Reactivity模块基本使用

安装响应式模块

```
pnpm install @vue/reactivity -w
```

```
<div id="app"></div>
<script type="module">
  import { reactive, effect } from
"/node_modules/@vue/reactivity/dist/reactivity.esm-browser.js";
  const state = reactive({ name: 'jw', age: 30 })
  effect(() => { // 副作用函数 默认执行一次，响应式数据变化后再次执行
    app.innerHTML = state.name + '今年' + state.age + '岁了'
  });
  setTimeout(() => {
    state.age++;
  }, 1000)
</script>
```

> *reactive方法会将对象变成proxy对象，effect中使用reactive对象时会进行依赖收集，稍后属性变化时会重新执行effect函数~。*

1.编写reactive函数

```
import { isObject } from "@vue/shared"
function createReactiveObject(target: object, isReadonly: boolean) {
    if (!isObject(target)) {
        return target
    }
}
// 常用的就是reactive方法
export function reactive(target: object) {
    return createReactiveObject(target, false)
}
// 后面的方法，不是重点我们先不进行实现...
/*
export function shallowReactive(target: object) {
    return createReactiveObject(target, false)
}
export function readonly(target: object) {
```

```
    return createReactiveObject(target, true)
}
export function shallowReadonly(target: object) {
    return createReactiveObject(target, true)
}
*/
```

```
export function isObject(value: unknown) : value is Record<any,any> {
    return typeof value === 'object' && value !== null
}
```

> *由此可知这些方法接受的参数必须是一个对象类型。否则没有任何效果*

```
const reactiveMap = new WeakMap(); // 缓存列表
const mutableHandlers: ProxyHandler<object> = {
    get(target, key, receiver) {
        // 等会谁来取值就做依赖收集
        const res = Reflect.get(target, key, receiver);
        return res;
    },
    set(target, key, value, receiver) {
        // 等会赋值的时候可以重新触发effect执行
        const result = Reflect.set(target, key, value, receiver);
        return result;
    }
}
function createReactiveObject(target: object, isReadonly: boolean) {
    if (!isObject(target)) {
        return target
    }
    const exisitingProxy = reactiveMap.get(target); // 如果已经代理过则直接
返回代理后的对象
    if (exisitingProxy) {
        return exisitingProxy;
    }
    const proxy = new Proxy(target, mutableHandlers); // 对对象进行代理
    reactiveMap.set(target,proxy)
    return proxy;
}
```

> *这里必须要使用Reflect进行操作，保证this指向永远指向代理对象*

```
let person = {
  name: 'jw',
  get aliasName() {
    return '**' + this.name + '**';
```

```
  }
}
let p = new Proxy(person, {
  get(target, key, receiver) {
    console.log(key)
    // return Reflect.get(target,key,receiver)
    return target[key]
  }
})
// 取aliasName时，我希望可以收集aliasName属性和name属性
p.aliasName
// 这里的问题出自于 target[key] ,target指代的是原对象并不是代理对象
```

> *将对象使用proxy进行代理，如果对象已经被代理过，再次重复代理则返回上次代理结果。那么，如果将一个代理对象传入呢?*

```
const enum ReactiveFlags {
    IS_REACTIVE = '__v_isReactive'
}
const mutableHandlers: ProxyHandler<object> = {
    get(target, key, receiver) {
        if(key === ReactiveFlags.IS_REACTIVE){ // 在get中增加标识，当获取IS_REACTIVE时返回true
            return true;
        }
    }
}
function createReactiveObject(target: object, isReadonly: boolean) {
    if (!isObject(target)) {
      return target;
    }
    if(target[ReactiveFlags.IS_REACTIVE]){ // 在创建响应式对象时先进行取值，看是否已经是响应式对象
        return target
    }
}
```

> *这样我们防止重复代理就做好了~~~，其实这里的逻辑相比Vue2真的是简单太多了。*

> *这里我们为了代码方便维护，我们将mutableHandlers抽离出去到baseHandlers.ts中。*

2.编写effect函数

```
export let activeEffect = undefined; // 当前正在执行的effect
export class ReactiveEffect {
  active = true; // 标记effect是否处于激活状态
  deps = []; // 收集effect中使用到的属性
  parent = undefined;
  constructor(public fn) {}
  run() {
    if (!this.active) {
      // 不是激活状态，就不需要考虑依赖收集，也就是不需要将这个effect放到全局变量上
      return this.fn();
    }

    try {
      this.parent = activeEffect; // 当前的effect就是他的父亲
      activeEffect = this; // 设置成正在激活的是当前effect
      return this.fn();
    } finally {
      activeEffect = this.parent; // 执行完毕后还原activeEffect
      this.parent = undefined;
    }
  }
}
export function effect(fn, options?) {
  const _effect = new ReactiveEffect(fn); // 创建响应式effect
  _effect.run(); // 让响应式effect默认执行
}
```

3.依赖收集

> *默认执行effect时会对属性，进行依赖收集*

```
get(target, key, receiver) {
    if (key === ReactiveFlags.IS_REACTIVE) {
        return true;
    }
    const res = Reflect.get(target, key, receiver);
    track(target, 'get', key);  // 依赖收集
    return res;
}
```

```
const targetMap = new WeakMap(); // 记录依赖关系
export function track(target, type, key) {
    if (activeEffect) {
        let depsMap = targetMap.get(target); // {对象: map}
```

```
        if (!depsMap) {
            targetMap.set(target, (depsMap = new Map()))
        }
        let dep = depsMap.get(key);
        if (!dep) {
            depsMap.set(key, (dep = new Set())) // {对象: { 属性 :[ dep,
dep ]}}
        }
        let shouldTrack = !dep.has(activeEffect)
        if (shouldTrack) {
            dep.add(activeEffect);
            activeEffect.deps.push(dep); // 让effect记住dep，这样后续可以用
于清理
        }
    }
}
```

> *将属性和对应的effect维护成映射关系，后续属性变化可以触发对应的effect函数重新run*

4.触发更新

```
set(target, key, value, receiver) {
    // 等会赋值的时候可以重新触发effect执行
    let oldValue = target[key]
    const result = Reflect.set(target, key, value, receiver);
    if (oldValue !== value) {
        trigger(target, 'set', key, value, oldValue)
    }
    return result;
}
```

```
export function trigger(target, type, key?, newValue?, oldValue?) {
    const depsMap = targetMap.get(target); // 获取对应的映射表
    if (!depsMap) {
        return
    }
    const dep = depsMap.get(key);
    dep && dep.forEach(effect => {
        if (effect !== activeEffect) effect.run(); // 防止在effect中修改数
据造成死循环
    })
}
```

5.分支切换与cleanup

在渲染时我们要避免副作用函数产生的遗留依赖问题

```
const state = reactive({ flag: true, name: 'jw', age: 30 })
effect(() => { // 副作用函数 (effect执行渲染了页面)
    console.log('render')
    document.body.innerHTML = state.flag ? state.name : state.age
});
setTimeout(() => {
    state.flag = false;
    setTimeout(() => {
        console.log('修改name，原则上不更新')
        state.name = 'zf'
    }, 1000);
}, 1000)
```

```
function cleanupEffect(effect) {
    const { deps } = effect; // 清理effect
    for (let i = 0; i < deps.length; i++) {
        deps[i].delete(effect);
    }
    effect.deps.length = 0;
}
class ReactiveEffect {
    active = true;
    deps = []; // 收集effect中使用到的属性
    parent = undefined;
    constructor(public fn) { }
    run() {
        try {
            this.parent = activeEffect; // 当前的effect就是他的父亲
            activeEffect = this; // 设置成正在激活的是当前effect
+           cleanupEffect(this);
            return this.fn(); // 先清理在运行
        }
    }
}
```

> *这里要注意的是：触发时会进行清理操作（清理effect），在重新进行收集（收集effect）。在循环过程中会导致死循环。*

```
let effect = () => {};
let s = new Set([effect])
s.forEach(item=>{s.delete(effect); s.add(effect)}); // 这样就导致死循环了
```

- 属性变化时执行对应dep中所有的effect （循环set）

- 清理时找到对应的dep删除对应的effect （set中移除了effect）
- 重新收集时dep再次收集了effect （set中添加了effect）

```
export function trigger(target, type, key?, newValue?, oldValue?) {
  const depsMap = targetMap.get(target);
  if (!depsMap) {
    return;
  }
  const dep = depsMap.get(key);

  if (dep) { // 这里将要执行的effect拷贝一份
    const effects = [...dep];
    effects.forEach((effect) => {
      if (effect !== activeEffect) effect.run();
    });
  }
}
```

6.停止effect

```
export class ReactiveEffect {
    stop(){
        if(this.active){
            cleanupEffect(this);
            this.active = false
        }
    }
}
export function effect(fn, options?) {
    const _effect = new ReactiveEffect(fn);
    _effect.run();

    const runner = _effect.run.bind(_effect); // 运行用户调用effect返回值再次渲染
    runner.effect = _effect;
    return runner; // 返回runner
}
```

7.调度执行

trigger触发时，我们可以自己决定副作用函数执行的时机、次数、及执行方式

```
export function effect(fn, options:any = {}) {
    const _effect = new ReactiveEffect(fn,options.scheduler); // 创建响应式effect

    _effect.run(); // 让响应式effect默认执行
```

```
    const runner = _effect.run.bind(_effect);
    runner.effect = _effect;
    return runner; // 返回runner
}

export function trigger(target, type, key?, newValue?, oldValue?) {
    const depsMap = targetMap.get(target); // 获取对应的映射表
    if (!depsMap) {
      return;
    }
    let dep = depsMap.get(key);
    if (dep) {
      // 这里将要执行的effect
      const effects = [...dep];
      effects.forEach((effect) => {
        if (effect !== activeEffect) {
          if (effect.scheduler) {
            // 如果有调度函数则执行调度函数
            effect.scheduler();
          } else {
            effect.run();
          }
        }
      });
    }
}
```

8.深度代理

```
 get(target, key, receiver) {
    if (key === ReactiveFlags.IS_REACTIVE) {
        return true;
    }
    // 等会谁来取值就做依赖收集
    const res = Reflect.get(target, key, receiver);
    track(target, 'get', key);

    if(isObject(res)){
        return reactive(res);
    }
    return res;
}
```

> *当取值时返回的值是对象，则返回这个对象的代理对象，从而实现深度代理*