vue3初探 + 响应式原理

知识点

- 源码结构
- 调试环境搭建
- vue3初探
- Composition API
- 响应式原理剖析
- vue3展望

调试环境搭建

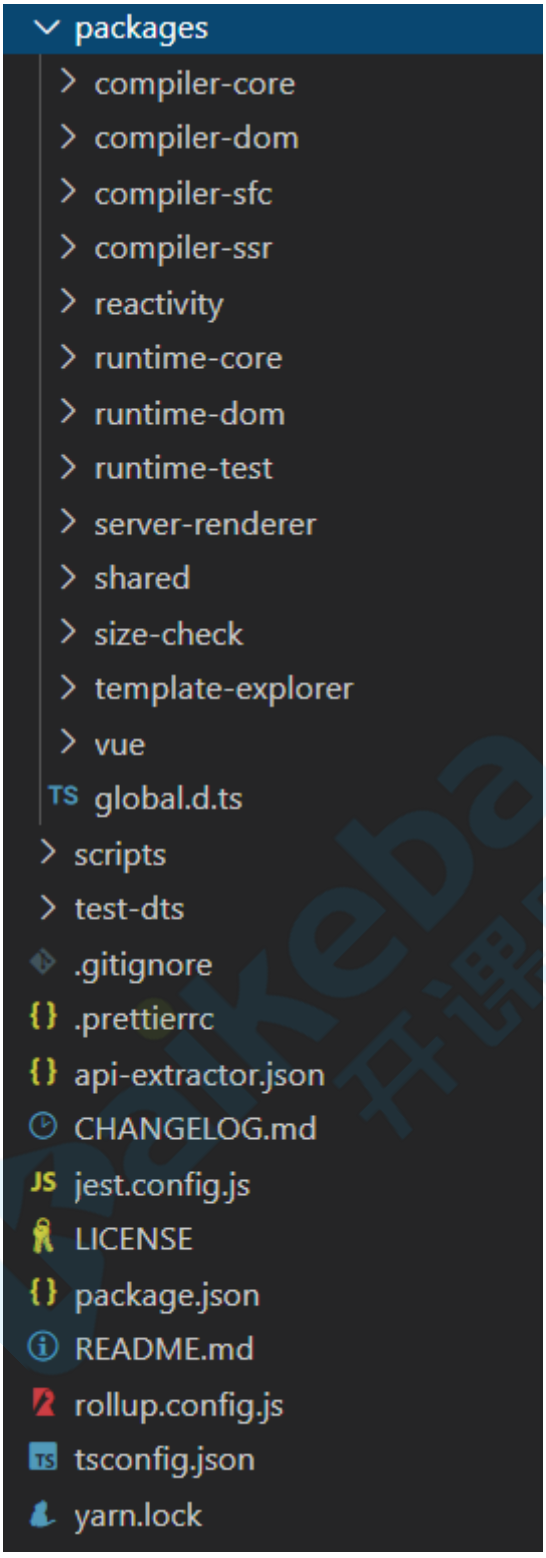
- 迁出Vue3源码：`git clone https://github.com/vuejs/vue-next.git`
- 安装依赖：`yarn`
- 生成sourcemap文件，package.json

```
"dev": "node scripts/dev.js --sourcemap"
```

- 编译：`yarn dev` / `npm run dev`

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

源码结构



源码位置是在package文件件内，实际上源码主要分为两部分，编译器和运行时环境。

- 编译器
  - compiler-core 核心编译逻辑
  - compiler-dom 针对浏览器平台编译逻辑
  - compiler-sfc 针对单文件组件编译逻辑
  - compiler-ssr 针对服务端渲染编译逻辑
- 运行时环境
  - runtime-core 运行时核心
  - runtime-dom 运行时针对浏览器的逻辑
  - runtime-test 浏览器外完成测试环境仿真

- reactivity 响应式逻辑
- template-explorer 模板浏览器
- vue 代码入口，整合编译器和运行时
- server-renderer 服务器端渲染
- share 公用方法

Vue 3初探

测试代码，~/packages/samples/01-hello-vue3.html

```
<!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'><h1>{{message}}</h1></div>
    <script>
        Vue.createApp({
          data: { message: 'Hello Vue 3!' }
        }).mount('#app')
    </script>
</body>
</html>
```

Composition API

Composition API字面意思是组合API，它是为了实现**基于函数的逻辑复用机制**而产生的。

基本使用

数据响应式，创建02-composition-api.html

```
<!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>Document</title>
  <script src="../dist/vue.global.js"></script>
</head>

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

```
    <h1>Composition API</h1>
    <div>count: {{ state.count }}</div>
  </div>

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

    // 声明组件
    const App = {
      // setup是一个新的组件选项，它是组件内使用Composition API的入口
      // 调用时刻是初始化属性确定后，beforeCreate之前
      setup() {
        // 响应化：接收一个对象，返回一个响应式的代理对象
        const state = reactive({ count: 0 })

        // 返回对象将和渲染函数上下文合并
        return { state }
      }
    }

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

</html>
```

计算属性

```
<div>doubleCount: {{doubleCount}}</div>
```

```
const { computed } = Vue;

const App = {
    setup() {
        const state = reactive({
            count: 0,
            // computed()返回一个不可变的响应式引用对象
            // 它封装了getter的返回值
            doubleCount: computed(() => state.count * 2)
        })
    }
}
```

事件处理

```
<div @click="add">count: {{ state.count }}</div>
```

```
const App = {
    setup() {
        // setup中声明一个add函数
        function add() {
            state.count++
        }

        // 传入渲染函数上下文
        return { state, add }
    }
}
```

侦听器

```
const { watch } = Vue;

const App = {
    setup() {
        // state.count变化cb会执行
        // 常用方式还有watch(()=>state.count, cb)
        watch(() => {
          console.log('count变了:' + state.count);
        })
    }
}
```

引用对象：单个原始值响应化

```
<div>counter: {{ counter }}</div>
```

```
const { ref } = Vue;
const App = {
    setup() {
        // 返回响应式的Ref对象
        const counter = ref(1)
        setTimeout(() => {
          // 要修改对象的value
          counter.value++
        }, 1000);
        // 添加counter
        return { state, add, counter }
    }
}
```

体验逻辑组合

03-logic-composition.html

```
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().mount(MyComp, '#app')
```

将对象展开，其内所有key都变成响应式对象

> 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

Vue3响应式原理

Vue2响应式原理回顾

```
// 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
  }
  // 增加数组类型判断，若是数组则覆盖其原型
  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) // 解决嵌套对象问题

  Object.defineProperty(obj, key, {
    get() {
      return val
    },
    set(newVal) {
      if (newVal !== val) {
        observe(newVal) // 新值是对象的情况
        val = newVal
        notifyUpdate()
      }
    }
  })
}

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

嵌套对象的情况下，需要再判断一次该嵌套是数组还是对象

在上边的observe中已经排除了是数组的可能性

> vue2响应式弊端：
>
> - 响应化过程需要递归遍历，消耗较大
> - 新加或删除属性无法监听
> - 数组响应化需要额外实现
> - Map、Set、Class等无法响应式
> - 修改语法有限制

Vue3响应式原理剖析

vue3使用ES6的Proxy特性来解决这些问题。

创建04-reactivity.js

```
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
}
```

测试代码

```
const state = reactive({
    foo: 'foo',
    bar: { a: 1 }
})
// 1.获取
state.foo // ok
// 2.设置已存在属性
state.foo = 'fooooooo' // ok
// 3.设置不存在属性
state.dong = 'dong' // ok
// 4.删除属性
delete state.dong // ok
```

嵌套对象响应式

测试：嵌套对象不能响应

```
// 4.设置嵌套对象属性
react.bar.a = 10 // no ok
```

添加对象类型递归

```
// 提取帮助方法
const isObject = val => val !== null && typeof val === 'object'

function reactive(obj) {
  //判断是否对象
  if (!isObject(obj)) {
    return obj
  }
  const observed = new Proxy(obj, {
    get(target, key, receiver) {
      // ...
      // 如果是对象需要递归
      return isObject(res) ? reactive(res) : res
    },
    //...
}
```

避免重复代理

重复代理，比如

```
reactive(data) // 已代理过的纯对象
reactive(react) // 代理对象
```

解决方式：将之前代理结果缓存，get时直接使用

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

function reactive(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
  }
  const observed = new Proxy(...)

  // 缓存代理结果
  toProxy.set(obj, observed)
  toRaw.set(observed, obj)
  return observed
}

// 测试效果
console.log(reactive(data) === state)
console.log(reactive(state) === state)
```

依赖收集

建立响应数据key和更新函数之间的对应关系。

用法

```
// 设置响应函数
effect(() => console.log(state.foo))

// 用户修改关联数据会触发响应函数
state.foo = 'xxx'
```

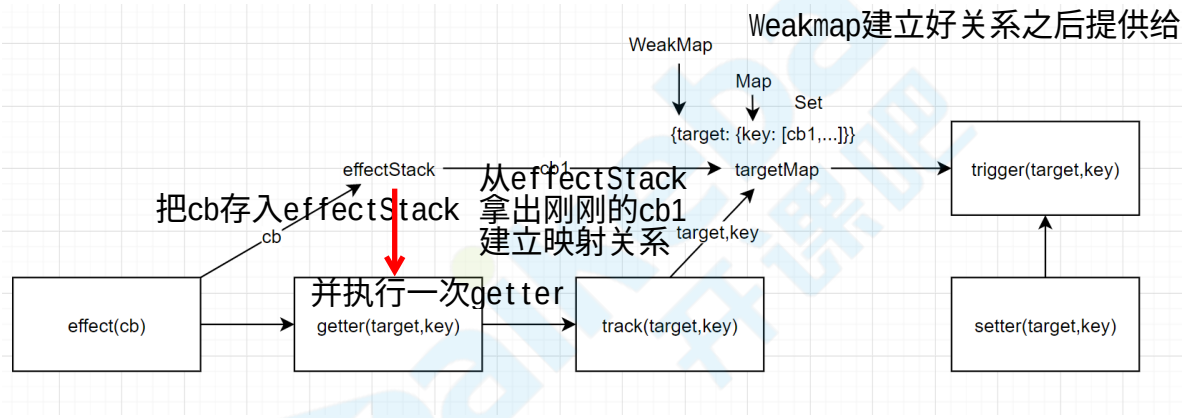
设计

实现三个函数：

effect：将回调函数保存起来备用，立即执行一次回调函数触发它里面一些响应数据的getter

track：getter中调用track，把前面存储的回调函数和当前target,key之间建立映射关系

trigger：setter中调用trigger，把target,key对应的响应函数都执行一遍



target,key和响应函数映射关系

```
// 大概结构如下所示
//    target | depsMap
//      obj  |   key   |  Dep
//               k1    |  effect1,effect2...
//               k2    |  effect3,effect4...

// {target: {key: [effect1,...]}}
```

实现

设置响应函数，创建effect函数

```
// 保存当前活动响应函数作为getter和effect之间桥梁
const effectStack = []

// effect任务：执行fn并将其入栈
function effect(fn) {
  const rxEffect = function() {
    // 1.捕获可能的异常
    try {
```

```
      // 2.入栈，用于后续依赖收集
      effectStack.push(rxEffect)
      // 3.运行fn，触发依赖收集
      return fn()
    } finally {
      // 4.执行结束，出栈
      effectStack.pop()
    }
  }
  // 默认执行一次响应函数
  rxEffect()
  // 返回响应函数
  return rxEffect
}
```

依赖收集和触发

```
function reactive(obj) {
  // ...
  const observed = new Proxy(obj, {
    get(target, key, receiver) {
      // ...
      // 依赖收集
      track(target, key)
      return isObject(res) ? reactive(res) : res
    },
    set(target, key, value, receiver) {
      // ...
      // 触发响应函数
      trigger(target, key)
      return res
    }
  })
}

// 映射关系表，结构大致如下：
// {target: {key: [fn1,fn2]}}
let targetMap = new WeakMap()
function track(target, key) {
  // 从栈中取出响应函数
  const effect = effectStack[effectStack.length - 1]
  if (effect) {
    // 获取target对应依赖表
    let depsMap = targetMap.get(target)
    if (!depsMap) {
      depsMap = new Map()
      targetMap.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, key) {
  // 获取依赖表
  const depsMap = targetMap.get(target)
  if (depsMap) {
    // 获取响应函数集合
    const deps = depsMap.get(key)
    if (deps) {
      // 执行所有响应函数
      deps.forEach(effect => {
        effect()
      })
    }
  }
}
```

vue3展望

vue3适合我吗？会迅速取代vue2吗？我从以下点出发给出个人看法

- 升级是否平滑？

  Vue3会兼容之前写法，仅Portal、Suspense等少量新api需要看看，Composition API则是可选的

- 相关生态是否跟上？

  正式版发布还有一段时间，相关工具、生态、库都跟上需要时间，vue3也许明年不会有大需求

- vue3比vue2好吗？
  - 杀手级特性：Composition API
  - 用户体验：响应式革新、time-slicing
  - 更好的类型推断支持
  - 兼容性