第一次绑定到元素上时调用。用这个钩子函数可以定义一个在绑定时执行一次的初始化动作。

inserted: 被绑定元素插入父节点时调用（父节点存在即可调用，不必存在于 document 中）。

update ： 所在组件的 VNode 更新时调用，但是可能发生在其孩子的 **VNode 更新之前**。指令的值可能发生了改变也可能没有。但是你可以通过比较更新前后的值来忽略不必要的模板更新

componentUpdated: 所在组件的 VNode **及其孩子的 VNode** 全部更新时调用。

unbind ：指令从元素上解绑时调用，只调用一次。

注：如果我们只需要使用 update 函数时，可以直接传入一个函数代替定义对象；

指令绑定的值：

1. data 中的属性，使用<div v-my-directive="constant string"/></div>
2. 绑定字符串需要加单引号，<div v-my-direcitve="'constant string'"></div>
3. 利用字面修饰符后无需使用单引号

<div v-my-directive.literal="constant string"></div>

1. 受对象字面量或任意合法的 JavaScript 表达式:

<div v-my-directive="{ title : 'Vue.js', author : 'You'}" ></div>

<div v-my-directive="isExist ? 'yes' : 'no'" ></div>

1. 指令的实例属性：
2. **el: 指令所绑定的元素，可以用来直接操作 DOM 。**
3. **binding: 一个对象，包含以下属性：**

**name**: 指令名，不包括 v- 前缀。

**value**: 指令的绑定值， 例如： v-my-directive="1 + 1", value 的值是 2。

**oldValue**: 指令绑定的前一个值，仅在 update 和 componentUpdated 钩子中可用。无论值是否改变都可用。

**expression**: 绑定值的字符串形式。 例如 v-my-directive="1 + 1" ， expression 的值是 "1 + 1"。

**arg**: 传给指令的参数。例如 v-my-directive:foo， arg 的值是 "foo"。

**modifiers**: 一个包含修饰符的对象。 例如： v-my-directive.foo.bar, 修饰符对象 modifiers 的值是 { foo: true, bar: true }。

1. **vnode: Vue 编译生成的虚拟节点，查阅 [VNode API](https://cn.vuejs.org/v2/api/" \l "VNode接口) 了解更多详情。**
2. **oldVnode: 上一个虚拟节点，仅在 update 和 componentUpdated 钩子中可用。**
3. 指令与组件的区别：

组件一般是指一个独立实体，组件之间的关系通常都是树状。

指令是一种Decorator模式，用以改写某个组件的默认行为，或者增强使其获得额外功能，一般来说可以在同一个组件上叠加若干个指令，使其获得多种功能。

1. **vue-route**
2. **基本用法：**
3. **router-link：**
   1. <router-link> 组件支持用户在具有路由功能的应用中（点击）导航。 通**过 to 属性指定目标地址，默认渲染成带有正确链接的 <a> 标签，可以通过配置 tag 属性生成别的标签，<router-link> 比起写死的 <a href="..."> 会好一些；**
   2. **props:路由参数：**

to:格式为字符串或者是对象；

写法：

<router-link to="home">Home</router-link>

<router-link :to="'home'">Home</router-link>

<router-link :to="{ path: 'home' }">Home</router-link>

<router-link :to="{ name: 'user', params: { userId: 123 }}">User</router-link>

<router-link :to="{ path: 'register', query: { plan: 'private' }}">Register</router-link>

replace: 设置 replace 属性的话，当点击时，会调用 router.replace() 而不是 router.push()，于是导航后不会留下 history 记录

append: 在当前（相对）路径前添加基路径。例如，我们从 /a 导航到一个相对路径 b，如果没有配置 append，则路径为 /b，如果配了，则为 /a/b

注：另一个路径不允许加/；

tag:将touter-link转换为另外一种标签；

active-class：设置 链接激活时使用的 CSS 类名。默认值可以通过路由的构造选项 linkActiveClass 来全局配置。

1. **router-view**

渲染对应的路由配置中 components 下的相应组件；可以和<transition> 、 <keep-alive>等配合使用，两个结合一起用，要确保在内层使用 <keep-alive>：

<transition>

<keep-alive>

<router-view></router-view>

</keep-alive>

</transition>

1. **路由信息对象：**

路由信息对象表示当前激活的路由的状态信息，是不可变得，每次成功的导航都会产生一个新的对象；

**出现的地方：**

* 组件内部的钩子函数中；
* 导航的钩子函数中
* watch监听的$route对象中的（to,from）参数：

例：

watch: {

'$route' (to, from) {

} }

* scrollBehavior 方法的参数

**路由信息对象的属性：**

**$route.path：**字符串，对应当前路由的路径，总是解析为绝对路径，如 "/foo/bar"。

**$route.params：**包含了 动态片段 和 全匹配片段，如果没有路由参数，就是一个空对象。

**$route.query：**一个 key/value 对象，表示 URL 查询参数。例如，对于路径 /foo?user=1，则有 $route.query.user == 1，如果没有查询参数，则是个空对象。

**$route.matched：**一个数组，包含当前路由的所有嵌套路径片段的 **路由记录** 。路由记录就是 routes 配置数组中的对象副本（还有在 children 数组）

**$route.name：**当前路由的名称

1. **router构造配置**

**mode:** 配置路由模式:hash | history | abstract

**base：**应用的基路径。例如，如果整个单页应用服务在 /app/ 下，然后 base 就应该设为 "/app/"。

**linkActiveClass:** 全局配置 <router-link> 的默认『激活 class 类名

1. **router实例：**

**属性：**

**router.app:** 配置了 router 的 Vue 根实例。

**router.mode:** 路由使用的 [模式](https://router.vuejs.org/zh-cn/api/options.html" \l "mode)。

**router.currentRoute:**当前路由信息对象；

**方法：**

**router.push(location) :**进行路由的跳转，并且保存到浏览器的history中

**router.replace(location)：**进行路由的替换，不会将上次的路由保存到history中

**router.go(n)：**执行浏览器的前进动作,n代表前进的步数，负数代表后退

**router.back()：** 执行浏览器的后退动作

**router.forward()**

**router.addRoutes(routes):** 动态添加更多的路由规则。参数必须是一个符合 routes 选项要求的数组(小心有坑)

1. **动态路由匹配：**

**书写方式：**

{ path: '/user/:id', component: User }

{ path: '/user/:username/post/:post\_id', component: User }

**上述的动态值通过** $route.params去接受

**路由参数的变化：**

使用路由参数时，例如从 /user/foo 导航到 user/bar，原来的组件实例会被复用。因为两个路由都渲染同个组件，比起销毁再创建，复用则显得更加高效。不过，这也意味着组件的生命周期钩子不会再被调用；

**解决办法：**

 watch（监测变化） $route 对象：

1. **嵌套路由：**

在 VueRouter 的参数中使用 children 配置，属性值为路由对象数组；

**注意：在子路由的path属性中设置路由地址时，不可以加“/”，否则将会以根路由进行添加；**

1. **多视图多组件渲染：**

多视图对应多组件，使用的路由对象属性为：components；

例：

html：

<router-view class="view one"></router-view>

<router-view class="view two" name="a"></router-view>

<router-view class="view three" name="b"></router-view>

js：

{

path: '/',

components: {

default: Foo,

a: Bar,

b: Baz

}

}

1. **重定向：**

属性为redirect，属性值可以为{name：“foo”}或“foo”或者为一个方法的返回值

**例：**

{ path: '/a', redirect: { name: 'foo' }}

1. **别名：**

**例：**{ path: '/a', component: A, alias: '/b' }

当用户访问/b时，路由显示的是/b,其实规则为/a

1. **路由的钩子函数：**
2. **全局钩子：（一般不使用）**

**使用router的实例对象进行注册；**当一个导航触发时，全局的 beforeEach钩子按照创建顺序调用。钩子是异步解析执行，此时导航在所有钩子 resolve 完之前一直处于 **等待中**

**三个参数：**

**next()：** 进行管道中的下一个钩子

**next(false):** 中断当前的导航。重置到 from 路由对应的地址

**next('/') 或next({ path: '/' })：**当前导航被中断，重定向到另一个新的导航：

**例：**

const router = new VueRouter({ ... })

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

1. **路由的独享钩子：**

**在路由的配置项中定义：beforeEnter属性；**

{

path: '/foo',

component: Foo,

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

}

}

1. **组件内的路由钩子：**

beforeRouteEnter：路由跳转之前执行，不能获取this，组件实例还没有创建；

beforeRouteUpdate：路由重新跳转之前，该方法在组件复用时调用，该钩子函数中可以访问组件的实例this

beforeRouteLeave：离开该组件的对应路由时调用，可以访问实例this，一般在用户未保存信息时，不允许跳转相应的路由，next（false）阻止；

关于beforeRouteEnter钩子不能访问this，可以通过一个回调给next来访问组件实例，例：

beforeRouteEnter (to, from, next) {

next(vm => {

// 通过 `vm` 访问组件实例

})

}

1. **数据的获取**
2. **导航完成之后获取**：先完成导航，然后在接下来的组件生命周期钩子中获取数据。在数据获取期间显示『加载中』之类的指示。
3. **导航完成之前获取**：导航完成前，在路由的 enter 钩子中获取数据，在数据获取成功后执行导航。
4. **注：**

**导航完成之后获取需要借助组件的watch属性，在动态路由中组件复用时，进行数据的重新请求；**

**导航完成之前获取借助beforeRouteUpdate, 在动态路由中组件复用时，进行数据的重新请求；**

1. **组件的懒加载**
2. **vue的异步组件和webpack的require.ensure()相结合，例：**

var index = resolve => require.ensure(['@/components/index'], () => resolve(require('@/components/index')))

1. **采用amd风格的require，例：**

var Hello = resolve => require(['@/components/Hello'], resolve)

1. **把组件按组分块：**

提供 require.ensure 第三个参数作为 chunk 的名称，Webpack 将相同 chunk 下的所有异步模块打包到一个异步块里面 —— 这也意味着我们无须明确列出 require.ensure 的依赖（传空数组就行）

例：

const Foo = resolve => require.ensure([], () => resolve (require('./Foo.vue')), 'group-foo')

const Bar = resolve => require.ensure([], () => resolve (require('./Bar.vue')), 'group-foo')

const Baz = resolve => require.ensure([], () => resolve (require('./Baz.vue')), 'group-foo')

1. **状态管理：（vuex）**
2. **简介：**
   1. 在一些大型应用中，有时我们会遇到单页面中包含着大量的组件及复杂的数据结构，而 且可能各组件还会互相影响各自的状态，在这种情况下组件树中的事件流会很快变得非常复 杂，也使调试变得异常困难。为了解决这种情况，我们往往会引入状态管理这种设计模式，来 降低这种情况下事件的复杂程度并且使调试变得可以追踪。
   2. 每个组件都会拥有自己的状态，也可以理解成自身实例中的 data 对 象。用户在操作的过程中，会通过一些交互行为，例如点击、输入、拖动等修改组件的 状态，而此时往往需要将用户引起的状态变化通知到其他相关组件，让他们也进行对应 的修改。由于 Vue.js 本身的事件流是依赖于 DOM 结构的，组件修改状态后需要经过 一系列冒泡才能达到顶部的组件，而且如果需要修改兄弟组件的状态还需要共同的父组 件再进行一次广播。这种方式无疑是低效而且不易维护的，我们也很难去追踪事件流的 走向。
   3. vuex 和 [redux](https://github.com/reactjs/redux" \t "_blank) 间的差异表示关注，redux 是 React 生态环境中最流行的 Flux 实现。Redux 事实上无法感知视图层，所以它能够轻松的通过一些[简单绑定](https://github.com/egoist/revue" \t "_blank)和Vue一起使用。vuex区别在于它是一个专门为 vue 应用所设计。这使得它能够更好地和vue进行整合，同时提供简洁的API和改善过的开发体验。
3. **配置使用：**

**安装vuex，在main.js中引入vuex进行使用；**

import vuex from “vuex”,

Vue.use(vuex)

**实例化Vuex：**

const store = new Vuex.Store({

state: {

count: 0

},

mutations: {

increment (state) {

state.count++

}

}

})

**将vuex挂载到Vue实例中：**

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>

`

})

现在，你可以通过 store.state 来获取状态对象，以及通过 store.commit 方法触发状态变更，我们通过提交 mutation 的方式，而非直接改变 this.$store.state.count，是因为我们想要更明确地追踪到状态的变化

1. **核心概念：**
2. **state：**
   * **数据的定义以及存放数据的属性，类似于组件内部的data属性；所有在vuex使用的数据都必须在此属性中进行定义与声明；不建议将应用所有的数据都放入state，交互性低，比较单一使用的数据，建议放在组件内部定义；**
   * **mapState辅助函数**

**写法：**

**对象写法：**

**computed：mapState({**

**//写法一：箭头函数可使代码更简洁**

**count:state=> state.count**

**//写法二：传字符串参数 'count' 等同于 `state => state.count`**

**count:state=> state.count**

**//写法三：为了能够使用 `this` 获取局部状态，必须使用常规函数**

**count(state){**

**return state.count+this.a**

**}**

**})**

**数组写法：**

当映射的计算属性的名称与 state 的子节点名称相同时

**computed:mapState([**

**“count”**

**])**

* + **对象展开运算符：**

注意运用此方法时，因为对象扩展运算符是在es提案的stage-3阶段，由于babel进行转码时，在webpack中默认配置的是es-2015,所以需要借助babel-plugin-transform-object-rest-spread插件进行编译；

**在babel的配置文件中如下配置：**

"plugins": ["transform-runtime", "transform-object-rest-spread"],

**在computed属性中的写法如下：**

...mapState({

// ...

})

上面的mapState的参数可以是对象，也可以是数组，跟mapState辅助函数中的配置一样

1. **Getters：**

**应用场景：**

有时候，我们需要对state数据进行一些操作后再在组件中进行使用，如果多个组件都需要操作后的这个数据，我们可能会在多个组件computed中都对state进行操作一次；

**解决办法：**

**使用getters：**

getters: {

doneTodos: state => {

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

}

}

组件中使用，跟state的使用一样，可以使用辅助函数，或者对象展开运算符

1. **Mutations:**

更改 Vuex 的 store 中的状态的唯一方法是提交 mutation。Vuex 中的 mutations 非常类似于事件：每个 mutation 都有一个字符串的 **事件类型 (type)** 和 一个 **回调函数 (handler)**。这个回调函数就是我们实际进行状态更改的地方，并且它会接受 state 作为第一个参数；

定义的方式跟getters一样，例：

mutations: {

increment (state) {

// 变更状态

state.count++

}

}

回调函数的第一个参数为state，第二个参数可以动态传入值，值的类型，可以为基本类型或对象；

**提交载荷：**

可以使用store.commit(‘increment’,10)

或store.commit({type:’increment’,amount:10})

**遵循vue的数据相应规则:**

在对象上添加新属性时，使用 Vue.set(obj, 'newProp', 123)或者

state.obj = { ...state.obj, newProp: 123 }

**mutation必须是同步函数：**

如果为异步，则无法跟踪数据

**在组件中直接提交（不推荐）**

methods: {

...mapMutations([

'increment' // 映射 this.increment() 为 this.$store.commit('increment')

]),

...mapMutations({

add: 'increment' // 映射 this.add() 为 this.$store.commit('increment')

})

}

1. **actions：**

action提交的是mutation，而不是直接更变状态。

action可以包含任意异步操作。

**注册：**

actions: {

increment (context) {

context.commit('increment')

}

}

1. 在actions的内部可以获取以一个context参数。这个参数相当于一个store实例，但不是当前应用的store实例本身，因为如果采用modules方式进行数据的管理时，store实例是一个数组；
2. 可以通过context参数获取state或者getters等等，一般可以通过此参数来提交commit；

**分发action：**

**应用场景：当一次action对应的数据的更改，必须依赖另外一个action中的数据更改时，必须在本此action中先去触发另外一个action操作；另外一个action可能为同步，也可能为异步，如果为异步需要借助promise或者async函数；**

// 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

})

**在组件中分发action:**

methods: {

...mapActions([

'increment' // 映射 this.increment() 为 this.$store.dispatch('increment')

]),

...mapActions({

add: 'increment' // 映射 this.add() 为 this.$store.dispatch('increment')

})

}

**通过dispatch触发异步action：**

***使用promise:***

actionA ({ commit }) {

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

setTimeout(() => {

commit('someMutation')

resolve()

}, 1000)

})

}

actionB ({ dispatch, commit }) {

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

commit('someOtherMutation')

})

}

***使用async函数:***

getData、getOtherData返回的是promise，或者是axiso的数据请求函数

async actionA ({ commit }) {

commit('gotData', await getData())

},

async actionB ({ dispatch, commit }) {

await dispatch('actionA') // 等待 actionA 完成

commit('gotOtherData', await getOtherData())

}

1. **modules：**

**应用场景：**

1. 当一个项目比较大时，所有的数据与数据管理都放在同一个store中时，此时的store对象就会变得比较臃肿；
2. 当某个大型项目，几个开发团队一起开发时，同时操作一个store，可能会出现冲突等问题，每个团队负责的产品链的数据管理相对独立时，可以采用module的方式管理；

**定义：**

将 store 分割成**模块（module）**。每个模块拥有自己的 state、mutation、action、getter、甚至是嵌套子模块——从上至下进行同样方式的分割

如：

const moduleA = {

state: { ... },

mutations: { ... },

actions: { ... },

getters: { ... }

}

const store = new Vuex.Store({

modules: {

a: moduleA,

b: moduleB

}

})

store.state.a // -> moduleA 的状态

**模块的局部状态：**

对于模块内部的 mutation 和 getter，接收的第一个参数是模块的局部状态对象

对于模块内部的 action，局部状态通过 context.state 暴露出来， 根节点状态则为 context.rootState

在getter中根节点状态会作为第三个参数暴露出来