Vue全家桶-Vue-router&Vuex

Vue-Router

资料

- Vue-router
- Vuex

介绍

Vue Router 是 Vue.js 官方的路由管理器。它和 Vue.js 的核心深度集成，让构建单页面应用变得易如反掌。包含的功能有：

- 嵌套的路由/视图表
- 模块化的、基于组件的路由配置
- 路由参数、查询、通配符
- 基于 Vue.js 过渡系统的视图过渡效果
- 细粒度的导航控制
- 带有自动激活的 CSS class 的链接
- HTML5 历史模式或 hash 模式，在 IE9 中自动降级
- 自定义的滚动条行为

起步

用 Vue.js + Vue Router 创建单页应用，是非常简单的。使用 Vue.js ，我们已经可以通过组合组件来组成应用程序，当你要把 Vue Router 添加进来，我们需要做的是，将组件 (components) 映射到路由 (routes)，然后告诉 Vue Router 在哪里渲染它们

> 安装

> `npm i vue-router -S`
>
> 在main.js中
>
> ```
> import Vue from 'vue'
> import VueRouter from 'vue-router'
>
> Vue.use(VueRouter)
> ```
>
> 推荐使用:vue add router 添加插件(记得提前提交)

基本使用

router.js

```
import Vue from 'vue'
//1.导入
import Router from 'vue-router'
import Home from './views/Home.vue'
import About from './views/About.vue'
//2.模块化机制 使用Router
Vue.use(Router)

//3.创建路由器对象
const router = new Router({
    routes:[{
      path: '/home',
      component: Home
    },
    {
      path: '/about',
      component: About
    }
  ]
})
```

```
export default router;
```

main.js

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

Vue.config.productionTip = false

new Vue({
  // 4.挂载根实例
  router,
  render: h => h(App)
}).$mount('#app')

```

做好以上配置之后

App.vue

```
<template>
  <div id="app">
    <div id="nav">
      <!-- 使用router-link组件来导航 -->
      <!-- 通过传入to属性指定连接 -->
      <!-- router-link默认会被渲染成一个a标签 -->
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link> |
    </div>
    <!-- 路由出口 -->
    <!-- 路由匹配的组件将被渲染到这里 -->
    <router-view/>
  </div>
</template>
```

打开浏览器.,切换Home和About超链接,查看效果

命名路由

在配置路由的时候,给路由添加名字,访问时就可以动态的根据名字来进行访问

```
const router = new Router({
    routes:[{
      path: '/home',
      name:"home",
      component: Home
    },
    {
      path: '/about',
      name:'about'
      component: About
    }
  ]
})
```

要链接到一个命名路由，可以给 `router-link` 的 `to` 属性传一个对象：

```
<router-link :to="{name:'home'}">Home</router-link> |
<router-link :to="{name:'about'}">About</router-link> |
```

动态路由匹配

我们经常需要把某种模式匹配到的所有路由，全都映射到同个组件。例如，我们有一个 `User` 组件，对于所有 ID 各不相同的用户，都要使用这个组件来渲染。那么，我们可以在 `vue-router` 的路由路径中使用“动态路径参数”(dynamic segment) 来达到这个效果

User.vue

```
<template>
  <div>
    <h3>用户页面</h3>
  </div>
</template>

<script>
export default {
};
</script>

<style lang="scss" scoped>
</style>
```

路由配置

```
const router = new Router({
    routes:[
        {
            path: '/user/:id',
            name: 'user',
            component: User,
        },
    ]
})
```

```
<router-link :to="{name:'user',params:
{id:1}}">User</router-link> |
```

访问

http://localhost:8080/user/1

http://localhost:8080/user/2

查看效果

当匹配到路由时,参数值会被设置到this.$route.params,可以在每个组件中使用,于是，我们可以更新 `User` 的模板，输出当前用户的 ID：

```
<template>
  <div>
    <h3>用户页面{{$route.params.id}}</h3>
  </div>
</template>
```

响应路由参数的变化

提醒一下，当使用路由参数时，例如从 `/user/1 导航到`/user/2`，**原来的组件实例会被复用**。因为两个路由都渲染同个组件，比起销毁再创建，复用则显得更加高效。**不过，这也意味着组件的生命周期钩子不会再被调用**。

复用组件时，想对路由参数的变化作出响应的话，你可以简单地 watch (监测变化) `$route` 对象：

```
/*使用watch(监测变化) $route对象
 watch: {
        $route(to, from) {
            console.log(to.params.id);

        }
    }, */
// 或者使用导航守卫
beforeRouteUpdate(to,from,next){
    //查看路由的变化
    //一定要调用next,不然就会阻塞路由的变化
    next();
}
```

404路由

```
const router = new Router({
    routes:[
        //....
        // 匹配不到理由时,404页面显示
        {
            path: '*',
            component: () => import('@/views/404')
        }
    ]
})
```

当使用*通配符*路由时，请确保路由的顺序是正确的，也就是说含有***通配符*的路由应该放在最后**。路由 `{ path: '*' }` 通常用于客户端 404 错误

当使用一个*通配符*时，`$route.params` 内会自动添加一个名为 `pathMatch` 参数。它包含了 URL 通过*通配符*被匹配的部分：

```
{
    path: '/user-*',
    component: () => import('@/views/User-admin.vue')
}
this.$route.params.pathMatch // 'admin'
```

匹配优先级

有时候，同一个路径可以匹配多个路由，此时，匹配的优先级就按照路由的定义顺序：谁先定义的，谁的优先级就最高。

查询参数

类似像地址上出现的这种:http://localhos:8080/page?id=1&title=foo

```
const router = new Router({
    routes:[
        //....
        {
            name:'/page',
            name:'page',
            component:()=>import('@/views/Page.vue')
        }

    ]
})
```

```
 <router-link :to="{name:'page',query:
{id:1,title:'foo'}}">User</router-link> |
```

访问http://localhos:8080/page?id=1&title=foo查看Page

Page.vue

```
<template>
    <div>
        <h3>Page页面</h3>
        <h3>{{$route.query.userId}}</h3>
    </div>
</template>

<script>
    export default {
        created () {
            //查看路由信息对象
            console.log(this.$route);
        },
    }
</script>

<style lang="scss" scoped>

</style>
```

路由重定向和别名

例子是从 `/` 重定向到 `/home`：

```
const router = new Router({
    mode: 'history',
    routes: [
        // 重定向
        {
            path: '/',
            redirect: '/home'
        }
        {
        path: '/home',
```

```
        name: 'home',
        component: Home
        },
    ]
})
```

重定向的目标也可以是一个命名的路由：

```
const router = new VueRouter({
  routes: [
    { path: '/', redirect: { name: 'name' }}
  ]
})
```

别名

```
{
    path: '/user/:id',
    name: 'user',
    component: User,
    alias: '/alias'
}
```

起别名,仅仅起起别名 用户访问http://loacalhost:8080/alias的时候,显示User组件

> 别名”的功能让你可以自由地将 UI 结构映射到任意的 URL，而不是受限于配置的嵌套路由结构。

路由组件传参

在组件中使用 `$route` 会使之与其对应路由形成高度耦合，从而使组件只能在某些特定的 URL 上使用，限制了其灵活性。

使用 `props` 将组件和路由解耦：

取代与 $route 的耦合

```
{
      path: '/user/:id',
      name: 'user',
      component: User,
      props:true
},
```

User.vue

```
<template>
<div>
    <h3>用户页面{{$route.params.id}}</h3>
    <h3>用户页面{{id}}</h3>
    </div>
</template>
<script>
    export default{
        //....
        props: {
            id: {
                type: String,
                default: ''
            },
        },
    }
</script>
```

props也可以是个函数

```
{
      path: '/user/:id',
      name: 'user',
      component: User,
      props: (route)=>({
        id: route.params.id,
        title:route.query.title
      })

}
```

User.vue

```
<template>
  <div>
    <h3>用户页面{{id}}-{{title}}</h3>
  </div>
</template>

<script>
export default {
    // ...
    props: {
        id: {
            type: String,
            default: ''
        },
        title:{
            type:String
        }
    },
};
</script>

```

编程式导航

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。

> **注意：在 Vue 实例内部，你可以通过 $router 访问路由实例。因此你可以调用 this.$router.push。**

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

该方法的参数可以是一个字符串路径，或者一个描述地址的对象。例如

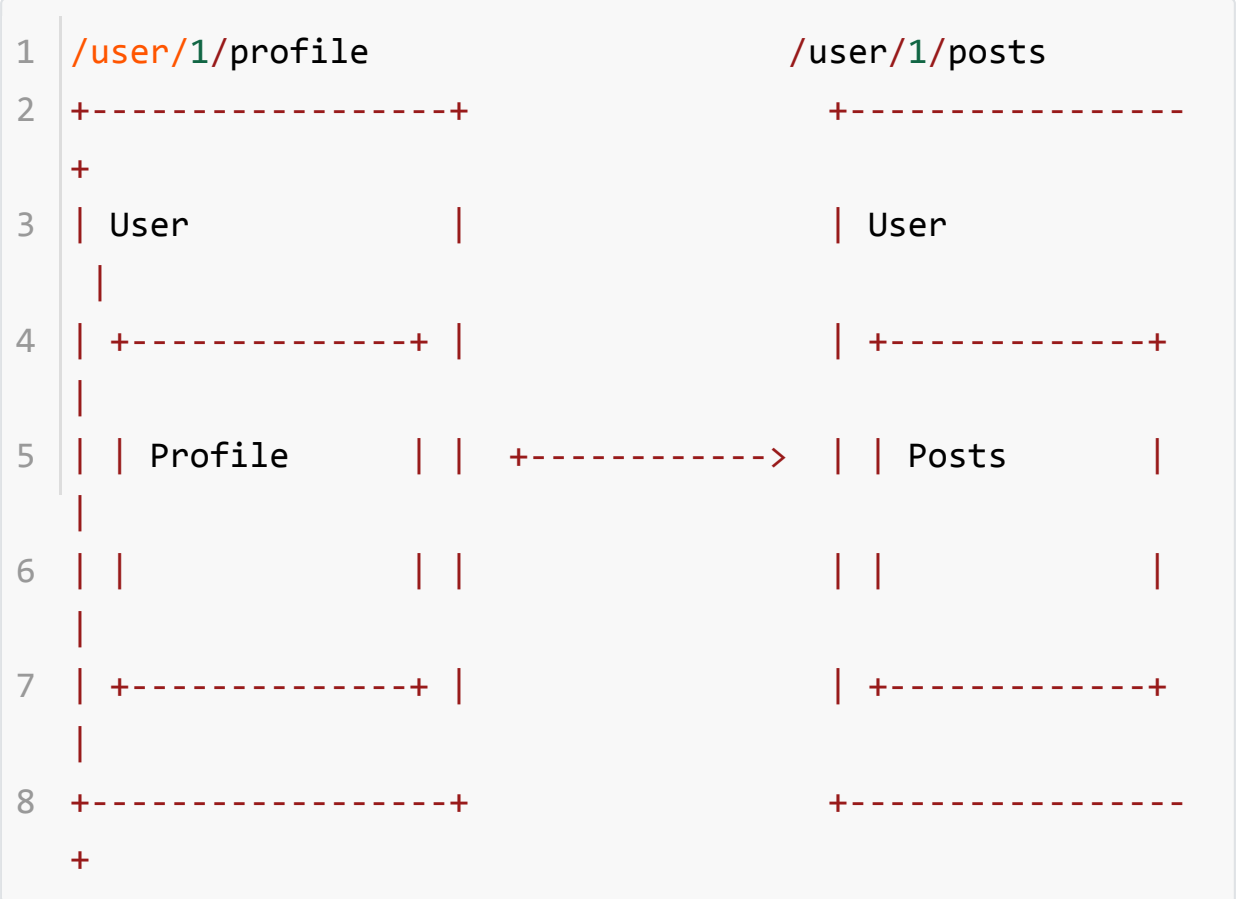
```
// 字符串
this.$router.push('home')

// 对象
this.$router.push({ path: 'home' })

// 命名的路由
this.$router.push({ name: 'user', params: { userId: '123' }})

// 带查询参数，变成 /register?plan=private
this.$.push({ path: 'register', query: { plan: 'private' }})
```

前进后退

```
// 在浏览器记录中前进一步，等同于 history.forward()
router.go(1)

// 后退一步记录，等同于 history.back()
router.go(-1)

// 前进 3 步记录
router.go(3)

// 如果 history 记录不够用，那就默默地失败呗
router.go(-100)
router.go(100)
```

嵌套路由

实际生活中的应用界面，通常由多层嵌套的组件组合而成。同样地，URL 中各段动态路径也按某种结构对应嵌套的各层组件



router.js

```
{
      path: '/user/:id',
      name: 'user',
      component: User,
      props: ({params,query})=>({
        id: params.id,
        title:query.title
      }),
      children:[
         // 当 /user/:id/profile 匹配成功,
        // Profile 会被渲染在 User 的 <router-view> 中
        {
          path:"profile",
          component: Profile
        },
        // 当 /user/:id/posts 匹配成功,
        // Posts 会被渲染在 User 的 <router-view> 中
        {
          path: "posts",
          component: Posts
        }
      ]

}
```

在 `User` 组件的模板添加一个 `<router-view>`：

```
<template>
  <div>
    <h3>用户页面{{$route.params.id}}</h3>
    <h3>用户页面{{id}}</h3>
    <router-view></router-view>
  </div>
</template>
```

App.vue

```
<template>
    <div id='app'>
        <!-- 嵌套理由 -->
      <router-link to="/user/1/profile">User/profile</router-link> |
      <router-link to="/user/1/posts">User/posts</router-link> |
    </div>
</template>
```

命名视图

有时候想同时 (同级) 展示多个视图，而不是嵌套展示，例如创建一个布局，有 `sidebar` (侧导航) 和 `main` (主内容) 两个视图，这个时候命名视图就派上用场了

```
{
      path: '/home',
      name: 'home',
      //注意这个key是components
      components: {
        default: Home, //默认的名字
        main: ()=>import('@/views/Main.vue'),
        sidebar: () => import('@/views/Sidebar.vue')
      }
},
```

App.vue

```
<router-view/>
<router-view name='main'/>
<router-view name='sidebar'/>
```

导航守卫

“导航”表示路由正在发生改变。

完整的导航解析流程

1. 导航被触发。
2. 在失活的组件里调用离开守卫。
3. 调用全局的 `beforeEach` 守卫。
4. 在重用的组件里调用 `beforeRouteUpdate` 守卫 (2.2+)。
5. 在路由配置里调用 `beforeEnter`。
6. 解析异步路由组件。
7. 在被激活的组件里调用 `beforeRouteEnter`。
8. 调用全局的 `beforeResolve` 守卫 (2.5+)。
9. 导航被确认。
10. 调用全局的 `afterEach` 钩子。

11. 触发 DOM 更新。
12. 用创建好的实例调用 `beforeRouteEnter` 守卫中传给 `next` 的回调函数。

全局守卫

你可以使用`router.beforeEach`注册一个全局前置守卫

```
const router = new VueRouter({ ... })

router.beforeEach((to, from, next) => {
  // ...
})
```

有个需求,用户访问在浏览网站时,会访问很多组件,当用户跳转到`/notes`,发现用户没有登录,此时应该让用户登录才能查看,应该让用户跳转到登录页面,登录完成之后才可以查看我的笔记的内容,这个时候全局守卫起到了关键的作用

有两个路由 `/notes`和`/login`

router.vue

```
const router = new VueRouter({
    routes:[
        {
            path: '/notes',
            name: 'notes',
            component: () => import('@/views/Notes')
        },
        {
            path: "/login",
            name: "login",
            component: () => import('@/views/Login')
        },
```

```
    ]
})

// 全局守卫
router.beforeEach((to, from, next) => {
    //用户访问的是'/notes'
    if (to.path === '/notes') {
        //查看一下用户是否保存了登录状态信息
        let user =
JSON.parse(localStorage.getItem('user'))
        if (user) {
            //如果有,直接放行
            next();
        } else {
            //如果没有,用户跳转登录页面登录
            next('/login')
        }
    } else {
        next();
    }
})
```

Login.vue

```
<template>
  <div>
    <input type="text" v-model="username">
    <input type="password" v-model="pwd">
    <button @click="handleLogin">提交</button>
  </div>
</template>

<script>
export default {
```

```
  data() {
    return {
      username: "",
      pwd: ""
    };
  },
  methods: {
    handleLogin() {
      // 1.获取用户名和密码
      // 2.与后端发生交互
      setTimeout(() => {
        let data = {
          username: this.username
        };
         //保存用户登录信息
        localStorage.setItem("user",
JSON.stringify(data));
        // 跳转我的笔记页面
        this.$router.push({ name: "notes" });
      }, 1000);
    },

  }
};
</script>

```

App.vue

```
<!-- 全局守卫演示 -->
<router-link to="/notes">我的笔记</router-link> |
<router-link to="/login">登录</router-link> |
<button @click="handleLogout">退出</button>
```

```
export default {
  methods: {
     handleLogout() {
      //删除登录状态信息
      localStorage.removeItem("user");
      //跳转到首页
      this.$router.push('/')
    }
  },
}
```

组件内的守卫

你可以在路由组件内直接定义以下路由导航守卫：

- `beforeRouteEnter`
- `beforeRouteUpdate` (2.2 新增)
- `beforeRouteLeave`

```
<template>
  <div>
    <h3>用户编辑页面</h3>
    <textarea name id cols="30" rows="10" v-model="content"></textarea>
    <button @click="saveData">保存</button>
    <div class="wrap" v-for="(item,index) in list" :key="index">
      <p>{{item.title}}</p>
    </div>
  </div>
</template>

<script>
export default {
```

```
  data() {
    return {
      content: "",
      list: [],
      confir: true
    };
  },
  methods: {
    saveData() {
        this.list.push({
          title: this.content
        });
        this.content = "";
      }

  },
  beforeRouteLeave(to, from, next) {
    // 导航离开该组件的对应路由时调用
    // 可以访问组件实例 `this`
    if (this.content) {
      alert("请确保保存信息之后,再离开");
      next(false);
    } else {
      next();
    }
  }
};
</script>
```

路由元信息实现权限控制

给需要添加权限的路由设置meta字段

```
{
      path: '/blog',
```

```
      name: 'blog',
      component: () => import('@/views/Blog'),
      meta: {
        requiresAuth: true
}
},
{
      // 路由独享的守卫
      path: '/notes',
      name: 'notes',
      component: () => import('@/views/Notes'),
      meta: {
        requiresAuth: true
      }
},
```

```
// 全局守卫
router.beforeEach((to, from, next) => {
  if (to.matched.some(record =>
record.meta.requiresAuth)) {
    // 需要权限
    if(!localStorage.getItem('user')){
      next({
        path:'/login',
        query:{
          redirect:to.fullPath
        }
      })
    }else{
      next();
    }

  } else {
    next();
```

```
  }
})
```

login.vue

```
//登录操作
handleLogin() {
    // 1.获取用户名和密码
    // 2.与后端发生交互
    setTimeout(() => {
        let data = {
            username: this.username
        };
        localStorage.setItem("user", JSON.stringify(data));
        // 跳转到之前的页面
        this.$router.push({path: this.$route.query.redirect });
    }, 1000);
}
```

数据获取

有时候，进入某个路由后，需要从服务器获取数据。例如，在渲染用户信息时，你需要从服务器获取用户的数据。我们可以通过两种方式来实现：

- **导航完成之后获取**：先完成导航，然后在接下来的组件生命周期钩子中获取数据。在数据获取期间显示“加载中”之类的指示。
- **导航完成之前获取**：导航完成前，在路由进入的守卫中获取数据，在数据获取成功后执行导航。

导航完成后获取数据

当你使用这种方式时，我们会马上导航和渲染组件，然后在组件的 `created` 钩子中获取数据。这让我们有机会在数据获取期间展示一个 loading 状态，还可以在不同视图间展示不同的 loading 状态。

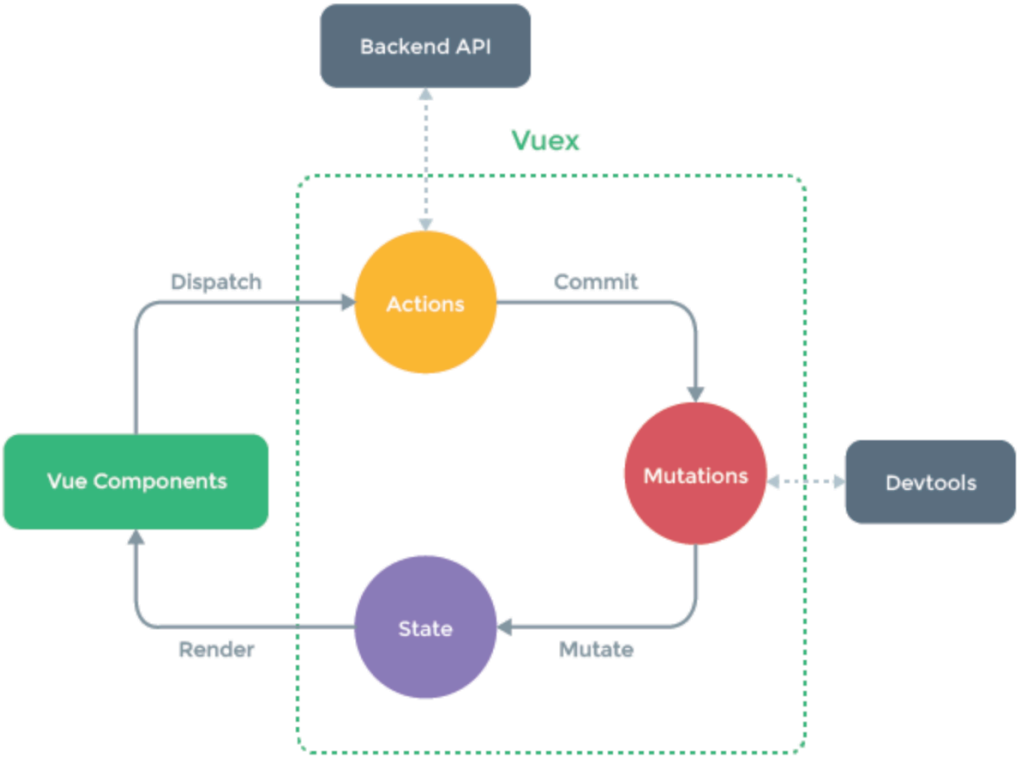
```
<template>
  <div class="post">
    <div v-if="loading" class="loading">Loading...
</div>

    <div v-if="error" class="error">{{ error }}</div>

    <div v-if="post" class="content">
      <h2>{{ post.title }}</h2>
      <p>{{ post.body }}</p>
    </div>
  </div>
</template>
```

```
export default {
  name: "Post",
  data() {
    return {
      loading: false,
      post: null,
      error: null
    };
  },
    // 组件创建完后获取数据，
    // 此时 data 已经被 监视 了
  created() {
     // 如果路由有变化，会再次执行该方法
    this.fetchData();
  },
  watch: {
```

```
    $route: "fetchData"
  },
  methods: {
    fetchData() {
      this.error = this.post = null;
      this.loading = true;
      this.$http.get('/api/post')
      .then((result) => {
          this.loading = false;
          this.post = result.data;
      }).catch((err) => {
          this.error = err.toString();
      });
    }
  }
};
```

Vuex

Vuex 是一个专为 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



安装vuex

```
vue add vuex
```

store.js

```
import Vue from 'vue'
import Vuex from 'vuex'
//确保开头调用Vue.use(Vuex)
Vue.use(Vuex)

export default new Vuex.Store({
  state: { //this.$store.state.count
    count:0
  },
  getters:{
    evenOrOdd:(state)=>{
//this.$store.getters.evenOrOdd
```

```
12        return state.count % 2 ===0 ? '偶数': '奇数'
13      }
14    },
15    mutations: {
16      increment(state){ //this.$store.commit('increment')
17        state.count++
18      },
19      decrement(state){ //this.$store.commit('decrement')
20        state.count--
21      }
22    },
23    actions: {
24      increment({commit}){
 //this.$store.dispatch('increment')
25        // 修改状态的唯一方式是提交mutation
26        commit('increment');
27      },
28      decrement({ commit }) {
//this.$store.dispatch('decrement')
29        commit('decrement');
30      },
31      incrementAsync({commit}){
//this.$store.dispatch('incrementAsync')
32         return new Promise((resolve, reject) => {
33          setTimeout(() => {
34            commit('increment');
35            resolve(10);
36          }, 1000);
37        })
38      }
39    }
40  })
```

我们可以在组件的某个合适的时机通过`this.$store.state`来获取状态对象,以及通过`this.$store.commit`方法触犯状态变更

```
this.$store.commit('increment');
```

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

```
// 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,极大地简化写法

```
computed:{
    ...mapState({
       "count"
    })
}
```

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: {
      ...mapGetters([
        'evenOrOdd'
    ])
  },
}
```

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.doneEvenOrOdd` 映射为 `this.$store.getters.evenOrOdd`
  doneEvenOrOdd: 'evenOrOdd'
})
```

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：

MapMutation

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('counter',[
      'increment',
      'decrement',
    ]),
  }
}
```

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

MapAction辅助函数

```
import { mapMutations } from 'vuex'

export default {
  // ...
  methods: {
    ...mapActions('counter',[
      'incrementAsync'
    ])
  }
}
```

提交方式

```
//在组件内部
// 以载荷形式分发
this.$store.dispatch('incrementAsync', {
  amount: 10
})

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

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：

做一个购物车案例

有两个模块`cart`和`products`

创建store文件夹

```
|---store
    ├── index.js
    └── modules
        ├── cart.js
        └── products.js
```

cart.js

如果希望你的模块具有更高的封装度和复用性，你可以通过添加 `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。

```
export default {
    //使当前模块具有更高的封装度和复用性
    namespaced: true,
    state: {
     ...
    },
    getters: {
        ...
    },
    mutations: {
        ...
    },
    actions: {
        ...
    },
}
```

products.js

```
export default {
    //使当前模块具有更高的封装度和复用性
    namespaced: true,
    state: {
        ...
    },
    getters: {
        ...
    },
```

```
    mutations: {
        ...
    },
    actions: {
        ...
    },
}
```

index.js

```
import Vue from 'vue'
import Vuex from 'vuex'
Vue.use(Vuex)
import cart from './modules/cart';
import products from './modules/products';
export default new Vuex.Store({
    modules:{
        cart,
        products,
    }
})
//this.$store.state.cart //获取cart的状态
//this.$store.state.products //获取products的状态
```

完整购物车案例

mock数据

新建vue.config.js

```
const products = [
    { id: 1, title: 'iphone11', price: 600, inventory: 10 },
    { id: 2, title: 'iphone11 pro', price: 800, inventory: 5 },
```

```
    { id: 3, title: 'iphone11 max', price: 1600, inventory: 6 },
]
module.exports = {
    devServer: {
        before(app, server) {
            app.get('/api/products', (req, res) => {
                res.json({
                    products:products
                })
            })
        }
    }
}
```

cart.js

```
export default {
    //使当前模块具有更高的封装度和复用性
    namespaced: true,
    state: {
        items: [],
    },
    getters: {
        //获取购物车中的商品
        cartProducts: (state, getters, rootState) => {
            return state.items.map(({ id, quantity }) => {
                const product = rootState.products.products.find(product => product.id === id)
                return {
                    title: product.title,
                    price: product.price,
```

```
                    quantity
                }
            })
        },
        // 购物车总价格
        cartTotalPrice: (state, getters) => {
            return getters.cartProducts.reduce((total, product) => {
                return total + product.price * product.quantity
            }, 0)
        }

    },
    mutations: {
        pushProductToCart(state, { id }) {
            state.items.push({
                id,
                quantity: 1
            })
        },
        incrementItemQuantity(state, { id }) {
            const cartItem = state.items.find(item => item.id === id);
            cartItem.quantity++;
        },
    },
    actions: {
        //添加商品到购物车
        addProductToCart({ commit, state }, product) {
            // 如果有库存
            if (product.inventory > 0) {
                const cartItem = state.items.find(item => item.id === product.id);
                if (!cartItem) {
```

```
                    commit('pushProductToCart', { id: product.id });
                } else {
                    commit('incrementItemQuantity', cartItem);
                }
                //提交products模块中decrementProductInventory方法
                //让商品列表的库存数量减1

                commit('products/decrementProductInventory', { id: product.id }, { root: true })
            }

        }
    },
}
```

products.js

```
import Axios from "axios";

export default {
    //使当前模块具有更高的封装度和复用性
    namespaced: true,
    state: {
        products: []
    },
    getters: {

    },
    mutations: {
        setProducts(state, products) {
            state.products = products;
```

```
        },
        //减少商品库存的方法
        decrementProductInventory(state, { id }) {
            const product = state.products.find(product => product.id === id)
            product.inventory--
        }
    },
    actions: {
        //获取所有商品的方法
        getAllProducts({ commit }) {
            Axios.get('/api/products')
                .then(res => {
                    console.log(res.data.products);

 commit('setProducts',res.data.products)
                })
                .catch(err => {
                    console.log(err);

                })
        }
    },
}
```

Products.vue

```
<template>
<div>
    <h3>商铺</h3>
    <ul>
        <li v-for='product in products' :key = 'product.id'>
```

```
            {{product.title}} - {{product.price | currency}}
            <br>
            <button :disabled='!product.inventory' @click='addProductToCart(product)'>添加到购物车</button>
        </li>
    </ul>
    <hr>
    </div>
</template>

<script>
    import { mapState,mapActions } from "vuex";
    export default {
        name: "ProductList",
        data() {
            return {};
        },
        computed: {
            products(){
                return this.$store.state.products.products
            }
        },
        methods: {
            ...mapActions('cart',[
                'addProductToCart'
            ])
        },
        created() {

 this.$store.dispatch("products/getAllProducts");
        }
    };
</script>
```

```
37
38 <style scoped>
39 </style>
```

Cart.vue

```
1  <template>
2  <div>
3      <h2>我的购物车</h2>
4      <i>请增加商品到您的购物车.</i>
5      <ul>
6          <li
7              v-for="product in products"
8              :key="product.id"
9              >{{product.title}}-{{product.price |
   currency}} x {{product.quantity}}
10         </li>
11     </ul>
12     <p>总价格:{{total | currency}}</p>
13     </div>
14 </template>
15
16 <script>
17     import { mapGetters,mapState } from "vuex";
18     export default {
19         name: "shoppingcart",
20         computed:{
21             ...mapGetters('cart',{
22                 products:'cartProducts',
23                 total:'cartTotalPrice'
24             })
25         }
26     };
27 </script>
```

```
28
29 <style scoped>
30 </style>
```

什么情况下我应该使用 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

插件

日志插件

Vuex 自带一个日志插件用于一般的调试:

```
1 import createLogger from 'vuex/dist/logger'
2
3 const store = new Vuex.Store({
4     plugins: [createLogger({
5         collapsed: false, // 自动展开记录的 mutation
6     })]
7 })
```

要注意，logger 插件会生成状态快照，所以仅在开发环境使用。