Vue3响应式介绍

- 在Vue2的时候使用defineProperty来进行数据的劫持, 需要对属性进行重写添加 `getter` 及 `setter` **性能差。**
- 当新增属性和删除属性时无法监控变化。需要通过 `$set` 、 `$delete` 实现
- 数组不采用defineProperty来进行劫持 （浪费性能，对所有索引进行劫持会造成性能浪费）需要对数组单独进行处理

> Vue3中使用Proxy来实现响应式数据变化。从而解决了上述问题。

CompositionAPI

- 在Vue2中采用的是OptionsAPI, 用户提供的data,props,methods,computed,watch等属性 (用户编写复杂业务逻辑会出现反复横跳问题)
- Vue2中所有的属性都是通过 `this` 访问， `this` 存在指向明确问题
- Vue2中很多未使用方法或属性依旧会被打包，并且所有全局API都在Vue对象上公开。Composition API对 tree-shaking 更加友好，代码也更容易压缩。
- 组件逻辑共享问题， Vue2 采用mixins 实现组件之间的逻辑共享； 但是会有数据来源不明确，命名冲突等问题。 Vue3采用CompositionAPI 提取公共逻辑非常方便

> 简单的组件仍然可以采用OptionsAPI进行编写，compositionAPI在复杂的逻辑中有着明显的优势~。 `reactivity` 模块中就包含了很多我们经常使用到的 `API` 例如：computed、reactive、ref、effect等

Reactive & Effect

基本使用

```html
<div id="app"></div>
<script src="./reactivity.global.js"></script>
<script>
```

Debug

```
    // const state = shallowReactive({ name: 'jw', age: 30 })
    // const state = readonly({ name: 'jw', age: 30 })
    const state = reactive({ name: 'jw', age: 30})
    effect(() => {
        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~

编写reactive函数

js

```
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)
}
```

Debug

```
}
*/
```

```js
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

```js
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
```

Debug

```
}
```

> 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?

```js
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
            return true;
        }
    }
}
function createReactiveObject(target: object, isReadonly: boolean) {
    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
        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。

编写effect函数

```js
let activeEffect; // 当前正在执行的effect
let effectStack = []; // 存放effect列表
class ReactiveEffect {
    active = true;
    deps = []; // 收集effect中使用到的属性
    constructor(public fn) { }
    run() {
```

Debug

```
        }
        if (!effectStack.includes(this)) { // 防止effect中修改内容导致重复更新
            try {
                effectStack.push(activeEffect = this);
                return this.fn();
            } finally {
                effectStack.pop();
                activeEffect = effectStack[effectStack.length - 1];
            }
        }
    }
}
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
}
```

依赖收集

> 默认执行 `effect` 时会对属性，进行依赖收集

js

```
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;
}
```

js

```
const targetMap = new WeakMap(); // 记录依赖关系
export function track(target, type, key) {
    if(!isTracking()){ // activeEffect !== undefined
        return
```

Debug

```
    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

触发更新

js

```
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;
}
```

js

```
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 deps = [];
```

Debug

```
    }
    const effects = [];
    for(const dep of deps){
        if(dep){
            effects.push(...dep); // 将effect全部存到effects中
        }
    }
    for(const effect of effects){
        if(effect !== activeEffect){ // 如果effect不是当前正在运行的effect
            effect.run(); // 重新执行一遍
        }
    }
}
```

停止effect

js

```
function cleanupEffect(effect){
    const {deps} = effect; // 清理effect
    for(let i = 0 ; i < deps.length;i++){
        deps[i].delete(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);
```

Debug

```
}
```

深度代理

```js
 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

Computed

接受一个 getter 函数，并根据 getter 的返回值返回一个不可变的响应式 ref 对象。

```js
class ComputedRefImpl {
    public effect;
    public _value;
    public dep;
    public _dirty = true;
    constructor(getter,public setter) {
        this.effect = new ReactiveEffect(getter,()=>{
            if(!this._dirty){ // 依赖的值变化更新dirty并触发更新
                this._dirty = true;
```

Debug

```
        });
    }
    get value(){ // 取值的时候进行依赖收集
        if(isTracking()){
            trackEffects(this.dep || (this.dep = new Set));
        }
        if(this._dirty){ // 如果是脏值，执行函数
            this._dirty = false;
            this._value = this.effect.run();
        }
        return this._value;
    }
    set value(newValue){
        this.setter(newValue)
    }
}
export function computed(getterOrOptions) {
    const onlyGetter = isFunction(getterOrOptions); // 传入的是函数就是getter
    let getter;
    let setter;
    if (onlyGetter) {
        getter = getterOrOptions;
        setter = () => { }
    } else {
        getter = getterOrOptions.get;
        setter = getterOrOptions.set;
    }
    // 创建计算属性
    return new ComputedRefImpl(getter, setter)
}
```

> 创建ReactiveEffect时，传入 `scheduler` 函数，稍后依赖的属性变化时调用此方法！

```
export function triggerEffects(dep) { // 触发dep 对应的effect执行
    for (const effect of dep) {
```

```
        }
    }
}
export function trackEffects(dep) { // 收集dep 对应的effect
    let shouldTrack = !dep.has(activeEffect)
    if (shouldTrack) {
        dep.add(activeEffect);
        activeEffect.deps.push(dep);
    }
}
```

deferredComputed

多次同步改值，只触发一次更新。 实现原理类似 `nextTick` 异步更新

js

```
const queue = [];
let queued = false;
const flush = () => { // 刷新队列
    for (let i = 0; i < queue.length; i++) {
        queue[i]()
    }
    queue.length = 0
    queued = false
}
const scheduer = (fn) => {
    queue.push(fn);
    if (!queued) {
        queued = true;
        Promise.resolve().then(flush)
    }
}
class DeferredComputedRefImpl {
    public dep;
    public _dirty = true;
    public _value;
```

Debug

```
        let scheduled = false; // 实现批处理逻辑，多次更新值只执行一次
        this.effect = new ReactiveEffect(getter, () => { // 此函数是scheduler
            if (!scheduled) {
                const valueToCompare = this._value; // 依赖的值变化时 会执行此回调
                scheduled = true;
                scheduer(() => {
                    if (this._get() !== valueToCompare) {
                        triggerEffects(this.dep); // 触发计算属性对应的effect重新执行
                    }
                    scheduled = false;
                });
                for (const e of this.dep) { // 当计算属性依赖计算属性时。需要立即更新dirty
                    if (e.computed) {
                        e.scheduler!()
                    }
                }
            }
            this._dirty = true;
        });
        this.effect.computed = true; // 标记为计算属性
    }
    _get() {
        if (this._dirty) { // 如果是脏值，执行函数
            this._dirty = false;
            this._value = this.effect.run();
        }
        return this._value;
    }
    get value() { // 取值的时候进行依赖收集
        if(isTracking()){
            trackEffects(this.dep || (this.dep = new Set));
        }
        return this._get();
    }
}

export function deferredComputed(getter) {
```

Ref

接受一个内部值并返回一个响应式且可变的 ref 对象。

Ref & ShallowRef

```js
function createRef(rawValue, shallow) {
    return new RefImpl(rawValue,shallow); // 将值进行装包
}
// 将原始类型包装成对象，同时也可以包装对象 进行深层代理
export function ref(value) {
    return createRef(value, false);
}
// 创建浅ref 不会进行深层代理
export function shallowRef(value) {
    return createRef(value, true);
}
```

```js
function toReactive(value) { // 将对象转化为响应式的
    return isObject(value) ? reactive(value) : value
}
class RefImpl {
    public _value;
    public dep;
    public __v_isRef = true;
    constructor(public rawValue, public _shallow) {
        this._value = _shallow ? rawValue : toReactive(rawValue); // 浅ref不需要再次代理
    }
    get value(){
        if(isTracking()){
            trackEffects(this.dep || (this.dep = new Set)); // 收集依赖
```

Debug

```
    }
    set value(newVal){
        if(newVal !== this.rawValue){
            this.rawValue = newVal;
            this._value = this._shallow ? newVal : toReactive(newVal);
            triggerEffects(this.dep);  // 触发更新
        }
    }
}
```

toRef & toRefs

js

```
class ObjectRefImpl {
    public __v_isRef = true
    constructor(public _object, public _key) { }
    get value() {
        return this._object[this._key];
    }
    set value(newVal) {
        this._object[this._key] = newVal;
    }
}
export function toRef(object, key) { // 将响应式对象中的某个属性转化成ref
    return new ObjectRefImpl(object, key);
}
export function toRefs(object) { // 将所有的属性转换成ref
    const ret = Array.isArray(object) ? new Array(object.length) : {};
    for (const key in object) {
        ret[key] = toRef(object, key);
    }
    return ret;
}
```

Debug

toRaw

将被代理的对象转回成原始对象

```js
get(target, key, receiver) {
    if(key === ReactiveFlags.RAW){
        return target
    }
}
export function toRaw(observed){
    const raw = observed && observed[ReactiveFlags.RAW];
    return raw ? toRaw(raw) : observed;
}
```

markRaw

标记对象不可以被代理

```js
function createReactiveObject(target: object, isReadonly: boolean) {
    // ....
    if ( target[ReactiveFlags.SKIP]){
        return target
    }
}
export function markRaw(value) {
    def(value,ReactiveFlags.SKIP,true);
    return value;
}
```

EffectScope

effectScope是一个函数，调用effectScope函数会返回一个对象，其中包含了run 和stop；在run中定义的所有effect函数，在调用了scope对象的stop()方法之后，所有的依赖都被停止了。

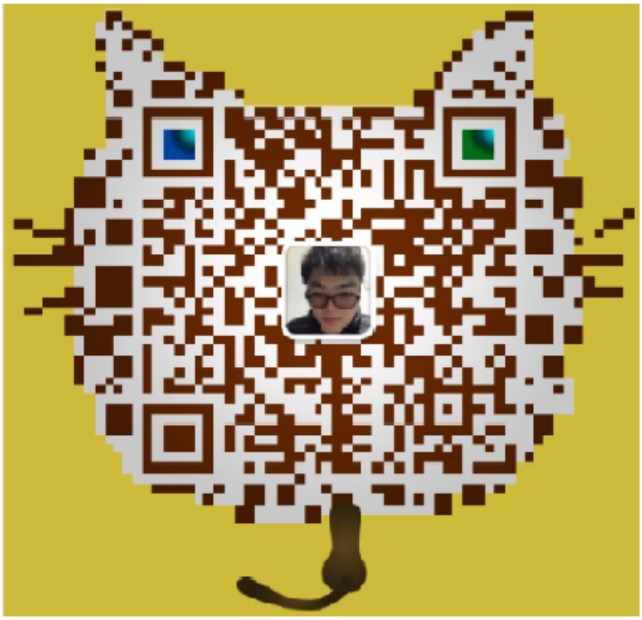
```
let activeEffectScope;
class EffectScope {
    active = true;
    effects = []
    run(fn) {
        if (this.active) {
            try {
                this.on(); // 运行的时候 标记当前正在运行的effect
                return fn();
            } finally {
                this.off();
            }
        }
    }
    on() {
        if (this.active) {
            effectScopeStack.push(this);
            activeEffectScope = this;
        }
    }
    off() {
        if (this.active) {
            effectScopeStack.pop();
            activeEffectScope = effectScopeStack[effectScopeStack.length - 1];

        }
    }
    stop(){
        if(this.active){ // 停止时依次调用stop
            this.effects.forEach(e=>e.stop());
            this.active = false
        }
    }
}
export function effectScope() {
    return new EffectScope();
}
```

```
    if(activeEffectScope.active){
        activeEffectScope.effects.push(effect);
    }
}
```

客服老师微信

我的微信



最后更新时间: 11/22/2021, 10:13:11 PM

← Vue3开发环境搭建