# 面试必备技能

# 职场礼仪

## 外在

1. 穿着不需要太正式(不要穿正装)， 也不能太随意
2. 言行举止得当，说话有礼貌，谦虚

## 自我介绍

1. 自我介绍要精简，不能太短，更不能太长，一般不要超过5分钟
2. 快速的体现自己的优点，价值
3. 个人基本信息 + 自己擅长的技术(原生，框架)
4. 不要说技术不相关的内容，面试官也不想听

## 常见问题回答技巧

### **你的优点是什么**

生活和工作及技能的优点都可以说，且要说的很自信

### **你的缺点是什么**

每个人都有缺点，但不要把自己的缺点和盘托出，尽量用缺点去变向的 突出你的优点，比如：我这个人很较真，尤其是在开发的时候，遇到bug不解决不下班

### **离职的原因是什么**

* 不能说上家公司坏话，最无奈的说法就是上家公司黄了，或者拖欠工资好几个月
* 合乎常理，没有太大漏洞
* 离职理由说一条就够了

### **你的期望薪资是多少**

* + 非终面不深谈薪资
  + 只说区间，不说具体数字
  + 底线是不低于当前薪资
  + 非要具体数字，区间取中间值，或者当前薪资的+20%

### **你有什么想问我的吗**

* 没有固定答案，面试官只是想看你和其他面试者有什么不一样的地方
* 可以关心公司的技术栈及发展的方向
* 你入职后的半年时间内可以给公司做哪些贡献
* 公司是否给新员工提供快速融入公司的平台，比如: 帮带计划

### **最快什么时候能入职**

* 一周之内
* 如果公司需求紧急可适当排除困难提前入职

### **三年或者五年的规划是什么**

* 做任何事情提前做规划的人的成功几率远远高于随波逐流的人群
* 比如: 三年之内让自己的技术栈沉淀的更深，会的更多，争取做一个全栈工程师
* 五年之内争取能够发挥自己所有的热度给公司贡献自己最大的力量，有机会的话争取做技术管理岗
* 记住：不想当将军的士兵不是好士兵

# 项目从0到1开发流程

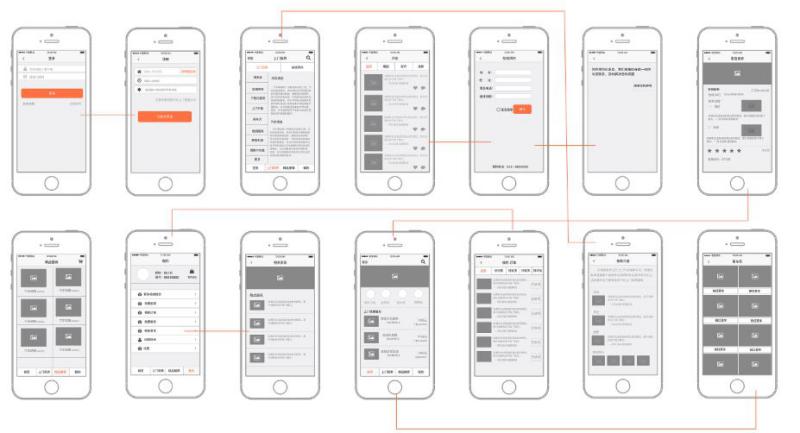
## 图解说明



## 具体流程说明

* 客户，甲方公司提出需求
* 产品经理，项目经理明确需求
* 产品经理处概念图同客户，甲方公司确认需求
* 产品经理出prd图或原型图(通常会开一次大会让参与开发人员明确任务)





* UI设计师出设计(标注)图
* 开发人员进行开发（前端人员，后端开发人员）
* 期间测试人员会对已经开发的部分进行测试，测出BUG后提交到‘bug日志’系统，开发人员每天去的首要任务是查看‘bug日志’有无自己开发的bug，并解决bug
* 在前后端开发完之后会进行‘前后端联调’， 联调指的是将前端的项目部署到后台服务器上，后端项目部署到‘测试环境’上运行项目，查看项目是否有bug并检查项目能够承受的‘高并发’能力
* 项目联调没有问题会邀请部分用户进行测试体验，如果有问题及时处理，如果没有问题就会部署到‘线上环境’， 项目可以被互联网用户访问

# 项目组人员架构

## 成员角色说明

1. 技术总监(CTO)
   1. 负责把控整个项目的开发运行，前后端全通的全能型人才
2. 产品经理(PM)
   1. 负责产品设计
   2. 明确项目需求并传达给对应的前后端开发人员
3. UI设计师
   1. 负责设计页面并出页面效果图供前端人员开发使用
4. 前端开发人员
   1. 负责前端页面开发
   2. 同后端对接
5. 后端开发人员
   1. 负责服务器搭建
   2. 处理应用逻辑
   3. 为前端提供接口数据
   4. 操作数据库
6. 测试工程师
   1. 负责检测前后端项目是否有bug，保证项目尽量完美
   2. 及时提出bug由前后端人员解决
7. 运维工程师
   1. 负责线上环境的安全，部署项目上线
   2. 操作数据库
8. SEO工程师
   1. 负责做网站的整体SEO优化
   2. 百度词条，关键字，外链

## 开发人员配置比例

### **纯甲方公司**

* 10-30人公司
  + 前端人员1人
  + 后端人员3人
  + 测试1人
  + 运维1人
* 30 – 60人公司
  + 前端人员1-2人
  + 后端人员3-5人
  + 测试1人
  + 运维1人
* 60-100人公司
  + 前端人员1-3人
  + 后端人员5-8人
  + 测试1人
  + 运维1人
* 100-300人公司
  + 前端人员3-5人
  + 后端人员10-20人
  + 测试1-2人
  + 运维2-3人
* 300人以上公司
  + 前端人员5 -8人
  + 后端人员10-20人
  + 测试1-2人
  + 运维2-3人

**以上数据仅供参考，不同的公司业务重点不一样，人员配置不一样，但再被问到前后端人员配置的时候应肯定的说出提前准备的答案，前后端人员比例: 1 : 3即可**

### **外包公司**

* 外包公司人员配置不是特别固定，一般根据业务需求按需招人
* 业务重点决定人员配置，前后端人员配置1 : 4即可，说的时候要肯定

# 项目技术架构

## 前端开发所有技术选择

* **原生JavaScript**
  + **目前很少有项目这样选择**
* **jQuery框架 + jQueryUI || BootStrap + jQuery插件**
  + **较老的项目，或者之前成型的大型项目**
* **Angular框架 + AngularUI库 + Angular扩展库** 
  + **时间不会太久的中小型项目**
  + **国内相对较少**
* **React + Antd + React-router + Redux + axios + React Native** 
  + **流行的中大型项目**
* **Vue + Vue-router + Vuex + MintUI || ElementUI + axios || fetch + 插件库(swiper, better-scroll/lodash/moment/等)**
  + **流行的小，中大型项目**
* **Vue(操作数据) + jQuery(操作DOM) + Vue常用扩展库**
  + **这种用法不是太多**

## 后端服务器开发选择

* Java
* Python
* PHP
* .net
* 原生应用
  + Android
  + IOS
  + React Native / Weex / Flutter

## 数据库

* MongoDB
* Mysql
* Oracle

# 个人必备技能

## 手写代码

### **数组去重**

|  |
| --- |
| */\* 方法1: 利用forEach()和indexOf()  说明: 本质是双重遍历, 效率差些 \*/* Array.**prototype**.unique = **function** () {  **const** arr = []  **this**.forEach(item => {  **if** (arr.indexOf(item)===-1) {  arr.push(item)  }  })  **return** arr }  */\* 方法2: 利用forEach() + 对象容器 说明: 只需一重遍历, 效率高些 \*/* Array.**prototype**.unique2 = **function** () {  **const** arr = []  **const** obj = {}  **this**.forEach(item => {  **if** (!obj.hasOwnProperty(item)) {  obj[item] = **true** arr.push(item)  }  })  **return** arr }  */\* 方法3: 利用ES6语法  1). from + Set  2). ... + Set 说明: 编码简洁 \*/* Array.**prototype**.unique3 = **function** () {  **return** Array.from(**new** Set(**this**)) } Array.**prototype**.unique4 = **function** () {  **return** [...**new** Set(**this**)] } |

### 自定义call/apply/bind()

|  |
| --- |
| */\* 自定义函数对象的call方法 \*/* **Function**.**prototype**.call = **function** (obj, ...args) {  *// 如果传入的是null/undefined, this指定为window* obj = obj || **window** *// this(...args)  // 给obj添加一个方法: 属性名任意, 属性值必须当前调用call的函数对象* obj.**tempFn** = **this** *// 通过obj调用这个方法* obj.**tempFn**(...args)  *// 删除新添加的方法* delete obj.**tempFn** }  */\* 自定义函数对象的apply方法 \*/* **Function**.**prototype**.apply = **function** (obj, args) {  *// 如果传入的是null/undefined, this指定为window* obj = obj || **window** *// this(...args)  // 给obj添加一个方法: 属性名任意, 属性值必须当前调用call的函数对象* obj.**tempFn** = **this** *// 通过obj调用这个方法* obj.**tempFn**(...args)  *// 删除新添加的方法* delete obj.**tempFn** }  */\*   自定义函数对象的bind方法  \*/* **Function**.**prototype**.bind = **function** (obj, ...args) {  **return** (...args2) => {  **this**.call(obj, ...args, ...args2)  } } |

### 函数节流与防抖

|  |
| --- |
| *<!--   1. 事件频繁触发可能造成的问题?  1). 一些浏览器事件:window.onresize、window.mousemove等，触发的频率非常高，会造成浏览器性能问题  2). 如果向后台发送请求，频繁触发，对服务器造成不必要的压力    2. 如何限制事件处理函数频繁调用  1). 函数节流  2). 函数防抖   3. 函数节流(throttle)  1). 理解:   在函数需要频繁触发时: 函数执行一次后，只有大于设定的执行周期后才会执行第二次  适合多次事件按时间做平均分配触发  2). 场景：  窗口调整（resize）  页面滚动（scroll）  DOM 元素的拖拽功能实现（mousemove）  抢购疯狂点击（mousedown）    4. 函数防抖(debounce)  1). 理解:  在函数需要频繁触发时: 在规定时间内，只让最后一次生效，前面的不生效。  适合多次事件一次响应的情况  2). 场景：  实时搜索联想（keyup）  文本输入的验证（连续输入文字后发送 AJAX 请求进行验证，验证一次就好） -->* <**script**>  */\* 实现函数节流的函数 \*/* **function** *throttle*(callback, delay) {  **let** start = 0 *// 必须保存第一次点击立即调用* **return function** () { *// 它的this是谁就得让callback()中的this是谁, 它接收的所有实参都直接交给callback()* **console**.log(**'throttle 事件'**)  **const** current = Date.now()  **if** (current - start > delay) { *// 从第2次点击开始, 需要间隔时间超过delay* callback.apply(**this**, arguments)  start = current  }  }  }    */\* 实现函数防抖的函数 \*/* **function** *debounce*(callback, delay) {  **return function** () {  **console**.log(**'debounce 事件...'**)  *// 保存this和arguments* **const** that = **this  const** args = arguments    *// 清除待执行的定时器任务* **if** (callback.**timeoutId**) {  clearTimeout(callback.**timeoutId**)  }  *// 每隔delay的时间, 启动一个新的延迟定时器, 去准备调用callback* callback.**timeoutId** = setTimeout(**function** () {  callback.apply(that, args)  *// 如果定时器回调执行了, 删除标记* delete callback.**timeoutId** }, delay)  }  } </**script**>  <**script**>  */\* 处理点击事件的回调函数 \*/* **function** *handleClick*(event) { *// 处理事件的回调* **console**.log(**'处理点击事件'**, **this**, event)  }   *// document.getElementById('throttle').onclick = handleClick* **document**.getElementById(**'throttle'**).**onclick** = *throttle*(*handleClick*, 2000)  **document**.getElementById(**'debounce'**).**onclick** = *debounce*(*handleClick*, 2000) </**script**> |

### 自定义new与instanceof工具函数

|  |
| --- |
| <**script**>  */\*   功能等同于new的工具函数  \*/* **function** *newInstance*(Fn, ...args) {  *// 创建一个新的对象* **const** obj = {}  *// 执行构造函数* **const** result = Fn.apply(obj, args) *// 相当于: obj.Fn()  // 如果构造函数执行的结果是对象, 返回这个对象* **if** (result instanceof Object) {  **return** result  }  *// 如果不是, 返回新创建的对象* **return** obj  }   **function** *Person*(name, age) {  **this**.**name** = name  **this**.**age** = age  **return** {}  *// return []  // return function (){}  // return 1  // return undefined* }   **const *p*** = **new** *Person*(**'tom'**, 12)  **console**.log(***p***)   **const *p2*** = *newInstance*(*Person*, **'Jack'**, 13)  **console**.log(***p2***)  </**script**>  <**script**>  */\*   功能等同于instanceof的工具函数  \*/* **function** *instanceOf*(instance, Type) {  **while**(instance.\_\_proto\_\_) {  **if** (instance.\_\_proto\_\_===Type.**prototype**) {  **return true** }  instance = instance.\_\_proto\_\_  }  **return false** }  **console**.log(*instanceOf*(***p***, Object), *instanceOf*(***p***, *Person*), *instanceOf*(*instanceOf*, **Function**))  **console**.log(*instanceOf*(***p***, String), *instanceOf*(***p***, **Function**))   </**script**> |

### 事件总线

|  |
| --- |
| */\*  自定义事件总线的模块 \*/* (**function** (window) {  **const** eventBus = {}   */\*   {  add: [callback1, callback2]  delete: [callback3]  }  \*/* **let** callbacksObj = {}    */\*   绑定事件监听  \*/* eventBus.on = **function** (eventName, callback) {  **const** callbacks = callbacksObj[eventName]  **if** (callbacks) {  callbacks.push(callback)  } **else** {  callbacksObj[eventName] = [callback]  }  }    */\*   分发事件  \*/* eventBus.emit = **function** (eventName, data) {  **const** callbacks = callbacksObj[eventName]  **if** (callbacks && callbacks.**length**>0) {  callbacks.forEach(callback => {  callback(data)  })  }  }   */\*   移除事件监听  \*/* eventBus.off = **function** (eventName) {  **if** (eventName) {  **delete** callbacksObj[eventName]  } **else** {  callbacksObj = {}  }  }   window.**eventBus** = eventBus })(**window**) |

### **消息订阅发布**

|  |
| --- |
| */\*  定义一个用来进行消息订阅和发布的对象模块: PubSub \*/* (**function** (window) {  **const** PubSub = {}   */\*   {  add: {  token1: callback1,   token2: callback2  },  update: {  token3: callback3  }  }  \*/* **let** callbacksObj = {} *// 保存所有回调的容器* **let** id = 0 *// 用于生成token的标记  // 1. 订阅消息* PubSub.subscribe = **function** (msgName, callback) {   *// 确定token* **const** token = **'token\_'** + ++id  *// 取出当前消息对应的callbacks* **const** callbacks = callbacksObj[msgName]  **if** (!callbacks) {  callbacksObj[msgName] = {  [token]: callback  }  } **else** {  callbacks[token] = callback  }  *// 返回token* **return** token  }    *// 2. 发布异步的消息* PubSub.publish = **function** (msgName, data) {  *// 取出当前消息对应的callbacks* **let** callbacks = callbacksObj[msgName]  *// 如果有值* **if** (callbacks) {  *// callbacks = Object.assign({}, callbacks)  // 启动定时器, 异步执行所有的回调函数* setTimeout(() => {  Object.values(callbacks).forEach(callback => {  callback(data)  })  }, 0)  }  }   *// 3. 发布同步的消息* PubSub.publishSync = **function** (msgName, data) {  *// 取出当前消息对应的callbacks* **const** callbacks = callbacksObj[msgName]  *// 如果有值* **if** (callbacks) {  *// 立即同步执行所有的回调函数* Object.values(callbacks).forEach(callback => {  callback(data)  })  }  }   */\*  4. 取消消息订阅  1). 没有传值, flag为undefined  2). 传入token字符串  3). msgName字符串  \*/* PubSub.unsubscribe = **function** (flag) {  *// 如果flag没有指定或者为null, 取消所有* **if** (flag===**undefined**) {  callbacksObj = {}  } **else if** (**typeof** flag===**'string'**) {  **if** (flag.indexOf(**'token\_'**)===0) { *// flag是token  // 找到flag对应的callbacks* **const** callbacks = Object.values(callbacksObj).find(callbacks => callbacks.hasOwnProperty(flag))  *// 如果存在, 删除对应的属性* **if** (callbacks) {  **delete** callbacks[flag]  }  } **else** { *// flag是msgName* **delete** callbacksObj[flag]  }    } **else** {  **throw new** Error(**'如果传入参数, 必须是字符串类型'**)  }  }   window.**PubSub** = PubSub })(**window**) |

## 必备技术栈内容

### **原生JavaScript**

* + 操作DOM
  + 面向对象： 闭包，原型，继承，作用域，执行上下文

### **H5，C3新特性**

* + 语义化标签
  + 本地存储
  + Canvas，audio，video
  + C3新增选择器

### **移动端**

* + 移动端适配
  + 移动端常见问题：1物理像素边框，事件点透，多指操作

### **ES6**

* + 箭头函数，promise对象，async函数，class类
  + 加分项: promise实现

### **模块化**

* + ES6
  + Commonjs

### **构建工具**

* + Webpack核心思想：模块化
  + Webpack基本配置: entry，output，module，plugin
  + 加分项: Webpack性能优化

### **框架**

* + jQuery
    - jQuery核心思想
    - 常用方法及特征
  + Vue
    - Vue基本语法
    - Vue-router
    - Vuex
    - MVVM VS MVC
  + React
    - React基本语法
    - React-router
    - Redux

# 项目开发遇到问题及解决方案

## Vue项目开发

## fastClick的300ms延迟解决方案

### **问题剖析:**

300 毫秒延迟的主要原因是解决双击缩放(double tap to zoom)。双击缩放，顾名思义，即用手指在屏幕上快速点击两次，iOS 自带的 Safari 浏览器会将网页缩放至原始比例。 那么这和 300 毫秒延迟有什么联系呢？ 假定这么一个场景。用户在 iOS Safari 里边点击了一个链接。由于用户可以进行双击缩放或者双击滚动的操作，当用户一次点击屏幕之后，浏览器并不能立刻判断用户是确实要打开这个链接，还是想要进行双击操作。因此，iOS Safari 就等待 300 毫秒，以判断用户是否再次点击了屏幕。 鉴于iPhone的成功，其他移动浏览器都复制了 iPhone Safari 浏览器的多数约定，包括双击缩放，几乎现在所有的移动端浏览器都有这个功能

### **解决方案: fastclick库**

引入fastclick库



## 外链样式页面刷新页面样式丢失问题

### **问题剖析:**

项目通常需要一份重置样式文件reset.css, 而该文件通常放在static中通过link引入，

<link rel="stylesheet" href="./static/css/reset.css">， 但在子路由组件中刷新页面会发现重 置样式文件失效，原因是加载路劲错误

### **解决方案:**

去掉点, 将相对路径改成绝对路径

## scoped下修改第三方样式数据(深度选择器)

### **问题剖析:**

在Vue组件化开发中为了避免组件之间样式冲突问题，可在style标签中加上scoped属性，等同于设置样式作用域，只在当前的组件生效，其原理就是给当前的选择器后添加一个随机生成的唯一字段（swiper-container[**data-v-7c66a6a2**]），但如果需要在有scoped的组件中修改外部组件的样式(想修改swiper库的样式),因为scoped的限制问题就无法修改

### **解决方案:**

1. 使用深度选择器: 原生css样式使用 >>>, 在stylus，sass，less中使用 /deep/
2. 语法示例: 外层选择器 /deep/ 修改样式的选择器

|  |
| --- |
| <**style lang="stylus" rel="stylesheet/stylus" scoped**>  **#homeContainer** .**swiper-container  width** 500**px  height** 300**px  border** 1**px solid** .**swiper-wrapper  width** 100%  **height** 100%  .**swiper-slide  width** 100%  **height** 100%   /deep/ .**swiper-pagination-bullet-active  width** 20**px  height** 20**px  background red** </**style**> |

## params传参刷新参数丢失问题

### **问题剖析:**

1. Vue-router进行路由跳转的时候可通过params传参
2. 如果在注册路由的时候没有使用占位符进行注册: ‘/home/:id’,首次路由跳转可以获取params参数，再次刷新页面params数据丢失

### **解决方案:**

**注册路由的时候写好占位符**

|  |
| --- |
| **export default** [  {  **path**: **'/home/:id'**,  **component**: Home,  **name**: **'home'**,  **meta**: {  **isShow**: **true** }  } ]  <**router-link :to="**{name: **'home'**, params: {id: 1}}**"**>home</**router-link**>  <**router-link :to="/home/1"**>home</**router-link**> |

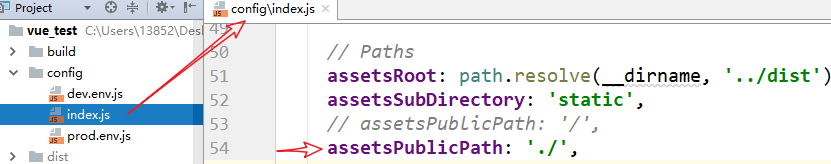
## 打包后访问页面404问题

### **问题剖析:**

1. 打包后生成的dist文件交给后台以后通常会部署到服务器项目的静态资源根目录
2. 如果没有部署到服务器项目静态资源的根目录下访问就是404,

### **解决方案:**

1. 打包之前修改打包的公共资源路径: assetsPublicPath: './'
2. 将绝对路径该成相对路径
3. 但大多情况不需要修改



## map文件过大问题

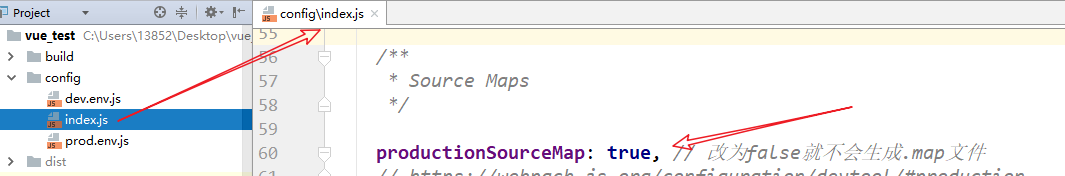
### **问题剖析:**

1. 打包生成的dist文件中的文件是压缩文件，如果发生错误不能很好的提示是哪里出了问题
2. .map文件就是为了提示用户哪里出了问题



### **解决方案**

1. 实际项目上线文件是不需要.map文件的，如果没有去掉无形导致项目体积增大
2. 设置修改: config/index ---🡪 productionSourceMap: false



## 生产环境 & 开发环境解决跨域问题

### **问题剖析:**

1. 跨域: 浏览器为了安全起见推出了同源策略（协议，域名，端口号三者完全一样就是同源，否则就是跨域）
2. 因为现在开发的项目都是前后端完全分离(前端有自己的服务器，如：vue脚手架服务器，同时还有服务器端服务器)的项目，必然存在跨域问题

### **解决方案之开发环境跨域:**

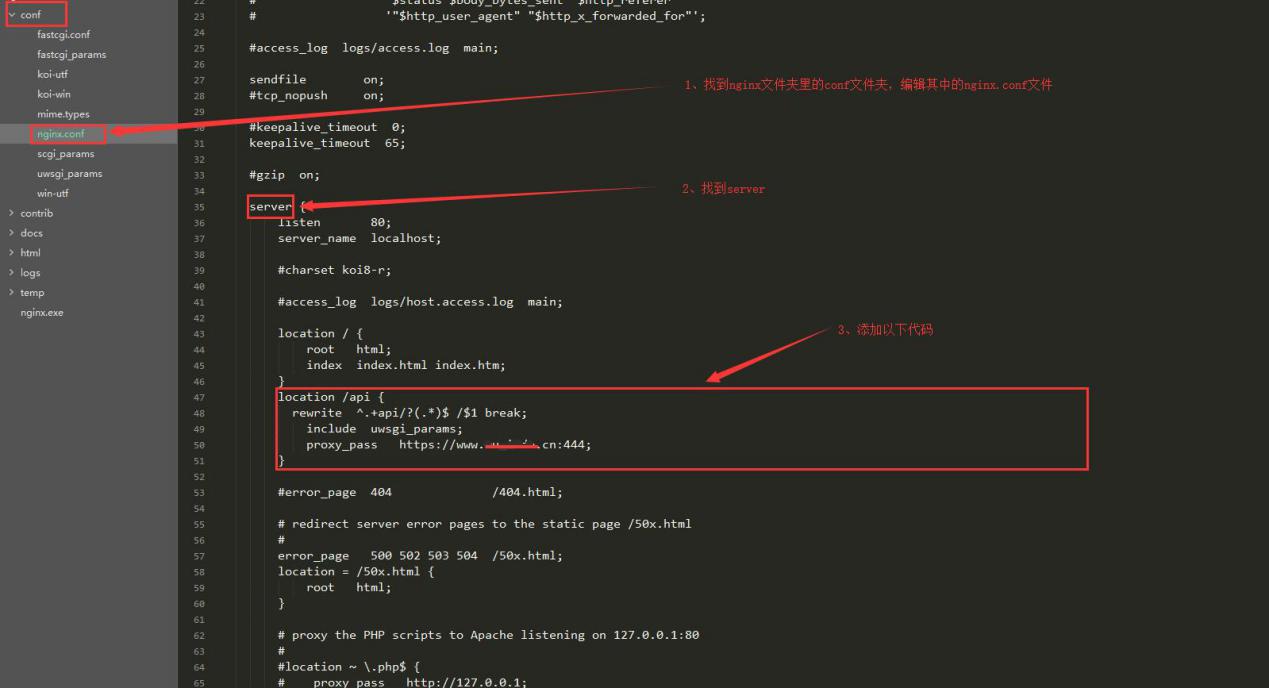
1. 配置正向代理
2. 思路: 页面请求本地服务器，本地服务器通过配置手动转换请求的服务器地址，请求服务器： /api/home --🡪 http://m.you.163.com/home

vue-cli2配置: config/index.js文件中配置

|  |
| --- |
| **proxyTable**: {  **'/api'**: { *// 匹配所有以 '/api'开头的请求路径* **target**: **'http://m.you.163.com'**, *// 代理目标的基础路径* **changeOrigin**: **true**, *// 支持跨域* **pathRewrite**: {*// 重写路径: 去掉路径中开头的'/api'* **'^/api'**: **''** }  } }, |

### **解决方案之生产环境跨域:**

1. 后台使用CORS解决跨域，存在风险，容易被人恶意攻击
2. 使用nginx反向代理
3. 思路: 页面请求发送给nginx服务器，nginx根据配置动态匹配要请求的服务器地址，最终的请求由nginx服务器发出解决跨域



|  |
| --- |
| location /api {  add\_header **'Access-Control-Allow-Origin' '\*'**;  proxy\_pass http:*//fanyi.baidu.com/v2transapi;* } |

## 使用swiper & better-scroll生成多个实例只有一个生效

### **问题剖析:**

1. 使用swiper或者better-scroll的时候需要实例化，一下以swiper为例说明，在new Swiper的时候通常传递一个class类名，其目的是为了让swiper找到页面中对应的DOM对象从而在该DOM对象上产生作用
2. 当一个页面中同时new多个swiper或者better-scroll的时候，使用的class类名一样的话则最先new的那个swiper生效，其他的无效
3. 场景: 当封住一个swiper或者better-scroll组件的时候在一个页面复用，最先使用的那个swiper组件能够使用，其他的失效

### **解决方案:**

**将class换成ref**

## swiper或者better-scroll在页面加载后不生效

### **问题剖析**

1. New的时机太早，在实例化对象的时候页面的数据还没有到达或者没有渲染完成

### **解决方案**

1. 在mounted中的this.$nextTick()中new swiper，因为nextTick保证下一次DOM完全渲染成功以后才会执行
2. 监视轮播图或者滑块数据，通常轮播图或者滑块数据需求发请求获取然后遍历生成，可利用watch监视其数据到达之后再去实例化

## Vue实例的响应式属性 VS 非响应式属性

### **问题剖析**

* 响应式式属性
  + Vue实例初始化之前定义在data中的属性
  + 修改响应式属性之后，会触发视图更新

|  |
| --- |
| data(){  **return** {    obj: { // obj是响应式属性  name: **'kobe' // name也是响应式属性** }  } }, |

* 非响应式属性
  + 在Vue实例初始化之后添加的属性
  + 添加或者修改非响应式属性不会触发视图更新

|  |
| --- |
| data(){  **return** {   obj: {  name: **'kobe'** }  } },  mounted(){  setTimeout(() => {  *// obj.age 刚开始没有定义在data中的obj对象身上，属于非响应式属性，直接添加不会引起页面重新渲染* **this**.obj.age = 30  **this**.$set(**this**.obj, **'age'**, 30)  *// obj本身属于data中的属性，属于响应式属性，修改本身会重新渲染 // this.obj = { // name: 'wade', // age: 38 // }* }, 2000)  } |

* 问题: 添加非响应式属性不会触发视图更新，如何解决

### 解决方案

* 使用Vue.set()
* This.$set(target, ‘propertyName||index’, value), target是目标对象或者数组
* 示例: **this**.obj.age = 30 --🡪 **this**.$set(**this**.obj, **'age'**, 30)
* 补充说明：computed中的计算属性会被缓存，实例化之后就算计算属性内容变化也不会更新视图，除非计算属性依赖的是this实例的响应式属性
* 补充说明2： Vue不允许在根数据对象上添加非 响应式属性

## 需求：路由组件再次被切换回来的时候如何停留在之前页面滚动的位置

### 问题剖析

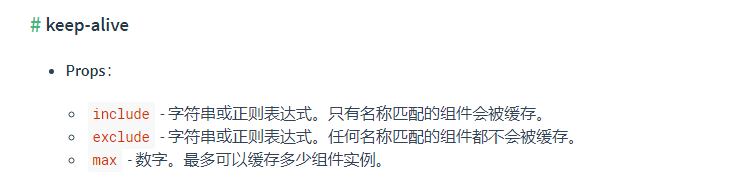
想要保留路由组件之前页面滚动的位置就要保留之前的状态

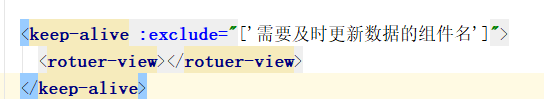
### 解决方案

1. 使用keep-alive缓存路由组件
2. 说明： keep-alive会缓存动态组件，而不是销毁，也就是意味着下次加载缓存组件的时候不会去new新的实例组件，同时状态也会用缓存之前的状态，即使再次加载组件的时候里边的状态有更新

### 追问：如果在keep-alive缓存的动态路由组件中有一个组件需要加载的时候即使更新状态怎么办，因为keep-alive的特性是缓存组件之前的状态

解决方案： 使用keep-alive的props属性：include或者exclude，将需要及时刷新的路由组件排除即可





# 框架必备技能

## Vue框架

## 路由

### 路由跳转

1. 生成路由器 router
2. 注册路由 routes = {path:’路由路径’, components: ‘路由组件’}
3. 使用路由器
   1. new Vue({router})
4. 请求路由
5. <router-link :to="{name: 'home', params: {id: 1}}">home</router-link>

### 路由传参

**1). params:**



**2). query:**



**Params和query区别**

****

**追问： 指定跳转方式能不能交换，如：params传参指定path，query传参指定name**

**答案： params只能用name，query既可以指定name，也可以指定path**

|  |
| --- |
| <**router-link :to='**{**name**: **"Test1"**, **params**: {**id**: **"111"**}}**'**>test1</**router-link**> *<!--无法匹配路由--> <!--<router-link :to='{path: "/hello/test1", params: {id: "111"}}'>test1</router-link>-->* <**router-link :to='**{**name**: **"Test2"**, **query**: {**id**: **"222"**}}**'**>test2</**router-link**> <**router-link :to='/hello/test2'**>test2</**router-link**> |

**3). meta:**



**4). props:**

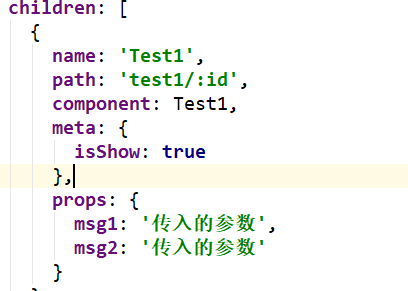
**布尔值模式: 解耦路由组件**

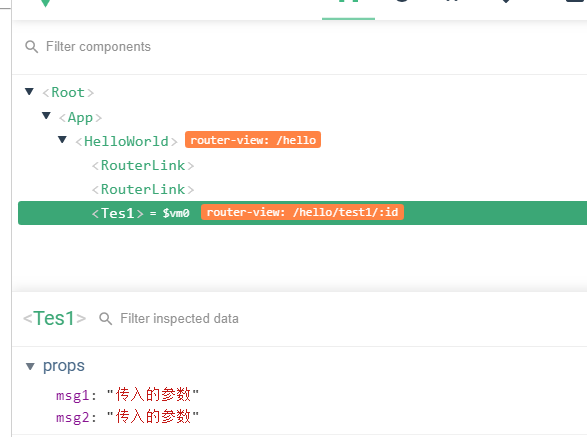
****

****

**对象模式：props静态传参**

****

****

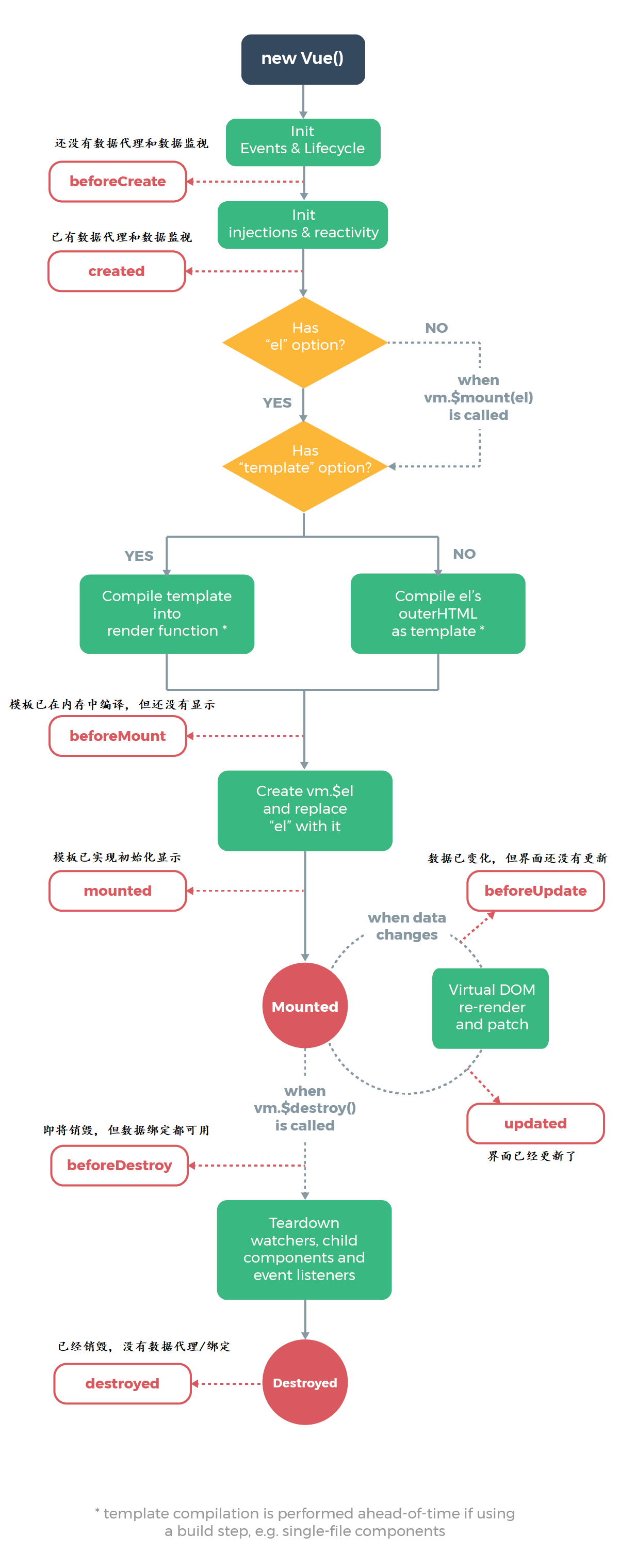
****

**函数模式: props根据不同的路由对象动态传参**

****

**备注： 不管通过哪种props方式传参，记住在组件内部通过props声明接收**

## 生命周期函数



## 组件通信方式

### props

父子组件间通信的基本方式

属性值的2大类型:

一般: 父组件-->子组件

函数: 子组件-->父组件

问题:

隔层组件间传递: 必须逐层传递(麻烦)

兄弟组件间: 必须借助父组件(麻烦)

### vue自定义事件

子组件与父组件的通信方式

用来取代function props

不适合隔层组件和兄弟组件间的通信

### 基于vue自定义事件的事件总线

适合于任何关系的组件间通信

缺点: 管理不够集中

### vuex

多组件共享状态(数据的管理)

组件间的关系也没有限制

功能比eventBus更强大, 更适用于vue项目

### slot插槽

通信是带数据的标签

注意: 标签是在父组件中解析

## 区别单向数据流与双向数据绑定

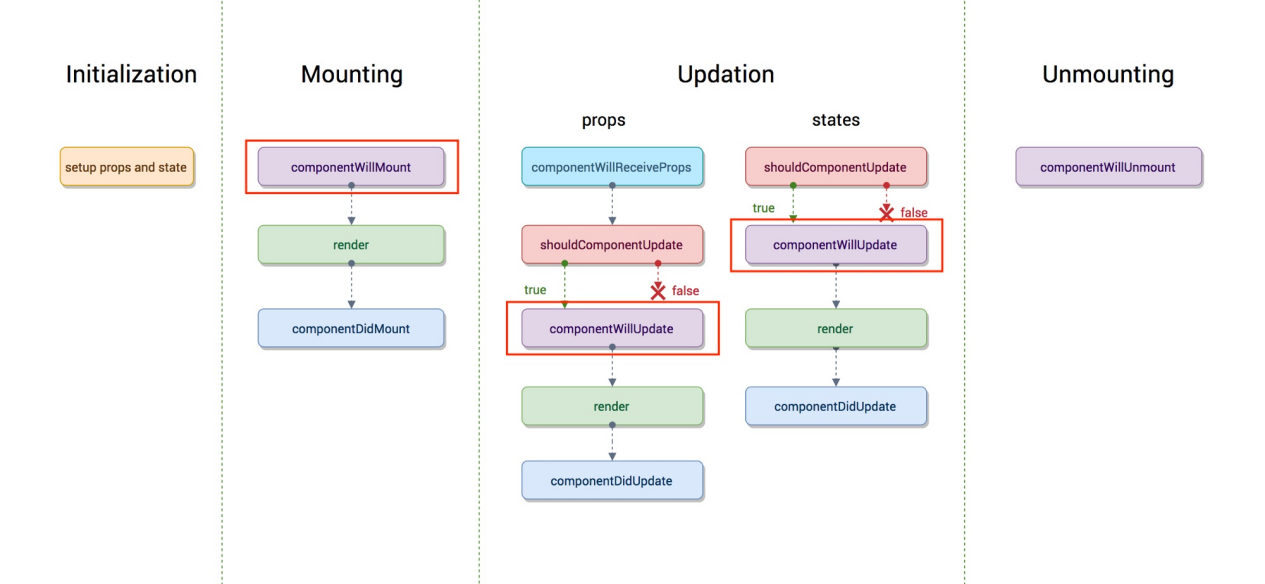
## React框架

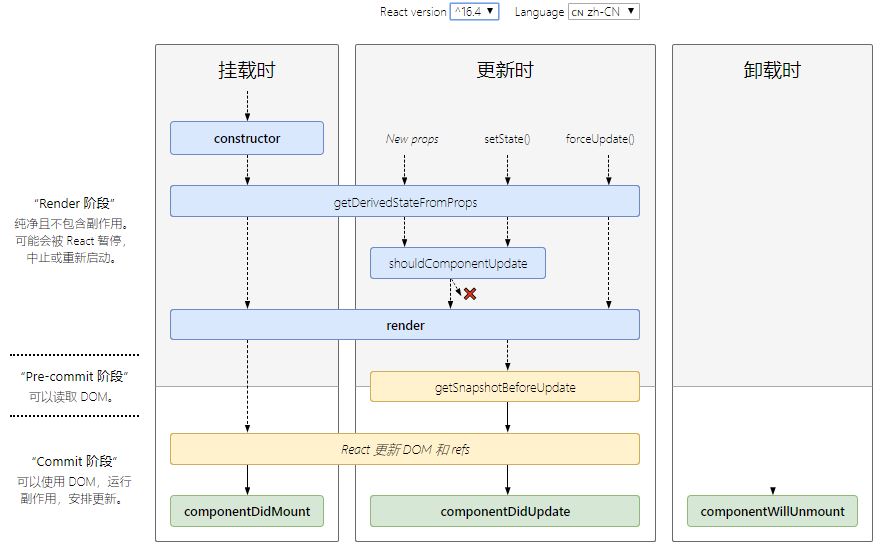
## React核心思想

1. 模块化，组件化
2. JSX语法
   1. 在js中直接写html
   2. 在html中可以写js，需要用{}包起来
3. 组件的3个基本属性
   1. state: 组件状态
   2. props: 组件外部向组件内传递属性数据
   3. refs: 标识组件对象或者真实dom对象

## React生命周期函数

|  |
| --- |
| componentWillMount() {  ***console***.log(**'---------- componentWillMount 组件将要挂载 ------------'**);  *// 在componentWillMount可以获取实例对象中的state，props属性，部分请求可以在此发送  // 要避免把所有的请求操作或者运算量大的代码放在该函数，会影响页面渲染时间  // console.log(this);  // console.log(this.state);  // console.log(this.props);* }  componentDidMount() {  ***console***.log(**'---------- componentDidMount 组件挂载完毕 ------------'**);  *setTimeout*(() => {  **this**.setState({  **msg**: 123  })  }, 2000) }  shouldComponentUpdate(nextProps, nextState) {  ***console***.log(**'---------- shouldComponentUpdate 组件是否应该更新 ------------'**);  *// 该生命周期函数必须返回一个布尔值，来决定是否更新或者不更新 true: 更新，false: 不更新  // 此生命周期函数的意义是用来做性能优化的，当传入的props，state和之前的没有变化或者不需要更新的时候，就return false  // 这样就不会触发React DOM diff算法* ***console***.log(nextProps, nextState);  **return true** }  componentWillReceiveProps(nextProps) {  ***console***.log(**'---------- componentWillUpdate 组件接收新的props值 ------------'**);  }  componentWillUpdate() {  ***console***.log(**'---------- componentWillUpdate 组件将要更新 ------------'**); } componentDidUpdate() {  ***console***.log(**'---------- componentDidUpdate 组件更新完毕 ------------'**); }  componentWillUnmount() {  ***console***.log(**'---------- componentWillUnmount 组件将要卸载 ------------'**); }  render () {  ***console***.log(**'render 渲染'**);  **return** (  <**div**>  App组件。。。  </**div**>  ); } |





## React路由

### 路由跳转

1. 路由器: HashRouter, BrowserRouter
2. 路由: <Route path='/home' component={Home}></Route>
3. 路由重定向: <Redirect to='/home'/>
4. 路由精准匹配： Switch + exact
   1. Switch匹配到指定路由就暂停匹配
   2. Exact要求必须匹配指定的路由，不会模糊匹配

### 路由传参

**1. params传参 推荐使用**

|  |
| --- |
| <**Link to='/home/homeChild/:id'**>home子路由</**Link**> <**Route path='/home/homeChild/1' component=**{HomeChild}></**Route**> |

****

**2. query传参**

|  |
| --- |
| <**Link to='/home/homeChild?id=1'**>home子路由</**Link**> <**Route path='/home/homeChild' component=**{HomeChild}></**Route**> |

****

**3. props传参**

|  |
| --- |
| <**Link to='/home/homeChild'**>home子路由</**Link**> <**Route path='/home/homeChild' render=**{() => <**HomeChild id=**{123}/>}></**Route**> |

****

## 组件通信

### props

* + 父传子: 非函数属性
  + 子传父: 函数属性
  + 隔层组件或兄弟组件传递比较麻烦

### 消息订阅与发布机制(pubsub)

* + 订阅消息: PubSub.subscribe(‘消息名称’, callback)
  + 发布消息: PubSub.publish(‘消息名称’, ‘发布数据’)

### Redux

* + 多用于中大型项目，组件嵌套层级较多，数据较为复杂的场景
  + 核心:
    - 何为 store

Store 是一个 javascript 对象，它保存了整个应用的 state。与此同时，Store 也承担以下职责：

允许通过 getState() 访问 state

通过 dispatch(action) 改变 state

通过 subscribe(listener) 注册 listeners

* + - 何为 action

action是一个纯 js对象，必须有一个 type 属性表明正在执行的 action 的类型。

实质上，action 是将数据从应用程序发送到 store 的有效载荷

* + - 4). 何为 reducer

一个reducer是一个纯函数，该函数以先前的state和一个action作为参数，并返回新的state。

## jQuery

## 核心思想

1. JavaScript函数库，函数化封装
   1. DOM操作封装
   2. 事件处理封装
2. 链式调用
   1. 调用jQuery方法之后可以继续调用jQuery的其他方法:

$('#box').find('p').css('background', 'red');

* 1. 原理: 链式调用的方法都是jQuery对象的方法，所以jQuery方法的返回值都是jQuery对象

1. 读写二合一
   1. jQuery对象的很多方法支持读写二合一，如: .css()
   2. 传一个参数就是读属性，传两个参数就是修改属性

$('#box').find('p').css('background', 'red')

1. 完善的ajax封装
   1. jQuery中封装了很多种发送ajax请求的方法
   2. $.get(), $.post(), $.ajax()

## DOM操作

|  |
| --- |
| <**div id="box"**>  <**p**>111</**p**>  <**p**>222</**p**>  <**p**>333</**p**> </**div**>  *// 内增* **$**(**'#box'**).append(**'<p>444</p>'**) *// 外增* **$**(**'#box'**).before(**'<h1>外部增加的标签</h1>'**);   *// 删除 + 查* **$**(**'#box'**).find(**':first'**).remove(); **$**(**'#box'**).find(**':last'**).empty();  *// 改* **$**(**'#box'**).find(**':last'**).replaceWith(**'<p>替换的标签</p>'**); |

## 事件处理

### 常规事件处理

常规事件处理

|  |
| --- |
| <**div id="box"**>  <**p**>111</**p**>  <**p**>222</**p**>  <**p**>333</**p**> </**div**>  <**input id="inputDOM" type="text"**> <**button id="btn"**>按钮</**button**>  **$**(**'#inputDOM'**).focus(**function** () {  alert(**'获取焦点'**); }); **$**(**'#inputDOM'**).blur(**function** () {  alert(**'失去焦点'**); });  **$**(**'#btn'**).on(**'click'**, **function** () {  alert(**'点击事件'**);  **$**(**'#box'**).hide(); }); |

### 事件委托/派

**概念**

* **定义: 将子元素的事件绑定在父元素身上**
* **原理:冒泡**
* **好处: 减少绑定的次数，新添加的子元素也能享用同类型事件**
* **触发事件的元素: 子元素 === event.target**

**代码实现: on事件，delegate事件**

|  |
| --- |
| <**div id="box"**>  <**p**>111</**p**>  <**p**>222</**p**>  <**p**>333</**p**> </**div**>  **$**(**'#box'**).on(**'click'**, **'p'**, **function** () {  **console**.log(**event**.**target**); }); **$**(**'#box'**).delegate(**'p'**, **'click'**, **function** () {  **console**.log(**event**.**target**); }); |

## ajax请求

|  |
| --- |
| *// get请求* **$**.get(**'/test1'**, {**data1**: **'参数1'**}, **function** (res) {  **console**.log(res); });  *// post请求* **$**.post(**'/test2'**, {**data1**: **'参数2'**}, **function** (res) {  **console**.log(res); }); *// ajax 请求* **$**.**ajax**({  **method**: **'GET'**,  **data**: {**data3**: **'参数3'**},  **url**: **'/test3'**,  success: **function** (res) {  **console**.log(res);  } }); *// 解决跨域： JSONP* **$**.getJSON(**'myUrl?callback=?'**, **function** (res) {  **console**.log(res); }) |

## 小程序

## 程序语法：

* 1. 没有DOM，一切基于组件化
  2. 小程序四个重要的文件: \*.wxml, \*.wxss, \*.js, \*.json

## 小程序发送请求

1. 语法: wx.request()
2. 注意点:
3. 发送的都是https请求
4. 一个域名下最多配置20个域名
5. 能处理的最大并发量是10个

## 小程序本地数据存储

1. 语法: wx.setStorage, wx.getStorage, wx.setStorageSync, wx.getStorageSync
2. 单个 key 允许存储的最大数据长度为 1MB，所有数据存储上限为 10MB。

## 小程序获取openID流程

1. 理解： openId是小程序用户的唯一标识，通常利用openId来验证小程序用户身份，做支付，账户管理等操作
2. openId由腾讯服务器提供的，必须满足： code + appID + appSecret
3. code是登录的临时凭证，通过wx.login()获取
4. 将code发送给当前应用的服务器
5. 当前应用的服务器将code + appID + appSecret发送给腾讯服务器换取:{openId, sessionKey}



## 小程序支付流程:

[https://pay.weixin.qq.com/wiki/doc/api/wxa/wxa\_api.php?chapter=7\_4&index=3#](https://pay.weixin.qq.com/wiki/doc/api/wxa/wxa_api.php?chapter=7_4&index=3)

1. 请求下单支付： 将商品信息发送给商家服务器
2. 获取openId: 从腾讯服务器获取当前用户的唯一标识
3. 统一下单支付: 获取预支付的订单，总价，将预支付订单信息返给客户端
4. 用户确认支付： wx.requestPayment发送请求给腾讯服务器请求扣款
5. 等待支付结果： 客户端等待(展现给用户看的)， 商家服务器等待(更新订单状态)



# 项目性能优化

## Vue & React性能优化

## UI库按需加载(打包)问题

### 8.1.1 问题剖析:

当开发的项目需要引入第三方的UI组件库(antd/mint-ui/element-ui)的时, 需要会打包所有组件的js和css, 而项目中只使用了其中少部分组件. 使用按需打包实现只打包使用的组件, 从而减小打包文件大小

### 8.1.2 解决方案:

下载bebel插件包: **[babel-plugin-component](https://github.com/QingWei-Li/babel-plugin-component)**  
  
添加配置: babel.config.js

plugins: [

["component", { // 使用babel-plugin-component插件包

"libraryName": "mint-ui", // 针对特定的库

"style": true // 相关样式自动引入

}]

]

## 路由组件懒加载 (React, Vue都适用)

### 8.2.1 问题剖析:

1. Vue开发中使用路由跳转页面时，通常会注册多个路由，对应的有多个路由组件
2. 在Vue打包后文件非常之大，如果没有路由懒加载的话，一上来加载所有页面的文件
3. 如果同时加载所有页面的文件内容的话会导致首屏加载显示过慢，甚至白屏，导致用户体验差

### 8.2.2 解决方案:

Table 7没有使用懒加载的方式

|  |
| --- |
| import Category from '../pages/CategoryList/Category/Category.vue';  import RecoFind from '../pages/Recommend/RecoFind/RecoFind.vue';  import RecoSelect from '../pages/Recommend/RecoSelect/RecoSelect.vue';  import Login from '../pages/Profile/Login/Login.vue';  import Register from '../pages/Profile/Register/Register.vue'; |

Table 8使用懒加载的方式

|  |
| --- |
| const Home = () => import('../pages/Home/Home.vue');  const Search = () => import('../pages/Search/Search.vue');  const CategoryList = () => import('../pages/CategoryList/CategoryList.vue');  const Recommend = () => import('../pages/Recommend/Recommend.vue');  const ShopCart = () => import('../pages/ShopCart/ShopCart.vue');  const Profile = () => import('../pages/Profile/Profile.vue'); |

Table 9公共代码

|  |
| --- |
| export default [  {  path: '/home',  component: Home,  meta: {  isShow: true  }  },  {  path: '/search',  component: Search  },  {  path: '/categorylist',  component: CategoryList,  meta: {  isShow: true  },  children: [  {  path: '/categorylist/category',  component: Category,  meta: {  isShow: true  }  },  {  path: '',  redirect: '/categorylist/category'  }  ]  },  {  path: '/recommend',  component: Recommend,  meta: {  isShow: true  },  children: [  {  path: '/recommend/find',  component: RecoFind,  meta: {  isShow: true  }  },  {  path: '/recommend/select',  component: RecoSelect,  meta: {  isShow: true  }  },  {  path: '',  redirect: '/recommend/find'  }  ]  },  {  path: '/shopcart',  component: ShopCart,  meta: {  isShow: true  }  },  {  path: '/profile',  component: Profile,  children: [  {  path: '/profile/login',  component: Login  },  {  path: '/profile/register',  component: Register  },  {  path: '',  redirect: '/profile/login'  }  ]  },  {  path: '/',  redirect: '/home'  }  ] |

## React性能优化值 shouldComponentUpdate

### 8.3.1 问题剖析

1. shouldComponentUpdate在组件的props，state发生变化的时候即将调用componentWillUpdate之前调用
2. 该生命周期函数必须返回一个布尔值，true代表继续更新，false停止本次更新，即不会创建新的虚拟DOM数去进行DOM虚拟算法比较

### 8.3.2 解决方案

**在shouldComponentUpdate中可以获取最新的nextProps，nextState，根据实际情况判断是否需要重新更新，如果不需要则return false;**

## 图片懒加载

### 8.4.1问题剖析

1. 当一个项目图片过多的时候如果一次性加载渲染代价较大，导致用户看到的效果时间延迟

### 8.4.2 解决方案

* Npm install vue-lazyload
* 声明使用: Vue.use(VueLazyLoad, {loading: loadingGIF})
* 组件使用: <img v-lazy="item.primaryPicUrl" alt="新品">

## 封装组件

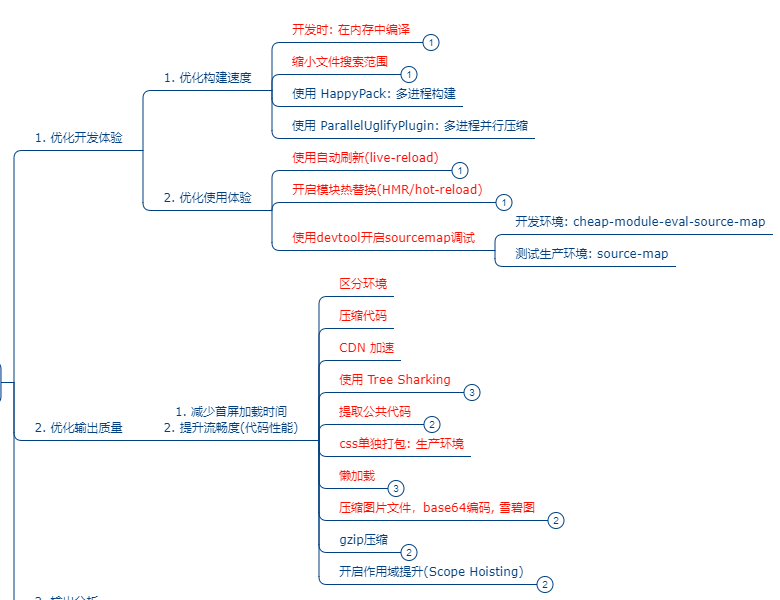
### 8.5.1 问题剖析

1. 一个大型项目中内容足够复杂，在很多页面中部分内容是重复的
2. 如果不做任何处理导致代码冗余，不利于提高性能

### 8.5.2 解决方案

* 抽取封装组件
* 封装组件思想:
  + 组件内部保留相同的内容
  + 不同的内容抽取出来由使用者传递进组件内部，提供组件的复用性
  + 组件内部可提前声明规定用户传递的参数的必要性和数据类型
  + 通过标签属性的形式传递，在组件内部通过props接收

## webpack打包优化



## HTML性能优化

* HTML标签有始终。 减少浏览器的判断时间
* 把script标签移到HTML文件末尾，因为JS会阻塞后面的页面的显示。
* 减少iframe的使用，因为iframe会增加一条http请求，阻止页面加载，即使内容为空，加载也需要时间
* id和class，在能看明白的基础上，简化命名，在含有关键字的连接词中连接符号用'-'，不要用'\_'
* 保持统一大小写，统一大小写有利于浏览器缓存，虽然浏览器不区分大小写，但是w3c标准为小写
* 清除空格，虽然空格有助于我们查看代码，但是每个空格相当于一个字符，空格越多，页面体积越大，像google、baidu等搜索引擎的首页去掉了所有可以去掉的空格、回车等字符，这样可以加快web页面的传输。可以借助于DW软件进行批量删除 html内标签之间空格，sublime text中ctrl+a，然后长按shift+tab全部左对齐，清除行开头的空格
* 减少不必要的嵌套，尽量扁平化，因为当浏览器编译器遇到一个标签时就开始寻找它的结束标签，直到它匹配上才能显示它的内容，所以当嵌套很多时打开页面就会特别慢。
* 减少注释，因为过多注释不光占用空间，如果里面有大量关键词会影响搜索引擎的搜索
* 使用css+div代替table布局，去掉格式化控制标签如：strong，b，i等，使用css控制
* 代码要结构化、语义化
* css和javascript尽量全部分离到单独的文件中

## css性能优化

1. 多利用继承，多个子元素公用的样式，如果该样式能继承的话就写在父元素身上
2. 尽量减少重绘重排的次数
3. 选择器命名规范，通过id寻找更快
4. 动画区域开启独立的图层
   1. 定位： position: absolute/relative
   2. Will-change: xxx;
5. Animation, transform等配合will-change属性使用
   1. will-change 为web开发者提供了一种告知浏览器该元素会有哪些变化的方法，这样浏览器可以在元素属性真正发生变化之前提前做好对应的优化准备工作。 这种优化可以将一部分复杂的计算工作提前准备好，使页面的反应更为快速灵敏。
6. 合并、压缩你的css文件，减少http请求，可以借助工具或者自动化构建。
7. 使用CSS sprite来处理你的图片

## JS性能优化

1. 合并压缩js
2. 减少对DOM的操作，避免重绘重排
3. 减少请求的个数，节省网络资源
4. 封装功能函数实现复用
5. 循环语句中避免定义变量
6. 慎用闭包
7. 函数节流，函数防抖

# 经典问题解析

## MVC和MVVM的区别

### MVC

1. 图解



**Model（模型）**是应用程序中用于处理应用程序数据逻辑的部分。  
　　通常模型对象负责在数据库中存取数据。

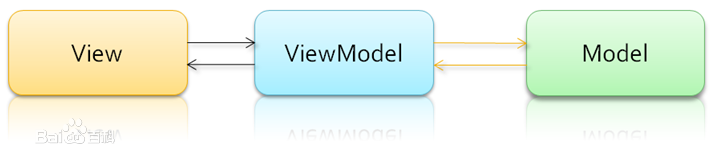
**View（视图）**是应用程序中处理数据显示的部分。  
　　通常视图是依据模型数据创建的。

**Controller（控制器）**是应用程序中处理用户交互的部分。  
　　通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据。

最典型的MVC就是JSP + [servlet](https://baike.baidu.com/item/servlet" \t "https://baike.baidu.com/item/MVC%E6%A1%86%E6%9E%B6/_blank) + [javabean](https://baike.baidu.com/item/javabean" \t "https://baike.baidu.com/item/MVC%E6%A1%86%E6%9E%B6/_blank)的模式

### MVVM

1. M： Model模型, 也就是包含数据的js对象(只包含属性, 不包含方法)
2. V： View视图，动态显示模型对象中的数据显示界面
3. VM： ViewModel视图模型, 本质是一个绑定器, 通过vm读取model中的数据显示到view上, 同时view输入数据改变, vm也可以将输入数据保存到model中
4. 图解



MVVM的优势:

不用亲自操作DOM, 数据是响应式的, 一旦数据变化

## 两个独立html页面怎么通信

### 问题描述:

如：淘宝A，B页面，A为商品页，B为购物车页，两个页面同时打开,在A页面点击商品添加至购物车，切换到B页面购物车怎么显示该商品信息

### 解决方案

1. 利用localStroage，A页面将数据存入localStroage，B页面读取localStorage中的数据
2. 关键点: B页面如何知道A页面什么时候存数据到localStorage
3. localStorage中有个‘storage’事件
4. window.addEventListener(‘storage’, callback)，A页面存数据，B页面监听事件
5. 代码示例

A页面

|  |
| --- |
| <**button id="add"**>添加一件商品到购物车</**button**>  <**button id="del"**>删除一件商品从购物车</**button**> <**script type="text/javascript"**>  **window**.**onload** = **function** () {  document.getElementById(**'add'**).**onclick** = **function** () {  *count*(**true**)  }   document.getElementById(**'del'**).**onclick** = **function** () {  *count*(**false**)  }   **function** *count*(flag) {  **let** count = **localStorage**.getItem(**'count'**)?***JSON***.parse(**localStorage**.getItem(**'count'**)).**count**:0;  **if**(flag){  count += 1;  }**else** {  **if**(!count) **return**;  count -= 1;  }  **let** countObj = {  count  }  **localStorage**.setItem(**'count'**, ***JSON***.stringify(countObj))  }  } |

B页面

|  |
| --- |
| <**div class="cart"**>  购物车产品数量:  <**span class="count"**>0</**span**> </**div**> <**script type="text/javascript"**>  document.querySelector(**'.count'**).**textContent** = **localStorage**.getItem(**'count'**)?***JSON***.parse(**localStorage**.getItem(**'count'**)).**count**:0;  **window**.addEventListener(**'storage'**, **function** (event) {  **console**.log(***JSON***.parse(event.**newValue**).**count**);  document.querySelector(**'.count'**).**textContent** = ***JSON***.parse(event.**newValue**).**count** }) </**script**> |

## 在Vue中某一个路由组件在滑屏的时候切换到其他组件，再切换回来如何保持页面在之前滑动的位置

### 问题描述

思考： 要页面保存在之前滑动的位置就要保存页面之前的状态

### 解决方案

1. 使用keep-alive去缓存当前的路由组件

2. 缓存路由组件的实例对象，下载加载当前页面的时候不去重新new当前组件的实例

9.3.3 追问： 如果使用keep-alive后某一个路由组件需要及时更新状态怎么办

|  |
| --- |
| <**keep-alive :exclude="**[**'Query'**]**"**>  <**router-view**></**router-view**> </**keep-alive**>  activated(){  ***console***.log(**'--------- activated --------'**); }, deactivated(){  ***console***.log(**'--------- deactivated --------'**); } |

## 在React或Vue中在子组件能不能修改props数据

### 问题描述

1. 父组件通过标签属性的形式向子组件传递数据

2. 传递的数据类型： 基本数据类型，引用数据类型

### 答案

1. React不能修改，修改会报错

2. Vue中可以修改，会有警告

### 追问： Vue子组件修改props的数据会不会影响父组件的状态值

1. 如果传递的基本数据类型，不会影响
2. 如果传递的引用数据类型，会影响

## 浏览器的同源策略

### 同源

协议、域名、端口都相同

### 同源策略

是浏览器最核心也最基本的安全功能，主要体现在同源策略会限制来自不同源的文档和脚本对当前源的文档数据的读取或设置某些属性，是用于隔离潜在恶意文件的重要安全机制

### 同源策略控制了不同源的交互

1. 跨域写： 默认允许，比如<a>、重定向和表单提交(因为表单提交不需要反馈数据)
2. 跨域资源嵌入： 默认允许, 比如: <script> / <link> / <img> / <video> / <audio> / <iframe>
3. 跨域读: 默认允许, 比如ajax请求, 读取跨域的Cookie, localStorage, sessionStorage, IndexDB

## XSS攻击

### [什么是XSS攻击](https://blog.tonyseek.com/post/introduce-to-xss-and-csrf/)

跨站脚本攻击(Cross Site Scripting), 因简称是CSS同样式表css重合，所以改名叫XSS

### 攻击原理

是注入攻击的一种。其特点是不对服务器端造成任何伤害，而是通过一些正常的站内交互途径，将包含js脚本的内容提交后台, 后台没有进行验证过滤返回给浏览器端显示。

例如发布评论，提交含有 JavaScript 的内容文本, 这时服务器端如果没有过滤或转义掉这些脚本，作为内容发布到了页面上，其他用户访问这个页面的时候就会运行这些脚本

场景：用户评论：

**正常情况：**

用户A提交评论内容为 字符串‘我要好好学习，天天向上’，服务器获取参数

在返回的评论数据中添加该字符串，前后获取遍历显示在页面，结果：<p>我要好好学习，天天向上</p>

**xss攻击:**

提交评价内容为: ‘<a href='东京热'>我要好好学习，天天大保健</a>’服务器获取参数在返回的评论数据中添加该字符串，前后获取遍历显示在页面，结果：<p><a href='xxx'>我要好好学习，天天大保健</a></p>

### 解决方案

如果提交的内容不能是html, 后台需要进行检查报错或去除标签结构, 只留下纯文本

如果提交的内容可以包含html, 后台需要读取出其文本内容再组织成html结构

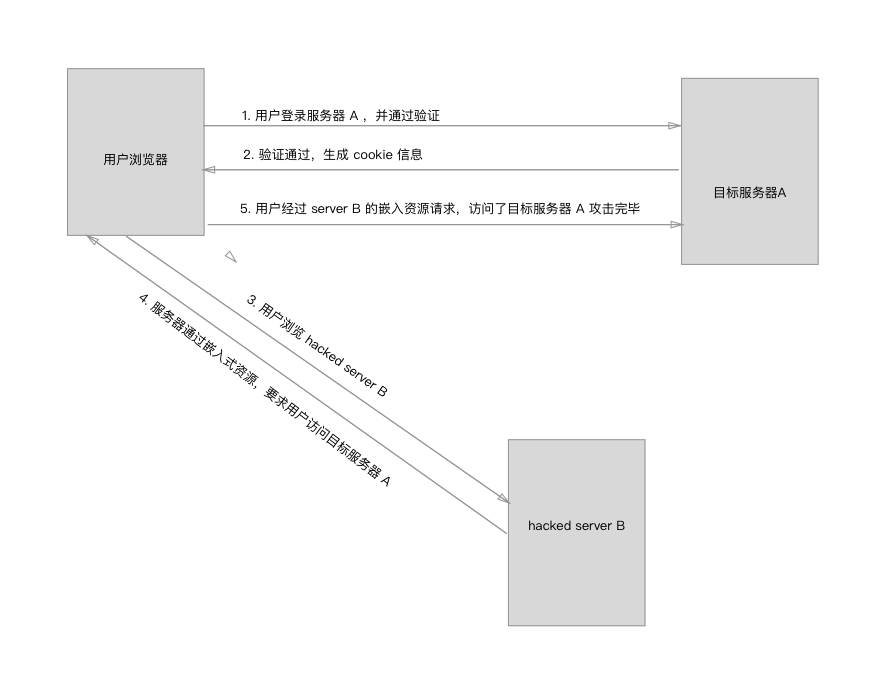
## XSRF/CSRF攻击

### [什么是XSRF](https://www.jianshu.com/p/0e52c58cf93f)

CSRF（Cross Site Request Forgery, 跨站域请求伪造）也称 XSRF, 是一种网络攻击方式

### CSRF 攻击的原理

1. 就是利用由于浏览器的同源策略对嵌入资源不做限制的行为进行跨站请求伪造的
2. 原理分析图



### 解决方案

1. referer请求头
2. Token
3. userId

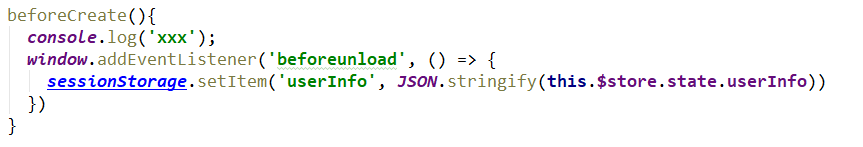
## Vue单页面应用中刷新页面，Vuex数据丢失

### Vuex特点

1. Vuex数据保存在运行内存中，vue实例初始化的时候为其分配内存
2. 当刷新页面的时候重新初始化Vue实例，所以重新为Vuex分配内存导致之前保存的数据丢失

### 如何解决

1. Vuex的数据都是每次组件加载时候动态请求获取数据保存
   1. 优点： 保证数据不会丢失
   2. 缺点: 性能差，因为网络问题可能有网络延迟
2. 将Vuex中的数据每次同步更新保存到sessionStorage中
   1. 优点: 每次页面刷新后从sessionStorage中获取保存的数据，不会丢失
   2. 缺点: state中的数据是动态的，就需要一直要同步更新到sessionStorage中，性能差
3. 在页面刷新之前获取Vuex的数据，将数据保存在sessionStorage中，页面加载后从sessionStorage中获取
   1. 优点: 减少动态更新sessionStorage的次数，性能好
   2. 重点: window.beforeunload()



### 问题截图示例



Figure 1数据丢失

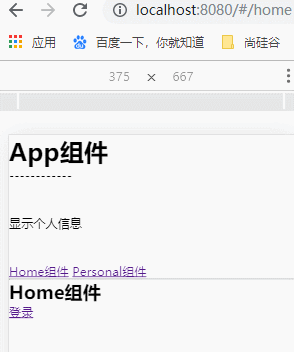


Figure 2数据不丢失