2021上半年各大厂核心面试题解析（第二期）

前情提要

上节课的面试专题讲的是webpack, 咱们自己手写实现了一个打包工具, 通过这种形式来了解webpack的运行原理.

有的同学说这种方式, 比强行看源码更能印象深刻, 也更方便自己的理解.

本节预告

这节课的面试专题选了Vue相关的, 同学们同样要作为候选人, 而我作为面试官, 一个个问题前后关联的抛出来.

随着我提出的一个个问题, 大家一定要随着去思考, 去考虑这个问题在面试的时候会怎么回答.

最后手写实现一个简单的vue框架, 帮忙大家理解.

经典面试题

一、你了解Vue的响应式原理吗?

首先要了解Vue中的三个核心类：

1. Observer: 给对象的属性添加 getter 和 setter，用于依赖收集和派发更新
2. Dep: 用于收集当前响应式对象的依赖关系,每个响应式对象包括子对象都拥有一个 Dep 实例, dep.subs是watcher实例的数组. 当数据有变更时,会通过 dep.notify()通知各个 watcher。
3. Watcher: 观察者对象, 实例分为render watcher(渲染),computed watcher(计算属性),user watcher(侦听器)三种

上面提到了两个看起来比较高级的名词:

- **依赖收集**

1. initState 时,对 computed 属性初始化时,触发 computed watcher 依赖收集
2. initState 时,对侦听属性初始化时,触发 user watcher 依赖收集
3. render()的过程,触发 render watcher 依赖收集
4. re-render 时,vm.render()再次执行,会移除所有 subs 中的 watcher 的订阅,重新赋值。

- **派发更新**

1. 组件中对响应的数据进行了修改,触发 setter 的逻辑
2. 调用 dep.notify()
3. 遍历所有的 subs（Watcher 实例）,调用每一个 watcher 的 update 方法。

总结一下原理：

当创建 Vue 实例时,vue 会遍历 data 选项的属性,利用 Object.defineProperty 为属性添加 getter 和 setter 对数据的读取进行劫持（getter 用来依赖收集,setter 用来派发更新）,并且在内部追踪依赖,在属性被访问和修改时通知变化。

每个组件实例会有相应的 watcher 实例,会在组件渲染的过程中记录依赖的所有数据属性进行依赖收集,之后依赖项被改动时,setter 方法会通知依赖与此 data 的 watcher 实例重新计算（派发更新）,从而使它关联的组件重新渲染。

二、计算属性的实现原理?

上面提到的watcher实例, 就有一个叫做computed watcher的东西, 这个就是计算属性的watcher。

computed watcher 持有一个 dep 实例, 通过 this.dirty 属性标记计算属性是否需要重新求值。

当computed的依赖值改变时, 就会通知订阅的watcher进行更新，对于computed的watcher会将dirty设置为true并且进行计算属性方法的调用.

1. computed所谓的缓存是指什么?

计算属性是基于它们的响应式依赖进行缓存的。只在相关响应式依赖发生改变时它们才会重新求值.

2. 那computed缓存存在的意义是什么?

比如computed内的操作非常耗时, 可能是遍历一个大数组. 计算一次可能要耗时1s, 那么当后续再通过计算属性获取的时候, 如果依赖的值没有变化, 就无需重新计算一遍了.

```
const largeArray = [
    {....},
    {....},
```

```
    {....},
    {....},
    ....
];
data: {
    id: 1,
},
computed: {
    currentItem: function () {
      return largeArray.find(item => item.id === this.id);
    }
  }
```

3. 以下情况, computed可以监听到数据的变化吗?

```
computed: {
    storageMsg: function () {
      return sessionStorage.getItem('xx');
    },
    time: function() {
        return Date.now()
    }
  }
```

三、Vue.nextTick的原理?

Vue是异步执行dom更新的，一旦观察到数据变化，Vue就会开启一个**异步队列**，然后把在同一个事件循环 (event loop) 当中观察到数据变化的 watcher 推送进这个队列。如果这个watcher被触发多次，只会被推送到队列一次。
这种缓冲行为可以有效的去掉重复数据造成的不必要的计算和DOm操作。而在下一个事件循环时，Vue会清空队列，并进行必要的DOM更新。

而vue内部这个异步队列是怎么开启的? 这里有一个优先级, Promise.then > MutationObserver > setImmediate > setTimeout

所以可以理解为, nextTick会优先尝试使用微任务, 如果浏览器不支持, 就用宏任务.

当你设置 vm.someData = 'new value'，DOM 并不会马上更新，而是在异步队列被清除，也就是下一个事件循环开始时执行更新时才会进行必要的DOM更新.

所以nextTick的回调是在下一轮事件循环里执行的.

一般在什么时候用到nextTick呢?

1. 在数据变化后要执行的某个操作，而这个操作需要使用随数据改变而改变的DOM结构的时候，这个操作都应该放进Vue.nextTick()的回调函数中

```
<template>
    <div v-if="loaded" ref="test"></div>
</template>

async showDiv() {
    this.loaded = true;
    await Vue.nextTick();
    this.$refs.test.xxxxx;
}
```

手写一个简单的Vue, 实现响应式更新

1. 首先新建一个目录, 分别建好咱们需要的核心文件

- index.html 主页面
- vue.js Vue主文件
- compiler.js 编译模板, 解析指令,
- dep.js 收集依赖关系, 存储观察者. 发布订阅
- observer.js 数据劫持
- watcher.js 观察者对象

2. index.html 内容

首先要有一个根元素, 咱们就让它的id为"app".

```
```

```
<!DOCTYPE html>
<html lang="cn">
<head>
  <title>My Vue</title>
</head>
<body>
  <div id="app"></div>
</body>
</html>
```

1. 初始化Vue class

   Vue的类就在vue.js文件里实现, 包含构造函数、接收配置等等.

   先来实现一下 constructor, 接收传入的数据并存储下来. 这里咱们内部的变量都用$命名, 便于区分.

```
export default class Vue {
    constructor(options = {}) {
        // 存储options, data, methods
        this.$options = options;
        this.$data = options.data;
        this.$methods = options.methods;
    }
}
```

2. 然后需要获取根元素, 咱们单独写一个方法来处理这件事, 同时简单检查一下传入的options.el是否合法.

```
export default class Vue {
    constructor(options = {}) {
        // 存储options, data, methods
        this.$options = options;
        this.$data = options.data;
        this.$methods = options.methods;
```

```
        this.initRootElement(options);
    }

    /**
     * 获取根元素，并存储到Vue实例. 这里简单兼容一下，检查一下传入的el是否合规.
     */
    initRootElement(options) {
        if (typeof options.el === 'string') {
            // 传入的是元素id或者class
            this.$el = document.querySelector(options.el);
        } else if (options.el instanceof HTMLElement) {
            this.$el = options.el;
        }

        if (!this.$el) {
            throw new Error('传入的el不合法，请传入css selector或者HTMLElement')
        }
    }
}
```

3. 到这里咱们先运行一下代码看看能不能如我们所愿, 检查el是否合法并且正常存储各种属性.

   新建index.js文件, 初始化Vue

```
import Vue from './myvue/vue.js';

const vm = new Vue({
    el: '#app',
    data: {
        msg: 'Hello Vue',
    },
    methods: {
        handler() {
            console.log(333)
        }
    }
```

```
})

console.log(vm);
```

index.html文件引入index.js文件, 运行一下看看

```
<!DOCTYPE html>
<html lang="cn">
<head>
  <title>My Vue</title>
</head>
<body>
  <div id="app"></div>
  <script src="./index.js"></script>
</body>
</html>
```

发现报错了, Uncaught SyntaxError: Cannot use import statement outside a module, 是因为import 需要在esmodule里使用。
在script标签上加上module即可.

```
<script src="./index.js" type="module"></script>
```

再运行一下, 发现可以正常输出vue实例了, $el也正常获取到了.

- 输入一个错误的el试试呢? 比如出传入#app1，一个不存在的元素.
- 直接输入一个Html元素呢?

4. 回想一下, 咱们在vue组件里是不是可以通过this来获取data里的属性? 所以, 下面我们要把data里的属性都挂载到vue实例上.

```
export default class Vue {
    constructor(options = {}) {
        // 存储options, data, methods
```

```
        this.$options = options;
        this.$data = options.data;
        this.$methods = options.methods;

        this.initRootElement(options);

        // 利用Object.defineProperty将data里的属性注入到vue实例中
        this._proxyData(this.$data)
    }

    /**
     * 获取根元素, 并存储到Vue实例. 这里简单兼容一下, 检查一下传入的el是否合规.
     */
    initRootElement(options) {
        if (typeof options.el === 'string') {
            // 传入的是元素id或者class
            this.$el = document.querySelector(options.el);
        } else if (options.el instanceof HTMLElement) {
            this.$el = options.el;
        }

        if (!this.$el) {
            throw new Error('传入的el不合法, 请传入css selector或者HTMLElement')
        }
    }

    _proxyData(data) {
        // 遍历所有data
        Object.keys(data).forEach(key => {
            // 将data属性注入到vue中
            Object.defineProperty(this, key, {
                enumerable: true,
                configurable: true,
                get() {
                    return data[key];
                },
                set(newValue) {
                    if (data[key] === newValue) {
                        return
```

```
                    }
                    data[key] = newValue
                }
            })
        })
    }
}
```

再来运行一下代码试试, 看看咱们的data是否都绑定到vue实例上了?

5. 接下来, 咱们先把几个核心类都声明好

具体的实现先不管, 先声明好每个类里的方法, 明确每个方法的作用. 后期再一一实现. 这也是构建大型系统时的一种方式, 先明确整体架构, 具体实现在最后实施.
记住, 一定要写好注释. 而对于这种外界可以使用的方法, 注释要用jsDoc的形式, 这样的话外界在使用的时候就可以直接看到你方法的注释.

- dep.js

```
export default class Dep {
    constructor() {
        // 存储所有的观察者
        this.subs = []
    }

    /** 添加观察者 */
    addSub(watcher) {

    }

    /** 发送通知 */
    notify() {

    }
}
```

- observer.js

```
export default class Observer {
    constructor(data) {
        this.traverse(data)
    }

    /** 递归遍历data里的所有属性 */
    traverse(data) {

    }

    /** 给传入的数据设置getter/setter */
    defineReactive(obj, key, val) {}
}
```

- watcher.js

```
// 获取更改前的值存储起来，并创建一个 update 实例方法，在值被更改时去执行实例的 callback 以达到视图的更新。
export default class Watcher {
    /**
     * vm: vue实例
     * key: data中的属性名
     * cb: 负责更新视图的回调函数
     */
    constructor(vm, key, cb) {

    }

    /** 当数据发生变化的时候更新视图 */
    update() {

    }
}
```

- compiler.js

```
export default class Compiler {
    constructor(vm) {
        this.compile(vm.$el)
    }

    /** 编译模版 */
    compile(el) {

    }
}
```

6. 想一下应该怎么调用这些方法? 在vue初始化的时候都应该做些什么?

```
import Observer from './observer.js';
import Compiler from './compiler.js';

constructor(options = {}) {
    // 存储options, data, methods
    this.$options = options;
    this.$data = options.data;
    this.$methods = options.methods;

    this.initRootElement(options);

    // 利用Object.defineProperty将data里的属性注入到vue实例中
    this._proxyData(this.$data);

    // 实例化observe对象，监听数据变化
    new Observer(this.$data);

    // 实例化compiler对象，解析指令和差值表达式
    new Compiler(this)
}
```

7. 完善 dep.js

发布订阅模式.

记住, dep是用来存储所有观察者的, 也就是watcher.

而我们watcher的定义, 每个watcher都会有一个update方法对吧, 用来更新视图的?

- addSub, 我们如果发现watcher没有update方法, 也就没必要添加到subs里了.
- notify, 是提供给外界调用的, 当数据有变更的时候, 外界会调用notify去通知各个watcher, 也就是执行watcher.update()

```
export default class Dep {
    constructor() {
        // 存储所有的观察者
        this.subs = []
    }

    /** 添加观察者 */
    addSub(watcher) {
        if (watcher && watcher.update) {
            this.subs.push(watcher)
        }
    }

    /** 发送通知 */
    notify() {
        this.subs.forEach(watcher => {
            watcher.update()
        })
    }
}
```

考虑几个问题:

- Dep 在哪里实例化? 在哪里addSub?
- Dep 的 notify 方法应该在哪里调用?

8. 完善watcher

观察者类.

```
import Dep from './dep.js';

export default class Watcher {
    /**
     * vm: vue实例
     * key: data中的属性名
     * cb: 负责更新视图的回调函数
     */
    constructor(vm, key, cb) {
        this.vm = vm
        // data中的属性名称
        this.key = key
        // 回调函数负责更新视图
        this.cb = cb
        // 把watcher对象记录到Dep类的静态属性target
        Dep.target = this
        // 触发get方法，在get方法中会调用addSub
        this.oldValue = vm[key]
        Dep.target = null
    }

    /** 当数据发生变化的时候更新视图 */
    update() {
        let newValue = this.vm[this.key]
        if (this.oldValue === newValue) {
            return
        }
        this.cb(newValue)
    }
}
```

考虑两个问题：

- 通过vm[key]获取oldValue前，为什么要将当前的实例挂在 Dep 上，为什么获取之后又要置为null?
- update 方法内部执行了 callback 函数，但是 update 在什么时候执行?

9. 完善compiler.js

```
import Watcher from './watcher.js';

export default class Compiler {
    constructor(vm) {
        this.el = vm.$el
        this.vm = vm
        this.methods = vm.$methods
        this.compile(vm.$el)
    }

    // 编译模版
    compile(el) {
        let childNodes = el.childNodes
        Array.from(childNodes).forEach(node => {
            if (this.isTextNode(node)) { // 处理文本节点
                this.compileText(node)
            } else if (this.isElementNode(node)) { // 处理元素节点
                this.compileElement(node)
            }
            // 如果还有子节点，递归调用
            if (node.childNodes && node.childNodes.length > 0) {
                this.compile(node)
            }
        })
    }

    // 编译元素节点，处理指令
    compileElement(node) {
        // console.log(node.attributes)
```

爪哇教育 Power Human With Education

```
        if (node.attributes.length) {
            Array.from(node.attributes).forEach(attr => { // 遍历所有元素节点
                let attrName = attr.name
                if (this.isDirective(attrName)) { // 判断是否是指令
                    attrName = attrName.indexOf(':') > -1 ?
attrName.substr(5) : attrName.substr(2) // 获取 v- 后面的值
                    let key = attr.value // 获取data名称
                    this.update(node, key, attrName)
                }
            })
        }
    }

    // 更新
    update(node, key, attrName) {
        const updateFn = this[attrName + 'Updater']
        updateFn && updateFn.call(this, node, this.vm[key], key, attrName)
    }

    // 解析 v-text
    textUpdater(node, value, key) {
        node.textContent = value
        new Watcher(this.vm, key, (newValue) => { // 创建watcher对象，当数据改变更
新视图
            node.textContent = newValue
        })
    }

    // 解析 v-model
    modelUpdater(node, value, key) {
        node.value = value
        new Watcher(this.vm, key, (newValue) => { // 创建watcher对象，当数据改变更
新视图
            node.value = newValue
        })
        // 双向绑定
        node.addEventListener('input', () => {
            this.vm[key] = node.value
        })
```

Power Human With Education

```
    }

    // 解析 v-html
    htmlUpdater(node, value, key) {
        node.innerHTML = value
        new Watcher(this.vm, key, newValue => {
            node.textContent = newValue
        })
    }

    // 解析 v-on:click
    clickUpdater(node, value, key, attrName) {
        node.addEventListener(attrName, this.methods[key])
    }

    // 编译文本节点，处理差值表达式，{{  }}
    compileText(node) {
        // 获取 {{  }} 中的值
        let reg = /\{\{(.+?)\}\}/
        let value = node.textContent
        if (reg.test(value)) {
            let key = RegExp.$1.trim() // 返回匹配到的第一个字符串，去掉空格
            node.textContent = value.replace(reg, this.vm[key])
            new Watcher(this.vm, key, (newValue) => { // 创建watcher对象，当数据改
变更新视图
                node.textContent = newValue
            })
        }
    }

    // 判断元素属性是否是指令
    isDirective(attrName) {
        return attrName.startsWith('v-')
    }

    // 判断是否是文本节点
    isTextNode(node) {
        return node.nodeType === 3
    }
```

```
    // 判断是否是元素节点
    isElementNode(node) {
        return node.nodeType === 1
    }
}
```

10. 完善observer.js

```
import Dep from './dep.js'

export default class Observer {
    constructor(data) {
        this.traverse(data)
    }

    /** 递归遍历data里的所有属性 */
    traverse(data) {
        if (!data || typeof data !== 'object') {
            return
        }
        Object.keys(data).forEach(key => {
            this.defineReactive(data, key, data[key])
        })
    }

    /** 给传入的数据设置getter/setter */
    defineReactive(obj, key, val) {
        const that = this
        this.traverse(val); // 递归设置

        const dep = new Dep() // 负责收集依赖，并发送通知

        Object.defineProperty(obj, key, {
            configurable: true,
            enumerable: true,
            get() {
```

```
                Dep.target && dep.addSub(Dep.target) // 收集依赖
                return val;
            },
            set(newValue) {
                if (newValue === val) {
                    return
                }
                val = newValue
                that.traverse(newValue) // newValue可能是个对象
                dep.notify() // 通知watcher数据更新了
            }
        })
    }
}
```

在模板编译的过程中，遇到模板中绑定的变量，就会解析，并创建 watcher，会在 Watcher 类的内部获取旧值，即当前的值。
这样就触发了 get，在 get 中就可以将这个 watcher 添加到 Dep 的 subs 数组中进行统一管理。
因为在代码中获取 data 中的值操作比较多，会经常触发 get，我们又要保证 watcher 不会被重复添加，所以在 Watcher 类中，获取旧值并保存后，立即将 Dep.target 赋值为 null，并且在触发 get 时对 Dep.target 进行了判空，存在才调用 Dep 的 addSub 进行添加。

11. 验证一下

    index.html

```
<div id="app">
    <h1>template表达式</h1>
    <h3>{{ msg }}</h3>
    <h3>{{ count }}</h3>
    <br />

    <h1>v-text</h1>
    <div v-text="msg"></div>
    <br />
```

```
    <h1>v-model</h1>
    <div v-html="myHtml"></div>
    <br />

    <h1>v-model</h1>
    <input type="text" v-model="msg">
    <input type="text" v-model="count">
    <button v-on:click="handler">按钮</button>
  </div>
```

index.js

```
const vm = new Vue({
    el: '#app',
    data: {
        msg: 'Hello Vue',
        count: 100,
        myHtml: '<ul><li>这是v-html编译的</li></ul>',
    },
    methods: {
        handler() {
            alert('handler')
        }
    }
})
```