Vue使用

Vue基本使用

模板（指令，插值）

插值，表达式

指定，动态属性

v-html: 会有xss风险，会覆盖子组件

computed和watch

computed有缓存，data不变则不会重新计算，提高运算的性能

watch默认是浅监听，只会监听对象第一层的数据变化

watch如何深度监听?

watch监听引用类型，拿不到old value

```
export default {
    data() {
        return {
            name: '夏衣旦'
            info: {
                city: '北京'
            }
        }
    },
    watch() {
        name(oldVal, val) {
            // 值类型，可以正常拿到oldVal和val
            console.log('watch name', oldVal, val)
        },
        info: {
            handler(oldVal, val) {
                // 引用类型，拿不到oldVal，因为指针相同，此时已经指向val
                // 注意：引用类型赋值是指针赋值的关系
                console.log('watch info', oldVal, val)
            },
            deep: true // 深度监听
        }
    }
}
```

class和style

使用动态属性 :class :style class可以是一个对象或数组的写法, style也可以是对象的写法

style要用驼峰式写法

条件渲染

v-if v-else的用法，可使用变量，也可以使用===表达式

v-if和v-show的区别?

v-if和v-show的使用场景?

循环列表渲染

如何遍历对象? v-for

```
// 遍历数组的方式
<ul>
    <li v-for="(item, index) in listArr" :key="item.id">
        {{index}}-{{item.id}}-{{item.title}}
    </li>
</ul>
// 遍历对象的方式
<ul>
    <li v-for="(val, key, index) in listObj" :key="key">
        {{index}}-{{key}}-{{val.title}}
    </li>
</ul>
```

key的重要性，key不能乱写（如random或者index），需要写一个和业务相关联的信息

v-for和v-if不能一起使用

事件

event参数，自定义参数

[观察]事件被绑定到哪里?

1. event是原生的
2. 事件被挂载到当前元素

事件修饰符

```
<!-- 阻止单击事件继续传播 -->
<a v-on:click.stop="doThis"></a>

<!-- 提交事件不再重载页面 -->
<form v-on:submit.prevent="onSubmit"></form>

<!-- 修饰符可以串联 -->
<a v-on:click.stop.prevent="doThat"></a>

<!-- 只有修饰符 -->
<form v-on:submit.prevent></form>

<!-- 添加事件监听器时使用事件捕获模式 -->
<!-- 即内部元素触发的事件先在此处理，然后才交由内部元素进行处理 -->
<div v-on:click.capture="doThis">...</div>

<!-- 只当在 event.target 是当前元素自身时触发处理函数 -->
<!-- 即事件不是从内部元素触发的 -->
```

```
<div v-on:click.self="doThat">...</div>
```

按键修饰符

```
<!-- 即使 Alt 或 Shift 被一同按下时也会触发 -->
<button @click.ctrl="onClick">A</button>

<!-- 有且只有 Ctrl 被按下的时候才触发 -->
<button @click.ctrl.exact="onClickExact">A</button>

<!-- 没有任何系统修饰符被按下的时候才触发 -->
<button @click.exact="onClick">A</button>
```

表单

v-model

常见表单项 textarea checkbox radio select

修饰符 lazy number trim

.trim 截取前后的空格

.lazy 相当于防抖的效果，等输入完的时候才会变化

.number 数字

refs

获取DOM元素节点的方式

Vue组件使用

props和$emit (父子组件的通讯方式)

组件间通讯-自定义事件 (兄弟组件的通讯)

```
import event from './event'

// 组件A
methods: {
    addTitle() {
        // 调用父组件的事件
        this.$emit('add', this.title)
        // 调用自定义事件 'onAddTitle'
        event.$emit('onAddTitle', this.title)
    }
}

// 组件B
mounted() {
    // 绑定自定义事件
    event.$on('onAddTitle', this.addTitleHandler)
},
beforeDestroy() {
    // 及时销毁，否则可能造成内存泄漏
    event.$off('onAddTitle', this.addTitleHandler)
},
```

```
methods: {
    addTitleHandler (title){
        console.log('on add title : ', title)
    }
}
```

组件生命周期

生命周期（单个组件）

```
1. 挂载阶段
2. 更新阶段
3. 销毁阶段
```

created和mounted有什么区别?

created只是把js的实例初始化了，但它只是存在于JS内存的一个变量而已，这个时候并没有开始渲染

mounted组件在页面渲染完了

生命周期方法

创建	**beforeCreate()**	**created()**
挂载	beforeMount()	mounted()
更新	beforeUpdate()	updated()
销毁	beforeDestroy()	destroyed()

生命周期（父子组件）

加载渲染	**父beforeCreate->父created->父beforeMount->子beforeCreate->子created->子beforeMount- >子mounted->父mounted**
子组件更新	父beforeUpdate->子beforeUpdate->子updated->父updated
父组件更新	父beforeUpdate->父updated
销毁过程	父beforeDestroy->子beforeDestroy->子destroyed->父destroyed

组件的调用顺序都是先父后子，渲染完成的顺序都是先子后父

组件的销毁操作是先父后子，销毁完成的顺序是先子后父

beforeDestroy() 里面我们会做哪些事情?

1. 及时的解绑事件，否则会造成内存泄漏
2. 销毁子组件
3. 事件监听器

Vue高级特性

自定义v-model

```
// 父组件
<template>
    <CustomModel v-model="name"/>
</template>
<script>
import CustomModel from './CustomModel'

export default {
    components: {
        CustomModel
    }
}
</script>

// 子组件
<template>
    // 1. input 使用了 :value 而不是v-model
    // 2. change1 和 model.event1 要对应起来
    // 3. text1 属性对应起来
    <input type="text" :value="text1" @input="$emit('change1',
$event.target.value)">
</template>
export default {
    model: {
        prop: 'text1', // 对应 props text1
        event: 'change1'
    },
    props: {
        text1: String,
        default() {
            return ''
        }
    }
}
```

$nextTick

原理：vue（和react）是异步渲染

data改变之后，DOM不会立刻渲染

页面渲染时，会将data的修改做整合，多次data修改只会渲染一次

$nextTick会在DOM渲染之后被触发，以获取最新DOM节点

slot插槽

基本使用：父组件往子组件插入一段内容

作用域插槽

　作用域插槽允许你传递一个模板而不是已经渲染好的元素给插槽，模板虽然是在父级作用域中渲染的，却能拿到子组件的数据

具名插槽

```
<slot name="header"></slot>
<slot></slot>
<slot name="footer"></slot>

<template v-slot:header>
    <h1>将插入header slot中</h1>
</template>
<p>将插入到main slot中，即未命名的slot</p>
<template v-slot:footer>
    <p>将插入footer slot中</p>
</template>
```

动态组件

需要根据数据，动态渲染的场景，即组件类型不确定

```
<component :is="component-name"/>
```

异步组件

import () 函数

按需加载，异步加载大组件

```
export default {
    components: {
        FormDemo: ()=>import('../FormDemo')
    }
}
```

keep-alive

缓存组件

频繁切换，但不需要重复渲染的时候

vue常见性能优化的解决方案之一

mixin

多个组件有相同的逻辑，抽离出来

配置信息会进行融合

缺点：

1. 变量来源不明确，不利于阅读
2. 多mixin可能会造成命名冲突
3. mixin和组件可能会出现多对多的关系，复杂度较高

```
import myMixin from './mixin'
export default {
    mixins: [myMixin]
}
```

Vuex 使用

基本概念

state

getters

action

mutation

用于Vue组件

dispatch

commit

mapState

mapGetters

mapActions

mapMutations

Vue-router使用

路由模式

1. hash模式：http://abc.com/#/user/10
2. H5 history模式：http://abc.com/user/20 （需要server端支持，因此无特殊需求可选择前者）

```
const router = new VueRouter({
    mode: 'history', // 使用h5 history模式
    routers: [...]
})
```

路由配置（动态路由，懒加载）

1. **动态路由**

```
const User = {
    // 获取参数如 10 20
    template: '<div>User {{ $route.params.id }} </div>'
}
const router = new VueRouter({
    routes: [
        // 动态路径参数，以冒号开头，能命中'/user/10' '/user/20'等格式的路由
        path: '/user/:id',
         component: User
    ]
})
```

2. **懒加载**

```
export default new VueRouter({
    routes: [
        {
            path: '/',
            component: ()=> import(
                './../components/Navigator'
            )
        },{
            path: '/feedback',
            component: ()=> import(
                './../components/FeedBack'
            )
        }
    ]
})
```

Vue原理

组件化和mvvm

“很久以前”就有组件化

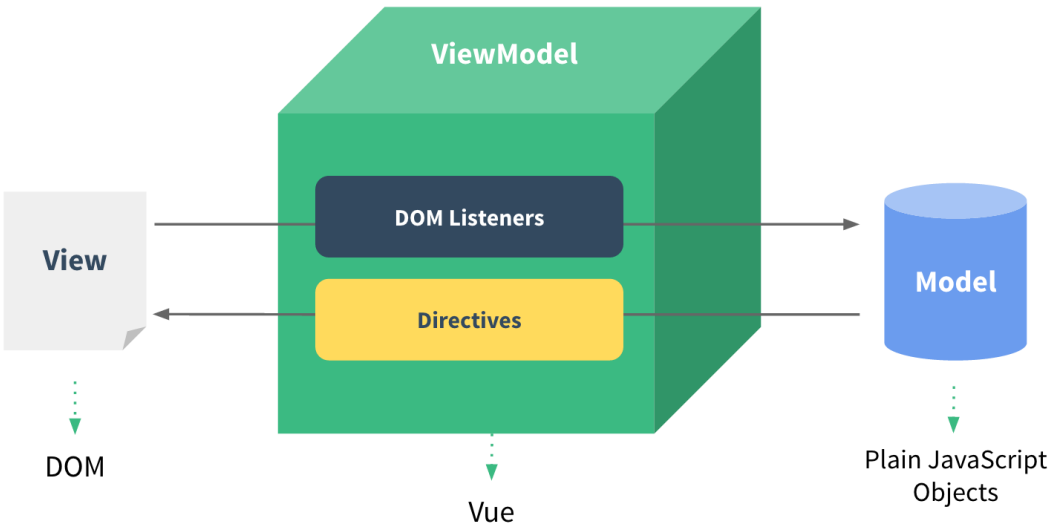
asp jsp php已经有组件化了

node.js中也有类似的组件化

数据驱动视图:

传统组件，只是静态渲染，更新还要依赖于操作DOM

数据驱动视图：Vue **MVVM**, React setState



响应式原理

定义

用Object.defineProperty（）方法getter setter监听data的属性；组件data的数据一旦变化，立刻触发视图的更新；

实现

Vue 2.0 实现：Object.defineProperty

```
// 触发更新视图
function updateView() {
    console.log('视图更新')
}

// 重新定义数组原型
const oldArrayProperty = Array.prototype
// 创建新对象，原型指向 oldArrayProperty ，再扩展新的方法不会影响原型
const arrProto = Object.create(oldArrayProperty);
['push', 'pop', 'shift', 'unshift', 'splice'].forEach(methodName => {
    arrProto[methodName] = function () {
        updateView() // 触发视图更新
        oldArrayProperty[methodName].call(this, ...arguments)
        // Array.prototype.push.call(this, ...arguments)
    }
})

// 重新定义属性，监听起来
function defineReactive(target, key, value) {
    // 深度监听
    observer(value)

    // 核心 API
    Object.defineProperty(target, key, {
        get() {
            return value
        },
        set(newValue) {
            if (newValue !== value) {
                // 深度监听
                observer(newValue)

                // 设置新值
                // 注意，value 一直在闭包中，此处设置完之后，再 get 时也是会获取最新的值
                value = newValue

                // 触发更新视图
                updateView()
            }
        }
    })
}

// 监听对象属性
function observer(target) {
    if (typeof target !== 'object' || target === null) {
```

```
        // 不是对象或数组
        return target
    }

    // 污染全局的 Array 原型
    // Array.prototype.push = function () {
    //     updateView()
    //     ...
    // }

    if (Array.isArray(target)) {
        target.__proto__ = arrProto
    }

    // 重新定义各个属性（for in 也可以遍历数组）
    for (let key in target) {
        defineReactive(target, key, target[key])
    }
}

// 准备数据
const data = {
    name: 'zhangsan',
    age: 20,
    info: {
        address: '北京' // 需要深度监听
    },
    nums: [10, 20, 30]
}

// 监听数据
observer(data)

// 测试
// data.name = 'lisi'
// data.age = 21
// // console.log('age', data.age)
// data.x = '100' // 新增属性，监听不到 —— 所以有 Vue.set
// delete data.name // 删除属性，监听不到 —— 所有已 Vue.delete
// data.info.address = '上海' // 深度监听
data.nums.push(4) // 监听数组
```

Vue 2.0 实现：Proxy

Object.defineProperty的缺点

1. 深度监听，需要递归到底，一次性计算量大
2. 无法监听新增属性/删除属性（Vue.set Vue.delete）
3. 无法原生监听数组，需要特殊处理

虚拟DOM和diff算法

背景

DOM操作很耗时

以前用jQuery，可以自行控制DOM操作的实际，手动调整

Vue和React是数据驱动试图，如何有效控制DOM操作?

解决方案

有了一定复杂度，像减少计算次数比较难

能不能把计算，更多的转移为JS计算，因为JS的执行速度更快

vdom - **用JS模拟DOM结构，计算出更小的变更，操作DOM**

```
<div id="div1" class="container">
    <p>vdom</p>
    <ul style="font-size: 20px">
        <li>a</li>
    </ul>
</div>
```

用JS去模拟以上DOM结构

```
{
    tag: 'div',
    props: {
        className: 'container',
        id: 'div1'
    },
    children: [
        {
            tag: 'p',
            children: 'vdom'
        },
        {
            tag: 'ul',
            props: {
                style: 'font-size: 20px'
            },
            children: [
                {
                    tag: 'li'.
                    children: 'a'
                }
            ]
        }
    ]
}
```

通过snabdom学习vdom

简洁强大的vdom库，易学易用

Vue参考snabdom实现的vdom和diff

Vue3.0重写了vdom的代码，优化了性能，但vdom基本理念不变，考点不变

vdom总结

用JS模拟DOM结构（vnode）

用diff算法做新旧vnode对比，得出最小的更新范围，最后更新DOM

数据驱动视图的模式下，有效控制DOM操作

vdom: patch的两种用法

1. patch(elem, vnode) // 将vnode渲染在elem元素
2. patch(vnode, newVnode) // 用newVnode去替换vnode

Diff算法

只比较同一层级，不跨级比较

tag不相同，则直接删掉重建，不再深度比较

tag和key，两者都相同，则认为是相同节点，不再深度比较

patchVnode()；addVnodes()；removeVnodes()；updateChildren() (key的重要性)

vdom的存在价值：数据驱动视图，控制DOM操作

模板编译

模板是vue开发最常用的部分，即与使用相关联的原理

模板不是html，有插值，指令，JS表达式

面试中会通过“组件渲染和更新过程”去考察

执行render函数生成vnode

前置知识：JS的with语法

```
const obj = { a : 100, b : 200 }
console.log(obj.a)
console.log(obj.b)
console.log(obj.c) // undefined
```

使用with，能改变{}内自由变量的查找方式，当作obj的属性来查找；如果找不到匹配的obj的属性，就会报错；但是with要慎用，因为它打破了作用域规则，使代码的易读性变差

```
with(obj){
    console.log(a)
    console.log(b)
    console.log(c) // 会报错
}
```

Vue的Compiler模板编译器：vue-template-compiler将模板编译成render函数

为什么需要模板编译?

1. 模板不是html，有指令，插值，JS表达式，能实现判断，循环
2. html是标签语言，只有JS才能实现判断，循环（只有JS才是图灵完备的）
3. 因此，模板一定是转换为某种JS模板，即编译模板

模板编译的过程

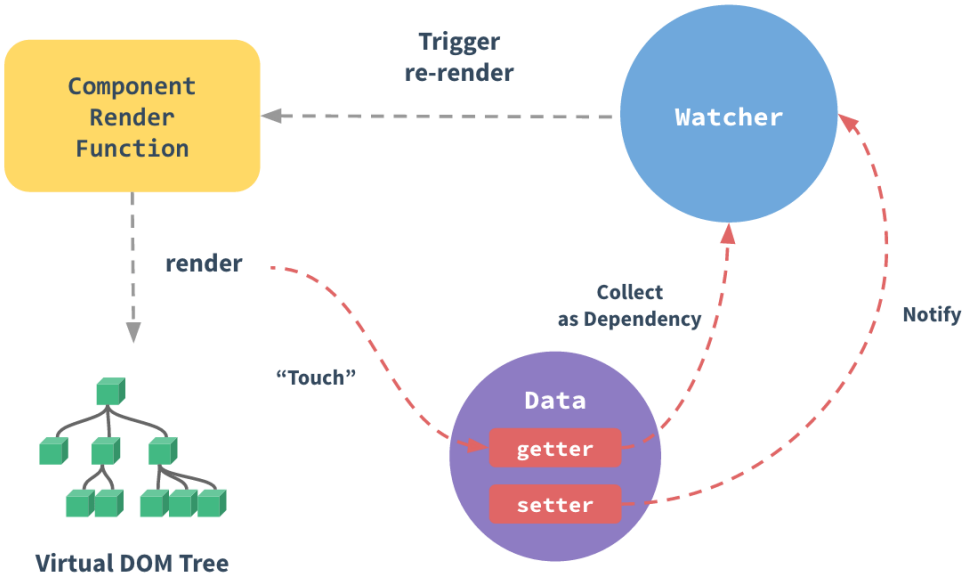
1. 模板编译为render函数，执行render函数返回vnode
2. 基于vnode再执行patch和diff
3. 使用webpack vue-loader，会在开发环境下编译模板（开发环境下编译模板，也是性能优化的一种方式）

Render()方法可以代替template

```
Vue.component('heading', {
    render: function(createElement) {
        return createElement(
            'h', this.level,
            [
                createElement('a', {
                    attrs: {
                        name: 'headerId',
                        href: '#'+'headerId'
                    }
                },'this is a tag')
            ]
        )
    }
})
```

组件渲染过程

1. 初次渲染过程
    1. 解析模板为render函数（或在开发环境已完成，vue-loader）
    2. 触发响应式，监听data对象的属性getter setter
    3. 执行render函数，生成vnode，patch(elem, vnode)
2. 更新过程
    1. 修改data，触发setter（此前getter中已被监听）
    2. 重新执行render函数，生成newVnode
    3. patch(vnode, newVnode)



3. 异步渲染（满足性能要求）

前端路由

面试题

Vue面试题

1. **v-show和v-if的区别**

   v-if只有当指令的表达式返回值为true的时候才会被渲染，为false的时候，元素是不存在于文档中的；

   v-show则不管指令表达式的返回值是什么，都会被渲染，并且只是基于css的样式diplay切换，元素始终存在于文档中

2. **什么时候用v-show，什么时候用v-if？**

   1.`v-if`在条件切换时，会对标签进行适当的创建和销毁，而`v-show`则仅在初始化时加载一次，因此`v-if`的开销相对来说会比`v-show`大。
   2.`v-if`是惰性的，只有当条件为真时才会真正渲染标签；如果初始条件不为真，则`v-if`不会去渲染标签。`v-show`则无论初始条件是否成立，都会渲染标签，它仅仅做的只是简单的CSS切换

   结论：组件需要频繁切换的时候，使用v-show，去减少开销

3. **为何v-for中要用key**

4. **描述vue组件生命周期（有父子组件的情况下）**

5. **vue组件如何通讯**

6. **描述组件渲染和更新的过程**

7. **双向数据绑定v-model的实现原理**

React面试题

1. **组件之间如何通讯**

   父子组件props

   自定义事件

   Redux和Context

2. **JSX本质是什么**

   createElement

   执行返回vnode

3. **Context是什么，如何应用?**

   父组件，向其下所有子孙组件传递信息

   如一些简单的公共信息：主题色，语言等

   复杂的公共信息，请用redux

4. **shouldComponentUpdate (简称SCU)用途**

   性能优化

   配合“不可变值”一起使用，否则会出错

5. **描述redux单项数据流**

   图解

6. **setState场景题**

```
componentDidMount() {
    // count初始值为0
    this.setState({count: this.state.count + 1})
    console.log('1', this.state.count) // 0
    this.setState({count: this.state.count + 1})
    console.log('2', this.state.count) // 0
    setTimeout(()=>{
        this.setState({count: this.state.count+1})
        console.log('3', this.state.count) //2
    })
    setTimeout(()=>{
        this.setState({count: this.state.count+1})
        console.log('4', this.state.count) //3
    })
}
// setState是异步的，并且会合并1和2的操作
```

7. **什么是纯函数**

   返回一个新值，没有副作用（不会“偷偷”修改其他值）

   重点：不可变值

   如 arr1 = arr.slice()

8. **React组件生命周期**

   单组件生命周期

   父子组件生命周期

9. **React发起ajax应该在哪个生命周期**

   同vue

   componentDidMount (DOM已经渲染完的生命周期上)

10. **渲染列表，为何使用key**

    同vue，必须用key，且不能是index和random

    diff算法中通过tag和key来判断，是否是same node

    减少渲染次数，提升渲染性能

11. **函数组件和class组件的区别**

    纯函数，输入props，输出JSX

    没有实例，没有生命周期，没有state

    不能扩展其他方法

12. **什么是受控组件**

    表单的值，受state控制

    需要自行监听onChange，更新state

    对比非受控组件

13. **何时使用异步组件**

    同vue

    加载大组件

    路由懒加载

14. **多个组件有公共逻辑，如何抽离**

    高阶组件

    Render Props

    mixin已被React废弃

15. **redux如何进行异步请求**

    使用异步action

    如redux-thunk

16. **react-router如何配置懒加载**

    lazy

```
import React, {Suspense, lazy} from 'react'

const Home = lazy(()=>import('./routes/Home'))
const About = lazy(()=>import('./routes/About'))
```

17. **PureComponent有何区别**

    实现了浅比较的shouldComponentUpdate

    优化性能

    但要结合不可变值使用

18. **React事件和DOM事件的区别**

    所有事件挂载到document上

    event不是原生的，是SyntheticEvent合成事件对象

    dispatchEvent

19. **React性能优化**

    1. sss渲染列表时加key
    2. 自定义事件，DOM事件及时销毁
    3. 合理使用异步组件
    4. 减少函数bind this的次数
    5. 合理使用SCU PureComponent和memo
    6. 合理使用Immutable.js
    7. webpack层面的优化
    8. 前端通用的性能优化，如图片懒加载
    9. 使用SSR （服务器端渲染）

20. **React和Vue的区别**

    共同点：

    1. 都支持组件化
    2. 都是数据驱动视图
    3. 都是用v dom操作dom

    不同点：

    1. React使用JSX拥抱JS，Vue使用模板拥抱html (开发认知)
    2. React函数式编程，Vue声明式编程
    3. React更多需要自力更生，Vue把想要的都给你

框架综合应用

1. 基于React设计一个todolist （组件结构，redux state数据结构）
2. 基于VUE设计一个购物车 (组件结构，vuex state数据结构)

Webpack面试题

1. 前端代码为何要进行构建和打包
2. module chunk bundle分别是什么意思？有何区别
3. loader和plugin的区别?
4. webpack如何实现懒加载
5. webpack常见性能优化
6. babel-runtime和babel-polyfill的区别