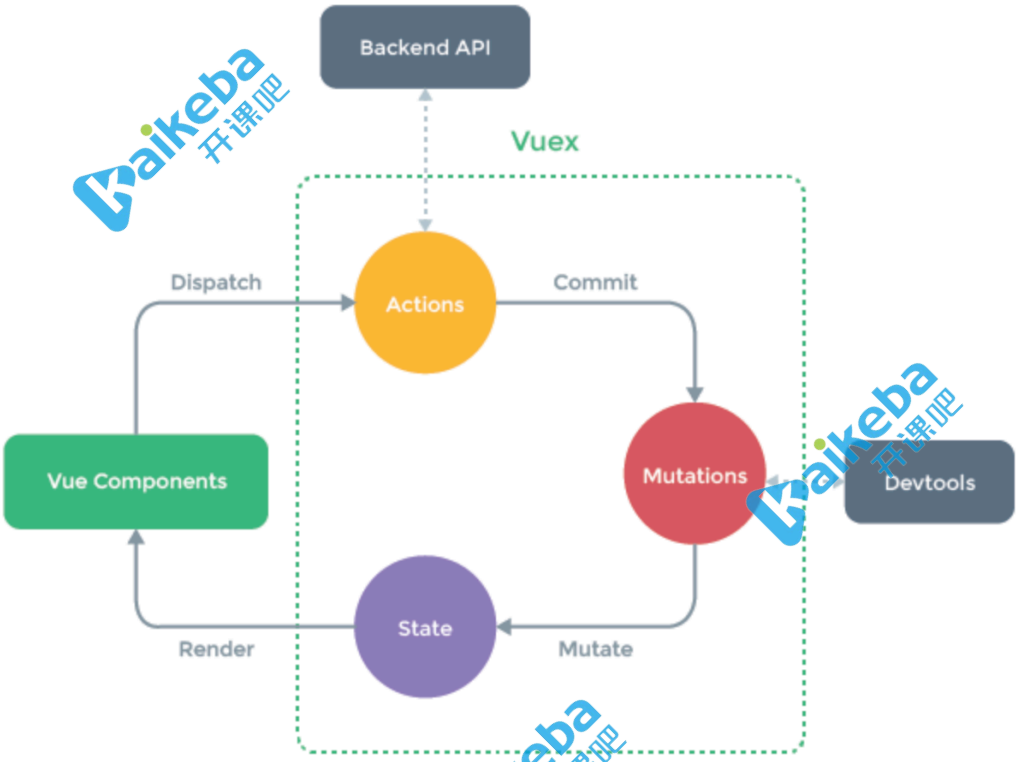
1.7 状态管理 - Vuex



vuex是一个vue应用状态管理库，它用于**统一状态存储**，并保证状态变更的**可预测性**。

> Vuex4用于和vue3搭配使用

1.7.1 起步

cdn方式

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

> 确保在vue后面引入`vuex`

NPM

```
npm install vuex@next --save
```

Yarn

```
yarn add vuex@next --save
```

快速起始

创建store实例并使用它：

```
import { createApp } from 'vue'
import { createStore } from 'vuex'

// 1.创建store实例
const store = createStore({
  state () {
    return {
      count: 0
    }
  },
  mutations: {
    inc (state) {
      state.count++
    }
  }
})

const app = createApp({ /* your root component */ })

// 2.使用store
app.use(store)
```

在视图中使用

```
<p>{{$store.state.count}}</p>
```

修改状态

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

1.7.2 核心用法

状态 - State

通常一个**应用只有一棵单状态树**，这样定位某个状态片段更加快速直接，调试时更容易获取当前app状态快照。

state常用手法

- 计算属性

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

- helper方法：mapState

  数组方式：最简单直接的方式

```
import { mapState } from 'vuex'
export default {
  computed: mapState(['count'])
}
```

  对象方式：更加灵活强大

```
import { mapState } from 'vuex'
export default {
  computed: mapState({
    // 箭头函数方式比较简洁
    count: state => state.count,

    // 传递字符串等效于上面用法
    countAlias: 'count',

    // 需要通过this访问组件内部状态时必须使用普通函数方式
    countPlusLocalState (state) {
      return state.count + this.localCount
    }
  })
}
```

> 展开mapState返回对象，用于和其他computed属性共存

```
computed: {
  localComputed() {},
  ...mapState({})
}
```

派生状态 - Getters

可以利用Getters方式将store状态派生出新的状态。

定义getters

```
getters: {
  // 方式1：Property-Style，最简单常用的方式
  doubleCount(state) {
    return state.count * 2
  }
  // 方式2：Method-Style，可以传参的方式
  nCount(state) {
    return (n) => {
      return state.count * n
    }
  }
}
```

使用getters

```
<p>{{$store.getters.doubleCount}}</p>
<p>{{$store.getters.nCount(3)}}</p>
```

使用计算属性简化

```
computed: {
  doubleCount() {
    return this.$store.getters.doubleCount
  }
},
methods: {
  // nCount适合用method映射
  nCount(n) {
    return this.$store.getter.nCount(n)
  }
}
```

使用mapGetters简化

```
computed: {
  ...mapGetters(['doubleCount'])
}
```

变更 - Mutations

Mutations是唯一改变状态的方式

定义mutations

```
mutations: {
  inc (state) {
    state.count++
  },
  incBy (state, n) {
    state.count += n
  }
}
```

> 使用常量定义mutations type也比较常见

```
// mutation-types.js
export const COUNT_INC = 'COUNT_INC'
```

```
// store.js
import { COUNT_INC } from './mutation-types'

const store = createStore({
  state: { ... },
  mutations: {
    // 利用计算属性定义type
    [COUNT_INC](state) {}
  }
})
```

提交mutation

```
this.$store.commit('inc') // 不带参数
this.$store.commit('incBy', 2) // 携带参数
```

> 对象风格的commit，type是必须的
>
> ```
> this.$store.commit({ type: 'incBy', num: 2 }) // 对象用法
> ```
>
> mutation通过载荷payload获取参数
>
> ```
> incBy (state, payload) {
>   state.count += payload.num
> }
> ```

利用mapMutations简化调用

```
import { mapMutations } from 'vuex'

export default {
  methods: {
    ...mapMutations(['inc','incBy']),
  }
}
```

模板中调用

```
<p @click="inc">{{count}}</p>
```

动作 - Actions

动作类似于mutations，它们主要用于：

- 实现复杂业务逻辑
- 处理异步操作

定义actions

```
actions: {
  inc (context) {
    setTimeout(() => {
      context.commit('inc')
    }, 1000)
  }
}
```

> 解构上下文简化写法
>
> ```
> actions: {
>   inc ({commit}) {
>     setTimeout(() => {
>       commit('inc')
>     }, 1000)
>   }
> }
> ```
>
> 上下文中还包含state、dispatch、getters等，可用于负责业务组合
>
> ```
> actions: {
>   inc ({ commit, dispatch, getters }) {}
> }
> ```

派发action

```
store.dispatch('inc')
store.dispatch('incBy', 10) // 带参数
```

使用mapActions简化

```
import { mapActions } from 'vuex'

export default {
  methods: {
    ...mapActions(['inc', 'incBy'])
  }
}
```

处理action结果

action返回一个Promise，调用者可以用来处理异步结果。

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

1.7.3 模块化 - Modules

可以利用模块化拆分store定义避免状态树过大时难以维护。

常用手法

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

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

```
const store = createStore({
  modules: {
    a: moduleA,
    b: moduleB
  }
})

store.state.a // -> `moduleA`'s state
store.state.b // -> `moduleB`'s state
```

范例：将count提取为模块

```
// count.js，可以将之前配置原封不动移过来
import { COUNT_INC } from "./mutation-types";

export default {
  state() {},
  getters: {
    doubleCount(state) {},
    nCount(state) {},
  },
  mutations: {
    inc(state) { },
    [COUNT_INC](state) { },
    incBy(state, n) {},
  },
  actions: {
    inc({ commit }) {},
  },
}
```

注册模块

```
// store/index.js
import count from "./count";

const store = createStore({
  modules: {
    count,
  },
});
```

试一下发现，除了count3之外都正常，这里通过`store.state.count`获取的是整个count模块的state对象，想要获取里面的值，应该用`store.state.count.count`

{ "count": 31 }

incBy: { "count": 31 }

COUNT_INC: { "count": 31 }

action inc: [object Object]10

doubleCount: 62

nCount: 310

`store.state.count.count`容易引起歧义，模块名或者模块内部的属性修改一下会更好

```
state() {
  return {
    value: 0, // 这里把之前的count改为value，注意其他引用的地方也要改
  };
},
```

使用时就可以像下面这样

```
...mapState({
  count1: state => state.count.value,
})
```

命名空间

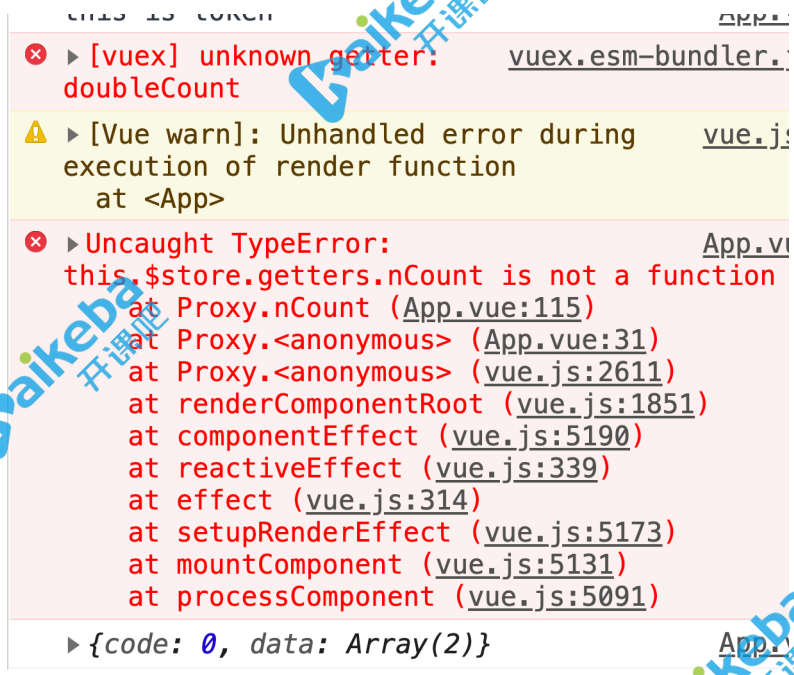
上面的模块化划分，仅仅将state隔离，actions/mutations依然注册在全局命名空间，意味着多个模块会同时响应相同的action/mutation类型，同时getters也注册在全局命名空间，在编写模块的getters时要小心不能出现重名，否则会报错。

为了保证更加隔离和重用效果，最好给模块加上命名空间选项：`namespaced: true`，这样所有actions/mutations/getters都将注册在独立的命名空间中。

范例：修改模块为独立命名空间

```
export default {
  namespaced: true,
}
```

App中引用将会报错



为了正常使用，需要在映射时添加模块名作为帮助方法的参数1：

```
...mapGetters("count", ["doubleCount"]),
...mapMutations("count", ["inc", "incBy"]),
...mapActions("count", { incAsync: "inc" }),
```

```
nCount(n) {
  // 代码里访问时使用path
  return this.$store.getters["count/nCount"](n);
},
```

在命名空间模块内访问全局资源

```
getters: {
  // 参数3/4用于访问全局状态/派生状态
  someGetter (state, getters, rootState, rootGetters) {
    rootState.foo // 根状态'foo'
    rootState['bar/foo'] // bar模块状态'foo'
  },
},

actions: {
  // 解构rootState, rootGetters访问全局状态/派生状态
  someAction ({ dispatch, commit, getters, rootState, rootGetters }) {
    rootState.foo // 根状态'foo'
    rootState['bar/foo'] // bar模块状态'foo'

    // 派发全局注册的action
    dispatch('someOtherAction', null, { root: true })
```

```
    // 派发bar模块中的action
    dispatch('bar/someOtherAction', null, { root: true })

    // 提交全局注册的mutation
    commit('someMutation', null, { root: true })
    // 提交bar模块中的mutation
    commit('bar/someMutation', null, { root: true })
  },
}
```