Vuex - 状态管理

一 开始

1. NPM安装

|  |
| --- |
| npm install vuex --save |

1. 引用

|  |
| --- |
| import Vuex from ‘vuex’;  Vue.use(Vuex); |

3.简单使用

|  |
| --- |
| // 如果在模块化构建系统中，请确保在开头调用了 Vue.use(Vuex)  const store = new Vuex.Store({  state: {  count: 0  },  mutations: {  increment (state) {  state.count++  }  }  }) |
| 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：  store.commit('increment')  console.log(store.state.count) // -> 1 |

二 概念

1. state - 类似Vue的data参数

◎ 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。

◎ 在Vue组件中获得Vuex状态

那么我们如何在 Vue 组件中展示状态呢？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：

|  |
| --- |
| // 创建一个 Counter 组件  const Counter = {  template: `<div>{{ count }}</div>`,  computed: {  count () {  return store.state.count  }  }  } |

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的全局状态单例。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)）：

|  |
| --- |
| 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>  `  }) |

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 （类似router的注册）访问到。让我们更新下 Counter 的实现：

|  |
| --- |
| const Counter = {  template: `<div>{{ count }}</div>`,  computed: {  count () {  return this.$store.state.count  }  }  } |

◎ 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：

|  |
| --- |
| // 在单独构建的版本中辅助函数为 Vuex.mapState  import { 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  }  })  } |

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-3 阶段），我们可以极大地简化写法：  computed: {  localComputed () { /\* ... \*/ },  // 使用对象展开运算符将此对象混入到外部对象中  ...mapState({  // ...  })  } |

2.Getters - 类似Vue的computed

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：

|  |
| --- |
| computed: {  doneTodosCount () {  return this.$store.state.todos.filter(todo => todo.done).length  }  } |

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『getters』（可以认为是 store 的计算属性）。Getters 接受 state 作为其第一个参数：

|  |
| --- |
| 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)  }  }  }) |

Getters 会暴露为 store.getters 对象：

|  |
| --- |
| store.getters.doneTodos // -> [{ id: 1, text: '...', done: true }] |

Getters 也可以接受其他 getters 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：

|  |
| --- |
| computed: {  doneTodosCount () {  return this.$store.getters.doneTodosCount  }  } |

◎ mapGetters 辅助函数

mapGetters 辅助函数仅仅是将 store 中的 getters 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 getters 混入 computed 对象中  ...mapGetters([ // 数组形式  '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({ // 对象形式  // 映射 this.doneCount 为 store.getters.doneTodosCount  doneCount: 'doneTodosCount'  }) |

3.Mutations - 类似Vue的methods

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：

|  |
| --- |
| const store = new Vuex.Store({  state: {  count: 1  },  mutations: {  increment (state) {  // 变更状态  state.count++  }  }  }) |

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：

|  |
| --- |
| store.commit('increment') |

◎ 提交载荷 —— 理解：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) |

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：

|  |
| --- |
| mutations: {  increment (state, payload) {  state.count += payload.amount  }  }  store.commit('increment', {  amount: 10  }) |

◎ 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：

|  |
| --- |
| store.commit({  type: 'increment', //定义type  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  }  } |

◎ 组件中提交

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' // 映射 this.increment() 为 this.$store.commit('increment')  ]),  ...mapMutations({  add: 'increment' // 映射 this.add() 为 this.$store.commit('increment')  })  }  } |

4.Actions

Actions类似mutations，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。

◎ 让我们来注册一个简单的 action：

|  |
| --- |
| const store = new Vuex({  state: {  count: 0  },  mutations: {  increment(state) {  state++;  }  },  actions: {  increment(context) { // 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 时，你就知道 context 对象为什么不是 store 实例本身了。 |
| 实践中，我们会经常会用到 ES2015 的 参数解构 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 - 提交的变种

|  |
| --- |
| store.dispatch('increment') |

乍一眼看上去感觉多此一举，我们直接分发 mutation 岂不更方便？实际上并非如此，还记得 mutation 必须同步执行这个限制么？Action 就不受约束！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  }) |

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 mutations：

|  |
| --- |
| 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)  )  }  } |

注意我们正在进行一系列的异步操作，并且通过提交 mutation 来记录 action 产生的副作用（即状态变更）。

◎ 在组件中分发Action

你在组件中使用 this.$store.dispatch('xxx') 分发 action，或者使用 mapActions 辅助函数将组件的 methods 映射为 store.dispatch 调用（需要先在根节点注入 store）：

|  |
| --- |
| import { mapActions } from 'vuex'  export default {  methods: {  ...mapActions([  'increment' // 映射 this.increment() 为 this.$store.dispatch('increment')  ]),  ...mapActions({  add: 'increment' // 映射 this.add() 为 this.$store.dispatch('increment')  })  }  } |

◎ 组合Actions

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的回调函数返回的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 这个 JavaScript 即将到来的新特性，我们可以像这样组合 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。

5. Modules

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。如果希望你的模块更加自包含或提高可重用性，你可以通过添加 namespaced: true 的方式使其成为命名空间模块。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']  }  }  }  }  }  }) |

◎ 在命名空间模块内访问全局内容（Global Assets）

如果你希望使用全局 state 和 getter，rootState 和 rootGetter 会作为第三和第四参数传入 getter，也会通过 context 对象的属性传入 action。

若需要在全局命名空间内分发 action 或提交 mutation，将 { root: true } 作为第三参数传给 dispatch 或 commit 即可。

|  |
| --- |
| modules: {  foo: {  namespaced: true,  getters: {  // 在这个模块的 getter 中，`getters` 被局部化了  // 你可以使用 getter 的第四个参数来调用 `rootGetters`  someGetter (state, getters, rootState, rootGetters) {  getters.someOtherGetter // -> 'foo/someOtherGetter'  rootGetters.someOtherGetter // -> 'someOtherGetter'  },  someOtherGetter: state => { ... }  },  actions: {  // 在这个模块中， dispatch 和 commit 也被局部化了  // 他们可以接受 `root` 属性以访问根 dispatch 或 commit  someAction ({ dispatch, commit, getters, rootGetters }) {  getters.someGetter // -> 'foo/someGetter'  rootGetters.someGetter // -> 'someGetter'  dispatch('someOtherAction') // -> 'foo/someOtherAction'  dispatch('someOtherAction', null, { root: true }) // -> 'someOtherAction'  commit('someMutation') // -> 'foo/someMutation'  commit('someMutation', null, { root: true }) // -> 'someMutation'  },  someOtherAction (ctx, payload) { ... }  }  }  } |

◎ 带命名空间的绑定函数

当使用 mapState, mapGetters, mapActions 和 mapMutations 这些函数来绑定命名空间模块时，写起来可能比较繁琐：

|  |
| --- |
| computed: {  ...mapState({  a: state => state.some.nested.module.a,  b: state => state.some.nested.module.b  })  },  methods: {  ...mapActions([  'some/nested/module/foo',  'some/nested/module/bar'  ])  } |

对于这种情况，你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  })  },  methods: {  ...mapActions('some/nested/module', [  '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 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）。

◎ 模块重用

有时我们可能需要创建一个模块的多个实例，例如：

● 创建多个 store，他们公用同一个模块

● 在一个 store 中多次注册同一个模块

如果我们使用一个纯对象来声明模块的状态，那么这个状态对象会通过引用被共享，导致状态对象被修改时会 store 或模块间数据互相污染的问题。

实际上这和 Vue 组件内的 data 是同样的问题。因此解决办法也是相同的——使用一个函数来声明模块状态（仅 2.3.0+ 支持）：

|  |
| --- |
| const MyReusableModule = {  state () {  return {  foo: 'bar'  }  },  // mutation, action 和 getter 等等...  } |