第三篇 具体的响应式方案

项目地址：https://github.com/Q1173419450/responder/tree/main

总结第四章

简单实现了一个 副作用 与 响应式数据 的类，做到数据变化触发到副作用的重新执行

定义了响应式的基本结构

new Proxy()、get、set、track、trigger、options

本章概括

本篇将会更加细节的完善响应式系统，使其能适应更多的数据结构。例如：深层次对象、数组、Map、Set 等

上一章的响应式缺失什么？

- 更细节的 Object 处理
  - 访问属性
  - 判断对象或者原型是否在给定的 key 上: key in obj
  - 使用 for...in 循环遍历对象时: for(const key in obj)
- 响应和浅响应
  - Reactive
  - shallowReactive
  - 只读和浅只读
  - Readonly
  - shallowReadonly
- 代理数组
  - Get
    - 索引访问：arr[0]
    - 访问长度：arr.length
    - for...in、for...of 循环
    - 原型方法：concat、join、every、some、find、findIndex、includes
  - set

    - 通过索引修改元素：arr[1] = 1;
    - 直接修改数组长度：arr.length = 0
    - 修改数组方法：push、pop、shift、unshift、splice、fill、sort
- Map 和 Set

知识补充

Proxy 和 Reflect

- Proxy： 能够代理对象，能让我们对对象进行一系列的操作
    - 代理：允许我们拦截和重写定义对象的基本操作
        - 拦截: `Proxy` 的第二个参数就是拦截对象的基本操作
    - 基本操作：获取值、改变值、删除值、遍历值等都属于基本操作
- Reflect: `Proxy` 的拦截器方法名与 `Reflect` 的方法名一致

> QA: 为什么可以直接操作对象，却需要用 Reflect 去操作呢?
>
> 答：其实就是 receiver 的第三个参数的作用，例如：有时候我们在做继承时，很可能导致不正常的响应更新，receiver 保证我们的 this 是我们想要的对象

```
const obj = {
  foo: 1,
  get bar() {
    return this.foo
  }
}

const p = new Proxy(obj, {
  get(target, key) {
    track(target, key, receiver);
    // target 为原始对象 obj，所以 this 的执行还是原对象，所以不触发响应
    // return target[key]
    // receiver 就可以标识我们在读的属性，避免 this 指向问题
    return Reflect.get(target, key, receiver)
  },
  set(target, key, value) {
    target[key] = value;
    trigger(target, key)
  }
})

effect(() => {
  console.log(p.bar)
```

```
24 })
25
26 // 当我们跟更新 foo 时，无法触发副作用
27 p.foo++
```

Proxy 的工作原理

ECMA 262 中的俩种对象：常规对象和异质对象

对象内部方法和内部槽

内部方法：当我们对对象进行操作时，引擎内部调用的方法。（我们只管使用，底层帮我们兜底，告诉我们什么方法能用什么不能用）

内部槽：`[[xxx]]` *代表内部槽，作为标识位*

不同类型的对象，相同的内部方法，可能有不同的逻辑

```
1 obj.foo // 内部方法就为 [[get]]
```

异质对象：则说明方法不是使用 ECMA 中某些规定的规范实现的。例如：Proxy（这就和我们版本不断迭代，最开始的规范和现在的规范总是不一样的，所以做了简单的区分）

所以 Proxy 的本质其实就是内部方法的实现不同，如果我们不做拦截，则会调用原始对象的内部方法

1、更细节的 Object 处理

1.1 访问属性

直接代理到 `get` 方法，则可以访问属性并，收集对应对象的依赖

api: [[get]] 内部方法名为 get

```
1 new Proxy(obj, {
2   get(target, key) {
3     let res = Reflect.get(target, key);
4     track(target, key);
5     return res;
6   };
7 })
```

1.2 key in obj

`key in obj` 的主要作用：判断值是否在对象上

api：`[[hasProperty]]` 内部方法名叫 `has`

```
new Proxy(obj, {
  has(target, key) {
    track(target, key);
    return Reflect.has(target, key)
  }
})
```

1.3 for(const key in obj)

对象自身拥有的键

api：`ownKeys`

```
// for...in 的专属 key
const ITERATE_KEY = Symbol();
new Proxy(obj, {
  ownKeys(target) {
    track(target, ITERATE_KEY)
    return Reflect.ownKeys(target)
  }
})
```

为什么需要传递唯一的 `key` 呢?

因为 `for...in` 这种形式没有与任何值进行绑定，所以需要传递唯一的 `key` 进行标识。

当对象的值发生新增或删除时，可以更好的处理响应。

1.4 对象遍历属性的隐式影响

1.4.1 新增属性

当我们对对象进行新增操作时，对应的 `for...in` 的 `effect` 不执行，所以当我们新增操作想要重新触发 `effect` 执行 `for...in` 操作时，我们就需要将对应的副作用加入到队列中，进行执行。

```
// main.js
const obj3 = reactive({ foo: 1 });
effect(() => {
  /* for in 算是读取数据，执行了一次 track */
  for(const key in obj3) {
    console.log(key, obj3[key]);
```

```
  }
})
obj3.bar = 2;

// effect.js
// 添加对应的 effect 到 执行队列中
const iterateEffects = depsMap.get(ITERATE_KEY)
iterateEffects && iterateEffects.forEach(effectFn => {
  if (effectFn !== activeEffect) {
    effectToRun.add(effectFn);
  }
})
```

1.4.2 修改属性

当我们不管对象是否有新增属性，就直接添加 `ITERATE_KEY` 响应时，又是有问题的，因为当我只是修改对象而没有新增属性， `effect` 又会执行。

这时候我们可以在 `set` 进行一个属性是否存在的判断，并添加是 `ADD` 还是 `SET` 的标识，就可以区分出这两种操作。

```
//baseHandlers.js
const TriggerType = {
  SET: 'SET',
  ADD: 'ADD',
}
const type = Object.prototype.hasOwnProperty.call(target, key) ? TriggerType.SET
trigger(target, key, type);

// effect.js
if (type === TriggerType.ADD) {
  const iterateEffects = depsMap.get(ITERATE_KEY)
  iterateEffects && iterateEffects.forEach(effectFn => {
    if (effectFn !== activeEffect) {
      effectToRun.add(effectFn);
    }
  })
}
```

1.4.3 删除属性

删除属性首先我们需要先代理删除

key: `[[Delete]]` 内部方法为 `deletePropery`

```
// baseHandlers.js
const TriggerType = {
  SET: 'SET',
  ADD: 'ADD',
  DELETE: 'DELETE'
}

function deleteProperty(target, key) {
  const hadKey = Object.prototype.hasOwnProperty.call(target, key);
  const res = Reflect.deleteProperty(target, key);

  /*
    1. 删除的是对象拥有的
    2. res 返回正常才行（可能失败）例如：不是 superReference、删除状态 等
  */
  if (res && hadKey) {
    trigger(target, key, TriggerType.DELETE);
  }
  return res;
}

// effect.js
if (type === TriggerType.ADD || type === TriggerType.DELETE) {
  const iterateEffects = depsMap.get(ITERATE_KEY)
  iterateEffects && iterateEffects.forEach(effectFn => {
    if (effectFn !== activeEffect) {
      effectToRun.add(effectFn);
    }
  })
}
```

2、合理触发响应

我们需要优化我们的响应式系统，让他能更高效的触发响应

2.1 值没变化不触发更新

当值没发生变化，不触发 `trigger`

```
function createSetter() {
  return function set(target, key, value, receiver) {
    const oldVal = target[key];
    const type = Object.prototype.hasOwnProperty.call(target, key) ? TriggerType

    let res = Reflect.set(target, key, value, receiver);
```

```
    // 值相同不触发更新
    if (oldVal !== value) {
      trigger(target, key, type);
    }
    return res;
  };
}
```

2.2 处理 NaN 特殊情况

因为 `NaN` 的特殊性，自己和自己比较为 `false`，所以我们把值和自身比较

```
function createSetter() {
  return function set(target, key, value, receiver) {
    const oldVal = target[key];
    const type = Object.prototype.hasOwnProperty.call(target, key) ? TriggerType

    let res = Reflect.set(target, key, value, receiver);
    // 都不是 NaN 的时候才触发响应
    if (oldVal !== value && (oldVal === oldVal || value === value)) {
      trigger(target, key, type);
    }
    return res;
  };
}
```

2.3 原型操作

当设置原型并进行访问时属性时，访问顺序会顺着原型链不断的查找下去，直到找到为止，但这会导致 副作用 会触发过多次，这时候就需要针对的进行优化

这时候 Proxy 的第四个参数就发挥了他的作用 receiver：**确定他的原型是其他对象还是自己**

```
// main.js
const obj1 = {};
const proto = { bar: 1 }
const child = reactive(obj1);
const parent = reactive(proto);
Object.setPrototypeOf(child, parent);

effect (() => {
  console.log(child.bar);
})

```

```
child.bar = 2;

// baseHandlers.js
function createGetter() {
  return function get(target, key, receiver) {
    if (key === 'raw') return target; // 原型相关：设置原始数据
    let res = Reflect.get(target, key, receiver);
    track(target, key);
    return res;
  };
}

function createSetter() {
  return function set(target, key, value, receiver) {
    const oldVal = target[key];
    const type = Object.prototype.hasOwnProperty.call(target, key) ? TriggerType

    let res = Reflect.set(target, key, value, receiver);
    //
    + if(target === receiver.raw) {
      if (oldVal !== value && (oldVal === oldVal || value === value)) {
        trigger(target, key, type);
      }
    }
    return res;
  };
}
```

3、浅响应 和 深响应 & 只读 和 浅只读

从 Vue3 暴露的 API 出发，设计源码

3.1 浅响应 和 深响应

深响应 Reactive

当我们需要监听深层次对象时，就需要对 set 进行对象判断

```
// 收集响应时
// 判断是否还是对象，进行一个递归处理
if (res !== null && typeof res === "object") {
  return reactive(res);
}

return res
```

浅响应 shallowReactive

在收集依赖时，给定一个参数，判断是否只收集第一层属性

3.2 只读 和 浅只读

数据只读，不能修改

只读 readOnly

收集依赖时，还是像 Reactive 一样对依赖递归收集，给定一个 readonly 判定值，set、deleteProperty 值时进行，给一个 warn 进行拦截

```
if (res !== null && typeof res === "object") {
  return isReadonly ? readonly(res) : reactive(res);
}
```

浅只读 shallowReadonly

则是和 浅响应 一样，使用 isShallow 进行拦截

4、代理数组

数组也是对象，所以我们在访问时，是可以正常响应，但数组有许多的遍历、修改方式，这些都要将其拦截下来，进行响应式操作

4.1 读取操作

4.2 通过索引设置、访问

数组通过索引访问本质也是对象访问属性，所以可以正常收集响应

4.3 直接设置、访问 length 属性

直接设置和访问 length 则需要到新增数据会改变 length（类似对象的 for..in 操作）、修改 length 也会隐式的影响数组元素

```
// array.js
const obj = reactive(['foo']);
effect(() => {
   console.log(obj.length);
})
obj[1] = 'bar';

// baseHandlers.js
```

```
const type = Array.isArray(target) ?
      (Number(key) < target.length ? TriggerType.SET : TriggerType.ADD) :
      Object.prototype.hasOwnProperty.call(target, key) ? TriggerType.SET : Trig

// effect.js
```

for...in、for...of 循环遍历

> for...in 循环

for...in 循环本质上是在修改 length 属性后，才会触发副作用，所以我们只需要监听 key 值为 length 属性的变化就可以了

```
function ownKeys(target) {
  /* 遍历数组只需要监听数组长度变化 */
  const key = Array.isArray(target) ? 'length' : ITERATE_KEY
  track(target, key)
  return Reflect.ownKeys(target)
}
```

> for...of 循环

for...of 不应该收集 Symbol.iterator 这类 symbol 值

```
 if(!isReadonly && typeof key !== 'symbol') {
   track(target, key);
 }
```

> 原型方法：concat、join、every、some、find、findIndex、includes

查找方法可能在访问 array 元素的时候得到一个 代理对象（我们在查找时，如果创建了不同的代理对象，查找等方法是找不到的）

```
const obj = {}
const arr = reactive([obj])
console.log(arr.includes(arr[0])) // false

export function reactive(target) {
  const existionProxy = reactiveMap.get(target);
  if (existionProxy) return existionProxy

  const proxy = createReactiveObject(target, mutableHandlers);
  reactiveMap.set(target, proxy);
```

```
11
12    return proxy
13  }
```

为了防止用户将原始值与响应式对象混用，重写了 原型方法，即当响应式对象找不到时，则去找其原始数据 raw

```
1  // array.js
2  const obj = {}
3  const arr = reactive([obj])
4  console.log(arr.includes(obj))
5
6  const arrayInstrumentations = {};
7  ['includes', 'indexOf', 'lastIndexOf'].forEach((method) => {
8    const originMethod = Array.prototype[method];
9    arrayInstrumentations[method] = function(...args) {
10     let res = originMethod.apply(this, args)
11     if (res === false) {
12       /* 第一次查找为 false，再去查找原生对象 */
13       res = originMethod.apply(this.raw, args);
14     }
15     return res
16   }
17 });
```

> 修改数组方法：push、pop、shift、unshift、splice、fill、sort

本质其实就是在没监听 length 属性的时候，不需要自己去收集 length 的响应

通过一个是否追踪响应的变量，来对响应式收集的控制

```
1  ['push', 'pop', 'shift', 'unshift', 'splice'].forEach((method) => {
2    const originMethod = Array.prototype[method];
3    arrayInstrumentations[method] = function(...args) {
4      pauseTracking()
5      let res = originMethod.apply(this, args)
6      enableTracking()
7      return res;
8    }
9  })
```

5、Map 和 Set

5.1 注意事项

Set.size

This 指向问题，proxy 没有 [[SetData]] 内部方法，所以将 this 指向 原始对象

get.delete()

也是执行方法的 this 指向问题

5.2 建立响应联系

Size 联系

拦截一系列方法 add、delete

5.3 避免污染原始数据

set 操作的监听（给定标识）

拦截方法中不应该操作代理数据，直接设置到 value.raw 中

5.4 拦截 forEach、迭代器、values、keys 方法

Entries、values、keys 返回值本身不可迭代，所以需要加入 [Symbol.iterator] 标识

6、Ref

主要api：ref、toRef

ref 是用来弥补 reactive 的一些缺陷而诞生的，reactive 我们知道是代理引用类型，而原始类型，reactive 无法代理

6.1、代理原始值

简单的构造一个函数，返回一个带 value 的对象，做到限制用户不胡乱使用

使用带访问器的属性，来进行 value 的获取

6.2、引用类型的响应丢失问题

```
// 响应会丢失，因为 log 中解构出来为普通对象
const obj = reactive({ foo: 1 })
console.log(...obj)
```