

Pinia

Pinia源码part2-defineStore

part1 讲解了createPinia的原理，part2就来讲一下defineStore原理。理解原理之前，先来了解一下defineStore如何使用的。

```
1 import { defineStore } from 'pinia'
2
3 // useStore could be anything like useUser, useCart
4 // the first argument is a unique id of the store across your application
5 export const useStore = defineStore('main', {
6   // other options...
7 })
```

```
1 import { useStore } from '@/stores/counter'
2
3 export default {
4   setup() {
5     const store = useStore()
6
7     return {
8       // you can return the whole store instance to use it in the template
9       store,
10    }
11  },
12 }
```

上面的官方例子就是使用defineStore的方法

 官方推荐，如果多个store，建议每一个store放在不同的文件，方便bundle code splitting而且对typescript 类型推断支持更好。

上面示例代码的 defineStore, 接受2个参数，第一个参数就是store id，第二个参数store 配置对象，里面可以放置state, getter, action 等等store需要的使用的数据。

defineStore, 返回了一个函数，然后export 出来。如果一个组件需要使用store只需要import进来就可以了。

那么，现在就开始读源码吧。defineStore 代码比较长，将会分成一段一段的理解。

```
1 // defineStore定义在 src/store.ts
2 // 先来理解一下defineStore函数签名
3 function defineStore(
```

```

4  // TODO: add proper types from above
5  idOrOptions: any,
6  setup?: any,
7  setupOptions?: any
8 ): StoreDefinition

```

正如函数签名，创建一个store 除了一开始的示例代码，还有其他创建的方法。

1. idOrOptions - 第一个参数一定需要提供的，可以为一个string或者对象类型。如果string类型，那么就是store id， 如果一个对象类型，那么对象里面一定会有一个id 属性，id属性的值就是store id。
2. setup - 第二个参数可选，之后的章节会讲解
3. setupOptions - 第三个参数可选，之后的章节会讲解

通过函数签名，可以知道，目前知道创建store的方法就有2个(更多的创建方法会在其他章节讲解)，

```

1  defineStore('storeId', {
2    state: ()=>({
3      num: 0
4    }),
5    getters: {
6      trible: (state)=>state.num*3
7    },
8    actions: {
9      incremental: (state)=>state=state+1
10   }
11 })
12 // 或者
13 defineStore({
14   id: 'storeId',
15   state: ()=>{},
16   getters: {},
17   actions: {}
18 })

```

 这里的getters 等等都是使用的策略模式，封装不同业务处理逻辑，方便扩展，也符合了单一职责的设计原理。

那么接着来讲defineStore,

```

1  const isSetupStore = typeof setup === 'function'
2  if (typeof idOrOptions === 'string') {
3    id = idOrOptions
4    // the option store setup will contain the actual options in this case
5    options = isSetupStore ? setupOptions : setup
6  } else {
7    options = idOrOptions
8    id = idOrOptions.id
9  }

```

上面的逻辑就是印证了之前说的不同创建store方法内部的解析逻辑。

接着就是useStore,

```

1 function useStore(pinia?: Pinia | null, hot?: StoreGeneric): StoreGeneric {
2   const currentInstance = getCurrentInstance()
3   pinia =
4     // in test mode, ignore the argument provided as we can always retrieve a
5     // pinia instance with getActivePinia()
6     (__TEST__ && activePinia && activePinia._testing ? null : pinia) ||
7     (currentInstance && inject(piniaSymbol))
8   if (pinia) setActivePinia(pinia)
9
10  if (__DEV__ && !activePinia) {
11    throw new Error(
12      `[ ]: getActivePinia was called with no active Pinia. Did you forget to install pin
13      ` +
14      `const pinia = createPinia()\n` +
15      `tapp.use(pinia)\n` +
16      `This will fail in production.`
17    )
18  }
19  pinia = activePinia!
20
21  if (!pinia._s.has(id)) {
22    // creating the store registers it in `pinia._s`
23    if (isSetupStore) {
24      createSetupStore(id, setup, options, pinia)
25    } else {
26      createOptionsStore(id, options as any, pinia)
27    }
28
29    /* istanbul ignore else */
30    if (__DEV__) {
31      // @ts-expect-error: not the right inferred type
32      useStore._pinia = pinia
33    }
34  }
35
36  const store: StoreGeneric = pinia._s.get(id)!
37
38  if (__DEV__ && hot) {
39    const hotId = `__hot:` + id
40    const newStore = isSetupStore
41      ? createSetupStore(hotId, setup, options, pinia, true)
42      : createOptionsStore(hotId, assign({}, options) as any, pinia, true)
43
44    hot._hotUpdate(newStore)
45
46    // cleanup the state properties and the store from the cache
47    delete pinia.state.value[hotId]

```

```

48     pinia._s.delete(hotId)
49   }
50
51   // save stores in instances to access them devtools
52   if (
53     __DEV__ &&
54     IS_CLIENT &&
55     currentInstance &&
56     currentInstance.proxy &&
57     // avoid adding stores that are just built for hot module replacement
58     !hot
59   ) {
60     const vm = currentInstance.proxy
61     const cache = '_pStores' in vm ? vm._pStores! : (vm._pStores = {})
62     cache[id] = store
63   }
64
65   // StoreGeneric cannot be casted towards Store
66   return store as any
67 }

```

代码不长对比之后的章节，一点点的理解，

```

1  const currentInstance = getCurrentInstance()
2  pinia =
3    // in test mode, ignore the argument provided as we can always retrieve a
4    // pinia instance with getActivePinia()
5    (__TEST__ && activePinia && activePinia._testing ? null : pinia) ||
6    (currentInstance && inject(piniaSymbol))
7  if (pinia) setActivePinia(pinia)
8
9  if (__DEV__ && !activePinia) {
10    throw new Error(
11      `[ ]: getActivePinia was called with no active Pinia. Did you forget to install pinia?` +
12      `\tconst pinia = createPinia()\n` +
13      `\tapp.use(pinia)\n` +
14      `This will fail in production.`
15    )
16  }

```

1. `getCurrentInstance` - 获取当前组件的实例，如果返回`null`，则代表没有在组件内部调用`useStore`
2. 获取`pinia`实例，获取的逻辑分成2部分，

```

1  (
2    __TEST__ && activePinia && activePinia._testing
3    ? null
4    : pinia
5  )

```

如果当前为test模式，activePinia 有值，activePinia._testing 没有开启,则返回用户调用useStore传入的第一个参数，就是pinia。因为useStore的2个参数都是可选，所以用户也可能没有传入pinia，那么就会最后返回null。

```
1 (currentInstance && inject(piniaSymbol))
```

如果currentInstance存在就表示useStore在组件内使用，可以直接使用 inject(piniaSymbol)获取pinia。回忆一下上一章节，已经通过使用app.provide(piniaSymbol, pinia)注册。

❶ 此处讲解用户调用useStore传入 pinia 作为参数可能会混乱，useStore在客户端使用一般都不用传入参数，但是pinia支持ssr，所以为了各个application之间区分，就需要手动传入pinia。如果有机会，会在后续章节也讲解pinia的ssr的实现

```
1 if (pinia) setActivePinia(pinia)
2
3 if (__DEV__ && !activePinia) {
4   throw new Error(
5     `[ ]: getActivePinia was called with no active Pinia. Did you forget to install pinia?`
6     + `\tconst pinia = createPinia()\n` +
7     + `\tapp.use(pinia)\n` +
8     + `This will fail in production.`
9   )
10 }
```

如果当前开发状态，无法获取activePinia，那么就抛出错误，没有安装pinia。回忆第一章，通过install方法的setActivePinia(pinia)，就是把pinia赋值给activePinia这个变量了。所以activePinia肯定不是null。

接着分析，

```
1 pinia = activePinia!
2
3 if (!pinia._s.has(id)) {
4   // creating the store registers it in `pinia._s`
5   if (isSetupStore) {
6     createSetupStore(id, setup, options, pinia)
7   } else {
8     createOptionsStore(id, options as any, pinia)
9   }
10
11   /* istanbul ignore else */
12   if (__DEV__) {
13     // @ts-expect-error: not the right inferred type
14     useStore._pinia = pinia
15   }
16 }
```


上面的代码首先尝试从pinia 的所有store找出与传入参数的id匹配的store数据，如果没有，则创建。在下一

章，详细分析创建的实现。

接着分析，

```
1 const store: StoreGeneric = pinia._s.get(id)!
2
3 if (__DEV__ && hot) {
4   const hotId = '__hot:' + id
5   const newStore = isSetupStore
6     ? createSetupStore(hotId, setup, options, pinia, true)
7     : createOptionsStore(hotId, assign({}, options) as any, pinia, true)
8
9   hot._hotUpdate(newStore)
10
11   // cleanup the state properties and the store from the cache
12   delete pinia.state.value[hotId]
13   pinia._s.delete(hotId)
14 }
```

获取刚刚创建好的store，然后判断是否当前是开发环境，而且是否需要为pinia实现vite的hmr热更新。如果需要，则删除缓存的旧store的数据，添加新创建store的数据。

 vite的hmr手动热更新api，让框架或者工具开发者手动调用热更新满足业务需要。对于一般开发者无需关心，因为hmr自动更新已经配置好了。

接着分析，

```
1 if (
2   __DEV__ &&
3   IS_CLIENT &&
4   currentInstance &&
5   currentInstance.proxy &&
6   // avoid adding stores that are just built for hot module replacement
7   !hot
8 ) {
9   const vm = currentInstance.proxy
10   const cache = '_pStores' in vm ? vm._pStores! : (vm._pStores = {})
11   cache[id] = store
12 }
13
14 // StoreGeneric cannot be casted towards Store
15 return store as any
```

首先判断是否开发环境，是否在客户端环境，是否在组件内部，组件实例是否有proxy，不需要热更新。如果前面所有条件满足，则获取组件的proxy，这个proxy可以用来修改组件内部的响应式数据。

尝试从proxy里面获取 _pStores，如果没有，则创建这样一个对象。

然后缓存刚刚创建好的store数据。最后返回创建好的store

```
1 useStore.$id = id
2
3 return useStore
```

然后添加\$id属性, 最后返回 useStore。

 useStore使用了惰性加载的设计，只有真正需要store的地方才开始创建store数据。