**Vue基础**

**一、Vue的实质是“数据驱动视图”。**

**二、模板语法**

Vue.js使用了基于HTML的模板语法，允许开发者声明式地将DOM绑定至底层 Vue实例的数据。所有Vue.js的模板都是合法的HTML，所以能被遵循规范的浏览器和HTML解析器解析。

在底层的实现上，Vue 将模板编译成虚拟 DOM 渲染函数。结合响应系统，Vue 能够智能地计算出最少需要重新渲染多少组件，并把 DOM 操作次数减到最少。

**文本**

数据绑定最常见的形式就是使用“Mustache”语法 (双大括号) 的文本插值：

<span>Message: {{ msg }}</span>

Mustache标签将会被替代为对应数据对象上msg属性的值。无论何时，绑定的数据对象上msg属性发生了改变，插值处的内容都会更新。

通过使用v-once，你也能执行一次性地插值，当数据改变时，插值处的内容不会更新。但请留心这会影响到该节点上的其它数据绑定：

<span v-once>这个将不会改变: {{ msg }}</span>

**原始 HTML**

双大括号会将数据解释为普通文本，而非 HTML 代码。为了输出真正的 HTML，你需要使用v-html指令：

<p>Using mustaches: {{ rawHtml }}</p>

<p>Using v-html directive: <span v-html="rawHtml"></span></p>

Using mustaches: <span>This should be red.</span>

Using v-html directive: This should be red.

这个span的内容将会被替换成为属性值rawHtml，直接作为HTML——会忽略解析属性值中的数据绑定。注意，不能使用v-html来复合局部模板，因为 Vue不是基于字符串的模板引擎。反之，对于用户界面 (UI)，组件更适合作为可重用和可组合的基本单位。

你的站点上动态渲染的任意 HTML 可能会非常危险，因为它很容易导致XSS 攻击。请只对可信内容使用 HTML 插值，绝不要对用户提供的内容使用插值。

**特性**

Mustache语法不能作用在HTML特性上，遇到这种情况应该使用v-bind 指令：

<div v-bind:id="dynamicId"></div>

对于布尔特性 (它们只要存在就意味着值为 true)，v-bind 工作起来略有不同，在这个例子中：

<button v-bind:disabled="isButtonDisabled">Button</button>

如果isButtonDisabled的值是null、undefined或false，则disabled特性甚至不会被包含在渲染出来的<button>元素中。

**使用 JavaScript 表达式**

迄今为止，在我们的模板中，我们一直都只绑定简单的属性键值。但实际上，对于所有的数据绑定，Vue.js都提供了完全的JavaScript 表达式支持。

{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

{{ message.split('').reverse().join('') }}

<div v-bind:id="'list-' + id"></div>

这些表达式会在所属 Vue实例的数据作用域下作为JavaScript被解析。有个限制就是，每个绑定都只能包含单个表达式，所以下面的例子都不会生效。

<!-- 这是语句，不是表达式 -->

{{ var a = 1 }}

<!-- 流控制也不会生效，请使用三元表达式 -->

{{ if (ok) { return message } }}

模板表达式都被放在沙盒中，只能访问全局变量的一个白名单，如Math和Date。你不应该在模板表达式中试图访问用户定义的全局变量。

**指令**

指令(Directives)是带有v-前缀的特殊特性。指令特性的值预期是单个 JavaScript表达式(v-for是例外情况)。指令的职责是，当表达式的值改变时，将其产生的连带影响，响应式地作用于DOM。回顾我们在介绍中看到的例子：

<p v-if="seen">现在你看到我了</p>

这里，v-if指令将根据表达式seen的值的真假来插入 /移除<p>元素。

**参数**

一些指令能够接收一个“参数”，在指令名称之后以冒号表示。例如，v-bind指令可以用于响应式地更新HTML特性：

<a v-bind:href="url">...</a>

在这里href是参数，告知v-bind指令将该元素的href特性与表达式url的值绑定。

另一个例子是v-on指令，它用于监听 DOM 事件：

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

在这里参数是监听的事件名。我们也会更详细地讨论事件处理。

**动态参数**

2.6.0 新增

从 2.6.0 开始，可以用方括号括起来的JavaScript表达式作为一个指令的参数：

<a v-bind:[attributeName]="url"> ... </a>

这里的attributeName会被作为一个JavaScript表达式进行动态求值，求得的值将会作为最终的参数来使用。例如，如果你的Vue实例有一个data属性attributeName，其值为"href"，那么这个绑定将等价于v-bind:href。

同样地，你可以使用动态参数为一个动态的事件名绑定处理函数：

<a v-on:[eventName]="doSomething"> ... </a>

同样地，当eventName的值为"focus"时，v-on:[eventName]将等价于v-on:focus。

**对动态参数的值的约束**

动态参数预期会求出一个字符串，异常情况下值为null。这个特殊的null值可以被显性地用于移除绑定。任何其它非字符串类型的值都将会触发一个警告。

**对动态参数表达式的约束**

动态参数表达式有一些语法约束，因为某些字符，例如空格和引号，放在HTML 特性名里是无效的。同样，在DOM中使用模板时你需要回避大写键名。

例如，下面的代码是无效的：

<!-- 这会触发一个编译警告 -->

<a v-bind:['foo' + bar]="value"> ... </a>

变通的办法是使用没有空格或引号的表达式，或用计算属性替代这种复杂表达式。

另外，如果你DOM 中使用模板 (直接在一个HTML文件里撰写模板)，需要留意浏览器会把特性名全部强制转为小写：

<!-- 在 DOM 中使用模板时这段代码会被转换为 `v-bind:[someattr]` -->

<a v-bind:[someAttr]="value"> ... </a>

**修饰符**

修饰符(modifier)是以半角句号.指明的特殊后缀，用于指出一个指令应该以特殊方式绑定。例如.prevent修饰符告诉v-on指令对于触发的事件调用event.preventDefault()：

<form v-on:submit.prevent="onSubmit">...</form>

在接下来对[v-on](https://cn.vuejs.org/v2/guide/events.html" \l "%E4%BA%8B%E4%BB%B6%E4%BF%AE%E9%A5%B0%E7%AC%A6)和[v-for](https://cn.vuejs.org/v2/guide/forms.html" \l "%E4%BF%AE%E9%A5%B0%E7%AC%A6)等功能的探索中，你会看到修饰符的其它例子。

**缩写**

v-前缀作为一种视觉提示，用来识别模板中Vue特定的特性。当你在使用Vue.js为现有标签添加动态行为 (dynamic behavior) 时，v-前缀很有帮助，然而，对于一些频繁用到的指令来说，就会感到使用繁琐。同时，在构建由Vue管理所有模板的单页面应用程序(SPA-single page application)时，v-前缀也变得没那么重要了。因此，Vue为v-bind和v-on这两个最常用的指令，提供了特定简写：

v-bind**缩写**

<!-- 完整语法 -->

<a v-bind:href="url">...</a>

<!-- 缩写 -->

<a :href="url">...</a>

…

v-on**缩写**

<!-- 完整语法 -->

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

<!-- 缩写 -->

<a @click="doSomething">...</a>

…

它们看起来可能与普通的 HTML 略有不同，但:与@对于特性名来说都是合法字符，在所有支持 Vue 的浏览器都能被正确地解析。而且，它们不会出现在最终渲染的标记中。缩写语法是完全可选的，但随着你更深入地了解它们的作用，你会庆幸拥有它们。

**三、组建中data定义**

Data必须是一个函数。

当我们定义这个 <button-counter> 组件时，你可能会发现它的 data 并不是像这样直接提供一个对象：

data: {

count: 0

}

取而代之的是，**一个组件的**data**选项必须是一个函数**，因此每个实例可以维护一份被返回对象的独立的拷贝：

data: function () {

return {

count: 0

}

}

如果 Vue 没有这条规则，点击一个按钮就可能会影响到其它所有实例。

**四、Computed计算属性**

Computed计算属性，用于简单的运算，会保留缓存，当相关依赖没有发生改变时多次访问computed中的函数时，立即返回之前的计算结果，不会再次调用函数获得数据。

var vm = new Vue({

el: '#demo',

data: {

firstName: 'Foo',

lastName: 'Bar'

},

computed: {

fullName: function () {

return this.firstName + ' ' + this.lastName

}

}

})

**五、自定义事件**

[**事件名**](https://cn.vuejs.org/v2/guide/components-custom-events.html#%E4%BA%8B%E4%BB%B6%E5%90%8D)

不同于组件和 prop，事件名不存在任何自动化的大小写转换。而是触发的事件名需要完全匹配监听这个事件所用的名称。举个例子，如果触发一个 camelCase名字的事件：

this.$emit('myEvent')

则监听这个名字的kebab-case版本是不会有任何效果的：

<!-- 没有效果 -->

<my-component v-on:my-event="doSomething"></my-component>

不同于组件和 prop，事件名不会被用作一个JavaScript变量名或属性名，所以就没有理由使用camelCase或PascalCase了。并且v-on事件监听器在DOM 模板中会被自动转换为全小写(因为HTML是大小写不敏感的)，所以v-on:myEvent将会变成v-on:myevent——导致myEvent不可能被监听到。

因此，我们推荐你始终使用kebab-case的事件名。

[**自定义组件的**v-model](https://cn.vuejs.org/v2/guide/components-custom-events.html#%E8%87%AA%E5%AE%9A%E4%B9%89%E7%BB%84%E4%BB%B6%E7%9A%84-v-model)

一个组件上的v-model默认会利用名为value的prop和名为input的事件，但是像单选框、复选框等类型的输入控件可能会将value特性用于[不同的目的](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/checkbox" \l "Value" \t "_blank)。model选项可以用来避免这样的冲突：

Vue.component('base-checkbox', {

model: {

prop: 'checked',

event: 'change'

},

props: {

checked: Boolean

},

template: `

<input

type="checkbox"

v-bind:checked="checked"

v-on:change="$emit('change', $event.target.checked)"

>

`

})

现在在这个组件上使用v-model的时候：

<base-checkbox v-model="lovingVue"></base-checkbox>

这里的lovingVue的值将会传入这个名为checked的prop。同时当<base-checkbox>触发一个change事件并附带一个新的值的时候，这个lovingVue的属性将会被更新。

注意你仍然需要在组件的props选项里声明checked这个prop。

[**将原生事件绑定到组件**](https://cn.vuejs.org/v2/guide/components-custom-events.html#%E5%B0%86%E5%8E%9F%E7%94%9F%E4%BA%8B%E4%BB%B6%E7%BB%91%E5%AE%9A%E5%88%B0%E7%BB%84%E4%BB%B6)

你可能有很多次想要在一个组件的根元素上直接监听一个原生事件。这时，你可以使用v-on的.native修饰符：

<base-input v-on:focus.native="onFocus"></base-input>

在有的时候这是很有用的，不过在你尝试监听一个类似<input>的非常特定的元素时，这并不是个好主意。比如上述<base-input>组件可能做了如下重构，所以根元素实际上是一个<label>元素：

<label>

{{ label }}

<input

v-bind="$attrs"

v-bind:value="value"

v-on:input="$emit('input', $event.target.value)"

>

</label>

这时，父级的.native监听器将静默失败。它不会产生任何报错，但是onFocus处理函数不会如你预期地被调用。

为了解决这个问题，Vue 提供了一个$listeners属性，它是一个对象，里面包含了作用在这个组件上的所有监听器。例如：

{

focus: function (event) { /\* ... \*/ }

input: function (value) { /\* ... \*/ },

}

有了这个$listeners属性，你就可以配合v-on="$listeners"将所有的事件监听器指向这个组件的某个特定的子元素。对于类似<input>的你希望它也可以配合v-model工作的组件来说，为这些监听器创建一个类似下述inputListeners的计算属性通常是非常有用的：

Vue.component('base-input', {

inheritAttrs: false,

props: ['label', 'value'],

computed: {

inputListeners: function () {

var vm = this

// `Object.assign` 将所有的对象合并为一个新对象

return Object.assign({},

// 我们从父级添加所有的监听器

this.$listeners,

// 然后我们添加自定义监听器，

// 或覆写一些监听器的行为

{

// 这里确保组件配合 `v-model` 的工作

input: function (event) {

vm.$emit('input', event.target.value)

}

}

)

}

},

template: `

<label>

{{ label }}

<input

v-bind="$attrs"

v-bind:value="value"

v-on="inputListeners"

>

</label>

`

})

现在<base-input>组件是一个完全透明的包裹器了，也就是说它可以完全像一个普通的<input>元素一样使用了：所有跟它相同的特性和监听器的都可以工作。

[.sync**修饰符**](https://cn.vuejs.org/v2/guide/components-custom-events.html#sync-%E4%BF%AE%E9%A5%B0%E7%AC%A6)

在有些情况下，我们可能需要对一个prop进行“双向绑定”。不幸的是，真正的双向绑定会带来维护上的问题，因为子组件可以修改父组件，且在父组件和子组件都没有明显的改动来源。

这也是为什么我们推荐以update:myPropName的模式触发事件取而代之。举个例子，在一个包含titleprop的假设的组件中，我们可以用以下方法表达对其赋新值的意图：

this.$emit('update:title', newTitle)

然后父组件可以监听那个事件并根据需要更新一个本地的数据属性。例如：

<text-document

v-bind:title="doc.title"

v-on:update:title="doc.title = $event"

></text-document>

为了方便起见，我们为这种模式提供一个缩写，即.sync修饰符：

<text-document v-bind:title.sync="doc.title"></text-document>

注意带有.sync修饰符的v-bind不能和表达式一起使用(例如v-bind:title.sync=”doc.title+‘!’”是无效的)。取而代之的是，你只能提供你想要绑定的属性名，类似v-model。

当我们用一个对象同时设置多prop 的时候，也可以将这.sync修饰符和v-bind配合使用：

<text-document v-bind.sync="doc"></text-document>

这样会把doc对象中的每一个属性(如title) 都作为一个独立的prop传进去，然后各自添加用于更新的v-on监听器。

将v-bind.sync用在一个字面量的对象上，例如v-bind.sync=”{title: doc.title}”，是无法正常工作的，因为在解析一个像这样的复杂表达式的时候，有很多边缘情况需要考虑。

**六、NPM打包原理**

npm是Node.js能够如此成功的主要原因之一。npm团队做了很多的工作，以确保npm保持向后兼容，并在不同的环境中保持一致。

npm是围绕着[语义版本控制（semver）](http://semver.org/)的思想而设计。

给定一个版本号：主版本号. 次版本号. 补丁版本号， 以下这三种情况需要增加相应的版本号：

* 主版本号：当API发生改变，并与之前的版本不兼容的时候；
* 次版本号：当增加了功能，但是向后兼容的时候；
* 补丁版本号：当做了向后兼容的缺陷修复的时候。

# npm使用一个名为package.json的文件，用户可以通过npm install--save命令把项目里所有的依赖项保存在这个文件里。

例如，运npminstall—save lodash会将以下几行添加到package.json文件中。

"dependencies": {

"lodash": "^4.17.4"

}

"dependencies": {

"lodash": "^4.17.4"

}

解析，

* ^字符，告诉npm，安装主版本等于4的任意一个版本即可。
* 现在运行npm进行安装，npm将安装lodash的主版本为4的最新版，**可能**是 lodash@4.25.5（@是npm约定用来确定包名的指定版本的）。
* 理论上，次版本号的变化并不会影响向后兼容性。因此，安装最新版的依赖库应该是能正常工作的，而且能引入自4.17.4版本以后的重要错误和安全方面的修复。
* 但是，即使不同的开发人员使用了相同的package.json文件，在他们自己的机器上也可能会安装同一个库的不同种版本，这样就会存在潜在的难以调试的错误和“在我的电脑上…”的情形。

大多数npm库都严重依赖于其他npm库，这会导致**嵌套依赖**关系，并增加无法匹配相应版本的几率。

虽然可以通过npm config set save-exact true命令关闭在版本号前面使用^的默认行为，但这个**只会影响顶级依赖关系**。由于每个依赖的库都有自己的package.json文件，而在它们自己的依赖关系前面可能会有^符号，所以无法通过package.json文件为嵌套依赖的内容提供保证。

为了解决这个问题，npm提供了[shrinkwrap](https://docs.npmjs.com/cli/shrinkwrap)命令。此命令将生成一个npm-shrinkwrap.json文件，为所有库和所有嵌套依赖的库记录确切的版本。

然而，即使存在npm-shrinkwrap.json这个文件，**npm也只会锁定库的版本，而不是库的内容**。即便npm现在也能阻止用户多次重复发布库的同一版本，但是npm管理员仍然具有强制更新某些库的权力。

这是引用自shrinkwrap文档的内容：

如果你希望锁定包中的特定字节，比如是为了保证能正确地重新部署或构建，那么你应该在源代码控制中检查依赖关系，或者采取一些其他的机制来校验内容，而不是靠校验版本。

npm 2会安装每一个包所依赖的所有依赖项。如果我们有这么一个项目，它依赖项目A，项目A依赖项目B，项目B依赖项目C，那么依赖树将如下所示：

node\_modules

- package-A

-- node\_modules

--- package-B

----- node\_modules

------ package-C

-------- some-really-really-really-long-file-name-in-package-c.js

node\_modules

- package-A

-- node\_modules

--- package-B

----- node\_modules

------ package-C

-------- some-really-really-really-long-file-name-in-package-c.js

这个结构可能会很长。这对于基于Unix的操作系统来说只不过是一个小烦恼，但对于Windows来说却是个破坏性的东西，因为有**很多程序无法处理超过260 个字符的文件路径名**。

npm 3采用了扁平依赖关系树来解决这个问题，所以我们的3个项目结构现在看起来如下所示：

node\_modules

- package-A

- package-B

- package-C

-- some-file-name-in-package-c.js

node\_modules

- package-A

- package-B

- package-C

-- some-file-name-in-package-c.js

这样，一个原来很长的文件路径名就从./node\_modules/package-A/node\_modules/package-B/node-modules/some-file-name-in-package-c.js变成了/node\_modules/some-file-name-in-package-c.js。

这种方法的**缺点**是，npm**必须首先遍历所有的项目依赖关系**，然后再决定如何生成扁平的node\_modules目录结构。npm必须**为所有使用到的模块构建一个完整的依赖关系树**，这是一个**耗时**的操作，是[**npm安装速度慢的一个很重要的原因**](https://github.com/npm/npm/issues/8826)。

想当然的以为每次运行npm install命令时，NPM 都得从互联网上下载所有内容。

但是，**npm是有本地缓存的**，它保存了已经下载的每个版本的压缩包。本地缓存的内容可以通过npm cache ls命令进行查看。**本地缓存的设计有助于减少安装时间**。

总而言之，npm是一个成熟、稳定、并且有趣的包管理器。

**七、Vue组件语法**

组件 (Component) 是Vue.js最强大的功能之一，使用组件化方式开发，可以封装可重用的代码，减少重复劳动。

接下来，我们来注册一个组件，语法：Vue.component(tagName, options)

注册组件如下：

Vue.component('component-a', {

template: '<div>component-a</div>'

})

component-a 是注册的组件标签，下面就可以使用这个组件了。

<div id="app">

<component-a></component-a>

</div>

// 创建根实例

new Vue({

el: '#app'

})

最后，渲染为：

<div id="app">

<div>component-a!</div>

</div>

通过Vue.component方式注册的组件是一个全局组件，我们还可以创建局部组件，通过某 Vue实例 / 组件的实例选项components注册仅在其作用域中可用的组件。

var Child = {

template: '<div>A custom component!</div>'

}

var vm = new Vue({

el: '#app',

components:{

'component-a':Child

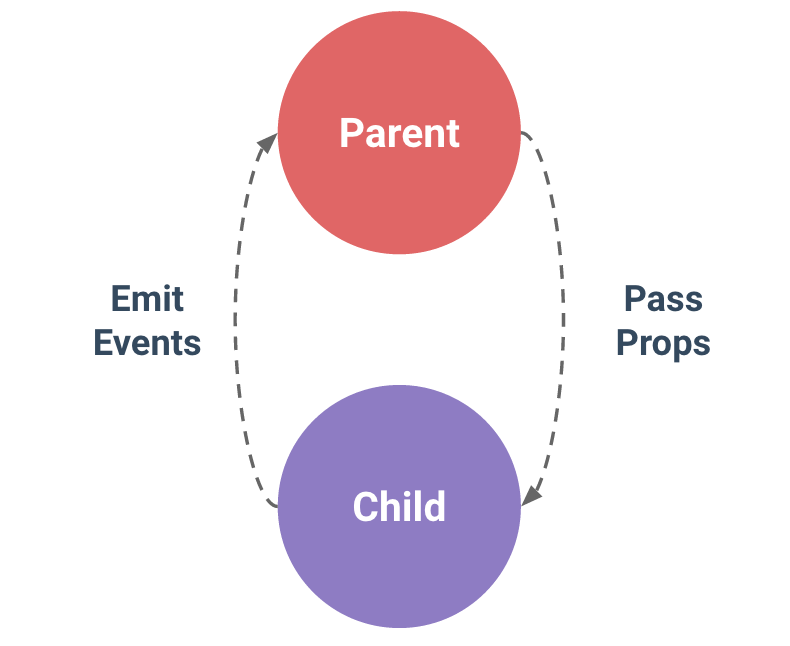
}

})

局部注册的组件将只在父组件模板中可用。

**组件通信**

组件A在它的模板中使用了组件B，它们之间必然需要相互通信，父组件可能要给子组件下发数据，子组件则可能要将它内部发生的事情告知父组件。  
我们先来看一张图：



从图上可以很清晰的看到，父子组件通信主要是使用prop和自定义事件，父组件通过prop给子组件下发数据，子组件通过事件给父组件发送消息。  
父子组件的关系可以总结为prop向下传递，事件向上传递。

来看个示例：

<div id="app">

<div>

接收到的子组件回复消息：{{replyMsg}}

</div>

<component-a v-bind:msg="msg" @reply="replay"></component-a>

</div>

<template id="componentA">

<div>

<div>我是组件a，接收到父类的消息是：{{msgfilter}}</div>

<div @click="reply">回复</div>

</div>

</template>

Vue.component('component-a', {

template: '#componentA',

props:['msg'],

data:function(){

return{

msgfilter:'处理后的消息-》'+this.msg

}

},

methods:{

reply:function(){

this.$emit('reply','子组件消息')

}

}

})

var vm = new Vue({

el: '#app',

data:{

msg:'父组件消息',

replyMsg:''

},

methods:{

replay:function(msg){

this.replyMsg = msg

}

}

})

需要注意的是，组件里面的data必须是一个函数，通过return形式返回

1. 父组件下发数据到子组件

在组件中通v-bind 添加一个参数将数据传递到子组件里，然后使用props接收传递过来的参数 props:['msg']  
<component-a v-bind:msg="msg"></component-a>  
渲染结果：



2. 子组件通过事件给父组件发送消息  
子组件通过$emit触发事件

this.$emit('reply','子组件消息')

在组件标签中通过v-on进行监听

<component-a v-bind:msg="msg" @reply="replay"></component-a>

监听到事件后触发reply函数，接收到子组件发送的消息

replay:function(msg){

this.replyMsg = msg

}

渲染结果：

<div id="app">

<div>

接收到的子组件回复消息：子组件消息

</div>

<div>

<div>我是组件a，接收到父类的消息是：处理后的消息-》父组件消息</div>

<div>回复</div>

</div>

</div>

以上可以看到我们是通过字符串数组来定义prop的，除此之外我们还可以用对象的形式来定义prop，用来为组件的prop 指定验证规则，  
type的值可以是这些：String Number Boolean Function Object Array Symbol, 例如：

props: {

name: String,

showDialog: {

type: Boolean,

default: false

}

}

**八、V-for指令**

v-for 指令基于一个数组渲染一个列表，它和 JavaScript 的遍历语法相似  
v-for="item in list"  
list 是一个数组，item 是当前遍历的数组元素  
v-for="(item,index) in list" 其中 index 是当前循环的索引，下标从 0 开始



<div id="app">

<table>

<tr class="thead">

<td>序号</td>

<td>姓名</td>

<td>年龄</td>

</tr>

<tr v-for="(item,index) in list">

<td v-text="index+1"></td>

<td v-text="item.name"></td>

<td v-text="item.age"></td>

</tr>

</table>

</div>

var vm = new Vue({

el: '#app',

data: {

list:[{

name:'章三',

age:18

},{

name:'李四',

age:23

}]

}

})

**Vuex**

**一、组件间通信**

使用vuex作用就是实现组件间数据共享。其原理为：

1. vue团队为了简化组件间的通信，将state抽象成一个单例模式，将其放到全局，让各个组件都能共享使用

2. vuex数据传递是单向的：action ---> mutation ---> state ---> component ---> action

* vue component指的就是我门定义的组件
* action 交互中产生的动作
* mutations 动作产生的修改数据的行为
* state 共享数据

3. vuex设计的时候相对修改的行为做单测（测试），开发了devtools来做测试，只能检测同步的操作

4. 规范定义：只能在mutations中做同步操作，所以增加了action来异步处理数据

5. 将mutations中的异步操作转移到actions中了，这样就可以测试同步的操作了

**二、State**

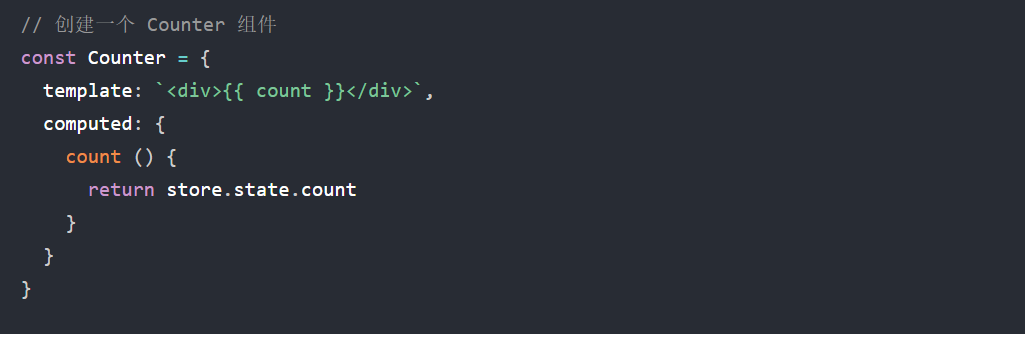
[**#**](https://vuex.vuejs.org/zh/guide/state.html#%E5%8D%95%E4%B8%80%E7%8A%B6%E6%80%81%E6%A0%91)**单一状态树**

Vuex使用单一状态树——是的，用一个对象就包含了全部的应用层级状态。至此它便作为一个“唯一数据源(SSOT)”而存在。这也意味着，每个应用将仅仅包含一个store实例。单一状态树让我们能够直接地定位任一特定的状态片段，在调试的过程中也能轻易地取得整个当前应用状态的快照。

单状态树和模块化并不冲突——在后面的章节里我们会讨论如何将状态和状态变更事件分布到各个子模块中。

[**#**](https://vuex.vuejs.org/zh/guide/state.html#%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)**在Vue组件中获得 Vuex状态**

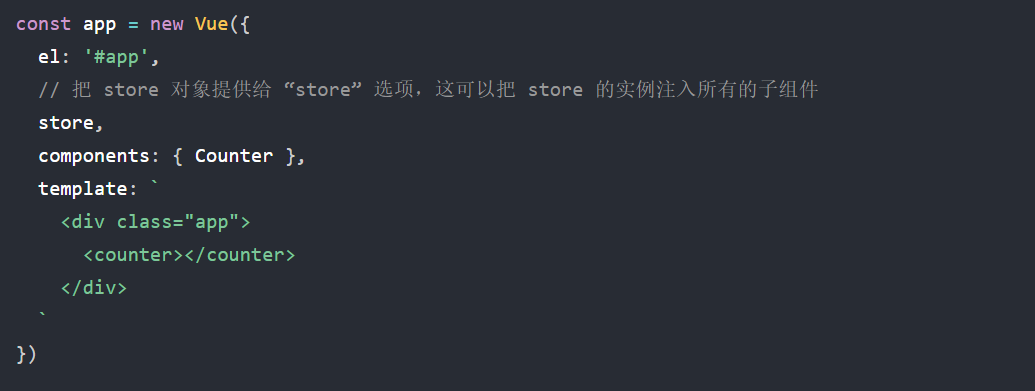
由于Vuex的状态存储是响应式的，从store实例中读取状态最简单的方法就是在计算属性中返回某个状态：



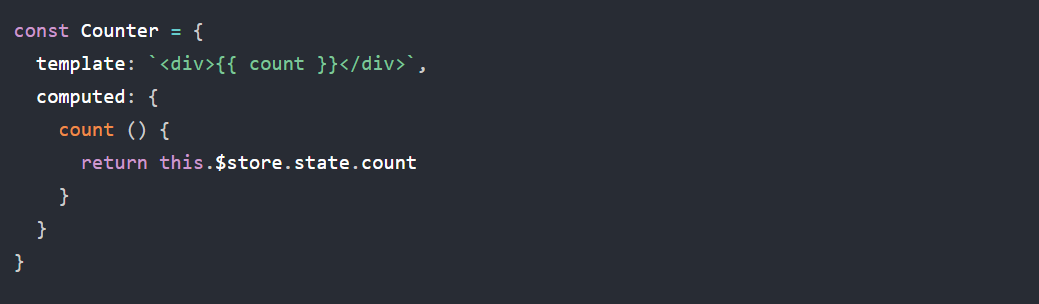
每当store.state.count变化的时候, 都会重新求取计算属性，并且触发更新相关联的DOM。

然而，这种模式导致组件依赖全局状态单例。在模块化的构建系统中，在每个需要使用state的组件中需要频繁地导入，并且在测试组件时需要模拟状态。

Vuex通过store选项，提供了一种机制将状态从根组件“注入”到每一个子组件中（需调用Vue.use(Vuex)）：



通过在根实例中注册store选项，该store实例会注入到根组件下的所有子组件中，且子组件能通过this.$store访问到。让我们更新下Counter的实现：

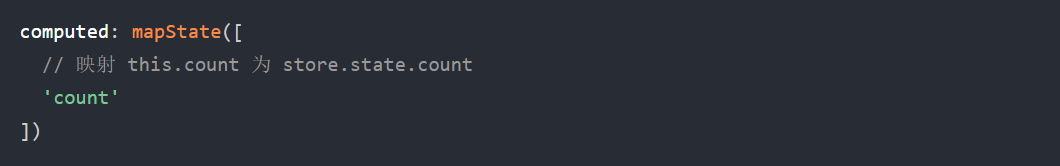


[**#**](https://vuex.vuejs.org/zh/guide/state.html#mapstate-%E8%BE%85%E5%8A%A9%E5%87%BD%E6%95%B0)mapState**辅助函数**

当一个组件需要获取多个状态时候，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，让你少按几次键：

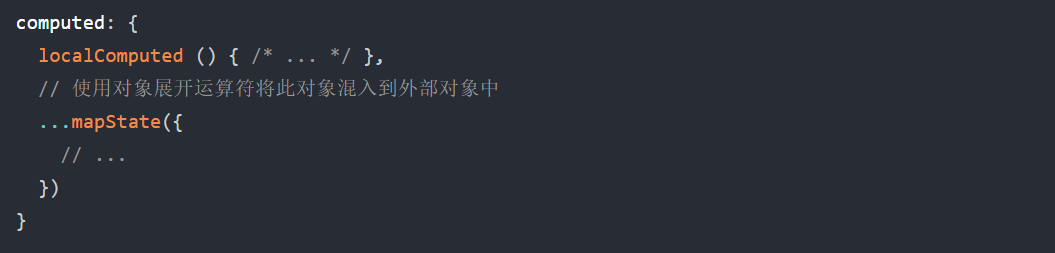


当映射的计算属性的名称与state的子节点名称相同时，我们也可以给mapState传一个字符串数组。



[**#**](https://vuex.vuejs.org/zh/guide/state.html#%E5%AF%B9%E8%B1%A1%E5%B1%95%E5%BC%80%E8%BF%90%E7%AE%97%E7%AC%A6)**对象展开运算符**

mapState函数返回的是一个对象。我们如何将它与局部计算属性混合使用呢？通常，我们需要使用一个工具函数将多个对象合并为一个，以使我们可以将最终对象传给computed属性。但是自从有了对象展开运算符（现处于 ECMAScript提案stage-4阶段），我们可以极大地简化写法：

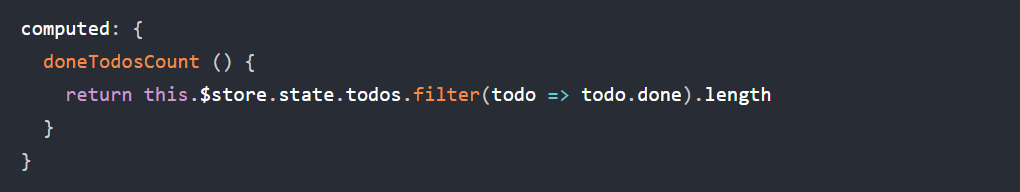


[**#**](https://vuex.vuejs.org/zh/guide/state.html#%E7%BB%84%E4%BB%B6%E4%BB%8D%E7%84%B6%E4%BF%9D%E6%9C%89%E5%B1%80%E9%83%A8%E7%8A%B6%E6%80%81)**组件仍然保有局部状态**

使用Vuex并不意味着你需要将所有的状态放入Vuex。虽然将所有的状态放到 Vuex会使状态变化更显式和易调试，但也会使代码变得冗长和不直观。如果有些状态严格属于单个组件，最好还是作为组件的局部状态。你应该根据你的应用开发需要进行权衡和确定。

**三、Getter**

有时候我们需要从store中的state中派生出一些状态，例如对列表进行过滤并计数：



如果有多个组件需要用到此属性，我们要么复制这个函数，或者抽取到一个共享函数然后在多处导入它——无论哪种方式都不是很理想。

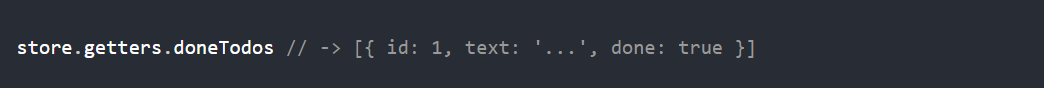
Vuex允许我们在store中定义“getter”（可以认为是store的计算属性）。就像计算属性一样,getter的返回值会根据它的依赖被缓存起来，且只有当它的依赖值发生了改变才会被重新计算。

Getters接受state作为其第一个参数：

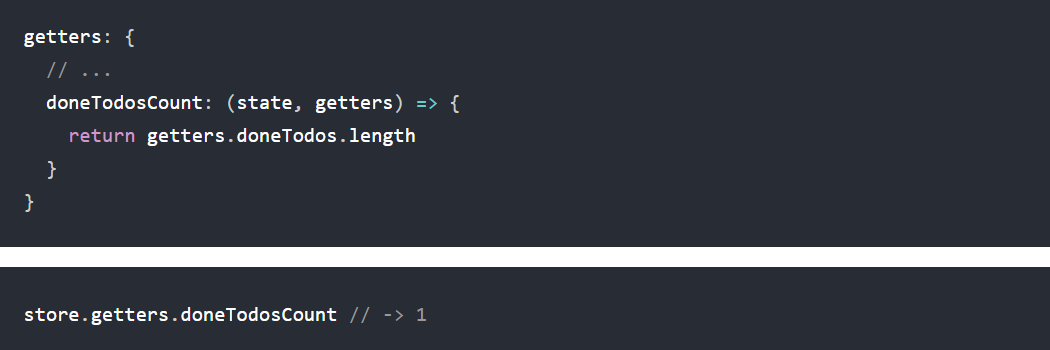


[**#**](https://vuex.vuejs.org/zh/guide/getters.html#%E9%80%9A%E8%BF%87%E5%B1%9E%E6%80%A7%E8%AE%BF%E9%97%AE)**通过属性访问**

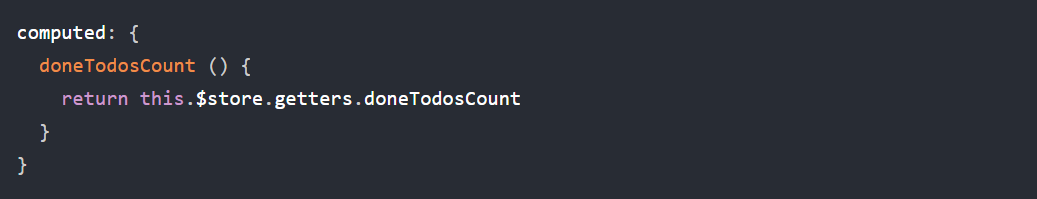
Getter会暴露为store.getters对象，你可以以属性的形式访问这些值：



Getter 也可以接受其他 getter 作为第二个参数：



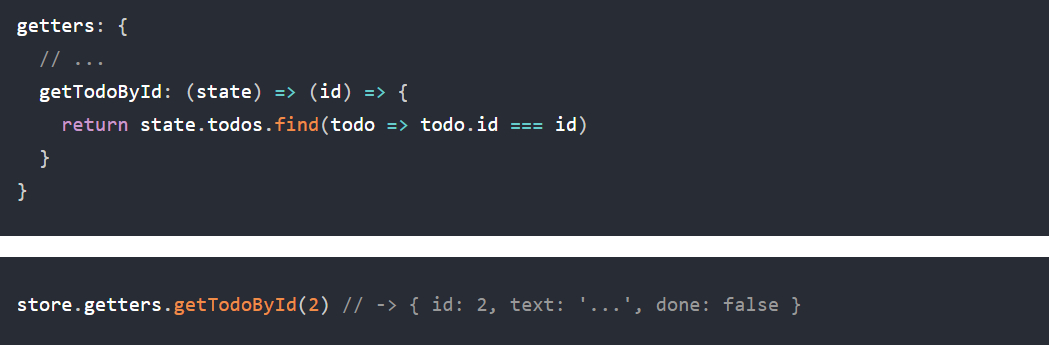
我们可以很容易地在任何组件中使用它：



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。

[**#**](https://vuex.vuejs.org/zh/guide/getters.html#%E9%80%9A%E8%BF%87%E6%96%B9%E6%B3%95%E8%AE%BF%E9%97%AE)**通过方法访问**

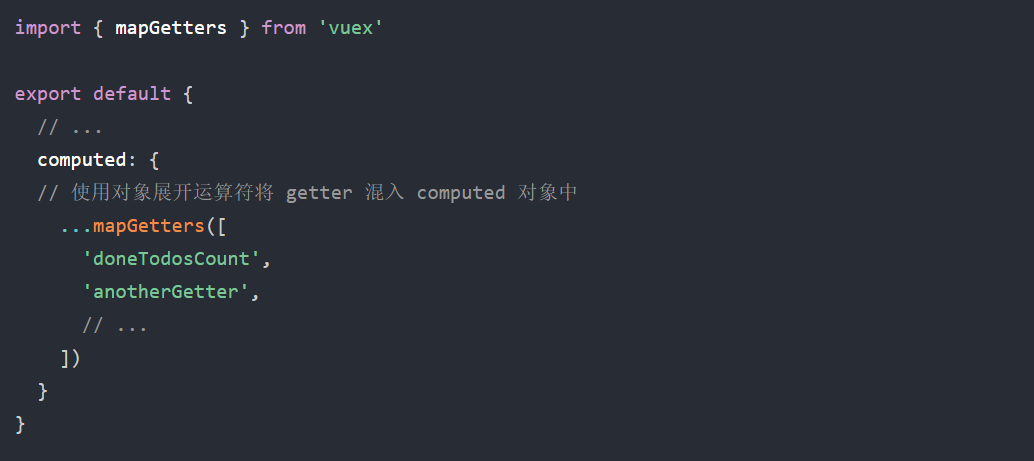
你也可以通过让getter返回一个函数，来实现给gette传参。在你对store 里的数组进行查询时非常有用。



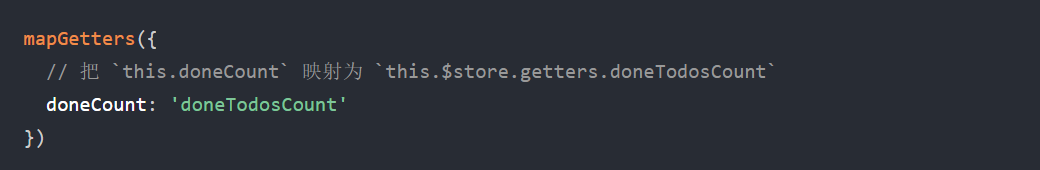
注意，getter在通过方法访问时，每次都会去进行调用，而不会缓存结果。

[**#**](https://vuex.vuejs.org/zh/guide/getters.html#mapgetters-%E8%BE%85%E5%8A%A9%E5%87%BD%E6%95%B0)mapGetters**辅助函数**

mapGetters辅助函数仅仅是将store中的getter映射到局部计算属性：

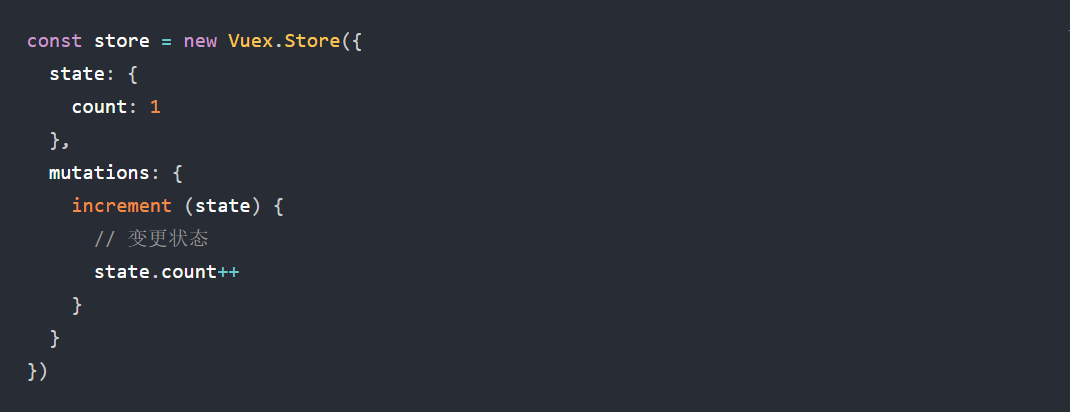


如果你想将一个getter属性另取一个名字，使用对象形式：

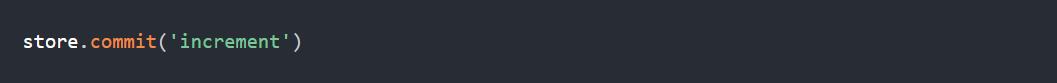


**四、mutations**

更改Vuex的store中的状态的唯一方法是提交mutation。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：

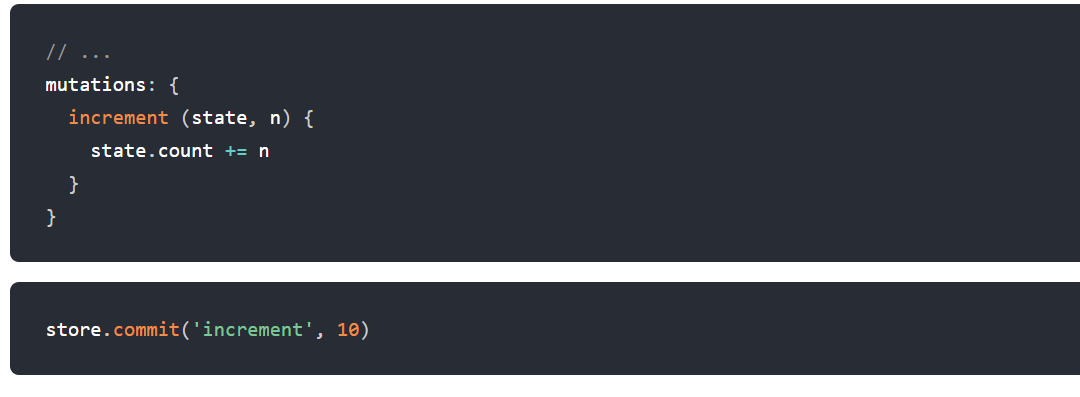


你不能直接调用一个mutation handler。这个选项更像是事件注册：“当触发一个类型为increment的mutation时，调用此函数。”要唤醒一个mutation handler，你需要以相应的type调用**store.commit**方法：

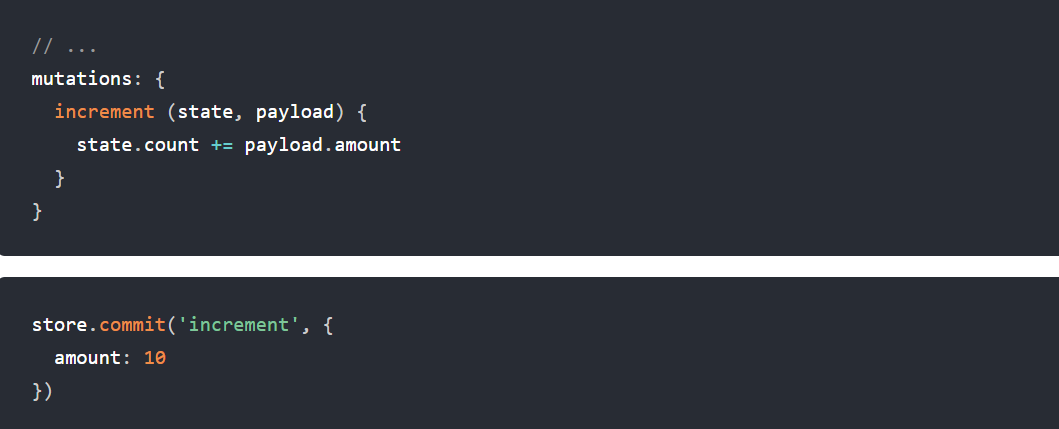


### [#](https://vuex.vuejs.org/zh/guide/mutations.html#%E6%8F%90%E4%BA%A4%E8%BD%BD%E8%8D%B7%EF%BC%88payload%EF%BC%89) 提交载荷（Payload）

你可以向store.commit传入额外的参数，即mutation的**载荷（payload）**：

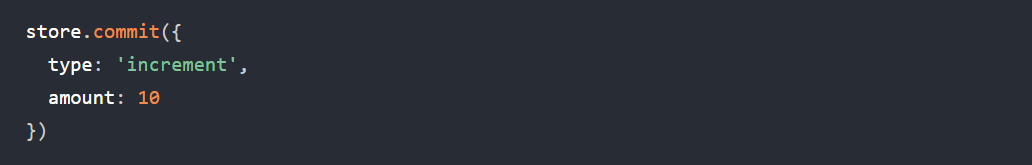


在大多数情况下，载荷应该是一个对象，这样可以包含多个字段并且记录的 mutation会更易读：

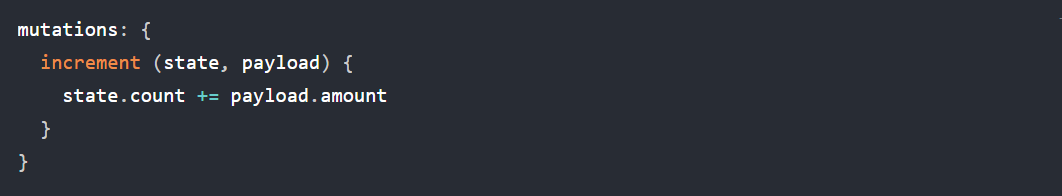


### [#](https://vuex.vuejs.org/zh/guide/mutations.html#%E5%AF%B9%E8%B1%A1%E9%A3%8E%E6%A0%BC%E7%9A%84%E6%8F%90%E4%BA%A4%E6%96%B9%E5%BC%8F) 对象风格的提交方式

提交mutation的另一种方式是直接使用包含type属性的对象：



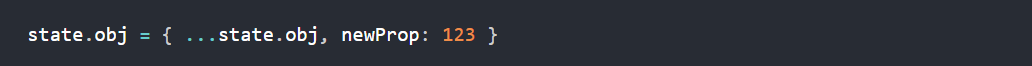
当使用对象风格的提交方式，整个对象都作为载荷传给mutation函数，因此 handler保持不变：



### [#](https://vuex.vuejs.org/zh/guide/mutations.html#mutation-%E9%9C%80%E9%81%B5%E5%AE%88-vue-%E7%9A%84%E5%93%8D%E5%BA%94%E8%A7%84%E5%88%99) Mutation 需遵守 Vue 的响应规则

既然Vuex的store中的状态是响应式的，那么当我们变更状态时，监视状态的 Vue组件也会自动更新。这也意味着Vuex中的mutation也需要与使用Vue一样遵守一些注意事项：

* 最好提前在你的store中初始化好所有所需属性。
* 当需要在对象上添加新属性时，你应该
* 使用Vue.set(obj, 'newProp', 123), 或者
* 以新对象替换老对象。例如，利用 stage-3 的对象展开运算符我们可以这样写：



### [#](https://vuex.vuejs.org/zh/guide/mutations.html#%E4%BD%BF%E7%94%A8%E5%B8%B8%E9%87%8F%E6%9B%BF%E4%BB%A3-mutation-%E4%BA%8B%E4%BB%B6%E7%B1%BB%E5%9E%8B) 使用常量替代 Mutation 事件类型

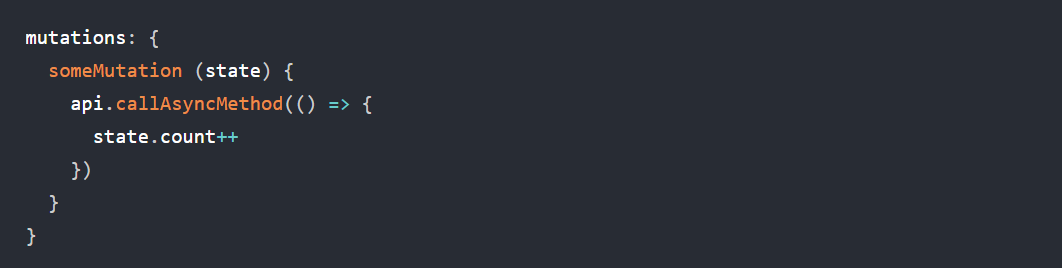
使用常量替代mutation事件类型在各种Flux实现中是很常见的模式。这样可以使linter之类的工具发挥作用，同时把这些常量放在单独的文件中可以让你的代码合作者对整个app包含的mutation一目了然：



用不用常量取决于你——在需要多人协作的大型项目中，这会很有帮助。但如果你不喜欢，你完全可以不这样做。

### [#](https://vuex.vuejs.org/zh/guide/mutations.html#mutation-%E5%BF%85%E9%A1%BB%E6%98%AF%E5%90%8C%E6%AD%A5%E5%87%BD%E6%95%B0) Mutation 必须是同步函数

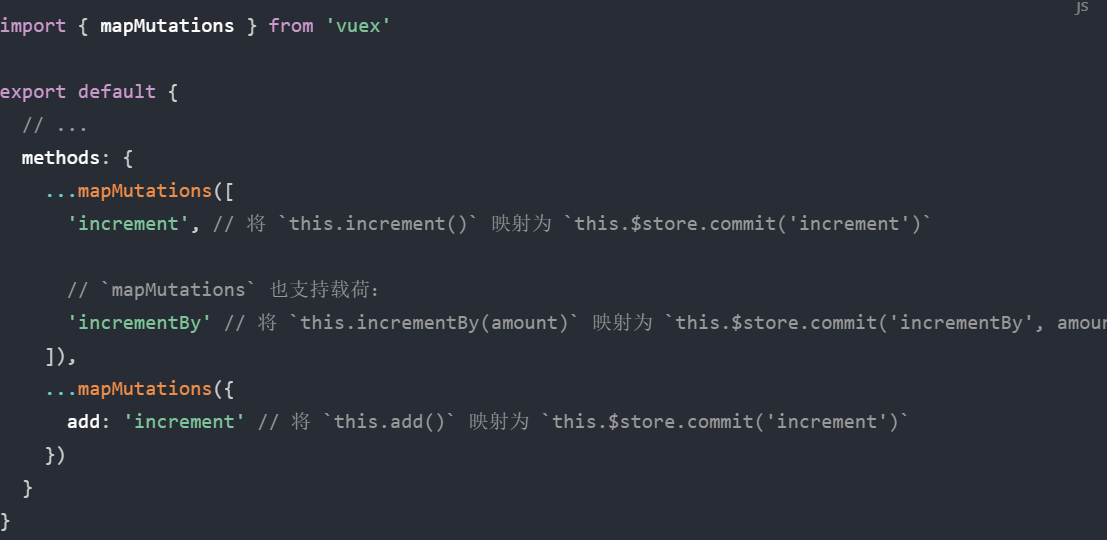
一条重要的原则就是要记住**mutation必须是同步函数**。为什么？请参考下面的例子：



现在想象，我们正在debug一app 并且观察devtool中的mutation日志。每一条 mutation 被记录，devtools都需要捕捉到前一状态和后一状态的快照。然而，在上面的例子中mutation中的异步函数中的回调让这不可能完成：因为当 mutation 触发的时候，回调函数还没有被调用，devtools不知道什么时候回调函数实际上被调用——实质上任何在回调函数中进行的状态的改变都是不可追踪的。

### [#](https://vuex.vuejs.org/zh/guide/mutations.html#%E5%9C%A8%E7%BB%84%E4%BB%B6%E4%B8%AD%E6%8F%90%E4%BA%A4-mutation) 在组件中提交 Mutation

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）。



### [#](https://vuex.vuejs.org/zh/guide/mutations.html#%E4%B8%8B%E4%B8%80%E6%AD%A5%EF%BC%9Aaction) 下一步：Action

在mutation中混合异步调用会导致你的程序很难调试。例如，当你调用了两个包含异步回调的 mutation 来改变状态，你怎么知道什么时候回调和哪个先回调呢？这就是为什么我们要区分这两个概念。在 Vuex 中，**mutation 都是同步事务**。

