[toc]

# pinia学习

## 快速入门

### 1.1.安装

```
yarn add pinia
# or with npm
npm install pinia
```

> 提示：如果您使用的是Vue2,您还需要安装组合式API包：`@vue/composition-api`

如果你使用的是vue CLI，你可以试试非官方插件

### 1.2.初始化配置

Vue3:

```
import { createPinia } from 'pinia'

app.use(createPinia())
```

#### 1.2.1应用示例

mian.js代码示例：

```
import { createApp } from 'vue'
import App from './App.vue'
import { createPinia } from 'pinia'

// 创建 pinia实例
const pinia = createPinia()

const app = createApp(App)

// 挂载到 Vue 根实例
app.use(pinia)
app.mount('#app')
```

index.ts代码示例：

```
import { difineStore } from 'pinia'

// 1.定义容器
// 参数1：容器的id ，必须唯一，将来 pinia 会把所有的容器挂载到跟容器
// 参数2：选项对象
// 返回值：一个函数，调用得到容器实例
export const useMainStore = difineStore({
    /**
    类似于组件的 data ，用来存储全局状态的
     1.必须是函数，这样是为了服务端渲染时避免交叉请求数据导致数据污染
     2.且是箭头函数：为了更好地 TS 类推导
    */
    state: () => {
      return{
          count: 100
          foo: 'bar'
      }
    },

    /*
    类似于组件的 computed，用来封装计算属性，有缓存功能
    */
    getters: {
        //注意事项：函数接受可选参数state
        count10(state){
            return state.count+10
        }

    },

    /*
    类似于组件的 methods ，用于封装业务逻辑

    */
    actions: {
        //注意不能使用箭头函数定义action
        changeState(){
            this.count++
            this.foo='hello'
            //$patch 也可使用
            // this.$patch({}) this.$patch(state=>{})
        }
    }
})
// 2.使用容器中的 state

// 3.修改 state

//4.容器中的 action 的使用
```

HelloWorld.vue示例：

```
<template>
    <p>
        {{mianStore.count}}
    </p>
    <p>
        {{mianStore.foo}}
    </p>
    <hr>
    <!----  解构赋值   -------->
    <p>
        {{foo}}
        {{count}}
    </p>

    <hr>
    <button @click='handleChangeState'>
        修改数据
    </button>
</template>
<script lang='ts' setup>
    import { useMianStore } from './index.ts'
    //解决解构赋值后数据响应式
    import {storeToRefs } from 'pinia'

    const mianStore = useMianStore()
    console.log(mianStore.count)
    console.log(mianStore.foo)

    //解构赋值(这是有问题的，非响应式的)
    const {count,foo}= mianStore
    //解决方案
    //pinia将数据作响应式处理
    const store=storeToRefs( mianStore)
    const {count,foo} = store

    const handleChangeState = () => {
        //方式一：最简单的方式
        mainStore.count++
        //方式二：如果需要修改多个数据，建议使用$patch 批量更新 不太建议
        mianStore.$patch({
            count:mianStore.count+1,
            foo:'hello'
        })
        //方式三：$patch 一个函数 建议使用
        mianStore.$patch(state => {
            state.count++
            state.foo='hello'
        })

        //方式四：逻辑较多，用actions
        mianStore.changeStore()
```

```
    }
</script>
```

### 1.2.2composing API示例：

Composing stores is about having stores that use each other, and this is supported in Pinia. There is one rule to follow:

If **two or more stores use each other**, they cannot create an infinite loop through *getters* or *actions*. They cannot **both** directly read each other state in their setup function:

```
const useX = defineStore('x', () => {
  const y = useY()

  // ❌ This is not possible because y also tries to read x.name
  y.name

  function doSomething() {
    // ☑ Read y properties in computed or actions
    const yName = y.name
    // ...
  }

  return {
    name: ref('I am X'),
  }
})

const useY = defineStore('y', () => {
  const x = useX()

  // ❌ This is not possible because x also tries to read y.name
  x.name

  function doSomething() {
    // ☑ Read x properties in computed or actions
    const xName = x.name
    // ...
  }

  return {
    name: ref('I am Y'),
  }
})
```

#### Nested Stores

Note that if one store uses another store, you can directly import and call the `useStore()` function within *actions* and *getters*. Then you can interact with the store just like you would from within a Vue component. See

Shared Getters and Shared Actions.

When it comes to *setup stores*, you can simply use one of the stores **at the top** of the store function:

```
import { useUserStore } from './user'

export const useCartStore = defineStore('cart', () => {
  const user = useUserStore()

  const summary = computed(() => {
    return `Hi ${user.name}, you have ${state.list.length} items in your cart. It
costs   ${state.price}.`
  })

  function purchase() {
    return apiPurchase(user.id, this.list)
  }

  return { summary, purchase }
})
```

#### Shared Getters

You can simply call `useOtherStore()` inside a *getter*:

```
import { defineStore } from 'pinia'
import { useUserStore } from './user'

export const useCartStore = defineStore('cart', {
  getters: {
    summary(state) {
      const user = useUserStore()

      return `Hi ${user.name}, you have ${state.list.length} items in your cart.
It costs        ${state.price}.`
    },
  },
})
```

#### Shared Actions

The same applies to *actions*:

```
import { defineStore } from 'pinia'
import { useUserStore } from './user'

export const useCartStore = defineStore('cart', {
```

```
  actions: {
    async orderCart() {
      const user = useUserStore()

      try {
        await apiOrderCart(user.token, this.items)
        // another action
        this.emptyCart()
      } catch (err) {
        displayError(err)
      }
    },
  },
})
```

如果您使用的是