Vuex

ЧТО ТАКОЕ VUEX?

Vuex — паттерн управления состоянием + библиотека для приложений на Vue.js.

Он служит централизованным хранилищем данных для всех компонентов приложения с правилами, гарантирующими, что состояние может быть изменено только предсказуемым образом.

Vuex интегрируется с официальным расширением vue-devtools, предоставляя «из коробки» дополнительные возможности для отладки и экспорт/импорт слепков состояния данных.

Подключение библиотеки:

```
<script src="https://unpkg.com/vuex"></script>
```

ЧТО ТАКОЕ «ПАТТЕРН УПРАВЛЕНИЯ СОСТОЯНИЕМ»?

Рассмотрим пример:

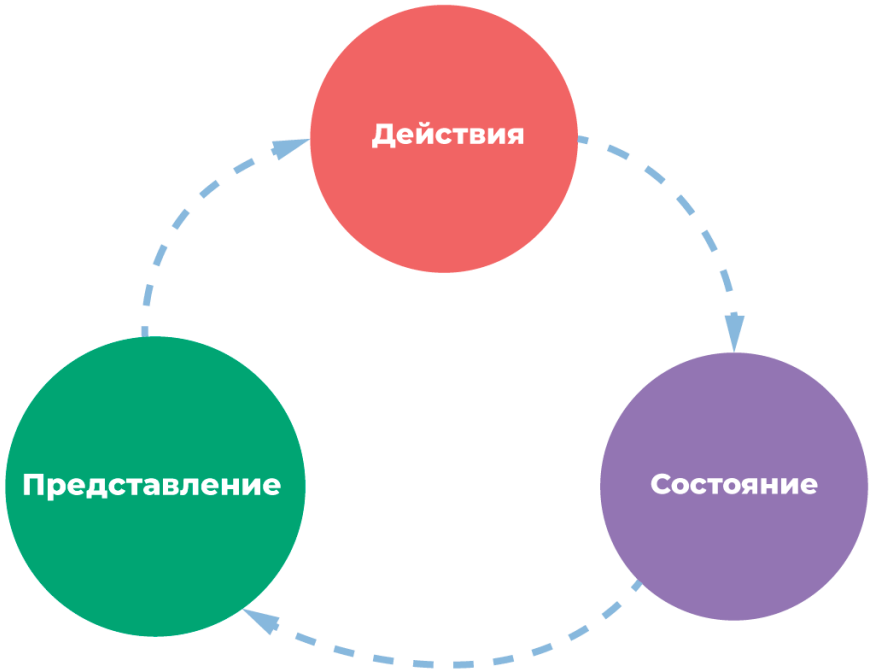
```
<div id="app"></div>
<script>
    new Vue({
        el:'#app',
        // состояние
        data: function() {
            return { count: 0 };
        },
        // представление
        template: `<div>
            <button @click="increment">Увеличить</button>
            <div>{{ count }}</div>
        </div>`,
        // действия
        methods: {
            increment() { this.count++; }
      }
    });
</script>
```

ЧТО ТАКОЕ «ПАТТЕРН УПРАВЛЕНИЯ СОСТОЯНИЕМ»?

Представленное на предыдущем слайде самостоятельное приложение состоит из следующих частей:

- **Состояние** — данные управляющие приложением;
- **Представление** — отображение **состояния**;
- **Действия** — возможные пути изменения состояния приложения в ответ на взаимодействие пользователя с **представлением**.

Вот простейшее представление концепции «однонаправленного потока данных»:



ЧТО ТАКОЕ «ПАТТЕРН УПРАВЛЕНИЯ СОСТОЯНИЕМ»?

Однако простота быстро исчезает, когда у нас появляется нескольких компонентов, основывающихся на одном и том же состоянии:

- Несколько представлений могут зависеть от одной и той же части состояния приложения.
- Действия из разных представлений могут оказывать влияние на одни и те же части состояния приложения.

Решая первую проблему, придётся передавать одни и те же данные входными параметрами в глубоко вложенные компоненты. Это часто сложно и утомительно, а для соседних компонентов такое и вовсе не сработает.

Решая вторую проблему, можно прийти к таким решениям, как обращение по ссылкам к родительским/дочерним экземплярам или попыткам изменять и синхронизировать несколько копий состояния через события.

Оба подхода хрупки и быстро приводят к появлению кода, который невозможно поддерживать.

ЧТО ТАКОЕ «ПАТТЕРН УПРАВЛЕНИЯ СОСТОЯНИЕМ»?

Для решения данных проблем было предложено вынести всё общее состояние приложения из компонентов и управлять им в глобальном единственном месте. При этом наше дерево компонентов становится одним большим «представлением» и любой компонент может получить доступ к состоянию приложения или вызывать действия для изменения состояния, независимо от того, где они находятся в дереве.

Чётко определяя и разделяя концепции, возникающие при управлении состоянием, и требуя соблюдения определённых правил, которые поддерживают независимость между представлениями и состояниями, мы лучше структурируем код и облегчаем его поддержку.

КОГДА СЛЕДУЕТ ИСПОЛЬЗОВАТЬ VUEX?

Vuex помогает управлять совместно используемым состоянием, ценой привнесения новых концепций и вспомогательного кода. Компромисс, когда кратковременная продуктивность страдает на благо долгосрочной.

Простые приложения могут легко обходиться без Vuex. Возможно, будет достаточно простого паттерна глобальной шины событий.

При разработке приложений среднего или крупного размера Vuex поможет лучше управлять состоянием приложения.

ВВЕДЕНИЕ

В центре любого Vuex-приложения находится хранилище. «Хранилище» — это контейнер, в котором хранится состояние вашего приложения. Два момента отличают хранилище Vuex от простого глобального объекта:

- Хранилище Vuex реактивно. Когда компоненты Vue полагаются на его состояние, то они будут реактивно и эффективно обновляться, если состояние хранилища изменяется.

- Нельзя напрямую изменять состояние хранилища. Единственный способ внести изменения — явно вызвать мутацию. Это гарантирует, что любое изменение состояния оставляет след и позволяет использовать инструментарий, чтобы лучше понимать ход работы приложения.

ВВЕДЕНИЕ

Пример создания и простого использования хранилища:

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

store.commit('increment'); //вызов изменения состояния
console.log(store.state.count); /* доступ к объекту состояния
для отслеживания, обычно
```

ВВЕДЕНИЕ

Пример с счётчиков:

```
<div id="app">
  <p>{{ count }}</p>
  <p><button @click="increment">+</button><button @click="decrement">-</button></p>
</div>
<script>
    const store = new Vuex.Store({
        state: { count: 0 },
        mutations: {
            increment: state => state.count++,
            decrement: state => state.count--
        }
    });
    new Vue({
        el: '#app',
        computed: {
            count: function () { return store.state.count }
        },
        methods: {
            increment () { store.commit('increment') },
            decrement () { store.commit('decrement') }
        }
    });
</script>
```

СОСТОЯНИЕ

Vuex использует **единое дерево состояния** — когда один объект содержит всё глобальное состояние приложения и служит «единственным источником истины». Это также означает, что в приложении будет только одно такое хранилище. Единое дерево состояния позволяет легко найти нужную его часть или делать снимки текущего состояния приложения в целях отладки.

Самый простой способ «получения» состояния — просто вернуть часть состояния хранилища в вычисляемом свойстве:

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

Этот паттерн потребует импортировать хранилище в каждом компоненте, который использует его состояние.

СОСТОЯНИЕ

Vuex предоставляет механизм «внедрения» хранилища во всех дочерних компонентах корневого компонента, у которого указана опция store.
Указывая опцию store в корневом экземпляре, мы обеспечиваем доступ к хранилищу во всех дочерних компонентах через this.$store.

```
const Counter = {
    template: `<div>{{ count }}</div>`,
    computed: { count: function() { return this.$store.state.count; } }
};
const app = new Vue({
    el: '#app',
    store,
    components: { Counter },
    template: `<div class="app">
            <counter></counter>
            <p><button @click="increment">+</button>
             <button @click="decrement">-</button></p>
        </div>`,
    methods: {
        increment () { this.$store.commit('increment') },
        decrement () {this.$store.commit('decrement') }
    }
});
```

MAPSTATE

Когда компонент должен использовать множество свойств хранилища, объявлять все эти вычисляемые свойства может быть утомительно. В таких случаях можно использовать функцию mapState, которая автоматически генерирует вычисляемые свойства, которым передается объект state первым аргументом:

```
const Counter = {
    template: `<div>{{ count }}</div>`,
    computed: Vuex.mapState({
        //Сокращение: count: state => state.count;
        count: function(state) {
            return state.count;
        }
    });
};
```

MAPSTATE

Можно в mapState задать синоним для возвращаемого из хранилища значения, т.е.:

- вместо:

```
const Counter = {
    template: `<div>{{countAlias}}</div>`,
    computed: Vuex.mapState({
        countAlias: function(state) {
            return state.count;
        }
    });
};
```

- можно написать:

```
const Counter = {
    template: `<div>{{countAlias}}</div>`,
    computed: Vuex.mapState({
        countAlias: 'count'
    });
};
```

MAPSTATE

Можно передавать массив строк в mapState, когда имя сопоставляемого вычисляемого свойства совпадает с именем в дереве состояний (в хранилище):

```
const Counter = {
    template: `<div>{{ count }}</div>`,
    computed: Vuex.mapState(['count'])
};
```

MAPSTATE

Пример комбинирования вычисляемого свойства с локальным состоянием компонента:

```
const Counter = {
    data: function(){
        return {
            localCount: 5
        }
    },
    template: `<div>{{ count }}</div>`,
    computed: Vuex.mapState({
        count:function(state){
            return state.count + this.localCount;
        }
    })
};
```

MAPSTATE

Функция mapState возвращает объект. С помощью оператора распространения объектов можно добавлять функции от mapState в сочетании с другими локальными вычисляемыми свойствами:

```
const Counter = {
    template: `<div>{{ localCount }} - {{ count }}</div>`,
    computed: {
        localCount: function(){
            return Math.random();
        },
        ...Vuex.mapState({
            count:function(state){
                return state.count;
            }
        })
    }
};
```

ГЕТТЕРЫ

Иногда может потребоваться вычислять производное состояние на основе состояния хранилища, например, отфильтровать список:

```
const store = new Vuex.Store({
    state: { products: [ { id: 1, name: 'name1', type: 'A' }, { id: 2, name: 'name2', type: 'B' }, { id: 3, name: 'name3', type: 'B' }, { id: 4, name: 'name4', type: 'A' } ] }
});

const app = new Vue({
    el: '#app',
    template:`<div>
        <div v-for="product in products">{{product.name}}</div>
    </div>`,
    store,
    computed: {
        products: function(){
            return this.$store.state.products.filter(function(prod){
                return prod.type === 'A';
            });
        }
    }
});
```

Если такие вычисления потребуются более чем в одном компоненте, придётся или дублировать функцию, или выносить её в общий метод, который затем импортировать во всех местах.

ГЕТТЕРЫ

Vuex позволяет определять «геттеры» в хранилище. Можно считать их вычисляемыми свойствами хранилища. Как и вычисляемые свойства, результаты геттера кэшируются, на основе его зависимостей и пересчитываются только при изменении одной из зависимостей. Геттеры получают состояние хранилища первым аргументом и другие гетеры вторым аргументом:

```
const store = new Vuex.Store({
    state: {
        products: [
            { id: 1, name: 'name1', type: 'A' },
            { id: 2, name: 'name2', type: 'B' },
            { id: 3, name: 'name3', type: 'B' },
            { id: 4, name: 'name4', type: 'A' },
        ]
    },
    getters: {
        productsTypeA: function(state, getters){
            state.products.filter(function(prod){
                return prod.type === 'A';
            });
        }
     }
});
```

ГЕТТЕРЫ

Геттеры доступны в объекте store.getters, и вы можете получить доступ к значениям как свойствам:

```
const app = new Vue({
    el: '#app',
    template:`
        <div>
            <div v-for="product in products">{{product.name}}</div>
        </div>
    `,
    store,
    computed: {
        products: function(){
            return this.$store.getters.productsTypeA
        }
    }
});
```

ГЕТТЕРЫ

Можно также передавать аргументы геттерам, возвращая функцию. Это например пригодится, когда необходимо возвращать массив по указанному критерию:

```
const store = new Vuex.Store({
    state: { products: [{ id: 1, name: 'name1', type: 'A' }, { id: 2, name: 'name2', type: 'B' }, { id: 3, name: 'name3', type: 'B' }, { id: 4, name: 'name4', type: 'A' }]
    },
    getters: {
        productsType: function(state, getters){
            return function(type) {
                return state.products.filter(function(prod){
                    return prod.type === type;
                });
            }
        }
    }
});
```

ГЕТТЕРЫ

Пример (продолжение):

```
/*!!! Обратите внимание, что геттеры, доступ к которым выполняется как к методам, будут запускаться каждый раз при их вызове, а результаты не будут кэшироваться.*/
const app = new Vue({
    el: '#app',
    data: { type:'A' },
    template:`<div>
        <div v-for="product in products">{{product.name}}</div>
        <button @click="type='B'">Вернуть тип B</button>
    </div>`,
    store,
    computed: {
        products: function(){
            return this.$store.getters.productsType(this.type);
        }
    }
});
```

ГЕТТЕРЫ

Геттеры, доступ к которым выполняется как к свойствам, будут кэшироваться как часть системы реактивности Vue:

```
const app = new Vue({
    el: '#app',
    data: { type:'A' },
    template:`<div>
        <div v-for="product in products">{{product.name}}</div>
        <button @click="type='B'">Вернуть тип B</button>
    </div>`,
    store,
    computed: {
        products: function(){
            if(this.type === 'A')
                return this.$store.getters.productsTypeA;
            else {
                return this.$store.getters.productsTypeB;
            }
        }
    }
});
```

MAPGETTERS

Функция mapGetters просто проксирует геттеры хранилища в локальные вычисляемые свойства компонента:

```
const app = new Vue({
    el: '#app',
    template:`<div>
        <div v-for="product in productsTypeA">{{product.name}}</div>
    </div>`,
    store,
    computed: {
        ...Vuex.mapGetters([
            'productsTypeA'
        ])
    }
});
```

MAPGETTERS

Функция mapGetters также позволяет задать синоним (другое имя) гетеру:

```
const app = new Vue({
    el: '#app',
    template:`<div>
        <div v-for="product in products">{{product.name}}</div>
    </div>`,
    store,
    computed: {
        ...Vuex.mapGetters({
            products: 'productsTypeA'
        })
    }
});
```

МУТАЦИИ

Единственным способом изменения состояния хранилища во Vuex являются мутации. Мутации во Vuex очень похожи на события: каждая мутация имеет строковый тип и функцию-обработчик. В этом обработчике и происходят, собственно, изменения состояния, переданного в функцию первым аргументом:

```
<div id="app">
  <p>{{ count }}</p><p><button @click="increment">+</button></p>
</div>
<script>
    const store = new Vuex.Store({
        state: { count: 0 },
        mutations: { increment: state => state.count++ }
    });
    new Vue({
        el: '#app', store, computed: Vuex.mapState(['count']),
        methods: {
            increment () {
                store.commit('increment');
            }
        }
    });
</script>
```

МУТАЦИИ

При вызове store.commit в мутацию можно также передать дополнительный параметр, называемый нагрузкой (payload):

```
<div id="app">
  <p>{{ count }}</p><p><button @click="increment">+</button></p>
</div>
<script>
    const store = new Vuex.Store({
        state: { count: 0 },
        mutations: { increment: (state, n) => state.count+=n }
    });
    new Vue({
        el: '#app', store, computed: Vuex.mapState(['count']),
        methods: {
            increment () {
                store.commit('increment', 10);
            }
        }
    });
</script>
```

МУТАЦИИ

В большинстве случаев нагрузка будет объектом, содержащим несколько полей. Запись мутаций в таком случае становится более описательной:

```
<div id="app">
  <p>{{ count }}</p><p><button @click="increment">+</button></p>
</div>
<script>
    const store = new Vuex.Store({
        state: { count: 0 },
        mutations: { increment: (state, payload) => state.count+= payload.amount }
    });
    new Vue({
        el: '#app', store, computed: Vuex.mapState(['count']),
        methods: {
            increment () {
                store.commit('increment', {
                    amount: 10
                });
            }
        }
    });
</script>
```

МУТАЦИИ

Другой способ вызвать мутацию — это передать в commit единственный объект, в котором type указан напрямую:

```
<div id="app">
  <p>{{ count }}</p><p><button @click="increment">+</button></p>
</div>
<script>
    const store = new Vuex.Store({
        state: { count: 0 },
        mutations: { increment: (state, payload) => state.count+= payload.amount }
    });
    new Vue({
        el: '#app', store, computed: Vuex.mapState(['count']),
        methods: {
            increment () {
                store.commit({
                    type: 'increment', amount: 10
                });
            }
        }
    });
</script>
```

МУТАЦИИ

Поскольку состояние хранилища Vuex — это реактивная переменная Vue, при возникновении мутации зависящие от этого состояния компоненты Vue обновляются автоматически. Кроме того, это значит, что мутации Vuex имеют те же самые подводные камни, что и реактивность в обычном Vue:

- Лучше инициализировать изначальное состояние хранилища, указав все поля в самом начале.

- При добавлении новых свойств объекту необходимо либо:

  - Использовать Vue.set(obj, 'newProp', 123), или

  - Целиком заменить старый объект новым. Например, используя синтаксис расширения объектов можно написать так:

```
state.obj = { ...state.obj, newProp: 123 };
```

МУТАЦИИ

!!!Нужно помнить одно важное правило: обработчики мутаций обязаны быть синхронными. (иначе инструмент devtools не сможет отследить мутации).

Для асинхронный функций, есть отдельные сущности Vuex называемые «Действия», рассматриваются в этой презентации немного позже.

MAPMUTATIONS

Мутации можно вызывать из кода компонентов, используя this.$store.commit('xxx'), или применяя вспомогательный метод mapMutations, который проксирует вызовы store.commit через методы компонентов (для этого требуется наличие корневой ссылки на хранилище $store):

```
<div id="app">
  <p>{{ count }}</p><p><button @click="add">+</button><button
@click="incrementBy(10)">+</button></p>
</div>
<script>
    const store = new Vuex.Store({
        state: { count: 0 },
        mutations: {
            increment: state => state.count++,
            incrementBy: (state, amount) => state.count+=amount
        }
    });
```

MAPMUTATIONS

Пример (продолжение):

```
    new Vue({
        el: '#app',
        store,
        computed: Vuex.mapState(['count']),
        methods: {
            ...Vuex.mapMutations(['incrementBy']),
            ...Vuex.mapMutations({add:'increment'})
        }
    });
</script>
```

ДЕЙСТВИЯ

Действия — похожи на мутации с несколькими отличиями:

❖ Вместо того, чтобы напрямую менять состояние, действия инициируют мутации;

❖ Действия могут использоваться для асинхронных операций.

```
<div id="app">
  <p>{{ count }}</p><p><button @click="add">+</button></p>
</div>
<script>
      const store = new Vuex.Store({
            state: { count: 0 },
            mutations: {
                increment(state) { state.count++; }
            },
            actions: {
                increment(context) { context.commit('increment'); }
            }
      });
/*Обработчики действий получают объект контекста, содержащий те же методы и свойства, что и сам экземпляр хранилища (для данного случая).*/
```

ДЕЙСТВИЯ

Пример (продолжение):

```
new Vue({
        el: '#app',
        store,
        computed: Vuex.mapState(['count']),
        methods: {
            //Действия запускаются методом store.dispatch
            add(){ store.dispatch('increment'); }
        }
});
</script>
```

ДЕЙСТВИЯ

Внутри действий можно выполнять асинхронные операции:

```
<div id="app">
  <p>{{ count }}</p><p><button @click="add">+</button></p>
</div>
<script>
    const store = new Vuex.Store({
        state: { count: 0 },
        mutations: {
            increment(state) { state.count++; }
        },
        actions: {
            /*На практике для упрощения кода часто используется
деструктуризация аргументов*/
            incrementAsync ({ commit }) {
                setTimeout(() => { commit('increment') }, 1000)
            }
        }
    });
```

ДЕЙСТВИЯ

Пример (продолжение):

```
new Vue({
        el: '#app',
        store,
        computed: Vuex.mapState(['count']),
        methods: {
            //Действия запускаются методом store.dispatch
            add(){ store.dispatch('incrementAsync'); }
        }
});
</script>
```

ДЕЙСТВИЯ

Действия поддерживают тот же формат для передачи нагрузки: а также объектный синтаксис:

```
const store = new Vuex.Store({
    state: { count: 0 },
    mutations: {
        increment(state, amount) { state.count+=amount; }
    },
    actions: {
        incrementAsync ({ commit }, obj) {
            setTimeout(() => { commit('increment', obj.amount) }, 1000)
        }
    }
});
new Vue({
    el: '#app', store, computed: Vuex.mapState(['count']),
    methods: {
        add: function(){ // вызов с нагрузкой
            store.dispatch('incrementAsync', { amount: 10 });
        }
    }
});
```

ДЕЙСТВИЯ

Действия поддерживают также объектный синтаксис:

```
new Vue({
    el: '#app',
    store,
    computed: Vuex.mapState(['count']),
    methods: {
        add: function(){
            store.dispatch({ // объектный синтаксис
                type: 'incrementAsync',
                amount: 10
            });
        }
    }
});
```

MAPACTIONS

Диспетчеризировать действия в компонентах можно при помощи this.$store.dispatch('xxx') или используя вспомогательную функцию mapActions, создающую локальные псевдонимы для действий в виде методов компонента (требуется наличие корневого $store):

```
<div id="app">
  <p>{{ count }}</p><p><button @click="add">+</button>
  <button @click="incrementByAsync(10)">+</button></p>
</div>
<script>
    const store = new Vuex.Store({
        state: { count: 0 },
        mutations: {
            increment: state => state.count++,
            incrementBy: (state, amount) => state.count+=amount
        },
        actions: {
            incrementByAsync ({ commit }, amount) {
                setTimeout(() => { commit('incrementBy', amount) }, 1000)
            },
            incrementAsync ({ commit }) {
                setTimeout(() => { commit('increment') }, 1000)
            }
        }
    });
```

MAPACTIONS

Пример (продожение):

```
 new Vue({
        el: '#app',
        store,
        computed: Vuex.mapState(['count']),
        methods: {
            ...Vuex.mapActions(['incrementByAsync']),
            ...Vuex.mapActions({add:'incrementAsync'})
        }
    });
</script>
```

КОМПОЗИЦИЯ ДЕЙСТВИЙ

❖ store.dispatch может обрабатывать Promise, возвращаемый обработчиком действия, и также возвращает Promise

```
<div id="app">
  <p>{{ count }}</p><p><button @click="add">+</button></p>
</div>
<script>
    const store = new Vuex.Store({
        state: { count: 0 },
        mutations: { increment: state => state.count++ },
        actions: {
            incrementAsync ({ commit }) {
                return new Promise((resolve, reject) => {
                    setTimeout(() => {
                      commit('increment');
                      resolve();
                    }, 1000)
                })
            }
        }
    });
```

КОМПОЗИЦИЯ ДЕЙСТВИЙ

Пример (продолжение):

```
 new Vue({
        el: '#app',
        store,
        computed: Vuex.mapState(['count']),
        methods: {
            add:function(){
                this.$store.dispatch('incrementAsync').then(() => {
                    alert("Успех");
                });
            }
        }
    });
</script>
```

МОДУЛИ

Из-за использования единого дерева состояния, все глобальные данные приложения оказываются помещены в один большой объект. По мере роста приложения, хранилище может существенно раздуться.

Чтобы помочь в этой беде, Vuex позволяет разделять хранилище на модули. Каждый модуль может содержать собственное состояние, мутации, действия, геттеры и даже встроенные подмодули.

```
const moduleA = {
  state: { ... },
  mutations: { ... },
  actions: { ... },
  getters: { ... }
}

const moduleB = {
  state: { ... },
  mutations: { ... },
  actions: { ... }
}
```

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

store.state.a // -> состояние модуля `moduleA`
store.state.b // -> состояние модуля `moduleB`
```

МОДУЛИ

Первым аргументом, который получают мутации и геттеры, будет локальное состояние модуля.

```
const moduleA = {
  state: { count: 0 },
  mutations: {
    increment(state) {
      // `state` указывает на локальное состояние модуля
      state.count++;
    }
  },

  getters: {
    doubleCount(state) {
      return state.count * 2;
    }
  }
};
```

МОДУЛИ

Аналогично, context.state в действиях также указывает на локальное состояние модуля, а корневое — доступно в context.rootState:

```
const moduleA = {
  // ...
  actions: {
    incrementIfOddOnRootSum({ state, commit, rootState }) {
      if ((state.count + rootState.count) % 2 === 1) {
        commit('increment');
      }
    }
  }
};
```

В геттеры корневое состояние передаётся 3-м параметром:

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

МОДУЛИ

По умолчанию действия, мутации и геттеры внутри модулей регистрируются в глобальном пространстве имён — это позволяет нескольким модулям реагировать на тот же тип мутаций/действий.

```
<div id="app">
  <p>Корень: {{ count }}</p>
  <p>МодульA: {{ countA }}</p>
  <p>МодульB: {{ countB }}</p>
  <p><button @click="add">+</button></p>
</div>
<script>
    let moduleA = {
        state: { count: 0 },
        mutations: {
            increment(state) { state.count++; }
        },
        actions: {
            incrementAsync({ commit }) {
                setTimeout(() => { commit('increment') }, 1000)
            }
        }
    }
```

МОДУЛИ

Пример (продолжение):

```
let moduleB = {
    state: { count: 0 },
    mutations: {
        increment(state) { state.count++; }
    },
    actions: {
        incrementAsync({ commit }) {
            setTimeout(() => {
                commit('increment')
            }, 1000)
        }
    }
}
const store = new Vuex.Store({
    state: { count: 0 },
    modules: {
        a: moduleA,
        b: moduleB
    }
});
```

МОДУЛИ

Пример (продолжение):

```
 new Vue({
        el: '#app',
        store,
        computed: {
            count () { return this.$store.state.count; },
            countA () { return this.$store.state.a.count; },
            countB () { return this.$store.state.b.count; }
        },
        methods: {
            add: function(){
                this.$store.dispatch('incrementAsync');
            }
        }
    });
</script>
```

МОДУЛИ

Если вы хотите сделать модули более самодостаточными и готовыми для переиспользования, вы можете создать его с собственным пространством имён, указав опцию namespaced: true.

```
<div id="app">
  <p>Корень: {{ count }}</p>
  <p>МодульA: {{ countA }}</p><p>МодульB: {{ countB }}</p>
  <p><button @click="add">+</button></p>
</div>
<script>
    let moduleA = {
        namespaced: true,
        state: { count: 0 },
        mutations: {
            increment(state) { state.count++; }
        },
        actions: {
            incrementAsyncA({ commit }) {
                setTimeout(() => { commit('increment') }, 1000)
            }
        }
    }
```

МОДУЛИ

Пример (продолжение):

```
let moduleB = {
    namespaced: true,
    state: { count: 0 },
    mutations: {
        increment(state) { state.count++; }
    },
    actions: {
        incrementAsync({ commit }) {
            setTimeout(() => {
                commit('increment')
            }, 1000)
        }
    }
}
const store = new Vuex.Store({
    state: { count: 0 },
    modules: {
        a: moduleA,
        b: moduleB
    }
});
```

МОДУЛИ

Пример (продолжение):

```
 new Vue({
        el: '#app',
        store,
        computed: {
            count () { return this.$store.state.count; },
            countA () { return this.$store.state.a.count; },
            countB () { return this.$store.state.b.count; }
        },
        methods: {
            add: function(){
                this.$store.dispatch('a/incrementAsync');
            }
        }
    });
</script>
```

Геттеры и действия с собственным пространством имён будут получать свои локальные getters, dispatch и commit.

МОДУЛИ

Если вы хотите использовать глобальное состояние и геттеры, rootState и rootGetters передаются 3-м и 4-м аргументами в функции геттеров, а также как свойства в объекте context, передаваемом в функции действий.

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

МОДУЛИ

Для запуска действий или совершения мутаций в глобальном пространстве имён нужно добавить { root: true } 3-м аргументом в dispatch и commit.

```
const moduleA = {
  // ...
  actions: {
      incrementAsync({ commit }) {
          setTimeout(() => {
              commit('increment', null, { root:true })
          }, 1000)
      }
  }
};
```

МОДУЛИ

Подключение модуля со своим пространством имён к компонентам с помощью вспомогательных функций mapState, mapGetters, mapActions и mapMutations это может выглядеть подобным образом:

```
new Vue({
    el: '#app', store,
    computed: {
        ...Vuex.mapState({
            count: (state) => state.count,
            countA: (state) => state.a.count,
            countB: (state) => state.b.count,
        })
    },
    methods: {
        ...Vuex.mapActions({
            add: 'a/incrementAsync'
        })
    }
});
```

МОДУЛИ

Вы можете зарегистрировать модуль уже и после того, как хранилище было создано, используя метод store.registerModule:

```
// регистрация модуля `myModule`
store.registerModule('myModule', {
  // ... state, getters и т.п.
});

// регистрация вложенного модуля `nested/myModule`
store.registerModule(['nested', 'myModule'], {
 // ... state, getters и т.п.
});
```

Состояние модуля будет доступно как store.state.myModule и store.state.nested.myModule.

ДОПОЛНИТЕЛЬНЫЕ ССЫЛКИ

Официальное руководство по Vuex:
https://vuex.vuejs.org/ru/guide/
Официальное руководство по Vue.js:
https://ru.vuejs.org/v2/guide/