

# 3. Vue状态管理&VueCLI

## 课程目标



- 初级：
  - 掌握 Vue 中状态的基础概念
  - 熟练使用状态进行数据管理
  - 熟练掌握 Vue CLI 基本用法
- 中级：
  - 掌握 Vue 中的几种状态组织形式，包括组件状态、全局状态（Pinia）
  - 了解 Vue 中状态的实现机制
  - 了解 Vue CLI 基本实现原理，并熟悉其执行流程
- 高级：
  - 深入理解 Vue 状态原理，响应式的实现
  - 理解 Pinia 框架基本原理
  - 理解 Vue CLI 核心流程源码，并能手写实现

## 课程大纲

1. Vuex介绍及深入使用
2. Vuex实现原理
3. Pinia使用指南& Pinia原理剖析
4. Vue CLI 使用及原理剖析
5. Vue CLI 插件及Preset
6. Vue CLI 配置实战

## 课程内容

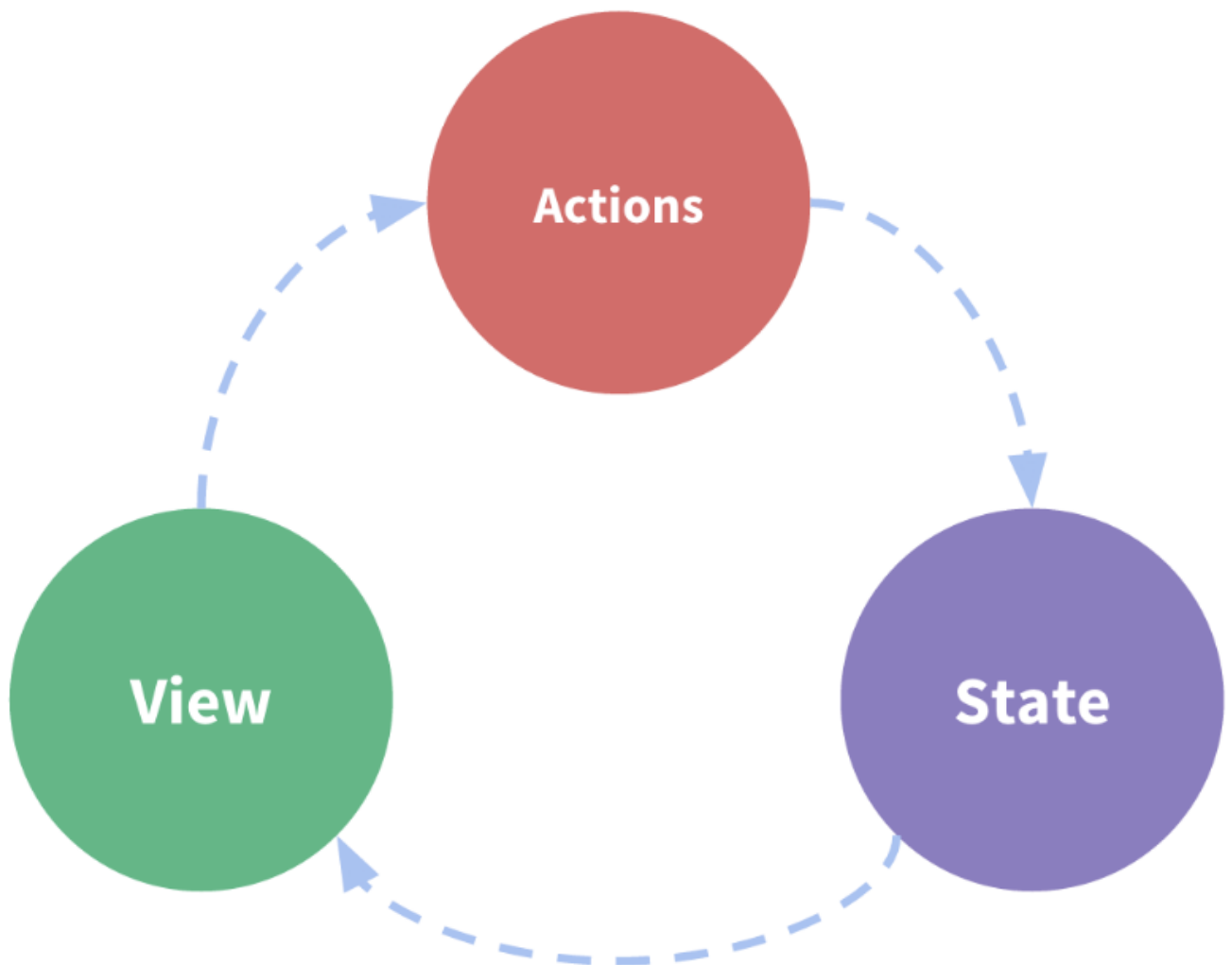
### 3. 为什么需要状态管理

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。

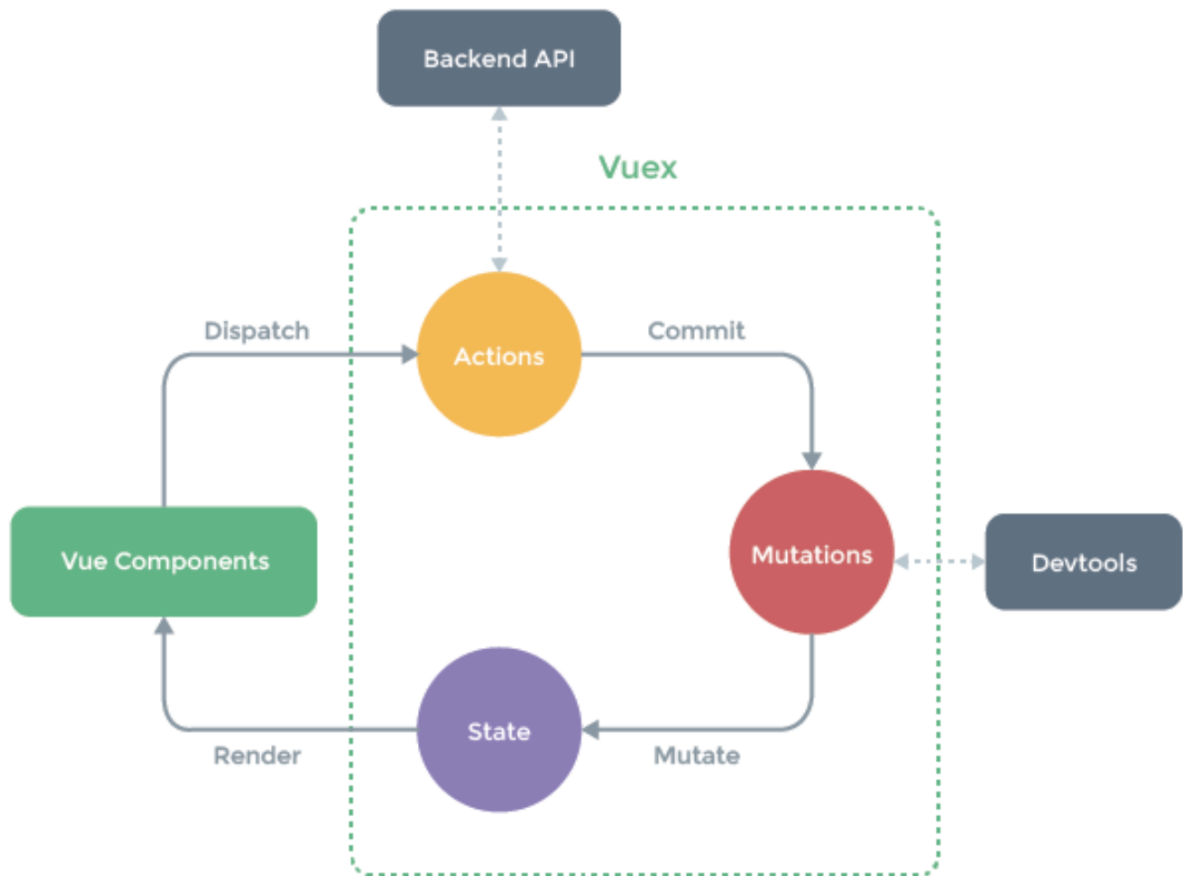
以下是一个表示“单向数据流”理念的简单示意：



在Vue 最重要就是 **数据驱动** 和 **组件化**，每个组件都有自己 `data`，`template` 和 `methods`，`data` 是数据，我们也叫做状态，通过 `methods` 中方法改变状态来更新视图，在单个组件中修改状态更新视图是很方便的，但是实际开发中是

- 多个组件（还有多层组件嵌套）共享同一个状态
- 兄弟组建需要通信

这个时候传参就会很繁琐，就需要进行状态管理，负责组件中的通信，方便维护代码。



需要注意的点：

- 改变状态的唯一途径就是提交mutations
- 如果是异步的，就派发(dispatch)actions，其本质还是提交mutations
- 怎样去触发actions呢？可以用组件Vue Components使用dispatch或者后端接口去触发
- 提交mutations后，可以动态的渲染组件Vue Components

### 3.1 Vuex、Pinia 主要解决的问题

- 多个视图依赖同一个状态
- 来自不同视图的行为需要变更同一个状态

### 3.2 使用 Vuex、Pinia 的好处

- 能够在 vuex 中集中管理共享的数据，易于开发和后期维护
- 能够高效地实现组件之间的数据共享，提高开发效率
- 在 vuex 中的数据都是响应式的

## 4. Vuex使用

## 4.1 安装

通过 `yarn create vite` 创建vue工程，安装 `vuex`

```
yarn add vuex
```

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：

- **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。
- **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。

## 4.2 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。

```
import { createStore } from 'vuex';

const defaultState = {
  count: 0,
};

// Create a new store instance.
export default createStore({
  state() {
    return defaultState;
  },
  mutations: {
    increment(state) {
      state.count++;
    },
  },
});
```

```

    actions: {
      increment(context) {
        context.commit('increment');
      },
    },
  },
});

import { createApp } from 'vue';
import App from './App.vue';
import store from './store';

createApp(App).use(store).mount('#app');

```

### 4.2.1 单一状态树

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”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。

### 4.2.2 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\(opens new window\)](#)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。

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：

```

const Counter = {
  template: `<div>{{ count }}</div>`,
  computed: {
    count () {

```

```

        return this.$store.state.count
      }
    }
  }
}

```

### 4.2.3 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
    }
  })
}

```

### 4.2.4 对象展开运算符

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\(opens new window\)](#)，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({
    // ...
  })
}

```

## 4.2.5 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。

## 4.3 vuex 核心概念Getter(修饰器)

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)  
    }  
  }  
})
```

### 4.3.1 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 }]
```

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

### 4.3.2 通过方法访问

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

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

注意，getter 在通过方法访问时，每次都会去进行调用，而不会缓存结果。

### 4.3.3 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'
})
```

## 4.4 核心概念 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')
```

### 4.4.1 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
})
```

#### 4.4.2 对象风格的提交方式

```
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
  }
}
```

#### 4.4.3 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**。

#### 4.4.4 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')`
    })
  }
}
```

## 4.5 核心概念Actions

Actions存在的意义是假设你在修改state的时候有异步操作，vuex作者不希望你将异步操作放在Mutations中，所以就给你设置了一个区域，让你放异步操作，这就是Actions。

```
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')
  }
}
```

### 4.5.1 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
})
```

## 4.5.2 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')`
    })
  }
}
```

## 4.6 核心概念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

```

### 4.6.1 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  
    }  
  }  
}
```

## 4.6.2 命名空间

默认情况下，模块内部的 action 和 mutation 仍然是注册在**全局命名空间**的——这样使得多个模块能够对同一个 action 或 mutation 作出响应。Getter 同样也默认注册在全局命名空间，但是目前这并非出于功能上的目的（仅仅是维持现状来避免非兼容性变更）。必须注意，不要在不同的、无命名空间的模块中定义两个相同的 getter 从而导致错误。

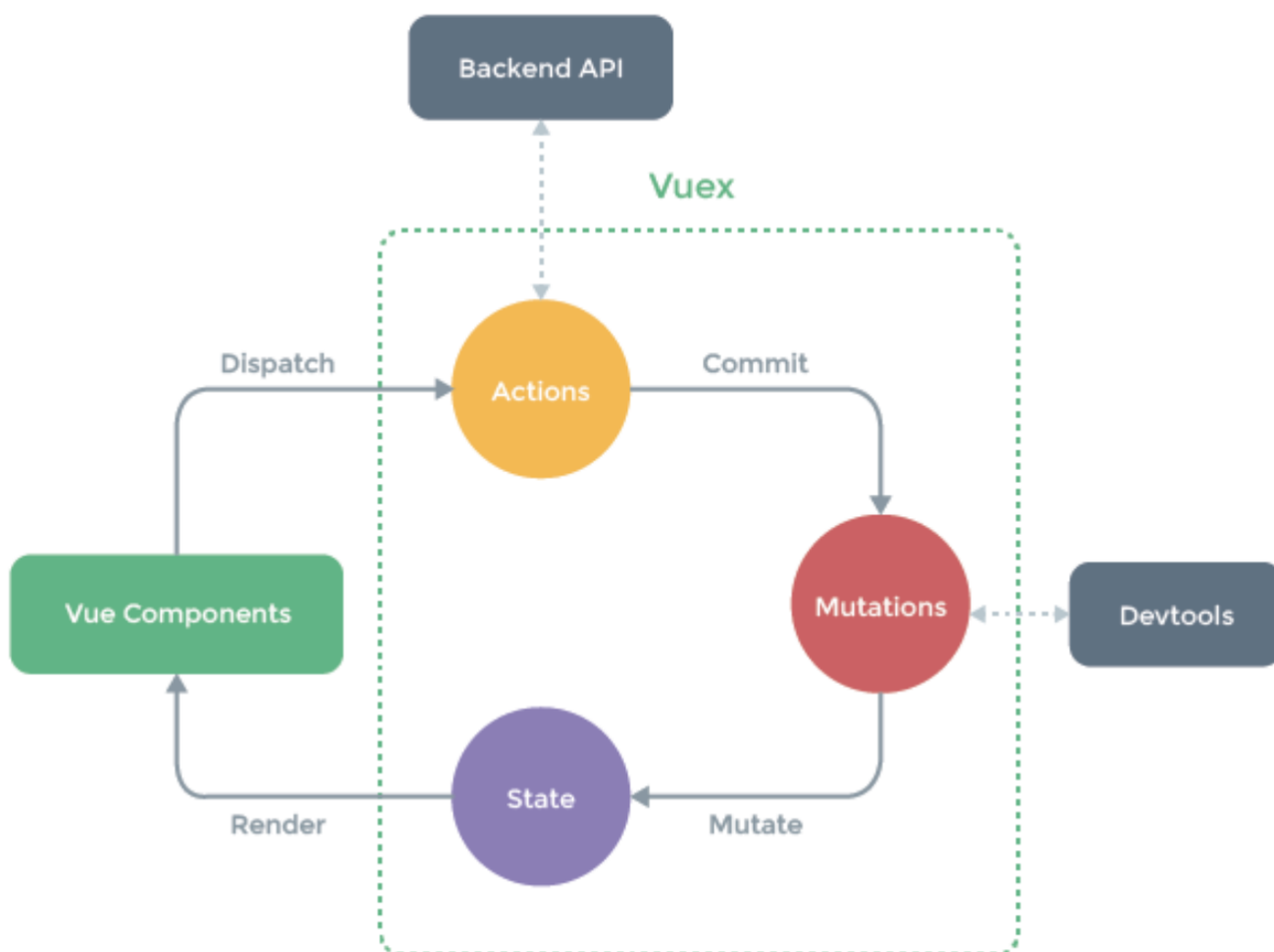
如果希望你的模块具有更高的封装度和复用性，你可以通过添加 `namespaced: true` 的方式使其成为带命名空间的模块。当模块被注册后，它的所有 getter、action 及 mutation 都会自动根据模块注册的路径调整命名。

```
computed: {  
  formatMessage() {  
    return this.message + 'world';  
  },  
  ...mapState('cartModule', ['count']),  
},  
methods: {  
  ...mapActions('cartModule', ['incrementIfOddOnRootSum']),  
},
```

## 4. Vuex原理

如图示，Vuex为Vue Components建立起了一个完整的生态圈，包括开发中的API调用一环。围绕这个生态圈，简要介绍一下各模块在核心流程中的主要功能：

- **Vue Components**: Vue组件。HTML页面上，负责接收用户操作等交互行为，执行dispatch方法触发对应action进行回应。
- **dispatch**: 操作行为触发方法，是唯一能执行action的方法。
- **actions**: 操作行为处理模块。负责处理Vue Components接收到的所有交互行为。包含同步/异步操作，支持多个同名方法，按照注册的顺序依次触发。向后台API请求的操作就在这个模块中进行，包括触发其他action以及提交mutation的操作。该模块提供了Promise的封装，以支持action的链式触发。
- **commit**: 状态改变提交操作方法。对mutation进行提交，是唯一能执行mutation的方法。
- **mutations**: 状态改变操作方法。是Vuex修改state的唯一推荐方法，其他修改方式在严格模式下将会报错。该方法只能进行同步操作，且方法名只能全局唯一。操作之中会有一些hook暴露出来，以进行state的监控等。
- **state**: 页面状态管理容器对象。集中存储Vue components中data对象的零散数据，全局唯一，以进行统一的状态管理。页面显示所需的数据从该对象中进行读取，利用Vue的细粒度数据响应机制来进行高效的状态更新。
- **getters**: state对象读取方法。图中没有单独列出该模块，应该被包含在了render中，Vue Components通过该方法读取全局state对象。





## 5. Pinia使用

### 5.1 简介

pinia 是由 vue 团队开发的，适用于 vue2 和 vue3 的状态管理库。

与 vue2 和 vue3 配套的状态管理库为 vuex3 和 vuex4，pinia被誉为 vuex5。

相比于 Vuex，Pinia 提供了更简洁直接的 API，并提供了组合式风格的 API，最重要的是，在使用 TypeScript 时它提供了更完善的类型推导。

- pinia 没有命名空间模块。
- pinia 无需动态添加（底层通过 `getCurrentInstance` 获取当前 vue 实例进行关联）。
- pinia 是平面结构（利于解构），没有嵌套，可以任意交叉组合。

### 5.2 安装

```
yarn add pinia --save
```

使用

```
import { createPinia } from 'pinia'  
app.use(createPinia())
```

#### 定义state 和 getters

```
//store.js  
import { defineStore } from 'pinia'  
export const PublicStore = defineStore('Public', { // Public项目唯一id  
  state: () => {  
    return {  
      userMsg: {},  
    }  
  },  
  getters: {  
    getUserMsg: (state) => {  
      return state.userMsg  
    },  
  },  
  // other options...  
})
```

## 5.3 Store 是什么？

Store (如 Pinia) 是一个保存状态和业务逻辑的实体，它并不与你的组件树绑定。换句话说，它**承载着全局状态**。它有点像一个永远存在的组件，每个组件都可以读取和写入它。它有三个概念，**state**、**getter** 和 **action**，我们可以假设这些概念相当于组件中的 `data`、`computed` 和 `methods`。

### 应该在什么时候使用 Store？

一个 Store 应该包含可以在整个应用中访问的数据。这包括在许多地方使用的数据，例如显示在导航栏中的用户信息，以及需要通过页面保存的数据，例如一个非常复杂的多步骤表单。

另一方面，你应该避免在 Store 中引入那些原本可以在组件中保存的本地数据，例如，一个元素在页面中的可见性。

并非所有的应用都需要访问全局状态，但如果你的应用确实需要一个全局状态，那 Pinia 将使你的开发过程更轻松。

## 5.4 定义Store

使用**`defineStore`**定义**store**，第一个参数必须是全局唯一的id，可以使用**`Symbol`**

```
import { defineStore } from 'pinia'

// 第一个参数必须是全局唯一，可以是哟
export const useCounterStore = defineStore('counter', {
  state: () => {
    return { count: 0 }
  },
  // 也可以这样定义
  // state: () => ({ count: 0 })
  actions: {
    increment() {
      this.count++
    },
  },
})
```

然后你就可以在一个组件中使用该 store 了：

```
<script setup>
import { useCounterStore } from '@stores/counter'
const counter = useCounterStore()
counter.count++
// 自动补全! ✨
counter.$patch({ count: counter.count + 1 })
// 或使用 action 代替
```

```
    counter.increment()
  </script>
  <template>
    <!-- 直接从 store 中访问 state -->
    <div>Current Count: {{ counter.count }}</div>
  </template>
```

## 定义强状态

```
import { defineStore } from 'pinia'

interface ICounterStoreState{
  counter: number
}
// 第一个参数必须是全局唯一，可以是哟
export const useCounterStore = defineStore('counter', {
  state: ():ICounterStoreState => {
    return { count: 0 }
  },
  // 也可以这样定义
  // state: () => ({ count: 0 })
  actions: {
    increment() {
      this.count++
    },
  },
})
```

## 使用和重置

```
<script setup lang="ts">
import { ref } from 'vue'
import {useCounterStore} from '../store/index.ts'
const counter = useCounterStore();

counter.count++
counter.increment()
// 重置
counter.$reset()

</script>
```

## 改变状态

```
counter.$patch({
  count: counter.count + 1,
  name: 'Abalam',
})
```

## 计算属性Getters

Getter 完全等同于 Store 状态的计算值，可以用 `defineStore()` 中的 `getters` 属性定义

```
export const useCounterStore = defineStore('counter', {
  state: ():ICounterStoreState => ({
    count: 0
  }),
  getters: {
    doubleCount: state => state.count * 2
  }
})

// 组建中可以直接使用
// counter.doubleCount
```

## 传递参数到getters

Getter 是计算属性，也可叫只读属性，因此不可能将任何参数传递给它们。但是可以从 getter 返回一个函数以接受任何参数。

```
import { defineStore } from 'pinia'

interface ICountStoreState {
  count: number;
}

export const useCounterStore = defineStore('counter', {
  state: ():ICounterStoreState => ({
    count: 0
  }),
  actions: {
    increment() {
      this.count++;
    }
  }
})
```

```

    },
    getters: {
      doubleCount: state => state.count * 2,
      getUserById: (state) => {
        return (userId) => state.users.find((user) => user.id === userId)
      },
    }
  })
})

```

## 动作Actions

Action 相当于组件中的 [method](#)。它们可以通过 `defineStore()` 中的 `actions` 属性来定义，**并且它们也是定义业务逻辑的完美选择。**

```

export const useTodos = defineStore('todos', {
  state: () => ({
    /** @type {{ text: string, id: number, isFinished: boolean }[]} */
    todos: [],
    /** @type {'all' | 'finished' | 'unfinished'} */
    filter: 'all',
    // 类型将自动推断为 number
    nextId: 0,
  }),
  getters: {
    finishedTodos(state) {
      // 自动补全! ✨
      return state.todos.filter((todo) => todo.isFinished)
    },
    unfinishedTodos(state) {
      return state.todos.filter((todo) => !todo.isFinished)
    },
  },
  /**
   * @returns {{ text: string, id: number, isFinished: boolean }[]}
   */
  filteredTodos(state) {
    if (this.filter === 'finished') {
      // 调用其他带有自动补全的 getters ✨
      return this.finishedTodos
    } else if (this.filter === 'unfinished') {
      return this.unfinishedTodos
    }
    return this.todos
  },
},
),

```

```

    actions: {
      // 接受任何数量的参数，返回一个 Promise 或不返回
      addTodo(text) {
        // 你可以直接变更该状态
        this.todos.push({ text, id: this.nextId++, isFinished: false })
      },
    },
  },
})

```

## 访问其他 store 的 action

想要使用另一个 store 的话，那你直接在 *action* 中调用就好了：

```

export const useCounterStore = defineStore('counter', {
  state: (): ICounterStoreState => ({
    count: 0
  }),
})

export const useSettingsStore = defineStore('settings', {
  state: () => ({
    preferences: null,
  }),
  actions: {
    async fetchUserPreferences() {
      const counter = useCounterStore()
      console.log(counter)
    },
  },
})

```

## 6. Pinia 原理

1. pinia中可以定义多个store，每个store都是一个reactive对象
2. pinia的实现借助了scopeEffect
3. 全局注册一个rootPinia，通过provide提供pinia
4. 每个store使用都必须在setup中，因为这里才能inject到pinia

### pinia的state的实现

```

export const symbolPinia = Symbol("rootPinia");

import { App, effectScope, markRaw, Plugin, ref, EffectScope, Ref } from "vue";
import { symbolPinia } from "../rootStore";

export const createPinia = () => {
  // 作用域scope 独立空间
  const scope = effectScope(true);
  // run方法发返回值就是这个fn的返回结果
  const state = scope.run(() => ref({}));
  // 将一个对象标记为不可被转为代理。返回该对象本身。
  const pinia = markRaw({
    install(app: App) {
      // pinia希望能被共享出去
      // 将pinia实例暴露到app上，所有的组件都可以通过inject注入进去
      app.provide(symbolPinia, pinia);
      // 可以在模板访问 直接通过 $pinia访问根pinia
      app.config.globalProperties.$pinia = pinia;
      // pinia也记录一下app 方便后续使用
      pinia._a = app;
    },
    // 所有的state
    state,
    _e: scope, // 管理整个应用的scope
    // 所有的store
    _s: new Map(),
  } as Plugin & IRootPinia);
  return pinia;
};

export interface IRootPinia {
  [key: symbol]: symbol;
  _a: App;
  state: Ref<any>;
  _e: EffectScope;
  _s: Map<string, any>;
}

```

## defineStore:

```

import {
  getCurrentInstance,
  inject,
  effectScope,
  EffectScope,

```

```

    reactive,
  } from "vue";
import { IRootPinia } from "../createPinia";
import { symbolPinia } from "../rootStore";

export function defineStore(options: IPiniaStoreOptions): any;
export function defineStore(
  id: string,
  options: Pick<IPiniaStoreOptions, "actions" | "getters" | "state">
): any;
export function defineStore(id: string, setup: () => any): any;
export function defineStore(idOrOptions: any, storeSetup?: any) {
  let id: string, options: any;
  if (typeof idOrOptions === "string") {
    id = idOrOptions;
    options = storeSetup;
  } else {
    // 这里就是一个参数的形式 id参数定义在对象内
    options = idOrOptions;
    id = idOrOptions.id;
  }
  // 注册一个store
  function useStore() {
    // 必须在setup中使用
    const currentInstance = getCurrentInstance();
    if (!currentInstance) throw new Error("pinia 需要在setup函数中使用");
    // 注入 pinia
    const pinia = inject<IRootPinia>(symbolPinia)!;
    // 还没注册
    if (!pinia._s.has(id)) {
      // counter:state:{count:0}
      createOptionsStore(id, options, pinia);
    }
    // 获取store
    const store = pinia._s.get(id);
    return store;
  }
  return useStore;
}

const createOptionsStore = (
  id: string,
  options: Pick<IPiniaStoreOptions, "actions" | "getters" | "state">,
  pinia: IRootPinia
) => {
  const { state, getters, actions } = options;
  // store单独的scope

```



```

let scope: EffectScope;
const setup = () => {
  // 缓存 state
  if (pinia.state.value[id]) {
    console.warn(`${id} store 已经存在!`);
  }
  const localState = (pinia.state.value[id] = state ? state() : {});
  return localState;
};
// scope可以停止所有的store 每个store也可以停止自己的
const setupStore = pinia._e.run(() => {
  scope = effectScope();
  return scope.run(() => setup());
});
// 一个store 就是一个reactive对象
const store = reactive({});
Object.assign(store, setupStore);
// 向pinia中放入store
pinia._s.set(id, store);
console.log(pinia)
};

export interface IPiniaStoreOptions {
  id?: string;
  state?: () => any;
  getters?: any;
  actions?: any;
}

```

## actions 和 getters

```

const createOptionsStore = (
  id: string,
  options: Pick<IPiniaStoreOptions, "actions" | "getters" | "state">,
  pinia: IRootPinia
) => {
  const { state, getters = {}, actions } = options;
  // store单独的scope
  let scope: EffectScope;
  const setup = () => {
    // 缓存 state
    if (pinia.state.value[id]) {
      console.warn(`${id} store 已经存在!`);
    }
    const localState = (pinia.state.value[id] = state ? state() : {});

```

```

    return Object.assign(
      localState,
      actions,
      Object.keys(getters).reduce(
        (computedGetter: { [key: string]: ComputedRef<any> }, name) => {
          // 计算属性可缓存
          computedGetter[name] = computed(() => {
            // 我们需要获取当前的store是谁
            return Reflect.apply(getters[name], store, [store]);
          });
          return computedGetter;
        },
        {}
      )
    );
  };
  // scope可以停止所有的store 每个store也可以停止自己的
  const setupStore = pinia._e.run(() => {
    scope = effectScope();
    return scope.run(() => setup());
  });
  // 一个store 就是一个reactive对象
  const store = reactive({});
  // 处理action的this问题
  for (const key in setupStore) {
    const prop = setupStore[key];
    if (typeof prop === "function") {
      // 扩展action
      setupStore[key] = wrapAction(key, prop, store);
    }
  }
  Object.assign(store, setupStore);
  // 向pinia中放入store
  pinia._s.set(id, store);
  setTimeout(() => {
    console.log(pinia);
  }, 2000);
};

const wrapAction = (key: string, action: any, store: any) => {
  return (...args: Parameters<typeof action>) => {
    // 触发action之前 可以触发一些额外的逻辑
    const res = Reflect.apply(action, store, args);
    // 返回值也可以做处理
    return res;
  };
};

```

## setupStore的原理

```
function useStore() {
  // 必须在setup中使用
  const currentInstance = getCurrentInstance();
  if (!currentInstance) throw new Error("pinia 需要在setup函数中使用");
  // 注入 pinia
  const pinia = inject<IRootPinia>(symbolPinia)!;
  // 还没注册
  if (!pinia._s.has(id)) {
    if (isSetupStore) {
      // 创建setupStore
      createSetupStore(id, storeSetup, pinia);
    } else {
      // counter:state:{count:0}
      createOptionsStore(id, options, pinia);
    }
  }
  // 获取store
  const store = pinia._s.get(id);
  return store;
}

const createSetupStore = (id: string, setup: () => any, pinia: IRootPinia) => {
  // 一个store 就是一个reactive对象
  const store = reactive({});
  // store单独的scope
  let scope: EffectScope;
  // scope可以停止所有的store 每个store也可以停止自己的
  const setupStore = pinia._e.run(() => {
    scope = effectScope();
    return scope.run(() => setup());
  });
  // 处理action的this问题
  for (const key in setupStore) {
    const prop = setupStore[key];
    if (typeof prop === "function") {
      // 扩展action
      setupStore[key] = wrapAction(key, prop, store);
    }
  }
  Object.assign(store, setupStore);
  // 向pinia中放入store
  pinia._s.set(id, store);
  return store;
};
```


```

const createOptionsStore = (
  id: string,
  options: Pick<IPiniaStoreOptions, "actions" | "getters" | "state">,
  pinia: IRootPinia
) => {
  const { state, getters = {}, actions } = options;
  const setup = () => {
    // 缓存 state
    if (pinia.state.value[id]) {
      console.warn(`${id} store 已经存在!`);
    }
    const localState = (pinia.state.value[id] = state ? state() : {});
    return Object.assign(
      localState,
      actions,
      Object.keys(getters).reduce(
        (computedGetter: { [key: string]: ComputedRef<any> }, name) => {
          // 计算属性可缓存
          computedGetter[name] = computed(() => {
            // 我们需要获取当前的store是谁
            return Reflect.apply(getters[name], store, [store]);
          });
          return computedGetter;
        },
        {}
      )
    );
  };
  const store = createSetupStore(id, setup, pinia);
};

```

## 7. Vue CLI 使用及原理剖析

Vue CLI 的功能职责类似于 create-react-app，它封装了一系列工具集，开发者可以通过调用对应命令，方便快捷完成系列开发相关辅助型操作。比如项目的开发模式启动，项目的打包，项目的构建产物分析等。

 注意，我们说的 Vue CLI 其实包含两个，一个是早期的 Vue CLI，打包基于 webpack，另一个是最新的 CLI，打包基于 vite，他们的地址分别为：

<https://cli.vuejs.org/zh/guide/installation.html>、<https://cn.vuejs.org/guide/scaling-up/tooling.html>

我们首先介绍的是前者，Vue CLI <https://github.com/vuejs/vue-cli>，但其实我们新项目更多回去选择 create-vue <https://github.com/vuejs/create-vue>

- Vue CLI 创建应用： `npm install -g @vue/cli`， `vue create hello`
- vue-create 创建： `npm create vue@3`

## vue-cli-service

Vue CLI 的背后是 `vue-cli-service` 提供支持，很多同学在这里就有疑问，为什么包叫 vue-cli-service，而我们运行的时候却是使用的 vue？

这里作为拓展知识给大家简单提一句，任何包的 package.json 文件中，都有两个参数用来指定入口，分别为：**main**、**lib**，前者指定的是包作为依赖时的入口文件配置，而后者则指定的是当该包被全局安装时创建软链接的文件。

## 常用命令

### serve

用法： `vue-cli-service serve [options] [entry]`

选项：

<code>--open</code>	在服务器启动时打开浏览器
<code>--copy</code>	在服务器启动时将 URL 复制到剪切版
<code>--mode</code>	指定环境模式（默认值：development）
<code>--host</code>	指定 host（默认值：0.0.0.0）
<code>--port</code>	指定 port（默认值：8080）
<code>--https</code>	使用 https（默认值：false）

`vue-cli-service serve` 命令会启动一个开发服务器 (基于 [webpack-dev-server](#)) 并附带开箱即用的模块热重载 (Hot-Module-Replacement)。

除了通过命令行参数，你也可以使用 `vue.config.js` 里的 `devServer` 字段配置开发服务器。

命令行参数 `[entry]` 将被指定为唯一入口 (默认值： `src/main.js`，TypeScript 项目则为 `src/main.ts`)，而非额外的追加入口。尝试使用 `[entry]` 覆盖 `config.pages` 中的 `entry` 将可能引发错误。

### build

用法： `vue-cli-service build [options] [entry|pattern]`

选项：

<code>--mode</code>	指定环境模式（默认值：production）
<code>--dest</code>	指定输出目录（默认值：dist）
<code>--modern</code>	面向现代浏览器带自动回退地构建应用
<code>--target</code>	app   lib   wc   wc-async（默认值：app）
<code>--name</code>	库或 Web Components 模式下的名字（默认值：package.json 中的 "name" 字段或入口文件名）
<code>--no-clean</code>	在构建项目之前不清除目标目录的内容
<code>--report</code>	生成 report.html 以帮助分析包内容
<code>--report-json</code>	生成 report.json 以帮助分析包内容
<code>--watch</code>	监听文件变化

`vue-cli-service build` 会在 `dist/` 目录产生一个可用于生产环境的包，带有 JS/CSS/HTML 的压缩，和为更好的缓存而做的自动的 vendor chunk splitting。它的 chunk manifest 会内联在 HTML 里。

这里还有一些有用的命令参数：

- `--modern` 使用[现代模式](#)构建应用，为现代浏览器交付原生支持的 ES2015 代码，并生成一个兼容老浏览器的包用来自动回退。
- `--target` 允许你将项目中的任何组件以一个库或 Web Components 组件的方式进行构建。更多细节请查阅[构建目标](#)。
- `--report` 和 `--report-json` 会根据构建统计生成报告，它会帮助你分析包中包含的模块们的大小。

## inspect

用法：vue-cli-service inspect [options] [...paths]

选项：

<code>--mode</code>	指定环境模式（默认值：development）
---------------------	-------------------------

你可以使用 `vue-cli-service inspect` 来审查一个 Vue CLI 项目的 webpack config。更多细节请查阅[审查 webpack config](#)。

## help

该命令用于查看 Vue CLI 提供的命令

npx vue-cli-service help

## Vue CLI 插件及 Preset

Vue 2 项目，大多使用 Vue CLI 初始化，打包时基于 webpack 的，因此 Vue CLI 提供了基于插件化机制的规则配置方案，Preset 预设也是内部针对于 Webpack 打包所需 loader 等内容进行了高度封装。

比如你想让 Vue 支持 ts，你可以安装对应 ts loader，`vue add @vue/typescript`

- `config.rule('ts')`
- `config.rule('ts').use('ts-loader')`
- `config.rule('ts').use('babel-loader')` (当配合 `@vue/cli-plugin-babel` 使用)
- `config.rule('ts').use('cache-loader')`
- `config.plugin('fork-ts-checker')`

一个 Vue CLI preset 是一个包含**创建新项目所需预定义选项和插件的 JSON 对象**，让用户无需在命令提示中选择它们。

在 `vue create` 过程中保存的 preset 会被放在你的 home 目录下的一个配置文件中 (`~/.vuerc`)。你可以通过直接编辑这个文件来调整、添加、删除保存好的 preset。

这里有一个 preset 的示例：

```
{
  "useConfigFiles": true,
  "cssPreprocessor": "sass",
  "plugins": {
    "@vue/cli-plugin-babel": {},
    "@vue/cli-plugin-eslint": {
      "config": "airbnb",
      "lintOn": ["save", "commit"]
    },
    "@vue/cli-plugin-router": {},
    "@vue/cli-plugin-vuex": {}
  }
}
```

## Vue CLI 配置实战【简单了解就行】

<https://cli.vuejs.org/zh/config/>

## 配置参考 | Vue CLI

 Vue.js 开发的标准工具

这是 Vue CLI 遗留的功能了，其实我们在往后的新项目中可能不会选择 Vue CLI，因为你可以选择 create-vue 基于 vite 的方式，或者你也可以根据自己项目需要，配置自己特有的打包构建环境。

## 课程总结

这节课我们深入学习了 Vue 中状态管理的常见方案。包括内部状态，我们使用 reactive 或 ref 等 Composition API 实现，当然你也可以使用选项式 API，因为大家非常熟，我们不做赘述。

关于集中状态管理机制，我们介绍了两个：Vuex、Pinia，并横向做了对比，大家回顾一下，为什么我建议大家后续新项目选择 Pinia？

理解基础概念及用法后，我带大家完整理解了一下这两个状态管理的实现原理。关于全局状态管理就记住两大点：

1. 内部发布订阅实现状态的更新与派发
2. 库与 Vue 之间通过 Vue 插件形式，将状态同步到 Vue 中。

最后给大家介绍了 Vue CLI 和 create-vue 的使用、基本原理以及如何选择。相信你对两者的认识又加深了一层。

## 参考地址

- <https://pinia.vuejs.org/zh/getting-started.html>
- <https://vuex.vuejs.org/zh/index.html>



