3.2 Vue3.0响应式原理

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

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

```
e=1.0">
  <title>Document</title>
</head>
<body>
  <script>
    'use strict'
    // 问题1: set 和 deleteProperty 中需要返回布尔类型的值
    // 在严格模式下，如果返回 false 的话会出现 Type Error 的异常

    // 问题2: Proxy 和 Reflect 中使用的 receiver

    // Proxy 中 receiver: Proxy 或者继承 Proxy 的对象
    // Reflect 中 receiver: 如果 target 对象中设置了 getter, getter 中的 this 指向 receiver
    const target = {
      foo: 'xxx',
      bar: 'yyy'
    }
    const proxy = new Proxy(target,{
      get(target,key, receiver) {
        return Reflect.get(target,key,receiver)
      },
      set(target, key,value,receiver) {
        return Reflect.set(target,key,value,receiver)
      },
      deleteProperty(target,key) {
        return Reflect.deleteProperty(target,key)
      }
    })
    proxy.foo = 'zzzz'
    delete proxy.foo
  </script>
</body>
</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

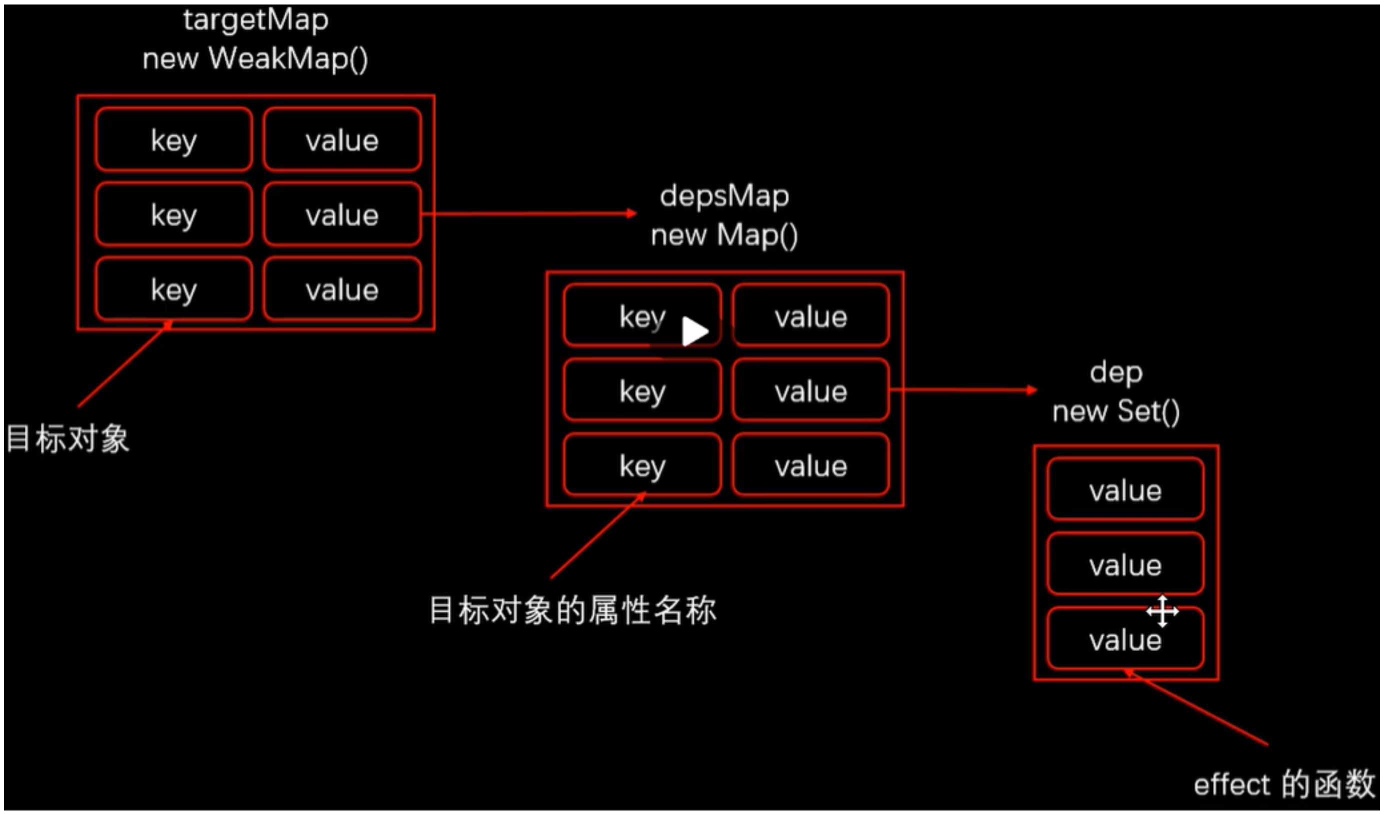
- 返回proxy对象

```javascript
const isObject = val => val !== null && typeof val === 'object'
const convert = target => isObject(target) ? reactive(target) : target
const hasOwnProperty = Object.prototype.hasOwnProperty
const hasOwn = (target, key) => hasOwnProperty.call(target, key)

export function reactive(target) {
  // 不是对象没有响应式可言,直接返回值
  if(!isObject(target)) return target
  const handler = {
    get(target,key,receiver) {
      const result = Reflect.get(target, key, receiver)
      console.log(result)
      return convert(result)
    },
    set(target,key,value,receiver) {
      const oldValue = Reflect.get(target, key, receiver)
      let result = true
      console.log('set')
      if(oldValue !== value) {
        result = Reflect.set(target, key, value, receiver)
      }
      return result
    },
    deleteProperty(target,key) {
      console.log('del')
      const hadKey = hasOwn(target, key)
      const result = Reflect.deleteProperty(target, key)
      if(hadKey && result) {

      }
      return result
    }
  }
  return new Proxy(target,handler)
}
```

4.收集依赖 effect&track



```
const isObject = val => val !== null && typeof val === 'object'
const convert = target => isObject(target) ? reactive(target) : target
const hasOwnProperty = Object.prototype.hasOwnProperty
const hasOwn = (target, key) => hasOwnProperty.call(target, key)

export function reactive(target) {
  // 不是对象没有响应式可言,直接返回值
  if(!isObject(target)) return target
  const handler = {
    get(target,key,receiver) {
      track(target, key)
      const result = Reflect.get(target, key, receiver)
      // 收集依赖
      console.log(result)
      return convert(result)
    },
    set(target,key,value,receiver) {
      const oldValue = Reflect.get(target, key, receiver)
      let result = true
      console.log('set')
      if(oldValue !== value) {
        result = Reflect.set(target, key, value, receiver)
      }
      return result
    },
    deleteProperty(target,key) {
      console.log('del')
      const hadKey = hasOwn(target, key)
      const result = Reflect.deleteProperty(target, key)
      if(hadKey && result) {

      }
      return result
    }
  }
  return new Proxy(target,handler)
}
let activeEffect = null
export function effect(callback) {
  activeEffect = callback
  callback() // 访问响应式对象属性，去收集依赖
  activeEffect = null
}
// 收集依赖
let targetMap = new WeakMap()
export function track(target , key) {
  if(!activeEffect) return
  let depsMap = targetMap.get(target)
  if(!depsMap) {
    targetMap.set(target, (depsMap = new Map()))
  }
  let dep = depsMap.get(key)
  if(!dep) {
    depsMap.set(key,(dep = new Set()))
  }
  dep.add(activeEffect)
}
```

5.触发更新trigger

```
const isObject = val => val !== null && typeof val === 'object'
const convert = target => isObject(target) ? reactive(target) : target
const hasOwnProperty = Object.prototype.hasOwnProperty
const hasOwn = (target, key) => hasOwnProperty.call(target, key)

export function reactive(target) {
  // 不是对象没有响应式可言,直接返回值
  if(!isObject(target)) return target
  const handler = {
    get(target,key,receiver) {
      track(target, key)
      const result = Reflect.get(target, key, receiver)
      // 收集依赖
      console.log(result)
      return convert(result)
    },
    set(target,key,value,receiver) {
      const oldValue = Reflect.get(target, key, receiver)
      let result = true
      console.log('set')
      if(oldValue !== value) {
        result = Reflect.set(target, key, value, receiver)
        // 触发更新
        trigger(target, key)
      }
      return result
    },
    deleteProperty(target,key) {
      console.log('del')
      const hadKey = hasOwn(target, key)
      const result = Reflect.deleteProperty(target, key)
      if(hadKey && result) {
        // 触发更新
        trigger(target, key)
      }
      return result
    }
  }
  return new Proxy(target,handler)
}
let activeEffect = null
export function effect(callback) {
  activeEffect = callback
  callback() // 访问响应式对象属性，去收集依赖
  activeEffect = null
}
// 收集依赖
let targetMap = new WeakMap()
export function track(target , key) {
  if(!activeEffect) return
  let depsMap = targetMap.get(target)
  if(!depsMap) {
    targetMap.set(target, (depsMap = new Map()))
  }
  let dep = depsMap.get(key)
  if(!dep) {
    depsMap.set(key,(dep = new Set()))
  }
  dep.add(activeEffect)
}
// 触发更新
export function trigger (target,key) {
  const depsMap = targetMap.get(target)
  if(!depsMap) return
  const dep = depsMap.get(key)
  if(dep) {
    dep.forEach(effect => {
      effect()
    });
  }
}
```

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

```
export function ref(raw) {
    // 判断 raw 是否是ref 创建的对象, 如果是的话直接返回
    if (isObject(raw) && raw.__v_isRef) {
      return
    }
    let value = convert(raw)
    const r = {
      __v_isRef: true,
      get value() {
        track(r,'value')
        return value
      },
      set value(newValue) {
        if(newValue !== value) {
          raw = newValue
          value = convert(raw)
          trigger(r,'value')
        }
      }
    }
    return r
}
```

7.toRefs

```
export function toRefs(proxy) {
  const ret = proxy instanceof Array ? new Array(proxy.length) :
{}
  for (const key in proxy) {
      ret[key] = toProxyRef(proxy,key)
  }
}
```

```
function toProxyRef (proxy, key) {
  const r = {
    __v_isRef: true,
    get value () {
      return proxy[key]
    },
    set value (newValue) {
      proxy[key] = newValue
    }
  }
  return r
}
```

8.computed

```
export function computed(getter) {
  const result = ref()
  effect(() => (result.value = getter()))
  return result;
}
```