

3.2 Vue3.0响应式原理

1.Vue3.0对比Vue2.0响应式差异

2. 核心方法

3.Proxy

3.reactivity

4.收集依赖 effect&track

5.触发更新trigger

6.ref

7.toRefs

8.computed

1.Vue3.0对比Vue2.0响应式差异

- Proxy对象实现属性监听
 - 不需要遍历所有属性,通过Object.defineProperty转换getter和setter
- 多层属性嵌套,在访问属性过程中处理下一级属性
- 默认监听动态添加的属性
- 默认监听属性的删除操作
- 默认监听数组索引和length属性
- 可以作为单独的模块使用

2. 核心方法

- reactive/ref/toRefs/computed
- effect
- track
- trigger

3.Proxy

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scal
```

```

    e=1.0">
6   <title>Document</title>
7   </head>
8   <body>
9   <script>
10    'use strict'
11    // 问题1: set 和 deleteProperty 中需要返回布尔类型的值
12    // 在严格模式下, 如果返回 false 的话会出现 Type Error 的异常
13
14    // 问题2: Proxy 和 Reflect 中使用的 receiver
15
16    // Proxy 中 receiver: Proxy 或者继承 Proxy 的对象
17    // Reflect 中 receiver: 如果 target 对象中设置了 getter, getter 中
    的 this 指向 receiver
18    const target = {
19      foo: 'xxx',
20      bar: 'yyy'
21    }
22    const proxy = new Proxy(target,{
23      get(target,key, receiver) {
24        return Reflect.get(target,key,receiver)
25      },
26      set(target, key,value,receiver) {
27        return Reflect.set(target,key,value,receiver)
28      },
29      deleteProperty(target,key) {
30        return Reflect.deleteProperty(target,key)
31      }
32    })
33    proxy.foo = 'zzzz'
34    delete proxy.foo
35  </script>
36 </body>
37 </html>

```

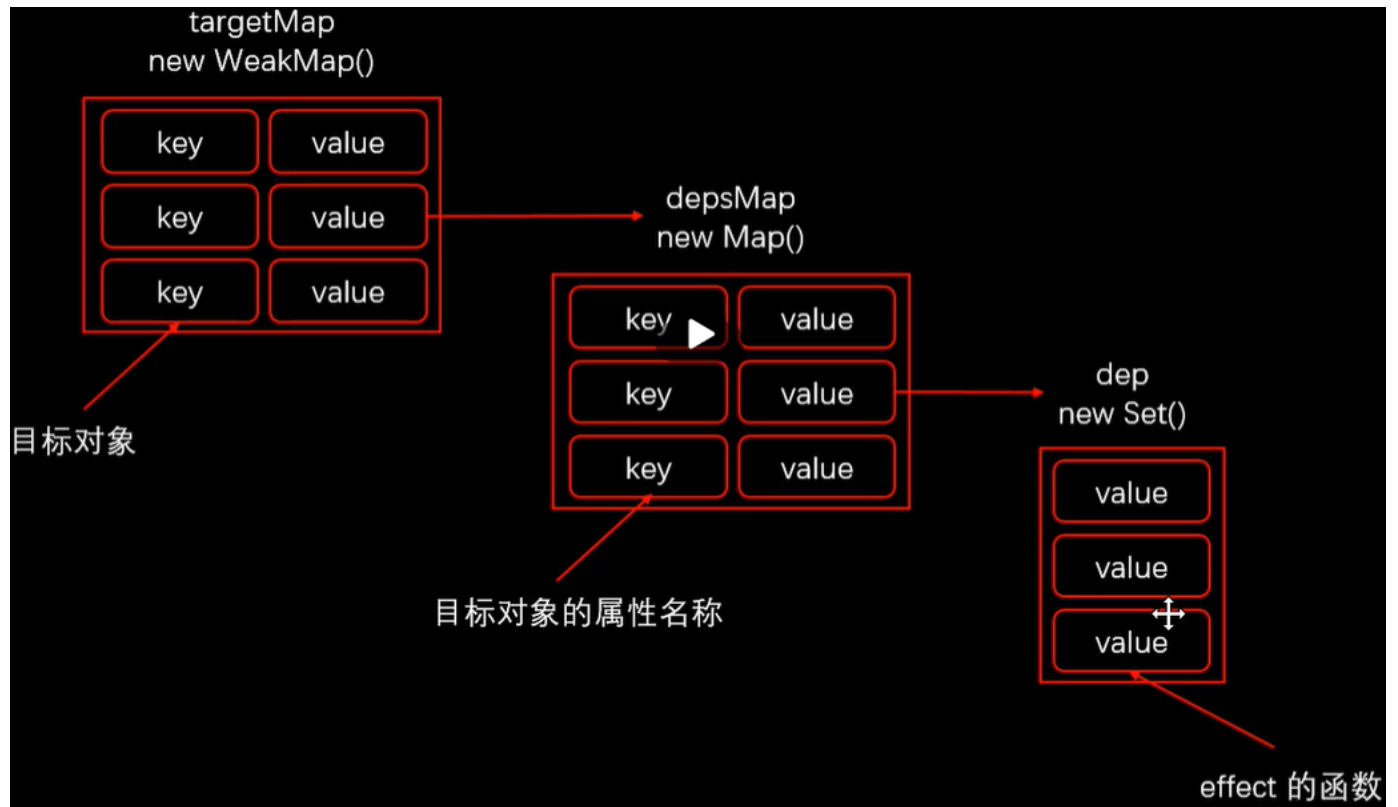
3.reactivity

- 接收一个参数,判断参数是否是对象
- 创建拦截器对象handler,设置get/set/deleteProperty

- 返回proxy对象

```
1 const isObject = val => val !== null && typeof val === 'object'
2 const convert = target => isObject(target) ? reactive(target) : t
  target
3 const hasOwnProperty = Object.prototype.hasOwnProperty
4 const hasOwn = (target, key) => hasOwnProperty.call(target, key)
5
6 export function reactive(target) {
7   // 不是对象没有响应式可言,直接返回值
8   if(!isObject(target)) return target
9   const handler = {
10     get(target,key,receiver) {
11       const result = Reflect.get(target, key, receiver)
12       console.log(result)
13       return convert(result)
14     },
15     set(target,key,value,receiver) {
16       const oldValue = Reflect.get(target, key, receiver)
17       let result = true
18       console.log('set')
19       if(oldValue !== value) {
20         result = Reflect.set(target, key, value, receiver)
21       }
22       return result
23     },
24     deleteProperty(target,key) {
25       console.log('del')
26       const hadKey = hasOwn(target, key)
27       const result = Reflect.deleteProperty(target, key)
28       if(hadKey && result) {
29
30       }
31       return result
32     }
33   }
34   return new Proxy(target,handler)
35 }
```

4.收集依赖 effect&track



```
1 const isObject = val => val !== null && typeof val === 'object'
2 const convert = target => isObject(target) ? reactive(target) : t
  target
3 const hasOwnProperty = Object.prototype.hasOwnProperty
4 const hasOwn = (target, key) => hasOwnProperty.call(target, key)
5
6 export function reactive(target) {
7   // 不是对象没有响应式可言,直接返回值
8   if(!isObject(target)) return target
9   const handler = {
10     get(target, key, receiver) {
11       track(target, key)
12       const result = Reflect.get(target, key, receiver)
13       // 收集依赖
14       console.log(result)
15       return convert(result)
16     },
17     set(target, key, value, receiver) {
```

```

18     const oldValue = Reflect.get(target, key, receiver)
19     let result = true
20     console.log('set')
21     if(oldValue !== value) {
22         result = Reflect.set(target, key, value, receiver)
23     }
24     return result
25 },
26 deleteProperty(target, key) {
27     console.log('del')
28     const hadKey = hasOwn(target, key)
29     const result = Reflect.deleteProperty(target, key)
30     if(hadKey && result) {
31
32     }
33     return result
34 }
35 }
36 return new Proxy(target, handler)
37 }
38 let activeEffect = null
39 export function effect(callback) {
40     activeEffect = callback
41     callback() // 访问响应式对象属性，去收集依赖
42     activeEffect = null
43 }
44 // 收集依赖
45 let targetMap = new WeakMap()
46 export function track(target, key) {
47     if(!activeEffect) return
48     let depsMap = targetMap.get(target)
49     if(!depsMap) {
50         targetMap.set(target, (depsMap = new Map()))
51     }
52     let dep = depsMap.get(key)
53     if(!dep) {
54         depsMap.set(key, (dep = new Set()))
55     }
56     dep.add(activeEffect)
57 }

```

5.触发更新trigger

```
1 const isObject = val => val !== null && typeof val === 'object'
2 const convert = target => isObject(target) ? reactive(target) : t
  target
3 const hasOwnProperty = Object.prototype.hasOwnProperty
4 const hasOwn = (target, key) => hasOwnProperty.call(target, key)
5
6 export function reactive(target) {
7   // 不是对象没有响应式可言,直接返回值
8   if(!isObject(target)) return target
9   const handler = {
10     get(target,key,receiver) {
11       track(target, key)
12       const result = Reflect.get(target, key, receiver)
13       // 收集依赖
14       console.log(result)
15       return convert(result)
16     },
17     set(target,key,value,receiver) {
18       const oldValue = Reflect.get(target, key, receiver)
19       let result = true
20       console.log('set')
21       if(oldValue !== value) {
22         result = Reflect.set(target, key, value, receiver)
23         // 触发更新
24         trigger(target, key)
25       }
26       return result
27     },
28     deleteProperty(target,key) {
29       console.log('del')
30       const hadKey = hasOwn(target, key)
31       const result = Reflect.deleteProperty(target, key)
32       if(hadKey && result) {
33         // 触发更新
34         trigger(target, key)
```

```

35     }
36     return result
37   }
38 }
39 return new Proxy(target,handler)
40 }
41 let activeEffect = null
42 export function effect(callback) {
43   activeEffect = callback
44   callback() // 访问响应式对象属性，去收集依赖
45   activeEffect = null
46 }
47 // 收集依赖
48 let targetMap = new WeakMap()
49 export function track(target , key) {
50   if(!activeEffect) return
51   let depsMap = targetMap.get(target)
52   if(!depsMap) {
53     targetMap.set(target, (depsMap = new Map()))
54   }
55   let dep = depsMap.get(key)
56   if(!dep) {
57     depsMap.set(key,(dep = new Set()))
58   }
59   dep.add(activeEffect)
60 }
61 // 触发更新
62 export function trigger (target,key) {
63   const depsMap = targetMap.get(target)
64   if(!depsMap) return
65   const dep = depsMap.get(key)
66   if(dep) {
67     dep.forEach(effect => {
68       effect()
69     });
70   }
71 }

```

6.ref

- ref可以把基本数据类型数据,转成响应式对象
- ref返回的对象,重新赋值成对象也是响应式的
- reactive返回的对象,重新赋值丢失响应式
- reactive返回的对象不可以解构

```
1 export function ref(raw) {
2   // 判断 raw 是否是ref 创建的对象, 如果是的话直接返回
3   if (isObject(raw) && raw.__v_isRef) {
4     return
5   }
6   let value = convert(raw)
7   const r = {
8     __v_isRef: true,
9     get value() {
10       track(r, 'value')
11       return value
12     },
13     set value(newValue) {
14       if(newValue !== value) {
15         raw = newValue
16         value = convert(raw)
17         trigger(r, 'value')
18       }
19     }
20   }
21   return r
22 }
```

7.toRefs

```
1 export function toRefs(proxy) {
2   const ret = proxy instanceof Array ? new Array(proxy.length) :
3     {}
4   for (const key in proxy) {
5     ret[key] = toProxyRef(proxy, key)
6   }
7 }
```



```
7 function toProxyRef (proxy, key) {  
8   const r = {  
9     __v_isRef: true,  
10    get value () {  
11      return proxy[key]  
12    },  
13    set value (newValue) {  
14      proxy[key] = newValue  
15    }  
16  }  
17  return r  
18 }
```

8.computed

```
1 export function computed(getter) {  
2   const result = ref()  
3   effect(() => (result.value = getter()))  
4   return result;  
5 }
```