

Vuex详解

王红元

微博: coderwhy

微信: 372623326



实力IT教育 www.520it.com

Vuex是做什么的？

n 官方解释：Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。

p 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。

p Vuex 也集成到 Vue 的官方调试工具 [devtools extension](#)，提供了诸如零配置的 time-travel 调试、状态快照导入导出等高级调试功能。

n 状态管理到底是什么？

p **状态管理模式**、**集中式存储管理**这些名词听起来就非常高大上，让人捉摸不透。

p 其实，你可以简单的将其看成把需要多个组件共享的变量全部存储在一个对象里面。

p 然后，将这个对象放在顶层的Vue实例中，让其他组件可以使用。

p 那么，多个组件是不是就可以共享这个对象中的所有变量属性了呢？

n 等等，如果是这样的话，为什么官方还要专门出一个插件Vuex呢？难道我们不能自己封装一个对象来管理吗？

p 当然可以，只是我们要先想想VueJS带给我们最大的便利是什么呢？没错，就是响应式。

p 如果你自己封装实现一个对象能不能保证它里面所有的属性做到响应式呢？当然也可以，只是自己封装可能稍微麻烦一些。

p 不用怀疑，Vuex就是为了提供这样一个在多个组件间共享状态的插件，用它就可以了。

管理什么状态呢？

n 但是，有什么状态时需要我们在多个组件间共享的呢？

p 如果你做过大型开放，你一定遇到过多个状态，在多个界面间的共享问题。

p 比如用户的登录状态、用户名称、头像、地理位置信息等等。

p 比如商品的收藏、购物车中的物品等等。

p 这些状态信息，我们都可以放在统一的地方，对它进行保存和管理，而且它们还是响应式的（待会儿我们就可以看到代码了，莫着急）。

n OK，从理论上理解了状态管理之后，让我们从实际的代码再来看看状态管理。

p 毕竟，Talk is cheap, Show me the code.(来自Linux)

n 我们先来看看但界面的状态管理吧。



单界面的状态管理

n 我们知道，要在单个组件中进行状态管理是一件非常简单的事情

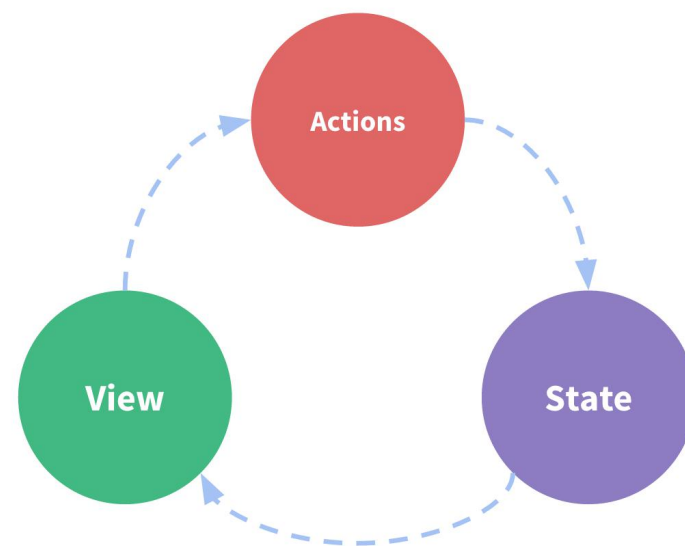
p 什么意思呢？我们来看下面的图片。

n 这图片中的三种东西，怎么理解呢？

p State：不用多说，就是我们的状态。（你姑且可以当做就是data中的属性）

p View：视图层，可以针对State的变化，显示不同的信息。（这个好理解吧？）

p Actions：这里的Actions主要是用户的各种操作：点击、输入等等，会导致状态的改变。



当前计数: 0

+1 -1

n 写点代码，加深理解：

p 看下右边的代码效果, 肯定会实现吧?

单界面状态管理的实现

```
<template>
  <div class="test">
    <div>当前计数: {{counter}}</div>
    <button @click="counter+=1">+1</button>
    <button @click="counter-=1">-1</button>
  </div>
</template>
```

```
<script>
export default {
  name: 'HelloWorld',
  data () {
    return {
      counter: 0
    }
  }
}
```

made by coderwhy
微博:coderwhy

```
</script>
```

```
<style scoped>
</style>
```

- n 在这个案例中，我们有木有状态需要管理呢？没错，就是个数counter。
- n counter需要某种方式被记录下来，也就是我们的State。
- n counter目前的值需要被显示在界面中，也就是我们的View部分。
- n 界面发生某些操作时（我们这里是用户的点击，也可以是用户的input），需要去更新状态，也就是我们的Actions
- n 这不就是上面的流程图了吗？

多界面状态管理

n Vue已经帮我们做好了单个界面的状态管理，但是如果是多个界面呢？

p 多个试图都依赖同一个状态（一个状态改了，多个界面需要进行更新）

p 不同界面的Actions都想修改同一个状态（Home.vue需要修改，Profile.vue也需要修改这个状态）

n 也就是说对于某些状态(状态1/状态2/状态3)来说只属于我们某一个试图，但是也有一些状态(状态a/状态b/状态c)属于多个试图共同想要维护的

p 状态1/状态2/状态3你放在自己的房间中，你自己管理自己用，没问题。

p 但是状态a/状态b/状态c我们希望交给一个大管家来统一帮助我们管理！！

p 没错，Vuex就是为我们提供这个大管家的工具。

n 全局单例模式（大管家）

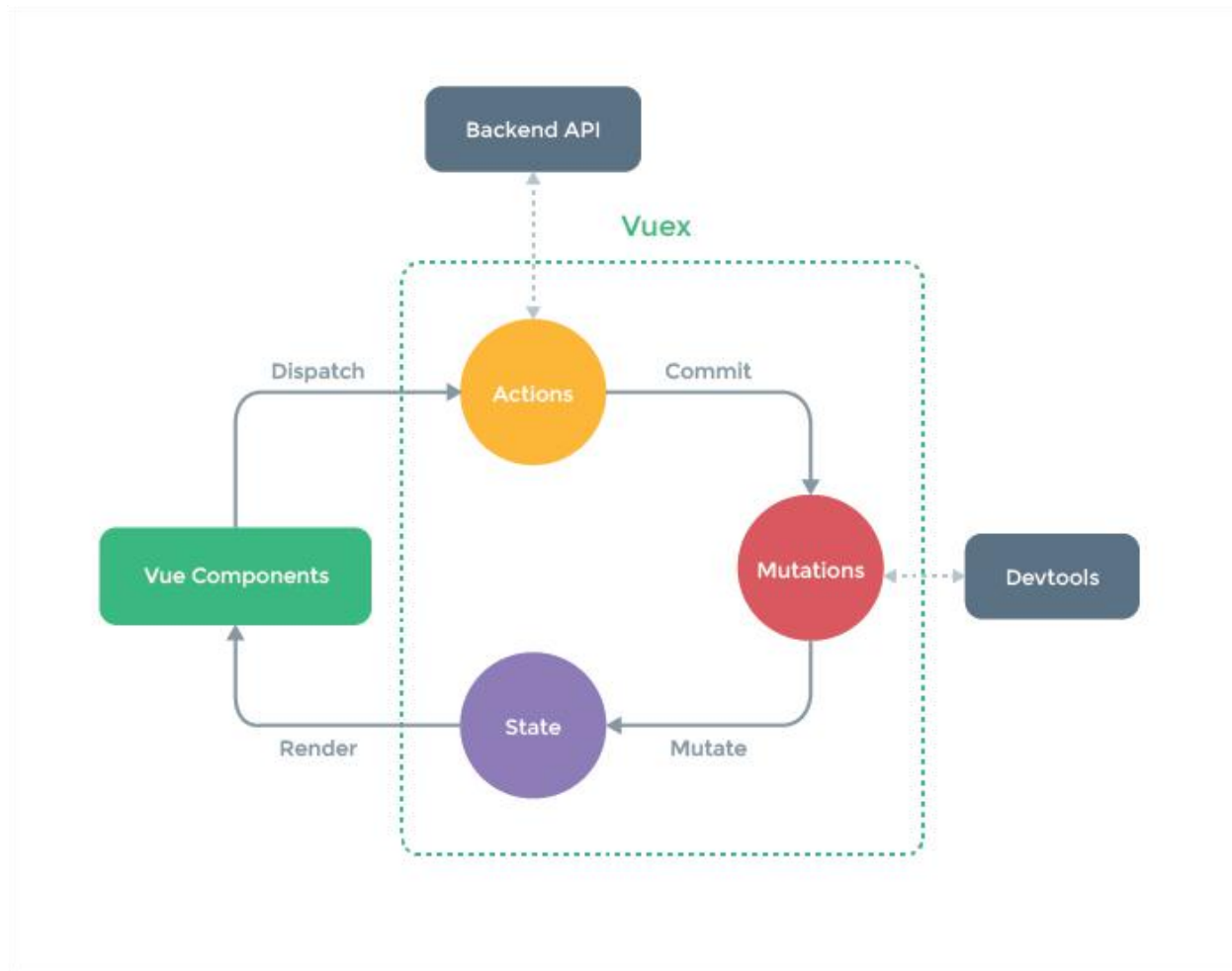
p 我们现在要做的就是将共享的状态抽取出来，交给我们的管家，统一进行管理。

p 之后，你们每个试图，按照我规定好的规定，进行访问和修改等操作。

p 这就是Vuex背后的基本思想。

Vuex状态管理图例

n 一起来看一副官方给出的图片



n 我们还是实现一下之前简单的案例

0



n 首先，我们需要在某个地方存放我们的Vuex代码：

p 这里，我们先创建一个文件夹store，并且在其中创建一个index.js文件

p 在index.js文件中写入如下代码：

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

Vue.use(Vuex)

const store = new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    }
  }
})

export default store
```

made by coderwhy
微博:coderwhy

n 其次，我们让所有的Vue组件都可以使用这个store对象

p 来到main.js文件，导入store对象，并且放在new Vue中

p 这样，在其他Vue组件中，我们就可以通过this.\$store的方式，获取到这个store对象了

```
import Vue from 'vue'  
import App from './App'  
import store from './store'
```

```
new Vue({  
  el: '#app',  
  store,  
  render: h => h(App)  
})
```

made by coderwhy
微博: coderwhy

使用Vuex的count

```
<template>
  <div id="app">
    <p>{{count}}</p>
    <button @click="increment">+1</button>
    <button @click="decrement">-1</button>
  </div>
</template>

<script>
  export default {
    name: 'App',
    components: {
    },
    computed: {
      count: function () {
        return this.$store.state.count
      }
    },
    methods: {
      increment: function () {
        this.$store.commit('increment')
      },
      decrement: function () {
        this.$store.commit('decrement')
      }
    }
  }
</script>

<style>
</style>
```

n 好的，这就是使用Vuex最简单的方式了。

n 我们来对使用步骤，做一个简单的小节：

p 1.提取出一个公共的store对象，用于保存在多个组件中共享的状态

p 2.将store对象放置在new Vue对象中，这样可以保证在所有的组件中都可以使用到

p 3.在其他组件中使用store对象中保存的状态即可

Ø 通过this.\$store.state.属性的方式来访问状态

Ø 通过this.\$store.commit('mutation中方法')来修改状态

n 注意事项：

p 我们通过提交mutation的方式，而非直接改变store.state.count。

p 这是因为Vuex可以更明确的追踪状态的变化，所以不要直接改变store.state.count的值。

n Vuex有几个比较核心的概念:

- p State

- p Getters

- p Mutation

- p Action

- p Module

n 我们对它进行一一介绍.

State单一状态树

n Vuex提出使用单一状态树, 什么是单一状态树呢？

p 英文名称是Single Source of Truth，也可以翻译成单一数据源。

n 但是，它是什么呢？我们来看一个生活中的例子。

p OK，我用一个生活中的例子做一个简单的类比。

p 我们知道，在国内我们有很多的信息需要被记录，比如上学时的个人档案，工作后的社保记录，公积金记录，结婚后的婚姻信息，以及其他相关的户口、医疗、文凭、房产记录等等（还有很多信息）。

p 这些信息被分散在很多地方进行管理，有一天你需要办某个业务时(比如入户某个城市)，你会发现你需要到各个对应的工作地点去打印、盖章各种资料信息，最后到一个地方提交证明你的信息无误。

p 这种保存信息的方案，不仅仅低效，而且不方便管理，以及日后的维护也是一个庞大的工作(需要大量的各个部门的人力来维护，当然国家目前已经在完善我们的这个系统了)。

n 这个和我们在应用开发中比较类似：

p 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。

p 所以Vuex也使用了单一状态树来管理应用层级的全部状态。

p 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。

Getters基本使用

n 有时候，我们需要从store中获取一些state变异后的状态，比如下面的Store中：

p 获取学生年龄大于20的个数。

```
const store = new Vuex.Store({  
  state: {  
    students: [  
      {id: 110, name: 'why', age: 18},  
      {id: 111, name: 'kobe', age: 21},  
      {id: 112, name: 'lucy', age: 25},  
      {id: 113, name: 'lilei', age: 30},  
    ]  
  }  
})
```

n 我们可以在Store中定义getters

```
computed: {  
  getGreaterAgesCount() {  
    return this.$store.state.students.filter(age => age >= 20).length  
  }  
},
```

```
getters: {  
  greaterAgesCount: state => {  
    return state.students.filter(s => s.age >= 20).length  
  }  
}
```

Getters作为参数和传递参数

n 如果我们已经有了一个获取所有年龄大于20岁学生列表的getters, 那么代码可以这样来写

```
getters: {  
  greaterAgesStus: state => {  
    return state.students.filter(s => s.age >= 20)  
  },  
  greaterAgesCount: (state, getters) => {  
    return getters.greaterAgesStus.length  
  }  
} made by coderwhy 微博:coderwhy
```

n getters默认是不能传递参数的, 如果希望传递参数, 那么只能让getters本身返回另一个函数.

p 比如上面的案例中,我们希望根据ID获取用户的信息

```
getters: {  
  stuByID: state => {  
    return id => {  
      return state.students.find(s => s.id === id)  
    }  
  }  
}
```

n Vuex的store状态的更新唯一方式：提交Mutation

n Mutation主要包括两部分：

p 字符串的**事件类型**（type）

p 一个**回调函数**（handler），该回调函数的第一个参数就是state。

n mutation的定义方式：

```
mutations: {  
  increment(state) {  
    state.count++  
  }  
}
```

n 通过mutation更新

```
increment: function () {  
  this.$store.commit('increment')  
}
```

Mutation传递参数

n 在通过mutation更新数据的时候, 有可能我们希望携带一些额外的参数

p 参数被称为是mutation的载荷(Payload)

n Mutation中的代码:

```
decrement(state, n) {  
  state.count -= n  
}
```

```
decrement: function () {  
  this.$store.commit('decrement', 2)  
}
```

n 但是如果参数不是一个呢?

p 比如我们有很多参数需要传递.

p 这个时候, 我们通常会以对象的形式传递, 也就是payload是一个对象.

p 这个时候可以再从对象中取出相关的信息.

```
changeCount(state, payload) {  
  state.count = payload.count  
}
```

```
changeCount: function () {  
  this.$store.commit('changeCount', {count: 0})  
}
```


Mutation提交风格

- n 上面的通过commit进行提交是一种普通的方式
- n Vue还提供了另外一种风格, 它是一个包含type属性的对象

```
this.$store.commit({  
  type: 'changeCount',  
  count: 100  
})
```

- n Mutation中的处理方式是将整个commit的对象作为payload使用, 所以代码没有改变, 依然如下:

```
changeCount(state, payload) {  
  state.count = payload.count  
}
```

Mutation响应规则

n Vuex的store中的state是响应式的, 当state中的数据发生改变时, Vue组件会自动更新.

n 这就要求我们必须遵守一些Vuex对应的规则:

p 提前在store中初始化好所需的属性.

p 当给state中的对象添加新属性时, 使用下面的方式:

Ø 方式一: 使用Vue.set(obj, 'newProp', 123)

Ø 方式二: 用新对象给旧对象重新赋值

n 我们来看一个例子:

p 当我们点击更新信息时, 界面并没有发生对应改变.

n 如何才能让它改变呢?

p 查看下面代码的方式一和方式二

p 都可以让state中的属性是响应式的.



```
<template>
  <div id="app">
    <p>我的个人信息: {{info}}</p>
    <button @click="updateInfo">更新信息</button>
  </div>
</template>

<script>
  export default {
    name: 'App',
    components: {
    },
    computed: {
      info() {
        return this.$store.state.info
      }
    },
    methods: {
      updateInfo() {
        this.$store.commit('updateInfo', {height: 1.88})
      }
    }
  }
</script>
```

```
const store = new Vuex.Store({
  state: {
    info: {
      name: 'why', age: 18
    }
  },
  mutations: {
    updateInfo(state, payload) {
      state.info['height'] = payload.height
    }
  }
})
```

我的个人信息: {"name": "why", "age": 18}

更新信息

点击更新信息, 但是界面未发生改变.

```
mutations: {
  updateInfo(state, payload) {
    // state.info['height'] = payload.height
    // 方式一: Vue.set()
    Vue.set(state.info, 'height', payload.height)
    // 方式二: 给info赋值一个新的对象
    state.info = {...state.info, 'height': payload.height}
  }
}
```

Mutation常量类型 – 概念

n 我们来考虑下面的问题:

- p 在mutation中, 我们定义了很多事件类型(也就是其中的方法名称).
- p 当我们的项目增大时, Vuex管理的状态越来越多, 需要更新状态的情况越来越多, 那么意味着Mutation中的方法越来越多.
- p 方法过多, 使用者需要花费大量的经历去记住这些方法, 甚至是多个文件间来回切换, 查看方法名称, 甚至如果不是复制的时候, 可能还会出现写错的情况.

n 如何避免上述的问题呢?

- p 在各种Flux实现中, 一种很常见的方案就是使用**常量**替代Mutation**事件**的类型.
- p 我们可以将这些常量放在一个单独的文件中, 方便管理以及让整个app所有的事件类型一目了然.

n 具体怎么做呢?

- p 我们可以创建一个文件: mutation-types.js, 并且在其中定义我们的常量.
- p 定义常量时, 我们可以使用ES2015中的风格, 使用一个常量来作为函数的名称.

Mutation常量类型 - 代码

```
index.js × mutation-types.js × App.vue ×  
1 export const UPDATE_INFO = 'UPDATE_INFO'  
2
```

```
import Vuex from 'vuex'  
import Vue from 'vue'  
import * as types from './mutation-types'  
  
Vue.use(Vuex)  
  
const store = new Vuex.Store({  
  state: {  
    info: {  
      name: 'why', age: 18  
    }  
  },  
  mutations: {  
    [types.UPDATE_INFO](state, payload) {  
      state.info = {...state.info, 'height': payload.height}  
    }  
  }  
})  
  
export default store
```

```
<script>  
  import {UPDATE_INFO} from './store/mutation-types';  
  
  export default {  
    name: 'App',  
    components: {  
    },  
    computed: {  
      info() {  
        return this.$store.state.info  
      }  
    },  
    methods: {  
      updateInfo() {  
        this.$store.commit(UPDATE_INFO, {height: 1.88})  
      }  
    }  
  }  
</script>
```

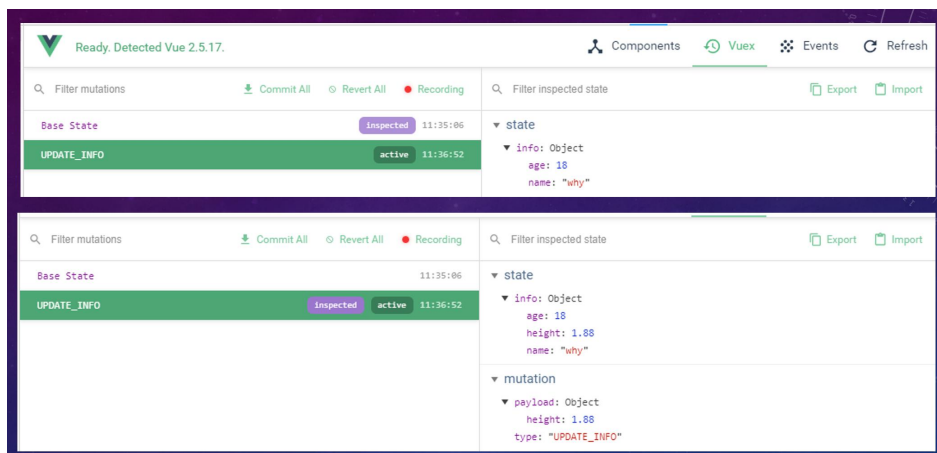
n 通常情况下, Vuex要求我们Mutation中的方法必须是同步方法.

p 主要的原因是当我们使用devtools时, 可以devtools可以帮助我们捕捉mutation的快照.

p 但是如果是异步操作, 那么devtools将不能很好的追踪这个操作什么时候会被完成.

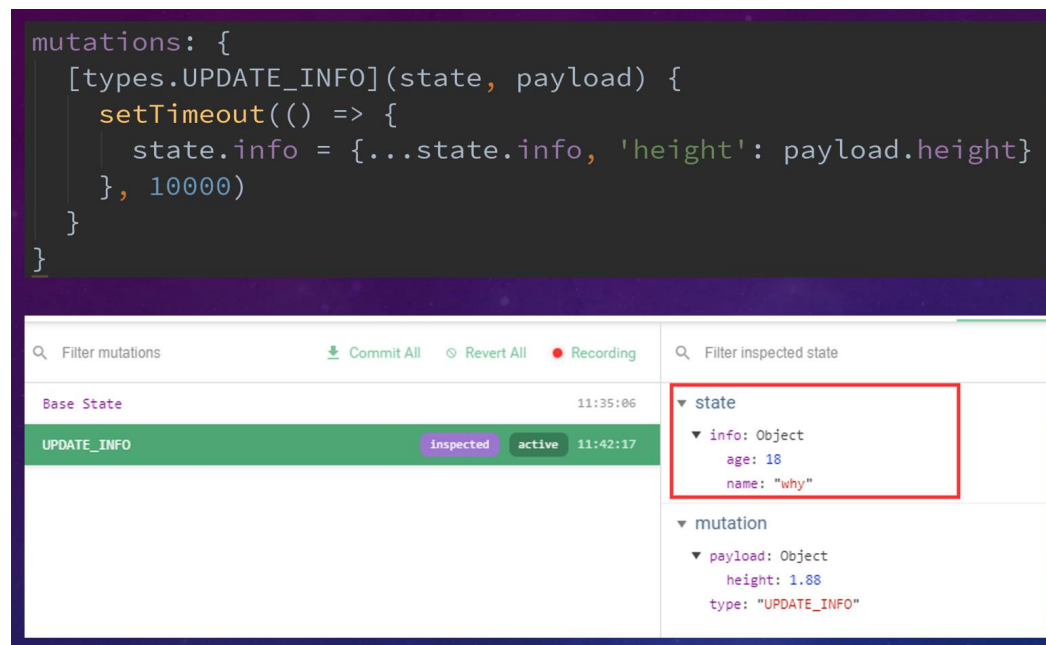
n 比如我们之前的代码, 当执行更新时, devtools中会有如下信息: 图1

n 但是, 如果Vuex中的代码, 我们使用了异步函数: 图2



n 你会发现state中的info数据一直没有被改变, 因为他无法追踪到.

n So, 通常情况下, 不要再mutation中进行异步的操作



Action的基本定义

n 我们强调, 不要再Mutation中进行异步操作.

p 但是某些情况, 我们确实希望在Vuex中进行一些异步操作, 比如网络请求, 必然是异步的. 这个时候怎么处理呢?

p Action类似于Mutation, 但是是用来代替Mutation进行异步操作的.

n Action的基本使用代码如下:

n context是什么?

p context是和store对象具有相同方法和属性的对象.

p 也就是说, 我们可以通过context去进行commit相关的操作, 也可以获取context.state等.

p 但是注意, 这里它们并不是同一个对象, 为什么呢? 我们后面学习Modules的时候, 再具体说.

n 这样的代码是否多此一举呢?

p 我们定义了actions, 然后又在actions中去进行commit, 这不是脱裤放屁吗?

p 事实上并不是这样, 如果在Vuex中有异步操作, 那么我们就可以在actions中完成了.

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


Action的分发

n 在Vue组件中, 如果我们调用action中的方法, 那么就需要使用dispatch

```
methods: {  
  increment() {  
    this.$store.dispatch('increment')  
  }  
}
```

n 同样的, 也是支持传递payload

```
methods: {  
  increment() {  
    this.$store.dispatch('increment', {cCount: 5})  
  }  
}
```

```
mutations: {  
  increment(state, payload) {  
    state.count += payload.cCount  
  }  
},  
actions: {  
  increment(context, payload) {  
    setTimeout(() => {  
      context.commit('increment', payload)  
    }, 5000)  
  }  
}
```

Action返回的Promise

n 前面我们学习ES6语法的时候说过, Promise经常用于异步操作.

p 在Action中, 我们可以将异步操作放在一个Promise中, 并且在成功或者失败后, 调用对应的resolve或reject.

n OK, 我们来看下面的代码:

```
actions: {  
  increment(context) {  
    return new Promise((resolve) => {  
      setTimeout(() => {  
        context.commit('increment')  
        resolve()  
      }, 1000)  
    })  
  }  
}
```

```
methods: {  
  increment() {  
    this.$store.dispatch('increment').then(res => {  
      console.log('完成了更新操作');  
    })  
  }  
}
```


认识Module

n Module是模块的意思, 为什么在Vuex中我们要使用模块呢?

p Vue使用单一状态树,那么也意味着很多状态都会交给Vuex来管理.

p 当应用变得非常复杂时,store对象就有可能变得相当臃肿.

p 为了解决这个问题, Vuex允许我们将store分割成模块(Module), 而每个模块拥有自己的state、mutations、actions、getters等

n 我们按照什么样的方式来组织模块呢?

p 我们来看左边的代码

```
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
```

n 上面的代码中, 我们已经有了整体的组织结构, 下面我们来看看具体的局部模块中的代码如何书写.

p 我们在moduleA中添加state、mutations、 getters

p mutation和getters接收的第一个参数是局部状态对象

```
const moduleA = {
  state: {
    count: 0
  },
  mutations: {
    increment(state) {
      state.count++
    }
  },
  getters: {
    doubleCount(state) {
      return state.count * 2
    }
  }
}

const moduleB = {
}

const store = new Vuex.Store({
  state: {
    gCount: 111
  },
  modules: {
    a: moduleA,
    b: moduleB
  }
})

export default store;
```

```
<script>
export default {
  name: 'App',
  components: {
  },
  computed: {
    count() {
      return this.$store.getters.doubleCount
    }
  },
  methods: {
    increment() {
      this.$store.commit('increment')
    }
  }
}
</script>
```

n 注意:

- Ø 虽然, 我们的doubleCount和increment都是定义在对象内部的.
- Ø 但是在调用的时候, 依然是通过this.\$store来直接调用的.

n actions的写法呢? 接收一个context参数对象

p 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')  
      }  
    }  
  }  
}
```

n 如果getters中也需要使用全局的状态, 可以接受更多的参数

```
const moduleA = {  
  // ...  
  getters: {  
    sumWithRootCount (state, getters, rootState) {  
      return state.count + rootState.count  
    }  
  }  
}
```

n 当我们的Vuex帮助我们管理过多的内容时, 好的项目结构可以让我们的代码更加清晰.

```
├─ index.html
├─ main.js
├─ api
│   └─ ... # 抽取API请求
├─ components
│   ├── App.vue
│   └─ ...
└─ store
    ├── index.js # 我们组装模块并导出 store 的地方
    ├── actions.js # 根级别的 action
    ├── mutations.js # 根级别的 mutation
    └─ modules
        ├── cart.js # 购物车模块
        └─ products.js # 产品模块
```