\* 1.lodash.js 里面的 \_.debounce(f) 方法可以在用户输入完之后在调用函数

\* 2.vue里面自带的事件处理，可以直接处理键盘按键事件

\* 3.vue里面 脚本必须写在dom下面，否则会渲染失败

\* 4.v-model 添加.lazy修饰符，使绑定值在change的时候才修改绑定的值

\* 5.v-model 添加.trim修饰符，可以去掉用户输入的首尾空白符

npm install --registry=https://registry.npm.taobao.org

npm view <packageName> versions 这个命令可以查看npm服务器上包的已发布的版本号

# # 动态参数

vue2.6.0之后可以使用动态参数

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

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

动态参数表达式里面不能有引号和空格，同时不区分大小写

# # 自定义组件的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。

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)"

>

`

})

使用v-model

<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。

# # Vue 新增的$attrs与$listeners的详解

inheritAttrs：默认值true,继承所有的父组件属性（除props的特定绑定）作为普通的HTML特性应用在子组件的根元素上，如果你不希望组件的根元素继承特性设置inheritAttrs: false,但是class属性会继承（简单的说，inheritAttrs：true 继承除props之外的所有属性；inheritAttrs：false 只继承class属性）

$attrs--继承所有的父组件属性（除了prop传递的属性、class 和 style ），一般用在子组件的子元素上，如第一个例子的<input v-bind="$attrs"/>

$listeners--属性，它是一个对象，里面包含了作用在这个组件上的所有监听器，你就可以配合 v-on="$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。（相当于子组件继承父组件的事件）

主要用途：

用在父组件传递数据给子组件或者孙组件

<body class="">

<div id="app" class="demo">

<base-input label="姓名" class="username-input" placeholder="Enter your username" data-date-picker="activated"></base-input>

</div>

<script src="js/vue-2.5.13.js"></script>

<script>

Vue.component("base-input", {

inheritAttrs: false, //此处设置禁用继承特性 （只继承class属性）

props: ["label"],

template: `

<label>

{{label}}

{{$attrs.placeholder}}

{{$attrs["data-date-picker"]}}

<input v-bind="$attrs"/>

</label>

`,

mounted: function() {

console.log(this.$attrs);

}

})

const app = new Vue({

el: '#app',

data: {

}

});

</script>

</body>

上面的例子渲染后的HTML如下：

<label class="username-input">

姓名

Enter your username

activated

<input placeholder="Enter your username" data-date-picker="activated">

</label>

如果把上面例子中的inheritAttrs: false去掉或者改为inheritAttrs: true，最终渲染为：

<label placeholder="Enter your username" data-date-picker="activated" class="username-input">

姓名

Enter your username

activated

<input placeholder="Enter your username" data-date-picker="activated">

</label>

同时子组件可以单独使用$attrs，如上面的{{$attrs.placeholder}}依然是继承父组件的placeholder属性

&attrs和&listeners示例代码

### A组件（App.vue）

<template>

 <div id="app">

 <child1

 :p-child1="child1"

 :p-child2="child2"

 v-on:test1="onTest1" //此处监听了两个事件，可以在B组件或者C组件中直接触发

 v-on:test2="onTest2">

 </child1>

 </div>

</template>

<script>

 import Child1 from './Child1.vue';

 export default {

 data () {

 return {};

 },

 components: { Child1 },

 methods: {

 onTest1 () {

 console.log('test1 running...');

 },

 onTest2 () {

 console.log('test2 running');

 }

 }

 };

</script>

### B组件（Child1.vue）

<template>

 <div class="child-1">

 <p>in child1:</p>

 <p>props: {{pChild1}}</p>

 <p>$attrs: {{$attrs}}</p>

 <hr>

**<child2 v-bind="$attrs" v-on="$listeners"></child2>**

 </div>

</template>

<script>

 import Child2 from './Child2.vue';

 export default {

 props: ['pChild1'],

 data () {

 return {};

 },

 inheritAttrs: false,

 components: { Child2 },

 mounted () {

 this.$emit('test1');

 }

 };

</script>

结果：

in child1:

props: v\_child1

$attrs: { “p-child2”: “v\_child2”}

### C 组件 (Child2.vue)

<template>

 <div class="child-2">

 <p>in child2:</p>

 <p>props: {{pChild2}}</p>

 <p>$attrs: {{$attrs}}</p>

 <hr>

 </div>

</template>

<script>

 export default {

 props: ['pChild2'],

 data () {

 return {};

 },

 inheritAttrs: false,

 mounted () {

 this.$emit('test2');

 }

 };

</script>

结果：

in child2:

props: v\_child2

$attrs: {}

# 全局混入

混入也可以进行全局注册。使用时格外小心！一旦使用全局混入，它将影响**每一个**之后创建的 Vue 实例。使用恰当时，这可以用来为自定义选项注入处理逻辑。

// 为自定义的选项 'myOption' 注入一个处理器。

Vue.mixin({

created: function () {

var myOption = this.$options.myOption

if (myOption) {

console.log(myOption)

}

}

})

new Vue({

myOption: 'hello!'

})

// => "hello!"

请谨慎使用全局混入，因为它会影响每个单独创建的 Vue 实例 (包括第三方组件)。大多数情况下，只应当应用于自定义选项，就像上面示例一样。推荐将其作为[**插件**](https://cn.vuejs.org/v2/guide/plugins.html)发布，以避免重复应用混入。

# 自定义指令

// 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()

}

})

如果想注册局部指令，组件中也接受一个 directives 的选项：

directives: {

focus: {

// 指令的定义

inserted: function (el) {

el.focus()

}

}

}

然后你可以在模板中任何元素上使用新的 v-focus 属性，如下：

<input v-focus>

## 钩子函数

一个指令定义对象可以提供如下几个钩子函数 (均为可选)：

* bind：只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置。
* inserted：被绑定元素插入父节点时调用 (仅保证父节点存在，但不一定已被插入文档中)。
* update：所在组件的 VNode 更新时调用，**但是可能发生在其子 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 (详细的钩子函数参数见下)。
* componentUpdated：指令所在组件的 VNode **及其子 VNode** 全部更新后调用。
* unbind：只调用一次，指令与元素解绑时调用。

## 钩子函数参数

* el：指令所绑定的元素，可以用来直接操作 DOM 。
* 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" 中，绑定值为 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：字符串形式的指令表达式。例如 v-my-directive="1 + 1"中，表达式为 "1 + 1"。
  + arg：传给指令的参数，可选。例如 v-my-directive:foo 中，参数为 "foo"。
  + modifiers：一个包含修饰符的对象。例如：v-my-directive.foo.bar 中，修饰符对象为 { foo: true, bar: true }。
* vnode：Vue 编译生成的虚拟节点。
* oldVnode：上一个虚拟节点，仅在 update 和 componentUpdated 钩子中可用。

### 示例：

html：

<div id="hook-arguments-example" v-demo:foo.a.b="message"></div>

js：

Vue.directive('demo', {

bind: function (el, binding, vnode) {

var s = JSON.stringify

el.innerHTML =

'name: ' + s(binding.name) + '<br>' +

'value: ' + s(binding.value) + '<br>' +

'expression: ' + s(binding.expression) + '<br>' +

'argument: ' + s(binding.arg) + '<br>' +

'modifiers: ' + s(binding.modifiers) + '<br>' +

'vnode keys: ' + Object.keys(vnode).join(', ')

}

})

new Vue({

el: '#hook-arguments-example',

data: {

message: 'hello!'

}

})

结果：

name: "demo"  
value: "自定义指令"  
expression: "message"  
argument: "foo"  
modifiers: {"a":true,"b":true}  
vnode keys: tag, data, children, text, elm, ns, context, fnContext, fnOptions, fnScopeId, key, componentOptions, componentInstance, parent, raw, isStatic, isRootInsert, isComment, isCloned, isOnce, asyncFactory, asyncMeta, isAsyncPlaceholder

## 函数简写

在很多时候，你可能想在 bind 和 update 时触发相同行为，而不关心其它的钩子。比如这样写:

Vue.directive('color-swatch', function (el, binding) {

el.style.backgroundColor = binding.value

})

# Vue Router

全局前置守卫：

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

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

// ...

})

路由级守卫：beforeEnter

const router = new VueRouter({

routes: [

{

path: '/foo',

component: Foo,

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

// ...

}

}

]

})

组件内守卫：

* beforeRouteEnter
* beforeRouteUpdate
* beforeRouteLeave

## 命名视图

有时候想同时 (同级) 展示多个视图，而不是嵌套展示，例如创建一个布局，有 sidebar (侧导航) 和 main (主内容) 两个视图，这个时候命名视图就派上用场了。你可以在界面中拥有多个单独命名的视图，而不是只有一个单独的出口。如果 router-view 没有设置名字，那么默认为 default。

<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>

一个视图使用一个组件渲染，因此对于同个路由，多个视图就需要多个组件。确保正确使用 components 配置 (带上 s)：

const router = new VueRouter({

routes: [

{

path: '/',

components: {

default: Foo,

a: Bar,

b: Baz

}

}

]

})

# Vuex

## State:

const Counter = {

template: `<div>{{ count }}</div>`,

computed: {

count () {

return this.$store.state.count

}

}

}

使用mapState

computed: mapState([

// 映射 this.count 为 store.state.count

'count'

])

computed: {

localComputed () { /\* ... \*/ },

// 使用对象展开运算符将此对象混入到外部对象中

...mapState({

'count'

// ...

})

}

## Getter

### 基本语法

const store = new Vuex.Store({

state: {

todos: [

{ id: 1, text: '...', done: true },

{ id: 2, text: '...', done: false }

]

},

getters: {

doneTodos: state => {

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

}

}

})

### Getter 接受 getter 作为第二参数

getters: {

// ...

doneTodosCount: (state, getters) => {

return getters.doneTodos.length

}

}

### 通过方法访问

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 }

### mapGetters 辅助函数

import { mapGetters } from 'vuex'

export default {

// ...

computed: {

// 使用对象展开运算符将 getter 混入 computed 对象中

...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

### 基本用法

const store = new Vuex.Store({

state: {

count: 1

},

mutations: {

increment (state) {

// 变更状态

state.count++

}

}

})

你需要以相应的 type 调用 **store.commit** 方法：

store.commit('increment')

### 提交载荷（Payload）

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

// ...

mutations: {

increment (state, payload) {

state.count += payload.amount

}

}

store.commit('increment', {

amount: 10

})

### 对象风格的提交方式

store.commit({

type: 'increment',

amount: 10

})

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：

mutations: {

increment (state, payload) {

state.count += payload.amount

}

}

这时候payload.type应该也能取到值

### 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：

1. 最好提前在你的 store 中初始化好所有所需属性。
2. 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](https://github.com/sebmarkbage/ecmascript-rest-spread)

我们可以这样写：

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

### Mutation 必须是同步函数

现在想象，我们正在 debug 一个 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。

### 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）。

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

基本用法：

const store = new Vuex.Store({

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

实践中，我们会经常用到 ES2015 的 [参数解构](https://github.com/lukehoban/es6features#destructuring) 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：

store.dispatch('increment')

来看一个更加实际的购物车示例，涉及到**调用异步 API** 和**分发多重 mutation**：

actions: {

checkout ({ commit, state }, products) {

// 把当前购物车的物品备份起来

const savedCartItems = [...state.cart.added]

// 发出结账请求，然后乐观地清空购物车

commit(types.CHECKOUT\_REQUEST)

// 购物 API 接受一个成功回调和一个失败回调

shop.buyProducts(

products,

// 成功操作

() => commit(types.CHECKOUT\_SUCCESS),

// 失败操作

() => commit(types.CHECKOUT\_FAILURE, savedCartItems)

)

}

}

### 组合 Action

actions: {

// ...

actionB ({ dispatch, commit }) {

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

commit('someOtherMutation')

})

}

}

如果我们利用 [async / await](https://tc39.github.io/ecmascript-asyncawait/) ，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())

}

}

## Module

由于使用单一状态树，应用的所有状态会集中到一个比较大的对象。当应用变得非常复杂时，store 对象就有可能变得相当臃肿。

为了解决以上问题，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 = new Vuex.Store({

modules: {

a: moduleA,

b: moduleB

}

})

store.state.a // -> moduleA 的状态

store.state.b // -> moduleB 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**。

const moduleA = {

state: { count: 0 },

mutations: {

increment (state) {

// 这里的 `state` 对象是模块的局部状态

state.count++

}

},

getters: {

doubleCount (state) {

return state.count \* 2

}

}

}

同样，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：

const moduleA = {

// ...

actions: {

incrementIfOddOnRootSum ({ state, commit, rootState }) {

if ((state.count + rootState.count) % 2 === 1) {

commit('increment')

}

}

}

}

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：

const moduleA = {

// ...

getters: {

sumWithRootCount (state, getters, rootState) {

return state.count + rootState.count

}

}

}

### 命名空间

默认情况下，模块内部的 action、mutation 和 getter 是注册在**全局命名空间**的——这样使得多个模块能够对同一 mutation 或 action 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 = new Vuex.Store({

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']

}

}

}

}

}

})

#### 带命名空间的绑定函数

##### 繁琐：

computed: {

...mapState({

a: state => state.some.nested.module.a,

b: state => state.some.nested.module.b

})

},

methods: {

...mapActions([

'some/nested/module/foo', // -> this['some/nested/module/foo']()

'some/nested/module/bar' // -> this['some/nested/module/bar']()

])

}

##### 简化

computed: {

...mapState('some/nested/module', {

a: state => state.a,

b: state => state.b

})

},

methods: {

...mapActions('some/nested/module', [

'foo', // -> this.foo()

'bar' // -> this.bar()

])

}

或者

import { createNamespacedHelpers } from 'vuex'

const { mapState, mapActions } = createNamespacedHelpers('some/nested/module')

export default {

computed: {

// 在 `some/nested/module` 中查找

...mapState({

a: state => state.a,

b: state => state.b

})

},

methods: {

// 在 `some/nested/module` 中查找

...mapActions([

'foo',

'bar'

])

}

}

### 模块动态注册

在 store 创建**之后**，你可以使用 store.registerModule 方法注册模块：

// 注册模块 `myModule`

store.registerModule('myModule', {

// ...

})

// 注册嵌套模块 `nested/myModule`

store.registerModule(['nested', 'myModule'], {

// ...

})

之后就可以通过 store.state.myModule 和 store.state.nested.myModule 访问模块的状态。

模块动态注册功能使得其他 Vue 插件可以通过在 store 中附加新模块的方式来使用 Vuex 管理状态。例如，[vuex-router-sync](https://github.com/vuejs/vuex-router-sync)

插件就是通过动态注册模块将 vue-router 和 vuex 结合在一起，实现应用的路由状态管理。

你也可以使用 store.unregisterModule(moduleName) 来动态卸载模块。注意，你不能使用此方法卸载静态模块（即创建 store 时声明的模块）。

### 严格模式

在严格模式下，无论何时发生了状态变更且不是由 mutation 函数引起的，将会抛出错误。这能保证所有的状态变更都能被调试工具跟踪到。

**不要在发布环境下启用严格模式**！严格模式会深度监测状态树来检测不合规的状态变更——请确保在发布环境下关闭严格模式，以避免性能损失。

const store = new Vuex.Store({

// ...

strict: process.env.NODE\_ENV !== 'production'

})

# Vue生命周期

beforecreated：el 和 data 并未初始化   
created:完成了 data 数据的初始化，el没有  
beforeMount：完成了 el 和 data 初始化   
mounted ：完成挂载

beforeUpdate,updated,activated,deactivated,beforeDestory,desoryed

beforeCreate: **function**() {

console.group('------beforeCreate创建前状态------');

console.log("%c%s", "color:red" , "el : " + **this**.$el); //undefined

console.log("%c%s", "color:red","data : " + **this**.$data); //undefined

console.log("%c%s", "color:red","message: " + **this**.message)

},

created: **function**() {

console.group('------created创建完毕状态------');

console.log("%c%s", "color:red","el : " + **this**.$el); //undefined

console.log("%c%s", "color:red","data : " + **this**.$data); //已被初始化

console.log("%c%s", "color:red","message: " + **this**.message); //已被初始化

},

beforeMount: **function**() {

console.group('------beforeMount挂载前状态------');

console.log("%c%s", "color:red","el : " + (**this**.$el)); //已被初始化

console.log(**this**.$el);

console.log("%c%s", "color:red","data : " + **this**.$data); //已被初始化

console.log("%c%s", "color:red","message: " + **this**.message); //已被初始化

},

mounted: **function**() {

console.group('------mounted 挂载结束状态------');

console.log("%c%s", "color:red","el : " + **this**.$el); //已被初始化

console.log(**this**.$el);

console.log("%c%s", "color:red","data : " + **this**.$data); //已被初始化

console.log("%c%s", "color:red","message: " + **this**.message); //已被初始化

},

beforeUpdate: **function** () {

console.group('beforeUpdate 更新前状态===============》');

console.log("%c%s", "color:red","el : " + **this**.$el);

console.log(**this**.$el);

console.log("%c%s", "color:red","data : " + **this**.$data);

console.log("%c%s", "color:red","message: " + **this**.message);

},

updated: **function** () {

console.group('updated 更新完成状态===============》');

console.log("%c%s", "color:red","el : " + **this**.$el);

console.log(**this**.$el);

console.log("%c%s", "color:red","data : " + **this**.$data);

console.log("%c%s", "color:red","message: " + **this**.message);

},

beforeDestroy: **function** () {

console.group('beforeDestroy 销毁前状态===============》');

console.log("%c%s", "color:red","el : " + **this**.$el);

console.log(**this**.$el);

console.log("%c%s", "color:red","data : " + **this**.$data);

console.log("%c%s", "color:red","message: " + **this**.message);

},

destroyed: **function** () {

console.group('destroyed 销毁完成状态===============》');

console.log("%c%s", "color:red","el : " + **this**.$el);

console.log(**this**.$el);

console.log("%c%s", "color:red","data : " + **this**.$data);

console.log("%c%s", "color:red","message: " + **this**.message)

}