## 准备工作

### 安装

*yarn* *add* pinia

*# or with npm*

*npm* *install* pinia

### 注册

#### Vue3 用法：

import { createPinia } from 'pinia'

app.*use*(*createPinia*())

#### vue2 用法

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,

})

## 开始使用

### 创建store

#### 创建

import { defineStore } from 'pinia'

*// useStore could be anything like useUser, useCart*

*// the first argument is a unique id of the store across your application*

export const useStore = *defineStore*('main', {

*// other options...*

})

#### 使用

import { useStore } from '@/stores/counter'

export default {

*setup*() {

const store = *useStore*()

return {

*// you can return the whole store instance to use it in the template*

store,

}

},

}

注意：store 是一个用包裹的对象reactive，意味着不需要.value，但是也不能对其进行结构

*storeToRefs API可以帮助我们对store进行结构操作*

import { storeToRefs } from 'pinia'

export default *defineComponent*({

*setup*() {

const store = *useStore*()

*// `name` and `doubleCount` are reactive refs*

*// This will also create refs for properties added by plugins*

*// but skip any action or non reactive (non ref/reactive) property*

const { name, doubleCount } = *storeToRefs*(store)

*// the increment action can be just extracted*

const { increment } = store

return {

name,

doubleCount

increment,

}

},

})

### State

#### 使用

import { defineStore } from 'pinia'

const useStore = *defineStore*('storeId', {

*// arrow function recommended for full type inference*

*state*: () => {

return {

*// all these properties will have their type inferred automatically*

counter: 0,

name: 'Eduardo',

isAdmin: true,

}

},

})

提示： 如果使用vue2，创建数据state 需要遵循与vue实例相同的规则

Data，并且添加新属性

#### 访问

const store = *useStore*()

store.counter++

#### 重置状态

可以通过调用store 上的方法将状态重置为初始值： $reset()

const store = *useStore*()

store.*$reset*()

#### 使用option api

有setup

import { useCounterStore } from '../stores/counterStore'

export default {

*setup*() {

const counterStore = *useCounterStore*()

return { counterStore }

},

computed: {

*tripleCounter*() {

return this.counterStore.counter \* 3

},

},

}

没有setup

如果您没有使用 Composition API，而您正在使用computed, methods, ...，则可以使用mapState()帮助器将状态属性映射为只读计算属性：

import { mapState } from 'pinia'

import { useCounterStore } from '../stores/counterStore'

export default {

computed: {

*// gives access to this.counter inside the component*

*// same as reading from store.counter*

...*mapState*(useCounterStore, ['counter'])

*// same as above but registers it as this.myOwnName*

...*mapState*(useCounterStore, {

myOwnName: 'counter',

*// you can also write a function that gets access to the store*

*double*: store => store.counter \* 2,

*// it can have access to `this` but it won't be typed correctly...*

*magicValue*(store) {

return store.someGetter + this.counter + this.double

},

}),

},

}

#### 更改状态

1. 可以直接使用store.counter++
2. 使用$patch方法（允许对多个属性更改）

store.*$patch*({

counter: store.counter + 1,

name: 'Abalam',

})

$patch 可以接受一个函数对属性进行操作

cartStore.*$patch*((state) => {

state.items.*push*({ name: 'shoes', quantity: 1 })

state.hasChanged = true

})

#### 更换整个state

$state 可以将store 设置为一个新的对象来替换store的整个状态

store.$state = { counter: 666, name: 'Paimon' }

注意：以通过更改 pinia实例的state来替换应用程序的整个状态。这在SSR激活中使用

pinia.state.value = {}

#### 监听/订阅状态

可以通过$subscribe() 方法查看状态以及变化；

使用$subscribe() 的优势在于，订阅只会在patches之后出发一次（例如当函数版本变动时）

cartStore.*$subscribe*((mutation, state) => {

*// import { MutationType } from 'pinia'*

mutation.type *// 'direct' | 'patch object' | 'patch function'*

*// same as cartStore.$id*

mutation.storeId *// 'cart'*

*// only available with mutation.type === 'patch object'*

mutation.payload *// patch object passed to cartStore.$patch()*

*// persist the whole state to the local storage whenever it changes*

localStorage.*setItem*('cart', JSON.*stringify*(state))

})

注意： 默认情况下，状态订阅被绑定到添加他们的组件上（如果store在组件的setup（）中）。当组件被卸载时，它们将自动删除，如果想要在组件卸载后保留需要传递 { detached: true } 作为第二个参数来从当前组件分离订阅状态

export default {

*setup*() {

const someStore = *useSomeStore*()

*// this subscription will be kept after the component is unmounted*

someStore.*$subscribe*(callback, { detached: true })

*// ...*

},

}

提示：

pinia您可以查看实例上的整个状态：

*watch*(

pinia.state,

(state) => {

*// persist the whole state to the local storage whenever it changes*

localStorage.*setItem*('piniaState', JSON.*stringify*(state))

},

{ deep: true }

)

### Getters

Getter 完全等同于store的计算属性。可以用defineStore（）中的getters 属性来定义。它们接受state作为第一个参数，鼓励使用箭头函数：

#### 创建

export const useStore = *defineStore*('main', {

*state*: () => ({

counter: 0,

}),

getters: {

*doubleCount*: (state) => state.counter \* 2,

},

})

#### Getter嵌套使用

大多数情况下getters只依赖于state，但是，它们也可能使用其他的getters。因此当定义一个常规函数时，我们可以通过this 访问整个store实例，但是需要定义返回类型的类型（在typescipt中）。

export const useStore = *defineStore*('main', {

*state*: () => ({

counter: 0,

}),

getters: {

*// automatically infers the return type as a number*

*doubleCount*(state) {

return state.counter \* 2

},

*// the return type \*\*must\*\* be explicitly set*

*doublePlusOne*(): number {

*// autocompletion and typings for the whole store ✨*

return this.doubleCount + 1

},

},

})

#### 使用：

<template>

<p>Double count is {{ store.doubleCount }}</p>

</template>

<script>

export default {

*setup*() {

const store = *useStore*()

return { store }

},

}

</script>

#### 传参

Getter只是在幕后计算的属性，因此不可能将任何参数传递给它们。但是，您可以从getter返回一个函数以接受任何参数：

export const useStore = *defineStore*('main', {

getters: {

*getUserById*: (state) => {

return (userId) => state.users.*find*((user) => user.id === userId)

},

},

})

应用

<script>

export default {

*setup*() {

const store = *useStore*()

return { getUserById: store.getUserById }

},

}

</script>

<template>

<p>User 2: {{ getUserById(2) }}</p>

</template>

注意：

行此操作时，**getter 不再缓存**，它们只是您调用的函数。但是，您可以在 getter 本身内部缓存一些结果，这并不常见，但应该证明性能更高：

export const useStore = *defineStore*('main', {

getters: {

*getActiveUserById*(state) {

const activeUsers = state.users.*filter*((user) => user.active)

return (userId) => activeUsers.*find*((user) => user.id === userId)

},

},

})

#### Store 嵌套getters 使用

要使用其他存储 getter，您可以直接在getter内部使用它：

import { useOtherStore } from './other-store'

export const useStore = *defineStore*('main', {

*state*: () => ({

*// ...*

}),

getters: {

*otherGetter*(state) {

const otherStore = *useOtherStore*()

return state.localData + otherStore.data

},

},

})

#### 用法

##### Setup （） 中（vue3）

export default {

*setup*() {

const store = *useStore*()

store.counter = 3

store.doubleCount *// 6*

},

}

##### Options API 中

*// Example File Path:*

*// ./src/stores/counterStore.js*

import { defineStore } from 'pinia',

const useCounterStore = *defineStore*('counterStore', {

*state*: () => ({

counter: 0

}),

getters: {

*doubleCounter*() {

return this.counter \* 2

}

}

})

Vue2使用setup（）

import { useCounterStore } from '../stores/counterStore'

export default {

*setup*() {

const counterStore = *useCounterStore*()

return { counterStore }

},

computed: {

*quadrupleCounter*() {

return counterStore.doubleCounter \* 2

},

},

}

不使用

import { mapState } from 'pinia'

import { useCounterStore } from '../stores/counterStore'

export default {

computed: {

*// gives access to this.doubleCounter inside the component*

*// same as reading from store.doubleCounter*

...*mapState*(useCounterStore, ['doubleCount'])

*// same as above but registers it as this.myOwnName*

...*mapState*(useCounterStore, {

myOwnName: 'doubleCounter',

*// you can also write a function that gets access to the store*

*double*: store => store.doubleCount,

}),

},

}

### Actions

Actions 相当于组件中的methods。可以使用defineStore（）中的actions 属性来定义它们，

支持一步逻辑操作

#### 创建

export const useStore = *defineStore*('main', {

*state*: () => ({

counter: 0,

}),

actions: {

*increment*() {

this.counter++

},

*randomizeCounter*() {

this.counter = Math.*round*(100 \* Math.*random*())

},

},

})

#### 使用

export default *defineComponent*({

*setup*() {

const main = *useMainStore*()

*// call the action as a method of the store*

main.*randomizeCounter*()

return {}

},

})

#### 访问其他store中的actions

import { useAuthStore } from './auth-store'

export const useSettingsStore = *defineStore*('settings', {

*state*: () => ({

preferences: null,

*// ...*

}),

actions: {

async *fetchUserPreferences*() {

const auth = *useAuthStore*()

if (auth.isAuthenticated) {

this.preferences = await *fetchPreferences*()

} else {

throw new Error('User must be authenticated')

}

},

},

})

#### 在setup中

export default {

*setup*() {

const store = *useStore*()

store.*randomizeCounter*()

},

}

可以直接在setup中，直接使用，vue2中使用setUp（）

import { useCounterStore } from '../stores/counterStore'

export default {

*setup*() {

const counterStore = *useCounterStore*()

return { counterStore }

},

methods: {

*incrementAndPrint*() {

this.counterStore.*increment*()

console.*log*('New Count:', this.counterStore.count)

},

},

}

#### 在option api 中

import { mapActions } from 'pinia'

import { useCounterStore } from '../stores/counterStore'

export default {

methods: {

*// gives access to this.increment() inside the component*

*// same as calling from store.increment()*

...*mapActions*(useCounterStore, ['increment'])

*// same as above but registers it as this.myOwnName()*

...*mapActions*(useCounterStore, { myOwnName: 'doubleCounter' }),

},

}

#### 订阅actions

可以使用actions.$onAction() 来观察actions以及结果。传递给它的回调函数在action本身之前执行。在处理promise之后，允许您在action resolves 之后执行函数。

onError允许您在操作抛出或拒绝时执行函数。这些对于在运行时跟踪错误很有用

const unsubscribe = someStore.*$onAction*(

({

name, *// name of the action*

store, *// store instance, same as `someStore`*

args, *// array of parameters passed to the action*

after, *// hook after the action returns or resolves*

onError, *// hook if the action throws or rejects*

}) => {

*// a shared variable for this specific action call*

const startTime = Date.*now*()

*// this will trigger before an action on `store` is executed*

console.*log*(`Start "${name}" with params [${args.*join*(', ')}].`)

*// this will trigger if the action succeeds and after it has fully run.*

*// it waits for any returned promised*

*after*((result) => {

console.*log*(

`Finished "${name}" after ${

Date.*now*() - startTime

}ms.\nResult: ${result}.`

)

})

*// this will trigger if the action throws or returns a promise that rejects*

*onError*((error) => {

console.*warn*(

`Failed "${name}" after ${Date.*now*() - startTime}ms.\nError: ${error}.`

)

})

}

)

*// manually remove the listener*

*unsubscribe*()

默认情况下，操作订阅绑定到添加它们的组件（如果商店位于组件的 内部setup()）。意思是，当组件被卸载时，它们将被自动删除。如果要在卸载组件后保留它们，请true作为第二个参数传递以将操作订阅与当前组件分离

export default {

*setup*() {

const someStore = *useSomeStore*()

*// this subscription will be kept after the component is unmounted*

someStore.*$onAction*(callback, true)

*// ...*

},

}

### Plugins

使用pinia.use() 将插件添加到pinia实例中。

一个简单例子，通过返回一个对象为所有的商店添加一个静态属性：

import { createPinia } from 'pinia'

*// add a property named `secret` to every store that is created after this plugin is installed*

*// this could be in a different file*

function *SecretPiniaPlugin*() {

return { secret: 'the cake is a lie' }

}

const pinia = *createPinia*()

*// give the plugin to pinia*

pinia.*use*(SecretPiniaPlugin)

*// in another file*

const store = *useStore*()

store.secret *// 'the cake is a lie*

#### 介绍

Pinia 的插件是一个函数，可以通过返回要添加到store中的属性。它有一个可选参数context：

export function *myPiniaPlugin*(context) {

context.pinia *// the pinia created with `createPinia()`*

context.app *// the current app created with `createApp()` (Vue 3 only)*

context.store *// the store the plugin is augmenting*

context.options *// the options object defining the store passed to `defineStore()`*

*// ...*

}

然后将此函数传递给pinia的pinia.use()

pinia.*use*(myPiniaPlugin)

#### 扩充store

可以通过在插件中返回一个属性对象为每一个store添加属性

pinia.*use*(() => ({ hello: 'world' }))

也可以直接在store中设置属性，如果可以的话，请返回版本，**以便它们可以被 devtools 自动跟踪**：

pinia.*use*(({ store }) => {

store.hello = 'world'

})

插件返回的任何属性都将由devtools自动追踪，因此为了hello在devtools中可见，请确保仅在开发模式中添加store.\_customProperties 属性

*// from the example above*

pinia.*use*(({ store }) => {

store.hello = 'world'

*// make sure your bundler handle this. webpack and vite should do it by default*

if (process.env.NODE\_ENV === 'development') {

*// add any keys you set on the store*

store.\_customProperties.*add*('hello')

}

})

需要注意的是store都会使用reactive包装，并且会自动解包它包含的任何Ref（ref()， computed(),……）等：

const sharedRef = *ref*('shared')

pinia.*use*(({ store }) => {

*// each store has its individual `hello` property*

store.hello = *ref*('secret')

*// it gets automatically unwrapped*

store.hello *// 'secret'*

*// all stores are sharing the value `shared` property*

store.shared = sharedRef

store.shared *// 'shared'*

})

#### 添加新状态

如果想在激活过程中添加新的状态属性或者属到 store，必须在两个地方添加：

* 在store中，您可以通过store.myState访问它
* 在store.$state中，它可以在devtools中使用，并且在SSR期间被序列化。

const globalSecret = *ref*('secret')

pinia.*use*(({ store }) => {

*// `secret` is shared among all stores*

store.$state.secret = globalSecret

store.secret = globalSecret

*// it gets automatically unwrapped*

store.secret *// 'secret'*

const hasError = *ref*(false)

store.$state.hasError = hasError

*// this one must always be set*

store.hasError = *toRef*(store.$state, 'hasError')

*// in this case it's better not to return `hasError` since it*

*// will be displayed in the `state` section in the devtools*

*// anyway and if we return it, devtools will display it twice.*

})

请注意，插件中发生的状态更改或添加（包括调用store.$patch()）发生在商店处于活动状态之前，因此**不会触发任何订阅**。

**警告**

如果您使用的是**Vue 2**，Pinia 会受到与Vue[相同的反应性警告](https://vuejs.org/v2/guide/reactivity.html#Change-Detection-Caveats)。创建新的状态属性时需要使用setfrom ，例如and ：@vue/composition-apisecrethasError

import { set } from '@vue/composition-api'

pinia.*use*(({ store }) => {

if (!store.$state.*hasOwnProperty*('hello')) {

const secretRef = *ref*('secret')

*// If the data is meant to be used during SSR, you should*

*// set it on the `$state` property so it is serialized and*

*// picked up during hydration*

*set*(store.$state, 'secret', secretRef)

*// set it directly on the store too so you can access it*

*// both ways: `store.$state.secret` / `store.secret`*

*set*(store, 'secret', secretRef)

store.secret *// 'secret'*

}

})

#### 添加新的外部属性

科普：

toRaw，将响应式对象（由 reactive定义的响应式）转换为普通对象。  
markRaw，标记一个对象，使其不能成为一个响应式对象。

当添加外部属性，来自其他的库的类实例或者简单的非响应式对象时，应该在将对象传递给pinia之前使用markRaw( ) 包装该对象

import { markRaw } from 'vue'

*// adapt this based on where your router is*

import { router } from './router'

pinia.*use*(({ store }) => {

store.router = *markRaw*(router)

})

#### 在插件内部调用$subscribe

在插件中使用store.$subscribe 和 store.$onAction

pinia.*use*(({ store }) => {

store.*$subscribe*(() => {

*// react to store changes*

})

store.*$onAction*(() => {

*// react to store actions*

})

})

#### 添加新的选项

可以定义stores 时创建新的选项，以便随后的插件中使用它们。例如，你可以创建一个debounce选项，允许你对任何操作进行debounce：

*defineStore*('search', {

actions: {

*searchContacts*() {

*// ...*

},

},

*// this will be read by a plugin later on*

debounce: {

*// debounce the action searchContacts by 300ms*

searchContacts: 300,

},

})

插件可以读取该选项来包装actions并替换原来的actions：

*// use any debounce library*

import debounce from 'lodash/debunce'

pinia.*use*(({ options, store }) => {

if (options.debounce) {

*// we are overriding the actions with new ones*

return Object.*keys*(options.debounce).*reduce*((debouncedActions, action) => {

debouncedActions[action] = *debounce*(

store[action],

options.debounce[action]

)

return debouncedActions

}, {})

}

})

注意：

使用setup语法时， 自定义选项作为第三个参数传入：

*defineStore*(

'search',

() => {

*// ...*

},

{

*// this will be read by a plugin later on*

debounce: {

*// debounce the action searchContacts by 300ms*

searchContacts: 300,

},

}

)

#### 编写插件

##### Pinia插件可以按如下方式编写：

import { PiniaPluginContext } from 'pinia'

export function *myPiniaPlugin*(context: PiniaPluginContext) {

*// ...*

}

##### 编写新的store属性：

当向stores添加新的属性时，应该扩展PiniaCustomProperties接口

import 'pinia'

declare module 'pinia' {

export interface PiniaCustomProperties {

*// by using a setter we can allow both strings and refs*

set *hello*(value: string | Ref<string>)

get *hello*(): string

*// you can define simpler values too*

simpleNumber: number

}

}

然后读取：

pinia.*use*(({ store }) => {

store.hello = 'Hola'

store.hello = *ref*('Hola')

store.simpleNumber = Math.*random*()

*// @ts-expect-error: we haven't typed this correctly*

store.simpleNumber = *ref*(Math.*random*())

})

PiniaCustomProperties是一个泛型类型，允许引用store的属性，

如果要将初始选项复制为options （这仅适用于option stores）

pinia.*use*(({ options }) => ({ $options: options }))

可以通过使用PiniaCustomProperties 的四个泛型类型来正确的输入

import 'pinia'

declare module 'pinia' {

export interface PiniaCustomProperties<Id, S, G, A> {

$options: {

id: Id

state?: () => S

getters?: G

actions?: A

}

}

}

注意：

在泛型扩展时，他们的命名必须与源码中完全相同

* S: State
* G: Getters
* A: Actions
* SS: Setup Store / Store

##### 编写新的状态

当添加新的属性时（同时添加到store和store.$state），需要将类型添加到PiniaCustomStateProperties。与PiniaCustomProperties 不同的是它只接收state 泛型

import 'pinia'

declare module 'pinia' {

export interface PiniaCustomStateProperties<S> {

hello: string

}

}

##### 编写新的创建选项

当为 defineStore() 创建新的选项时，需要扩展 DefineStoreOptionsBase 。 与 PiniaCustomProperties 不同的是，它只公开两种泛型：state和store类型，允许限制定义的类型。例如， 可以使用actions 的名称

import 'pinia'

declare module 'pinia' {

export interface DefineStoreOptionsBase<S, Store> {

*// allow defining a number of ms for any of the actions*

debounce?: Partial<Record<keyof StoreActions<Store>, number>>

}

}

**提示**

还有一种类型可以从 Store 类型StoreGetters中提取getter 。您也可以**仅**通过分别扩展类型和来扩展设置商店​​或选项商店 的选项。DefineStoreOptions DefineSetupStoreOptions

## 问题以及解决