Vuex

1.介绍

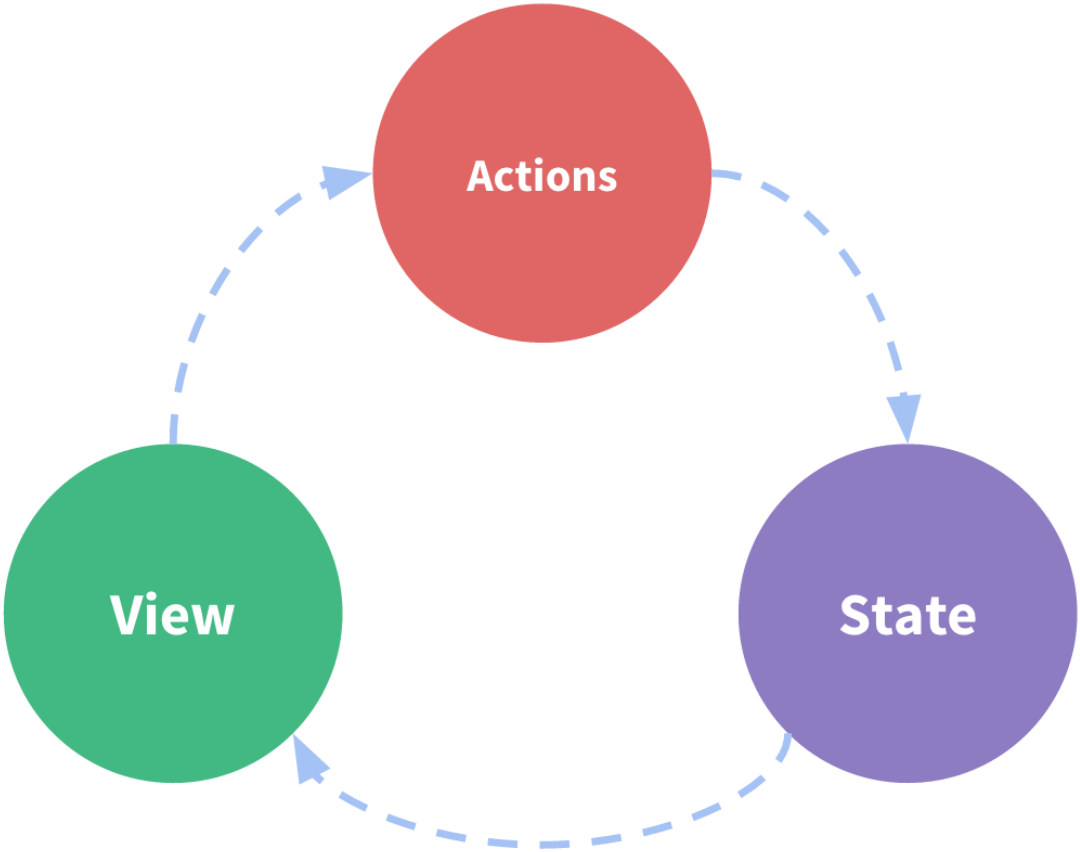
Vuex 是一个专为 Vue.js 应用程序开发的**状态管理模式**。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。

> 1. 状态机可以将所有需要共享的数据进行统一维护，当某个组件需要时，直接引入即可。
> 2. 状态机可以对请求代码进行封装，其他任何组件可以直接调用。

这个状态自管理应用包含以下几个部分：

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

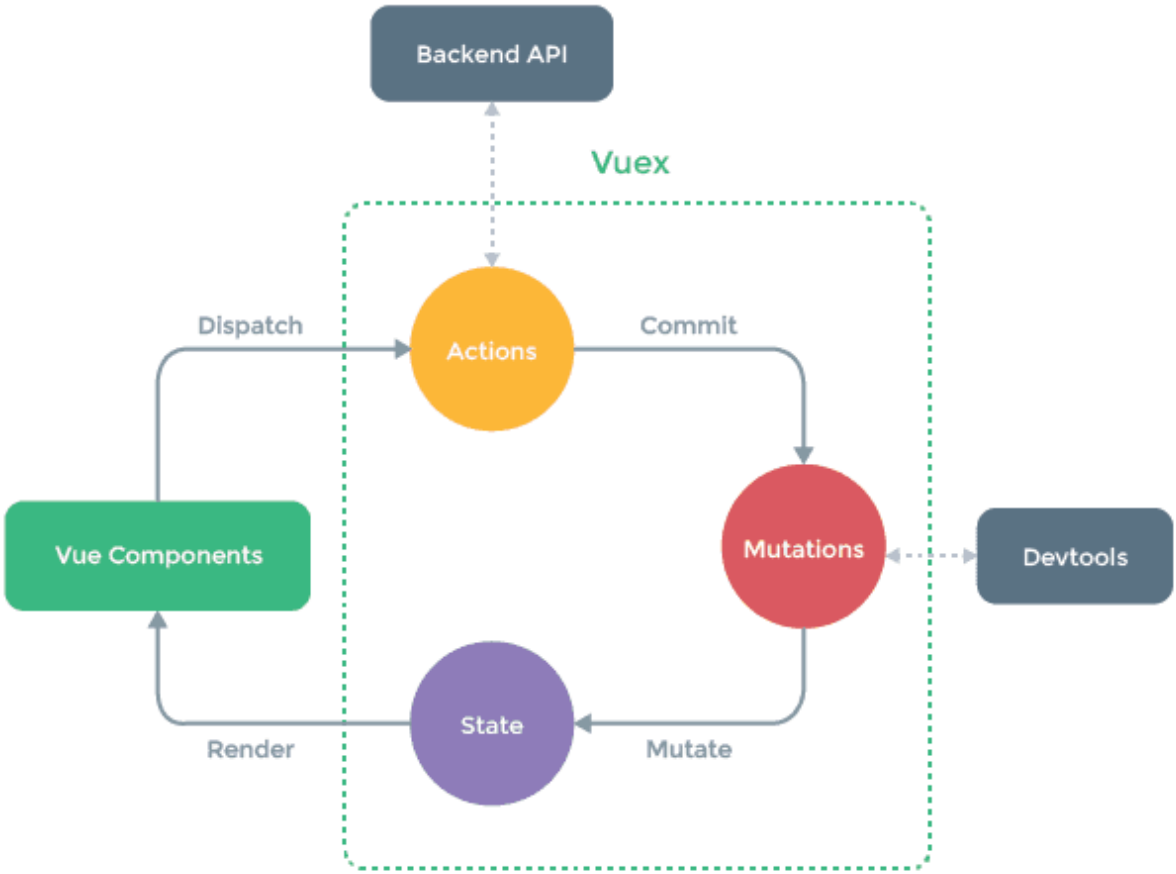
以下是一个表示“单向数据流”理念的简单示意：



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。

因此，我们可以不把组件的共享状态抽取出来，以一个全局单例模式管理。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。

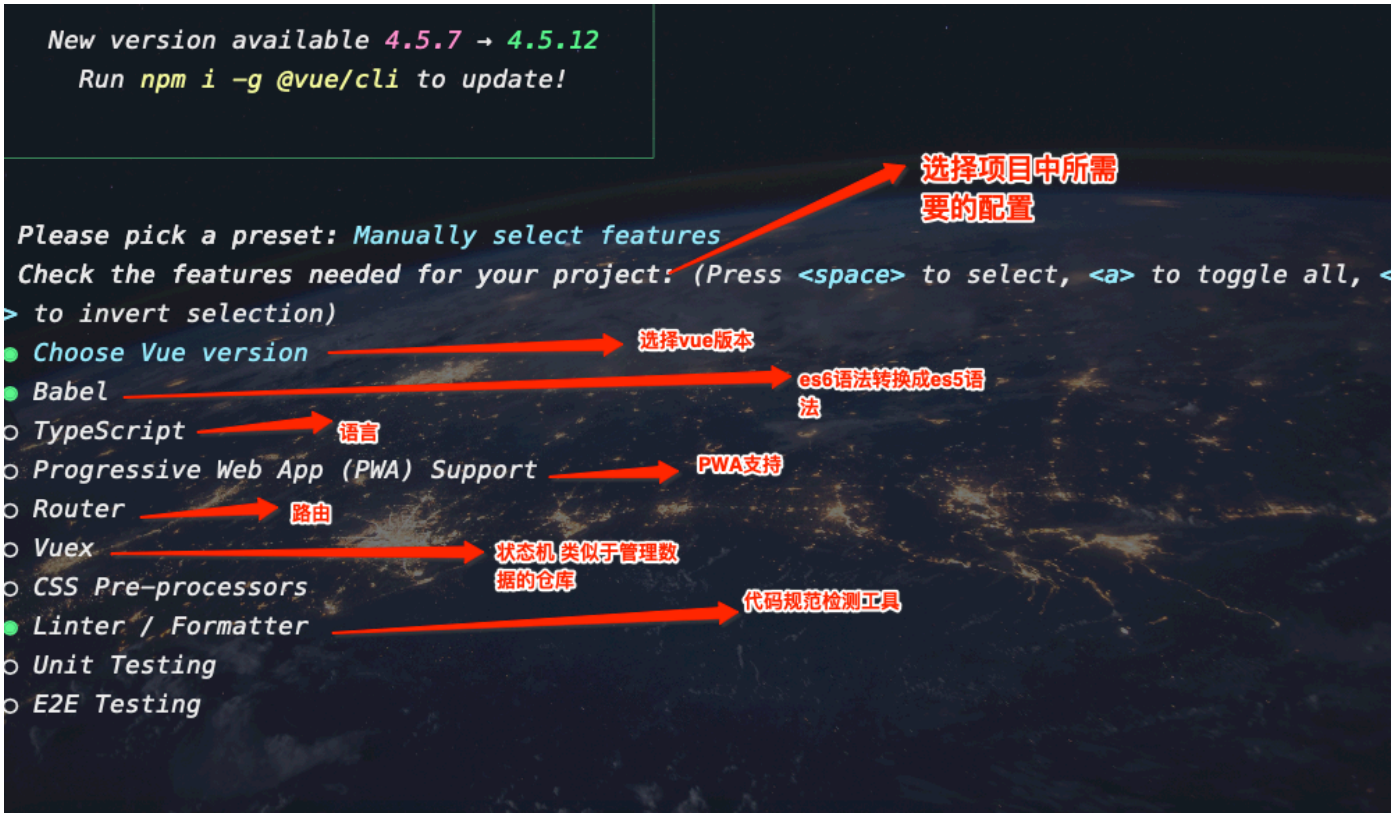


Vuex 和单纯的全局对象有以下两点不同：

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. 使用时不能直接改变 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。

2.安装

1）可在搭建脚手架时选择Vuex选项，自动在脚手架中配置Vuex




```
//main.js 进行Vuex的store的注册
import Vue from 'vue'
import App from './App.vue'
import router from './router'
import store from './store'

Vue.config.productionTip = false
```

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

2）也可使用npm install vuex --save安装Vuex，手动在项目中进行配置。

3.核心概念

State

在`State`中存放状态，可将状态理解为组件中的`data`，只不过在`state`中一般存放的是组件共享的数据，而在组件内的`data`中一般存放组件的私有数据。

存储在 Vuex 中的数据和 Vue 实例中的 data 遵循相同的规则。

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

Vue.use(Vuex)

export default new Vuex.Store({
  state: {
    count: 1
  },
  mutations: {
  },
  actions: {
  },
  modules: {
```

```
  }
})
```

组件访问state中的数据

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 (opens new window)中返回某个状态。

- 方式一：
  在vue代码中，使用`$store.state.xxx`和`this.$store.state.xxx`来获取数据

```
<template>
  <div class="about">
    <h1>
      <!-- 在结构代码中 直接使用$store获取 -->
      {{ $store.state.count }}
      {{ stateCount }}
    </h1>
  </div>
</template>
<script>
export default {
  computed: {
    stateCount() {
      // 在js中需要使用this.$store来获取
      return this.$store.state.count;
    },
  },
};
</script>
```

- 方式二：
  通过 **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：

```
<template>
  <div class="about">
    <h1>
      <!-- 在结构代码中 直接使用$store获取 -->
      {{ $store.state.count }}
      {{ count }}
    </h1>
  </div>
</template>
<script>
// 使用辅助函数
import { mapState } from "vuex";
export default {
  computed: {
    //使用对象展开运算符将此对象混入到外部对象中
    ...mapState(["count"]),
  },
};
</script>
```

Getters

有时候我们需要从 store 中的 state 中派生出一些状态，例如对count进行乘法操作：

```
computed: {
  multiplyCount () {
    return this.$store.state.count * 10
  }
}
```

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 接受 state 作为其第一个参数：

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

Vue.use(Vuex)

export default new Vuex.Store({
  state: {
    count: 1,
  },
  getters: {
    multiplyCount: state => {
      return state.count * 10
    }
  },
  mutations: {
  },
  actions: {
  },
  modules: {
```

```
  }
})
```

组件访问getters中的数据

- 方式一：Getter 会暴露为 store.getters 对象，可以以属性的形式访问这些值

```
$store.getters.multiplyCount
```

Getter 也可以接受其他 getter 作为第二个参数：

```
getters: {
  // ...
  multiplyCountSquare: (state, getters) => {
    return getters.multiplyCount * getters.multiplyCount
  }
}
```

```
$store.getters.multiplyCountSquare
```

我们可以很容易地在任何组件中使用它：

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

注意，getter 在通过属性访问时是作为 Vue 的响应式系统的一部分缓存其中的。

- 方式二：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([
      'multiplyCount',
    ])
  }
}
```

Mutations突变

Mutation 都是同步事务，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：

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

Vue.use(Vuex)

export default new Vuex.Store({
  state: {
    count: 1,
  },
  getters: {
  },
  mutations: {
```

```
    increment(state) {
      // 变更状态
      state.count++
    }
  },
  actions: {
  },
  modules: {
  }
})
```

使用

方式一:

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：

```
<template>
  <div class="about">
    {{ $store.state.count }}
    <button @click="clickMutation">点我啊</button>
  </div>
</template>
<script>
// 使用辅助函数
import { mapState } from "vuex";
export default {
  methods: {
    clickMutation() {
```

```
      this.$store.commit("increment");
    },
  },
};
</script>
```

提交突变载荷（Payload）

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

提交方式一：

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

提交方式二：

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

接受载荷

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

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。

> Mutation 必须是同步操作，内部不可放异步操作

方式二：

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',
      'incrementBy'
    ])
  }
}
```

Actions

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：

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

Vue.use(Vuex)
```

```
export default new Vuex.Store({
  state: {
    count: 1,
  },
  mutations: {
    increment(state, param) {
      // 变更状态
      state.count += param.aCount
    }
  },
  actions: {
    // 在组件中进行分发时也可传递参数
    // this.$store.dispatch("incrementAsync", { aCount:
20 });
    incrementAsync(context, param) {
      context.commit('increment', param)
    },
  },
  // 使用解构可简写为如下代码
  /* incrementAsync ({ commit }) {
    commit('increment')
  } */
  },
  modules: {
  }
})
```

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。

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）。

方式一：

```
this.$store.dispatch('incrementAsync')
```

方式二：

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

Modules

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 = new Vuex.Store({
  modules: {
    a: moduleA,
    b: moduleB
  }
})

store.state.a // -> moduleA 的状态
store.state.b // -> moduleB 的状态
```