Vuex

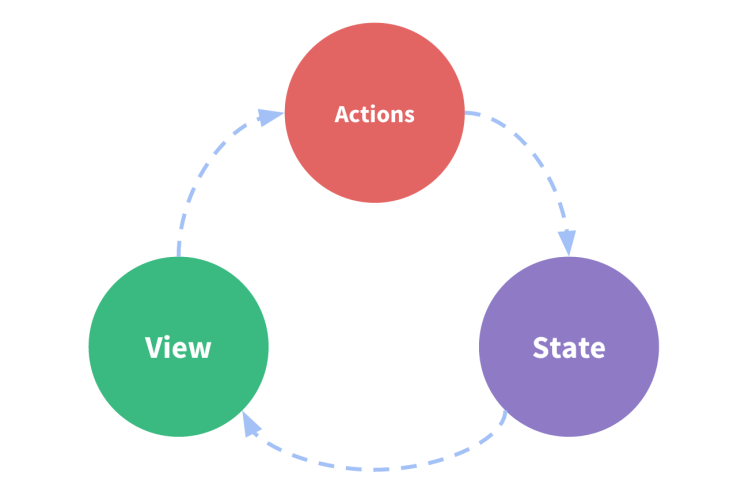
1. Vuex是什么

专为 Vue.js 应用程序开发的**状态管理模式**。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。

1. 什么是“状态管理模式”



以下是一个表示“单向数据流”理念的极简示意:



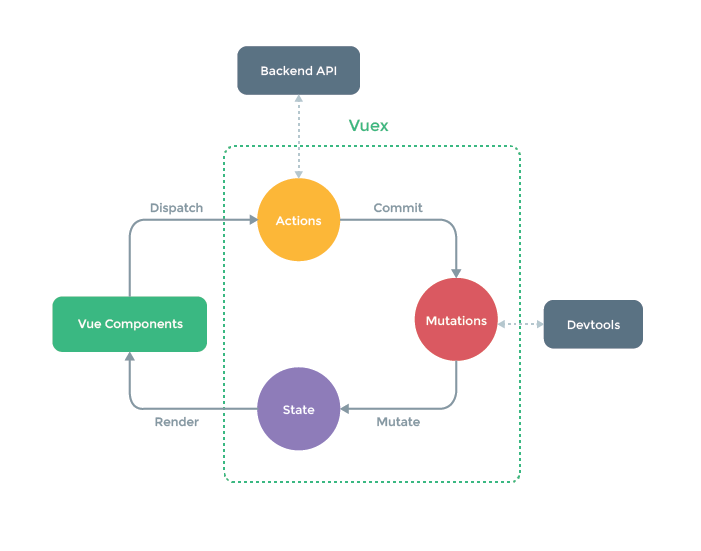
但是，当我们的应用遇到**多个组件共享状态**时，单向数据流的简洁性很容易被破坏：

* 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。

我们为什么不把组件的共享状态抽取出来，以一个全局单例模式管理呢？

在这种模式下，**我们的组件树构成了一个巨大的“视图”**，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**管理中的各种概念并强制遵守一定的规则，我们的代码将会变得更结构化且易维护。



1. 什么情况系使用Vuex

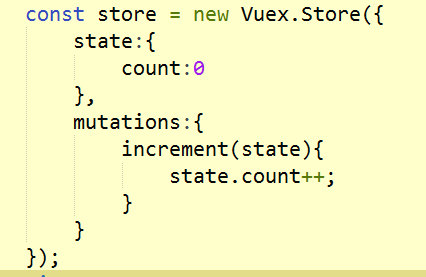
如果您需要构建是一个中大型单页应用，您很可能会考虑如何更好地在组件外部管理状态，Vuex 将会成为自然而然的选择。引用 Redux 的作者 Dan Abramov 的话说就是：

**Flux 架构就像眼镜：您自会知道什么时候需要它.**

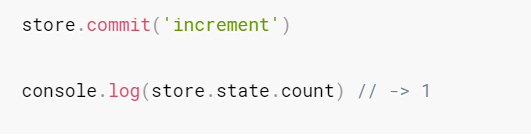
1. 开始

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。
3. 最简单的Store



可以通过 store.state 来获取状态对象，以及通过 store.commit 方法触发状态变更：



备注：

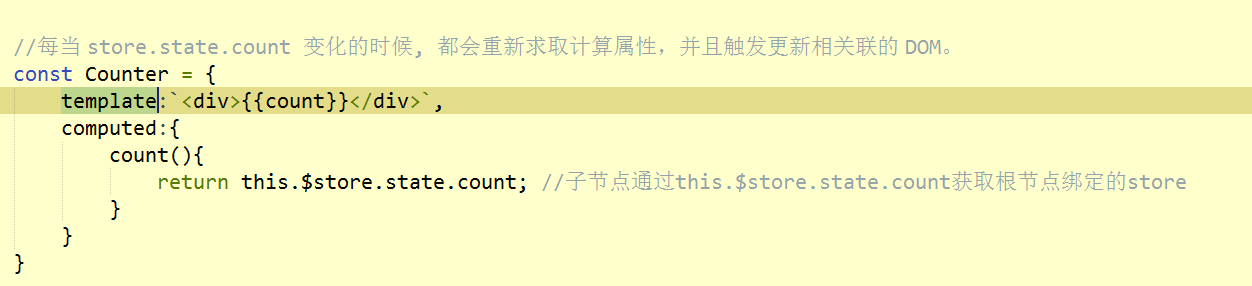
1. 通过提交 mutation 的方式，而非直接改变 store.state.count，是因为我们想要更明确地追踪到状态的变化。
2. 让我们有机会去实现一些能记录每次状态改变，保存状态快照的调试工具。有了它，我们甚至可以实现如时间穿梭般的调试体验。
3. **由于 store 中的状态是响应式的，在组件中调用 store 中的状态简单到仅需要在计算属性中返回即可。触发变化也仅仅是在组件的 methods 中提交 mutation。**
4. State

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状态。至此它便作为一个“唯一数据源 ([**SSOT**](https://en.wikipedia.org/wiki/Single_source_of_truth))”而存在。**每个应用将仅仅包含一个 store 实例。**单一状态树让我们能够直接地定位任一特定的状态片段，在调试的过程中也能轻易地取得整个当前应用状态的快照。

在Vue组件中获取Vuex状态

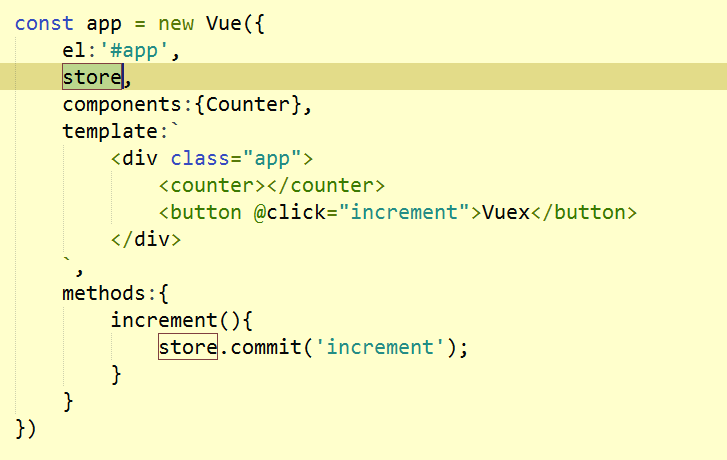
由于 Vuex 的状态存储是响应式的，从 store 实例中读取状态最简单的方法就是在[**计算属性**](https://cn.vuejs.org/guide/computed.html)中返回某个状态：



每当 store.state.count 变化的时候, 都会重新求取计算属性，并且触发更新相关联的 DOM。

**然而，这种模式导致组件依赖全局状态单例。在模块化的构建系统中，在每个需要使用 state 的组件中需要频繁地导入，并且在测试组件时需要模拟状态。**

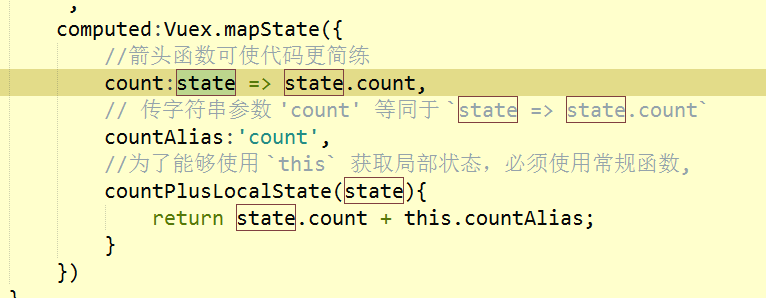
**Vuex 通过 store 选项，提供了一种机制将状态从根组件“注入”到每一个子组件中（需调用 Vue.use(Vuex)）**：



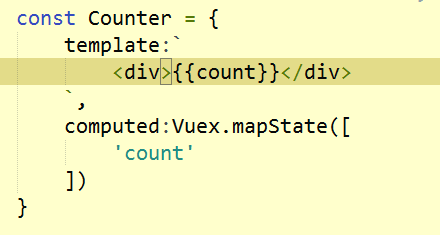
通过在根实例中注册 store 选项，该 store 实例会注入到根组件下的所有子组件中，且子组件能通过 this.$store访问到。让我们更新下 Counter 的实现：

**mapState辅助函数**

当一个组件需要获取多个状态时候，将这些状态都声明为计算属性会有些重复和冗余。为了解决这个问题，我们可以使用 mapState 辅助函数帮助我们生成计算属性，让你少按几次键：



当映射的计算属性的名称与 state 的子节点名称相同时，我们也可以给 mapState 传一个字符串数组。

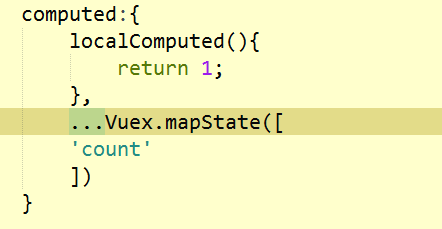


**对象展开运算符**

mapState 函数返回的是一个对象。我们如何将它与局部计算属性混合使用呢？

通常，我们需要使用一个工具函数将多个对象合并为一个，以使我们可以将最终对象传给 computed 属性。

mapState 函数返回的是一个对象。我们如何将它与局部计算属性混合使用呢？通常，我们需要使用一个工具函数将多个对象合并为一个，以使我们可以将最终对象传给 computed 属性。但是自从有了[**对象展开运算符**](https://github.com/sebmarkbage/ecmascript-rest-spread)（现处于 ECMASCript 提案 stage-3 阶段），我们可以极大地简化写法：

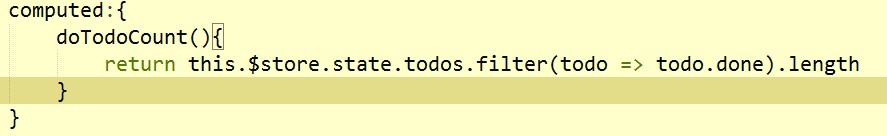


**组件仍然保有局部状态**

使用 Vuex 并不意味着你需要将**所有的**状态放入 Vuex。虽然将所有的状态放到 Vuex 会使状态变化更显式和易调试，但也会使代码变得冗长和不直观。如果有些状态严格属于单个组件，最好还是作为组件的局部状态。你应该根据你的应用开发需要进行权衡和确定。

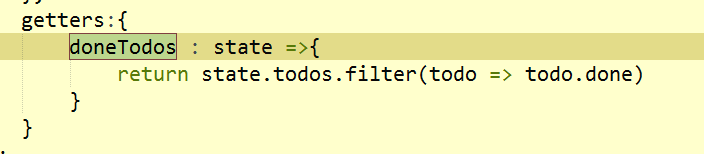
1. Getter

有时候我们需要从 store 中的 state 中派生出一些状态，例如对列表进行过滤并计数：



如果有多个组件需要用到此属性，我们要么复制这个函数，或者抽取到一个共享函数然后在多处导入它——无论哪种方式都不是很理想。

Vuex 允许我们在 store 中定义“getter”（可以认为是 store 的计算属性）。就像计算属性一样，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。

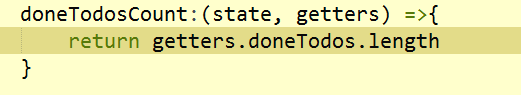


Getter 会暴露为 store.getters 对象：

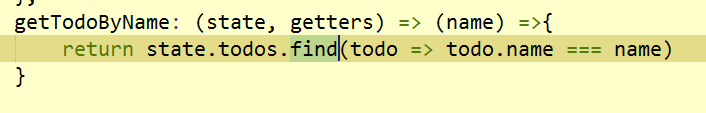


Getter 也可以接受其他 getter 作为第二个参数：

接受getters作为第二个参数



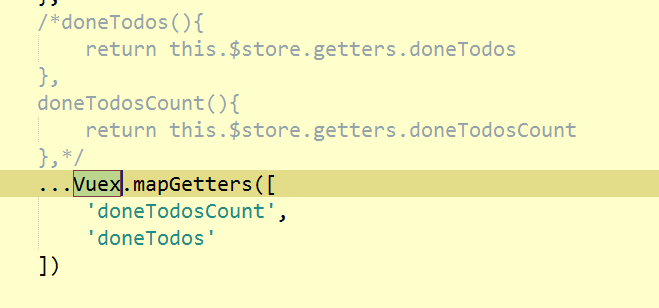
让 getter 返回一个函数，来实现给 getter 传参，在你对 store 里的数组进行查询时非常有用

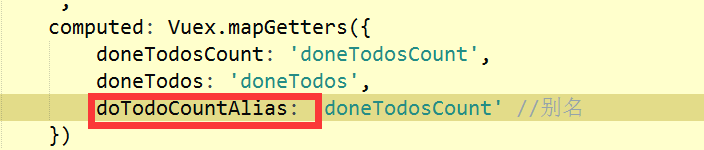




mapGetters 辅助函数

mapGetters 辅助函数仅仅是将 store 中的 getter 映射到局部计算属性：

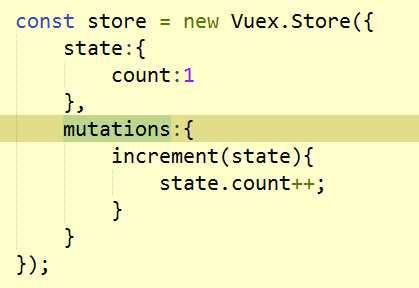




1. mutation

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

Vuex 中的 mutation 非常类似于事件：**每个 mutation 都有一个字符串的 事件类型 (type) 和 一个 回调函数 (handler)。**这个回调函数就是我们实际进行状态更改的地方，并且它会接受 state 作为第一个参数：

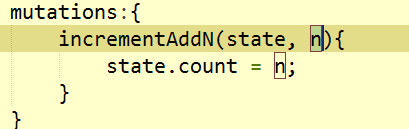


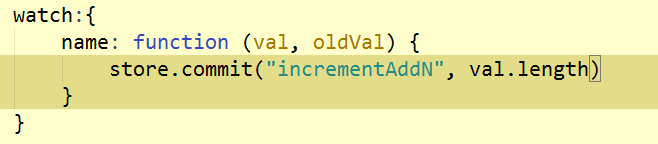
你不能直接调用一个 mutation handler。这个选项更像是事件注册：“当触发一个类型为 increment 的 mutation 时，调用此函数。”要唤醒一个 mutation handler，你需要以相应的 type 调用 **store.commit** 方法：



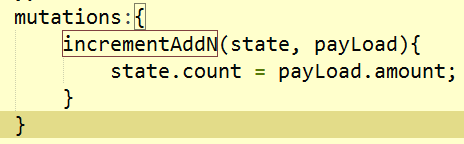
提交载荷（PayLoad）

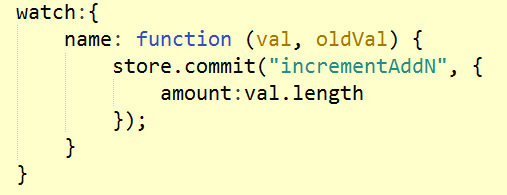
你可以向 store.commit 传入额外的参数，即 mutation 的 **载荷（payload）**：





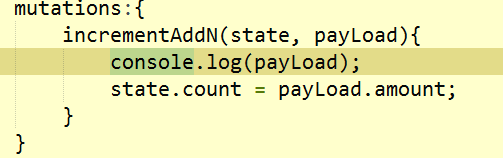
在大多数情况下，**载荷应该是一个对象**，这样可以包含多个字段并且记录的 mutation 会更易读：

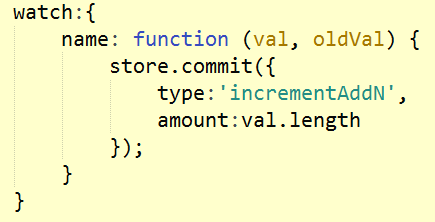




1. 对象风格的提交方式

提交 mutation 的另一种方式是直接使用包含 type 属性的对象：





Mutation 需遵守Vue的响应规则

既然 Vuex 的 store 中的状态是响应式的，那么当我们变更状态时，监视状态的 Vue 组件也会自动更新。这也意味着 Vuex 中的 mutation 也需要与使用 Vue 一样遵守一些注意事项：

1. 最好提前在你的 store 中初始化好所有所需属性。
2. 当需要在对象上添加新属性时，你应该
   1. 使用 Vue.set(obj, 'newProp', 123), 或者
   2. 以新对象替换老对象



使用常量替代Mutation事件类型

使用常量替代 mutation 事件类型在各种 Flux 实现中是很常见的模式。这样可以使 linter 之类的工具发挥作用，同时把这些常量放在单独的文件中可以让你的代码合作者对整个 app 包含的 mutation 一目了然：

//mutation-types.js



用不用常量取决于你——在需要多人协作的大型项目中，这会很有帮助。但如果你不喜欢，你完全可以不这样做。

Mutation 必须是同步函数

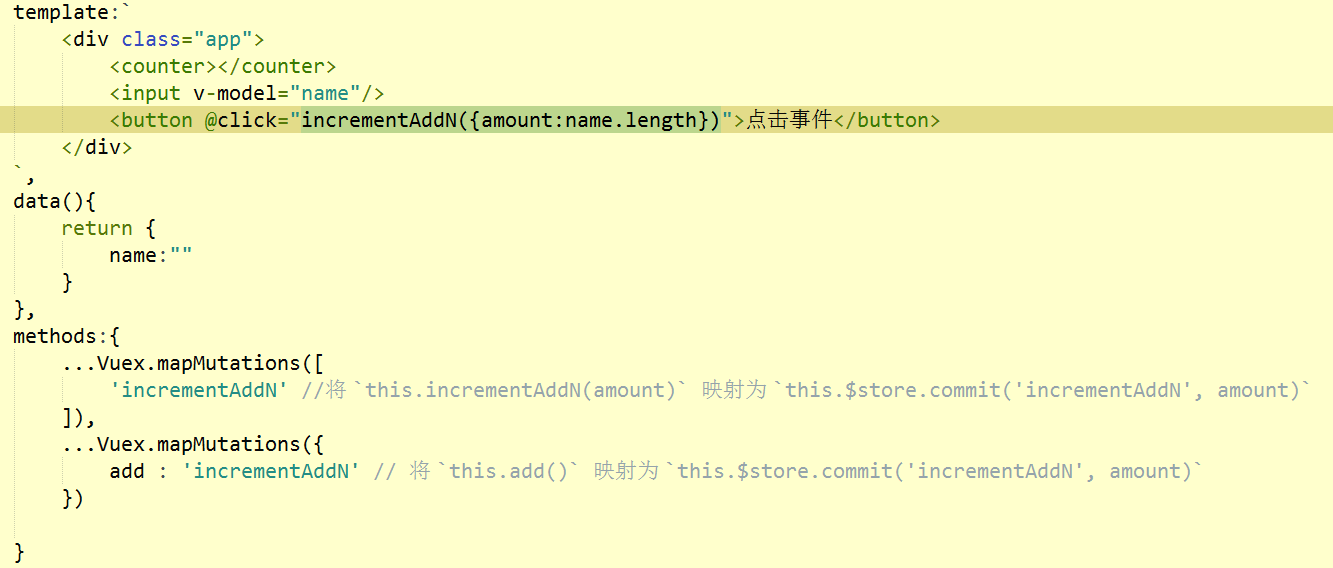
一条重要的原则就是要记住 **mutation 必须是同步函数**。为什么？请参考下面的例子：



现在想象，我们正在 debug 一个 app 并且观察 devtool 中的 mutation 日志。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。然而，在上面的例子中 mutation 中的异步函数中的回调让这不可能完成：因为当 mutation 触发的时候，回调函数还没有被调用，devtools 不知道什么时候回调函数实际上被调用——实质上任何在回调函数中进行的的状态的改变都是不可追踪的。

在组件中提交Mutation

你可以在组件中使用 this.$store.commit('xxx') 提交 mutation，或者使用 mapMutations 辅助函数将组件中的 methods 映射为 store.commit 调用（需要在根节点注入 store）。



当你能调用了两个包含异步回调的 mutation 来改变状态，你怎么知道什么时候回调和哪个先回调呢？这就是为什么我们要区分这两个概念。在 Vuex 中，**mutation 都是同步事务**：

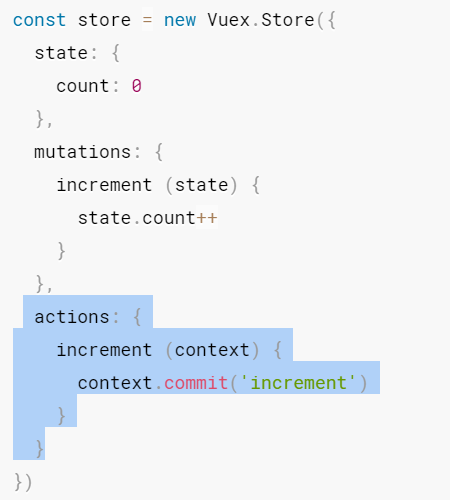


1. Action

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。

让我们来注册一个简单的 action：



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**](https://vuex.vuejs.org/zh-cn/modules.html)时，你就知道 context 对象为什么不是 store 实例本身了。

//ES6 参数解构语法



分发Action

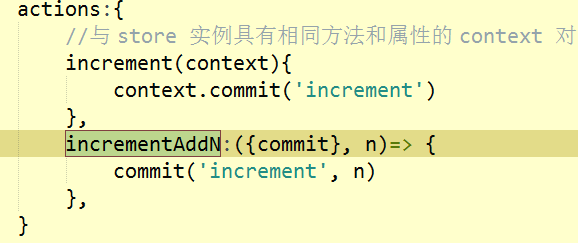


乍一眼看上去感觉多此一举，我们直接分发 mutation 岂不更方便？实际上并非如此，还记得 **mutation 必须同步执行**这个限制么？Action 就不受约束！我们可以在 action 内部执行**异步**操作：



Actions 支持同样的载荷方式和对象方式进行分发：

直接参数：





对象方式：





Type类型：

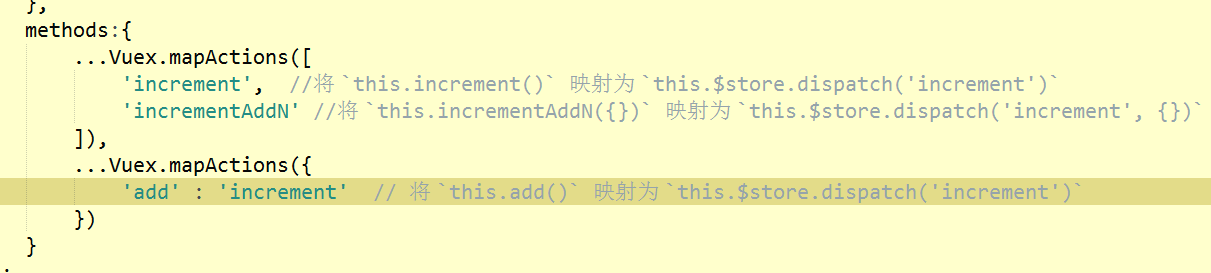


调用异步API和分发多重mutation：



注意我们正在进行一系列的异步操作，并且通过提交 mutation 来记录 action 产生的副作用（即状态变更）。

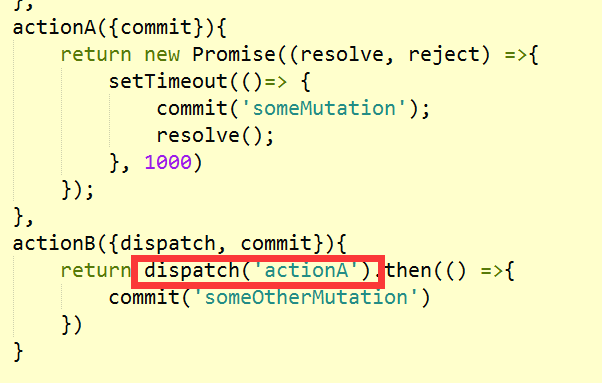
在组件中分发Action



组合Action

Action 通常是异步的，那么如何知道 action 什么时候结束呢？更重要的是，我们如何才能组合多个 action，以处理更加复杂的异步流程？

首先，你需要明白 store.dispatch 可以处理被触发的 action 的处理函数返回的 Promise，并且 store.dispatch 仍旧返回 Promise：



**一个 store.dispatch 在不同模块中可以触发多个 action 函数。在这种情况下，只有当所有触发函数完成后，返回的 Promise 才会执行。**