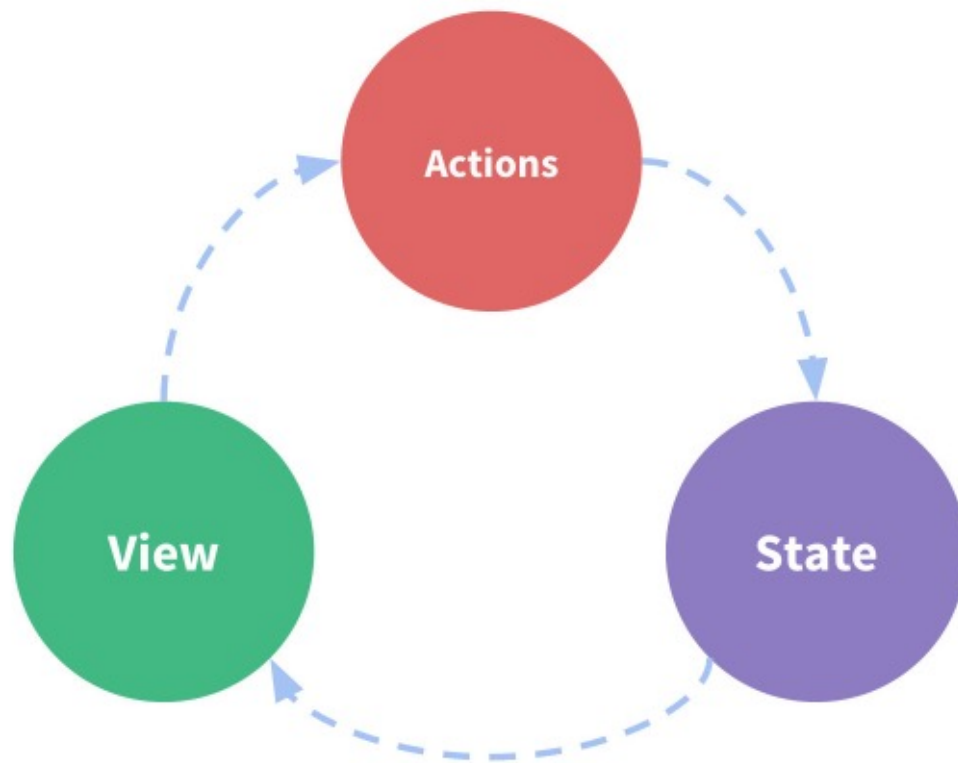


Vuex的状态管理

王红元 coderwhy

什么是状态管理

- 在开发中，我们会的应用程序需要处理各种各样的数据，这些数据需要保存在我们应用程序中的某一个位置，对于这些数据的管理我们就称之为是 **状态管理**。
- 在前面我们是如何管理自己的状态呢？
 - 在Vue开发中，我们使用组件化的开发方式；
 - 而在组件中我们定义data或者在setup中返回使用的数据，这些数据我们称之为state；
 - 在模块template中我们可以使用这些数据，模块最终会被渲染成DOM，我们称之为View；
 - 在模块中我们会产生一些行为事件，处理这些行为事件时，有可能会修改state，这些行为事件我们称之为actions；





复杂的状态管理

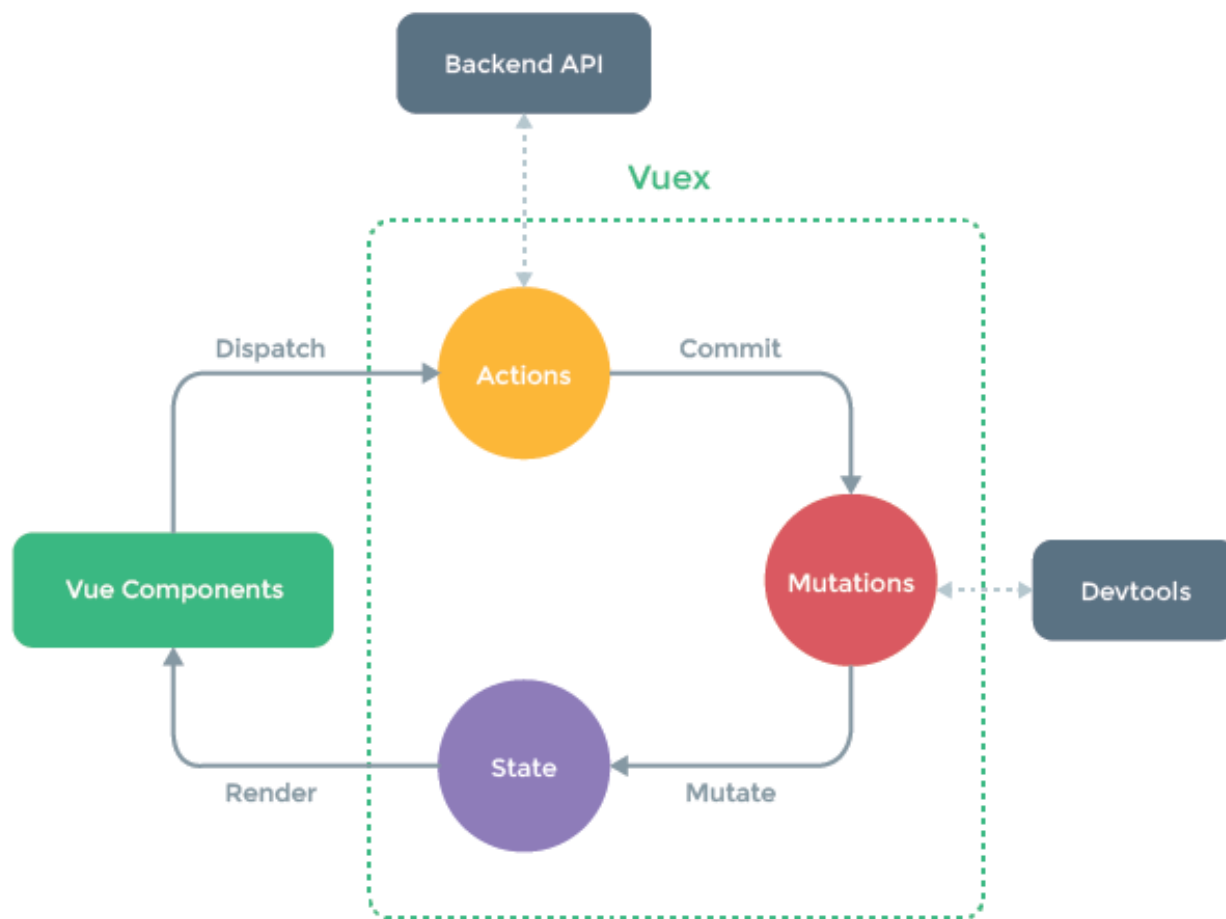
- JavaScript开发的应用程序，已经变得越来越复杂了：
 - JavaScript需要管理的状态越来越多，越来越复杂；
 - 这些状态包括服务器返回的数据、缓存数据、用户操作产生的数据等等；
 - 也包括一些UI的状态，比如某些元素是否被选中，是否显示加载动效，当前分页；
- 当我们的应用遇到**多个组件共享状态**时，单向数据流的简洁性很容易被破坏：
 - 多个视图依赖于同一状态；
 - 来自不同视图的行为需要变更同一状态；
- 我们是否可以通过组件数据的传递来完成呢？
 - 对于一些简单的状态，确实可以通过props的传递或者Provide的方式来共享状态；
 - 但是对于复杂的状态管理来说，显然单纯通过传递和共享的方式是不足以解决问题的，比如兄弟组件如何共享数据呢？



Vuex的状态管理

- 管理不断变化的state本身是非常困难的：
 - 状态之间相互会存在依赖，一个状态的变化会引起另一个状态的变化，View页面也有可能引起状态的变化；
 - 当应用程序复杂时，state在什么时候，因为什么原因而发生了变化，发生了怎么样的变化，会变得非常难以控制和追踪；
- 因此，我们是否可以考虑将组件的内部状态抽离出来，以一个全局单例的方式来管理呢？
 - 在这种模式下，我们的组件树构成了一个巨大的“试图View”；
 - 不管在树的哪个位置，任何组件都能获取状态或者触发行为；
 - 通过定义和隔离状态管理中的各个概念，并通过强制性的规则来维护视图和状态间的独立性，我们的代码边会变得更加结构化和易于维护、跟踪；
- 这就是Vuex背后的基本思想，它借鉴了Flux、Redux、Elm（纯函数语言，redux有借鉴它的思想）：

Vuex的状态管理



Vuex的安装

■ 依然我们要使用vuex，首先第一步需要安装vuex：

□ 我们这里使用的是vuex4.x，安装的时候需要添加 next 指定版本；

```
npm install vuex@next
```

- 每一个Vuex应用的核心就是store（仓库）：
 - store本质上是一个容器，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什么区别呢？**
- 第一：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被更新；
- 第二：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显示**提交 (commit) mutation**；
 - 这样使得我们可以方便的跟踪每一个状态的变化，从而让我们能够通过一些工具帮助我们更好的管理应用的状态；
- 使用步骤：
 - 创建Store对象；
 - 在app中通过插件安装；

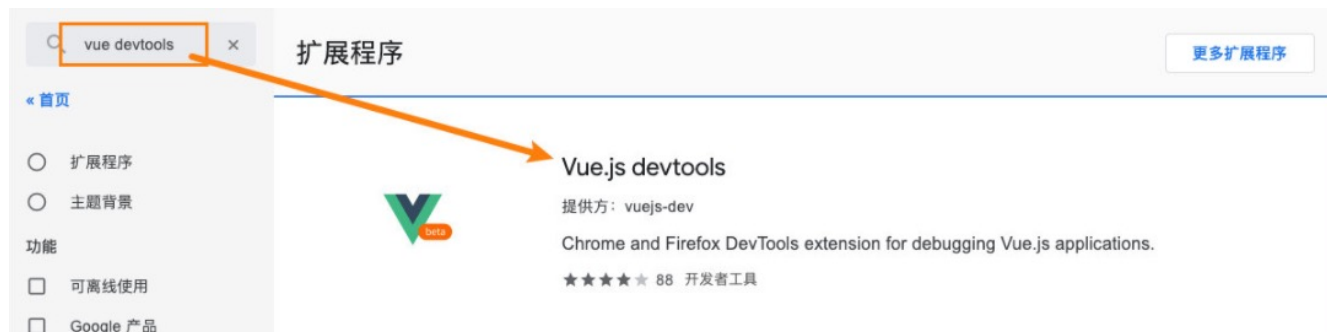


组件中使用store

- 在组件中使用store，我们按照如下的方式：
 - 在模板中使用；
 - 在options api中使用，比如computed；
 - 在setup中使用；

Vue devtool

- vue其实提供了一个devtools，方便我们对组件或者vuex进行调试：
 - 我们需要安装beta版本支持vue3，目前是6.0.0 beta15；
- 它有两种常见的安装方式：
 - 方式一：通过chrome的商店；
 - 方式二：手动下载代码，编译、安装；
- 方式一：通过Chrome商店安装：
 - 由于某些原因我们可能不能正常登录Chrome商店，所以可以选择第二种；



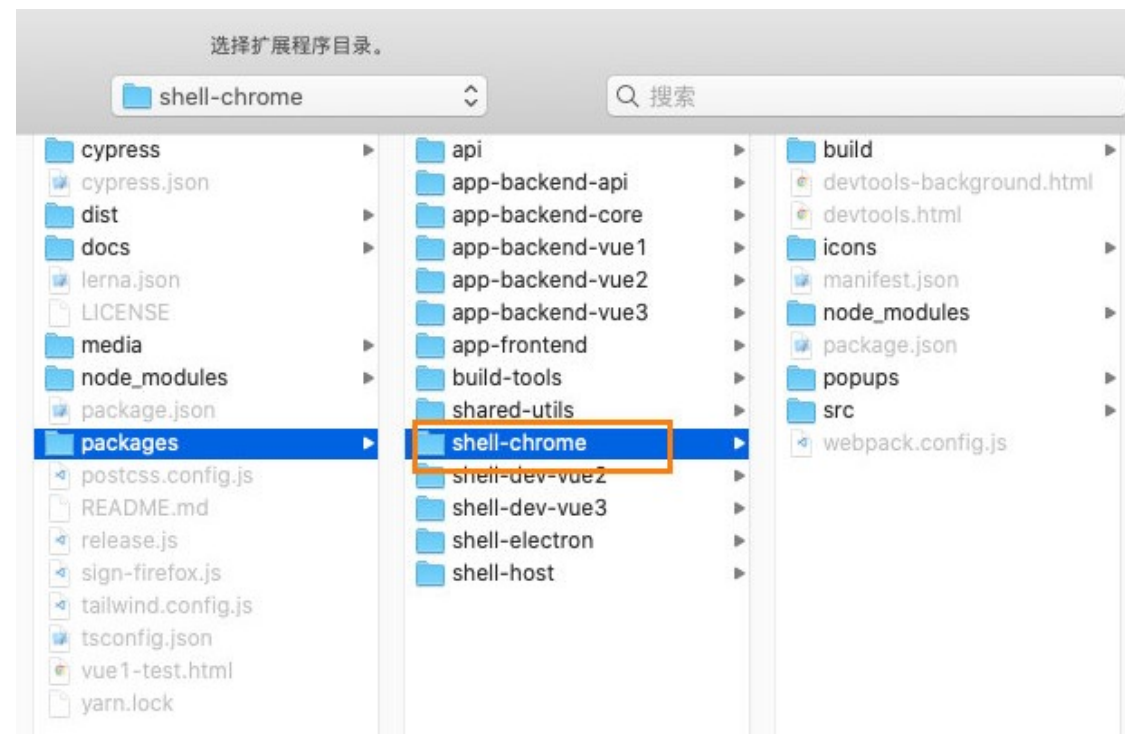
手动安装devtool

■ 方式二：手动下载代码，编译、安装

❑ <https://github.com/vuejs/devtools/tree/v6.0.0-beta.15> 下载代码；

❑ 执行 `yarn install` 安装相关的依赖；

❑ 执行 `yarn run build` 打包；



单一状态树

■ Vuex 使用单一状态树：

- 用一个对象就包含了全部的应用层级的状态；
- 采用的是SSOT，Single Source of Truth，也可以翻译成单一数据源；
- 这也意味着，每个应用将仅仅包含一个 store 实例；
- 单状态树和模块化并不冲突，后面我们会讲到module的概念；

■ 单一状态树的优势：

- 如果你的状态信息是保存到多个Store对象中的，那么之后的管理和维护等等都会变得特别困难；
- 所以Vuex也使用了单一状态树来管理应用层级的全部状态；
- 单一状态树能够让我们最直接的方式找到某个状态的片段，而且在之后的维护和调试过程中，也可以非常方便的管理和维护；

组件获取状态

- 在前面我们已经学习过如何在组件中获取状态了。
- 当然，如果觉得那种方式有点繁琐（表达式过长），我们可以使用计算属性：

```
computed: {  
  counter() {  
    return this.$store.state.counter  
  }  
},
```

- 但是，如果我们有很多个状态都需要获取话，可以使用mapState的辅助函数：
 - mapState的方式一：对象类型；
 - mapState的方式二：数组类型；
 - 也可以使用展开运算符和来原有的computed混合在一起；

在setup中使用mapState

- 在setup中如果我们单个获取装是非常简单的：
 - 通过useStore拿到store后去获取某个状态即可；
 - 但是如果我们需要使用 mapState 的功能呢？
- 默认情况下，Vuex并没有提供非常方便的使用mapState的方式，这里我们进行了一个函数的封装：

```
import { useStore, mapState } from 'vuex';
import { computed } from 'vue';

export function useState(mapper) {
  const store = useStore();

  const stateFns = mapState(mapper)

  const state = {}
  Object.keys(stateFns).forEach(fnKey => {
    state[fnKey] = computed(stateFns[fnKey].bind({$store: store}))
  })

  return state
}
```

```
setup() {
  const state = useState({
    name: state => state.name,
    age: state => state.age,
    height: state => state.height
  })

  return {
    ...state
  }
}
```

getters的基本使用

- 某些属性我们可能需要经过变化后来使用，这个时候可以使用getters：

```
const store = createStore({
  state() {
    return {
      counter: 0,
      name: "coderwhy",
      age: 18,
      height: 1.88,
      books: [
        {name: "vuejs", count: 2, price: 110},
        {name: "react", count: 3, price: 120},
        {name: "webpack", count: 4, price: 130},
      ]
    }
  },
  getters: {
    totalPrice(state) {
      let totalPrice = 0;
      for (const book of state.books) {
        totalPrice += book.count * book.price
      }
      return totalPrice
    }
  }
})
```

```
<div>
  <h2>{{ $store.getters.totalPrice }}</h2>
</div>
```

getters第二个参数

- getters可以接收第二个参数：

```
getters: {  
  totalPrice(state, getters) {  
    let totalPrice = 0;  
    for (const book of state.books) {  
      totalPrice += book.count * book.price;  
    }  
  
    return totalPrice + ", " + getters.myName;  
  },  
  myName(state) {  
    return state.name;  
  },  
},
```

getters的返回函数

- getters中的函数本身，可以返回一个函数，那么在使用地方相当于可以调用这个函数：

```
getters: {  
  totalPrice(state) {  
    return (price) => {  
      let totalPrice = 0;  
      for (const book of state.books) {  
        if (book.price < price) continue  
        totalPrice += book.count * book.price  
      }  
  
      return totalPrice  
    }  
  }  
}
```


mapGetters的辅助函数

- 这里我们也可以使用mapGetters的辅助函数。

```
computed: {  
  ...mapGetters(["totalPrice", "myName"]),  
  ...mapGetters({  
    finalPrice: "totalPrice",  
    finalName: "myName",  
  }),  
},
```

- 在setup中使用

```
import { useStore, mapGetters } from 'vuex';  
import { computed } from 'vue';  
  
export function useGetters(mapper) {  
  const store = useStore();  
  
  const stateFns = mapGetters(mapper)  
  
  const state = {}  
  Object.keys(stateFns).forEach(fnKey => {  
    state[fnKey] = computed(stateFns[fnKey].bind({ $store: store }))  
  })  
  
  return state  
}
```

Mutation基本使用

- 更改 Vuex 的 store 中的状态的唯一方法是提交 mutation :

```
mutations: {  
  increment(state) {  
    state.counter++  
  },  
  decrement(state) {  
    state.counter--  
  }  
},
```

Mutation携带数据

- 很多时候我们在提交mutation的时候，会携带一些数据，这个时候我们可以使用参数：

```
mutations: {  
  addNumber(state, payload) {  
    state.counter += payload  
  }  
},
```

- payload为对象类型

```
addNumber(state, payload) {  
  state.counter += payload.count  
}
```

- 对象风格的提交方式

```
$store.commit({  
  type: "addNumber",  
  count: 100  
})
```



Mutation常量类型

■ 定义常量：mutation-type.js

```
e > JS mutation-types.js > [🔗] ADD_NUMBER  
export const ADD_NUMBER = 'ADD_NUMBER'
```

■ 定义mutation

```
[ADD_NUMBER](state, payload) {  
  state.counter += payload.count  
}
```

■ 提交mutation

```
$store.commit({  
  type: ADD_NUMBER,  
  count: 100  
})
```

mapMutations辅助函数

- 我们也可以借助于辅助函数，帮助我们快速映射到对应的方法中：

```
methods: {  
  ...mapMutations({  
    addNumber: ADD_NUMBER,  
  }),  
  ...mapMutations(["increment", "decrement"]),  
},
```

- 在setup中使用也是一样的：

```
const mutations = mapMutations(['increment', 'decrement']);  
const mutations2 = mapMutations({  
  addNumber: ADD_NUMBER  
})
```

mutation重要原则

- 一条重要的原则就是要记住 **mutation 必须是同步函数**
 - 这是因为devtool工具会记录mutation的日记；
 - 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；
 - 但是在mutation中执行异步操作，就无法追踪到数据的变化；
 - 所以Vuex的重要原则中要求 mutation必须是同步函数；

actions的基本使用

■ Action类似于mutation，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；

```
mutations: {  
  increment(state) {  
    state.counter++  
  },  
  actions: {  
    increment(context) {  
      context.commit("increment")  
    }  
  },  
}
```

■ 这里有一个非常重要的参数context：

- context是一个和store实例均有相同方法和属性的context对象；
- 所以我们可以从其中获取到commit方法来提交一个mutation，或者通过 context.state 和 context.getters 来获取 state 和 getters；
- 但是为什么它不是store对象呢？这个等到我们讲Modules时再具体来说；

actions的分发操作

■ 如何使用action呢？进行action的分发：

□ 分发使用的是 store 上的dispatch函数；

```
add() {  
  this.$store.dispatch("increment");  
},
```

■ 同样的，它也可以携带我们的参数：

```
add() {  
  this.$store.dispatch("increment", { count: 100 });  
},
```

■ 也可以以对象的形式进行分发：

```
add() {  
  this.$store.dispatch({  
    type: "increment",  
    count: 100,  
  });  
},
```


actions的辅助函数

■ action也有对应的辅助函数：

- 对象类型的写法；
- 数组类型的写法；

```
methods: {  
  ...mapActions(["increment", "decrement"]),  
  ...mapActions({  
    add: "increment",  
    sub: "decrement"  
  })  
},
```

```
const actions1 = mapActions(["decrement"]);  
const actions2 = mapActions({  
  add: "increment",  
  sub: "decrement"  
})
```

actions的异步操作

- Action 通常是异步的，那么如何知道 action 什么时候结束呢？
 - 我们可以通过让action返回Promise，在Promise的then中来处理完成后的操作；

```
actions: {  
  increment(context) {  
    return new Promise((resolve) => {  
      setTimeout(() => {  
        context.commit("increment")  
        resolve("异步完成")  
      }, 1000);  
    })  
  }  
}
```

```
const store = useStore();  
const increment = () => {  
  store.dispatch("increment").then(res => {  
    console.log(res, "异步完成");  
  })  
}
```

module的基本使用

■ 什么是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；

```
const moduleA = {  
  state: () => ({}),  
  mutations: {},  
  actions: {},  
  getters: {}  
}  
  
const moduleB = {  
  state: () => ({}),  
  mutations: {},  
  actions: {}  
}
```

```
const store = createStore({  
  modules: {  
    a: moduleA,  
    b: moduleB  
  }  
})  
  
store.state.a // -> moduleA 的状态  
store.state.b // -> moduleB 的状态
```

module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：

```
mutations: {  
  changeName(state) {  
    state.name = "coderwhy"  
  }  
},  
getters: {  
  info(state, getters, rootState) {  
    return `name:${state.name} age:${state.age} height:${state.height}`  
  }  
}
```

```
actions: {  
  changeNameAction({state, commit, rootState}) {  
    commit("changeName", "kobe")  
  }  
}
```

module的命名空间

- 默认情况下，模块内部的action和mutation仍然是注册在**全局的命名空间**中的：
 - 这样使得多个模块能够对同一个 action 或 mutation 作出响应；
 - Getter 同样也默认注册在全局命名空间；
- 如果我们希望模块具有更高的封装度和复用性，可以添加 `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；

```
namespaced: true,
// 访问: $store.state.user.xxx
state() {
  return {
    name: "why",
    age: 18,
    height: 1.88
  }
},
mutations: {
  // 访问: $store.commit("user/changeName")
  changeName(state) {
    state.name = "coderwhy"
  }
},
```

```
getters: {
  // 访问: $store.getters["user/info"]
  // 这里会有四个参数:
  info(state, getters, rootState, rootGetters) {
    return `name:${state.name} age:${state.age} height:${state.height}`
  }
},
actions: {
  // 这里一共有六个参数
  changeNameAction({commit, dispatch, state, rootState, getters, rootGetters}) {
    commit("changeName", "kobe")
  }
}
```

module修改或派发根组件

- 如果我們希望在action中修改root中的state，那么有如下的方式：

```
// 这里一共有六个参数
changeNameAction({commit, dispatch, state, rootState, getters, rootGetters}) {
  commit("changeName", "kobe");

  commit("changeRootName", null, {root: true});
  dispatch("changeRootNameAction", null, {root: true})
}
```

module的辅助函数

■ 如果辅助函数有三种使用方法：

- 方式一：通过完整的模块空间名称来查找；
- 方式二：第一个参数传入模块空间名称，后面写上要使用的属性；
- 方式三：通过 createNamespacedHelpers 生成一个模块的辅助函数；

```
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']()  
  ]),  
},
```

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


module的辅助函数

■ 第三种方式：createNamespacedHelpers

```
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'
    ])
  }
}
```


对useState和useGetters修改

```
export function useState(mapper, moduleId) {  
  let stateFn = mapState;  
  if (moduleId) {  
    stateFn = createNamespacedHelpers(moduleId).mapState  
  }  
  return useUtil(mapper, stateFn)  
}
```

```
export function useGetters(mapper, moduleId) {  
  let mapModuleGetter = mapGetters;  
  if (moduleId) {  
    mapModuleGetter = createNamespacedHelpers(moduleId).mapGetters  
  }  
  return useUtil(mapper, mapModuleGetter)  
}
```

自定义指令 format-time bug修复

```
export default function(app) {  
  // let formatString = "YYYY-MM-DD HH:mm:ss";  
  app.directive("format-time", {  
    created(el, bindings) {  
      bindings.formatString = "YYYY-MM-DD HH:mm:ss"  
      if (bindings.value) {  
        bindings.formatString = bindings.value;  
      }  
    },  
    mounted(el, bindings) {  
      console.log("format mounted");  
      const textContent = el.textContent;  
      let timestamp = parseInt(textContent);  
      if (textContent.length === 10) {  
        timestamp = timestamp * 1000  
      }  
      el.textContent = dayjs(timestamp).format(bindings.formatString);  
    }  
  })  
}
```



nexttick

- 官方解释：将回调推迟到下一个 DOM 更新周期之后执行。在更改了一些数据以等待 DOM 更新后立即使用它。
- 比如我们有下面的需求：
 - 点击一个按钮，我们会修改在h2中显示的message；
 - message被修改后，获取h2的高度；
- 实现上面的案例我们有三种方式：
 - 方式一：在点击按钮后立即获取到h2的高度（错误的做法）
 - 方式二：在updated生命周期函数中获取h2的高度（但是其他数据更新，也会执行该操作）
 - 方式三：使用nexttick函数；
- nexttick是如何做到的呢？





historyApiFallback

- historyApiFallback是开发中一个非常常见的属性，它主要的作用是解决SPA页面在路由跳转之后，进行页面刷新时，返回404的错误。
- boolean值：默认是false
 - 如果设置为true，那么在刷新时，返回404错误时，会自动返回 index.html 的内容；
- object类型的值，可以配置rewrites属性：
 - 可以配置from来匹配路径，决定要跳转到哪一个页面；
- 事实上devServer中实现historyApiFallback功能是通过connect-history-api-fallback库的：
 - 可以查看[connect-history-api-fallback](#) 文档

```
location / {  
  root    /usr/local/docker/web/gzfsweb/;  
  # vue工程用的路由是history模式  
  try_files $uri $uri/ /index.html;  
  index   index.html index.htm;  
}
```

