一. 路由（Vue2）

1. SPA 与前端路由

路由是根据不同的url地址来显示不同的页面或内容的功能，这个概念很早是由后端提出的，既浏览器向不同的地址发送请求，后端返回相应的内容。

SPA 指的是一个 web 网站只有唯一的一个 HTML 页面，所有组件的展示与切换都在这唯一的一个页面内完成。此时，不同组件之间的切换需要通过前端路由来实现。

前端路由通常是通过监听URL hash属性值的变化，切换页面，hash 属性是一个`可读可写的字符串`，该字符串是 URL 的锚部分（从 # 号开始的部分）。

前端路由的工作方式

前端路由，指的是 Hash 地址与组件之间的对应关系。

- 用户点击了页面上的路由链接
- 导致了 URL 地址栏中的 Hash 值发生了变化
- 前端路由监听了到 Hash 地址的变化
- 前端路由把当前 Hash 地址对应的组件渲染都浏览器中

2. vue-router基本使用

vue-router 是 vue.js 官方给出的路由解决方案。它只能结合 vue 项目进行使用，能够轻松的管理 SPA 项目中组件的切换。

vue-router 目前有 3.x 的版本和 4.x 的版本。其中：

- vue-router 3.x 只能结合 vue2 进行使用
- vue-router 4.x 只能结合 vue3 进行使用

官方文档：https://router.vuejs.org/zh/installation.html

vue-router的基本使用步骤：

- 在项目中安装 vue-router
- 定义路由组件
- 声明路由链接和占位符
- 创建路由模块
- 导入并挂载路由模块

vue-router安装

```
npm install vue-router@3
```

创建路由组件

在项目中定义 Discover.vue、Friends.vue、My.vue 三个组件，将来要使用 vue-router 来控制它们的展示与切换：

Discover.vue：

```
<template>
    <div>
        <h1>发现音乐</h1>
    </div>
</template>
```

Friends.vue：

```
<template>
    <div>
        <h1>关注</h1>
    </div>
</template>
```

My.vue：

```
<template>
    <div>
        <h1>我的音乐</h1>
    </div>
</template>
```

声明路由链接和占位标签

可以使用 `<router-link>` 标签来声明路由链接，并使用 `<router-view>` 标签来声明路由占位符。示例代码如下：

App.vue：

```
<template>
  <div>
    <h1>APP组件</h1>
    <!-- 声明路由链接 -->
    <router-link to="/discover">发现音乐</router-link>
    <router-link to="/my">我的音乐</router-link>
    <router-link to="/friend">关注</router-link>
    <!-- 声明路由占位标签 -->
    <router-view></router-view>
  </div>
</template>
```

创建路由模块

在项目中创建 index.js 路由模块，加入以下代码：

```
import VueRouter from 'vue-router'
import Vue from 'vue'
import Discover from '@/components/Discover.vue'
import Friends from '@/components/Friends.vue'
import My from '@/components/My.vue'

//将VueRouter设置为Vue的插件
Vue.use(VueRouter)
```

```
const router = new VueRouter({
     // 指定hash属性与组件的对应关系
     routes: [
        { path: '/discover', component: Discover },
        { path: '/friends', component: Friends },
        { path: '/my', component: My },
    ]
})

export default router
```

挂载路由模块

在main.js中导入并挂载router

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

Vue.config.productionTip = false

new Vue({
  render: h => h(App),
  router:router
}).$mount('#app')
```

3. vue-router进阶

路由重定向

路由重定向指的是：用户在访问地址 A 的时候，强制用户跳转到地址 C ，从而展示特定的组件页面。

通过路由规则的 redirect 属性，指定一个新的路由地址，可以很方便地设置路由的重定向：

```
const router = new VueRouter({
     // 指定hash属性与组件的对应关系
     routes: [
         // 当用户访问 / 时，跳转到 /discover
        { path: '/', redirect: '/discover'},
        { path: '/discover', component: Discover },
        { path: '/friends', component: Friends },
        { path: '/my', component: My },
    ]
})
```

嵌套路由

在 Discover.vue 组件中，声明 toplist和 playlist的子路由链接以及子路由占位符。示例代码如下：

```
<template>
    <div>
        <h1>发现音乐</h1>
        <!-- 子路由链接 -->
        <router-link to="/discover/toplist">推荐</router-link>
        <router-link to="/discover/playlist">歌单</router-link>
        <hr>
        <router-view></router-view>
    </div>
</template>
```

在 src/router/index.js 路由模块中，导入需要的组件，并使用 children 属性声明子路由规则：

```
const router = new VueRouter({
     // 指定hash属性与组件的对应关系
     routes: [
        { path: '/', redirect: '/discover'},
        {
            path: '/discover',
            component: Discover,
            // 通过children属性，嵌套声明子路由
            children: [
                {path:"toplist",component:TopList},
                {path:"playlist",component:PlayList},
            ]

        },
        { path: '/friends', component: Friends },
        { path: '/my', component: My },
    ]
})
```

动态路由

思考：有如下 3 个路由链接：

```
<router-link to="/product/1">商品1</router-link>
<router-link to="/product/2">商品2</router-link>
<router-link to="/product/3">商品3</router-link>
```

```
const router = new VueRouter({
    // 指定hash属性与组件的对应关系
    routes: [
        { path: '/product/1', component: Product },
        { path: '/product/2', component: Product },
        { path: '/product/3', component: Product },
    ]
})
```

上述方式复用性非常差。

动态路由指的是：把 Hash 地址中可变的部分定义为参数项，从而提高路由规则的复用性。在 vue-router 中使用英文的冒号（:）来定义路由的参数项。示例代码如下：

```
{ path: '/product/:id',component:Product }
```

通过动态路由匹配的方式渲染出来的组件中，可以使用 $route.params 对象访问到动态匹配的参数值，**比如在商品详情组件的内部，根据id值，请求不同的商品数据。**

```
<template>
    <h1>Product组件</h1>
    <!-- 获取动态的id值 -->
    <p>{{$route.params.id}}</p>
</template>

<script>
export default {
//   组件的名称
  name: 'Product'
}
</script>
```

为了简化路由参数的获取形式，vue-router 允许在路由规则中开启 props 传参。示例代码如下：

```
{ path: '/product/:id',component: Product, props: true }
```

```
<template>
    <h1>Product组件</h1>
    <!-- 获取动态的id值 -->
    <p>{{id}}</p>
</template>

<script>
export default {
//   组件的名称
  name: 'Product',
  props : [id]
}
</script>
```

编程式导航

声明式	编程式
`<router-link :to="...">`	`router.push(...)`

除了使用 `<router-link>` 创建 a 标签来定义导航链接，我们还可以借助 router 的实例方法，通过编写代码来实现。

想要导航到不同的 URL，则使用 `router.push` 方法。这个方法会向 history 栈添加一个新的记录，所以，当用户点击浏览器后退按钮时，则回到之前的 URL。

当你点击 `<router-link>` 时，这个方法会在内部调用，所以说，点击 `<router-link :to="...">` 等同于调用 `router.push(...)`。

```
<template>
    <button @click="gotoProduct(2)">跳转到商品2</button>
</template>

<script>
export default {
    methods:{
        gotoProduct: function(id){
            this.$router.push('/movie/${id}')
        }
    }
}
</script>
```

导航守卫

导航守卫可以控制路由的访问权限。示意图如下：

全局导航守卫会拦截每个路由规则，从而对每个路由进行访问权限的控制。

你可以使用 `router.beforeEach` 注册一个全局前置守卫：

```
router.beforeEach((to, from, next) => {
    if (to.path === '/main' && !isAuthenticated) {
        next('/login')
    }
    else {
        next()
    }
})
```

- `to`: 即将要进入的目标
- `from`: 当前导航正要离开的路由
- 在守卫方法中如果声明了 **next** 形参，则必须调用 next() 函数，否则不允许用户访问任何一个路由！
    - 直接放行：next()
    - 强制其停留在当前页面：next(false)
    - 强制其跳转到登录页面：next('/login')

二. 状态管理

1. 简单状态管理

经常被忽略的是，Vue 应用中响应式 `data` 对象的实际来源——当访问数据对象时，一个组件实例只是简单的代理访问。所以，如果你有一处需要被多个实例间共享的状态，你可以使用一个 reactive 方法让对象作为响应式对象。

```
const { createApp, reactive } = Vue
const sourceOfTruth = reactive({
  message: 'Hello'
})
```

```
const appA = createApp({
  data() {
    return sourceOfTruth
  }
}).mount('#app-a')

const appB = createApp({
  data() {
    return sourceOfTruth
  }
}).mount('#app-b')
```

```
<div id="app-a">App A: {{ message }}</div>

<div id="app-b">App B: {{ message }}</div>
```

现在当 `sourceOfTruth` 发生变更，`appA` 和 `appB` 都将自动地更新它们的视图。虽然现在我们有了一个真实数据来源，但调试将是一场噩梦。应用的任何部分都可以随时更改任何数据，而不会留下变更过的记录。

```
const appB = createApp({
  data() {
    return sourceOfTruth
  },
  mounted() {
    sourceOfTruth.message = 'Goodbye' // both apps will render 'Goodbye' message
now
  }
}).mount('#app-b')
```

为了解决这个问题，可以采用一个简单的 **store 模式**：

```
const store = {
  debug: true,

  state: reactive({
    message: 'Hello!'
  }),

  setMessageAction(newValue) {
    if (this.debug) {
      console.log('setMessageAction triggered with', newValue)
    }

    this.state.message = newValue
  },

  clearMessageAction() {
    if (this.debug) {
      console.log('clearMessageAction triggered')
    }
```

```
    this.state.message = ''
  }
}
```

需要注意，所有 store 中 state 的变更，都放置在 store 自身的 action 中去管理。这种集中式状态管理能够被更容易地理解哪种类型的变更将会发生，以及它们是如何被触发。当错误出现时，现在也会有一个 log 记录 bug 之前发生了什么。

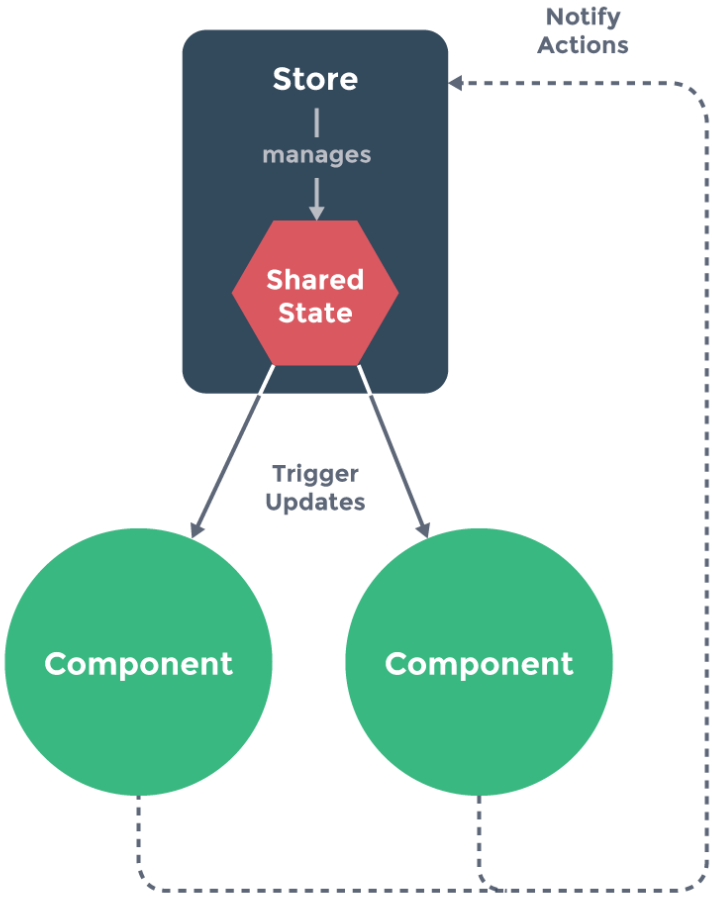
此外，每个实例/组件仍然可以拥有和管理自己的私有状态：

```
<div id="app-a">{{sharedState.message}}</div>

<div id="app-b">{{sharedState.message}}</div>
```

```
const appA = createApp({
  data() {
    return {
      privateState: {},
      sharedState: store.state
    }
  },
  mounted() {
    store.setMessageAction('Goodbye!')
  }
}).mount('#app-a')

const appB = createApp({
  data() {
    return {
      privateState: {},
      sharedState: store.state
    }
  }
}).mount('#app-b')
```



随着我们进一步扩展约定，即组件不允许直接变更属于 store 实例的 state，而应执行 action 来分发 (dispatch) 事件通知 store 去改变，最终达成了 Flux 架构。这样约定的好处是，能够记录所有 store 中发生的 state 变更，同时实现能做到记录变更、保存状态快照、历史回滚/时光旅行的先进的调试工具。

2. Vuex介绍

由于状态零散地分布在许多组件和组件之间的交互中，大型应用复杂度也经常逐渐增长。为了解决这个问题，Vue 提供 Vuex：我们有受到 Elm 启发的状态管理库。

Vuex 是一个专为 Vue.js 应用程序开发的**状态管理模式 + 库**。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。

示例

让我们从一个简单的 Vue 计数应用开始：

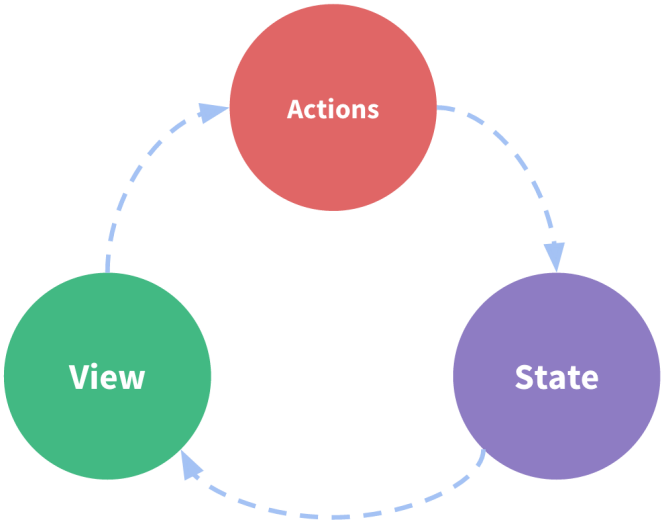
```
const Counter = {
  // 状态
  data () {
    return {
      count: 0
    }
  },
```

```
  // 视图
  template: `
    <div>{{ count }}</div>
  `,
  // 操作
  methods: {
    increment () {
      this.count++
    }
  }
}
createApp(Counter).mount('#app')
```

这个状态自管理应用包含以下几个部分：

- **状态**，驱动应用的数据源；
- **视图**，以声明方式将**状态**映射到视图；
- **操作**，响应在**视图**上的用户输入导致的状态变化。

以下是一个表示“单向数据流”理念的简单示意：



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。

对于问题一，传参的方法对于多层嵌套的组件将会非常繁琐，**并且对于兄弟组件间的状态传递无能为力。**

对于问题二，我们经常会采用父子组件直接引用或者通过事件来变更和同步状态的多份拷贝。以上的这些模式非常脆弱，通常会导致无法维护的代码。

因此，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通过强制规则维持视图和状态间的独立性，我们的代码将会变得更结构化且易维护。

这就是 Vuex 背后的基本思想，借鉴了 Flux、Redux 和 The Elm Architecture。与其他模式不同的是，Vuex 是专门为 Vue.js 设计的状态管理库，以利用 Vue.js 的细粒度数据响应机制来进行高效的状态更新。

什么情况下应该使用 Vuex?

Vuex 可以帮助我们管理共享状态，并附带了更多的概念和框架。这需要对短期和长期效益进行权衡。

如果您不打算开发大型单页应用，使用 Vuex 可能是繁琐冗余的。确实是如此——如果您的应用够简单，您最好不要使用 Vuex。一个简单的 store 模式就足够您所需了。但是，如果您需要构建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。

安装

```
npm install vuex@3
```

3. 最简单的 Store

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。

安装Vuex 之后，让我们来创建一个 store。创建过程直截了当——仅需要提供一个初始 state 对象和一些 mutation：

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

Vue.use(Vuex)

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

现在，你可以通过 `store.state` 来获取状态对象，并通过 `store.commit` 方法触发状态变更：

```
store.commit('increment')

console.log(store.state.count) // -> 1
```

为了在 Vue 组件中访问 `this.$store` property，你需要为 Vue 实例提供创建好的 store。Vuex 提供了一个从根组件向所有子组件，以 `store` 选项的方式“注入”该 store 的机制：

```
new Vue({
  el: '#app',
  store: store,
})
```

在 Vue 组件中， 可以通过 `this.$store` 访问store实例。现在我们可以从组件的方法提交一个变更：

```
methods: {
  increment() {
    this.$store.commit('increment')
    console.log(this.$store.state.count)
  }
}
```

再次强调，我们通过提交 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。这个简单的约定能够让你的意图更加明显，这样你在阅读代码的时候能更容易地解读应用内部的状态改变。此外，这样也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。

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

Vuex 使用**单一状态树**——是的，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)”而存在。这也意味着，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。

单状态树和模块化并不冲突——在后面的章节里我们会讨论如何将状态和状态变更事件分布到各个子模块中。

在 Vue 组件中获得 Vuex 状态

那么我们如何在 Vue 组件中展示状态呢？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中返回某个状态：

```
// 创建一个 Counter 组件
const Counter = {
  template: `<div>{{ count }}</div>`,
  computed: {
    count () {
      return store.state.count
    }
  }
}
```

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 Vue 的插件系统将 store 实例从根组件中“注入”到所有的子组件里。且子组件能通过 `this.$store` 访问到。让我们更新下 `Counter` 的实现：

```
const Counter = {
  template: `<div>{{ count }}</div>`,
  computed: {
    count () {
      return this.$store.state.count
    }
  }
}
```

`mapState` 辅助函数

当一个组件需要获取多个状态的时候，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：

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

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

    // 为了能够使用 `this` 获取局部状态，必须使用常规函数
    countPlusLocalState (state) {
      return state.count + this.localCount
    }
  })
}
```

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。

```
computed: mapState([
  // 映射 this.count 为 store.state.count
  'count'
])
```

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。但是自从有了对象展开运算符，我们可以极大地简化写法：

```
computed: {
  localComputed () { /* ... */ },
  // 使用对象展开运算符将此对象混入到外部对象中
  ...mapState({
    // ...
  })
}
```

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。

5. 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：

```
computed: {
  doneTodosCount () {
    return this.$store.state.todos.filter(todo => todo.done).length
  }
}
```

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）。

Getter 接受 state 作为其第一个参数：

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

通过属性访问

Getter 会暴露为 `store.getters` 对象，你可以以属性的形式访问这些值：

```
store.getters.doneTodos // -> [{ id: 1, text: '...', done: true }]
```

Getter 也可以接受其他 getter 作为第二个参数：

```
getters: {
  // ...
  doneTodosCount (state, getters) {
    return getters.doneTodos.length
  }
}
```

```
store.getters.doneTodosCount // -> 1
```

我们可以很容易地在任何组件中使用它：

```
computed: {
  doneTodosCount () {
    return this.$store.getters.doneTodosCount
  }
}
```

通过方法访问

你也可以通过让 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。

```
getters: {
  // ...
  getTodoById: (state) => (id) => {
    return state.todos.find(todo => todo.id === id)
  }
}
```

```
store.getters.getTodoById(2) // -> { id: 2, text: '...', done: false }
```

`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：

```
import { mapGetters } from 'vuex'

export default {
  // ...
  computed: {
  // 使用对象展开运算符将 getter 混入 computed 对象中
    ...mapGetters([
      'doneTodosCount',
      'anotherGetter',
      // ...
    ])
  }
}
```

如果你想将一个 getter 属性另取一个名字，使用对象形式：

```
...mapGetters({
  // 把 `this.doneCount` 映射为 `this.$store.getters.doneTodosCount`
  doneCount: 'doneTodosCount'
})
```

6. 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：

```
const store = createStore({
  state: {
    count: 1
  },
  mutations: {
    increment (state) {
      // 变更状态
      state.count++
    }
  }
})
```

你不能直接调用一个 mutation 处理函数。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 处理函数，你需要以相应的 type 调用 **store.commit** 方法：

```
store.commit('increment')
```

提交载荷（Payload）

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

```
// ...
mutations: {
  increment (state, n) {
    state.count += n
  }
}
```

```
store.commit('increment', 10)
```

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：

```
// ...
mutations: {
  increment (state, payload) {
    state.count += payload.amount
  }
}
```

```
store.commit('increment', {
  amount: 10
})
```

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：

```
store.commit({
  type: 'increment',
  amount: 10
})
```

当使用对象风格的提交方式，整个对象都作为载荷传给 mutation 函数，因此处理函数保持不变：

```
mutations: {
  increment (state, payload) {
    state.count += payload.amount
  }
}
```

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
export const SOME_MUTATION = 'SOME_MUTATION'
```

```
// store.js
import { createStore } from 'vuex'
import { SOME_MUTATION } from './mutation-types'

const store = createStore({
  state: { ... },
  mutations: {
    // 我们可以使用 ES2015 风格的计算属性命名功能来使用一个常量作为函数名
    [SOME_MUTATION] (state) {
      // 修改 state
    }
  }
})
```

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：

```
mutations: {
  someMutation (state) {
    api.callAsyncMethod(() => {
      state.count++
    })
  }
}
```

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`）。

```
import { mapMutations } from 'vuex'

export default {
  // ...
  methods: {
    ...mapMutations([
      'increment', // 将 `this.increment()` 映射为
`this.$store.commit('increment')`

      // `mapMutations` 也支持载荷：
      'incrementBy' // 将 `this.incrementBy(amount)` 映射为
`this.$store.commit('incrementBy', amount)`
    ]),
    ...mapMutations({
      add: 'increment' // 将 `this.add()` 映射为 `this.$store.commit('increment')`
    })
  }
}
```

在 mutation 中混合异步调用会导致你的程序很难调试。例如，当你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**：

```
store.commit('increment')
// 任何由 "increment" 导致的状态变更都应该在此刻完成。
```

为了处理异步操作，让我们来看一看Action。

7. 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：

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

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。

实践中，我们会经常用到 ES2015 的参数解构来简化代码（特别是我们需要调用 `commit` 很多次的时候）：

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

分发 Action

Action 通过 `store.dispatch` 方法触发：

```
store.dispatch('increment')
```

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

Actions 支持同样的载荷方式和对象方式进行分发：

```
// 以载荷形式分发
store.dispatch('incrementAsync', {
  amount: 10
})

// 以对象形式分发
store.dispatch({
  type: 'incrementAsync',
  amount: 10
})
```

来看一个更加实际的购物车示例，涉及到**调用异步 API** 和**分发多重 mutation**：

```
actions: {
  checkout ({ commit, state }, products) {
    // 把当前购物车的物品备份起来
    const savedCartItems = [...state.cart.added]
    // 发出结账请求，然后乐观地清空购物车
    commit(types.CHECKOUT_REQUEST)
    // 购物 API 接受一个成功回调和一个失败回调
    shop.buyProducts(
      products,
      // 成功操作
      () => commit(types.CHECKOUT_SUCCESS),
      // 失败操作
      () => commit(types.CHECKOUT_FAILURE, savedCartItems)
    )
  }
}
```

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

你在组件中使用 `this.$store.dispatch('xxx')` 分发 action，或者使用 `mapActions` 辅助函数将组件的 methods 映射为 `store.dispatch` 调用（需要先在根节点注入 `store`）：

```
import { mapActions } from 'vuex'

export default {
  // ...
  methods: {
```

```
    ...mapActions([
      'increment', // 将 `this.increment()` 映射为
`this.$store.dispatch('increment')`

      // `mapActions` 也支持载荷：
      'incrementBy' // 将 `this.incrementBy(amount)` 映射为
`this.$store.dispatch('incrementBy', amount)`
    ]),
    ...mapActions({
      add: 'increment' // 将 `this.add()` 映射为
`this.$store.dispatch('increment')`
    })
  }
}
```

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：

```
actions: {
  actionA ({ commit }) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        commit('someMutation')
        resolve()
      }, 1000)
    })
  }
}
```

现在你可以：

```
store.dispatch('actionA').then(() => {
  // ...
})
```

在另外一个 action 中也可以：

```
actions: {
  // ...
  actionB ({ dispatch, commit }) {
    return dispatch('actionA').then(() => {
      commit('someOtherMutation')
    })
  }
}
```

最后，如果我们利用 async / await，我们可以如下组合 action：

```
// 假设 getData() 和 getOtherData() 返回的是 Promise

actions: {
  async actionA ({ commit }) {
    commit('gotData', await getData())
  },
  async actionB ({ dispatch, commit }) {
    await dispatch('actionA') // 等待 actionA 完成
    commit('gotOtherData', await getOtherData())
  }
}
```

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。

8. 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——从上至下进行同样方式的分割：

```
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
```

模块的局部状态

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**。

```
const moduleA = {
  state: () => ({
    count: 0
  }),
```

```
  mutations: {
    increment (state) {
      // 这里的 `state` 对象是模块的局部状态
      state.count++
    }
  },
  getters: {
    doubleCount (state) {
      return state.count * 2
    }
  }
}
```

同样，对于模块内部的 action，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')
      }
    }
  }
}
```

对于模块内部的 getter，根节点状态会作为第三个参数暴露出来：

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

模块动态注册

在 store 创建**之后**，你可以使用 `store.registerModule` 方法注册模块

```
import { createStore } from 'vuex'

const store = createStore({ /* 选项 */ })

// 注册模块 `myModule`
store.registerModule('myModule', {
  // ...
})

// 注册嵌套模块 `nested/myModule`
store.registerModule(['nested', 'myModule'], {
  // ...
})
```

之后就可以通过 `store.state.myModule` 和 `store.state.nested.myModule` 访问模块的状态。