36｜数据流原理：Vuex & Pinia源码剖析

2022-01-12 大圣

《玩转Vue 3全家桶》 课程介绍 >

讲述：大圣

时长 08:04 大小 7.39M

你好，我是大圣。

上一讲我们分析了 Vite 原理，今天我们来剖析 Vuex 的原理。其实在之前的课程中，我们已经实现过一个迷你的 Vuex，整体代码逻辑比较简单，基于 Vue 提供的响应式函数 reactive 和 computed 的能力，我们封装了一个独立的共享数据的 store，并且对外暴露了 commit 和 dispatch 方法修改和更新数据，这些原理就不赘述了。

今天我们探讨一下下一代 Vuex5 的提案，并且看一下实际的代码是如何实现的，你学完之后可以对比之前 gvuex mini 版本，感受一下两者的区别。

Vuex5 提案

由于 Vuex 有模块化 namespace 的功能，所以模块 user 中的 mutation add 方法，我们需要使用 `commit('user/add')` 来触发。这样虽然可以让 Vuex 支持更复杂的项目，但是**这种字符串类型的拼接功能，在 TypeScript4 之前的类型推导中就很难实现**。然后就有了 Vuex5 相关提案的讨论，整个讨论过程都是在 GitHub 的 issue 里推进的，你可以访问 GitHub 链接去围观。

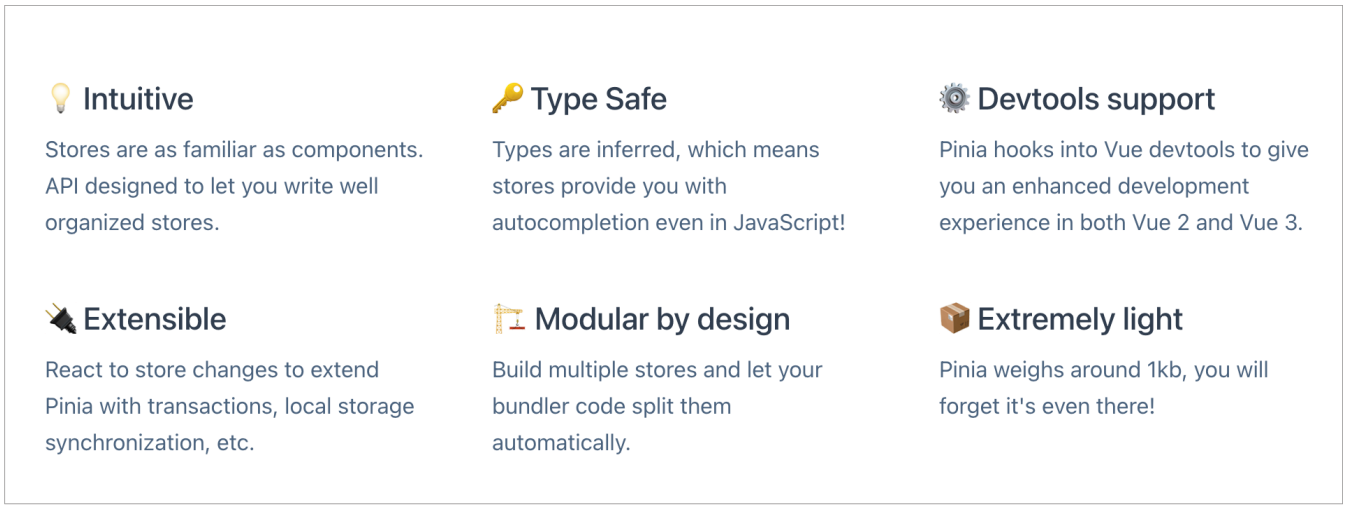
Vuex5 的提案相比 Vuex4 有很大的改进，解决了一些 Vuex4 中的缺点。Vuex5 能够同时支持 Composition API 和 Option API，并且去掉了 namespace 模式，使用组合 store 的方式更好地支持了 TypeScript 的类型推导，还去掉了容易混淆的 Mutation 和 Action 概念，只保留了 Action，并且**支持自动的代码分割**。

我们也可以通过对这个提案的研究，来体验一下在一个框架中如何讨论新的语法设计和实现，以及如何通过 API 的设计去解决开发方式的痛点。你可以在 Github 的提案 RFCs 中看到 Vuex5 的设计文稿，而 Pinia 正是基于 Vuex5 设计的框架。

现在 Pinia 已经正式合并到 Vue 组织下，成为了 Vue 的官方项目，尤雨溪也在多次分享中表示 **Pinia 就是未来的 Vuex**，接下来我们就好好学习一下 Pinia 的使用方式和实现的原理。

Pinia

下图是 Pinia 官网的介绍，可以看到类型安全、Vue 的 Devtools 支持、易扩展、只有 1KB 的体积等优点。快来看下 Pinia 如何使用吧。



首先我们在项目根目录下执行下面的命令去**安装 Pinia 的最新版本**。

复制代码

```
npm install pinia@next
```

然后在 src/main.js 中，我们导入 createPinia 方法，通过 createPinia 方法创建 Pinia 的实例后，再通过 app.use 方法注册 Pinia。

复制代码

```
import { createApp } from 'vue'
import { createPinia } from 'pinia'
import App from './App.vue'
const pinia = createPinia()
const app = createApp(App)
app.use(pinia).mount('#app')

```

然后我们可以在 store 文件夹中创建一个 count.js。下面的代码中我们通过 Pinia 的 defineStore 方法定义了一个 store，store 内部通过 state 返回一个对象，并且通过 Actions 配置修改数据的方法 add。这里使用的语法和 Vuex 比较类似，只是删除了 Mutation 的概念，**统一使用 Actions 来配置**。

复制代码

```


import { defineStore } from 'pinia'

export const useCounterStore = defineStore('count', {
  id:'count',
  state: () => {
    return { count: 1 }
  },
  actions: {
    add() {
      this.count++
    },
  },
})

```

然后我们可以使用 Composition 的方式在代码中使用 store。注意上面的 store 返回的其实就是一个 Composition 风格的函数，使用 useCounterStore 返回 count 后，可以在

add 方法中直接使用 count.add 触发 Actions，实现数据的修改。

复制代码

```
import { useCounterStore } from '../stores/count'

const count = useCounterStore()
function add(){
  count.add()
}


```

我们也可以使用 Composition 风格的语法，去创建一个 store。使用 ref 或者 reactive 包裹后，通过 defineStore 返回，这样 store 就非常接近我们自己分装的 Composition 语法了，也去除了很多 Vuex 中特有的概念，学习起来更加简单。

复制代码

```
export const useCounterStore = defineStore('count', () => {
  const count = ref(0)
  function increment() {
    count.value++
  }

  return { count, increment }
})

```

Pinna 源码

然后我们通过阅读 Pinia 的源码，来看下 Pinia 是如何实现的。

首先我们进入到 Pinia 的 GitHub 中，我们可以在 packages/pinia/src/createPinia.ts 中看到 createPinia 函数的实现。

下面的代码中，我们通过 effectScope 创建一个作用域对象，并且通过 ref 创建了响应式的数据对象 state。然后通过 install 方法支持了 app.use 的注册，内部通过 provide 的语法和全局的 $pinia 变量配置 Pinia 对象，并且通过 use 方法和 toBeInstalled 数组实现了 Pinia 的插件机制。**最后还通过 pinia.use(devtoolsPlugin) 实现了对 VueDevtools 的支持。**

复制代码

```
export function createPinia(): Pinia {
  const scope = effectScope(true)
  // NOTE: here we could check the window object for a state and directly set
  // if there is anything like it with Vue 3 SSR
  const state = scope.run(() => ref<Record<string, StateTree>>({}))!

  let _p: Pinia['_p'] = []
  // plugins added before calling app.use(pinia)
  let toBeInstalled: PiniaPlugin[] = []

  const pinia: Pinia = markRaw({
    install(app: App) {
      // this allows calling useStore() outside of a component setup after
      // installing pinia's plugin
      setActivePinia(pinia)
      if (!isVue2) {
        pinia._a = app
        app.provide(piniaSymbol, pinia)
        app.config.globalProperties.$pinia = pinia
        toBeInstalled.forEach((plugin) => _p.push(plugin))
        toBeInstalled = []
      }
    },

    use(plugin) {
      if (!this._a && !isVue2) {
        toBeInstalled.push(plugin)
      } else {
        _p.push(plugin)
      }
      return this
    },

    _p,
    _a: null,
    _e: scope,
    _s: new Map<string, StoreGeneric>(),
    state,
  })
  if (__DEV__ && IS_CLIENT) {
    pinia.use(devtoolsPlugin)
  }

  return pinia
}

```

通过上面的代码，我们可以看到 Pinia 实例就是 ref({}) 包裹的响应式对象，项目中用到的 state 都会挂载到 Pinia 这个响应式对象内部。

然后我们去看下创建 store 的 defineStore 方法，defineStore 内部通过 useStore 方法去定义 store，并且每个 store 都会标记唯一的 ID。

首先通过 getCurrentInstance 获取当前组件的实例，如果 useStore 参数没有 Pinia 的话，就使用 inject 去获取 Pinia 实例，**这里 inject 的数据就是 createPinia 函数中 install 方法提供的**。

然后设置 activePinia，项目中可能会存在很多 Pinia 的实例，设置 activePinia 就是设置当前活跃的 Pinia 实例。这个函数的实现方式和 Vue 中的 componentInstance 很像，每次创建组件的时候都设置当前的组件实例，这样就可以在组件的内部通过 getCurrentInstance 获取，最后通过 createSetupStore 或者 createOptionsStore 创建组件。

这就是上面代码中我们使用 Composition 和 Option 两种语法创建 store 的不同执行逻辑，最后通过 pinia._s 缓存创建后的 store，_s 就是在 createPinia 的时候创建的一个 Map 对象，防止 store 多次重复创建。**到这 store 创建流程就结束了。**

复制代码

```
export function defineStore(
  // TODO: add proper types from above
  idOrOptions: any,
  setup?: any,
  setupOptions?: any
): StoreDefinition {
  let id: string
  let options:...
  const isSetupStore = typeof setup === 'function'
  if (typeof idOrOptions === 'string') {
    id = idOrOptions
    // the option store setup will contain the actual options in this case
    options = isSetupStore ? setupOptions : setup
  } else {
    options = idOrOptions
    id = idOrOptions.id
  }

  function useStore(pinia?: Pinia | null, hot?: StoreGeneric): StoreGeneric {
    const currentInstance = getCurrentInstance()
```

```
    pinia =
      // in test mode, ignore the argument provided as we can always retrieve
      // pinia instance with getActivePinia()
      (__TEST__ && activePinia && activePinia._testing ? null : pinia) ||
      (currentInstance && inject(piniaSymbol))
    if (pinia) setActivePinia(pinia)

    pinia = activePinia!

    if (!pinia._s.has(id)) {
      // creating the store registers it in `pinia._s`
      if (isSetupStore) {
        createSetupStore(id, setup, options, pinia)
      } else {
        createOptionsStore(id, options as any, pinia)
      }

      /* istanbul ignore else */
      if (__DEV__) {
        // @ts-expect-error: not the right inferred type
        useStore._pinia = pinia
      }
    }

    const store: StoreGeneric = pinia._s.get(id)!

    // save stores in instances to access them devtools
    if (
      __DEV__ &&
      IS_CLIENT &&
      currentInstance &&
      currentInstance.proxy &&
      // avoid adding stores that are just built for hot module replacement
      !hot
    ) {
      const vm = currentInstance.proxy
      const cache = '_pStores' in vm ? vm._pStores! : (vm._pStores = {})
      cache[id] = store
    }

    // StoreGeneric cannot be casted towards Store
    return store as any
  }

  useStore.$id = id

  return useStore
}

```

在 Pinia 中 createOptionsStore 内部也是调用了 createSetupStore 来创建 store 对象。下面的代码中，我们通过 assign 方法实现了 setup 函数，这里可以看到 computed 的实现，内部就是通过 pinia._s 缓存获取 store 对象，调用 store 的 getters 方法来模拟，最后依然通过 createSetupStore 创建。

复制代码

```
function createOptionsStore<
  Id extends string,
  S extends StateTree,
  G extends _GettersTree<S>,
  A extends _ActionsTree
>(
  id: Id,
  options: DefineStoreOptions<Id, S, G, A>,
  pinia: Pinia,
  hot?: boolean
): Store<Id, S, G, A> {
  const { state, actions, getters } = options

  const initialState: StateTree | undefined = pinia.state.value[id]

  let store: Store<Id, S, G, A>

  function setup() {

    pinia.state.value[id] = state ? state() : {}
    return assign(
      localState,
      actions,
      Object.keys(getters || {}).reduce((computedGetters, name) => {
        computedGetters[name] = markRaw(
          computed(() => {
            setActivePinia(pinia)
            // it was created just before
            const store = pinia._s.get(id)!
            return getters![name].call(store, store)
          })
        )
        return computedGetters
      }, {} as Record<string, ComputedRef>)
    )
  }

  store = createSetupStore(id, setup, options, pinia, hot)

  return store as any
}
```

最后我们来看一下 createSetupStore 函数的实现。这个函数也是 Pinia 中最复杂的函数实现，内部的 $patch 函数可以实现数据的更新。如果传递的参数 partialStateOrMutator 是函数，则直接执行，否则就通过 mergeReactiveObjects 方法合并到 state 中，最后生成 subscriptionMutation 对象，**通过 triggerSubscriptions 方法触发数据的更新。**

复制代码

```
function $patch(
  partialStateOrMutator:
    | _DeepPartial<UnwrapRef<S>>
    | ((state: UnwrapRef<S>) => void)
): void {
  let subscriptionMutation: SubscriptionCallbackMutation<S>
  isListening = isSyncListening = false
  // reset the debugger events since patches are sync
  /* istanbul ignore else */
  if (__DEV__) {
    debuggerEvents = []
  }
  if (typeof partialStateOrMutator === 'function') {
    partialStateOrMutator(pinia.state.value[$id] as UnwrapRef<S>)
    subscriptionMutation = {
      type: MutationType.patchFunction,
      storeId: $id,
      events: debuggerEvents as DebuggerEvent[],
    }
  } else {
    mergeReactiveObjects(pinia.state.value[$id], partialStateOrMutator)
    subscriptionMutation = {
      type: MutationType.patchObject,
      payload: partialStateOrMutator,
      storeId: $id,
      events: debuggerEvents as DebuggerEvent[],
    }
  }
  nextTick().then(() => {
    isListening = true
  })
  isSyncListening = true
  // because we paused the watcher, we need to manually call the subscriptio
  triggerSubscriptions(
    subscriptions,
    subscriptionMutation,
    pinia.state.value[$id] as UnwrapRef<S>
  )
}

```

然后定义 partialStore 对象去存储 ID、$patch、Pinia 实例，并且新增了 subscribe 方法。再调用 reactive 函数把 partialStore 包裹成响应式对象，通过 pinia._s.set 的方法实现 store 的挂载。

最后我们通过 pinia._s.get 获取的就是 partialStore 对象，defineStore 返回的方法 useStore 就可以通过 useStore 去获取缓存的 Pinia 对象，实现对数据的更新和读取。

这里我们也可以看到，除了直接执行 Action 方法，还可以通过调用内部的 `count.$patch({count:count+1})` 的方式来实现数字的累加。

复制代码

```
const partialStore = {
  _p: pinia,
  // _s: scope,
  $id,
  $onAction: addSubscription.bind(null, actionSubscriptions),
  $patch,
  $reset,
  $subscribe(callback, options = {}) {
    const removeSubscription = addSubscription(
      subscriptions,
      callback,
      options.detached,
      () => stopWatcher()
    )
    const stopWatcher = scope.run(() =>
      watch(
        () => pinia.state.value[$id] as UnwrapRef<S>,
        (state) => {
          if (options.flush === 'sync' ? isSyncListening : isListening) {
            callback(
              {
                storeId: $id,
                type: MutationType.direct,
                events: debuggerEvents as DebuggerEvent,
              },
              state
            )
          }
        },
        assign({}, $subscribeOptions, options)
      )
    )!

    return removeSubscription
  }
```

```
36
37
38   const store: Store<Id, S, G, A> = reactive(
39     assign({}, partialStore )
40   )
41
42   // store the partial store now so the setup of stores can instantiate each o
43   // creating infinite loops.
44   pinia._s.set($id, store)
45
46
47
```

我们可以看出一个简单的 store 功能，真正需要支持生产环境的时候，也需要很多逻辑的封装。

代码内部除了 __dev__ 调试环境中对 Devtools 支持的语法，还有很多适配 Vue 2 的语法，并且同时支持 Optipn 风格和 Composition 风格去创建 store。createSetupStore 等方法内部也会通过 Map 的方式实现缓存，并且 setActivePinia 方法可以在多个 Pinia 实例的时候获取当前的实例。

这些思路在 Vue、vue-router 源码中都能看到类似的实现方式，这种性能优化的思路和手段也值得我们学习，在项目开发中也可以借鉴。

总结

最后我们总结一下今天学到的内容吧。由于课程之前的内容已经手写了一个迷你的 Vuex，这一讲我们就越过 Vuex4，直接去研究了 Vuex5 的提案。

Vuex5 针对 Vuex4 中的几个痛点，去掉了容易混淆的概念 Mutation，并且去掉了对 TypeScript 不友好的 namespace 功能，使用组合 store 的方式让 Vuex 对 TypeScript 更加友好。

Pinia 就是 Vuex5 提案产出的框架，现在已经是 Vue 官方的框架了，也就是 Vuex5 的实现。在 Pinia 的代码中，我们通过 createPinia 创建 Pinia 实例，并且可以通过 Option 和 Composition 两种风格的 API 去创建 store，返回 useStore 函数获取 Pinia 的实例后，就可以进行数据的修改和读取。

思考

最后留一个思考题吧。对于数据共享语法，还有 provide/inject 和自己定义的 Composition，什么时候需要使用 Pinia 呢？

欢迎到评论区分享你的想法，也欢迎你把这一讲的内容分享给你的朋友们，我们下一讲再见！

上一篇 35 | Vite原理：写一个迷你的Vite

下一篇 37 | 前端路由原理：vue-router源码剖析

精选留言 (1)

写留言

james
2022-01-13

没有关联的组件之间可以使用 pinia