vue3源码剖析02

学习目标

- composition-api体验
- Vue3响应式源码学习
- 响应式原理：Vue2 vs Vue3
- 造轮子之旅

composition-api

文档

https://vue-composition-api-rfc.netlify.com

初体验

```
<div id="app">
  <h1>composition-api</h1>
  <p @click="add">{{state.counter}}</p>
  <p>{{state.doubleCounter}}</p>
</div>

<script src="../dist/vue.global.js"></script>
<script>
  const { createApp, reactive, computed } = Vue

  const app = createApp({
    setup() {
      const state = reactive({
        counter: 0,
        doubleCouter: computed(() => counter * 2)
      })
```

```
      const add = () => {
        data.counter++
      }

      return { state, add }
    },
  }).mount('#app')
</script>
```

更好的逻辑复用和代码组织

```
<meta charset="UTF-8">
<script src="../dist/vue.global.js"></script>

<div id="app">
  <h1>logic reuse</h1>
</div>

<script>
  const { createApp, reactive, onMounted, onUnmounted, toRefs } = Vue;

  // 鼠标位置侦听
  function useMouse() {
    // 数据响应化
    const state = reactive({ x: 0, y: 0 })
    const update = e => {
      state.x = e.pageX
      state.y = e.pageY
    }
    onMounted(() => {
      window.addEventListener('mousemove', update)
    })
    onUnmounted(() => {
      window.removeEventListener('mousemove', update)
    })
    // 转换所有key为响应式数据
    return toRefs(state)
  }
  // 事件监测
  function useTime() {
    const state = reactive({ time: new Date() })
    onMounted(() => {
      setInterval(() => {
        state.time = new Date()
      }, 1000)
    })
    return toRefs(state)
  }
```

```
  // 逻辑组合
  const MyComp = {
    template: `
      <div>x: {{ x }} y: {{ y }}</div>
      <p>time: {{time}}</p>
    `,
    setup() {
      // 使用鼠标逻辑
      const { x, y } = useMouse()
      // 使用时间逻辑
      const { time } = useTime()
      // 返回使用
      return { x, y, time }
    }
  }
  createApp(MyComp).mount('#app')
</script>
```

对比mixins，好处显而易见：

- x,y,time来源清晰
- 不会与data、props等命名冲突

更好的维护性

更好的类型推断

Vue最初选项API中存在大量this上下文，对TypeScript类型推断很不友好。在composition-api中仅利用纯变量和函数，规避了对this的使用，自然的拥有良好的类型推断能力。

Vue3中响应式源码学习

测试代码

```
<div id="app">
  {{foo}}
</div>

<script src="../dist/vue.global.js"></script>
<script>
  const { createApp } = Vue
  createApp({
    data() {
      return {
        foo: 'foo'
      }
    }
  }).mount('#app')
</script>
```

整体流程

applyOptions中对data选项做**响应式处理使用的是reactive函数**

setupRenderEffect函数中**使用effect函数做依赖收集**

```
const setupRenderEffect: SetupRenderEffectFn = (
  instance,  instance = {uid: 0, vnode: {…}, type: {…},
  initialVNode,  initialVNode = {__v_isVNode: true, __v
  container,  container = div#app {align: "", title: ""
  anchor,  anchor = null
  parentSuspense,  parentSuspense = null
  isSVG,  isSVG = false
  optimized  optimized = false
) => {
  // create reactive effect for rendering
  instance.update = effect(function componentEffect() {
    if (!instance.isMounted) {
      let vnodeHook: VNodeHook | null | undefined
      const { el, props } = initialVNode
      const { bm, m, parent } = instance
```

响应式原理：vue2 vs vue3

数据变化可侦测，从而对使用数据的地方进行更新。

vue2的方式

Object.defineProperty()

```
// 拦截每个key，从而可以侦测数据变化
function defineReactive(obj, key, val) {
  Object.defineProperty(obj, key, {
    get() {
      return val
    },
    set(v) {
      val = v
      update()
    }
  })
}

function update() {
  console.log(obj.foo);
}
const obj = {}
defineReactive(obj, 'foo', 'foo')

obj.foo = 'fooooooo'
```

vue3的方式

Proxy

```
// 代理整个对象，从而侦测数据变化
function defineReactive(obj) {
  return new Proxy(obj, {
    get(target, key) {
      return target[key]
    },
    set(target, key, val) {
      target[key] = val
      update()
    }
  })
}

function update() {
  console.log(obj.foo);
}

const obj = {}
const observed = defineReactive(obj)

observed.foo = 'fooooooo'
```

Vue2 vs Vue3

vue2中需要递归遍历对象所有key，速度慢

```
// 1.对象响应化：遍历每个key，定义getter、setter
function observe(obj) {
  if (typeof obj !== 'object' || obj == null) {
    return
  }

  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)
        val = newVal
        dep.notify()
      }
    }
  })
}
```

数组响应式需要额外实现

```
// 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)
      dep.notify()
    }
  }
)
```

新增或删除属性无法监听，需要使用特殊api

```
Vue.set(obj, 'foo', 'bar')
Vue.delete(obj, 'foo')
```

不支持Map、Set、Class等数据结构

vue3实现

vue3中利用es6 proxy实现数据响应式，很好的解决了以上问题。

```
function reactive(obj) {
  if (typeof obj !== 'object' && obj != null) {
    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}`)
      return res
    },
    set(target, key, value, receiver) {
      const res = Reflect.set(target, key, value, receiver)
      console.log(`设置${key}:${value}`)
      return res
    },
    deleteProperty(target, key) {
      const res = Reflect.deleteProperty(target, key)
      console.log(`删除${key}:${res}`)
      return res
    }
  })
  return observed
}

// 测试
const state = reactive({ foo: 'foo' })
// 获取
state.foo // ok
// 设置已存在属性
state.foo = 'fooooooo' // ok
// 设置不存在属性
state.dong = 'dong' // ok
// 删除属性
delete state.dong // ok
```

造个轮子

首先实现reactive(obj)，借助Proxy代理传入的obj，这样可以拦截对obj的各种访问。

```
const baseHandler = {
  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}`)
```

```
    return res
  },
  set(target, key, value, receiver) {
    const res = Reflect.set(target, key, value, receiver)
    console.log(`设置${key}:${value}`)
    return res
  },
  deleteProperty(target, key) {
    const res = Reflect.deleteProperty(target, key)
    console.log(`删除${key}:${res}`)
    return res
  }
}

function reactive(obj) {
  if (typeof obj !== 'object' && obj != null) {
    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, baseHandler)
  return observed
}

// 测试
const state = reactive({ foo: 'foo' })

// 获取
state.foo // ok
// 设置已存在属性
state.foo = 'fooooooo' // ok
// 设置不存在属性
state.dong = 'dong' // ok
// 删除属性
delete state.dong // ok
```

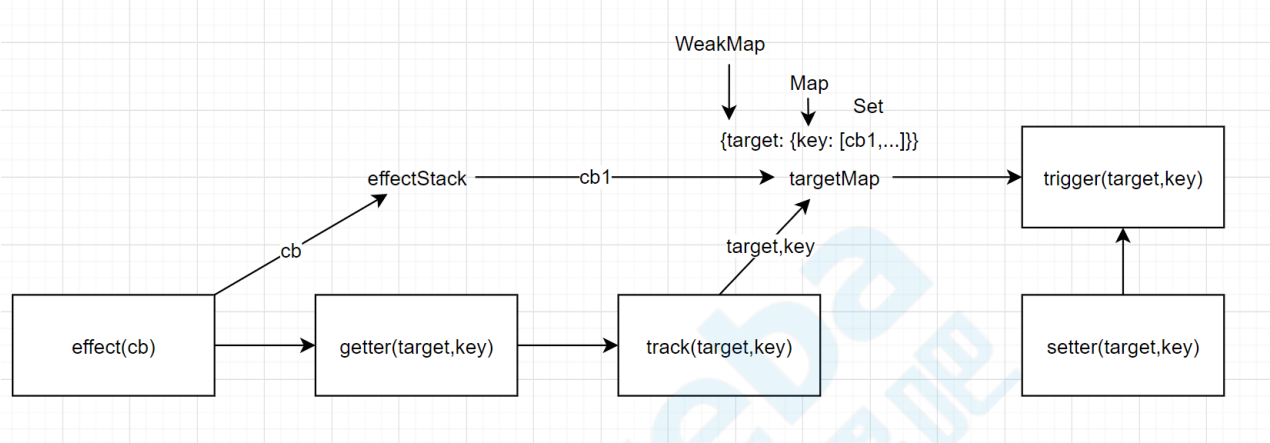
上面代码对嵌套对象`reactive({ bar: {baz:1} })`没有做处理，需要递归处理

```
// 传入对象应该是一个非null的object
const isObject = v => typeof v === 'object' && v !== null

const baseHandler = {
  get(target, key, receiver) {
    const res = Reflect.get(target, key, receiver)
    // 判断res是对象，递归处理它
    return isObject(res) ? reactive(res) : res
  },
```

```
}
function reactive(obj) {
  // reactive()只接受非null的object
  if (!isObject(obj)) {
    return obj
  }
}
```

下面是依赖收集的实现，原理如下图：



相关api有

- effect(fn)：传入fn，返回的函数将是响应式的，内部代理的数据发生变化，它会再次执行
- track(target, key)：建立响应式函数与其访问的目标(target)和键(key)之间的映射关系
- trigger(target, key)：根据track()建立的映射关系，找到对应响应式函数并执行它

基本结构：

```
// 临时存储响应式函数
const effectStack = []

// 将传入fn转换为一个响应式函数
function effect(fn, options = {}) {}

// 存放响应式函数和目标、键之间的映射关系
const targetMap = new WeakMap()

// 依赖收集，创建映射关系
function track(target, key) {}

// 根据映射关系获取响应函数
function trigger(target, key) {}
```

实现effect()/track()/trigger()

```
function effect(fn, options = {}) {
  // 创建reactiveEffect
  const e = createReactiveEffect(fn, options)

  // 执行一次触发依赖收集
  e()

  return e
}

function createReactiveEffect(fn, options) {
  // 封装一个高阶函数，除了执行fn，还要将自己放入effectStack为依赖收集做准备
  const effect = function reactiveEffect(...args) {
    if (!effectStack.includes(effect)) {
      try {
        // 1.effect入栈
        effectStack.push(effect)
        // 2.执行fn
        return fn(...args)
      } finally {
        // 3.effect出栈
        effectStack.pop()
      }
    }
  }
  return effect
}

function track(target, key) {
  // 获取响应式函数
  const effect = effectStack[effectStack.length - 1]
  if (effect) {
    // 获取target映射关系map，不存在则创建
    let depMap = targetMap.get(target)
    if (!depMap) {
      depMap = new Map()
      targetMap.set(target, depMap)
    }

    // 获取key对应依赖集合，不存在则创建
    let deps = depMap.get(key)
    if (!deps) {
      deps = new Set()
      depMap.set(key, deps)
    }

    // 将响应函数添加到依赖集合
```

```
    deps.add(effect)
  }
}

function trigger(target, key) {
  // 获取target对应依赖map
  const depMap = targetMap.get(target)
  if (!depMap) {
    return
  }
  // 获取key对应集合
  const deps = depMap.get(key)
  if (deps) {
    // 执行所有响应函数
    deps.forEach(dep => dep())
  }
}

// 测试
const state = reactive({ foo: 'foo' })

effect(() => {
  console.log('effect', state.foo);
})
```

结合视图验证一下

```
<div id="app"></div>

<script src="03-reactivity.js"></script>
<script>
  const obj = {
    name: 'kkb',
    age: 8
  }

  const data = reactive(obj)

  // effect()定义我们的更新函数
  effect(() => {
    app.innerHTML = `
      <h1>${data.name}今年${data.age}岁了</h1>
    `
  })

  // 修改一下数值
  setInterval(() => {
```

```
    data.age++
  }, 1000);
</script>
```

> 计算属性也很常用：
>
> ```
> const double = computed(() => data.age * 2)
>
> // effect()定义我们的更新函数
> effect(() => {
> app.innerHTML = `
> <h1>${data.name}今年${data.age}岁了</h1>
> 乘以2是${double.value}岁
> `
> })
> ```
>
> 计算属性可以基于effect实现

computed(fn)：可以使传入fn使之成为响应式函数，fn内部依赖的数值发生变化，该函数应该重新执行获得最新的计算结果。

```
function computed(fn) {
  // 创建一个特殊的effect：
  // 这个effect创建时不会立刻执行，且会在其他effect后面执行
  const runner = effect(fn, { computed: true, lazy: true })
  // 返回一个对象包含响应函数和最新值的getter
  // 这样computed首次获取值时才收集依赖
  return {
    effect: runner,
    get value() {
      return runner()
    }
  }
}

function trigger(target, key) {
  // ...
  if (deps) {
    // 将普通effect和computed区分开
    const effects = new Set()
    const computedRunners = new Set()
    deps.forEach(dep => {
      if (dep.computed) {
        computedRunners.add(dep)
      } else {
        effects.add(dep)
      }
```

```
    })
    computedRunners.forEach(computed => computed())
    effects.forEach(effect => effect())
  }
}
```

作业

按课上讲解手写vue3响应式相关API: reactive、effect、trigger、track等
要求：学习中心提交代码截图和代码
通过标准：能够正常运转，完成既定功能