

核心概念

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。与 vuex 4 相匹配的版本是 vue 3。学习本文需要一些前置技能，默认读者已熟练掌握了 vue2 + vuex 的基础用法，以及对 vue3 有过初步的了解。本文目的在于对 vuex 的源码原理进行深入研究，达到对其运用自如的目标，同时对状态管理库的设计有基本的认识。

安装运行环境

```
1 yarn create vite vue3-vuex4<项目名, 自定义>
2 cd vue3-vuex4
3 yarn // 安装 node_modules
4 yarn add vuex@next // 安装 vuex
5 yarn dev // 启动项目并访问 localhost:3000
```

本文讲解的版本 vue^3.2.25, vuex^4.0.2

从用法开始

- createStore 不传参数

在 vue3 中使用 vuex，入口代码如下（main.js）：

```
1 import { createApp } from 'vue'
2 import { createStore } from 'vuex'
3 import App from './App.vue'
4
5 const app = createApp(App);
6 const store = createStore({})
7
8 app.use(store);
9 console.log(store)
10 app.mount('#app')
```

查看 store 如下：

```

Store2 {_committing: false, _actions: {...}, _actionSubscribers: Array(0), _mutations: {...}, _wrappedGetters:
{...}, ...}
  ▶ commit: f boundCommit(type, payload, options2)
  ▶ dispatch: f boundDispatch(type, payload)
  ▶ getters: {}
  ▶ strict: false
  ▶ _actionSubscribers: []
  ▶ _actions: {}
  ▶ _committing: false
  ▶ _devtools: undefined
  ▶ _makeLocalGettersCache: {}
  ▶ _modules: ModuleCollection2 {root: Module2}
  ▶ _modulesNamespaceMap: {}
  ▶ _mutations: {}
  ▶ _state: Proxy {data: {...}}
  ▶ _subscribers: []
  ▶ _wrappedGetters: {}
  ▶ state: Proxy
  ▶ [[Prototype]]: Object
    ▶ commit: f commit(_type, _payload, _options)
    ▶ dispatch: f dispatch(_type, _payload)
    ▶ hasModule: f hasModule(path)
    ▶ hotUpdate: f hotUpdate(newOptions)
    ▶ install: f install(app, injectKey)
    ▶ registerModule: f registerModule(path, rawModule, options)
    ▶ replaceState: f replaceState(state)
    ▶ subscribe: f subscribe(fn, options)
    ▶ subscribeAction: f subscribeAction(fn, options)
    ▶ unregisterModule: f unregisterModule(path)
    ▶ watch: f watch$(getter, cb, options)
    ▶ _withCommit: f _withCommit(fn)
    ▶ constructor: f Store2(options)
      state: (...)
    ▶ get state: f ()
    ▶ set state: f (v)
    ▶ [[Prototype]]: Object

```

> |

我们知道，app.use 注册插件（vue2 则为 Vue.use）时，会自动调用参数的 install 方法（例如 vue-router），定位到 node_modules/vuex/dist/vuex.esm-bundler.js 文件，找到 install 定义的位置：

```

1  var storeKey = 'store'
2  ...
3  Store.prototype.install = function install (app, injectKey) {
4    app.provide(injectKey || storeKey, this);
5    app.config.globalProperties.$store = this;
6    // 下面的逻辑是开发环境启用开发工具的，核心代码是上面这两行
7    var useDevtools = this._devtools !== undefined
8      ? this._devtools
9      : (process.env.NODE_ENV !== 'production') || __VUE_PROD_DEVTOOLS__;
10
11    if (useDevtools) {
12      addDevtools(app, this);
13    }
14  };

```

跨组件传递状态的方法中，有 provide/inject 组合因此下面的代码可以访问到 store：

```
1 // App.vue 文件修改
2 <script setup>
3 import { inject } from 'vue'
4 import HelloWorld from './components/HelloWorld.vue'
5
6 const store = inject('store')
7 </script>
8
9 <template>
10   <pre>{{ JSON.stringify(store, null, 2) }}</pre>
11   <HelloWorld msg="hello vue" />
12 </template>
```

呈现结果：

```

{
  "_committing": false,
  "_actions": {},
  "_actionSubscribers": [],
  "_mutations": {},
  "_wrappedGetters": {},
  "_modules": {
    "root": {
      "runtime": false,
      "_children": {},
      "_rawModule": {},
      "state": {},
      "context": {}
    }
  },
  "_modulesNamespaceMap": {},
  "_subscribers": [],
  "_makeLocalGettersCache": {},
  "strict": false,
  "getters": {},
  "_state": {
    "data": {}
  }
}

```

hello vue

Recommended IDE setup: [VSCode](#) + [Volar](#)

[Vite Documentation](#) | [Vue 3 Documentation](#)

count is: 0

Edit `components/HelloWorld.vue` to test hot module replacement.

而 `app.config.globalProperties.$store` 将 vue 组件实例的原型上（全局）挂载了 `$store`，所以在 vue2 组件常见以下写法：

```

1  this.$store.commit(...);
2  this.$store.dispatch(...);
3
4  // 当然，template 中直接写成下面这样也是可以的，因为实例 this 在模板内是省略的
5
6  // import { inject } from 'vue'
7  // const store = inject('store')
8
9  {{ JSON.stringify($store, null, 2) }}

```

- createStore 加入参数

```
1 // main.js
2 ...
3
4 const app = createApp(App);
5 const store = createStore({
6   state: {
7     count: 1
8   },
9   mutations: {
10     ADD(state, payload) {
11       state.count += payload
12     }
13   },
14   actions: {
15     add(context) {
16       context.commit('ADD', 1)
17     }
18   },
19   getters: {
20     countByGetter: state => state.count
21   },
22   modules: {}
23 })
24
25 app.use(store);
26 console.log(store)
27 app.mount('#app')
```

修改 App.vue:

```
1 <script setup>
2 import { inject } from 'vue'
3
4 const { dispatch, getters } = inject('store')
5 </script>
6
7 <template>
8   <pre>{{ JSON.stringify($store, null, 2) }}</pre>
```

```
9   <button @click="dispatch('add')">{{getters.countByGetter}}</button>
10 </template>
```

当然在类组件（尤其是 vue2 中）使用 vuex 辅助函数的话应该不少同学应该更熟悉：

```
1 // 改造 HelloWorld.vue 组件
2 <script>
3 import { mapGetters, mapActions } from 'vuex'
4
5 export default {
6   computed: {
7     ...mapGetters(['countByGetter'])
8   },
9   methods: {
10    ...mapActions(['add'])
11   }
12 }
13 </script>
14
15 <template>
16   <p>HelloWorld 使用 store</p>
17   <button @click="add">++ {{countByGetter}}</button>
18 </template>
19
```

这样 App.vue 引入 HelloWorld.vue 就能看到一样的变化了。

核心源码分析

为便于分析，我们将 vuex 的核心源码分 3 部分解读，每部分将结合具体的业务代码进行展开（源码文件 node_modules/vuex/dist/vuex.esm-bundler.js）。

1. 创建 store

```
1 var Store = function Store (options) {
2   ...
3   // 这部分就是前文控制台打印的诸多属性
4   this._committing = false;
5   this._actions = Object.create(null);
6   this._actionSubscribers = [];
```

```

7   this._mutations = Object.create(null);
8   this._wrappedGetters = Object.create(null);
9   this._modules = new ModuleCollection(options); // store._modules 持有参数转换的各种信息
10  this._modulesNamespaceMap = Object.create(null);
11  this._subscribers = [];
12  this._makeLocalGettersCache = Object.create(null);
13  this._devtools = devtools;
14  ...
15  var store = this;
16  var ref = this;
17  var dispatch = ref.dispatch;
18  var commit = ref.commit;
19  this.dispatch = function boundDispatch (type, payload) {
20    return dispatch.call(store, type, payload)
21  };
22  this.commit = function boundCommit (type, payload, options) {
23    return commit.call(store, type, payload, options)
24  };
25  // dispatch 和 commit 将原型上的同名方法重写，目的就是保证
26  // 当解构 commit/dispatch 时，this 指向依旧为 store 实例。
27  var state = this._modules.root.state;
28
29  installModule(this, state, [], this._modules.root); // -----特别关注
30  resetStoreState(this, state); // -----特别关注
31  ...
32 }

```

可见 ModuleCollection 及末尾的 installModule 和 resetStoreState 是需要关注的方法。

- **ModuleCollection** 将参数 **options** 进行转换，本质是对 **options** 对象的包装与扩充，扩充结果作为 **store._modules** 的值。

```

1  // ModuleCollection
2  var ModuleCollection = function ModuleCollection (rawRootModule) {
3    // rawRootModule 就是 createStore 的参数，由此，
4    // 我们查找 ModuleCollection.prototype.register 方法
5    this.register([], rawRootModule, false);
6  };
7  ...
8  // 务必按照编号顺序理解 ----- (0)
9  ModuleCollection.prototype.register = function register (path, rawModule, runtime) {
10   var this$1$1 = this;

```

```

11  if ( runtime === void 0 ) runtime = true;
12  ...
13
14  var newModule = new Module(rawModule, runtime);
15  if (path.length === 0) { // 模块是全局模块, 挂在 root 属性下 -----(1)
16      this.root = newModule;
17  } else { // 命名空间下的子模块, 根据路径关系被父模块所引用 -----(3)
18      var parent = this.get(path.slice(0, -1));
19      parent.addChild(path[path.length - 1], newModule);
20      // -----(4) 子模块最终在这里执行结束, 根据 13、15 行, 所有的模块均为 Module 实例
21  }
22
23  // rawModule 就是 createStore 的参数, 如果定义了 modules 配置, 就会对子模块
24  // ——注册, 但 register 第一个参数不再是空数组, 子模块会挂载对应的命名空间下
25  if (rawModule.modules) { // -----(2)
26      forEachValue(rawModule.modules, function (rawChildModule, key) {
27          this.$1.$1.register(path.concat(key), rawChildModule, runtime);
28      });
29  }
30  };

```

因为 Module 类的实现（下面的代码）我们知道（rawModule 是 createStore 的参数 options），Module 的实例通过 _rawModule 引用着最初的 options。结论就是，register 方法执行以后，ModuleCollection 实例引用 root（15行）module，root module 通过 addChild（18 行）将后代子模块的实例递归引用（在 _children 属性下）。

```

1  var Module = function Module (rawModule, runtime) {
2      ...
3      this._children = Object.create(null);
4      this._rawModule = rawModule;
5      ...
6  };

```

启用 modules 配置（为便于理解模块化，这里直接使用与 root 一样的参数）验证上面的结果：

```

1  const moduleRoot = {
2      state: {
3          count: 1
4      },
5      mutations: {

```



```
6     ADD(state, payload) {
7         state.count += payload
8     }
9 },
10 actions: {
11     add(context) {
12         context.commit('ADD', 1)
13     }
14 },
15 getters: {
16     countByGetter: state => state.count
17 },
18 };
19
20 const moduleA = {
21     namespaced: true,
22     state: {
23         ...moduleRoot.state
24     },
25     mutations: {
26         ...moduleRoot.mutations
27     },
28     actions: {
29         ...moduleRoot.actions
30     },
31     getters: {
32         ...moduleRoot.getters
33     }
34 };
35 const moduleB = const moduleA = {
36     namespaced: true,
37     state: {
38         ...moduleRoot.state
39     },
40     mutations: {
41         ...moduleRoot.mutations
42     },
43     actions: {
44         ...moduleRoot.actions
45     },
```

```

46   getters: {
47     ...moduleRoot.getters
48   }
49 };
50
51 const store = createStore({
52   ...moduleRoot,
53   modules: {
54     moduleA,
55     moduleB
56   }
57 })
58 console.log(store)

```

我们可以得到 store:

```

1  {
2    ...
3    "_mutations": {
4      "ADD": [Function],
5      "moduleA/ADD": [Function],
6      "moduleB/ADD": [Function]
7    },
8    _modules: {
9      root: {
10       context: {},
11       runtime: false,
12       state: {count: 1, moduleA: {...}, moduleB: {...}},
13       _children: [{ // moduleA:
14         runtime: false, _children: {...}, _rawModule: {...}, state: {...}, context: {...}
15       }, { // moduleB
16         runtime: false, _children: {...}, _rawModule: {...}, state: {...}, context: {...}
17       }],
18       _rawModule: {state: {...}, mutations: {...}, actions: {...}, getters: {...}, modules: {...}}
19     }
20   },
21   "getters": {
22     "countByGetter": 1,
23     "moduleA/countByGetter": 1,
24     "moduleB/countByGetter": 1

```

```

25   },
26   "_actions": {
27     "add": [Function],
28     "moduleA/add": [Function],
29     "moduleB/add": [Function]
30   },
31 }

```

moduleA、moduleB 与 store._modules.root 格式一样，都是 Module 的实例。由上述结构可见，createStore 就是按照 options 的结构生成 modules 树（含 state），根据命名空间，把所有 module 的 getters、actions、mutations 集中在 store 实例的根属性上。这意味着如果没有配置命名空间，这些属性将变成全局模块的属性，很容易出现同名冲突。

- installModule 的作用就是把 state、actions、mutations、getters 分别注册到相应的模块名称下

```

1  function installModule (store, rootState, path, module, hot) {
2    var isRoot = !path.length;
3    var namespace = store._modules.getNamespace(path);
4    if (module.namespaced) { // 模块指定命名空间时，会保存在 _modulesNamespaceMap 下
5      store._modulesNamespaceMap[namespace] = module;
6    }
7    // state 的注册
8    if (!isRoot && !hot) { // 初始化的子模块执行，状态会按照空间缓存在上层 state 下，见上 12 行
9      var parentState = getNestedState(rootState, path.slice(0, -1));
10     var moduleName = path[path.length - 1];
11     store._withCommit(function () {
12       parentState[moduleName] = module.state;
13     });
14   }
15
16   var local = module.context = makeLocalContext(store, namespace, path);
17   // 注册 mutations，效果见上方第 3~6 行
18   module.forEachMutation(function (mutation, key) {
19     var namespacedType = namespace + key;
20     registerMutation(store, namespacedType, mutation, local);
21   });
22   // 注册 actions，效果与见上方第 25~28 行
23   module.forEachAction(function (action, key) {
24     var type = action.root ? key : namespace + key;
25     var handler = action.handler || action;

```

```

26     registerAction(store, type, handler, local);
27 });
28 // 注册 getters, 上方 20~23 行
29 module.forEachGetter(function (getter, key) {
30     var namespacedType = namespace + key;
31     registerGetter(store, namespacedType, getter, local);
32 });
33 // 遍历当前模块存在下级列表, 进行递归注册, 这次因为 path 参数"加长"了, 所以会有第 8 行的逻辑
34 module.forEachChild(function (child, key) {
35     installModule(store, rootState, path.concat(key), child, hot);
36 });
37 }

```

◦ **resetStoreState** 将树状结构的 state, 统一使用 reactive 代理后, 挂载 store._state 下, 这样未来对 state 的更新, 将具有被追踪的能力。**resetStoreState** 除了初始化会调用, 重置 store 时以及动态注册模块时都会用到。

```

1 function resetStoreState (store, state, hot) {
2     var oldState = store._state;
3     ...
4     store._state = reactive({
5         data: state
6     });
7     ...
8 }

```

2. store 分发

根据前文, store._modules.root.context 的输出结构如下:

```

1 {
2     "commit": Function,
3     "dispatch": Function,
4     "getters": Object,
5     "state": {
6         "count": 1,
7         "moduleA": {"count": 1},
8         "moduleB": {"count": 1}
9     },
10    "_children": [moduleA, moduleB]

```

```
11 }
```

如果给你一个这样的对象，让你实现一个函数，可以通过参数返回 state 中的不同部分，这个函数就是 mapState，你会怎么做？比如这样：

```
1  /*
2  * 假如 path 是一个字符串数组，根据 mapState 的用法，
3  * 其返回值应当形如 { computedPropA() {}, computedPropB() {}, ... }
4  * 例如 mapState(['x', 'y'])
5  * 返回值 { x() { return context.state.x; }, y() { return context.state.y } }
6  * 可以像下面这样(先忽略命名空间)
7  */
8  // round 1
9  function mapState(path) {
10     return path.reduce((prev, key) => ({
11         ...prev,
12         [key]() {
13             return context.state[key]
14         }
15     }), {});
16 }
17 // round 2 加上命名空间
18 function mapState(namespace, path) {
19     return path.reduce((prev, key) => ({
20         ...prev,
21         [key]() {
22             return context.state[namespace][key]
23         }
24     }), {});
25 }
26 // round 3 参数类型扩展，允许 key-value 形态
27 function mapState(namespace, states) {
28     const path = normalizeMap(states) // 转化一下参数类型
29     return path.reduce((prev, key) => ({
30         ...prev,
31         [key]() {
32             return context.state[namespace][key]
33         }
34     }), {});
35 }
```

```

36 // round 4 context 作为上下文，直接从组件实例中取
37 function mapState(namespace, states) {
38   const path = normalizeMap(states)
39   return path.reduce((prev, key) => ({
40     ...prev,
41     [key]() {
42       const state = this.$store.state; // 还记得最开始 install 时的 provide 吗
43       return state[namespace][key]
44     }
45   }), {}));
46 }
47 // round 5 上面要么取的全局空间的状态，要么取命名空间的状态，都要的话怎么办？加入函数！
48 function mapState(namespace, states) {
49   const path = normalizeMap(states)
50   return path.reduce((prev, key) => ({
51     ...prev,
52     [key]() {
53       const state = this.$store.state;
54       const getters = this.$store.getters;
55       return typeof key === 'function' ? key(state, getters) : state[namespace][key]
56     }
57   }), {}));
58 }
59 // 真实的使用场景 1：默认全局空间使用 state
60 computed: {
61   ...mapState({
62     a: (state, getters) => state.moduleA.count,
63     globalCount: (state, getters) => state.count
64   })
65 }
66 // 真实的使用场景 2：
67 computed: {
68   ...mapState('moduleA', ['count']) // 模块 A 的数据
69 }
70 // 真实的使用场景 3：
71 computed: { // 带命名空间访问 this['moduleB/count'], 略丑
72   ...mapState(['moduleA/count', 'moduleB/count'])
73 }
74 // 真实的使用场景 4：同 2，但可以重命名
75 computed: {

```

```

76   ...mapState('moduleB', {
77     countB: state => state.count
78   })
79 }
80 // 真实的使用场景 5: 提前固定命名空间
81 import { createNamespacedHelpers } from 'vuex'
82 const { mapState } = createNamespacedHelpers('moduleA')
83
84 computed: {
85   // 下面的参数 state 就是 context.state.moduleA 了
86   ...mapState({
87     countA: state => state.count
88   })
89 },

```

有了以上的基础，源码理解起来便十分顺畅了：

```

1  var mapState = normalizeNamespace(function (namespace, states) {
2    var res = {};
3    normalizeMap(states).forEach(function (ref) {
4      var key = ref.key;
5      var val = ref.val;
6
7      res[key] = function mappedState () {
8        var state = this.$store.state;
9        var getters = this.$store.getters;
10       if (namespace) {
11         var module = getModuleByNamespace(this.$store, 'mapState', namespace);
12         if (!module) {
13           return
14         }
15         state = module.context.state;
16         getters = module.context.getters;
17       }
18       return typeof val === 'function'
19         ? val.call(this, state, getters)
20         : state[val]
21     };
22     // mark vuex getter for devtools
23     res[key].vuex = true;

```

```

24   });
25   return res
26 });

```

同理，mapActions，mapMutations，mapGetters 不再赘述。

3. state 变更引发副作用和视图更新

上文已经提到过 resetStoreState 方法，源码有这样的一段逻辑十分重要：

```

1  import { inject, reactive, watch } from 'vue'
2  ...
3  function resetStore (store, hot) {
4    ...
5    store._state = reactive({
6      data: state
7    });
8  }

```

如果像下面这样写就会发现触发 store.state 的变更将不能引发页面的更新：

```

1  store._state = {
2    data: state
3  };

```

reactive 的逻辑在学习 vue3 部分可以了解到，它对一个对象的访问进行了深度代理，所以当执行 this.\$store.commit => mutations => state.count++ 时，track 了 count 属性的副作用函数、render 函数（template 编译结果），将会重新调用。eg：

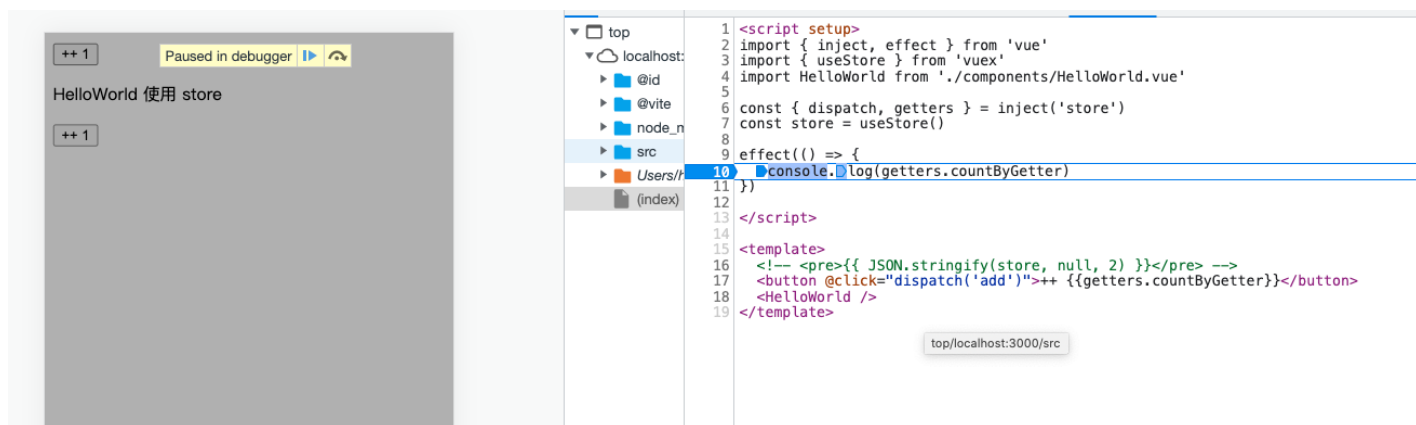
```

1  // App.vue
2  effect(() => {
3    console.log(getters.countByGetter)
4  })
5  <template>
6    <!-- <pre>{{ JSON.stringify(store, null, 2) }}</pre> -->
7    <button @click="dispatch('add')">++ {{getters.countByGetter}}</button>
8    <HelloWorld />
9  </template>

```

按钮的点击更改了 store.state.count，计算值 store.getters.countByGetter 初次使用时已经被追踪了（effect 和 render 两处），所以点击导致的状态变化，将引发 effect 的回调、render 的重新执

行，断点如下：



不过追踪与渲染的过程发生在 vue 源码中，vuex 仅仅负责响应式数据的维护。熟悉 vuex4 后，可以更进一步地学习 [Pinia](#)，即 vuex5。