[Vuex框架原理与源码分析](https://tech.meituan.com/2017/04/27/vuex-code-analysis.html)

2017年04月27日 作者: 明裔 [文章链接](https://tech.meituan.com/2017/04/27/vuex-code-analysis.html) 18976字 38分钟阅读

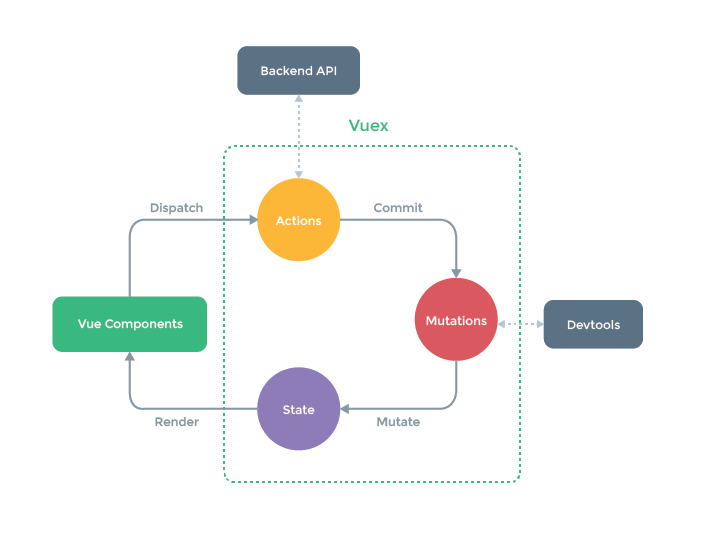
Vuex是一个专为Vue服务，用于管理页面数据状态、提供统一数据操作的生态系统。它集中于MVC模式中的Model层，规定所有的数据操作必须通过 action - mutation - state change 的流程来进行，再结合Vue的数据视图双向绑定特性来实现页面的展示更新。统一的页面状态管理以及操作处理，可以让复杂的组件交互变得简单清晰，同时可在调试模式下进行时光机般的倒退前进操作，查看数据改变过程，使code debug更加方便。

最近在开发的项目中用到了Vuex来管理整体页面状态，遇到了很多问题。决定研究下源码，在答疑解惑之外，能深入学习其实现原理。

先将问题抛出来，使学习和研究更有针对性：  
1. 使用Vuex只需执行 Vue.use(Vuex)，并在Vue的配置中传入一个store对象的示例，store是如何实现注入的？  
2. state内部是如何实现支持模块配置和模块嵌套的？  
3. 在执行dispatch触发action（commit同理）的时候，只需传入（type, payload），action执行函数中第一个参数store从哪里获取的？  
4. 如何区分state是外部直接修改，还是通过mutation方法修改的？  
5. 调试时的“时空穿梭”功能是如何实现的？

注：本文对有Vuex有实际使用经验的同学帮助更大，能更清晰理解Vuex的工作流程和原理，使用起来更得心应手。初次接触的同学，可以先参考Vuex[官方文档](http://vuex.vuejs.org/)进行基础概念的学习。

一、框架核心流程

进行源码分析之前，先了解一下官方文档中提供的核心思想图，它也代表着整个Vuex框架的运行流程。

vuex-core

如图示，Vuex为Vue Components建立起了一个完整的生态圈，包括开发中的API调用一环。围绕这个生态圈，简要介绍一下各模块在核心流程中的主要功能：

* Vue Components：Vue组件。HTML页面上，负责接收用户操作等交互行为，执行dispatch方法触发对应action进行回应。
* dispatch：操作行为触发方法，是唯一能执行action的方法。
* actions：操作行为处理模块。负责处理Vue Components接收到的所有交互行为。包含同步/异步操作，支持多个同名方法，按照注册的顺序依次触发。向后台API请求的操作就在这个模块中进行，包括触发其他action以及提交mutation的操作。该模块提供了Promise的封装，以支持action的链式触发。
* commit：状态改变提交操作方法。对mutation进行提交，是唯一能执行mutation的方法。
* mutations：状态改变操作方法。是Vuex修改state的唯一推荐方法，其他修改方式在严格模式下将会报错。该方法只能进行同步操作，且方法名只能全局唯一。操作之中会有一些hook暴露出来，以进行state的监控等。
* state：页面状态管理容器对象。集中存储Vue components中data对象的零散数据，全局唯一，以进行统一的状态管理。页面显示所需的数据从该对象中进行读取，利用Vue的细粒度数据响应机制来进行高效的状态更新。
* getters：state对象读取方法。图中没有单独列出该模块，应该被包含在了render中，Vue Components通过该方法读取全局state对象。

Vue组件接收交互行为，调用dispatch方法触发action相关处理，若页面状态需要改变，则调用commit方法提交mutation修改state，通过getters获取到state新值，重新渲染Vue Components，界面随之更新。

二、目录结构介绍

打开Vuex项目，看下源码目录结构。



dir\_structure

Vuex提供了非常强大的状态管理功能，源码代码量却不多，目录结构划分也很清晰。先大体介绍下各个目录文件的功能： \* module：提供module对象与module对象树的创建功能； \* plugins：提供开发辅助插件，如“时光穿梭”功能，state修改的日志记录功能等； \* helpers.js：提供action、mutations以及getters的查找API； \* index.js：是源码主入口文件，提供store的各module构建安装； \* mixin.js：提供了store在Vue实例上的装载注入； \* util.js：提供了工具方法如find、deepCopy、forEachValue以及assert等方法。

三、初始化装载与注入

了解大概的目录及对应功能后，下面开始进行源码分析。[index.js](https://github.com/vuejs/vuex/blob/dev/src/index.js)中包含了所有的核心代码，从该文件入手进行分析。

3.1 装载实例

先看个简单的例子：

*/\*\**

*\* store.js文件*

*\* 创建store对象，配置state、action、mutation以及getter*

*\**

*\*\*/*

**import** Vue **from** 'vue'

**import** Vuex **from** 'vuex'

*// install Vuex框架*

Vue.use(Vuex)

*// 创建并导出store对象。为了方便，不配置任何参数*

**export** **default** **new** Vuex.Store()

store.js文件中，加载Vuex框架，创建并导出一个空配置的store对象实例。

*/\*\**

*\* vue-index.js文件*

*\**

*\**

*\*\*/*

**import** Vue **from** 'vue'

**import** App **from** './../pages/app.vue'

**import** store **from** './store.js'

**new** Vue({

el: '#root',

router,

store,

render: h => h(App)

})

然后在index.js中，正常初始化一个页面根级别的Vue组件，传入这个自定义的store对象。

如**问题1**所述，以上实例除了Vue的初始化代码，只是多了一个store对象的传入。一起看下源码中的实现方式。

3.2 装载分析

index.js文件代码执行开头，定义局部 Vue 变量，用于判断是否已经装载和减少全局作用域查找。

**let** Vue

然后判断若处于浏览器环境下且加载过Vue，则执行install方法。

*// auto install in dist mode*

**if** (**typeof** window !== 'undefined' && window.Vue) {

install(window.Vue)

}

install方法将Vuex装载到Vue对象上，Vue.use(Vuex) 也是通过它执行，先看下Vue.use方法实现：

**function** (plugin: Function | Object) {

*/\* istanbul ignore if \*/*

**if** (plugin.installed) {

**return**

}

*// additional parameters*

**const** args = toArray(arguments, 1)

args.unshift(**this**)

**if** (**typeof** plugin.install === 'function') {

*// 实际执行插件的install方法*

plugin.install.apply(plugin, args)

} **else** {

plugin.apply(null, args)

}

plugin.installed = true

**return** **this**

}

若是首次加载，将局部Vue变量赋值为全局的Vue对象，并执行applyMixin方法，install实现如下：

**function** **install** (\_Vue) {

**if** (Vue) {

console.error(

'[vuex] already installed. Vue.use(Vuex) should be called only once.'

)

**return**

}

Vue = \_Vue

applyMixin(Vue)

}

来看下applyMixin方法内部代码。如果是2.x.x以上版本，可以使用 hook 的形式进行注入，或使用封装并替换Vue对象原型的\_init方法，实现注入。

**export** **default** **function** (Vue) {

**const** version = Number(Vue.version.split('.')[0])

**if** (version >= 2) {

**const** usesInit = Vue.config.\_lifecycleHooks.indexOf('init') > -1

Vue.mixin(usesInit ? { init: vuexInit } : { beforeCreate: vuexInit })

} **else** {

*// override init and inject vuex init procedure*

*// for 1.x backwards compatibility.*

**const** \_init = Vue.prototype.\_init

Vue.prototype.\_init = **function** (options = {}) {

options.init = options.init

? [vuexInit].concat(options.init)

: vuexInit

\_init.call(**this**, options)

}

}

具体实现：将初始化Vue根组件时传入的store设置到this对象的$store属性上，子组件从其父组件引用$store属性，层层嵌套进行设置。在任意组件中执行 this.$store 都能找到装载的那个store对象，vuexInit方法实现如下：

**function** **vuexInit** () {

**const** options = **this**.$options

*// store injection*

**if** (options.store) {

**this**.$store = options.store

} **else** **if** (options.parent && options.parent.$store) {

**this**.$store = options.parent.$store

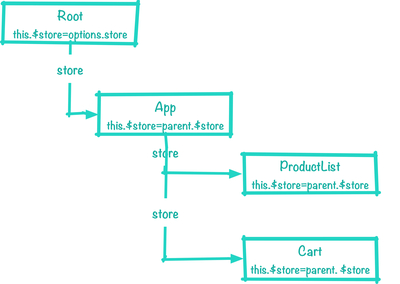
}

}

看个图例理解下store的传递。

页面Vue结构图：

cart\_vue\_structure

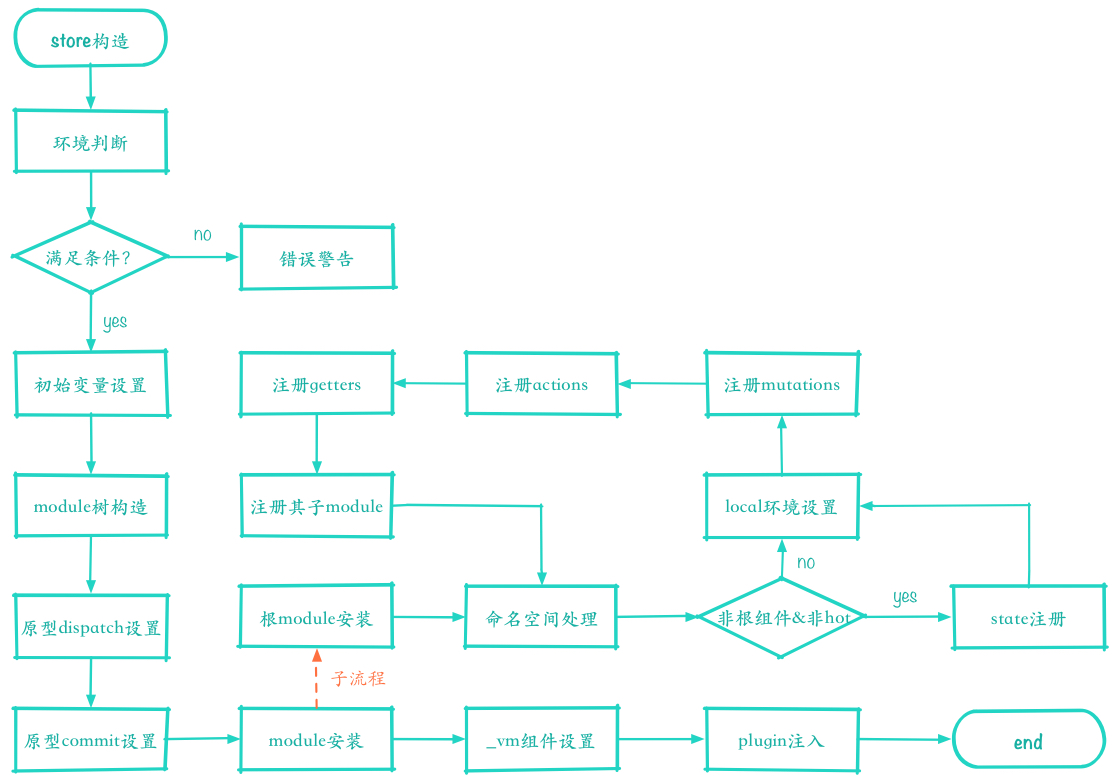
对应store流向：

cart\_vue\_structure

四、store对象构造

上面对Vuex框架的装载以及注入自定义store对象进行分析，解决了**问题1**。接下来详细分析store对象的内部功能和具体实现，来解答 **为什么actions、getters、mutations中能从arguments[0]中拿到store的相关数据?** 等问题。

store对象实现逻辑比较复杂，先看下构造方法的整体逻辑流程来帮助后面的理解：



cart\_vue\_structure

4.1 环境判断

开始分析store的构造函数，分小节逐函数逐行的分析其功能。

**constructor** (options = {}) {

assert(Vue, `must call Vue.use(Vuex) before creating a store instance.`)

assert(**typeof** Promise !== 'undefined', `vuex requires a Promise polyfill in this browser.`)

在store构造函数中执行环境判断，以下都是Vuex工作的必要条件：  
1. 已经执行安装函数进行装载；  
2. 支持Promise语法。

assert函数是一个简单的断言函数的实现，一行代码即可实现。

**function** **assert** (condition, msg) {

**if** (!condition) **throw** **new** Error(`[vuex] ${msg}`)

}

4.2 数据初始化、module树构造

环境判断后，根据new构造传入的options或默认值，初始化内部数据。

**const** {

state = {},

plugins = [],

strict = false

} = options

*// store internal state*

**this**.\_committing = false *// 是否在进行提交状态标识*

**this**.\_actions = Object.create(null) *// acitons操作对象*

**this**.\_mutations = Object.create(null) *// mutations操作对象*

**this**.\_wrappedGetters = Object.create(null) *// 封装后的getters集合对象*

**this**.\_modules = **new** ModuleCollection(options) *// Vuex支持store分模块传入，存储分析后的modules*

**this**.\_modulesNamespaceMap = Object.create(null) *// 模块命名空间map*

**this**.\_subscribers = [] *// 订阅函数集合，Vuex提供了subscribe功能*

**this**.\_watcherVM = **new** Vue() *// Vue组件用于watch监视变化*

调用 new Vuex.store(options) 时传入的options对象，用于构造ModuleCollection类，下面看看其功能。

**constructor** (rawRootModule) {

*// register root module (Vuex.Store options)*

**this**.root = **new** Module(rawRootModule, false)

*// register all nested modules*

**if** (rawRootModule.modules) {

forEachValue(rawRootModule.modules, (rawModule, key) => {

**this**.register([key], rawModule, false)

})

}

ModuleCollection主要将传入的options对象整个构造为一个module对象，并循环调用 this.register([key], rawModule, false) 为其中的modules属性进行模块注册，使其都成为module对象，最后options对象被构造成一个完整的组件树。ModuleCollection类还提供了modules的更替功能，详细实现可以查看源文件[module-collection.js](https://github.com/vuejs/vuex/blob/dev/src/module/module-collection.js)。

4.3 dispatch与commit设置

继续回到store的构造函数代码。

*// bind commit and dispatch to self*

**const** store = **this**

**const** { dispatch, commit } = **this**

**this**.dispatch = **function** **boundDispatch** (type, payload) {

**return** dispatch.call(store, type, payload)

}

**this**.commit = **function** **boundCommit** (type, payload, options) {

**return** commit.call(store, type, payload, options)

}

封装替换原型中的dispatch和commit方法，将this指向当前store对象。dispatch和commit方法具体实现如下：

dispatch (\_type, \_payload) {

*// check object-style dispatch*

**const** {

type,

payload

} = unifyObjectStyle(\_type, \_payload) *// 配置参数处理*

*// 当前type下所有action处理函数集合*

**const** entry = **this**.\_actions[type]

**if** (!entry) {

console.error(`[vuex] unknown action type: ${type}`)

**return**

}

**return** entry.length > 1

? Promise.all(entry.map(handler => handler(payload)))

: entry[0](payload)

}

前面提到，dispatch的功能是触发并传递一些参数（payload）给对应type的action。因为其支持2种调用方法，所以在dispatch中，先进行参数的适配处理，然后判断action type是否存在，若存在就逐个执行（注：上面代码中的this.\_actions[type] 以及 下面的 this.\_mutations[type] 均是处理过的函数集合，具体内容留到后面进行分析）。

commit方法和dispatch相比虽然都是触发type，但是对应的处理却相对复杂，代码如下。

commit (\_type, \_payload, \_options) {

*// check object-style commit*

**const** {

type,

payload,

options

} = unifyObjectStyle(\_type, \_payload, \_options)

**const** mutation = { type, payload }

**const** entry = **this**.\_mutations[type]

**if** (!entry) {

console.error(`[vuex] unknown mutation type: ${type}`)

**return**

}

*// 专用修改state方法，其他修改state方法均是非法修改*

**this**.\_withCommit(() => {

entry.forEach(**function** **commitIterator** (handler) {

handler(payload)

})

})

*// 订阅者函数遍历执行，传入当前的mutation对象和当前的state*

**this**.\_subscribers.forEach(sub => sub(mutation, **this**.state))

**if** (options && options.silent) {

console.warn(

`[vuex] mutation type: ${type}. Silent option has been removed. ` +

'Use the filter functionality in the vue-devtools'

)

}

}

该方法同样支持2种调用方法。先进行参数适配，判断触发mutation type，利用\_withCommit方法执行本次批量触发mutation处理函数，并传入payload参数。执行完成后，通知所有\_subscribers（订阅函数）本次操作的mutation对象以及当前的state状态，如果传入了已经移除的silent选项则进行提示警告。

4.4 state修改方法

\_withCommit是一个代理方法，所有触发mutation的进行state修改的操作都经过它，由此来统一管理监控state状态的修改。实现代码如下。

\_withCommit (fn) {

*// 保存之前的提交状态*

**const** committing = **this**.\_committing

*// 进行本次提交，若不设置为true，直接修改state，strict模式下，Vuex将会产生非法修改state的警告*

**this**.\_committing = true

*// 执行state的修改操作*

fn()

*// 修改完成，还原本次修改之前的状态*

**this**.\_committing = committing

}

缓存执行时的committing状态将当前状态设置为true后进行本次提交操作，待操作完毕后，将committing状态还原为之前的状态。

4.5 module安装

绑定dispatch和commit方法之后，进行严格模式的设置，以及模块的安装（installModule）。由于占用资源较多影响页面性能，严格模式建议只在开发模式开启，上线后需要关闭。

*// strict mode*

**this**.strict = strict

*// init root module.*

*// this also recursively registers all sub-modules*

*// and collects all module getters inside this.\_wrappedGetters*

installModule(**this**, state, [], **this**.\_modules.root)

4.5.1 初始化rootState

上述代码的备注中，提到installModule方法初始化组件树根组件、注册所有子组件，并将其中所有的getters存储到this.\_wrappedGetters属性中，让我们看看其中的代码实现。

**function** **installModule** (store, rootState, path, module, hot) {

**const** isRoot = !path.length

**const** namespace = store.\_modules.getNamespace(path)

*// register in namespace map*

**if** (namespace) {

store.\_modulesNamespaceMap[namespace] = module

}

*// 非根组件设置 state 方法*

**if** (!isRoot && !hot) {

**const** parentState = getNestedState(rootState, path.slice(0, -1))

**const** moduleName = path[path.length - 1]

store.\_withCommit(() => {

Vue.set(parentState, moduleName, module.state)

})

}

······

判断是否是根目录，以及是否设置了命名空间，若存在则在namespace中进行module的存储，在不是根组件且不是 hot 条件的情况下，通过getNestedState方法拿到该module父级的state，拿到其所在的 moduleName ，调用 Vue.set(parentState, moduleName, module.state) 方法将其state设置到父级state对象的moduleName属性中，由此实现该模块的state注册（首次执行这里，因为是根目录注册，所以并不会执行该条件中的方法）。getNestedState方法代码很简单，分析path拿到state，如下。

**function** **getNestedState** (state, path) {

**return** path.length

? path.reduce((state, key) => state[key], state)

: state

}

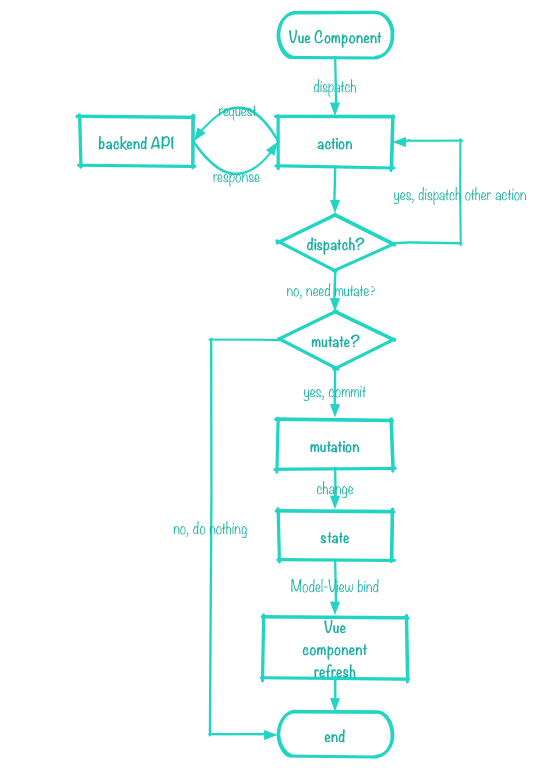
4.5.2 module上下文环境设置

**const** local = module.context = makeLocalContext(store, namespace, path)

命名空间和根目录条件判断完毕后，接下来定义local变量和module.context的值，执行makeLocalContext方法，为该module设置局部的 dispatch、commit方法以及getters和state（由于namespace的存在需要做兼容处理）。

4.5.3 mutations、actions以及getters注册

定义local环境后，循环注册我们在options中配置的action以及mutation等。逐个分析各注册函数之前，先看下模块间的逻辑关系流程图：



complete\_flow

下面分析代码逻辑：

*// 注册对应模块的mutation，供state修改使用*

module.forEachMutation((mutation, key) => {

**const** namespacedType = namespace + key

registerMutation(store, namespacedType, mutation, local)

})

*// 注册对应模块的action，供数据操作、提交mutation等异步操作使用*

module.forEachAction((action, key) => {

**const** namespacedType = namespace + key

registerAction(store, namespacedType, action, local)

})

*// 注册对应模块的getters，供state读取使用*

module.forEachGetter((getter, key) => {

**const** namespacedType = namespace + key

registerGetter(store, namespacedType, getter, local)

})

registerMutation方法中，获取store中的对应mutation type的处理函数集合，将新的处理函数push进去。这里将我们设置在mutations type上对应的 handler 进行了封装，给原函数传入了state。在执行 commit('xxx', payload) 的时候，type为 xxx 的mutation的所有handler都会接收到state以及payload，这就是在handler里面拿到state的原因。

**function** **registerMutation** (store, type, handler, local) {

*// 取出对应type的mutations-handler集合*

**const** entry = store.\_mutations[type] || (store.\_mutations[type] = [])

*// commit实际调用的不是我们传入的handler，而是经过封装的*

entry.push(**function** **wrappedMutationHandler** (payload) {

*// 调用handler并将state传入*

handler(local.state, payload)

})

}

action和getter的注册也是同理的，看一下代码（注：前面提到的 this.actions 以及 this.mutations在此处进行设置）。

**function** **registerAction** (store, type, handler, local) {

*// 取出对应type的actions-handler集合*

**const** entry = store.\_actions[type] || (store.\_actions[type] = [])

*// 存储新的封装过的action-handler*

entry.push(**function** **wrappedActionHandler** (payload, cb) {

*// 传入 state 等对象供我们原action-handler使用*

**let** res = handler({

dispatch: local.dispatch,

commit: local.commit,

getters: local.getters,

state: local.state,

rootGetters: store.getters,

rootState: store.state

}, payload, cb)

*// action需要支持promise进行链式调用，这里进行兼容处理*

**if** (!isPromise(res)) {

res = Promise.resolve(res)

}

**if** (store.\_devtoolHook) {

**return** res.catch(err => {

store.\_devtoolHook.emit('vuex:error', err)

**throw** err

})

} **else** {

**return** res

}

})

}

**function** **registerGetter** (store, type, rawGetter, local) {

*// getters只允许存在一个处理函数，若重复需要报错*

**if** (store.\_wrappedGetters[type]) {

console.error(`[vuex] duplicate getter key: ${type}`)

**return**

}

*// 存储封装过的getters处理函数*

store.\_wrappedGetters[type] = **function** **wrappedGetter** (store) {

*// 为原getters传入对应状态*

**return** rawGetter(

local.state, *// local state*

local.getters, *// local getters*

store.state, *// root state*

store.getters *// root getters*

)

}

}

action handler比mutation handler以及getter wrapper多拿到dispatch和commit操作方法，因此action可以进行dispatch action和commit mutation操作。

4.5.4 子module安装

注册完了根组件的actions、mutations以及getters后，递归调用自身，为子组件注册其state，actions、mutations以及getters等。

module.forEachChild((child, key) => {

installModule(store, rootState, path.concat(key), child, hot)

})

4.5.5 实例结合

前面介绍了dispatch和commit方法以及actions等的实现，下面结合一个官方的[购物车](https://github.com/vuejs/vuex/tree/dev/examples/shopping-cart)实例中的部分代码来加深理解。

Vuex配置代码：

/

\* store-index.js store配置文件

\*

/

import Vue from 'vue'

import Vuex from 'vuex'

import \* as actions from './actions'

import \* as getters from './getters'

import cart from './modules/cart'

import products from './modules/products'

import createLogger from '../../../src/plugins/logger'

Vue.use(Vuex)

const debug = process.env.NODE\_ENV !== 'production'

export default new Vuex.Store({

actions,

getters,

modules: {

cart,

products

},

strict: debug,

plugins: debug ? [createLogger()] : []

})

Vuex组件module中各模块state配置代码部分：

*/\*\**

*\* cart.js*

*\**

*\*\*/*

**const** state = {

added: [],

checkoutStatus: null

}

*/\*\**

*\* products.js*

*\**

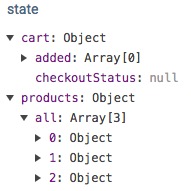
*\*\*/*

**const** state = {

all: []

}

加载上述配置后，页面state结构如下图：



cart\_state

state中的属性配置都是按照option配置中module path的规则来进行的，下面看action的操作实例。

Vuecart组件代码部分：

*/\*\**

*\* Cart.vue 省略template代码，只看script部分*

*\**

*\*\*/*

**export** **default** {

methods: {

*// 购物车中的购买按钮，点击后会触发结算。源码中会调用 dispatch方法*

checkout (products) {

**this**.$store.dispatch('checkout', products)

}

}

}

Vuexcart.js组件action配置代码部分：

**const** actions = {

checkout ({ commit, state }, products) {

**const** savedCartItems = [...state.added] *// 存储添加到购物车的商品*

commit(types.CHECKOUT\_REQUEST) *// 设置提交结算状态*

shop.buyProducts( *// 提交api请求，并传入成功与失败的cb-func*

products,

() => commit(types.CHECKOUT\_SUCCESS), *// 请求返回成功则设置提交成功状态*

() => commit(types.CHECKOUT\_FAILURE, { savedCartItems }) *// 请求返回失败则设置提交失败状态*

)

}

}

Vue组件中点击购买执行当前module的dispatch方法，传入type值为 ‘checkout’，payload值为 ‘products’，在源码中dispatch方法在所有注册过的actions中查找’checkout’的对应执行数组，取出循环执行。执行的是被封装过的被命名为wrappedActionHandler的方法，真正传入的checkout的执行函数在wrappedActionHandler这个方法中被执行，源码如下（注：前面贴过，这里再看一次）：

**function** **wrappedActionHandler** (payload, cb) {

**let** res = handler({

dispatch: local.dispatch,

commit: local.commit,

getters: local.getters,

state: local.state,

rootGetters: store.getters,

rootState: store.state

}, payload, cb)

**if** (!isPromise(res)) {

res = Promise.resolve(res)

}

**if** (store.\_devtoolHook) {

**return** res.catch(err => {

store.\_devtoolHook.emit('vuex:error', err)

**throw** err

})

} **else** {

**return** res

}

}

handler在这里就是传入的checkout函数，其执行需要的commit以及state就是在这里被传入，payload也传入了，在实例中对应接收的参数名为products。commit的执行也是同理的，实例中checkout还进行了一次commit操作，提交一次type值为types.CHECKOUT\_REQUEST的修改，因为mutation名字是唯一的，这里进行了常量形式的调用，防止命名重复，执行跟源码分析中一致，调用 function wrappedMutationHandler (payload) { handler(local.state, payload) } 封装函数来实际调用配置的mutation方法。

看到完源码分析和上面的小实例，应该能理解dispatch action和commit mutation的工作原理了。接着看源码，看看getters是如何实现state实时访问的。

4.6 store.\_vm组件设置

执行完各module的install后，执行resetStoreVM方法，进行store组件的初始化。

*// initialize the store vm, which is responsible for the reactivity*

*// (also registers \_wrappedGetters as computed properties)*

resetStoreVM(**this**, state)

综合前面的分析可以了解到，Vuex其实构建的就是一个名为store的vm组件，所有配置的state、actions、mutations以及getters都是其组件的属性，所有的操作都是对这个vm组件进行的。

一起看下resetStoreVM方法的内部实现。

**function** **resetStoreVM** (store, state) {

**const** oldVm = store.\_vm *// 缓存前vm组件*

*// bind store public getters*

store.getters = {}

**const** wrappedGetters = store.\_wrappedGetters

**const** computed = {}

*// 循环所有处理过的getters，并新建computed对象进行存储，通过Object.defineProperty方法为getters对象建立属性，使得我们通过this.$store.getters.xxxgetter能够访问到该getters*

forEachValue(wrappedGetters, (fn, key) => {

*// use computed to leverage its lazy-caching mechanism*

computed[key] = () => fn(store)

Object.defineProperty(store.getters, key, {

get: () => store.\_vm[key],

enumerable: true *// for local getters*

})

})

*// use a Vue instance to store the state tree*

*// suppress warnings just in case the user has added*

*// some funky global mixins*

**const** silent = Vue.config.silent

*// 暂时将Vue设为静默模式，避免报出用户加载的某些插件触发的警告*

Vue.config.silent = true

*// 设置新的storeVm，将当前初始化的state以及getters作为computed属性（刚刚遍历生成的）*

store.\_vm = **new** Vue({

data: { state },

computed

})

*// 恢复Vue的模式*

Vue.config.silent = silent

*// enable strict mode for new vm*

**if** (store.strict) {

*// 该方法对state执行$watch以禁止从mutation外部修改state*

enableStrictMode(store)

}

*// 若不是初始化过程执行的该方法，将旧的组件state设置为null，强制更新所有监听者(watchers)，待更新生效，DOM更新完成后，执行vm组件的destroy方法进行销毁，减少内存的占用*

**if** (oldVm) {

*// dispatch changes in all subscribed watchers*

*// to force getter re-evaluation.*

store.\_withCommit(() => {

oldVm.state = null

})

Vue.nextTick(() => oldVm.$destroy())

}

}

resetStoreVm方法创建了当前store实例的\_vm组件，至此store就创建完毕了。上面代码涉及到了严格模式的判断，看一下严格模式如何实现的。

**function** **enableStrictMode** (store) {

store.\_vm.$watch('state', () => {

assert(store.\_committing, `Do not mutate vuex store state outside mutation handlers.`)

}, { deep: true, sync: true })

}

很简单的应用，监视state的变化，如果没有通过 this.\_withCommit() 方法进行state修改，则报错。

4.7 plugin注入

最后执行plugin的植入。

plugins.concat(devtoolPlugin).forEach(plugin => plugin(**this**))

devtoolPlugin提供的功能有3个：

*// 1. 触发Vuex组件初始化的hook*

devtoolHook.emit('vuex:init', store)

*// 2. 提供“时空穿梭”功能，即state操作的前进和倒退*

devtoolHook.on('vuex:travel-to-state', targetState => {

store.replaceState(targetState)

})

*// 3. mutation被执行时，触发hook，并提供被触发的mutation函数和当前的state状态*

store.subscribe((mutation, state) => {

devtoolHook.emit('vuex:mutation', mutation, state)

})

源码分析到这里，Vuex框架的实现原理基本都已经分析完毕。

五、总结

最后我们回过来看文章开始提出的5个问题。

1.  **问**：*使用Vuex只需执行*Vue.use(Vuex)*，并在Vue的配置中传入一个store对象的示例，store是如何实现注入的？*

**答**：Vue.use(Vuex) 方法执行的是install方法，它实现了Vue实例对象的init方法封装和注入，使传入的store对象被设置到Vue上下文环境的$store中。因此在Vue Component任意地方都能够通过this.$store访问到该store。

2.  **问**：*state内部支持模块配置和模块嵌套，如何实现的？*

**答**：在store构造方法中有makeLocalContext方法，所有module都会有一个local context，根据配置时的path进行匹配。所以执行如dispatch('submitOrder', payload)这类action时，默认的拿到都是module的local state，如果要访问最外层或者是其他module的state，只能从rootState按照path路径逐步进行访问。

3.  **问**：*在执行dispatch触发action(commit同理)的时候，只需传入(type, payload)，action执行函数中第一个参数store从哪里获取的？*

**答**：store初始化时，所有配置的action和mutation以及getters均被封装过。在执行如dispatch('submitOrder', payload)的时候，actions中type为submitOrder的所有处理方法都是被封装后的，其第一个参数为当前的store对象，所以能够获取到 { dispatch, commit, state, rootState } 等数据。

4.  **问**：*Vuex如何区分state是外部直接修改，还是通过mutation方法修改的？*

**答**：Vuex中修改state的唯一渠道就是执行 commit('xx', payload) 方法，其底层通过执行 this.\_withCommit(fn) 设置\_committing标志变量为true，然后才能修改state，修改完毕还需要还原\_committing变量。外部修改虽然能够直接修改state，但是并没有修改\_committing标志位，所以只要watch一下state，state change时判断是否\_committing值为true，即可判断修改的合法性。

5.  **问**：*调试时的”时空穿梭”功能是如何实现的？*

**答**：devtoolPlugin中提供了此功能。因为dev模式下所有的state change都会被记录下来，’时空穿梭’ 功能其实就是将当前的state替换为记录中某个时刻的state状态，利用 store.replaceState(targetState) 方法将执行this.\_vm.state = state 实现。

源码中还有一些工具函数类似registerModule、unregisterModule、hotUpdate、watch以及subscribe等，如有兴趣可以打开源码看看，这里不再细述。

六、作者简介

明裔，美团外卖高级前端研发工程师，2014年加入美团外卖，负责Web主站开发。先后参与了外卖B端、C端、配送等全业务线系统开发后，目前主要负责商家券活动系统。

**最后，附上一条硬广，美团外卖长期诚聘高级前端工程师/前端技术专家，欢迎发送简历至：mabingbing02#meituan.com。**

[前端](https://tech.meituan.com/tags/%E5%89%8D%E7%AB%AF.html), [大零售](https://tech.meituan.com/tags/%E5%A4%A7%E9%9B%B6%E5%94%AE.html), [Vuex](https://tech.meituan.com/tags/vuex.html), [Vue](https://tech.meituan.com/tags/vue.html), [美团外卖](https://tech.meituan.com/tags/%E7%BE%8E%E5%9B%A2%E5%A4%96%E5%8D%96.html), [框架源码分析](https://tech.meituan.com/tags/%E6%A1%86%E6%9E%B6%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90.html)

#看看其他

[前一篇: 基于rsync的文件增量同步方案](https://tech.meituan.com/2017/04/27/incre-sync-use-rsync.html)[后一篇: 美团DSP广告策略实践](https://tech.meituan.com/2017/05/05/mt-dsp.html)

#一起聊聊

如发现文章有错误、对内容有疑问，都可以关注美团技术团队微信公众号（meituantech），在后台给我们留言。



我们每周会挑选出一位热心小伙伴，送上一份精美的小礼品。快来扫码关注我们吧！

一行代码，亿万生活。

* [网站首页](https://tech.meituan.com/)
* [文章存档](https://tech.meituan.com/archives)
* [关于我们](https://tech.meituan.com/about)