Vue3响应式原理剖析

课堂目标

- 体验vue3响应式
- vue2响应式原理回顾
- vue3响应式原理实现
- 深入源码

体验vue3响应式

- 迁出Vue3源码：`git clone https://github.com/vuejs/vue-next.git`
- 安装依赖：`yarn`
- 编译：`yarn dev`

> 生成结果：packages\vue\dist\vue.global.js

- vue3响应式初体验

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>hello vue3</title>
    <script src="../dist/vue.global.js"></script>
</head>

<body>
    <div id='app'></div>
    <script>
        const { createApp, reactive} = Vue;

        // 声明组件
        const App = {
            template: `
                <div>count: {{ count }}</div>
            `,
            setup() {
                const state = reactive({ count: 0 })
                setInterval(() => {
                  state.count++
                }, 1000);
                return state
            }
        }

        createApp().mount(App, '#app')
    </script>
</body>
```

```
</html>
```

vue2响应式原理回顾

```
// Vue 2.0响应式原理
// 1.对象响应化：遍历每个key，定义getter、setter
// 2.数组响应化：覆盖数组原型方法，额外增加通知逻辑
const originalProto = Array.prototype
const arrayProto = Object.create(originalProto)
;['push', 'pop', 'shift', 'unshift', 'splice', 'reverse', 'sort'].forEach(
  method => {
    arrayProto[method] = function() {
      originalProto[method].apply(this, arguments)
      notifyUpdate()
    }
  }
)

function observe(obj) {
  if (typeof obj !== 'object' || obj == null) {
    return
  }
  if (Array.isArray(obj)) {
    Object.setPrototypeOf(obj, arrayProto)
  } else {
    const keys = Object.keys(obj)
    for (let i = 0; i < keys.length; i++) {
      const key = keys[i]
      defineReactive(obj, key, obj[key])
    }
  }
}

function defineReactive(obj, key, val) {
  observe(val)

  Object.defineProperty(obj, key, {
    get() {
      return val
    },
    set(newVal) {
      if (newVal !== val) {
        observe(newVal)
        notifyUpdate()
        val = newVal
      }
    }
  })
}

function notifyUpdate() {
  console.log('页面更新!')
}
```

vue3响应式实现

vue3使用ES6的Proxy特性来实现响应式。

```
const toProxy = new WeakMap() // 形如obj:observed
const toRaw = new WeakMap() // 形如observed:obj

function isObject(obj) {
  return typeof obj === 'object' || obj === null
}
function hasOwn(obj, key) {
  return obj.hasOwnProperty(key)
}

function reactive(obj) {
  //判断是否对象
  if (!isObject(obj)) {
    return obj
  }
  // 查找缓存，避免重复代理
  if (toProxy.has(obj)) {
    return toProxy.get(obj)
  }
  if (toRaw.has(obj)) {
    return obj
  }
  // Proxy相当于在对象外层加拦截
  // http://es6.ruanyifeng.com/#docs/proxy
  const observed = new Proxy(obj, {
    get(target, key, receiver) {
      // Reflect用于执行对象默认操作，更规范、更友好
      // Proxy和Object的方法Reflect都有对应
      // http://es6.ruanyifeng.com/#docs/reflect
      const res = Reflect.get(target, key, receiver)
      // console.log(`获取${key}:${res}`)
      track(target, key)
      return isObject(res) ? reactive(res) : res
    },
    set(target, key, value, receiver) {
      const hadKey = hasOwn(target, key)
      const oldVal = target[key]
      // console.log(`${key} oldVal:${oldVal}`)
      // console.log(`${key} newVal:${value}`)
      const res = Reflect.set(target, key, value, receiver)

      // console.log(`设置${key}:${value}`)

      if (!hadKey) {
        console.log(`新增${key}:${value}`)
        trigger(target, 'ADD', key)
      } else if (oldVal !== value) {
        console.log(`设置${key}:${value}`)
        trigger(target, 'SET', key)
      }
      return res
    },
    deleteProperty(target, key) {
      const hadKey = hasOwn(target, key)
      const res = Reflect.deleteProperty(target, key)
      if (res && hadKey) {
        console.log(`删除${key}:${res}`)
        trigger(target, 'DELETE', key)
      }

      return res
    }
  })

  toProxy.set(obj, observed)
  toRaw.set(observed, obj)

  return observed
}

// 依赖收集：建立target.key和响应函数之间对应关系
const activeReactiveEffectStack = []
// 映射关系表，结构大致如下：
// {target: {key: [fn1,fn2]}}
const targetsMap = new WeakMap()
function track(target, key) {
  // 从栈中取出响应函数
  const effect = activeReactiveEffectStack[activeReactiveEffectStack.length - 1]
  if (effect) {
    // 获取target对应依赖表
    let depsMap = targetsMap.get(target)
    if (!depsMap) {
      // 首次访问不存在需创建
      depsMap = new Map()
      targetsMap.set(target, depsMap)
    }
    // 获取key对应的响应函数集
    let deps = depsMap.get(key)
    if (!deps) {
      deps = new Set()
      depsMap.set(key, deps)
    }
    // 将响应函数加入到对应集合
    if (!deps.has(effect)) {
      deps.add(effect)
    }
  }
}

// 触发target.key对应响应函数
function trigger(target, type, key) {
  // 获取依赖表
  const depsMap = targetsMap.get(target)
  if (depsMap) {
    // 获取响应函数集合
    const deps = depsMap.get(key)
    const effects = new Set()
    if (deps) {
        // 执行所有响应函数
        deps.forEach(effect => {
          // effect()
          effects.add(effect)
        })
      }

      // 数组新增或删除
      if (type === 'ADD' || type === 'DELETE') {
        if (Array.isArray(target)) {
          const deps = depsMap.get('length')
          if (deps) {
            deps.forEach(effect => {
              effects.add(effect)
            })
          }
        }
      }
      // 获取已存在的Dep Set执行
      effects.forEach(effect => effect())
    }
  }

// effect任务：执行fn并将其入栈
function effect(fn) {
  const rxEffect = function(...args) {
    return run(rxEffect, fn, args)
  }
  // 默认执行一次响应函数
  rxEffect()
  // 返回响应函数
  return rxEffect
}

function run(effect, fn, args) {
  try {
    activeReactiveEffectStack.push(effect)
    return fn(...args) //执行fn以收集依赖
  } finally {
    activeReactiveEffectStack.pop()
  }
}
```

相关源码结构

查看package/reactivity/src