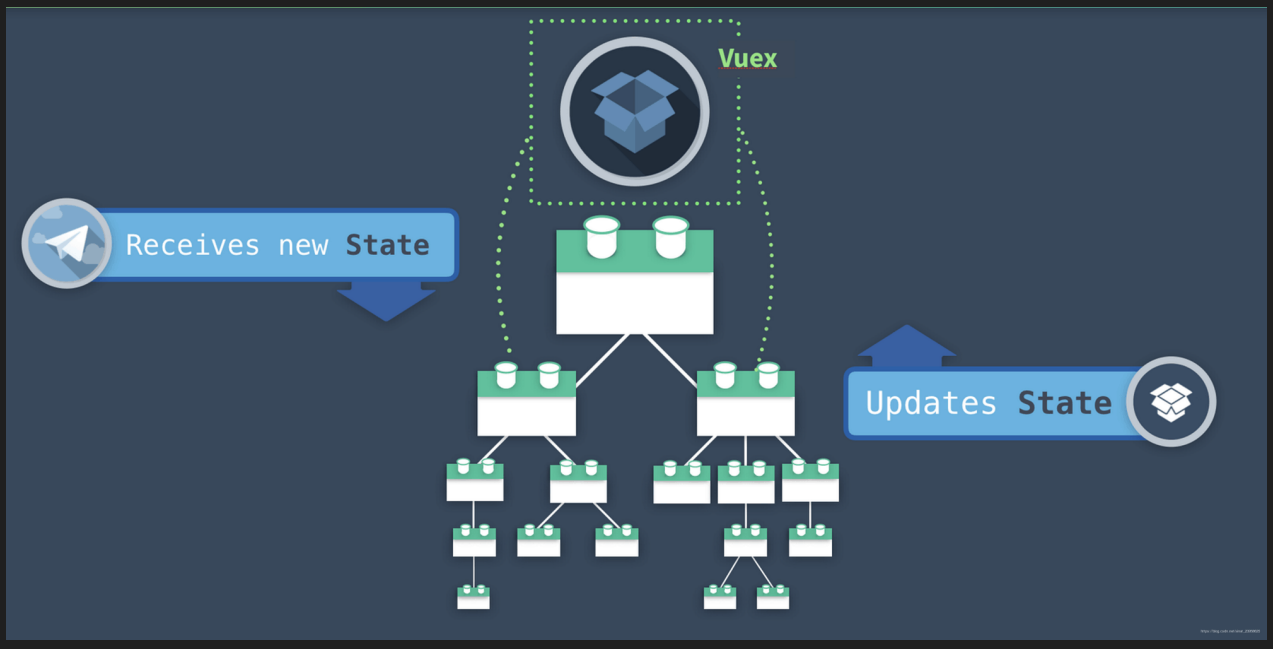
Vuex 状态管理学习笔记

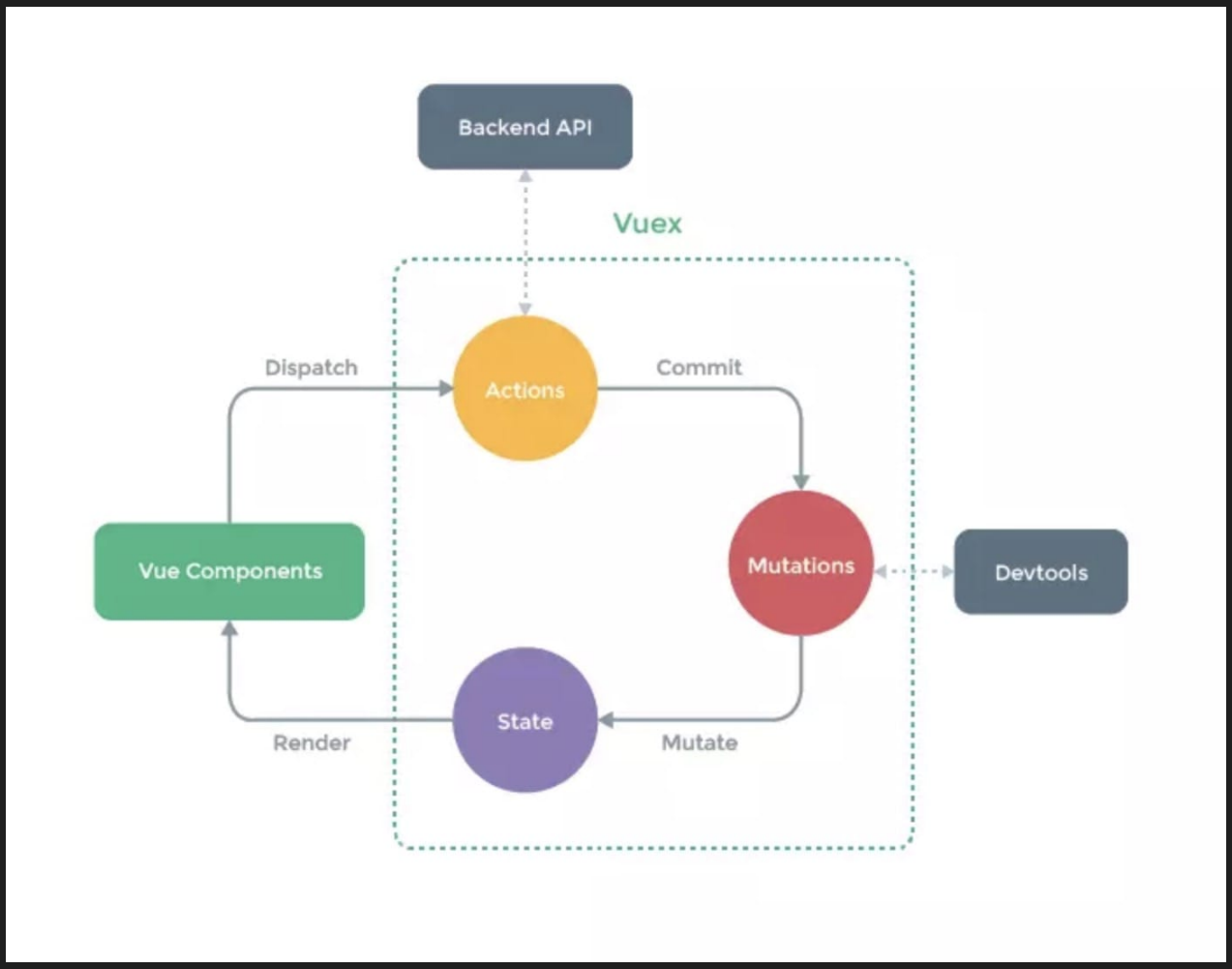


我们在使用 Vue.js 开发复杂的应用时，经常会遇到多个组件共享同一个状态，亦或是多个组件会去更新同一个状态，在应用代码量较少的时候，我们可以组件间通信（父传子 `v-bind` prop 传递和 子传父 `v-on` 自定义事件）去维护修改数据，或者是通过事件总线来进行数据的传递以及修改。

但是当应用逐渐庞大以后，代码就会变得难以维护，从父组件开始通过 prop 传递多层嵌套的数据由于层级过深而显得异常脆弱、数据来源也会不清晰，而事件总线也会因为组件的增多、代码量的增大而显得交互错综复杂，难以捋清其中的传递关系。

正是因为Vuex使用了 Vue.js 内部的 **“响应式机制”**，所以 Vuex 是一个专门为 Vue.js 设计并与之高度契合的框架（优点是更加简洁高效，当然也只能跟 Vue.js 搭配使用）

> 每一个 Vuex 应用的核心就是 `store` （仓库）。 `store` 基本上就是一个容器，它包含着你的应用中大部分的**状态** `state` 。Vuex 和单纯的全局对象有以下两点不同：
>
> 1. Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。
> 2. 你不能直接改变 `store` 中的状态。改变 `store` 中的状态的唯一途径就是显式地 **提交** `commit mutation` 。
>
>   这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用。



基础使用

```
yarn add vuex
```

通过上述命令安装 Vuex 后，我们在 Vue 项目的 `src` 目录下添加一个 `store` 文件夹（如果使用了 VueCLI 可以自动初始化一个含有 Vuex 的项目，但我这里还是选择呈现一个手动的流程），然后可以新建一个 `index.js` 作为总领出口。

```
import Vue from 'vue'
import Vuex from 'vuex'

const store = new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment (state) {
      state.count++
    }
  },
  actions: {
    increment (context) {
      context.commit('increment')
```

```
    }
  }
})
```

Getter

类似于 Vue 中的 **计算属性**（可以认为是 `store` 的计算属性），`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。

Getter 方法接受 `state` 作为其第一个参数：

```
const store = new Vuex.Store({
  state: {
    todos: [
      { id: 1, text: 'aaa', done: true },
      { id: 2, text: 'bbb', done: false }
    ]
  },
  getters: {
    doneTodos: state => {
      return state.todos.filter(todo => todo.done)
    }
  }
})
```

Mutation vs. Action

`mutation` 接受两个参数 `state` 和 `payload` (载荷)，`mutation` 必须是同步函数！

通过执行回调函数修改 `state` 的状态，可以向 `store.commit` 传入额外的参数 `payload`，`payload` 可以是一个对象，这样可以包含多个字段并且记录的 mutation 会更易读。

而 `action` 相对于 `mutation`，有以下不同：

- `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**。
- Action 函数接受一个与 `store` 实例具有相同方法和属性的 `context` 对象，除了使用 `context.commit` 提交 `mutations`，也可以使用 `context.getter` 和 `context.state` 来获取 `getter` 和 `state`。有一点需要注意点是，`context` 并不是 `state` 实例本身。

  在使用 `commit` 时，常使用参数结构的方式来简化代码：

```
actions: {
  increment ({ commit }) {
    commit('increment')
  }
}
```

「分发」语法糖：

只要遵守 mutations 更改原则使用上没什么特别难理解的，但在组件中使用时，我们难免地会嫌弃 `this.$store.state.xxx` 的写法太过冗长，所以可以使用 `mapXXX` 一族的分发方法来快速的将状态仓库中的状态导入当前组件：

```
import { mapState } from 'vuex'

export default {
  data() {
    return {
      ...mapState(['count']) // 这样我们就可以通过 this.count 来访问
this.$store.state.count 了
    }
  }
}
```

对于计算属性也有特别的支持：

```
// 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
    count1: state => state.count,

    // 传字符串参数 'count' 等同于 `state => state.count`
    count2: 'count',

    // 这里是利用 store 当中的状态和组件内部的状态相结合来组成一个计算属性
    // 为了能够使用当前组件 `this` 获取状态，必须使用常规函数而不能是箭头函数
    countPlusLocalState (state) {
      return state.count + this.localCount
    }
  })
}
```

而为什么推荐使用 `action` 而不是直接分发 `mutation` 呐？因为 `mutation` 必须执行同步函数，而在 `action` 中可以执行异步函数。与 `mutation` 类似， `actions` 支持同样的载荷 `payload` 方式和对象方式进行分发：

```
// store 定义中:
```

```
actions: {
  incrementAsync ({ commit }) {
    setTimeout(() => {
      commit('increment')
    }, 1000)
  }
}

// 以载荷形式分发
store.dispatch('increaseMore', {
  amount: 10
})
// 或 以对象形式分发
store.dispatch({
  type: 'increaseMore',
  amount: 10
})
```

在组件中可以使用以下方式分发 action：

```
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')
    })
  }
}
```

源码浅析：

从流程上，首先肯定是万年不变的 Vue 2.x `.install` 安装方法，同样和 VueRouter 一样他们都有防止自己被重复注册的机制：

```
/*暴露给外部的插件install方法，供Vue.use调用安装插件*/
export function install (_Vue) {
  if (Vue) {
    /* 避免重复安装（Vue.use 内部也会检测一次是否重复安装同一个插件）*/
    if (process.env.NODE_ENV !== 'production') {
      console.error(
        '[vuex] already installed. Vue.use(Vuex) should be called only once.'
      )
    }
```

```
    return
  }
  /* 保存Vue，同时用于检测是否重复安装 */
  Vue = _Vue
  /* 将 vuexInit 混淆进 Vue 的 beforeCreate(Vue2.0) 或 _init 方法(Vue1.0) */
  applyMixin(Vue)
}
```

那么 mixin 进 Vue 实例中的 `vuexInit` 方法究竟是什么呢?

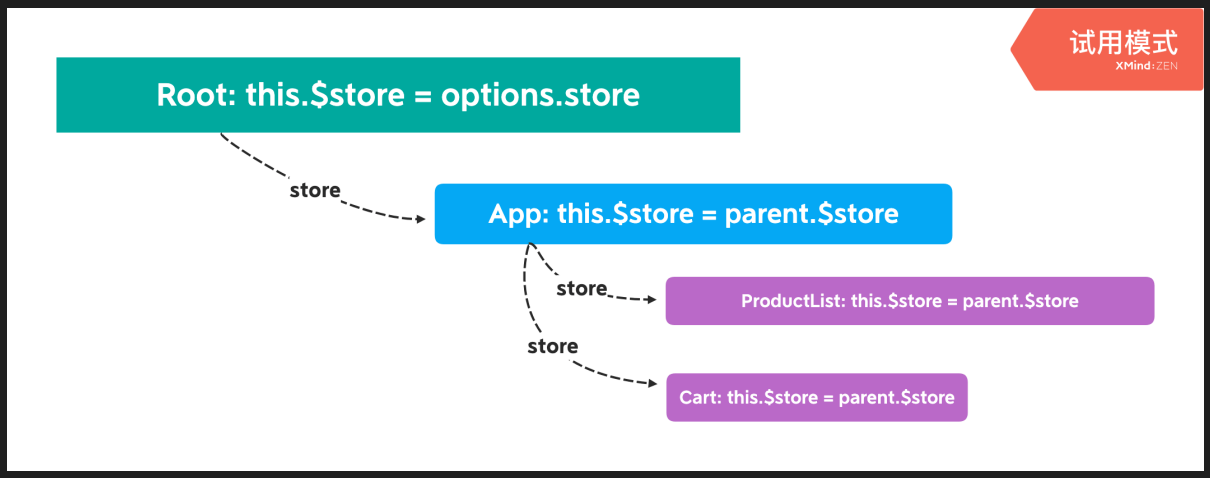
```
/* Vuex的 init 钩子，会存入每一个 Vue 实例等钩子列表 */
function vuexInit () {
  const options = this.$options
  // store injection
  if (options.store) {
    /*存在 store 其实代表的就是 Root 节点，直接执行 store（是 function 时）或者使用 store（非function）*/
    this.$store = typeof options.store === 'function'
      ? options.store()
      : options.store
  } else if (options.parent && options.parent.$store) {
    /* 子组件直接从父组件中获取$store，这样就保证了所有组件都公用了全局的同一份 store */
    this.$store = options.parent.$store
  }
}
```

这样一来，所有的组件都获取到了 **同一份内存地址** 的 Store 实例，于是我们可以在每一个组件中通过 `this.$store` 愉快地访问全局的 Store 实例了。

看个图例理解下store的传递。

- 页面Vue结构图：

- 对应 store 流向：

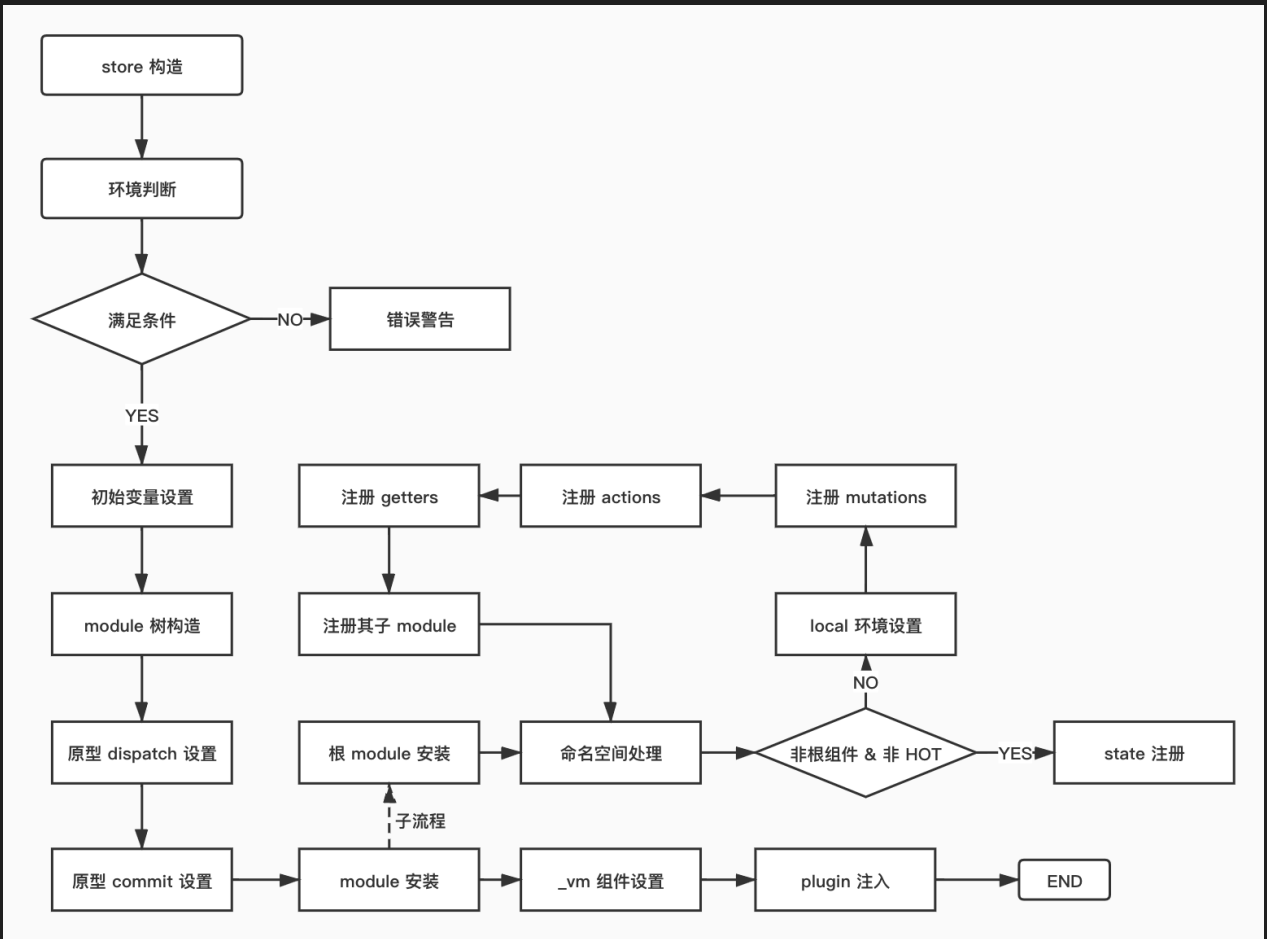


Vuex.Store 构造函数如下：

```
constructor (options = {}) {
    /*
      在浏览器环境下，如果插件还未安装（!Vue即判断是否未安装），则它会自动安装。
      它允许用户在某些情况下避免自动安装。
    */
    if (!Vue && typeof window !== 'undefined' && window.Vue) {
      install(window.Vue)
    }

    if (process.env.NODE_ENV !== 'production') {
      assert(Vue, `must call Vue.use(Vuex) before creating a store instance.`)
      assert(typeof Promise !== 'undefined', `vuex requires a Promise polyfill in this
browser.`)
      assert(this instanceof Store, `Store must be called with the new operator.`)
    }

    const {
      /* 一个插件的数组：即包含应用在 store 上的插件方法。这些插件直接接收 store 作为唯一参数，可以
监听 mutation（用于外部地数据持久化、记录或调试）或者提交 mutation （用于内部数据，例如 websocket
或 某些观察者）*/
      plugins = [],
      /* 使 Vuex store 进入严格模式，在严格模式下，任何 mutation 处理函数以外修改 Vuex state 都
会抛出错误。*/
      strict = false
    } = options

    /*从 option 中取出 state，如果 state 是 function 则执行，最终得到一个对象 */
    let {
      state = {}
    } = options
    if (typeof state === 'function') {
      state = state()
    }

    // store 的内部状态（一些私有变量，尽量不在用户空间使用的）
    /* 用来判断严格模式下是否是用mutation修改state的 */
    this._committing = false
    /* 存放 action */
    this._actions = Object.create(null)
    /* 存放 mutation */
    this._mutations = Object.create(null)
    /* 存放 getter */
    this._wrappedGetters = Object.create(null)
    /* module 收集器 */
    this._modules = new ModuleCollection(options)
    /* 根据 namespace 存放 module */
    this._modulesNamespaceMap = Object.create(null)
    /* 存放订阅者 */
    this._subscribers = []
    /* 用以实现 Watch 的 Vue 实例 */
```

```
    this._watcherVM = new Vue()

    /*将 dispatch 与 commit 调用的 this 绑定为 store 对象本身，否则在组件内部 this.dispatch
时的 this 会指向组件的 vm */
    const store = this
    const { dispatch, commit } = this
    /* 为 dispatch 与 commit 绑定 this（Store实例本身） */
    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)
    }

    /* 严格模式(使 Vuex store 进入严格模式，在严格模式下，任何 mutation 处理函数以外修改 Vuex
state 都会抛出错误)*/
    this.strict = strict

    /* 初始化根 module，这也同时递归注册了所有子 modle，收集所有 module 的 getter 到
_wrappedGetters中去，this._modules.root代表根 module 才独有保存的 Module 对象 */
    installModule(this, state, [], this._modules.root)

    /* 通过 vm 重设 store，新建 Vue 对象使用 Vue 内部的响应式实现注册 state 以及 computed */
    resetStoreVM(this, state)

    /* 调用插件 */
    plugins.forEach(plugin => plugin(this))

    /* devtool插件 */
    if (Vue.config.devtools) {
      devtoolPlugin(this)
    }
  }
```

Store 构造后所引发的关系图：



主要 API 与功能分析：

`installModule`

```
/* 初始化module */
function installModule (store, rootState, path, module, hot) {
  /* 是否是根module */
  const isRoot = !path.length
  /* 获取module的namespace */
  const namespace = store._modules.getNamespace(path)

  /* 如果有namespace则在 _modulesNamespaceMap 中注册 */
  if (module.namespaced) {
    store._modulesNamespaceMap[namespace] = module
  }

  // set state
  if (!isRoot && !hot) {
    /* 获取父级的 state */
    const parentState = getNestedState(rootState, path.slice(0, -1))
    /* module 的 name */
    const moduleName = path[path.length - 1]
    store.`_withCommit`(() => {
      /* 将子 module 设置称响应式的 */
      Vue.set(parentState, moduleName, module.state)
```

```
    })
  }

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

  /* 遍历注册 mutation */
  module.forEachMutation((mutation, key) => {
    const namespacedType = namespace + key
    registerMutation(store, namespacedType, mutation, local)
  })

  /* 遍历注册 action */
  module.forEachAction((action, key) => {
    const namespacedType = namespace + key
    registerAction(store, namespacedType, action, local)
  })

  /* 遍历注册 getter */
  module.forEachGetter((getter, key) => {
    const namespacedType = namespace + key
    registerGetter(store, namespacedType, getter, local)
  })

  /* 递归安装 mudule */
  module.forEachChild((child, key) => {
    installModule(store, rootState, path.concat(key), child, hot)
  })
}
```

`installModule` 的作用主要是用为 `module` 加上 `namespace` 名字空间（如果有）后，注册 `mutation` 、 `action` 以及 `getter` ，同时递归安装所有子 `module` 。

`resetStoreVM`

```
/* 通过 vm 重设 store，新建 Vue 对象使用 Vue 内部的响应式机制实现注册 state 以及 computed */
function resetStoreVM (store, state, hot) {
  /* 存放之前的 vm 对象 */
  const oldVm = store._vm

  // 绑定 store 公有的 getters
  store.getters = {}
  const wrappedGetters = store._wrappedGetters
  const computed = {}

  /* 通过 Object.defineProperty 为每一个 getter 方法设置 get 方法，比如获取
this.$store.getters.test 的时候获取的是 store._vm.test，也就是 Vue 对象的 computed 属性 */
  forEachValue(wrappedGetters, (fn, key) => {
    // 使用当前域中的 computed 对象来实现懒计算
    computed[key] = () => fn(store)
    Object.defineProperty(store.getters, key, {
```

```
      get: () => store._vm[key],
      enumerable: true // for local getters
    })
  })

  // 使用一个 Vue 实例来存储状态树
  const silent = Vue.config.silent
  /* Vue.config.silent 暂时设置为 true 的目的是在 new 一个 Vue 实例的过程中不会报出一切警告 */
  Vue.config.silent = true
  /*  这里 new 了一个 Vue 对象，运用Vue内部的响应式实现注册 state 以及 computed */
  store._vm = new Vue({
    data: {
      $$state: state
    },
    computed
  })
  Vue.config.silent = silent

  /* 使能严格模式，保证修改 store 只能通过mutation */
  if (store.strict) {
    enableStrictMode(store)
  }

  if (oldVm) {
    /* 解除旧 vm 的 state 的引用，以及销毁旧的 Vue 对象 */
    if (hot) {
      // 发送所有已经订阅了的侦听器上的响应回调
      // 为热重载，强制执行重新计算
      store._withCommit(() => {
        oldVm._data.$$state = null
      })
    }
    Vue.nextTick(() => oldVm.$destroy())
  }
}
```

`resetStoreVM` 首先会遍历 `wrappedGetters`，使用 `Object.defineProperty` 方法为每一个 `getter` 绑定上 `get` 方法，这样我们就可以在组件里访问 `this.$store.getter.test` 就等同于访问 `store._vm.test`。

之后 Vuex 采用了 `new` 一个 Vue 对象来实现数据的“响应式化”，运用 Vue.js 内部提供的数据双向绑定功能来实现 `store` 的数据与视图的同步更新。

关于严格模式：

Vuex的 `Store` 构造类的 `option` 有一个 `strict` 的参数，可以控制 `Vuex` 执行严格模式，严格模式下，所有修改 `state` 的操作必须通过 `mutation` 实现，否则会抛出错误。

```
/* 使能严格模式 */
function enableStrictMode (store) {
  store._vm.$watch( function () { return this._data.$$state }, () => {
    if (process.env.NODE_ENV !== 'production') {
      /* 检测 store 中的 _committing 的值，如果是 false 代表不是通过 mutation 的方法修改的 */
      assert(store._committing, `Do not mutate vuex store state outside mutation
handlers.`)
    }
  }, { deep: true, sync: true })
}
```

由于 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` 变化时判断是否 `_committing` 值为 `true`，即可判断修改的合法性。

简而言之就是监视 `state` 的变化，如果没有通过 `this._withCommit()` 方法进行 `state` 修改则报错。

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。

`commit`

```
/* 调用 mutation 的 commit 方法 */
commit (_type, _payload, _options) {
  /* 校验参数 */
  const {
    type,
    payload,
    options
  } = unifyObjectStyle(_type, _payload, _options)

  const mutation = { type, payload }
  /* 取出 type 对应的 mutation 的方法 */
  const entry = this._mutations[type]
  if (!entry) {
    if (process.env.NODE_ENV !== 'production') {
      console.error(`[vuex] unknown mutation type: ${type}`)
    }
    return
  }
  /* 执行 mutation 中的所有方法 */
  this._withCommit(() => {
    entry.forEach(function commitIterator (handler) {
      handler(payload)
    })
  })
  /* 通知所有订阅者 */
  this._subscribers.forEach(sub => sub(mutation, this.state))
```

```
  if (
    process.env.NODE_ENV !== 'production' &&
    options && options.silent
  ) {
    console.warn(
      `[vuex] mutation type: ${type}. Silent option has been removed. ` +
      'Use the filter functionality in the vue-devtools'
    )
  }
}
```

`commit` 方法会根据 `type` 找到并调用 `_mutations` 中的所有 `type` 对应的 `mutation` 方法，所以当没有 `namespace` 的时候， `commit` 方法会触发所有 `module` 中的 `mutation` 方法。再执行完所有的 `mutation` 之后会执行 `_subscribers` 中的所有订阅者。我们来看一下 `_subscribers` 是什么：

其实是 `Store` 给外部提供了一个 `subscribe` 方法，用以注册一个订阅函数，会 `push` 到 `Store` 实例的 `_subscribers` 中，同时返回一个从 `_subscribers` 中注销该订阅者的方法。

```
/* 注册一个订阅函数，返回取消订阅的函数 */
subscribe (fn) {
  const subs = this._subscribers
  if (subs.indexOf(fn) < 0) {
    subs.push(fn)
  }
  return () => {
    const i = subs.indexOf(fn)
    if (i > -1) {
      subs.splice(i, 1)
    }
  }
}
```

在 `commit` 结束以后则会调用这些 `_subscribers` 中的订阅者。

这个订阅者模式提供给外部一个监视 `state` 变化的可能。 `state` 通过 `mutation` 改变时，可以有效捕获这些变化。

`dispatch`

```
/* 调用 action 的 dispatch 方法 */
dispatch (_type, _payload) {
  // 检查 object 形式的 dispatch
  const {
    type,
    payload
  } = unifyObjectStyle(_type, _payload)

  /* actions 中取出 type 对应的 action */
```

```
  const entry = this._actions[type]
  if (!entry) {
    if (process.env.NODE_ENV !== 'production') {
      console.error(`[vuex] unknown action type: ${type}`)
    }
    return
  }

  /* 是数组则包装 Promise 形成一个新的 Promise，只有一个则直接返回第 0 个 */
  return entry.length > 1
    ? Promise.all(entry.map(handler => handler(payload)))
    : entry[0](payload)
}
```

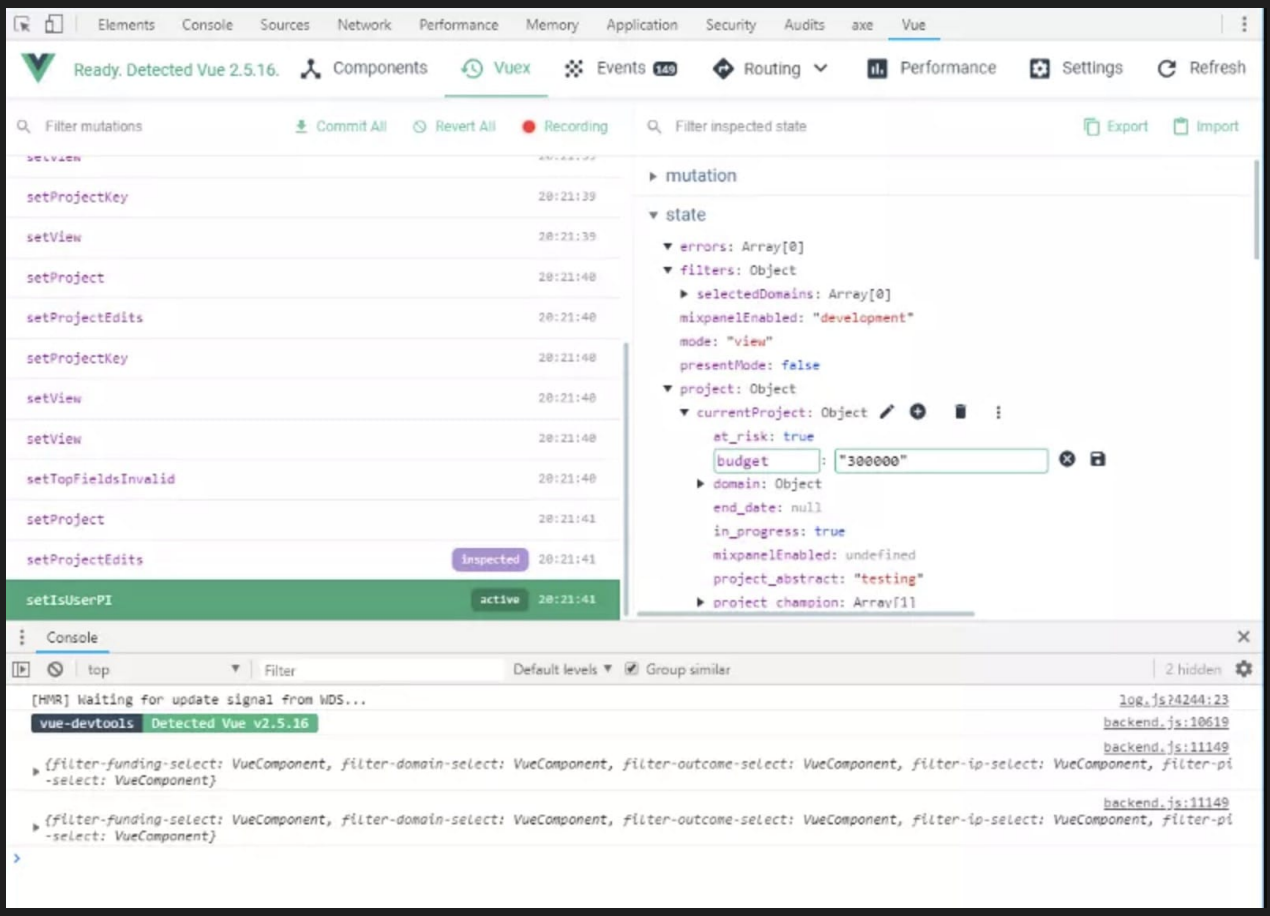
```
/* 遍历注册 action */
function registerAction (store, type, handler, local) {
  /* 取出 type 对应的 action */
  const entry = store._actions[type] || (store._actions[type] = [])
  entry.push(function wrappedActionHandler (payload, cb) {
    let res = handler.call(store, {
      dispatch: local.dispatch,
      commit: local.commit,
      getters: local.getters,
      state: local.state,
      rootGetters: store.getters,
      rootState: store.state
    }, payload, cb)
    /* 判断是否是 Promise */
    if (!isPromise(res)) {
      /* 不是 Promise 对象的时候转化称 Promise 对象 */
      res = Promise.resolve(res)
    }
    if (store._devtoolHook) {
      /* 存在 devtool 插件的时候触发 vuex 的 error 给 devtool */
      return res.catch(err => {
        store._devtoolHook.emit('vuex:error', err)
        throw err
      })
    } else {
      return res
    }
  })
}
```

因为 `registerAction` 的时候将 `push` 进 `_actions的action` 的过程进行了一层封装（`wrappedActionHandler`），所以我们在进行 `dispatch` 的第一个参数中能够获取 `state`、`commit` 等方法。

之后，执行结果 `res` 会被判断是否是 Promise，不是则会将其转化成 Promise 对象。`dispatch` 时则从 `_actions` 中取出，只有一个的时候直接返回，是多个则用 `Promise.all` 处理再返回。

关于 DevTools

一件有意思的事情是在 Vue DevTools 里面我们也可以侦测到 Vuex 的状态变化，这当然也是我们一定要坚持 mutation 原则的意义：




```
/* 从 window 对象的 __VUE_DEVTOOLS_GLOBAL_HOOK__中 获取 devtool 插件 */
const devtoolHook =
  typeof window !== 'undefined' &&
  window.__VUE_DEVTOOLS_GLOBAL_HOOK__

export default function devtoolPlugin (store) {
  if (!devtoolHook) return

  /* devtool 插件实例存储在 store 的 _devtoolHook 上 */
  store._devtoolHook = devtoolHook

  /* 出发 vuex 的初始化事件，并将 store 的引用地址传给 devtool 插件，使插件获取 store 的实例 */
  devtoolHook.emit('vuex:init', store)

  /* 监听 travel-to-state 事件 */
  devtoolHook.on('vuex:travel-to-state', targetState => {
    /* 重制 state */
    store.replaceState(targetState)
  })

  /* 订阅 store 的变化 */
  store.subscribe((mutation, state) => {
    devtoolHook.emit('vuex:mutation', mutation, state)
```

```
  })
}
```

其他值得一提的问题：

1. **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。

2. **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**：[**首先要说的是，这个问题出自 美团技术文档，本文也有很多处援引了其中的精辟见解，但我个人觉得此处他们写的并不好，读起来我反正挺迷惑的。不清楚他说的 「封装」 到底是什么意思**]
>
> 我的理解是根据 Vuex 文档原文的 Actions 章节：
>
> > 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。
>
> 所以跳转到 Modules 章节：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 —— 即从上至下进行同样方式的分割。**
>
> 所以最后 `action` 执行的时候 context 对象是根据当前对应的 module 来的。