目录

[vue 2](#_Toc23809)

[● 响应式原理 2](#_Toc18617)

[● 双向数据绑定原理 3](#_Toc12793)

[● Proxy 与 Object.defineProperty 对比 3](#_Toc27118)

[● 虚拟 DOM 实现原理 4](#_Toc15395)

[● diff 4](#_Toc6068)

[● key 的作用 6](#_Toc3362)

[● 常见指令 7](#_Toc22631)

[● v-on和v-bind 7](#_Toc26650)

[● v-if 和 v-show 7](#_Toc18410)

[● 自定义指令 7](#_Toc4463)

[● 插槽 7](#_Toc30967)

[● scoped 属性 7](#_Toc14666)

[● 实现键盘监听 8](#_Toc2558)

[● data 为什么必须是函数 8](#_Toc16261)

[● keep-alive 8](#_Toc20987)

[● spa 优缺点 9](#_Toc17320)

[● spa 和 ssr 区别 9](#_Toc27876)

[● computed 和 watch 区别 10](#_Toc18972)

[● Vue 项目有哪些优化方式 11](#_Toc27805)

[Vue-生命周期 11](#_Toc18379)

[● Vue 2.0 生命周期 11](#_Toc23200)

[● Vue 3.0 生命周期 13](#_Toc12211)

[● nextTick 14](#_Toc22449)

[Vue-组件通信 15](#_Toc14917)

[● 所有组件通信方式 15](#_Toc21073)

[● 父子组件通信 15](#_Toc7298)

[● 事件总线Eventbus 17](#_Toc20957)

[● 所有组件通信方式 19](#_Toc21811)

[Vue-router 19](#_Toc879)

[● Vue-router 19](#_Toc10417)

[● Vue-router 懒加载 19](#_Toc24317)

[● vue-router 导航拦截 20](#_Toc26643)

[● vue-router 导航解析流程 20](#_Toc11857)

[● vue-router 路由模式 20](#_Toc18119)

[Vuex 21](#_Toc19653)

[● vuex 原理 21](#_Toc22998)

[● store 21](#_Toc28146)

[● vuex index.js 22](#_Toc30309)

[● state 22](#_Toc27304)

[● mapState 23](#_Toc4406)

[● getters 24](#_Toc20212)

[● getters 属性访问 25](#_Toc3892)

[● getters 方法访问 25](#_Toc30969)

[● mapGetters 25](#_Toc14154)

[● mutation 26](#_Toc22501)

[● mutation 载荷Payload(其实就是额外参数) 27](#_Toc14364)

[● mutation 组件提交/mapMutations 27](#_Toc4945)

[● action 28](#_Toc2136)

[● 分发Action 29](#_Toc26306)

[● 组件中分发Action/mapActions 29](#_Toc24247)

[● 组合Action 30](#_Toc6704)

[● module 31](#_Toc9962)

[● module局部状态 32](#_Toc1977)

[● module命名空间 32](#_Toc17157)

[● module组件调用 34](#_Toc24551)

# vue

* 响应式原理

[Vue数据的双向绑定、响应式与MVVM\_wytraining-程序员宅基地 - 程序员宅基地 (cxyzjd.com)](https://www.cxyzjd.com/article/wytraining/111040184)

[深入响应式原理 — Vue.js (vuejs.org)](https://cn.vuejs.org/v2/guide/reactivity.html)

[深入vue响应式原理（包含vue3.0） - 掘金 (juejin.cn)](https://juejin.cn/post/6844904078724562951)

|  |
| --- |
| Vue.js 实现****响应式****的核心是利用了 ES5 的 Object.defineProperty。  响应式的核心： vue初始化会用 Object.defineProperty()给data中的每一个属性添加getter和setter，同时创建dep和watcher进行依赖收集和派发更新，最后通过diff算法对比新旧vnode差异，通过patch即时更新DOM (通过 Object.defineProperty API 劫持数据的变化，在数据被访问的时候收集依赖，然后在数据被修改的时候通知依赖更新。)  Object.defineProperty(obj, prop, descriptor)  //obj 是要在其上定义属性的对象；  //prop 是要定义或修改的属性的名称；  //descriptor 是将被定义或修改的属性描述符。  比较核心的是 descriptor，它有很多可选键值。这里我们最关心的是 get 和 set，get 是一个给属性提供的 getter 方法，当我们访问了该属性的时候会触发 getter 方法；set 是一个给属性提供的 setter 方法，当我们对该属性做修改的时候会触发 setter 方法。  一旦对象拥有了 getter 和 setter，我们可以简单地把这个对象称为****响应式对象****。 |

* 双向数据绑定原理

[vue双向数据绑定原理 - 掘金 (juejin.cn)](https://juejin.cn/post/6844903854740340749)

[面试官：说说你对双向绑定的理解? · Issue #2 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/2)

|  |
| --- |
| ****什么是数据劫持**** vue.js采用数据劫持结合发布者-订阅者模式的方式，通过 Object.defineProperty() 来劫持各个属性的 getter/setter ， 在数据变动时发布消息给订阅者，触发响应的监听回调。  ****基于数据劫持双向绑定的实现思路**** a. 利用Proxy或Object.defineProperty生成的Observer针对对象/对象的属性进行"劫持",在属性发生变化后通知订阅者 b. 解析器Compile解析模板中的Directive(指令)，收集指令所依赖的方法和数据,等待数据变化然后进行渲染 c. Watcher属于Observer和Compile桥梁，它将接收到的Observer产生的数据变化，并根据Compile提供的指令进行视图渲染，使得数据变化促使视图变化 IMG_256  ****v-model指令****  1.v-model的原理：v-bind数据绑定 与 v-on处理函数绑定的语法糖  <input v-model="value">  2.想要v-model生效，它必须：   1. 接收一个value属性：data => DOM 属性绑定（v-bind）：把data的值绑定到元素上，用于显示数据 2. 在value值改变的时候，触发input事件：DOM => data 事件绑定（v-on）：对元素绑定一个事件，当值变化后把数据传递给控制层，进而来影响数据层   <input v-bind:value="value" v-on:input="value=$event.target.value"> |

* Proxy 与 Object.defineProperty 对比

<https://www.cxyzjd.com/article/wytraining/111040184>

<https://juejin.cn/post/7063661600615432200>

|  |
| --- |
| Object.defineProperty 与 Proxy  1.Object.defineProperty：   * 只能对属性进行劫持，需要递归遍历对象的每个属性，执行Object.defineProperty把每一层对象数据都变成响应式的（如果定义的响应式数据过于复杂，会有很大的性能负担） * 不能检测对象属性的添加和删除（需要重新遍历）   2.Proxy   * 在getter中去递归响应式，真正访问到的内部对象才会变成响应式，而不是无脑递归，提升了性能 * 劫持的是整个对象，能检测到对象属性的添加和删除 |

* 虚拟 DOM 实现原理

[怎么理解虚拟 DOM？ - liudddd - 博客园 (cnblogs.com)](https://www.cnblogs.com/ldld/p/14037487.html" \l ":~:text=%E8%99%9A%E6%8B%9F DOM%EF%BC%88Virtual DOM%EF%BC%89%E6%9C%AC%E8%B4%A8%E4%B8%8A%E6%98%AFJS %E5%92%8C DOM,%E4%B9%8B%E9%97%B4%E7%9A%84%E4%B8%80%E4%B8%AA%E6%98%A0%E5%B0%84%E7%BC%93%E5%AD%98%EF%BC%8C%E5%AE%83%E5%9C%A8%E5%BD%A2%E6%80%81%E4%B8%8A%E8%A1%A8%E7%8E%B0%E4%B8%BA%E4%B8%80%E4%B8%AA%E8%83%BD%E5%A4%9F%E6%8F%8F%E8%BF%B0 DOM %E7%BB%93%E6%9E%84%E5%8F%8A%E5%85%B6%E5%B1%9E%E6%80%A7%E4%BF%A1%E6%81%AF%E7%9A%84 JS %E5%AF%B9%E8%B1%A1%E3%80%82)

[面试官：什么是虚拟DOM？如何实现一个虚拟DOM？说说你的思路 | web前端面试 - 面试官系列 (vue3js.cn)](https://vue3js.cn/interview/vue/vnode.html" \l "%E4%B8%80%E3%80%81%E4%BB%80%E4%B9%88%E6%98%AF%E8%99%9A%E6%8B%9Fdom)

|  |
| --- |
| 实际上它只是一层对真实DOM的抽象，以JavaScript 对象 (VNode 节点) 作为基础的树，用对象的属性来描述节点，最终可以通过一系列操作使这棵树映射到真实环境上。  在Javascript对象中，虚拟DOM 表现为一个 Object对象。并且最少包含标签名 (tag)、属性 (attrs) 和子元素对象 (children) 三个属性，不同框架对这三个属性的名命可能会有差别。  创建虚拟DOM就是为了更好将虚拟的节点渲染到页面视图中，所以虚拟DOM对象的节点与真实DOM的属性一一照应。  你用传统的原生api或jQuery去操作DOM时，浏览器会从构建DOM树开始从头到尾执行一遍流程  当你在一次操作时，需要更新10个DOM节点，浏览器没这么智能，收到第一个更新DOM请求后，并不知道后续还有9次更新操作，因此会马上执行流程，最终执行10次流程  而通过VNode，同样更新10个DOM节点，虚拟DOM不会立即操作DOM，而是将这10次更新的diff内容保存到本地的一个js对象中，最终将这个js对象一次性attach到DOM树上，避免大量的无谓计算  很多人认为虚拟 DOM 最大的优势是 diff 算法，减少 JavaScript 操作真实 DOM 的带来的性能消耗。虽然这一个虚拟 DOM 带来的一个优势，但并不是全部。虚拟 DOM 最大的优势在于抽象了原本的渲染过程，实现了跨平台的能力，而不仅仅局限于浏览器的 DOM，可以是安卓和 IOS 的原生组件，可以是近期很火热的小程序，也可以是各种GUI |

* diff

[面试官：了解过vue中的diff算法吗？说说看 · Issue #24 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/24)

|  |
| --- |
| diff 算法是一种通过同层的树节点进行比较的高效算法  其有两个特点：   * 比较只会在同层级进行, 不会跨层级比较 * 在diff比较的过程中，循环从两边向中间比较   diff 算法在很多场景下都有应用，在 vue 中，作用于虚拟 dom 渲染成真实 dom 的新旧 VNode 节点比较  diff整体策略为：深度优先，同层比较  当数据发生改变时，set方法会调用Dep.notify通知所有订阅者Watcher，订阅者就会调用patch给真实的DOM打补丁，更新相应的视图。 |

|  |
| --- |
| [虚拟DOM和Diff算法 - 入门级 - 掘金 (juejin.cn)](https://juejin.cn/post/6844903767473651720)  虚拟DOM：  将真实的dom节点用JavaScript来模拟出来，而Dom变化的对比，放到 Js 层来做。  只是一层对真实DOM的抽象，以JavaScript 对象 (VNode 节点) 作为基础的树，用对象的属性来描述节点，最终可以通过一系列操作使这棵树映射到真实环境上。  tag表示标签名，attrs就是dom的属性，每个dom如果有children的话，就会在children中以数组的形式展示，数组的每一项就又是一个虚拟dom结构。  createElement 创建 VNode 的过程，每个 VNode 有 children，children 每个元素也是一个VNode，这样就形成了一个虚拟树结构，用于描述真实的DOM树结构。  虚拟 DOM 解决的关键问题有以下两个。  1、研发体验/研发效率的问题：这一点前面已经反复强调过，DOM 操作模式的每一次革新，背后都是前端对效率和体验的进一步追求。虚拟 DOM 的出现，为数据驱动视图这一思想提供了高度可用的载体，使得前端开发能够基于函数式 UI 的编程方式实现高效的声明式编程。  2、跨平台的问题：虚拟 DOM 是对真实渲染内容的一层抽象。若没有这一层抽象，那么视图层将和渲染平台紧密耦合在一起，为了描述同样的视图内容，你可能要分别在 Web 端和 Native 端写完全不同的两套甚至多套代码。但现在中间多了一层描述性的虚拟 DOM，它描述的东西可以是真实 DOM，也可以是iOS 界面、安卓界面、小程序......同一套虚拟 DOM，可以对接不同平台的渲染逻辑，从而实现“一次编码，多端运行”。 |
| diff算法：  diff 算法是一种通过同层的树节点进行比较的高效算法  其有两个特点：  比较只会在同层级进行, 不会跨层级比较  在diff比较的过程中，循环从两边向中间比较  当数据发生改变时，set方法会调用Dep.notify通知所有订阅者Watcher，订阅者就会调用patch给真实的DOM打补丁，更新相应的视图   * 当数据发生改变时，订阅者watcher就会调用patch给真实的DOM打补丁 * 通过isSameVnode进行判断，相同则调用patchVnode方法 * patchVnode做了以下操作：   + 找到对应的真实dom，称为el   + 如果都有都有文本节点且不相等，将el文本节点设置为Vnode的文本节点   + 如果oldVnode有子节点而VNode没有，则删除el子节点   + 如果oldVnode没有子节点而VNode有，则将VNode的子节点真实化后添加到el   + 如果两者都有子节点，则执行updateChildren函数比较子节点 * updateChildren主要做了以下操作：   + 设置新旧VNode的头尾指针   + 新旧头尾指针进行比较，循环向中间靠拢，根据情况调用patchVnode进行patch重复流程、调用createElem创建一个新节点，从哈希表寻找 key一致的VNode 节点再分情况操作 |
| 虚拟diff是基于虚拟dom的一个算法。虚拟dom是react自己维护的一个对象，这个对象可以和真实dom的元素一一对应。react可以基于虚拟dom的变化来改变真实dom，并且尽量进行更少的操作。虚拟diff算法就是用于快速找出和上一次的虚拟dom的异同，并基于上一次的dom进行高效的修改操作。  虚拟diff算法是这样实现的：虚拟diff是让新旧虚拟dom对象进行同层遍历，一旦发现节点不同，则将对整个节点及其后面的子节点全部删除，并重新添加。哪怕你只是跨层级的交换对象，也会导致后面的元素删除。  当然，这样是有一定的弊端的，如果在同层节点上面，只进行节点的顺序修改，但按照上述发现不同就立刻删除的操作，是不合适的。因此引入了一个key机制，key机制就可以用于发现同层之间是否只是节点的顺序改变，进而不直接删除和添加。  [贝壳找房面试 | BK201-Drama](https://bk201-drama.github.io/2022/03/26/%E8%B4%9D%E5%A3%B3%E6%89%BE%E6%88%BF%E9%9D%A2%E8%AF%95/) |

* key 的作用

[面试官：你知道vue中key的原理吗？说说你对它的理解？ · Issue #18 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/18)

[(2条消息) Vue中key属性的作用\_考拉比利的博客-CSDN博客\_key属性](https://blog.csdn.net/zyj362633491/article/details/86654014)

|  |
| --- |
| key是给每一个vnode的唯一id，也是diff的一种优化策略，可以根据key，更准确， 更快的找到对应的vnode节点  [#](https://vue3js.cn/interview/vue/key.html" \l "%E5%9C%BA%E6%99%AF%E8%83%8C%E5%90%8E%E7%9A%84%E9%80%BB%E8%BE%91)**场景背后的逻辑**  当我们在使用v-for时，需要给单元加上key  如果不用key，Vue会采用就地复地原则：最小化element的移动，并且会尝试尽最大程度在同适当的地方对相同类型的element，做patch或者reuse。  如果使用了key，Vue会根据keys的顺序记录element，曾经拥有了key的element如果不再出现的话，会被直接remove或者destoryed  用+new Date()生成的时间戳作为key，手动强制触发重新渲染   * 当拥有新值的rerender作为key时，拥有了新key的Comp出现了，那么旧key Comp会被移除，新key Comp触发渲染   总体来说，当使用列表渲染时，永远添加key属性，这样可以提高列表渲染的效率，提高了页面的性能。 |

* 常见指令

[(4条消息) Vue基础——常用指令\_南北老头的博客-CSDN博客](https://blog.csdn.net/NanBeiW/article/details/109509227" \l ":~:text=Vue%E5%B8%B8%E7%94%A8%E6%8C%87%E4%BB%A4 vue .js%E5%AE%98%E6%96%B9%E7%BB%99%E8%87%AA%E5%B7%B1%E7%9A%84%E5%AE%9A%E4%B8%BA%E6%98%AF%E6%95%B0%E6%8D%AE%E6%A8%A1%E6%9D%BF%E5%BC%95%E6%93%8E,%E5%B9%B6%E7%BB%99%E5%87%BA%E4%BA%86%E4%B8%80%E5%A5%97%E6%B8%B2%E6%9F%93%E6%95%B0%E6%8D%AE%E7%9A%84 %E6%8C%87%E4%BB%A4. v-if v-show v-for v-text,%E5%AE%9A%E4%B9%89%E5%AF%B9%E8%B1%A1%E5%8F%AF%E4%BB%A5%E6%8F%90%E4%BE%9B%E5%A6%82%E4%B8%8B%E5%87%A0%E4%B8%AA%E9%92%A9%E5%AD%90%E5%87%BD%E6%95%B0 (%E5%9D%87%E4%B8%BA%E5%8F%AF%E9%80%89)%EF%BC%9A bind%EF%BC%9A%E5%8F%AA%E8%B0%83%E7%94%A8%E4%B8%80%E6%AC%A1%EF%BC%8C %E6%8C%87%E4%BB%A4 %E7%AC%AC%E4%B8%80%E6%AC%A1%E7%BB%91%E5%AE%9A%E5%88%B0%E5%85%83%E7%B4%A0%E6%97%B6%E8%B0%83%E7%94%A8%E3%80%82 %E5%9C%A8%E8%BF%99%E9%87%8C%E5%8F%AF%E4%BB%A5%E8%BF%9B%E8%A1%8C%E4%B8%80%E6%AC%A1%E6%80%A7%E7%9A%84%E5%88%9D%E5%A7%8B%E5%8C%96%E8%AE%BE%E7%BD%AE%E3%80%82 inserted%EF%BC%9A%E8%A2%AB%E7%BB%91%E5%AE%9A%E5%85%83%E7%B4%A0%E6%8F%92%E5%85%A5%E7%88%B6%E8%8A%82%E7%82%B9%E6%97%B6%E8%B0%83%E7%94%A8 (%E4%BB%85%E4%BF%9D%E8%AF%81%E7%88%B6%E8%8A%82%E7%82%B9%E5%AD%98%E5%9C%A8%EF%BC%8C%E4%BD%86%E4%B8%8D%E4%B8%80%E5%AE%9A%E5%B7%B2%E8%A2%AB%E6%8F%92%E5%85%A5%E6%96%87%E6%A1%A3%E4%B8%AD)%E3%80%82)

|  |
| --- |
| V-show  V-on  V-model  V-for  V-if  V-else-if  V-else |

* v-on和v-bind

|  |
| --- |
| ****v -bind**** ：是用来绑定属性 的 。****v -on**** ：用来绑定事件 的 。****v**** -****on**** 使用 的 时候一般写在methods方法里面 v -****bind**** 使用 的 时候一般写在data里面。 |

* v-if 和 v-show

[面试官：Vue中的v-show和v-if怎么理解？ · Issue #4 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/4)

|  |
| --- |
| <Model v-show="isShow" />  <Model v-if="isShow" />   * 当表达式为true的时候，都会占据页面的位置 * 当表达式都为false时，都不会占据页面位置   控制手段：v-show隐藏则是为该元素添加css--display:none，dom元素依旧还在。v-if显示隐藏是将dom元素整个添加或删除.  编译过程：v-if切换有一个局部编译/卸载的过程，切换过程中合适地销毁和重建内部的事件监听和子组件；v-show只是简单的基于css切换  编译条件：v-if是真正的条件渲染，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。只有渲染条件为假时，并不做操作，直到为真才渲染  v-show 由false变为true的时候不会触发组件的生命周期  v-if由false变为true的时候，触发组件的beforeCreate、create、beforeMount、mounted钩子，由true变为false的时候触发组件的beforeDestory、destoryed方法  性能消耗：v-if有更高的切换消耗；v-show有更高的初始渲染消耗； |

* 自定义指令

[Vue.js 自定义指令 | 菜鸟教程 (runoob.com)](https://www.runoob.com/vue2/vue-custom-directive.html)

[这15个Vue自定义指令，让你的项目开发爽到爆 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/108308393)

* 插槽

[Vue3.0的插槽是如何实现的？ - 掘金 (juejin.cn)](https://juejin.cn/post/7058605762070708232)

[面试官：说说你对slot的理解？slot使用场景有哪些？ · Issue #16 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/16)

* scoped 属性

[vue：style标签中的scoped属性（作用域）和lang属性的介绍 - 云+社区 - 腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/1889827)

|  |
| --- |
| scoped是指定样式的局部作用域。  在vue中：App.vue相当于根[容器](https://cloud.tencent.com/product/tke?from=10680" \t "https://cloud.tencent.com/developer/article/_blank)，不设置scoped。所以一般在App.vue中引用公共样式。而在其它.vue页面中用scoped，代表当前样式只作用于当前.vue页面。不作用于其它.vue页面。 |

* 实现键盘监听

[vue监听键盘事件相关 - 掘金 (juejin.cn)](https://juejin.cn/post/6921536009859497991)

|  |
| --- |
| <!-- 只有在 `key` 是 `Enter` 时调用 `vm.submit()` -->  <input v-on:keyup.enter="submit"> |

* data 为什么必须是函数

[面试官：为什么data属性是一个函数而不是一个对象？ · Issue #9 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/9)

|  |
| --- |
| * 根实例对象data可以是对象也可以是函数（根实例是单例），不会产生数据污染情况 * 组件实例对象data必须为函数，目的是为了防止多个组件实例对象之间共用一个data，产生数据污染。采用函数的形式，initData时会将其作为工厂函数都会返回全新data对象 |

* keep-alive

[面试官：怎么缓存当前的组件？缓存后怎么更新？说说你对keep-alive的理解是什么？ · Issue #19 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/19)

|  |
| --- |
| keep-alive是vue中的内置组件，能在组件切换过程中将状态保留在内存中，防止重复渲染DOM。  使用原则：当我们在某些场景下不需要让页面重新加载时我们可以使用keepalive  举个栗子:  当我们从首页–>列表页–>商详页–>再返回，这时候列表页应该是需要keep-alive  从首页–>列表页–>商详页–>返回到列表页(需要缓存)–>返回到首页(需要缓存)–>再次进入列表页(不需要缓存)，这时候可以按需来控制页面的keep-alive  在路由中设置keepAlive属性判断是否需要缓存  {  path: 'list',  name: 'itemList', // 列表页  component (resolve) {  require(['@/pages/item/list'], resolve)  },  meta: {  keepAlive: true,  title: '列表页'  }}  使用<keep-alive>  <div id="app" class='wrapper'>  <keep-alive>  <!-- 需要缓存的视图组件 -->  <router-view v-if="$route.meta.keepAlive"></router-view>  </keep-alive>  <!-- 不需要缓存的视图组件 -->  <router-view v-if="!$route.meta.keepAlive"></router-view></div> |

* spa 优缺点

[面试官：说说你对SPA（单页应用）的理解? · Issue #3 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/3)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 我们熟知的JS框架如react,vue,angular,ember都属于SPA   |  | **单页面应用（SPA）** | **多页面应用（MPA）** | | --- | --- | --- | | 组成 | 一个主页面和多个页面片段 | 多个主页面 | | 刷新方式 | 局部刷新 | 整页刷新 | | url模式 | 哈希模式 | 历史模式 | | SEO搜索引擎优化 | 难实现，可使用SSR方式改善 | 容易实现 | | 数据传递 | 容易 | 通过url、cookie、localStorage等传递 | | 页面切换 | 速度快，用户体验良好 | 切换加载资源，速度慢，用户体验差 | | 维护成本 | 相对容易 | 相对复杂 |  **单页应用优缺点** 优点：   * 具有桌面应用的即时性、网站的可移植性和可访问性 * 用户体验好、快，内容的改变不需要重新加载整个页面 * 良好的前后端分离，分工更明确   缺点：   * 不利于搜索引擎的抓取 * 首次渲染速度相对较慢 |

* spa 和 ssr 区别

[SPA和SSR页面的区别 | 8月更文挑战 - 掘金 (juejin.cn)](https://juejin.cn/post/6993613932296404999)

|  |
| --- |
| 1. ****SSR服务端渲染****   将组件或页面通过服务器生成html，再返回给浏览器，如nuxt.js |
| SPA  优点：客户端渲染、数据传输量小、减少服务器端压力、交互/响应速度快、前后端完全分离；  缺点：首屏加载慢、对SEO不友好，不利于百度，360等搜索引擎收录快照；  SSR  优点：对于SEO友好、首屏加载速度快；  缺点：页面重复加载次数高、开发效率低、数据传输量大、服务器压力大；  SPA、SSR分别适合什么样的应用场景？  ****SPA****：对项目性能要求高、页面加载速度快、要求客户端渲染、对SEO要求低；  ****SSR****：对项目SEO要求高、首次打开响应速度快； |

* computed 和 watch 区别

[Vue核心知识：computed、methods和watch的区别 - 云+社区 - 腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/1705551" \l ":~:text=methods%EF%BC%8Cwatch %E5%92%8C computed %E9%83%BD%E6%98%AF%E4%BB%A5%E5%87%BD%E6%95%B0%E4%B8%BA%E5%9F%BA%E7%A1%80%E7%9A%84%EF%BC%8C%E4%BD%86%E5%90%84%E8%87%AA%E5%8D%B4%E9%83%BD%E4%B8%8D%E5%90%8C watch,%E5%92%8C computed %E9%83%BD%E6%98%AF%E4%BB%A5 Vue %E7%9A%84%E4%BE%9D%E8%B5%96%E8%BF%BD%E8%B8%AA%E6%9C%BA%E5%88%B6%E4%B8%BA%E5%9F%BA%E7%A1%80%E7%9A%84%EF%BC%8C%E5%BD%93%E6%9F%90%E4%B8%80%E4%B8%AA%E6%95%B0%E6%8D%AE%E5%8F%91%E7%94%9F%E5%8F%98%E5%8C%96%E7%9A%84%E6%97%B6%E5%80%99%EF%BC%8C%E6%89%80%E6%9C%89%E4%BE%9D%E8%B5%96%E8%BF%99%E4%B8%AA%E6%95%B0%E6%8D%AE%E7%9A%84%E2%80%9C%E7%9B%B8%E5%85%B3%E2%80%9D%E6%95%B0%E6%8D%AE%E2%80%9C%E8%87%AA%E5%8A%A8%E2%80%9D%E5%8F%91%E7%94%9F%E5%8F%98%E5%8C%96%EF%BC%8C%E4%B9%9F%E5%B0%B1%E6%98%AF%E8%87%AA%E5%8A%A8%E8%B0%83%E7%94%A8%E7%9B%B8%E5%85%B3%E7%9A%84%E5%87%BD%E6%95%B0%E5%8E%BB%E5%AE%9E%E7%8E%B0%E6%95%B0%E6%8D%AE%E7%9A%84%E5%8F%98%E5%8A%A8)

[VUE computed和watch的区别 - 简书 (jianshu.com)](https://www.jianshu.com/p/5f83e06ae32f)

|  |
| --- |
| **watch 和 computed 区别**   1. 功能上：computed是计算属性，watch是监听一个值的变化，然后执行对应的回调。 2. 是否调用缓存：computed中的函数所依赖的属性没有发生变化，那么调用当前的函数的时候会从缓存中读取，而watch在每次监听的值发生变化的时候都会执行回调。 3. 是否调用return：computed中的函数必须要用return返回，watch中的函数不是必须要用return 4. watch擅长处理的场景：一个数据影响多个数据 -------搜索框。 5. computed擅长处理的场景：一个数据受多个数据影响 -- 使用场景：当一个值受多个属性影响的时候--------购物车商品结算 |
| 不需要在data  new Vue({  el: '#root',  template: ` <div> <span>Name: {{name}}</span> </div> `,  data: { firstName: 'Jokcy', lastName: 'Lou' },  computed: {  name () {  return `${this.firstName} ${this.lastName}`  }  }  }) |
| 需要在data或者传入的props  watch: {  obj: {  handler(newName, oldName) {  console.log('obj.a changed');  },  immediate: true,  deep: true  }} |

* Vue 项目有哪些优化方式

# Vue-生命周期

* Vue 2.0 生命周期

[(4条消息) 简单说说vue中的el属性\_Mingeon的博客-CSDN博客\_el属性](https://blog.csdn.net/qq_36512241/article/details/113055174)

[Vue - 生命周期详解 - 简书 (jianshu.com)](https://www.jianshu.com/p/672e967e201c)

[面试官：说说你对Vue生命周期的理解? · Issue #6 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/6)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **生命周期** | **描述** | | --- | --- | | beforeCreate | 执行时组件实例还未创建，通常用于插件开发中执行一些初始化任务 | | created | 组件初始化完毕，各种数据可以使用，常用于异步数据获取 | | beforeMount | 未执行渲染、更新，dom未创建 | | mounted | 初始化结束，dom已创建，可用于获取访问数据和dom元素 | | beforeUpdate | 更新前，可用于获取更新前各种状态 | | updated | 更新后，所有状态已是最新 | | beforeDestroy | 销毁前，可用于一些定时器或订阅的取消 | | destroyed | 组件已销毁，作用同上 |   看生命周期的图。  [44114780-3aca-11eb-85f6-6fac77c0c9b3.png (1080×1558) (vue-js.com)](https://static.vue-js.com/44114780-3aca-11eb-85f6-6fac77c0c9b3.png)  created是在组件实例一旦创建完成的时候立刻调用，这时候页面dom节点并未生成  mounted是在页面dom节点渲染完毕之后就立刻执行的.  触发时机上created是比mounted要更早的两者相同点：都能拿到实例对象的属性和方法  讨论这个问题本质就是触发的时机，放在mounted请求有可能导致页面闪动（页面dom结构已经生成），但如果在页面加载前完成则不会出现此情况建议：放在create生命周期当中 |
| beforeCreated之前肯定不可以。  中间过程不确定，但created已经初始化完了。  也就是说，在created之后（包括created）才能调用data，methods |
| mounted：Vue实例初始化完毕，访问数据和DOM操作。 |
| beforeUpdate：data数据最新，页面未更新  updated：data和页面最新 |
| beforeDestroy：身上的data和method都可用  destroyed：完全销毁不可用 |

* Vue 3.0 生命周期

[Vue 3 生命周期完整指南 - 腾讯云开发者社区-腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/1808384)

|  |
| --- |
| **在**组合API**中使用Vue 3生命周期钩子**  在组合API中，我们需要将生命周期钩子导入到项目中，才能使用，这有助于保持项目的轻量性。  // 组合 APIimport { onMounted } from 'vue'  除了beforecate和created(它们被setup方法本身所取代)，我们可以在setup方法中访问的API生命周期钩子有9个选项:   * onBeforeMount – 在挂载开始之前被调用：相关的 render 函数首次被调用。 * onMounted – 组件挂载时调用 * onBeforeUpdate – 数据更新时调用，发生在虚拟 DOM 打补丁之前。这里适合在更新之前访问现有的 DOM，比如手动移除已添加的事件监听器。 * onUpdated – 由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。 * onBeforeUnmount – 在卸载组件实例之前调用。在这个阶段，实例仍然是完全正常的。 * onUnmounted – 卸载组件实例后调用。调用此钩子时，组件实例的所有指令都被解除绑定，所有事件侦听器都被移除，所有子组件实例被卸载。 * onActivated – 被 keep-alive 缓存的组件激活时调用。 * onDeactivated – 被 keep-alive 缓存的组件停用时调用。 * onErrorCaptured – 当捕获一个来自子孙组件的错误时被调用。此钩子会收到三个参数：错误对象、发生错误的组件实例以及一个包含错误来源信息的字符串。此钩子可以返回 false 以阻止该错误继续向上传播。   使用事例：  // 组合 API  import { onMounted } from 'vue'  export default {  setup () {  onMounted(() => {  console.log('mounted in the composition api!')  })  }}  **将 Vue2 的生命周期钩子代码更新到 Vue3**  这个从Vue2 到Vue3的生命周期映射是直接从[Vue 3 Composition API](https://v3.vuejs.org/guide/composition-api-introduction.html" \l "watcheffect" \t "https://cloud.tencent.com/developer/article/_blank)文档中获得的:   * beforeCreate -> 使用 setup() * created -> 使用 setup() * beforeMount -> onBeforeMount * mounted -> onMounted * beforeUpdate -> onBeforeUpdate * updated -> onUpdated * beforeDestroy -> onBeforeUnmount * destroyed -> onUnmounted * errorCaptured -> onErrorCaptured |

* nextTick

[面试官：Vue中的$nextTick有什么作用？ · Issue #14 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/14)

[面试题：Vue中$nextTick原理 - 云+社区 - 腾讯云 (tencent.com)](https://cloud.tencent.com/developer/article/1633546)

[深入浅出理解vm.$nextTick - 掘金 (juejin.cn)](https://juejin.cn/post/6844904090657357837)

|  |
| --- |
| Vue 在更新 DOM 时是异步执行的。只要侦听到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据变更。如果同一个 watcher 被多次触发，只会被推入到队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的。然后，在下一个的事件循环“tick”中，Vue 刷新队列并执行实际 (已去重的) 工作。Vue 在内部对异步队列尝试使用原生的 Promise.then、MutationObserver 和 setImmediate，如果执行环境不支持，则会采用 setTimeout(fn, 0) 代替。    也就是说我们在设置this.msg = 'some thing'的时候，Vue并没有马上去更新DOM数据，而是将这个操作放进一个队列中；如果我们重复执行的话，队列还会进行去重操作；等待同一事件循环中的所有数据变化完成之后，会将队列中的事件拿出来处理。    这样做主要是为了提升性能，因为如果在主线程中更新DOM，循环100次就要更新100次DOM；但是如果等事件循环完成之后更新DOM，只需要更新1次。还不了解事件循环的童鞋，记得对小壹大喊一声【事件循环】哦。    为了在数据更新操作之后操作DOM，我们可以在数据变化之后立即使用Vue.nextTick(callback)；这样回调函数会在DOM更新完成后被调用，就可以拿到最新的DOM元素了。  如果没有 nextTick 更新机制，那么 num 每次更新值都会触发视图更新(上面这段代码也就是会更新10万次视图)，有了nextTick机制，只需要更新一次，所以nextTick本质是一种优化策略  如果想要在修改数据后立刻得到更新后的DOM结构，可以使用Vue.nextTick() |

# Vue-组件通信

* 所有组件通信方式

[面试官：Vue组件间通信方式都有哪些? · Issue #12 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/12)

[vue组件间通信六种方式（完整版） - SegmentFault 思否](https://segmentfault.com/a/1190000019208626)

* 父子组件通信

|  |
| --- |
| 方法一、props/$emit  父组件A通过props的方式向子组件B传递，B to A 通过在 B 组件中 $emit, A 组件中 v-on 的方式实现。   1. 父组件向子组件传值   ****总结：父组件通过props向下传递数据给子组件。注：组件中的数据共有三种形式：data、props、computed**** |
| 接下来我们通过一个例子，说明父组件如何向子组件传递值：在子组件Users.vue中如何获取父组件App.vue中的数据 users:["Henry","Bucky","Emily"]  //App.vue父组件  <template>  <div id="app">  <users v-bind:users="users"></users>//前者自定义名称便于子组件调用，后者要传递数据名  </div>  </template>  <script>  import Users from "./components/Users"  export default {  name: 'App',  data(){  return{  users:["Henry","Bucky","Emily"]  }  },  components:{  "users":Users  }  }  //users子组件  <template>  <div class="hello">  <ul>  <li v-for="user in users">{{user}}</li>//遍历传递过来的值，然后呈现到页面  </ul>  </div>  </template>  <script>  export default {  name: 'HelloWorld',  props:{  users:{ //这个就是父组件中子标签自定义名字  type:Array,  required:true  }  }  }  </script> |
| 1. 子组件向父组件传值（通过事件形式）   接下来我们通过一个例子，说明子组件如何向父组件传递值：当我们点击“Vue.js Demo”后，子组件向父组件传递值，文字由原来的“传递的是一个值”变成“子向父组件传值”，实现子组件向父组件值的传递。 |
| // 子组件  <template>  <header>  <h1 @click="changeTitle">{{title}}</h1>//绑定一个点击事件  </header>  </template>  <script>  export default {  name: 'app-header',  data() {  return {  title:"Vue.js Demo"  }  },  methods:{  changeTitle() {  this.$emit("titleChanged","子向父组件传值");//自定义事件 传递值“子向父组件传值”  }  }  }  </script>  // 父组件  <template>  <div id="app">  <app-header v-on:titleChanged="updateTitle" ></app-header>//与子组件titleChanged自定义事件保持一致  // updateTitle($event)接受传递过来的文字  <h2>{{title}}</h2>  </div>  </template>  <script>  import Header from "./components/Header"  export default {  name: 'App',  data(){  return{  title:"传递的是一个值"  }  },  methods:{  updateTitle(e){ //声明这个函数  this.title = e;  }  },  components:{  "app-header":Header,  }  }  </script> |

* 事件总线Eventbus

[Vue3全局组件通信之EventBus - 简书 (jianshu.com)](https://www.jianshu.com/p/d8d55d8f0c48)

|  |
| --- |
|  |
| $emit/$on  ****这种方法通过一个空的Vue实例作为中央事件总线（事件中心），用它来触发事件和监听事件,巧妙而轻量地实现了任何组件间的通信，包括父子、兄弟、跨级****。当我们的项目比较大时，可以选择更好的状态管理解决方案vuex。  1.具体实现方式：  var Event=new Vue();  Event.$emit(事件名,数据);  Event.$on(事件名,data => {});  2.举个例子  假设兄弟组件有三个，分别是A、B、C组件，C组件如何获取A或者B组件的数据  <div id="itany">  <my-a></my-a>  <my-b></my-b>  <my-c></my-c>  </div>  <template id="a">  <div>  <h3>A组件：{{name}}</h3>  <button @click="send">将数据发送给C组件</button>  </div>  </template>  <template id="b">  <div>  <h3>B组件：{{age}}</h3>  <button @click="send">将数组发送给C组件</button>  </div>  </template>  <template id="c">  <div>  <h3>C组件：{{name}}，{{age}}</h3>  </div>  </template> |
| $emit/$on  <script>  var Event = new Vue();//定义一个空的Vue实例  var A = {  template: '#a',  data() {  return {  name: 'tom'  }  },  methods: {  send() {  Event.$emit('data-a', this.name);  }  }  }  var B = {  template: '#b',  data() {  return {  age: 20  }  },  methods: {  send() {  Event.$emit('data-b', this.age);  }  }  }  var C = {  template: '#c',  data() {  return {  name: '',  age: ""  }  },  mounted() {//在模板编译完成后执行  Event.$on('data-a',name => {  this.name = name;//箭头函数内部不会产生新的this，这边如果不用=>,this指代Event  })  Event.$on('data-b',age => {  this.age = age;  })  }  }  var vm = new Vue({  el: '#itany',  components: {  'my-a': A,  'my-b': B,  'my-c': C  }  });  </script> |

* 所有组件通信方式

|  |
| --- |
| props、$emit/$on、vuex、$parent / $children、$attrs/$listeners和provide/inject |
| Vuex |

# Vue-router

* Vue-router

[Vue 爬坑之路（三）—— 使用 vue-router 跳转页面 - Wise.Wrong - 博客园 (cnblogs.com)](https://www.cnblogs.com/wisewrong/p/6277262.html)

|  |
| --- |
| Vue-router是基于spa的 |

* Vue-router 懒加载

[vue-router的路由懒加载 - 掘金 (juejin.cn)](https://juejin.cn/post/7000184451590258701)

|  |
| --- |
| routes: [  { name: 'userShoppingCart',  path: '/userShoppingCart/:chatid',  component: ()=> import('../pages/user/userShoppingCart.vue') },  ]  把组件按组分块  有时候我们想把某个路由下的所有组件都打包在同个异步块 (chunk) 中。只需要使用 [命名 chunk (opens new window)](https://link.juejin.cn?target=https://webpack.js.org/guides/code-splitting-require/%23chunkname" \o "https://webpack.js.org/guides/code-splitting-require/#chunkname" \t "_blank)，一个特殊的注释语法来提供 chunk name (需要 Webpack > 2.4)。  const Foo = () => import(/\* webpackChunkName: "group-foo" \*/ './Foo.vue')  const Bar = () => import(/\* webpackChunkName: "group-foo" \*/ './Bar.vue')  const Baz = () => import(/\* webpackChunkName: "group-foo" \*/ './Baz.vue')  Webpack 会将任何一个异步模块与相同的块名称组合到相同的异步块中。 |

* vue-router 导航拦截

[一篇搞定Vue Router 路由守卫（导航守卫） - 掘金 (juejin.cn)](https://juejin.cn/post/6998720826962886693)

|  |
| --- |
| 全局路由钩子：  beforeEach(to,from, next)、beforeResolve(to,from, next)、afterEach(to,from)；  独享路由钩子：  beforeEnter(to,from, next)；  组件内路由钩子：  beforeRouteEnter(to,from, next)、beforeRouteUpdate(to,from, next)、beforeRouteLeave(to,from, next)  导航守卫回调参数  to：目标路由对象；  from：即将要离开的路由对象；  next：他是最重要的一个参数，他相当于佛珠的线，把一个一个珠子逐个串起来。 |

* vue-router 导航解析流程
* vue-router 路由模式

[vue-router路由模式详解 - 古兰精 - 博客园 (cnblogs.com)](https://www.cnblogs.com/goloving/p/9147551.html)

[详解vue-router中路由的二种模式 - 掘金 (juejin.cn)](https://juejin.cn/post/7063294535094239263)

|  |
| --- |
| **hash方式**  优点:由于发送https请求时不会带上hash值,因此无论是刷新页面或者改变hash值都不会影响到服务器端。  缺点:   1. 当使用hash值做路由时，锚点功能就失效了 2. hash方式传递参数的限制是基于浏览器url最大参数限制的。 3. hash方式导致浏览器地址栏出现#不美观   **history方式**  优点:   1. 很方便的获取到传递的参数。并且能传递最大640kb的对象 2. 后端也能很方便的获取到路由的地址   缺点:   1. F5刷新页面可能会导致页面404,前端url必须与发送到服务器请求的url相同 2. 改变url地址后,会重新请求服务器。 |

# Vuex

* vuex 原理

[状态管理Vuex | uni-app官网 (dcloud.io)](https://uniapp.dcloud.io/tutorial/vue-vuex.html" \l "%E6%A0%B8%E5%BF%83%E6%A6%82%E5%BF%B5)

[Vuex 是什么？ | Vuex (vuejs.org)](https://vuex.vuejs.org/zh/)

|  |
| --- |
| Vuex 是一个专为 Vue.js 应用程序开发的****状态管理模式 + 库****。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。  但是，当我们的应用遇到****多个组件共享状态****时，单向数据流的简洁性很容易被破坏：   * 多个视图依赖于同一状态。 * 来自不同视图的行为需要变更同一状态。   对于问题一，传参的方法对于多层嵌套的组件将会非常繁琐，并且对于兄弟组件间的状态传递无能为力。对于问题二，我们经常会采用父子组件直接引用或者通过事件来变更和同步状态的多份拷贝。以上的这些模式非常脆弱，通常会导致无法维护的代码。  因此，我们为什么不把组件的共享状态抽取出来，以一个全局单例模式管理呢？在这种模式下，我们的组件树构成了一个巨大的“视图”，不管在树的哪个位置，任何组件都能获取状态或者触发行为！  通过定义和隔离状态管理中的各种概念并通过强制规则维持视图和状态间的独立性，我们的代码将会变得更结构化且易维护。 |

* store

[开始 | Vuex (vuejs.org)](https://vuex.vuejs.org/zh/guide/" \l "%E6%9C%80%E7%AE%80%E5%8D%95%E7%9A%84-store)

|  |
| --- |
| 每一个 Vuex 应用的核心就是 store（仓库）。“store”基本上就是一个容器，它包含着你的应用中大部分的****状态 (state)****。Vuex 和单纯的全局对象有以下两点不同：  Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。  你不能直接改变 store 中的状态。改变 store 中的状态的唯一途径就是显式地****提交 (commit) mutation****。这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用。 |

* vuex index.js

[最详细Vuex教程（超详细）-基本使用\_叙白wh的博客-CSDN博客\_vuex](https://blog.csdn.net/weixin_55537701/article/details/118031388)

|  |
| --- |
| state  mutations: 修改 state 的唯一途径  actions  getters  modules  // 创建一个新的 store 实例const store = createStore({  state () { //加（）里面就要return，不加直接state: { count:0 }  //state是函数还是变量的问题跟data一样，最好选择函数声明避免多实例数据污染  return {  count: 0  }  },  mutations: {  increment (state) {  state.count++  }  }})  组件中调用位置：  state、getters在computed  mutations、actions在method |

* state

[State | Vuex (vuejs.org)](https://vuex.vuejs.org/zh/guide/state.html" \l "%E5%9C%A8-vue-%E7%BB%84%E4%BB%B6%E4%B8%AD%E8%8E%B7%E5%BE%97-vuex-%E7%8A%B6%E6%80%81)

|  |
| --- |
| 由于 Vuex 的状态存储是响应式的，从 store 实例中读取状态最简单的方法就是在[计算属性](https://cn.vuejs.org/guide/computed.html" \t "https://vuex.vuejs.org/zh/guide/_blank)中返回某个状态：  每当 store.state.count 变化的时候, 都会重新求取计算属性，并且触发更新相关联的 DOM。  Vuex 通过 Vue 的插件系统将 store 实例从根组件中“注入”到所有的子组件里。且子组件能通过 this.$store 访问到。  const Counter = {  template: `<div>{{ count }}</div>`,  computed: {  count () {  return this.$store.state.count  }  }} |

* mapState

|  |
| --- |
| 当一个组件需要获取多个状态的时候，将这些状态都声明为计算属性会有些重复和冗余。为了解决这个问题，我们可以使用 mapState 辅助函数帮助我们生成计算属性  // 在单独构建的版本中辅助函数为 Vuex.mapStateimport { mapState } from 'vuex'  export default {  // ...  computed: mapState({  // 箭头函数可使代码更简练  count: state => state.count,  // 传字符串参数 'count' 等同于 `state => state.count`  countAlias: 'count',  // 为了能够使用 `this` 获取局部状态，必须使用常规函数  countPlusLocalState (state) {  return state.count + this.localCount  }  })}  当映射的计算属性的名称与 state 的子节点名称相同时，我们也可以给 mapState 传一个字符串数组。  computed: mapState([  // 映射 this.count 为 store.state.count  'count'  ]) |
|  |

* getters

[Getter | Vuex (vuejs.org)](https://vuex.vuejs.org/zh/guide/getters.html)

|  |
| --- |
| Vuex 允许我们在 store 中定义“getter”（可以认为是 store 的计算属性）。  state中的数据一般需要处理以后再使用，getter中放的就是那些处理以后的state数据，这样其他组件用的时候直接用就可以，不用再次处理了。  Getter 接受 state 作为其第一个参数：  const store = createStore({  state: {  todos: [  { id: 1, text: '...', done: true },  { id: 2, text: '...', done: false }  ]  },  getters: {  doneTodos (state) {  return state.todos.filter(todo => todo.done) //后面是数组的操作函数  }  }}) |

* getters 属性访问

|  |
| --- |
| Getter 会暴露为 store.getters 对象，你可以以属性的形式访问这些值：  store.getters.doneTodos // -> [{ id: 1, text: '...', done: true }]  Getter 也可以接受其他 getter 作为第二个参数：  getters: {  // ...  doneTodosCount (state, getters) {  return getters.doneTodos.length  }}  store.getters.doneTodosCount // -> 1  我们可以很容易地在任何组件中使用它：（在computed中可以监听后续的变化）  computed: {  doneTodosCount () {  return this.$store.getters.doneTodosCount  }}  注意，getter 在通过属性访问时是作为 Vue 的响应式系统的一部分缓存其中的。 |

* getters 方法访问

|  |
| --- |
| 你也可以通过让 getter 返回一个函数，来实现给 getter 传参。在你对 store 里的数组进行查询时非常有用。  getters: {  // ...  getTodoById: (state) => (id) => {  return state.todos.find(todo => todo.id === id)  }}  store.getters.getTodoById(2) // -> { id: 2, text: '...', done: false }  注意，getter 在通过方法访问时，每次都会去进行调用，而不会缓存结果。 |

* mapGetters

|  |
| --- |
| mapGetters 辅助函数仅仅是将 store 中的 getter 映射到局部计算属性：  import { mapGetters } from 'vuex'  export default {  // ...  computed: {  // 使用对象展开运算符将 getter 混入 computed 对象中  // 在组件template中要有'doneTodosCount'(即自己的this.doneTodosCount属性)  ...mapGetters([  'doneTodosCount',  'anotherGetter',  // ...  ])  }}  如果你想将一个 getter 属性另取一个名字，使用对象形式：  ...mapGetters({  // 把 `this.doneCount` 映射为 `this.$store.getters.doneTodosCount`  doneCount: 'doneTodosCount'}) |

* mutation

[Mutation | Vuex (vuejs.org)](https://vuex.vuejs.org/zh/guide/mutations.html)

|  |
| --- |
| 更改 Vuex 的 store 中的状态的唯一方法是提交 mutation。  一条重要的原则就是要记住 ****mutation 必须是同步函数****  Vuex 中的 mutation 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：  const store = createStore({  state: {  count: 1  },  mutations: {  increment (state) {  // 变更状态  state.count++  }  }})  你不能直接调用一个 mutation 处理函数。这个选项更像是事件注册：“当触发一个类型为 increment 的 mutation 时，调用此函数。”要唤醒一个 mutation 处理函数，你需要以相应的 type 调用 ****store.commit**** 方法：  store.commit('increment') |

* mutation 载荷Payload(其实就是额外参数)

|  |
| --- |
| 你可以向 store.commit 传入额外的参数，即 mutation 的****载荷（payload）****：  // ...  mutations: {  increment (state, n) {  state.count += n  }}  store.commit('increment', 10) |

* mutation 组件提交/mapMutations

|  |
| --- |
| 你可以在组件中使用 this.$store.commit('xxx') 提交 mutation，或者使用 mapMutations 辅助函数将组件中的 methods 映射为 store.commit 调用（需要在根节点注入 store）。  import { mapMutations } from 'vuex'  export default {  // ...  methods: {  ...mapMutations([  'increment', // 将 `this.increment()` 映射为 `this.$store.commit('increment')`  // `mapMutations` 也支持载荷：  'incrementBy' // 将 `this.incrementBy(amount)` 映射为 `this.$store.commit('incrementBy', amount)`  ]),  ...mapMutations({  add: 'increment' // 将 `this.add()` 映射为 `this.$store.commit('increment')`  })  }} |

* action

[Action | Vuex (vuejs.org)](https://vuex.vuejs.org/zh/guide/actions.html)

|  |
| --- |
| Action 类似于 mutation，不同在于：   * Action 提交的是 mutation，而不是直接变更状态。 * Action 可以包含任意异步操作。   const store = createStore({  state: {  count: 0  },  mutations: {  increment (state) {  state.count++  }  },  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。当我们在之后介绍到 [Modules](https://vuex.vuejs.org/zh/guide/modules.html) 时，你就知道 context 对象为什么不是 store 实例本身了。  实践中，我们会经常用到 ES2015 的[参数解构](https://github.com/lukehoban/es6features" \l "destructuring" \t "https://vuex.vuejs.org/zh/guide/_blank)来简化代码（特别是我们需要调用 commit 很多次的时候）：  actions: {  increment ({ commit }) {  commit('increment')  }} |

* 分发Action

|  |
| --- |
| Action 通过 store.dispatch 方法触发：我们可以在 action 内部执行****异步****操作：  actions: {  incrementAsync ({ commit }) {  setTimeout(() => {  commit('increment')  }, 1000)  }} |
| Actions 支持同样的载荷方式和对象方式进行分发：  // 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}) |

* 组件中分发Action/mapActions

|  |
| --- |
| import { mapActions } from 'vuex'  export default {  // ...  methods: {  ...mapActions([  'increment', // 将 `this.increment()` 映射为 `this.$store.dispatch('increment')`  // `mapActions` 也支持载荷：  'incrementBy' // 将 `this.incrementBy(amount)` 映射为 `this.$store.dispatch('incrementBy', amount)`  ]),  ...mapActions({  add: 'increment' // 将 `this.add()` 映射为 `this.$store.dispatch('increment')`  })  }} |

* 组合Action

|  |
| --- |
| 组合Actions其实就是两个方面的异步操作：  ①在index.js中的actions中，action方法返回promise，在组件中dispatch（）.then（）继续完成任务。  ②action中，将方法action写为异步async action1，然后在action2里面可以调用await dispatch（action1），然后在写自己的commit。 |
| Action 通常是异步的，那么如何知道 action 什么时候结束呢？更重要的是，我们如何才能组合多个 action，以处理更加复杂的异步流程？  首先，你需要明白 store.dispatch 可以处理被触发的 action 的处理函数返回的 Promise，并且 store.dispatch 仍旧返回 Promise：  actions: {  actionA ({ commit }) {  return new Promise((resolve, reject) => {  setTimeout(() => {  commit('someMutation')  resolve()  }, 1000)  })  }}  现在你可以：  store.dispatch('actionA').then(() => {  // ...})  在另外一个 action 中也可以：  actions: {  // ...  actionB ({ dispatch, commit }) {  return dispatch('actionA').then(() => {  commit('someOtherMutation')  })  }}  最后，如果我们利用 [async / await](https://tc39.github.io/ecmascript-asyncawait/" \t "https://vuex.vuejs.org/zh/guide/_blank)，我们可以如下组合 action：  // 假设 getData() 和 getOtherData() 返回的是 Promise  actions: {  async actionA ({ commit }) {  commit('gotData', await getData())  },  async actionB ({ dispatch, commit }) {  await dispatch('actionA') // 等待 actionA 完成  commit('gotOtherData', await getOtherData())  }}  一个 store.dispatch 在不同模块中可以触发多个 action 函数。在这种情况下，只有当所有触发函数完成后，返回的 Promise 才会执行。 |

* module

[Module | Vuex (vuejs.org)](https://vuex.vuejs.org/zh/guide/modules.html)

|  |
| --- |
| Vuex 允许我们将 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：  const moduleA = {  state: () => ({ ... }),  mutations: { ... },  actions: { ... },  getters: { ... }}  const moduleB = {  state: () => ({ ... }),  mutations: { ... },  actions: { ... }}  const store = createStore({  modules: {  a: moduleA,  b: moduleB  }})  store.state.a // -> moduleA 的状态  store.state.b // -> moduleB 的状态 |

* module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  对于模块内部的 getter，根节点状态会作为第三个参数暴露出来： |

* module命名空间

|  |
| --- |
| 默认情况下，模块内部的 action 和 mutation 仍然是注册在****全局命名空间****的——这样使得多个模块能够对同一个 action 或 mutation 作出响应。  如果希望你的模块具有更高的封装度和复用性，你可以通过添加 namespaced: true 的方式使其成为带命名空间的模块。当模块被注册后，它的所有 getter、action 及 mutation 都会自动根据模块注册的路径调整命名。例如： |
| const store = createStore({  modules: {  account: {  namespaced: true,  // 模块内容（module assets）  state: () => ({ ... }), // 模块内的状态已经是嵌套的了，使用 `namespaced` 属性不会对其产生影响  getters: {  isAdmin () { ... } // -> getters['account/isAdmin']  },  actions: {  login () { ... } // -> dispatch('account/login')  },  mutations: {  login () { ... } // -> commit('account/login')  },  // 嵌套模块  modules: {  // 继承父模块的命名空间  myPage: {  state: () => ({ ... }),  getters: {  profile () { ... } // -> getters['account/profile']  }  },  // 进一步嵌套命名空间  posts: {  namespaced: true,  state: () => ({ ... }),  getters: {  popular () { ... } // -> getters['account/posts/popular']  }  }  }  }  }}) |

* module组件调用

|  |
| --- |
| 可以将模块的空间名称字符串作为第一个参数传递给上述函数，这样所有绑定都会自动将该模块作为上下文。于是上面的例子可以简化为：  computed: {  ...mapState('some/nested/module', {  a: state => state.a,  b: state => state.b  }),  ...mapGetters('some/nested/module', [  'someGetter', // -> this.someGetter  'someOtherGetter', // -> this.someOtherGetter  ])},  methods: {  ...mapActions('some/nested/module', [  'foo', // -> this.foo()  'bar' // -> this.bar()  ])} |