VUEX 4

状态管理

PART01

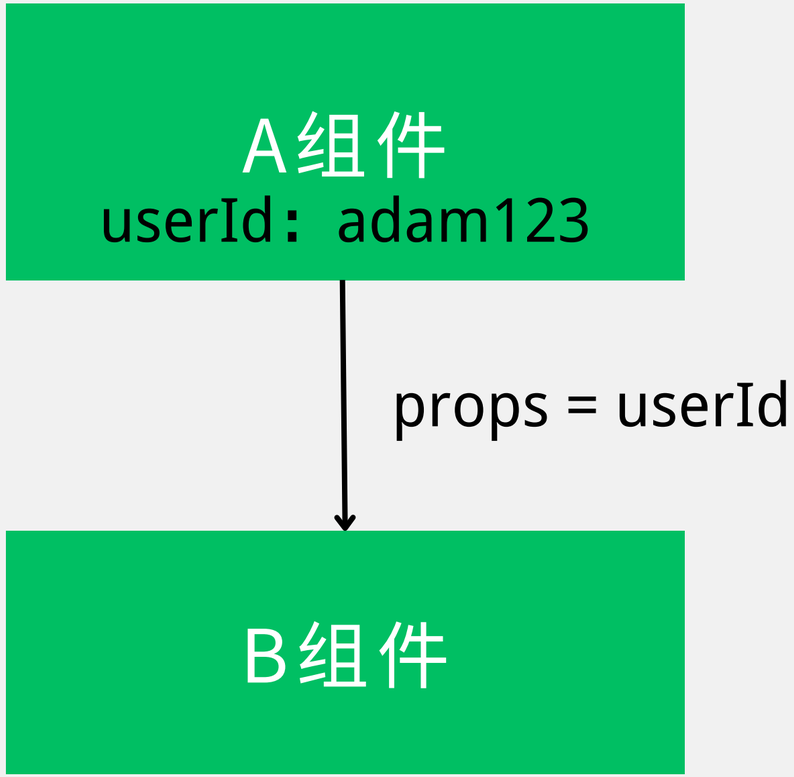
什么是
状态管理

什么是状态管理

状态管理是一种软件开发中的设计模式，它主要关注如何在应用程序中存储、操作和维护状态。状态是指应用程序在特定时间点的数据和信息，例如用户信息、应用设置、购物车内容等。在前端开发中，状态管理主要解决组件之间的状态共享和通信问题。

简单来说，状态管理就像一个中央仓库，存储着应用程序中所有组件的共享状态。这个仓库让开发者能够更加清晰、有序地处理组件间的状态共享和通信，使得整个应用变得更加稳定和可预测。

在没有状态管理的情况下，组件之间的状态共享和通信可能变得非常复杂，导致代码难以维护和扩展。状态管理库（如 Vuex）提供了一种集中式、规范的方式来管理状态，使开发者能够更有效地处理应用中的状态变化。



组件之间数据传递

A组件内有一个数据是用户账号userId

此时B组件也需要用到userId这个数据

则可以通过props进行数据的传递

状态管理vuex

有一个仓库（store）用于管理所有的状态（state），当项目中的组件要用到数据时则到仓库直接获取数据。

	PROPS	VUEX
组件层级关系	数据需要沿着组件层级逐级传递，当组件层级较深时，这种方式会导致代码冗余和难以维护。	状态存储在集中式的仓库中，任何组件都可以直接访问和更新状态，避免了层级传递的问题。
状态共享	如果多个组件需要共享同一个状态，开发者需要在多个地方维护并同步状态，这会导致代码的复杂性增加。	由于状态存储在集中式的仓库中，多个组件可以直接访问和更新共享状态，降低了代码的复杂性。
状态修改和追踪	状态的修改可能会分散在各个组件中，导致状态变化难以追踪和预测。	所有状态的修改都通过预定义的规则进行（如 Vuex 中的 mutations），使得状态变化更加可预测和可追踪。
跨组件通信	跨组件通信需要通过事件监听和回调函数实现，这会增加代码的复杂性。	跨组件通信可以通过共享状态和预定义的操作（如 Vuex 中的 actions）实现，简化了跨组件通信的处理。
可维护性和可扩展性	随着应用的复杂度增加，状态传递和共享的管理变得困难，影响代码的可维护性和可扩展性。	集中式的状态管理和预定义的规则有助于应对复杂应用，提高代码的可维护性和可扩展性。

ACHIEVEMENT
SHOW

PART02
VUEX

vuex介绍

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式和库。它采用集中式存储管理应用的所有组件状态，并通过一些规则保证状态以一种可预测的方式发生变化。Vuex 的核心概念包括 state、getters、mutations、actions 和 modules。在复杂的单页面应用（SPA）中，Vuex 可以帮助开发者更好地管理和组织代码，提高应用的可维护性和可扩展性。

vuex介绍

Vuex 的核心概念包括 state、 getters、 mutations、 actions 和 modules。

仓库store

state(状态) 如number: 5

getters(计算属性) 如doubleNumber = number*2

mutations(修改状态) 如number+1

actions(修改状态) 允许包含异步操作

modules(模块化机制)

*VUEX*的核心概念

STATE

STATE 是 VUEX 中的数据源，用于存储应用程序的状态信息。它是一个对象，包含了应用中所有需要共享的状态数据。所有组件可以从 STATE 中读取数据，并在需要时通过 MUTATIONS 修改数据。在 VUEX 中，STATE 应该是唯一数据源，以保持应用状态的一致性。

GETTERS

GETTERS 是 VUEX 中的计算属性，类似于 VUE 中的 COMPUTED 属性。它们用于从 STATE 中派生出一些新的状态，例如对 STATE 中的数据进行过滤、排序等操作。GETTERS 是基于 STATE 的响应式计算，当依赖的 STATE 发生变化时，GETTERS 也会自动更新。

MUTATIONS

MUTATIONS 是 VUEX 中唯一修改 STATE 的方法。它们是一些预定义的函数，用于对 STATE 进行同步更新。MUTATIONS 应该是纯函数，不涉及异步操作和副作用，这有助于保持状态的可预测性和可追踪性。在组件中，通常使用 COMMIT 方法调用 MUTATIONS 来修改 STATE。

ACTIONS

ACTIONS 类似于 MUTATIONS，但它们允许包含异步操作。ACTIONS 不直接修改 STATE，而是通过提交（COMMIT）MUTATIONS 来实现。ACTIONS 可以执行异步操作，例如 API 请求、定时任务等，然后在操作完成后提交 MUTATIONS。在组件中，通常使用 DISPATCH 方法调用 ACTIONS。

MODULES

MODULES 是 VUEX 中的模块化机制。在复杂的应用中，将所有状态、MUTATIONS、ACTIONS 和 GETTERS 放在一个文件中可能会导致代码难以维护。MODULES 允许将 STORE 分割成多个模块，每个模块具有自己的 STATE、MUTATIONS、ACTIONS 和 GETTERS。

VUEX的角色

集中式状态管理

在大型应用中，组件之间的状态共享和通信变得更加复杂。VUEX 通过集中管理应用的状态，简化了这些操作，提高了代码的可读性和可维护性。

可预测的状态变化

VUEX 使用严格的规则，确保状态的变化是可追踪和可预测的。这有助于避免潜在的 BUG 和不稳定的行为。

调试工具支持

借助 VUE DEVTOOLS 等调试工具，开发者可以轻松地查看和追踪 VUEX 中的状态变化，提高开发效率。

PART03

VUEX的准备工作

1.安装

```
npm install vuex@next
```

2. 配置

src/store/index.js


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

const store = createStore({
  state: {
    number: 5,
  },
});

export default store;
```

3. 全局注册

main.js

```
import { createApp } from "vue";
import App from "./App.vue";

import "./assets/main.css";

import store from "./store";

const app = createApp(App);
app.use(store);
app.mount("#app");
```

PART04

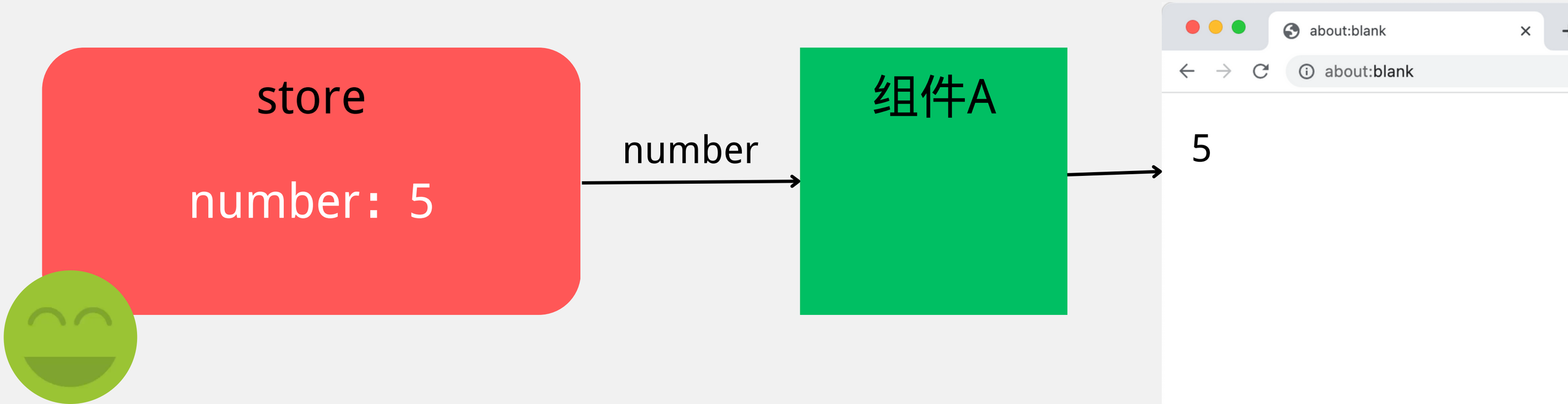
VUEX的应用

1.state 状态

接下来要做的事

1.在仓库中设置状态number

2.在组件中去获取number并渲染到浏览器上



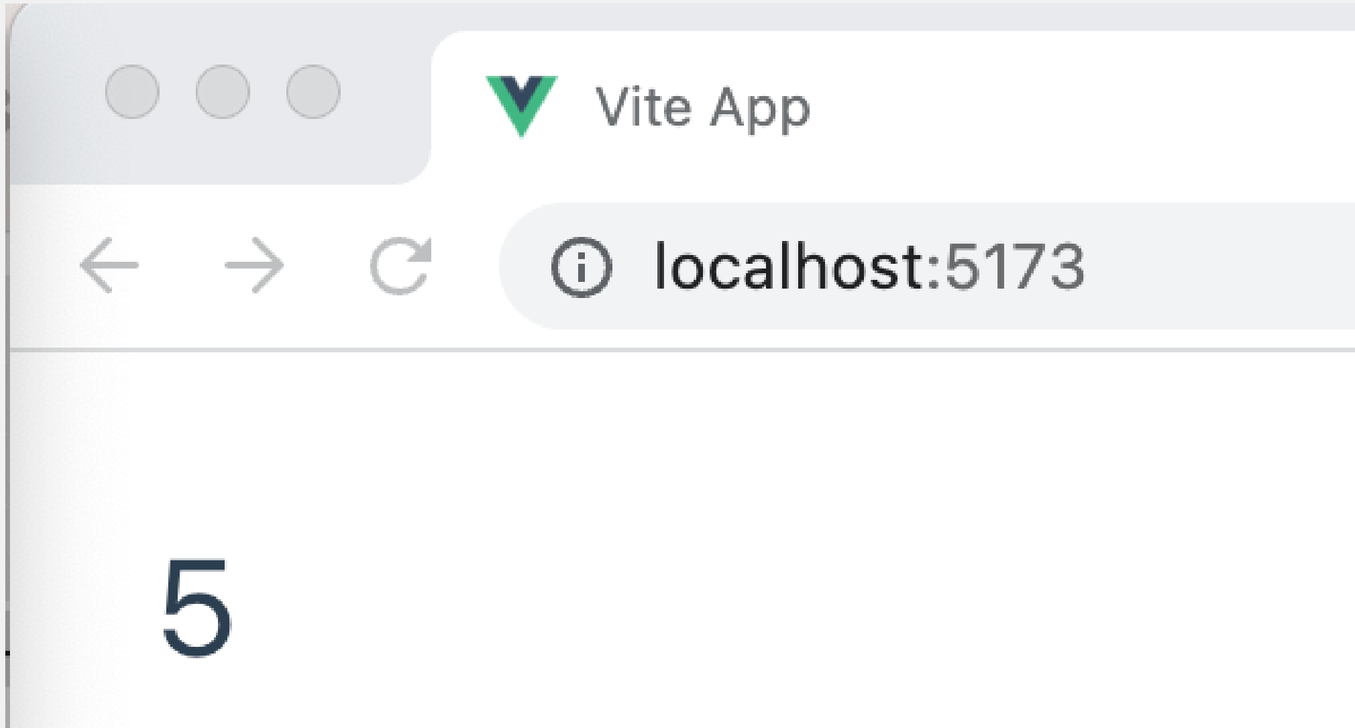
1 src/store/index.js

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

const store = createStore({
  state: {
    number: 5,
  },
});

export default store;
```

2 App.vue

```
<template>
  <h1>{{ this.$store.state.number }}</h1>
</template>
```

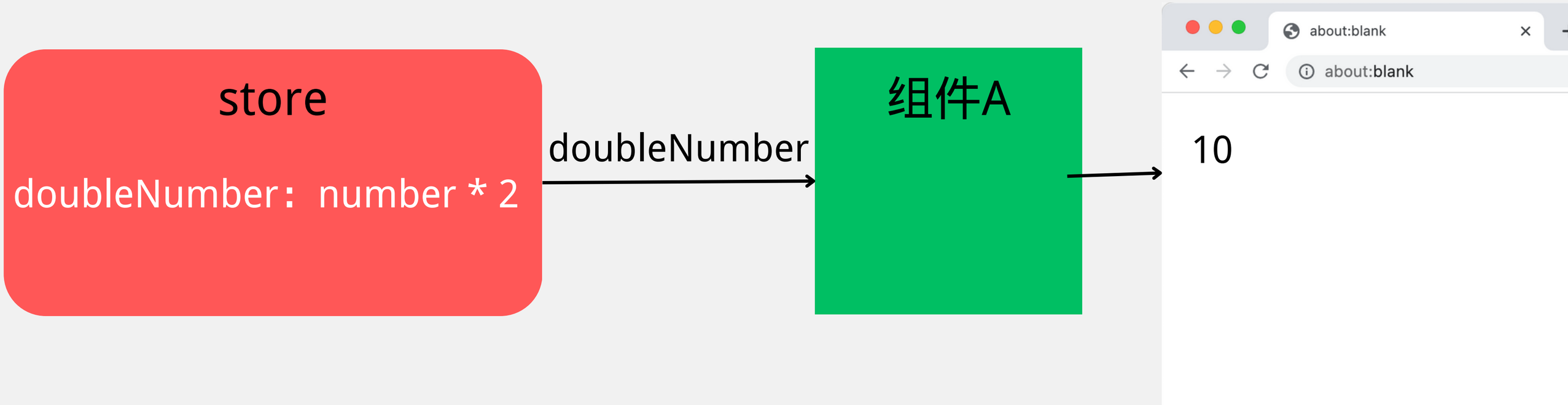
3 浏览器



2.getters 计算属性

接下来要做的事

1. 在仓库中增加计算属性getters
2. 计算doubleNumber
3. 在组件中获取doubleNumber并渲染到浏览器上



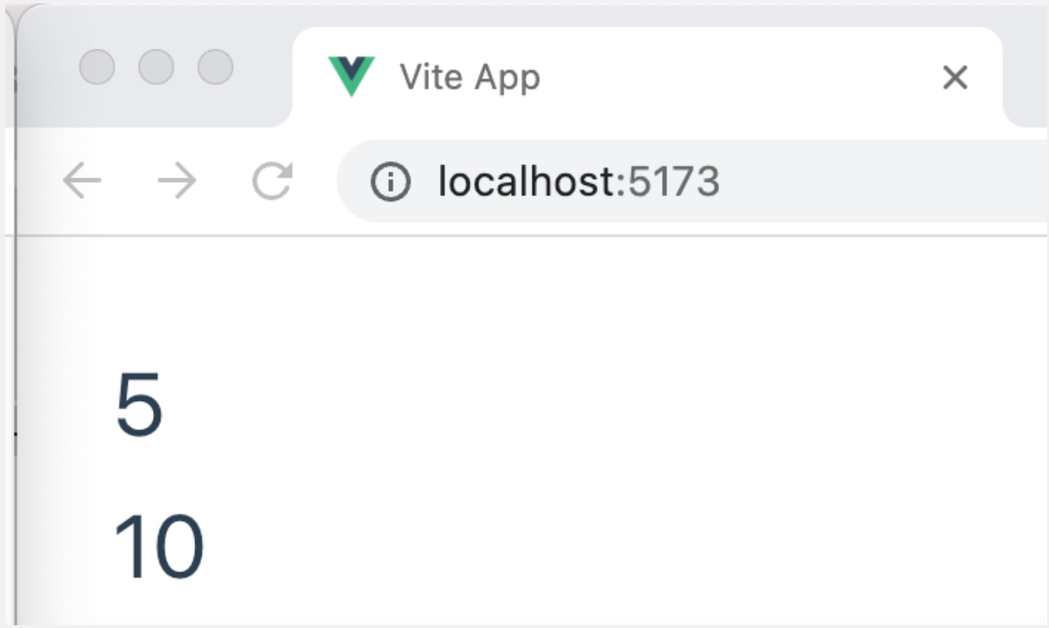
1 src/store/index.js

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

const store = createStore({
  state: {
    number: 5,
  },
  getters: {
    doubleNumber(state) {
      return state.number * 2;
    },
  },
});

export default store;
```

2 App.vue

```
<template>
  <div>
    <h1>{{ this.$store.state.number }}</h1>
    <h1>{{ this.$store.getters.doubleNumber }}</h1>
  </div>
</template>
```

3 浏览器



3.mutations 修改状态

接下来要做的事

1.在仓库中增加一个mutations，命名为increment，用于将number+1
2.在组件中做一个按钮以调用increment，从而将number变成6

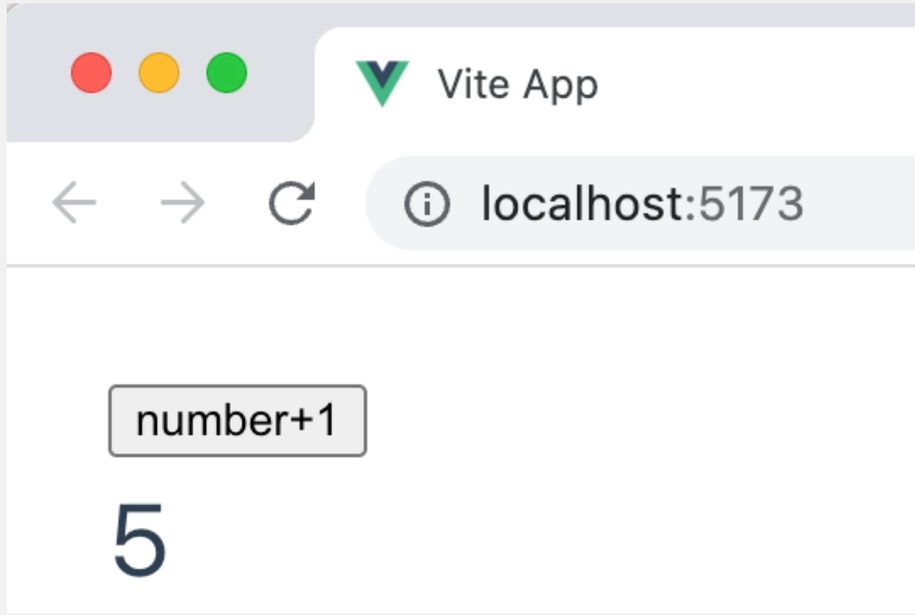
store

increment：number++

组件A

number+1

调用increment



1 src/store/index.js

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

const store = createStore({
  state: {
    number: 5,
  },
  mutations: {
    increment(state) {
      state.number ++
    },
  },
});

export default store;
```

2 App.vue

```
<template>
  <div>
    <button @click="increNumber">number+1</button>
    <h1>{{ this.$store.state.number }}</h1>
  </div>
</template>

<script>
export default {
  methods: {
    increNumber() {
      this.$store.commit("increment");
    },
  },
};
</script>
```

载荷

1 src/store/index.js

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

const store = createStore({
  state: {
    number: 5,
  },
  mutations: {
    increment(state, playload) {
      state.number += playload;
    },
  },
});

export default store;
```

2 App.vue

```
<template>
  <div>
    <input
      type="number"
      v-model="playload"
    />
    <button @click="increNumber(playload)">+</button>
    <h1>{{ this.$store.state.number }}</h1>
  </div>
</template>

<script>
export default {
  data() {
    return {
      playload: 0,
    };
  },
  methods: {
    increNumber(playload) {
      this.$store.commit("increment", playload);
    },
  },
};
</script>
```

4.actions 修改状态

Mutations:

- 当你需要同步地改变状态时，使用 mutations。
- Mutations 是用来修改 state 的唯一途径，它必须是同步的。

Actions:

- 当你需要异步地改变状态时，使用 actions。
- Actions 可以包含任意异步操作，比如 API 调用、延时操作等。
- 通常在 actions 中完成异步操作后，使用 context.commit 提交一个 mutation 来修改 state。

总结：

- 当你需要进行同步操作，直接修改 state 时，使用 mutations。
- 当你需要进行异步操作，如 API 请求，处理数据后再修改 state 时，使用 actions。

在 actions 完成异步操作后，通过提交 mutations 来修改 state。

4.actions 修改状态

假设现在仓库中有个state叫userName，它的值需要进行网络请求获取，我们是否可以在mutations中进行网络请求（如下图）？

```
import { createStore } from "vuex";
import axios from "axios";
const store = createStore({
  state: {
    userName: null,
  },
  mutations: {
    async changeUserName(state) {
      const user = await axios.get("https://api.github.com/users/yyx990803");
      state.userName = user.data.name;
    },
  },
});

export default store;
```

4.actions 修改状态

答：并不行。mutations中只能进行同步操作，如设计异步操作需要在actions中进行。

```
import { createStore } from "vuex";
import axios from "axios";
const store = createStore({
  state: {
    userName: null,
  },
  mutations: {
    changeUserName(state, data) {
      state.userName = data;
    },
  },
  actions: {
    async changeUserName(context) {
      const user = await axios.get("https://api.github.com/users/yyx990803");
      context.commit("changeUserName", user.data.name);
    },
  },
});

export default store;
```

4.actions 修改状态

接下来要做的事

1. 在仓库中增加一个state，名为userName
2. 在仓库中增加一个mutations, 名为changeUserName, 将载荷赋予userName
3. 因为这个userName的值需要进行网络请求后获取，涉及异步操作，所以需要用到actions
4. 在仓库中增加一个actions，也名为changeUserName，用于网络请求获取数据后，调用mutations修改userName

1 src/store/index.js

```
import { createStore } from "vuex";
import axios from "axios";

const store = createStore({
  state: {
    userName: null,
  },
  mutations: {
    changeUserName(state, data) {
      //将第二个参数赋予userName
      state.userName = data;
    },
  },
  actions: {
    async changeUserName(context) {
      // 异步操作（网络请求）获取用户名Evan You
      const user = await axios.get("https://api.github.com/users/yyx990803");
      // 调用mutation，将网络请求获取的用户名作为第二个参数
      context.commit("changeUserName", user.data.name);
    },
  },
});

export default store;
```

App.vue

```
<template>
  <h1>{{ this.$store.state.userName }}</h1>
</template>

<script>
export default {
  created() {
    this.$store.dispatch("changeUserName");
  },
};
</script>
```

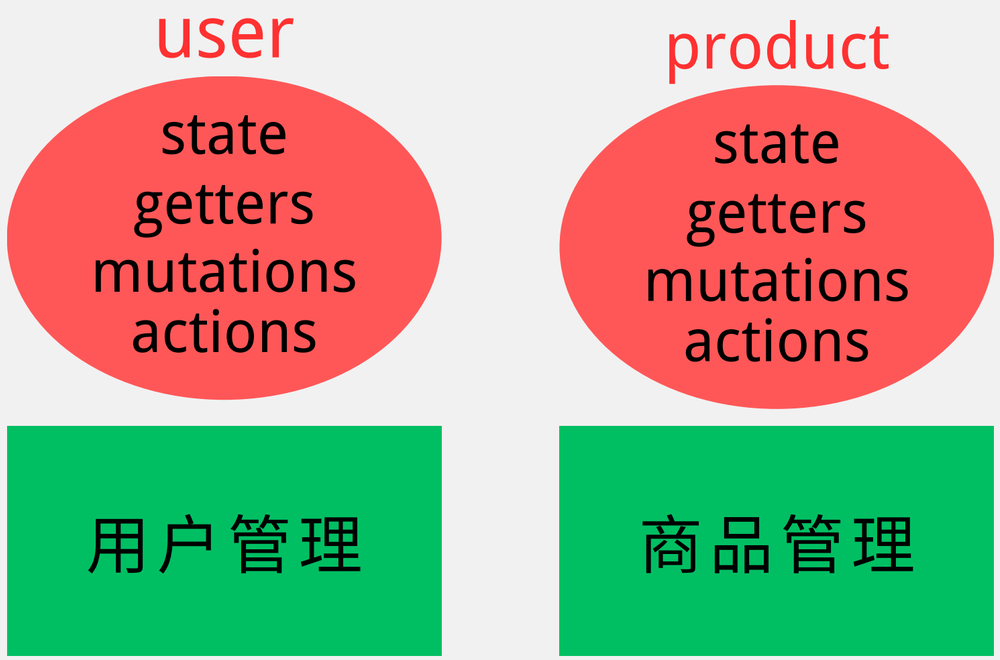
5.modules 模块化

Modules（模块）是 Vuex 中的一个重要概念，它用于将状态管理分解为更小的、独立的部分。这在大型项目中非常有用，因为它可以帮助你将整个应用程序的状态划分为更具可读性和可维护性的部分。

比如：

一个在线商城应用，分为“用户管理”“商品管理”两个模块。

那么可以制作user和product两个仓库来分别管理各自的状态



user.js

```
export default {
  namespaced: true,
  state: {
    userName: "adam",
    isLoggedIn: false,
  },
  getters: {
    // 省略
  },
  mutations: {
    // 省略
  },
  actions: {
    // 省略
  },
};
```

index.js

```
import { createStore } from "vuex";
import user from "./user";
import product from "./product";

const store = createStore({
  modules: {
    user,
    product
  },
});

export default store;
```

product.js

```
export default {
  namespaced: true,
  state: {
    productList: [],
  },
  getters: {
    // 省略
  },
  mutations: {
    // 省略
  },
  actions: {
    // 省略
  },
};
```

在组件中

获取user模块的state

```
this.$store.state.user.userName
```

获取user模块的getters

```
this.$store.getters["user/getUserInfo"]
```

提交user模块的mutations

```
this.$store.commit("user/someUserMutations");
```

分发user模块的actions

```
this.$store.dispatch("user/someUserActions");
```