

面试



最后一次课请一个百度的T6 实战一下

课堂目标

1. 掌握前端工程师面试的方方面面
2. 掌握常见面试题
3. 简历优化
4. 职业生涯软技能

知识要点

1. javascript
2. ES6
3. this
4. promise
5. 浏览器
6. 安全
7. vue
8. react
9. 工程化
10. 计算机基础

起步

面试

JD分析& 面试准备

◎面试准备 ◎JD分析 ◎技术栈

一面常见面试题

◎JavaScript CSS基础面试题 ◎闭包
◎this ◎函数作用域 ◎CSS进阶
◎ES6 & ES7语法回顾

二面常见面试题

◎进阶面试题 ◎promise原理 ◎react原理
◎vue原理 ◎webpack架构

三面常见面试题

◎项目架构设计 ◎职业生涯

面试心态和 面试学习法

◎面试驱动学习 ◎查缺补漏

面试准备

不打无准备之仗，面试就是一个考试 所谓台上三分钟，台下十年功，必须要通过某个门槛，才能通过，拿到理想的offer，所以需要我们之前提前2~3个月来精心的去准备这场战斗

talk is cheap show me the money code

JD分析

去考试之前，要先审题，大家首先要分析jd，有针对的去准备，不是每个岗位都适合自己

职位诱惑:

上市公司,大牛团队,福利优厚,晋升空间大

职位描述:

岗位职责:

- 1、参与项目、产品需求分析与设计,负责前端架构设计及技术路线;
- 2、组织前端开发团队完成前端功能设计、开发、测试和交付。
- 3、负责前端核心功能的架构与代码模板编写,对系统核心模块进行开发和维护;
- 4、制定前端开发规范,参与制定技术标准,编写相应的技术文档,对通用技术进行整理;
- 5、负责前端开发人员的招聘和考核,定期开展前端开发工程师培训,促进团队成员的进步。

岗位要求:

- 1、5年以上Web前端研发经验;
- 2、能够熟练的设计和开发基于Javascript的复杂web应用并有实践经验;
- 3、必须精通VueJS或Angular2/3/4/5框架,熟悉ReactJs, NodeJs等框架,有产品设计经验优先;
- 4、熟练掌握gulp, webpack, browserify等工具,并且对实现细节有研究,有ionic, Electron等Hybird框架经验尤佳;
- 5、能凭借丰富的开发经验快速定位并解决各种前端问题;
- 6、具有微信公众号开发经验,了解小程序的开发;
- 7、具备良好的识别和设计通用框架及模块的能力;
- 8、逻辑思维强、注重团队协作;

工作地址

北京 - 海淀区 - 中关村

[查看地图](#)

关键字 架构设计、精通vue或者angular、公众号、小程序、沟通

职位诱惑:

晋升空间,技术氛围

职位描述:

工作职责:

- 1、负责餐饮生态前端技术的架构设计，并参与整体的技术中台规划建设。保证架构的可持续性发展，并具备一定程度的应变能力。
- 2、负责产品需求到实现过程中的设计与技术选型工作，并参与核心业务模块、通用业务系统的开发工作。
- 3、负责提升各技术项目中的代码质量、设计质量和工程质量。
- 4、保持积极、负责的工作态度，对团队充满正能量、对项目充满强自驱力。

职位要求:

- 1、3年以上的前端领域开发经验，有主导前端架构设计的经历
- 2、扎实的计算机基础，对数据结构和算法有一定的了解
- 3、Javascript/Hybrid/Node至少有一个方向有深入的了解
- 4、至少了解一门服务器端相关技术，熟练掌握Linux

加分项:

- 1、对性能优化、多端适配、开发者框架有实际产品经验者优先
- 2、有长期维护技术专栏，个人项目或站点者优先
- 3、有React/React Native 或 Express/Koa 开发经验者优先

工作地址

北京 - 朝阳区 - 望京 - 北京市朝阳区望京东路4号恒基伟业C座（美团点评北京总部）

[查看地图](#)

关键字：餐饮生态(pc 移动端) 前端架构设计、计算机基础、react/native、nodejs

其他

1. 蚂蚁 1. 精通React，熟悉前端各种轮子 2. https://www.zhipin.com/job_detail/82122f558bfb9c403R72ty5GVY~.html
2. 美团点评
 1. 对前端主流框架，包括Vue、React，有实际项目经验，能够深入剖析框架部分及实现原理；
 2. 熟练掌握webpack，gulp等工程化工具，并利用这些工具持续改善团队的线上/线下研发效率；
 3. 对网站性能优化、体验提升有实际项目经验；
 4. 熟练使用NodeJS，并使用NodeJS进行过完整的系统设计与实现；
 5. https://www.zhipin.com/job_detail/d722670c076b99e103V83d6-GVQ~.html
3. vipkid
 1. 对常见开源JavaScript框架/项目有过深入研究，理解其源代码，能对其扩展或优化并贡献过代码，或者自己实际编写过框架和基础JavaScript库；

开课吧web全栈架构师

2. 熟悉前端工程化与组件化开发，并有实践经验（如 Webpack/Gulp、Vue/React 等或其他前端工具），熟悉MVC、MVVM等前端开发模型；
3. https://www.zhipin.com/job_detail/1d1c025b03356c0f1Xdy39y-GFs~.html
4. 字节跳动
 1. 3年以上前端开发经验，能理解目前流行的框架(react/redux/mobx)的设计思路并能进行源码分析；
 2. https://www.zhipin.com/job_detail/2b28341ba5c80d0003Z609S5E1M~.html?ka=rcmd_list_job_13
5. 好未来
 1. 熟悉掌握React、Vue框架，至少独立负责2个以上基于该技术栈的项目
 2. 掌握Node.js，对前端技术发展的历史、现在有全面了解，对未来有自己的想法
 3. https://www.zhipin.com/job_detail/c6b102c957c17d2b1XB40tq5FFE~.html?ka=rcmd_list_job_3
6. 百度
 1. 具备扎实的计算机基础，对数据结构和算法设计有一定理解
 2. 具有一定的软件工程意识，对数据结构和算法设计有充分理解；
 3. https://www.zhipin.com/job_detail/33adee8812f18fa803R83di7E1s~.html?ka=search_list_11

简历

程序员的简历不需要特别花哨，但是很多人不会写简历

1. 基本信息 姓名 年龄 手机 邮箱
2. 学历
3. 工作经历
4. 开源项目
5. 技术点(最好是源码级) 最重要的 一定要有杀手技能

附加信息

有前端团队的管理经验。

使用 **Html**、**Css** 编写 **PC** 端页面，能够快速定位并解决浏览器兼容性问题，完美还原 **UI** 设计。

使用 **Html5**、**Css3** 编写移动端 **H5** 页面， 适配 **Android**、**Ios** 系统不同分辨率机型，解决移动端遇到的问题，完美还原 **UI** 设计。

使用 **Jquery**、**Javascript**、**Zepto.js**，实现 **PC** 端和移动端交互效果。

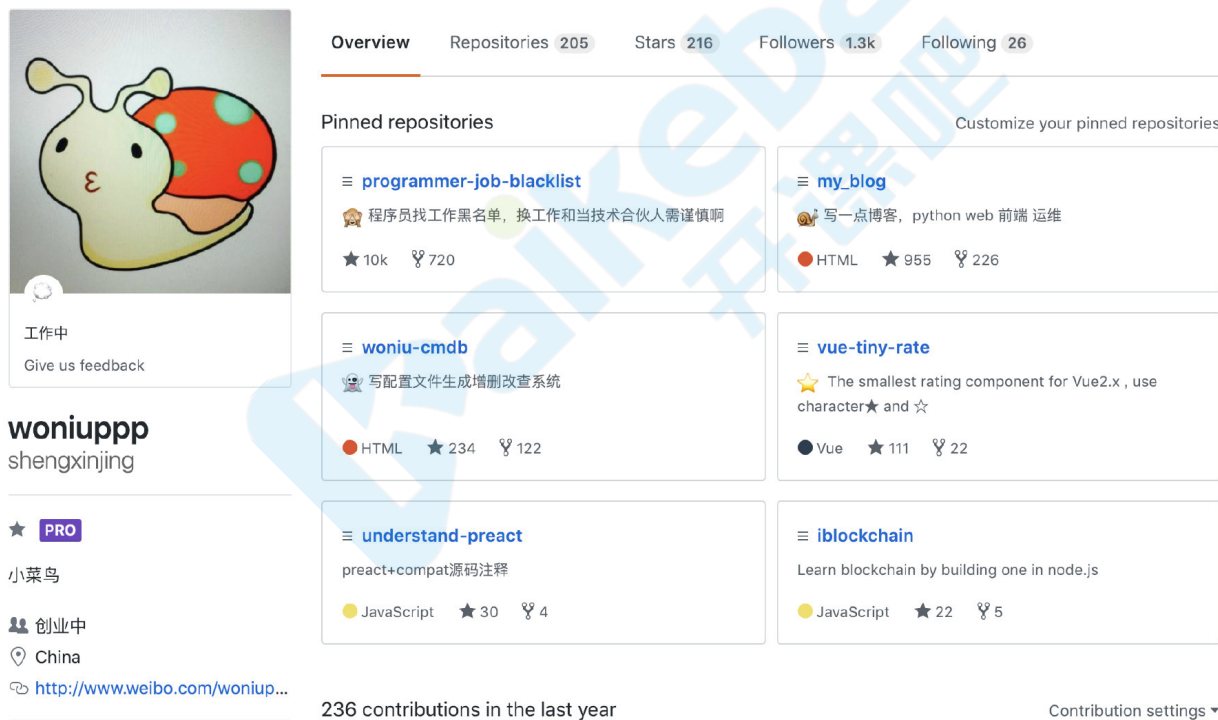
使用 **Mvvm** 框架 **Vue.js** 并应用到开发项目中。

个人简介

- 擅长前端和python开发
- 擅长react全家桶开发, react+redux+react-router, 看过源码, 了解原理
- 自己实现过迷你的reactjs
- 自己实现过迷你的vuejs
- 擅长前端工程化的架构设计
- 看过angular的源码并用es6+webpack写了一个简版angular, 擅长mvvm库做组件化开发
- 懂运维,python实现过一套简单的监控系统, 后端代码仿照memcached架构, 基于epoll和自己定义的状态机, 网络用优化后的netstring协议
- 熟悉区块链开发
- 熟练构建完整的交易所前端

能击中面试官内心的, 就是要展示你是一个专业的程序员, 最直观的 就是 把控项目的能力, 比如开源项目和源码! 需要日常做准备

<https://github.com/shengxinjing> 欢迎follow 🐌



Overview Repositories 205 Stars 216 Followers 1.3k Following 26

Pinned repositories Customize your pinned repositories

- programmer-job-blacklist**
程序员找工作黑名单, 换工作和当技术合伙人需谨慎啊
★ 10k 🍴 720
- my_blog**
写一点博客, python web 前端 运维
HTML ★ 955 🍴 226
- woniu-cmdb**
写配置文件生成增删改查系统
HTML ★ 234 🍴 122
- vue-tiny-rate**
The smallest rating component for Vue2.x, use character★ and ☆
Vue ★ 111 🍴 22
- understand-preact**
preact+compat源码注释
JavaScript ★ 30 🍴 4
- iblockchain**
Learn blockchain by building one in node.js
JavaScript ★ 22 🍴 5

woniuppp
shengxinjing

★ PRO
小菜鸟
👤 创业中
📍 China
🔗 <http://www.weibo.com/woniuppp>

236 contributions in the last year Contribution settings ▾

技术栈

ES6 (javascript)

1. let const

2. 箭头函数
3. class
4. promise
5. 解构
6. import

<<<<<< HEAD 如何统计当前网页出现过多少种html标签

1. 基本的dom操作
2. 去重

http协议

浏览器里大部分都是http协议

=====

如何统计当前网页出现过多少种html标签

| | | | | | | 627873b57b909222a62e528942b8316e18d8467c

```
new Set([...document.getElementsByTagName('*')].map(e=>e.tagName)).size
```

请写出打印结果

```
function Foo() {
  Foo.a = function() {
    console.log(1)
  }
  this.a = function() {
    console.log(2)
  }
}
Foo.prototype.a = function() {
  console.log(3)
}
Foo.a = function() {
  console.log(4)
}
Foo.a();
let obj = new Foo();
obj.a();
Foo.a();
```

```
// 421
function Foo() {
  Foo.a = function() {
    console.log(1)
  }
  this.a = function() {
    console.log(2)
  }
}

// 以上只是 Foo 的构建方法，没有产生实例，此刻也没有执行

Foo.prototype.a = function() {
  console.log(3)
}

// 现在在 Foo 上挂载了原型方法 a，方法输出值为 3

Foo.a = function() {
  console.log(4)
}

// 现在在 Foo 上挂载了直接方法 a，输出值为 4

Foo.a();
// 立刻执行了 Foo 上的 a 方法，也就是刚刚定义的，所以
// # 输出 4

let obj = new Foo();
/* 这里调用了 Foo 的构建方法。Foo 的构建方法主要做了两件事：
1. 将全局的 Foo 上的直接方法 a 替换为一个输出 1 的方法。
2. 在新对象上挂载直接方法 a，输出值为 2。
*/

obj.a();
// 因为有直接方法 a，不需要去访问原型链，所以使用的是构建方法里所定义的 this.a,
// # 输出 2

Foo.a();
// 构建方法里已经替换了全局 Foo 上的 a 方法，所以
// # 输出 1
```

数组扁平化+去重+排序

```
Array.from(new Set(arr.flat(Infinity))).sort((a,b)=>{ return a-b})
```

防抖节流(性能优化实现过)

this

this指向常见错误

1. Bind
2. apply call
3. 箭头函数

promise

Promise 是 ES6 新增的语法，解决了回调地狱的问题

async await 一起解决异步问题

简单算法

复杂度

1. 冒泡
2. 快排

二面

更注意广度和深度

js运行机制

JS 在运行的中执行，所以本质上来说JS 中的异步还是同步行为。

不同的任务源会被分配到不同的 Task 队列中，任务源可以分为 微任务（microtask）和 宏任务（macrotask）。在 ES6 规范中，microtask 称为 `jobs`，macrotask 称为 `task`

```
console.log('script start')

shu

new Promise(resolve => {
  console.log('Promise')
  resolve()
})
.then(function() {
  console.log('promise1')
})
.then(function() {
  console.log('promise2')
```

```
    })

    console.log('script end')
    // script start => Promise => script end => promise1 => promise2 => setTimeout
```

以上代码虽然 `setTimeout` 写在 `Promise` 之前，但是因为 `Promise` 属于微任务而 `setTimeout` 属于宏任务，所以会有以上的打印。

微任务包括 `process.nextTick` , `promise` , `Object.observe` , `MutationObserver`

宏任务包括 `script` , `setTimeout` , `setInterval` , `setImmediate` , `I/O` , `UI rendering`

很多人有个误区，认为微任务快于宏任务，其实是错误的。因为宏任务中包括了 `script` ，浏览器会先执行一个宏任务，接下来有异步代码的话就先执行微任务。

所以正确的一次 Event loop 顺序是这样的

1. 执行同步代码，这属于宏任务
2. 执行栈为空，查询是否有微任务需要执行
3. 执行所有微任务
4. 必要的话渲染 UI
5. 然后开始下一轮 Event loop，执行宏任务中的异步代码

通过上述的 Event loop 顺序可知，如果宏任务中的异步代码有大量的计算并且需要操作 DOM 的话，为了更快的界面响应，我们可以把操作 DOM 放入微任务中。

《你不知道的javascript》

实现一个new

```
function _new(fn, ...arg) {
    const obj = Object.create(fn.prototype);
    const ret = fn.apply(obj, arg);
    return ret instanceof Object ? ret : obj;
}

// 实现一个new
var Dog = function(name) {
    this.name = name
}
Dog.prototype.bark = function() {
    console.log('wangwang')
}
Dog.prototype.sayName = function() {
    console.log('my name is ' + this.name)
}
let sanmao = new Dog('三毛')
let simao = _new(Dog, '四毛')
```

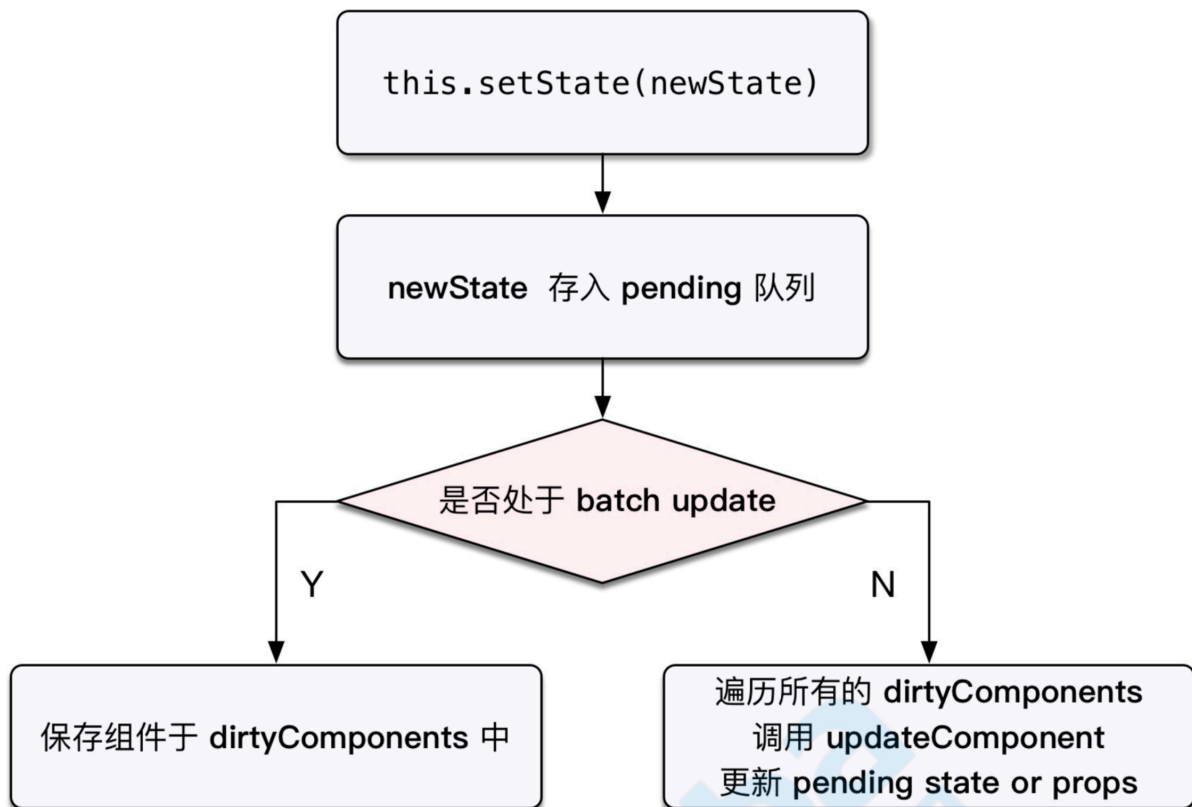
```
sanmao.sayName()  
simao.sayName()
```

Object.defineProperty缺点

1. Object.defineProperty无法监控到数组下标的变化，导致通过数组下标添加元素，不能实时响应；
2. Object.defineProperty只能劫持对象的属性，从而需要对每个对象，每个属性进行遍历，如果，属性值是对象，还需要深度遍历。Proxy可以劫持整个对象，并返回一个新的对象。
3. Proxy不仅可以代理对象，还可以代理数组。还可以代理动态增加的属性。

React中 setState什么时候同步，什么时候异步, 下面代码执行结果

```
class Example extends React.Component {  
  constructor() {  
    super();  
    this.state = {  
      val: 0  
    };  
  }  
  
  componentDidMount() {  
    this.setState({val: this.state.val + 1});  
    console.log(this.state.val);    // 第 1 次 log  
  
    this.setState({val: this.state.val + 1});  
    console.log(this.state.val);    // 第 2 次 log  
  
    setTimeout(() => {  
      this.setState({val: this.state.val + 1});  
      console.log(this.state.val); // 第 3 次 log  
  
      this.setState({val: this.state.val + 1});  
      console.log(this.state.val); // 第 4 次 log  
    }, 0);  
  }  
  
  render() {  
    return null;  
  }  
};
```



在React中，如果是由React引发的事件处理（比如通过onClick引发的事件处理），调用`setState`不会同步更新`this.state`，除此之外的`setState`调用会同步执行`this.state`。所谓“除此之外”，指的是绕过React通过`addEventListener`直接添加的事件处理函数，还有通过`setTimeout/setInterval`产生的异步调用。

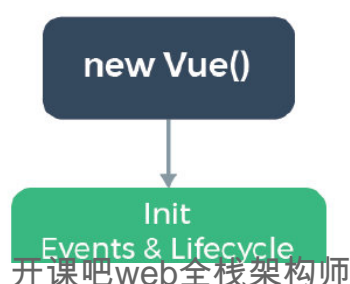
原因：在React的`setState`函数实现中，会根据一个变量`isBatchingUpdates`判断是直接更新`this.state`还是放到队列中回头再说，而`isBatchingUpdates`默认是`false`，也就表示`setState`会同步更新`this.state`，但是，有一个函数`batchedUpdates`，这个函数会把`isBatchingUpdates`修改为`true`，而当React在调用事件处理函数之前就会调用这个`batchedUpdates`，造成的后果，就是由React控制的事件处理过程`setState`不会同步更新`this.state`。

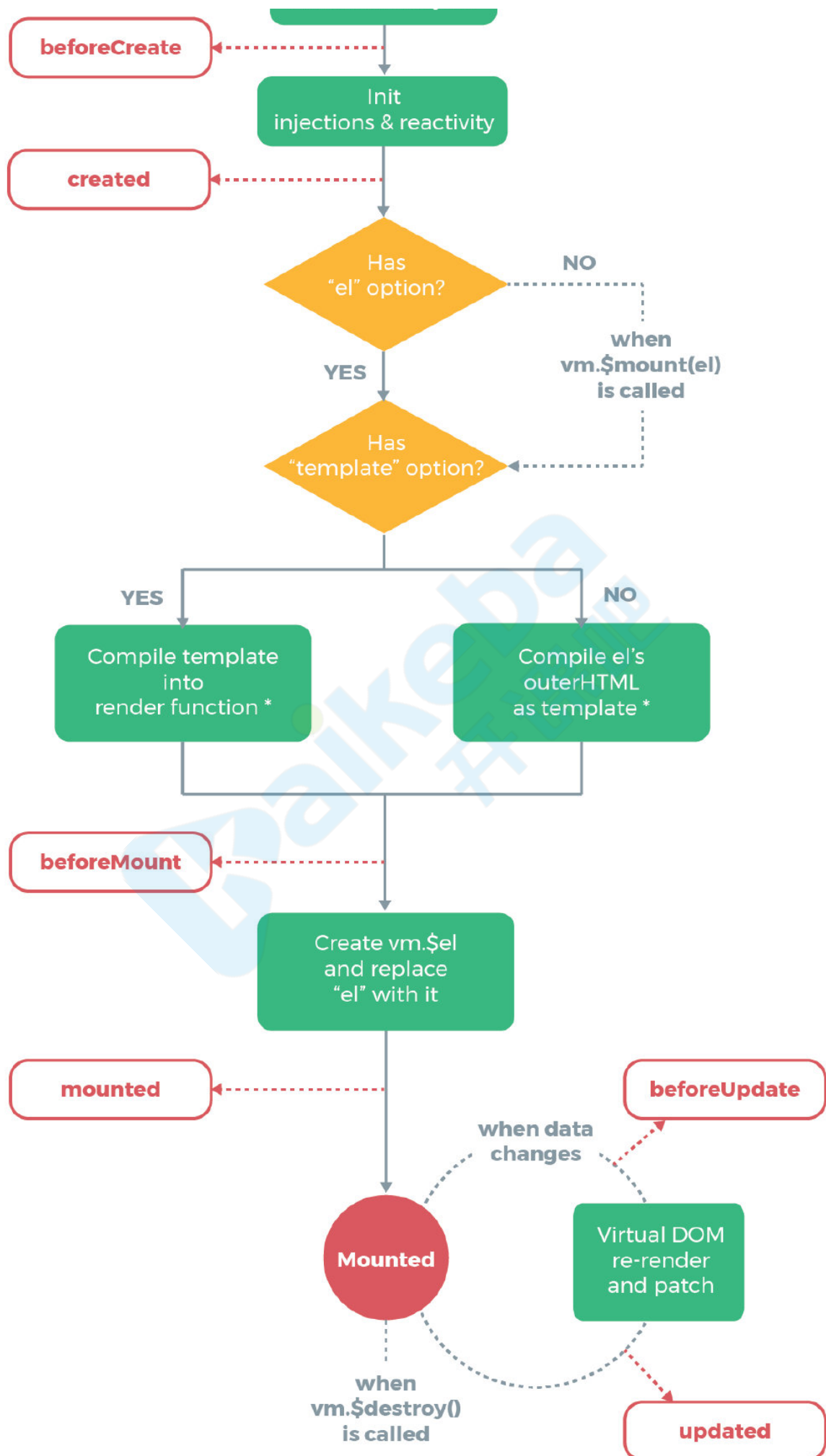
安全

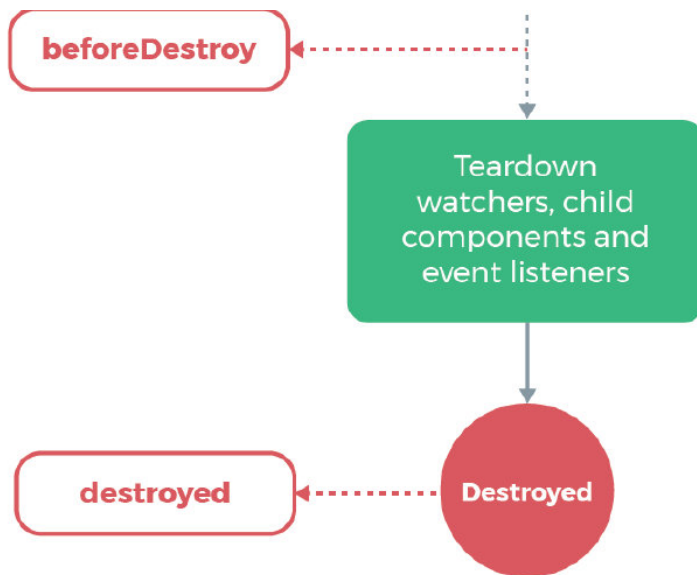
1. xss
2. csrf
3. 密码安全

vue源码

生命周期







* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

重点是响应式 defineProperty

```
class Dep {
  constructor() {
    // 存数所有的依赖
    this.deps = []
  }

  // 在deps中添加一个监听器对象
  addDep(dep) {
    if(this.deps.indexOf(dep) === -1) {
      this.deps.push(dep)
    }
  }

  depend() {
    Dep.target.addDep(this)
  }

  // 通知所有监听器去更新视图
  notify() {
    this.deps.forEach((dep) => {
      dep.update()
    })
  }
}

Dep.target = null
```

```

// 监听器
class Watcher {
  constructor(vm, key, cb) {
    // 在new一个监听器对象时将该对象赋值给Dep.target，在get中会用到
    // 将 Dep.target 指向自己
    // 然后触发属性的 getter 添加监听
    // 最后将 Dep.target 置空
    this.cb = cb
    this.vm = vm
    this.key = key
    this.value = this.get()
  }
  get() {
    Dep.target = this
    let value = this.vm[this.key]
    return value
  }
  // 更新视图的方法
  update() {
    this.value = this.get()
    this.cb.call(this.vm, this.value)
  }
}

class KVue {
  constructor(options) {
    this.$data = options.data
    this.$options = options
    this.$store = options.store || {}

    this.observer(this.$data)
    // 新建一个Watcher观察者对象，这时候Dep.target会指向这个Watcher对象
    // new Watcher()
    // 在这里模拟render的过程，为了触发test属性的get函数
    console.log('模拟render, 触发test的getter', this.$data)
    if(options.created){
      options.created.call(this)
    }
    this.$compile = new Compile(options.el, this)
  }
  observer(value) {
    if (!value || (typeof value !== 'object')) {
      return
    }
    Object.keys(value).forEach((key) => {
      this.proxyData(key)
    })
  }
}

```

```

        this.defineReactive(value, key, value[key])
    })
}
defineReactive(obj, key, val) {

    const dep = new Dep()
    Object.defineProperty(obj, key, {
        enumerable: true,
        configurable: true,
        get() {

            // 将Dep.target (即当前的Watcher对象存入Dep的deps中)

            Dep.target && dep.addDep(Dep.target)
            return val
        },
        set(newVal) {
            if (newVal === val) return
            val = newVal

            // 在set的时候触发dep的notify来通知所有的Watcher对象更新视图
            dep.notify()
        }
    })
}
proxyData(key) {

    Object.defineProperty(this, key, {
        configurable: false,
        enumerable: true,
        get() {

            return this.$data[key]
        },
        set(newVal) {
            this.$data[key] = newVal
        }
    })
}
}
}

```

compile

```

class Compile {
    constructor(el, vm) {
        this.$vm = vm
    }
}

```



```

    this.$el = document.querySelector(el)
    if (this.$el) {
        this.$fragment = this.node2Fragment(this.$el)
        this.compileElement(this.$fragment)
        this.$el.appendChild(this.$fragment)
    }
}

node2Fragment(el) {
    // 新建文档碎片 dom接口
    let fragment = document.createDocumentFragment()
    let child
    // 将原生节点拷贝到fragment
    while (child = el.firstChild) {
        fragment.appendChild(child)
    }
    return fragment
}

compileElement(el) {
    let childNodes = el.childNodes

    Array.from(childNodes).forEach((node) => {
        let text = node.textContent
        // 表达式文本
        // 就是识别{{}}中的数据
        let reg = /\{\{(.*)\}\}/
        // 按元素节点方式编译
        if (this.isElementNode(node)) {
            this.compile(node)
        } else if (this.isTextNode(node) && reg.test(text)) {
            // 文本 并且有{{}}
            this.compileText(node, RegExp.$1)
        }
        // 遍历编译子节点
        if (node.childNodes && node.childNodes.length) {
            this.compileElement(node)
        }
    })
}

compile(node) {
    let nodeAttrs = node.attributes
    Array.from(nodeAttrs).forEach((attr) => {
        // 规定: 指令以 v-xxx 命名
        // 如 <span v-text="content"></span> 中指令为 v-text
        let attrName = attr.name // v-text
        let exp = attr.value // content
        if (this.isDirective(attrName)) {
            let dir = attrName.substring(2) // text

```

```

        // 普通指令
        this[dir] && this[dir](node, this.$vm, exp)
    }
    if(this.isEventDirective(attrName)){
        let dir = attrName.substring(1) // text
        this.eventHandler(node, this.$vm, exp, dir)
    }
    })
}
compileText(node, exp) {
    this.text(node, this.$vm, exp)
}

isDirective(attr) {
    return attr.indexOf('k-') == 0
}

isEventDirective(dir) {
    return dir.indexOf('@') === 0
}

isElementNode(node) {
    return node.nodeType == 1
}

isTextNode(node) {
    return node.nodeType == 3
}

text(node, vm, exp) {
    this.update(node, vm, exp, 'text')
}

html(node, vm, exp) {
    this.update(node, vm, exp, 'html')
}

model(node, vm, exp) {
    this.update(node, vm, exp, 'model')
    let val = vm.exp
    node.addEventListener('input', (e)=>{
        let newValue = e.target.value
        vm[exp] = newValue
        val = newValue
    })
}

update(node, vm, exp, dir) {
    let updaterFn = this[dir + 'Updater']

```

```

    updaterFn && updaterFn(node, vm[exp])
    new Watcher(vm, exp, function(value) {
        updaterFn && updaterFn(node, value)
    })
}

// 事件处理
eventHandler(node, vm, exp, dir) {
    let fn = vm.$options.methods && vm.$options.methods[exp]
    if (dir && fn) {
        node.addEventListener(dir, fn.bind(vm), false)
    }
}

textUpdater(node, value) {
    node.textContent = value
}

htmlUpdater(node, value) {
    node.innerHTML = value
}

modelUpdater(node, value) {
    node.value = value
}
}

```

react解析

1. jsx
2. 虚拟dom
3. setState
4. 单向数据流

三面

面试技巧

工程化

团队合作

领导力

谈薪

职业生涯

个人技能树

1. 内力的修炼
2. git
3. 工程化
4. 算法数据结构

软技能

1. 英语
2. 如何成为一个高手
3. 刻意练习

<https://zhuanlan.zhihu.com/p/23558753>

图谱

面试学习法

面试通过：

练习谈薪，拿到大offer 甚至入职

面试没过：

不要沮丧，当成学习的反馈，查缺补漏

1. 咱们团队后面发展的定位
2. 你觉得我还有哪些地方需要提升
 1. 面试官很贵 面试算是免费聊

你薪资要求多少 XX可以可以

你说OK可以

你的薪资用人部门倒还好，但是hr会稍微控制一下成本 所有都是可以稍微谈一下的

如何谈薪水，大话题

钱是弹出来的，比如你去菜市场

1. 一定要尝试
2. 你被压的薪资，就是HR的kpi
3. 问你的薪资，只是想按照一个合理的涨幅给你
4. 面试你想去的公司钱，先面3-6次，最好有几个offer
5. 最好3-6个月 出去面一次

小黄鸭调试法

技术知识图谱

1. 前端基础技能

1. JS
2. html
3. cssw
4. <JS高级程序设计> 或者犀牛书
5. 《css世界》
6. ES6

1. 阮一峰 或者高程作者那本
2. promise
3. proxy
4. class
5. 解构
6. set
7. 函数式

7. Typescript

1. 强类型

8. Vue

1. 组件化
2. vuex vue-router 研究下原理
3. 性能优化
4. 项目实战
5. 自动化测试
6. vue源码

9. react

1. 类似
2. jsx

2. 其他端(node)

1. node

- 1. 基础
- 2. web开发
- 3. 前后端分离
- 4. 数据库
- 5. 部署
 - 1. Pm2 + nginx
 - 2. docker
- 2. webpack
 - 1. 工程化
 - 2. 脚手架
 - 3. 命令行工具
 - 4. 源码
- 3. 自动化
 - 1. 自动化部署
 - 2. 开发流程
- 4. 行为监控
- 5. 小程序
 - 1. 微信小程序
 - 2. 跨端
 - 1. taro uni-app
 - 2. 云开发
 - 1. serverless
 - 2. 云存储
 - 3. 云函数
 - 4. 云数据库
- 6. app
 - 1. rn
 - 2. flutter
- 3. 工程师基本素养
 - 1. 算法和数据结构
 - 1. 《啊哈算法》
 - 2. 《算法第四版》
 - 2. 网络协议
 - 1. 图解http协议(网络是怎么连起来的)
 - 3. 数据库
 - 1. 高性能mysql
 - 4. 操作系统
 - 1.
 - 5. 软件工程
 - 1. 敏捷开发

- 2. git协作
- 3. <人月神话>
- 4. 构建之法
- 6. 设计模式
 - 1. 常见设计模式
- 4. 软技能

回顾

面试

课堂目标

知识要点

起步

面试准备

JD分析

简历

技术栈

ES6 (javascript)

http协议

如何统计当前网页出现过多少种html标签

请写出打印结果

数组扁平化+去重+排序

防抖节流(性能优化实现过)

this

promise

简单算法

二面

js运行机制

实现一个new
安全
vue源码
react解析
三面
面试技巧
工程化
团队合作
领导力
谈薪
职业生涯
个人技能树
软技能
技术知识图谱

回顾

