Vue3——pinia部分(小满版本)

> 一颗可爱的小菠萝，乍一看是不是很像海绵宝宝的家呢？他就是pinia~ 具体做什么的，你很快就会了解到了，他是VueX的后继者，已经把VueX拍死在了沙滩上了，通常你叫他商店store也行

第一章 安装-介绍

> 前言 全局状态管理工具
>
> Pinia.js 有如下特点：
>
> - 完整的 ts 的支持；
> - 足够轻量，压缩后的体积只有 1kb 左右；
> - 去除 mutations，只有 state，getters，actions；
> - actions 支持同步和异步；
> - 代码扁平化没有模块嵌套，只有 store 的概念，store 之间可以自由使用，每一个 store 都是独立的
> - 无需手动添加 store，store 一旦创建便会自动添加；
> - 支持 Vue3 和 Vue2

让我们来看看他跟VueX的区别

pinia	**VueX**
State	State
Getters	Getters
	Mutaions
Actions 同步异步都支持	Actions

[pinia官方文档][https://pinia.vuejs.org/] or [pinia在GitHub地址][https://github.com/vuejs/pinia]

1. 起步 安装

```
yarn add pinia
//你自己选一个就行了，看你用哪个包管理器
npm install pinia//小满在这里有加一个-S，这是生产模式，还有一个是-D是开发模式
//生产模式(依赖-S)：会把包添加到 package.json 的 dependencies 下，这些包在项目打包上线后依然需要使用项目才能正常运行
//开发模式(依赖-D)：会把包添加到 package.json 的 devDependencies 下，这些包只在做项目的时候会使用到，在项目打包上线后不依赖于这些包项目依然可以正常运行
```

2. 引入注册 Vue3

> 在main.ts或者js中引入，这得看你用的是js还是ts了哈哈
>
> 因为main.ts是用来放公共API的地方，所以在这里引入~

```
import { createApp } from 'vue'
import App from './App.vue'
import {createPinia} from 'pinia'//引入
//Vue3叫createPinia
//Vue2叫PiniaVuePlugin

const store = createPinia()//调用一下
let app = createApp(App)

app.use(store)//使用pinia

app.mount('#app')
```

Vue2 使用

```
import { createPinia, PiniaVuePlugin } from 'pinia'

Vue.use(PiniaVuePlugin)
const pinia = createPinia()

new Vue({
  el: '#app',
  // other options...
  // ...
  // note the same `pinia` instance can be used across multiple Vue apps on
  // the same page
  pinia,
})
```

第二章（初始化仓库 Store）

1. 新建一个文件夹 Store

2. 新建文件 [name].ts

> 当然，通常是叫index会好一点，不过你想怎么起的话其实无所谓

3. 定义仓库 Store

```
import { defineStore } from 'pinia'
```

> 前三步一步到位了兄弟们

4. 我们需要知道存储是使用定义的 `defineStore()`，并且它需要一个唯一的名称，作为第一个参数传递

> 这儿名称抽离出去了
>
> 新建文件 store-namespace/index.ts，**嗯，其实就是命名空间**

```
//namespace下的index.ts文件，抽离出来的名字都在这里了
export const enum Names {//暴露出去并且枚举
    Test = 'TEST'
}
```

> store 引入

```
import { defineStore } from 'pinia'
import { Names } from './store-namespace'//抽离出去的名字最终还是需要引入回来store文件进行使用的

export const useTestStore = defineStore(Names.Test, {
    state()=>{
        return{
            current:1
            name:"小满、班花姐姐，洛佬"
        }
    }
})
//通过源码我们可以看到这里是：
state?:()=> S;//意思就是一个箭头函数，可选填入
```

> 这个*名称*，也称为 *id*，是必要的，Pania 使用它来将商店连接到 devtools。将返回的函数命名为 *use...* 是可组合项之间的约定，以使其使用习惯。

5. 定义值

> **State 箭头函数 返回一个对象 在对象里面定义值**
>
> 箭头函数在源码定义了，刚刚才聊到的，对吧

```
import { defineStore } from 'pinia'
import { Names } from './store-namespce'
//defineStore一共就两个参数，一个是名字，一个是我们要放的数据内容
export const useTestStore = defineStore(Names.Test, {//请注意，这是Name.Test  不是Name,Test
     state:()=>{
         return {
             current:1
         }
     }
})
```

```
import { defineStore } from 'pinia'
import { Names } from './store-namespce'

export const useTestStore = defineStore(Names.Test, {
     state:()=>{
         return {
             current:1
         }
     },
     //类似于computed 可以帮我们去修饰我们的值
     getters:{

     },
     //类似methods 可以操作异步 和 同步 提交state
     actions:{

     }
})
```

> 要使用的话，直接在你想要使用的文件里面引入store商店就行了
>
> ```
> //例子
> import {useTestStore} from "./store"
> //然后进行调用一下
> const Test = useTestStore()
> ```

案例

> 以下为纯代码，无注释的可直接运行代码

```
//store下的index文件
import {defineStore} from "pinia";
import {Names} from "./store-name";

export const useTestStore = defineStore(Names.TEST,{
    state:()=>{
        return{
            name:"洛洛",
            Nickname:"洛佬，肥洛，狗洛，洛爆了，有事请问洛洛"
        }
    }
})
```

```
//store下的store-name文件
export const enum Names{
    TEST = 'TEST'
}
```

```
//App.vue主文件
<template>
<div>大家好！{{Test.name}}是一个外号为：{{Test.Nickname}}的人</div>
</template>

<script setup lang="ts">
import {useTestStore} from "./store"
const Test = useTestStore()
</script>


<style scoped>

</style>
```

第三章 — state

> 拥有5种方式来进行修改值

1.State 是允许直接修改值的 例如 current++

```
<template>
     <div>
         <button @click="Add">+</button>
          <div>
             {{Test.current}}
          </div>
     </div>
</template>

<script setup lang='ts'>
import {useTestStore} from './store'
const Test = useTestStore()
const Add = () => {
```

```
    Test.current++ //就直接修改就完事了，在vuex中是不被允许的
}

</script>

<style>

</style>
```

2. 批量修改 State 的值

> 在他的实例上有 $patch 方法可以批量修改多个值

```
<template>
     <div>
         <button @click="Add">+</button>
          <div>
             {{Test.current}}
          </div>
          <div>
            {{Test.age}}
          </div>
     </div>
</template>

<script setup lang='ts'>
import {useTestStore} from './store'
const Test = useTestStore()
const Add = () => {
    Test.$patch({
       current:200,
       age:300
    })
}

</script>

<style>

</style>
```

3. 批量修改函数形式

> 推荐使用函数形式 可以自定义修改逻辑
>
> 可以在里面进行if判断，for循环啥的都可以，明显灵活很多

```
<template>
     <div>
         <button @click="Add">+</button>
          <div>
             {{Test.current}}
          </div>
          <div>
```

```
            {{Test.age}}
          </div>
      </div>
</template>

<script setup lang='ts'>
import {useTestStore} from './store'
const Test = useTestStore()
const Add = () => {
    Test.$patch((state)=>{//这里的state就是我们在store仓库中的state(然后里面return了数值)
       state.current++;
       state.age = 40
    })
}

</script>

<style>

</style>
```

> 与第二章案例配套代码

```
<template>
<div>大家好! {{Test.name}}来自{{Test.form}}</div>
  <button @click="change">修改</button>
</template>

<script setup lang="ts">
import {useTestStore} from "./store"
const Test = useTestStore()

const change = ()=>{
  Test.$patch((state)=>{//state就是我们store仓库内的数据
    state.name = "小余"
    state.form = "QQ群855139333"
  })
}
</script>


<style scoped>

</style>
```

4. 通过原始对象修改整个实例

> `$state` 您可以通过将 store 的属性设置为新对象来替换 store 的整个状态
>
> 缺点就是必须修改整个对象的所有属性

```
<template>
```

```
    <div>
        <button @click="Add">+</button>
         <div>
            {{Test.current}}
         </div>
         <div>
           {{Test.age}}
         </div>
    </div>
</template>

<script setup lang='ts'>
import {useTestStore} from './store'
const Test = useTestStore()
const Add = () => {
    Test.$state = {
       current:10,//全部都得修改，不然会报错，你也可以修改成一样的数值或者内容，但是不能不写
       age:30
    }
}

</script>

<style>

</style>
```

> 配合第二章的案例

```
//App.vue文件
<template>
<div>大家好！{{Test.name}}来自{{Test.form}}</div>
  <button @click="change">修改</button>
</template>

<script setup lang="ts">
import {useTestStore} from "./store"
const Test = useTestStore()

const change = ()=>{
  Test.$state = {
    name:"小余",
    form:"泉州"
  }
}
</script>


<style scoped>

</style>
```

5. 通过 actions 修改

> 定义 Actions
>
> 在 actions 中直接使用 this 就可以指到 state 里面的值

```
import { defineStore } from 'pinia'
import { Names } from './store-naspace'
export const useTestStore = defineStore(Names.TEST, {
  state:()=>{
      return {
         current:1,
         age:30
      }
  },

  actions:{
      setCurrent () {////不能使用箭头函数，因为this的指向会出现问题
          this.current++
          //this是由定义好的store实例调用，箭头函数只会保存当前作用域的this，所以需要传统方式定义函数，根据调用者来改变this指向
      }
  }
})
```

使用方法直接在实例调用

```
<template>
     <div>
         <button @click="Add">+</button>
          <div>
             {{Test.current}}
          </div>
          <div>
            {{Test.age}}
          </div>
     </div>
</template>

<script setup lang='ts'>
import {useTestStore} from './store'
const Test = useTestStore()
const Add = () => {
     Test.setCurrent()
}

</script>

<style>

</style>
```

第四章 — 结构store - 源码解析

> 在pinia是不允许直接解构，是会失去响应式的

```
const Test = useTestStore()
 //pinia解构不具有响应式
const { current, name } = Test//这是解构

console.log(current, name);
```

差异对比

> 修改 Test current 解构完之后的数据不会变
>
> 而源数据是会变的

```
<template>
  <div>origin value"这是不解构的"{{Test.current}}</div>
  <div>
    pinia:{{ current }}--{{ name }}//这是解构的，从Test中解构出来，直接使用，不在前面加
Test
    change :
    <button @click="change">change</button>
  </div>
</template>

<script setup lang='ts'>
import { useTestStore } from './store'

const Test = useTestStore()

const change = () => {
   Test.current++
}

const { current, name } = Test

console.log(current, name);


</script>

<style>
</style>
```

解决方案可以使用 storeToRefs

```
import { storeToRefs } from 'pinia'

const Test = useTestStore()

const { current, name } = storeToRefs(Test)//在解构之前先包一层
```

> 其原理跟 toRefs 一样的给里面的数据包裹一层 toref

> 源码 通过 toRaw 使 store 变回原始数据防止重复代理(这个在前面Vue3基础部分的源码部分有讲过为什么会发生重复代理的，可以翻一翻
>
> `Vue3基础笔记(小满版本)`)
>
> 循环 store 通过 isRef isReactive 判断 如果是响应式对象直接拷贝一份给 refs 对象 将其原始对象包裹 toRef 使其变为响应式对象

源码解析 - storeToRefs

```
//源码
function storeToRefs(store) {
    // See https://github.com/vuejs/pinia/issues/852
    // It's easier to just use toRefs() even if it includes more stuff
    if (isVue2) {
        // @ts-expect-error: toRefs include methods and others
        console.log("store变化之前",store)
        return toRefs(store);
        //- 刚开始就先判断一个是不是对象，是的话将其里面的值放到数组中初始化一下(ps：真的很严谨)
        //然后每个属性去做一下toRef，然后把这个内容做一个返回。也判断了一下是不是Proxy对象。如果是Proxy对象就走下面的那个判断了
        console.log("store变化之后",store)
    }
    else {
        store = toRaw(store);//将store转化为原始对象，toRaw把reactive套上的Proxy外壳给脱掉了
        const refs = {};
        for (const key in store) {//for循环做了一个toRef的一个操作
            const value = store[key];//拷贝(复制)了一层store
            if (isRef(value) || isReactive(value)) {
                // @ts-expect-error: the key is state or getter
                refs[key] =
                    // ---
                    toRef(store, key);//然后进行包裹
            }
        }
        return refs;
    }
}
```

> 相较于小满录制视频的3月份来说，制作笔记的时间是同年11月份，此时pinia已经进行了优化，源码部分对Vue2进行了一个兼容，我根据自己的理解重新写上了注释

第五章 — Actions,getters

Actions（支持同步异步）

1.同步

> 同步 直接调用即可

```
//商店store下的index.ts文件
import { defineStore } from 'pinia'
```

```
import { Names } from './store-naspace'
export const useTestStore = defineStore(Names.TEST, {
    state: () => ({
        counter: 0,
    }),
    actions: {
        increment() {
            this.counter++
        },
        randomizeCounter() {
            this.counter = Math.round(100 * Math.random())//同步写法，直接使用即可
        },
    },
})
```

```
<template>
     <div>
         <button @click="Add">+</button>
          <div>
             {{Test.counter}}
          </div>
     </div>
</template>

<script setup lang='ts'>
import {useTestStore} from './store'
const Test = useTestStore()
const Add = () => {
     Test.randomizeCounter()//直接就是进行一个调用，这里的Test是对从store中解构出来的
userTestStore的一个调用
}

</script>

<style>

</style>
```

2.异步

> 异步 可以结合 async await 修饰
>
> Promise 的构造函数接收一个参数，是函数，并且传入两个参数：`resolve`，`reject`，分别表示异步操作执行成功后的回调函数和异步操作执行失败后的回调函数。其实这里用 “成功” 和 “失败” 来描述并不准确，按照标准来讲，resolve 是将 Promise 的状态置为 fullfiled，reject 是将 Promise 的状态置为 rejected。不过在我们开始阶段可以先这么理解

```
import { defineStore } from 'pinia'
import { Names } from './store-naspace'

type Result = {
    name: string
    isChu: boolean
}
```

```
const Login = (): Promise<Result> => {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve({
                name: '小满',
                isChu: true
            })
        }, 3000)
    })
}

export const useTestStore = defineStore(Names.TEST, {
    state: () => ({
        user: <Result>{},//定义泛型
        name: "123"
    }),
    actions: {
        async getLoginInfo() {//异步操作
            const result = await Login()//调用了上面的Login异步操作
            this.user = result;
            this.user = setName("大飞机")//
        },
        setName(name:string){
            this.name = name;
        }
    },
})
```

template

```
<template>
     <div>
         <button @click="Add">test</button>
          <div>
             {{Test.user}}
          </div>
     </div>
</template>

<script setup lang='ts'>
import {useTestStore} from './store'
const Test = useTestStore()
const Add = () => {
     Test.getLoginInfo()
}

</script>

<style>

</style>
```

3.多个 action 互相调用 getLoginInfo setName

```
    state: () => ({
        user: <Result>{},
        name: "default"
    }),
    actions: {
        async getLoginInfo() {
            const result = await Login()
            this.user = result;
            this.setName(result.name)
        },
        setName (name:string) {
            this.name = name;
        }
    },
```

getters

> 在使用 this 访问时，需要定义返回类型（在 TypeScript 中），这是因为 TypeScript 中的一个已知限制 这不会影响使用箭头函数定义的 getter，也不会影响不使用 this 的 getter

1.箭头函数

> 1. 使用箭头函数不能使用 this this 指向已经改变指向 undefined 修改值请用 state
>
>   主要作用类似于 computed 数据修饰并且有缓存
>
> 2. Getter 完全等同于 Store 状态的计算值，在 defineStore () 中的 getters 属性中定义
>   当它接收 “状态” 作为第一个参数时，鼓励箭头函数的使用

```
    getters:{
       newPrice:(state)=>  `$${state.user.price}`
    },
```

2.普通函数

> 2. 普通函数形式可以使用 this
> 3. getter 只会依赖状态，可能会使用到其他 getter，因此可以在定义常规函数时通过 this 访问到整个 store 的实例

```
    getters:{
       newCurrent ():number {
           return ++this.current
       }
    },
```

3.getters 互相调用

> 与计算属性一样，可以组合多个 getter。通过 this 访问任何其他 getter

```
    getters:{
       newCurrent ():number | string {
           return ++this.current + this.newName
       },
       newName ():string {
           return `$-${this.name}`
       }
    },
```

第六章 — （API）

1.$reset

> 重置store到他的初始状态

```
state: () => ({
     user: <Result>{},
     name: "default",
     current:1
}),
```

> Vue 例如我把值改变到了 10

```
const change = () => {
     Test.current++
}
```

> 视频写法

```
const reset = ()=>{
    Test.$reset()
}//直接进行初始化，调用reset即可
```

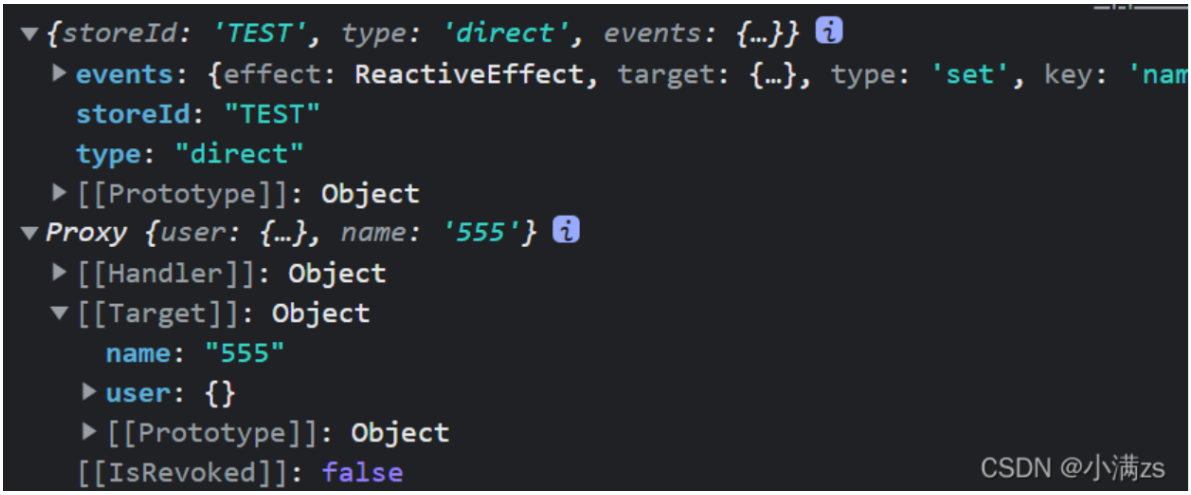
> 调用 $reset ();
>
> 将会把 state 所有值 重置回 原始状态

2. 订阅 state 的改变

> 类似于 Vuex 的 abscribe 只要有 state 的变化就会走这个函数

```
Test.$subscribe((args,state)=>{//subscribe是订阅的意思
   console.log(args,state);//返回两个值如下
})
```

返回值



第二个参数

> 如果你的组件卸载之后还想继续调用请设置第二个参数
>
> 第二个参数是一个对象 {detached: true}，在组件销毁后依然监听状态的改变

```
Test.$subscribe((args,state)=>{
   console.log(args,state);

},{
  detached:true
})
```

3. 订阅 Actions 的调用

> 只要有 actions 被调用就会走这个函数
>
> 第一个参数是工厂函数(就是返回的是一个对象的)

```
Test.$onAction((args)=>{
   console.log(args);
   args.after(()=>{
       console.log('after');
   })
})
//args能够捕获到外面写入的数据内容
//App.vue文件(就是你使用pinia的文件，不一定是App.vue)
const change = ()=>{
    Test.setUser('123')//被args捕获到
}
```

第七章 — pinia插件

> pinia 和 vuex 都有一个通病 页面刷新状态会丢失
>
> 我们可以写一个 pinia 插件缓存他的值

```
const __piniaKey = '__PINIAKEY__'
//定义兜底变量


type Options = {
   key?:string
}
//定义入参类型



//将数据存在本地
const setStorage = (key: string, value: any): void => {//设置的存储函数

localStorage.setItem(key, JSON.stringify(value))

}


//存缓存中读取
const getStorage = (key: string) => {

return (localStorage.getItem(key) ? JSON.parse(localStorage.getItem(key) as
string) : {})//判断有没有key，没有就返回空对象

}


//利用函数柯丽华接受用户入参
const piniaPlugin = (options: Options) => {

//将函数返回给pinia  让pinia  调用 注入 context
return (context: PiniaPluginContext) => {

const { store } = context;

const data = getStorage(`${options?.key ?? __piniaKey}-${store.$id}`)//将数据取出来

store.$subscribe(() => {

setStorage(`${options?.key ?? __piniaKey}-${store.$id}`, toRaw(store.$state));//

})

//返回值覆盖pinia 原始值
return {

...data

}
```

```
}

}

//初始化pinia
const pinia = createPinia()

//注册pinia 插件
pinia.use(piniaPlugin({
key: "pinia"
}))
```