# 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, 干净利落即可
2. 言行举止得当，说话有礼貌，谦虚中带自信

## 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) + 项目
4. 不要说太多技术不相关的内容，面试官也不想听

## 常见问题回答技巧

### 你的优点是什么

1. 生活和工作及技能的优点都可以说，且要说的很自信
2. 比如: 喜欢研究技术, 技术的底子还是可以的
3. 比如: 性格比较开朗/随和, 与同事之间的关系都还处得不错, 这样也有利于项目开发

### 你的缺点是什么

1. 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，
2. 比如: 我不太擅长主动跟陌生人唠嗑, 但聊技术/聊需求没任何问题
3. 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

1. 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
2. 合乎常理，没有太大漏洞
3. 离职理由说一条就够了

### 你的期望薪资是多少

1. 非终面不要主动深谈薪资
2. 只说区间，不说具体数字
3. 底线是不低于当前薪资
4. 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

1. 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
2. 可以关心公司的技术栈及发展的方向
3. 你入职后的半年时间内可以给公司做哪些贡献
4. 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

1. 一周之内
2. 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

1. 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
2. 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
3. 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
4. 记住：不想当将军的士兵不是好士兵

# 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

1. 10-30人公司
   1. 前端人员1人
   2. 后端人员3人
   3. 测试1人
   4. 运维1人
2. 30 – 60人公司
   1. 前端人员1-2人
   2. 后端人员3-5人
   3. 测试1人
   4. 运维1人
3. 60-100人公司
   1. 前端人员1-3人
   2. 后端人员5-8人
   3. 测试1人
   4. 运维1人
4. 100-300人公司
   1. 前端人员3-5人
   2. 后端人员10-20人
   3. 测试1-2人
   4. 运维2-3人
5. 300人以上公司
   1. 前端人员5 -8人
   2. 后端人员10-20人
   3. 测试1-2人
   4. 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

1. 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
2. 业务重点决定人员配置，前后端人员配置1 : 3左右即可，说的时候要肯定

# 项目从0到1开发流程

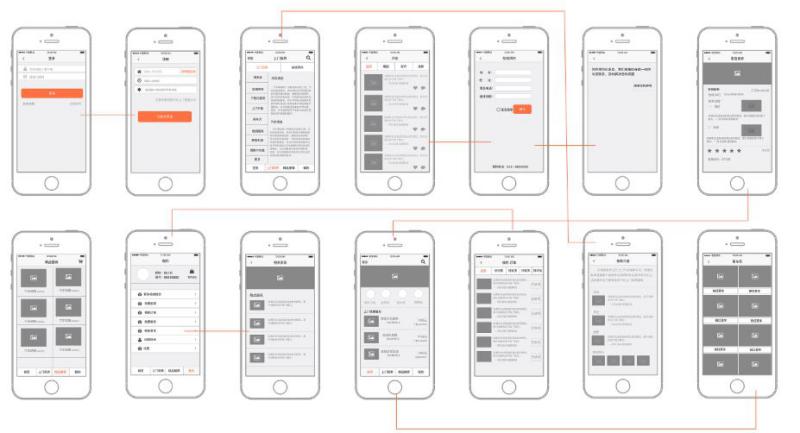
## 图解说明



## 具体流程说明

1. 客户，甲方公司提出需求
2. 产品经理，项目经理明确需求
3. 产品经理出概念图同客户(甲方公司)确认需求
4. 产品经理出prd图/原型图(通常会拉上相关人员开会讨论确定需求)





1. UI设计师出设计(标注)图
2. 开发人员进行开发（前端人员，后端开发人员）
3. 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
4. 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
5. 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

# 项目技术架构

## 前端开发所有技术选择

1. 原生JavaScript

目前很少有项目这样选择

1. jQuery框架 + jQueryUI || BootStrap + jQuery插件

较老的项目，或者之前成型的大型项目

1. Angular框架 + AngularUI库 + Angular扩展库
   1. 时间不会太久的中小型项目
   2. 国内相对较少
2. React + Antd + React-router + Redux + axios + React Native

流行的中大型项目

1. Vue + Vue-router + Vuex + MintUI || ElementUI + axios + 插件库(swiper, better-scroll/lodash/date-fns/等)

流行的小，中大型项目

1. 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

1. Java
2. Python
3. PHP
4. .net
5. 原生应用
   1. Android
   2. IOS
   3. React Native / Weex / Flutter

## 数据库

1. MongoDB
2. Mysql
3. Oracle

# 必备技术栈概要

## 原生JavaScript

1. 面向对象： 作用域(链)，原型(链)，闭包，执行上下文(栈)，继承，
2. 操作DOM

## H5，C3新特性

1. 语义化标签
2. 本地存储
3. Canvas，audio，video
4. C3新增选择器

## 移动端

1. 移动端适配
2. 移动端常见问题：1物理像素边框，事件点透，多指操作

## ES6+

1. 箭头函数，promise，async/await，class
2. 加分项: promise深入使用与自定义实现

## 模块化

1. ES6
2. Commonjs

## 构建工具

1. Webpack核心思想：模块化打包
2. Webpack基本配置: entry, output, module&rules, plugins, devServer, devtool, resolve
3. 打包各种资源: JS / CSS/img/html/...
4. 加分项: 自定义webpack打包环境, Webpack性能优化

## 前后台交互

1. 原生ajax请求 xhr/fetch
2. ajax请求跨域问题
3. ajax请求库axios的理解和使用
4. axios的二次封装
5. 加分项: axios的核心实现源码分析

## 框架

1. jQuery
   1. 整体理解
   2. jQuery核心函数
   3. jQuery核心对象
2. Vue
   1. Vue基本核心语法
   2. Vue-router
   3. Vuex
   4. MVVM实现原理
3. React
   1. React基本核心语法
   2. React-router
   3. Redux

# JS异步编程

## 前置知识

1. JS语言的执行环境是"单线程", 也就是我们写的所有js代码都是在一个线程(主线程)上执行
2. 理解单线程:
   1. 就是指一次只能完成一件任务。
   2. 如果有多个任务，就必须排队，前一个任务完成，再执行后面一个任务，以此类推
3. js执行任务的2种模式
   1. 同步（Synchronous）
      1. 后一个任务等待前一个任务结束，然后再执行，
      2. 程序的执行顺序与任务的排列顺序是一致的、同步的
   2. 异步（Asynchronous）
      1. 每一个任务有一个或多个回调函数（callback），前一个任务结束后，不是执行后一个任务，而是执行回调函数，
      2. 后一个任务则是不等前一个任务结束就执行，所以程序的执行顺序与任务的排列顺序是不一致的、异步的。
4. 为什么需要异步JS?
   1. 浏览器端:
      1. 耗时很长的操作都应该异步执行，避免浏览器失去响应
      2. 给时机让浏览器能去更新界面, 响应用户操作
   2. Node端:
      1. "异步模式"是唯一的模式，执行环境是单线程的
      2. 处理请求的回调函数/数据库操作/文件读写操作都必须是异步处理, 否则处理多个请求时很快就会失去响应
5. JS常用的异步编码方式
   1. 回调函数
   2. 事件机制
   3. 消息订阅与发布 / 全局事件总线
   4. Promise
   5. async & await

## 回调函数

1. 异步编程最基本的方法, 所有模式都是在此基础上进行封装扩展而来
2. 回调函数的执行模式:
   1. 异步执行
   2. 同步执行
3. 异步回调的缺点:
   1. 不利于代码的阅读和维护，各个部分之间高度耦合。
   2. 容易导致回调地狱问题

## 事件监听

1. 事件驱动模式:
   1. 任务的执行不取决于代码定义的顺序，而取决于某个事件是否发生
   2. 事件监听函数定义时不会执行, 只有当事件发生(分发事件)后才执行
2. 分类:
   1. 原生DOM事件
   2. 自定义事件
3. 操作:
   1. 在某个元素/组件对象上绑定特定事件监听
   2. 在某个元素/组件对象上分发事件
4. 特点:
   1. 针对某个DOM元素绑定监听和分发事件
   2. 针对某个组件对象绑定监听和分发事件

## 消息订阅/发布与全局事件总线

1. 消息订阅/发布:

订阅全局消息

发布全局消息

1. 全局事件总线:

绑定事件监听

分发事件

1. 特点:

分发事件后, 所有同名的事件监听回调都会调用

## Promise

1. 实现异步编程新的通用解决方案
2. 相对于纯回调的优势
   1. 指定异步回调函数的方式更灵活(可以在启动异步任务后,甚至可以在任务完成后)
   2. 通过then的链式调用解决回调地狱的问题
3. 不足: 还需要指定回调函数

## async/await

1. 基于promise的语法糖, 简化了promise对象的使用(不再使用回调函数编码)
2. 以同步编码方式实现的异步流程
3. 是js异步编程的终极解决方案(基本上可以这样说)

## JS事件循环机制

1. js是单线程运行的
2. js的回调函数可以异步执行, 也可以同步执行
3. js通过event-loop机制实现了js的单线程异步执行
   1. JS引擎解析执行js代码总是在主线程执行(WebWorks除外)
   2. 浏览器有在分线程执行的对应管理模块(浏览器是多线程执行的)
      1. 定时器
      2. DOM事件监听
      3. ajax请求
      4. Promise
      5. MutationObserver
   3. JS引擎有专门的回调队列, 缓存待执行的回调函数
      1. 宏队列
      2. 微队列

## Promise深入理解

1. 如何改变promise的状态?
2. 一个promise指定多个成功/失败回调函数, 都会调用吗?
3. promise.then()返回的新promise的结果状态由什么决定?
   1. 返回一个非promise值 resolved
   2. 抛出异常 rejected
   3. 返回一个promise
      1. 成功了 resolved
      2. 失败了 rejected
      3. pending pending
4. 改变promise状态和指定回调函数谁先谁后?
5. promise如何串连多个操作任务?
6. promise异常传(穿)透?
7. 中断promise链

## 自定义Promise

1. 定义整体结构
2. Promise构造函数的实现
3. promise.then()/catch()的实现
4. Promise.resolve()/reject()的实现
5. Promise.all/race()的实现
6. Promise.resolveDelay()/rejectDelay()的实现
7. ES6 class版本

# 前后台交互

## 接口的理解/测试/模拟

1. 前后台交互接口: 请求地址 / 请求方式 / 请求参数格式 / 响应数据格式
2. 测试接口: 使用postman
3. 模拟(mock)接口: 使用mockjs / json-server / webpack / node&express

## ajax请求

1. 区别ajax请求与一般的HTTP请求
2. 原生ajax请求: XHR与fetch
3. XHR的基本编码流程

## 解决ajax跨域问题的常见技巧

1. JSONP： json with padding（垫子）
2. CORS: 服务器端设置响应头： Access-Control-Allow-Origin: www.taobao.com
3. 代理

## axios的理解和使用

### axios的特点

* 1. 基于xhr/http包 + promise的异步ajax请求库
  2. 浏览器端/node端都可以使用
  3. 支持请求／响应拦截器
  4. 支持请求取消
  5. 请求/响应数据转换
  6. 批量发送多个请求

### axios的常用语法

axios(config): 通用/最本质的发任意类型请求的方式

axios(url[, config]): 可以只指定url发get请求

axios.request(config): 等同于axios(config)

axios.get(url[, config]): 发get请求

axios.delete(url[, config]): 发delete请求

axios.post(url[, data, config]): 发post请求

axios.put(url[, data, config]): 发put请求

axios.create([config]): 创建一个新的axios(它没有下面的功能)

axios.defaults.xxx: 请求的默认全局配置

axios.interceptors.request.use(): 添加请求拦截器

axios.interceptors.response.use(): 添加响应拦截器

axios.Cancel(): 用于创建取消请求的错误对象

axios.CancelToken(): 用于创建取消请求的token对象

axios.isCancel(): 是否是一个取消请求的错误

axios.all(promises): 用于批量执行多个异步请求

axios.spread(): 用来指定接收所有成功数据的回调函数的方法

## axios的二次封装

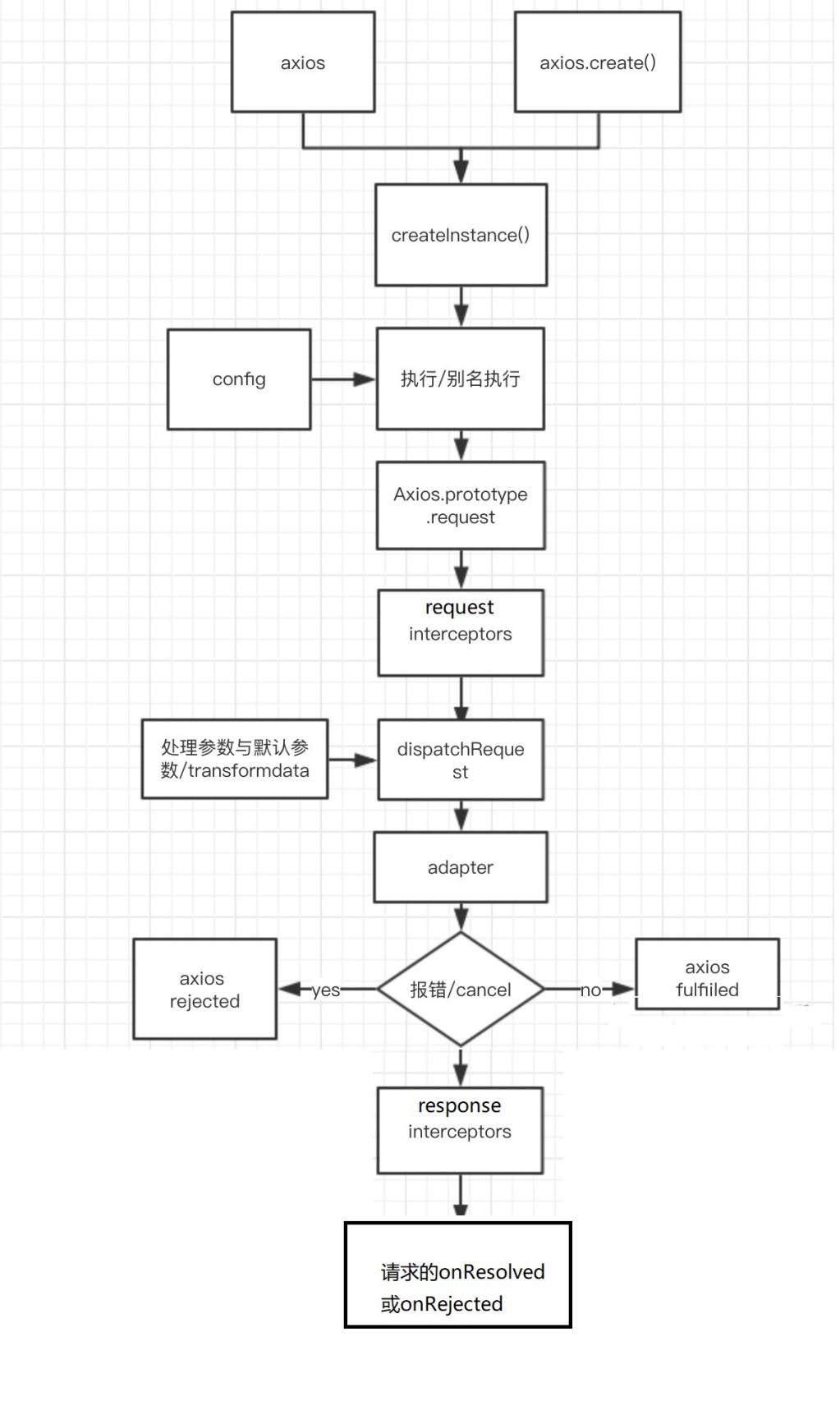
1. 请求loading
2. token处理: 通过请求头携带token数据, 对token进行校验处理
3. 异步请求成功的数据不是response, 而是response.data
4. 统一处理请求异常
5. 对请求体参数进行urlencode处理, 而不使用默认的json方式(后台接口不支持)

## axios的核心实现分析

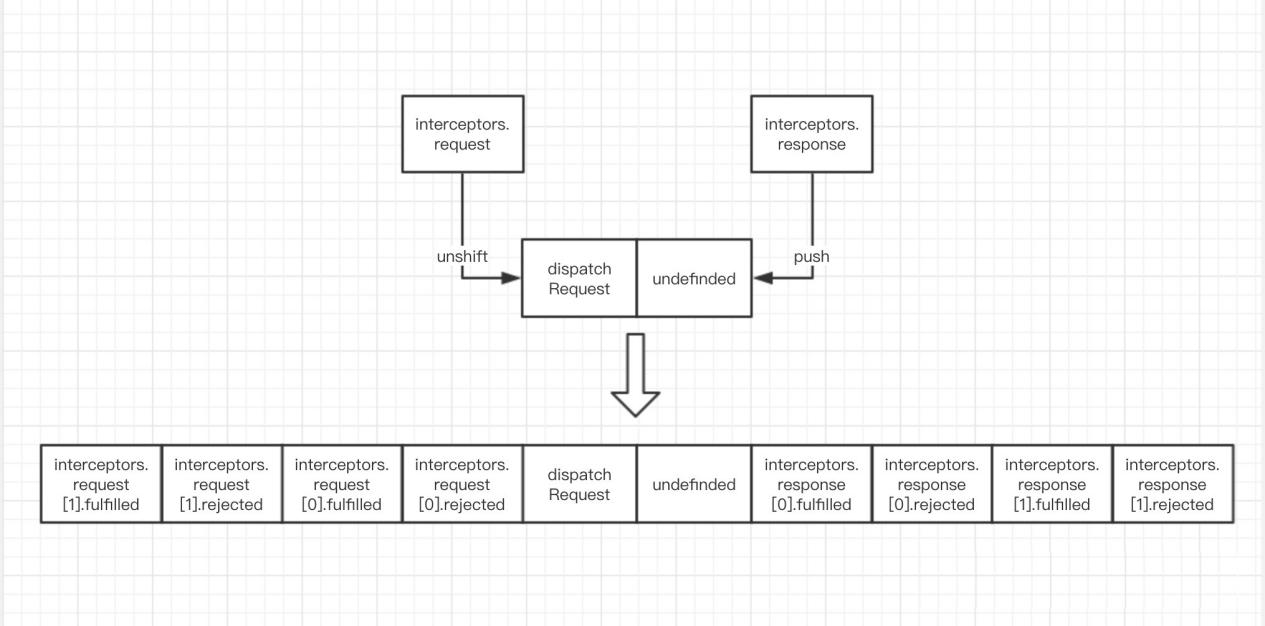
### instance与axios的区别?

1. 相同:
   1. 都是一个能发任意请求的函数: request(config)
   2. 都有发特定请求的各种方法: get()/post()/put()/delete()
   3. 都有默认配置和拦截器的属性: defaults/interceptors
2. 不同:
   1. 默认配置不一样, 且相互之间是独立的
   2. instance没有axios后面添加的一些方法: create()/CancelToken()/all()

### axios运行的整体流程?



### axios的请求/响应拦截器是什么?



1. 请求拦截器:

在真正发送请求前执行的回调函数

可以对请求进行检查或配置进行特定处理

成功的回调函数, 传递的默认是config(也必须是)

失败的回调函数, 传递的默认是error

1. 响应拦截器

在请求得到响应后执行的回调函数

可以对响应数据进行特定处理

成功的回调函数, 传递的默认是response

失败的回调函数, 传递的默认是error

### axios的请求/响应数据转换器是什么?

1. 请求转换器: 对请求头和请求体数据进行特定处理的函数

if (utils.isObject(data)) {

setContentTypeIfUnset(headers, 'application/json;charset=utf-8');

return JSON.stringify(data);

}

1. 响应转换器: 将响应体json字符串解析为js对象或数组的函数

response.data = JSON.parse(response.data)

# Webpack打包项目

## webpack的理解

1. 本质上，webpack 是一个现代 JavaScript 应用程序的静态模块打包工具
2. 当 webpack 处理应用程序时，它会在内部构建一个 依赖图(dependency graph)，此依赖图会映射项目所需的每个模块，并生成一个或多个 bundle。

## webpack的几个核心概念

1. **入口(entry)**: 打包依赖图的入口js
2. **输出(output)**: 打包生成的bundle
3. **loader**: webpack 只能理解 JS 和 JSON 文件, loader打包其它类型模块
4. **插件(plugin)**: 用来处理各种各样的任务(webpack和loader处理不了)
5. **模式(mode)**: 用于启用 webpack 内置的一系列配置
6. **浏览器兼容性(browser compatibility)**: 通过加载polyfill来兼容低版本浏览器

## webpack常用一级配置

1. entry
2. output
3. module & rules
4. plugins
5. mode
6. devServer
7. devtool
8. resolve
9. optimization

## 说说几个常见的loader

1. babel-loader
2. css-loader/style-loader/less-loader/stylus-loader/ sass-loader/postcss-loader
3. file-loader / url-loader
4. eslint-loader
5. vue-loader/vue-style-loader

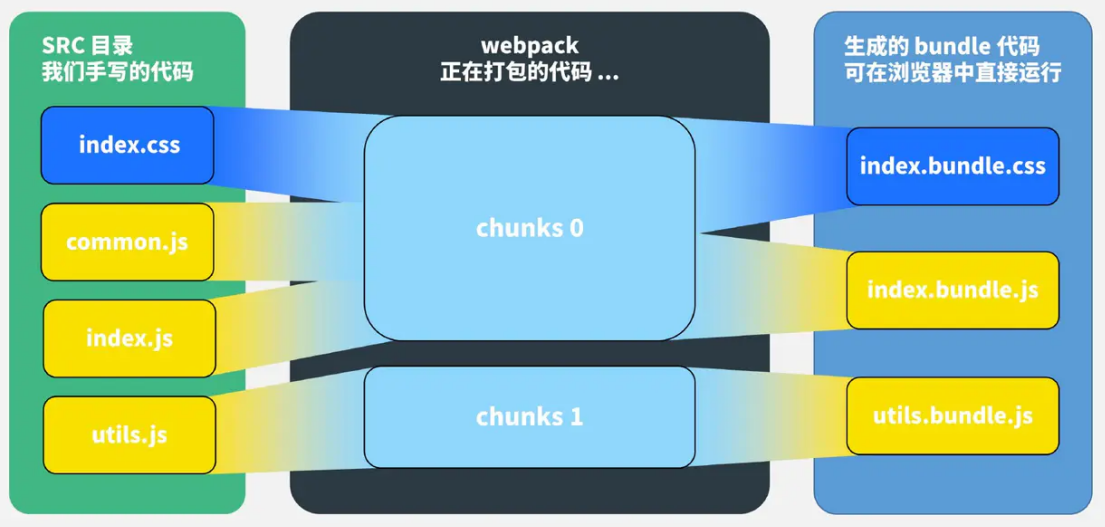
## 说说几个常见的plugin

1. html-webpack-plugin
2. copy-webpack-plugin
3. clean-webpack-plugin
4. optimize-css-assets-webpack-plugin
5. webpack-bundle-analyzer
6. webpack.HotModuleReplacementPlugin

## 区别plugin与loader

1. loader: 用于加载特定类型的资源文件, webpack本身只能打包js, 如果打包css就需要css-loader/style-loader, 如果打包图片就需要file-loader/url-loader
2. plugin: 用来扩展webpack其它方面的功能, 如页面引入打包文件需要html-webpack-plugin, 删除文件需要clean-webpack-plugin

## 区别module，chunk 和 bundle ?



1. module: 我们手写下一个一个的文件
2. webpack开始打包后, 会根据文件引用关系生成内存中的几个 chunk 文件
3. 保存到本地的是bundle文件, 一般一个chunk对应一个bundle, 但也可能一个chunk对应多个bundle(拆分样式打包)
4. 拆分js是在生成chunk前, 所以拆分出的js都是一个单独的chunk
5. 拆分css是在生成chunk后, 所以拆分出的css与js共用一个chunk

## 区别webpackPrefetch与webpackPreload

webpack4添加的魔法注释功能, 可以使用preload-webpack-plugin@next进行处理

1. webpackPreload:
   1. 同时并行请求加载, 针对当前就需要的bundle文件
   2. <link href=’’ rel=preload>
2. webpackPrefetch:
   1. 空闲时才请求加载, 针对其它路由需要的bundle文件
   2. <link href=’’ rel=prefectch>

## 区别hash、chunkhash、contenthash

1. 理解
   1. hash整体打包内容的md5值
   2. chunkhash当前chunk内容的md5值
   3. contenthash当前bundle内容的md5值
2. 使用
   1. js/css/img使用contenthash
   2. img/audio/video使用hash时本质还是用的contenthash
3. **注意**

**用hash有什么问题?**

**用chunkhash有什么问题?**

## 区别source-map 中inline与eval, cheap与module

1. Inline
   1. 生成全部的source map后整体内联到打包文件中
   2. 更新代码时完全重新生成source map
2. eval
   1. 以模块为单位生成source map内联到打包文件中
   2. 更新代码时只会重新生成相应的source map
   3. 主要是更新打包速度快于inline
3. cheap
   1. 只记录代码行号, 不记录列号
   2. 只映射自定义模块
4. module
   1. 只记录代码行号, 不记录列号
   2. 只映射第三方模块

**最佳实践:**

**开发环境------devtool: 'cheap-module-eval-source-map'**

**测试生产环境-----devtool: 'cheap-module-source-map'**

**上线生产环境-----devtool: 'none'**

## 说说webpack中babel的使用

1. 使用@babel/core和@babel/preset-env
   1. 只能编译ES6的新语法(转换为ES5相应的语法)
   2. 问题: 不能处理ES6的新API, 在相对低版本浏览器中不能运行
2. 使用@babel/polyfill
   1. 内部通过core-js提供了新API的实现
   2. 问题: 默认是打包整体包, 导致打包文件太大
3. 实现polyfill的按需要引入打包
   1. useBuiltIns: 'usage'
   2. 问题: polyfill辅助函数是在每个模块中内置定义的, 很可能重复定义了
4. 使用@babel/plugin-transform-runtime
   1. 引入单独helpers定义的辅助函数, 不再产生重复定义的多余代码
   2. 减小打包文件

## webpack如何优化打包文件

1. 目标

兼容性 / 减小打包文件/ 懒加载 / 预加载 / 首屏加载优化

1. 常用技巧
   1. 兼容低版本浏览器
   2. 拆分打包 & 压缩
   3. 异步/懒加载
   4. 预取/预加载
   5. 打包文件hash化(利用浏览缓存)
   6. Tree Shaking
   7. Scope Hoisting(作用域提升)
   8. 服务器(nginx)开启gzip
   9. 打包文件分析

## webpack如何优化打包

1. 目标

加快打包 / 提升开发调试体验

1. 常用技巧
   1. loader增加include匹配特定条件
   2. 合理配置extensions扩展名
   3. 配置resolve.alias字段, 指定常用的路径别名
   4. dll第三方模块进行预打包==> 使HardSourceWebpackPlugin更简洁更快
   5. eslint代码规范检查
   6. sourcemap 源码映射
   7. live-reload / hot-reload
   8. 加快loader处理: 多线程/多进程loader

# 手写代码(工具函数库)

<https://github.com/zxfjd3g/atguigu-utils>

## 函数相关

### 函数对象的call() / apply() / bind()

### 函数的节流(throttle)与防抖(debounce)

## 数组相关

### 数组声明式系列方法

### 数组去重(unique)

### 数组合并(concat)与切片(slice)

### 数组扁平化(flatten)

### 数组取真与分块

### 数组取差异与合并

### 删除数组中部分元素

### 得到数组的部分元素

## 对象相关

### 自定义new和instanceof工具函数

### 合并多个对象

### 对象/数组浅拷贝与深拷贝

## 字符串相关

### 字符串倒序

### 字符串是否是回文

### 截取字符串

## 事件总线

### EventBus: 包含所有功能的全局事件总线对象

### EventBus.on(eventName, listener): 绑定事件监听

### EventBus.emit(eventName, data): 分发事件

### EventBus.off(eventName): 解绑事件监听

## 消息订阅与发布

### PubSub: 包含所有功能的订阅/发布消息的管理者

### PubSub.subscribe(msg, subscriber): 订阅消息

### PubSub.publish(msg, data): 异步发布消息

### PubSub.publishSync(msg, data): 同步发布消息

### PubSub.unsubscribe(flag): 取消订阅

## Promise

### 定义整体结构

### Promise构造函数的实现

### promise.then()/catch()的实现

### Promise.resolve()/reject()的实现

### Promise.all/race()的实现

### Promise.resolveDelay()/rejectDelay()的实现

## ajax函数封装: axios

## 数据结构与排序算法

### 自定义栈: Stack

### 自定义队列: Queue

### 自定义优先级队列: PriorityQueue

### 简单排序: 冒泡 / 选择 / 插入

### 高级排序: 希尔

# 原理结构图分析

## 原型与原型链

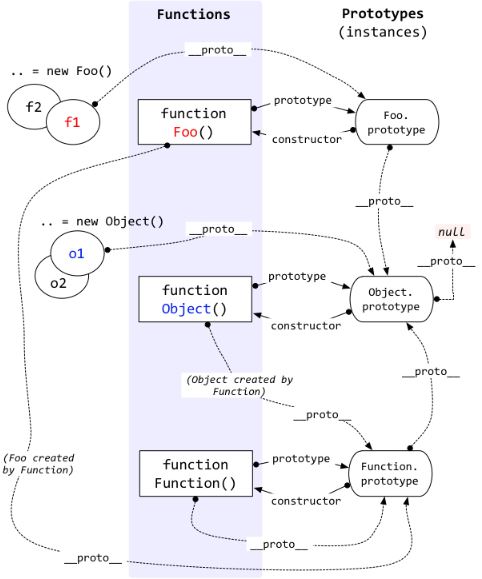
*function Foo () {}*

*const f1 = new Foo()*

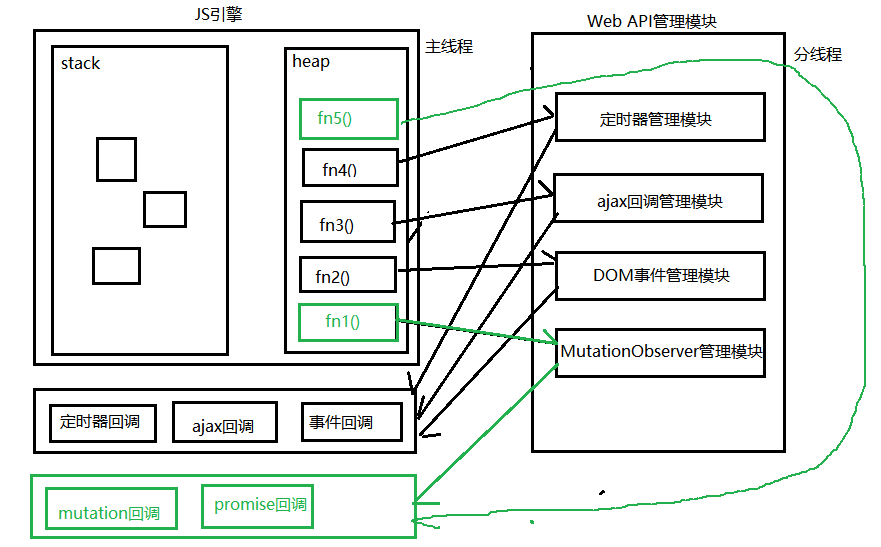
*const f2 = new Foo()*

*const o1 = new Object()*

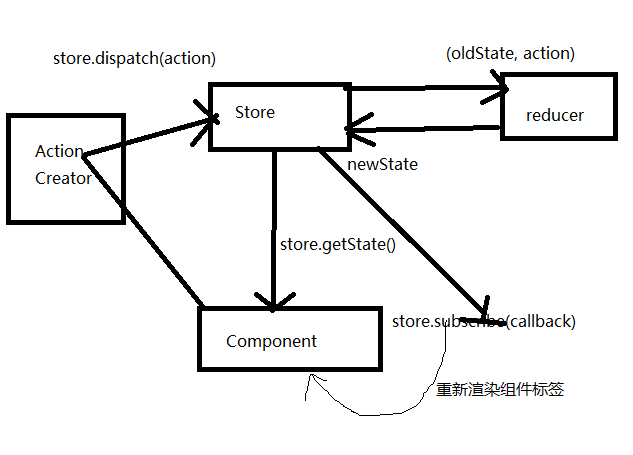
*const o2 = {}*



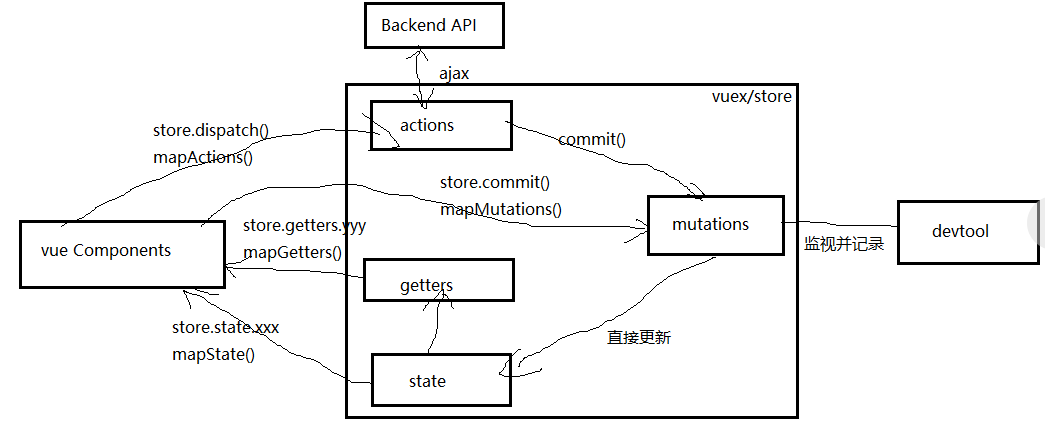
## JS事件循环(Event Loop)机制



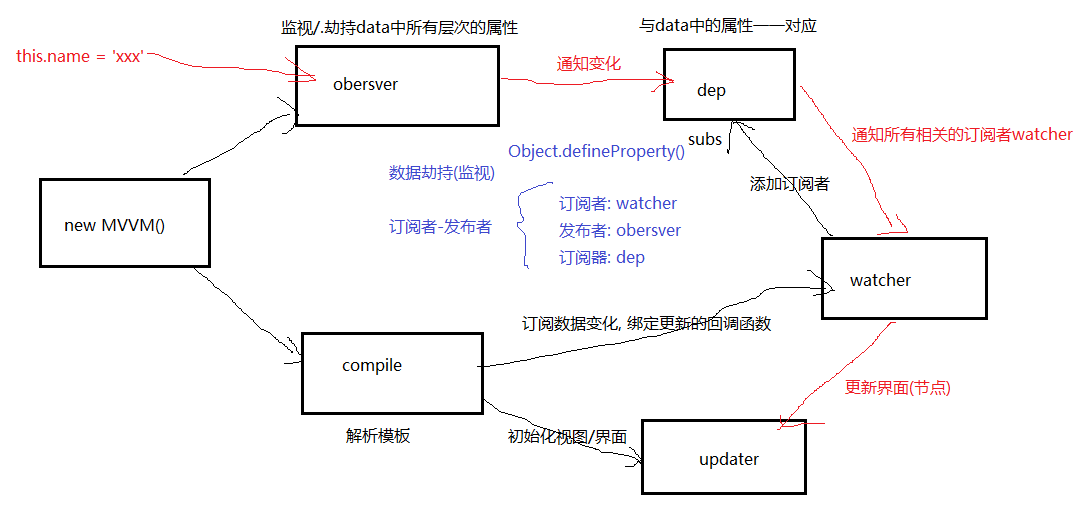
## redux结构图



## vuex结构图



## Vue的数据绑定原理图

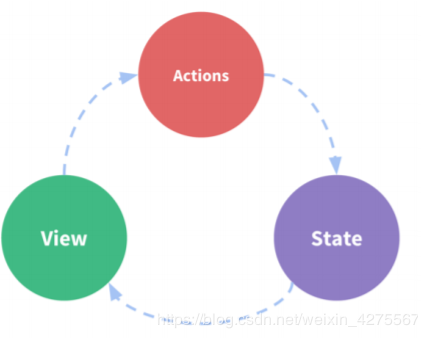


# 框架必备技能

## Vue框架

## 区别单向数据流与双向数据绑定

1. 单向数据流



用户操作界面, 调用更新状态数据的方法, 一旦状态数据变化, 自动更新相应的界面

1. 双向数据绑定

在原在的基础上, 通过v-model自动给表单项输入绑定监听, 将输入数据不通过action直接保存到state中

## 生命周期函数

1. vue的生命周期: 创建=>挂载=>更新=>销毁
2. vue的生命周期勾子:
   1. 初始化(一次): beforeCreate() => created() => beforeMount() => mounted()
   2. 更新(n次): beforeUpdate() => updated()
   3. 销毁(一次): beforeDestroy() => destroyed()
   4. 组件缓存/激活: deactivated() / activated()
   5. 捕获子组件错误: errorCaptured()
3. 一些细节

(1) beforeCreate(): 在实例初始化之后调用, data和methods都还没有初始化完成, 通过this不能访问

(2) created(): 此时data和methods都已初始化完成, 可以通过this去操作, 可以在此发ajax请求

(3) beforeMount(): 模板已经在内存中编译, 但还没有挂载到页面上, 不能通过ref找到对应的标签对象

(4) mounted(): 页面已经初始显示, 可以通过ref找到对应的标签, 也可以选择此时发ajax请求

(5) beforeUpdate(): 在数据更新之后, 界面更新前调用, 只能访问到原有的界面

(6) updated(): 在界面更新之后调用, 此时可以访问最新的界面

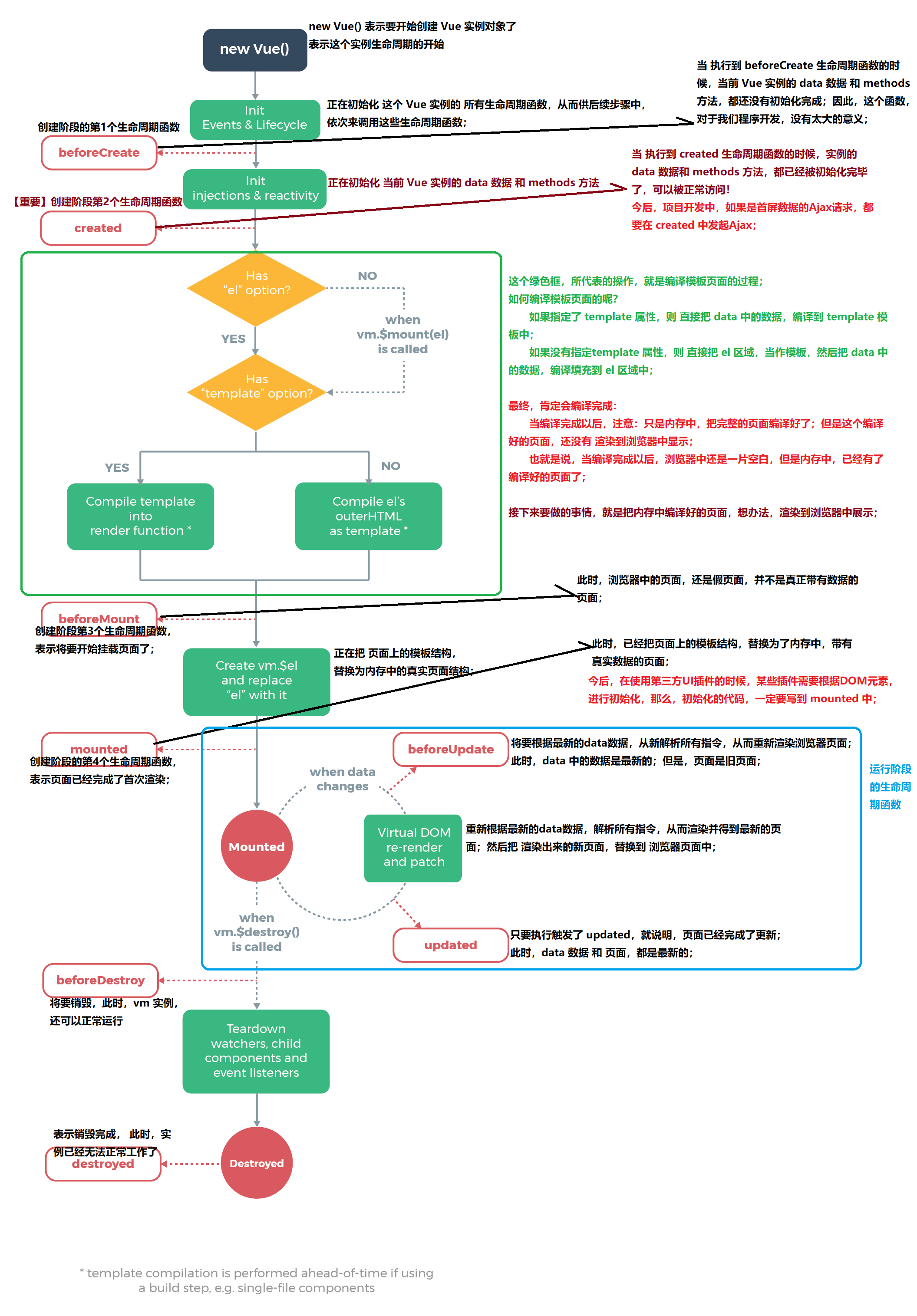
(7) beforeDestroy(): 实例销毁之前调用, 此时实例仍然可以正常工作

(8) destroyed(): Vue 实例销毁后调用, 实例已经无法正常工作了

(9) deactivated():组件失活, 但没有死亡

(10) activated(): 组件激活, 被复用

(11) errorCaptured(): 用于捕获子组件的错误,return false可以阻止错误向上冒泡(传递)



## 组件通信方式





## 路由

### 路由跳转

1. 创建路由器: new VueRouter({ })
2. 注册路由: routes: [ {path:’路由路径’, component: ‘路由组件’} ]
3. 注册路由器: new Vue({router})
4. 路由链接: <router-link :to="{name: 'home', params: {id: 1}}">home</router-link>
5. 编程式路由跳转: router.push()/replace()/back()
6. 自动跳转的路由: { path: '/', redirect: '/home' }

### 路由传参

**1). params:**

{name: ‘xxx’, path: “/person/:id”, component: Person}

router.push(‘/person/3’)

router.push({name: ‘xxx’, params: {id: 3}})

this.$route.params.id



**2). query:**

 **this.$router.push(‘/xxx?id=3’)**

小细节： 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='{path:"/hello/test2", query**: {**id**: **"222"**}**}'**>test2</**router-link**> |

**3). meta:**



**4). props:**

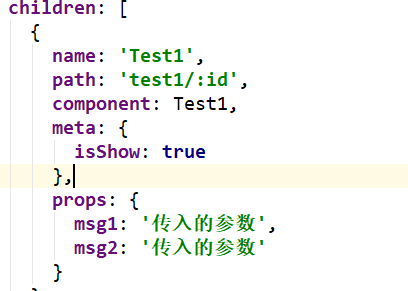
**布尔值模式: 解耦路由组件**

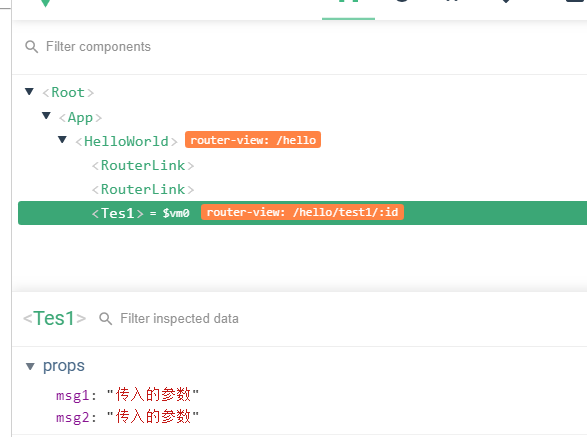
****

****

**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

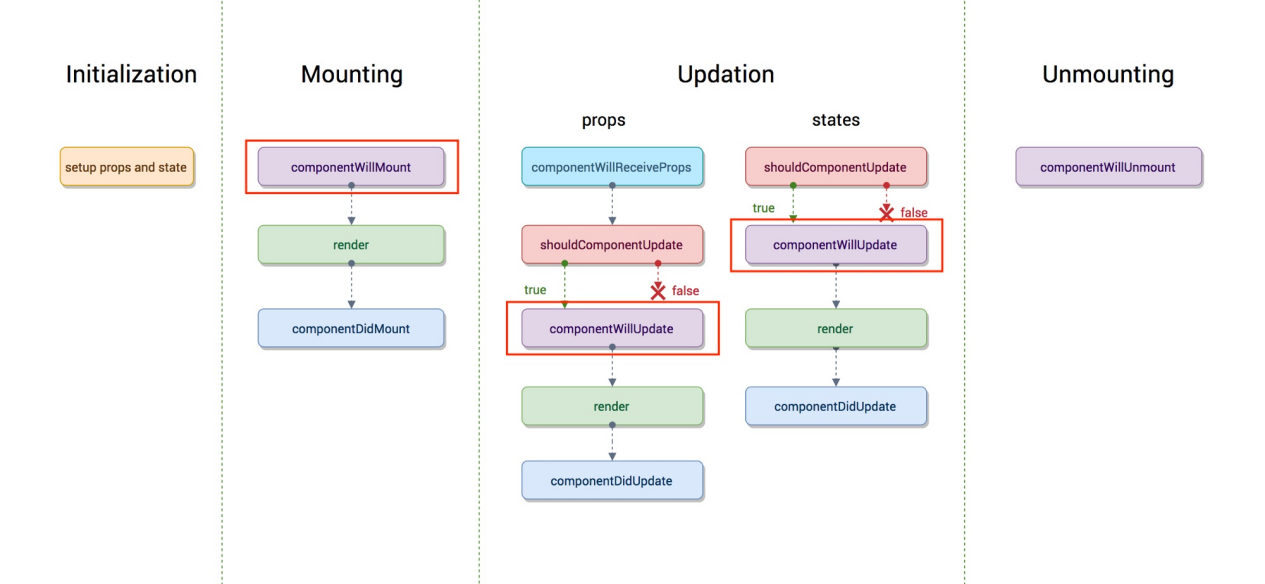
## React框架

## 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------'**);}  componentDidMount() {  ***console***.log(**'-----componentDidMount 组件已挂载完毕--------'**);  *setTimeout*(() => {  **this**.setState({  **msg**: 123  })  }, 2000) } this.setState({}) 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 组件已更新完毕--------'**); }  componentWillUnmount() {  ***console***.log(**'------componentWillUnmount 组件将要卸载-------'**); }  render () {  ***console***.log(**'render 渲染'**);  **return** (  <**div**>  App组件。。。  </**div**>  ); } |



## 组件通信

### props

1. 父传子: 非函数属性
2. 子传父: 函数属性
3. 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)

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

### Redux

1. 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
2. 何为 store
   1. store 是一个 javascript 对象，它保存了整个应用的 state。
   2. 允许通过 getState() 访问 state
   3. 通过 dispatch(action) 改变 state
   4. 通过 subscribe(listener) 注册 listeners
3. 何为 action
   1. action是一个纯 js对象，
   2. 必须有一个 type 属性表明正在执行的 action 的类型, 还有一个携带数据的属性
   3. 实质上，action 是将数据从应用程序发送到 store 的载体
4. 何为 reducer

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

array.recuce((preTotal, item) => preTotal + item , 0)

function count (state={}, action) {

retrn newState

}

## React路由

### 路由跳转

1. 路由器: HashRouter, BrowserRouter
2. 路由: <Route path='/home' component={Home}></Route>
3. 路由链接: <Link to='/home'> HOME</Link>
4. 路由重定向: <Redirect to='/home'/>
5. 编程式路由导航: this.props.history.push/replace('/home')
6. 路由精准匹配exact: 要求必须完全匹配指定的路由，默认进行的模糊匹配

### 路由传参

1. params传参 推荐使用

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

****

1. query传参 (不方便, 不建议使用)

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

****

1. props传参

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

****

## jQuery

## 整体理解

1. jQuery是一个功能强大的函数库, 封装了以下功能
   1. DOM的CRUD
   2. DOM事件处理
   3. ajax请求
   4. 各种小的工具函数
2. 核心语法
   1. jQuery核心函数
   2. jQuery核心对象
3. 特点:
   1. 链式调用
   2. 读写合一
   3. 更好的浏览器兼容性

## jQuery核心函数

1. 是什么?

就是jquery库向外暴露的$和jQuery, 一般都用$

1. 它有2种使用方式:
   1. 作为函数使用
   2. 作为对象使用
2. $作为函数使用
   1. 参数为函数: 指定回调函数在页面加载完后执行
   2. 参数是选择器字符串: 查找所有匹配的元素, 返回jQuery对象
   3. 参数是标签格式字符串: 创建DOM对象, 返回jQuery对象
   4. 参数是DOM元素: 返回包含此DOM元素的jQuery对象
3. $作为函数对象使用
   1. 发送ajax请求的各种方法: $.ajax()/get()/post()/getJSON()
   2. 一些工具方法: $.each(array)/type(data)

## jQuery核心对象

1. 2种情况产生的对象 (内部包含n个dom元素对象)
   1. 执行$函数返回的对象
   2. 调用jQuery核心对象的方法返回的对象
2. 主要方法
   1. 进行DOM进行增删改的方法
   2. 对内部的DOM元素进行进一步过滤查找的方法
   3. 进行事件处理的方法: 绑定监听/解绑监听/事件委派

## 编码测试

|  |
| --- |
| <!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="https://cdn.bootcss.com/jquery/1.11.3/jquery.js"**></**script**>  <**script**>  */\*  $作为函数使用  a. 参数为函数: 指定回调函数在页面加载完后执行  b. 参数是选择器字符串: 查找所有匹配的元素, 返回jQuery对象  c. 参数是标签格式字符串: 创建DOM对象, 返回jQuery对象  d. 参数是DOM元素: 返回包含此DOM元素的jQuery对象  \*/* $(() => {  *// 1. 给test1添加点击监听, 提示按钮的文本内容* $(**'#btn1'**).click(**function** () {  alert($(**this**).html())  $(**this**).html(**'test1111'**)  })   *// 2. 在页面的最后添加一个新的按钮* $(**'<button>test</button>'**)  .click(**function** (event) {  alert(event.**target**.**innerHTML**)  })  .appendTo(**'body'**)    */\*   $作为函数对象使用  a. 发送ajax请求的各种方法: $.ajax()/get()/post()/jsonp()  b. 一些工具方法: $.each(array)/type(data)  \*/* $.ajax({  **url**: **'https://api.github.com/search/repositories'**,  **type**: **'GET'**,  **data**: {  **q**: **'v'**,  **sort**: **'stars'** },  success: (data) => {  **console**.log(data)  },  error: () => {  alert(**'请求错误'**)  }  })   **const** arr = [1, 3, 5]  $.each(arr, (index, value) => **console**.log(index, value))  **console**.log($.type(arr), $.type(arr[0]))    */\*   2种情况产生的jQuery核心对象 (内部包含n个dom元素对象)  a. 执行$函数返回的对象  b. 调用jQuery核心对象的方法返回的对象  \*/* **const** $lis = $(**'li'**)  **const** $firstLi = $lis.first()  **console**.log($firstLi.html())   */\*   jQuery对象的主要方法  a. 进行DOM进行增删改的方法  b. 对内部的DOM元素进行进一步过滤查找的方法  c. 进行事件处理的方法: 绑定监听/解绑监听/事件委派  \*/* $(**'#btn2'**).click(**function** () {  $(**'ul'**).find(**'li>button'**).not(**':first'**).click(**function** () {  $(**this**).parent().remove()  })  })   $(**'#btn3'**).click(**function** () {  $(**'<li><span>X</span><button>testX</button></li>'**).appendTo(**'ul'**)  })   $(**'#btn4'**).click(**function** () {  $(**'ul'**).delegate(**'li:not(:first)>button'**, **'click'**, **function** () {  $(**this**).parent().remove()  })  })   $(**'#btn5'**).click(**function** () {  $(**'#btn1'**).unbind(**'click'**)  $(**'ul'**).undelegate(**'click'**)  })  })  </**script**> </**head**> <**body**>  *<!--   需求1: 点击test1按钮, 提示按钮的文本内容  需求2: 在页面的最后添加一个新的按钮  需求3: 发送ajax请求, 提示请求结果  需求4: 遍历数组, 得到不同数据的类型  -->* <**button id="btn1"**>test1</**button**>  <**ul**>  <**li**>  <**span**>A</**span**>  <**button**>testA</**button**>  </**li**>  <**li**>  <**span**>B</**span**>  <**button**>testB</**button**>  </**li**>  <**li**>  <**span**>C</**span**>  <**button**>testC</**button**>  </**li**>  </**ul**>   *<!--   需求5: 给列表中所有按钮添加点击监听, 点击删除当前行  需求6: 在最后添加一行  需求7: 与需求5功能一样, 但使用事件委托  需求8: 解除btn1上的点击监听和列表的委托点击监听  -->* <**button id="btn2"**>给列表中所有button添加点击监听</**button**><**br**>  <**button id="btn3"**>向列表中添加一个列</**button**><**br**>  <**button id="btn4"**>给列表中所有button添加点击监听(使用事件委托)</**button**><**br**>  <**button id="btn5"**>解绑事件监听</**button**><**br**> </**body**> </**html**> |

## 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:
   1. 发送的都是https请求
   2. 一个域名下最多配置20个域名
   3. 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)



# 项目开发遇到问题及解决方案

## fastClick的300ms延迟解决方案

### 问题剖析:

1. 移动端浏览器在派发点击事件的时候，通常会出现300ms左右的延迟
2. 原因: 移动端的双击会缩放导致click判断延迟

### 解决方案: fastclick库



实现原理: 在检测到touchend事件的时候，会通过DOM自定义事件立即出发模拟一个click事件，并把浏览器在300ms之后真正的click事件阻止掉

## 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，原因是加载路径错误

### 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的属性字段（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**> |

## 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. vue-cli2: 设置修改config/index ==> productionSourceMap: false
3. vue-cli3设置修改: vue.config.js ==> 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
3. 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'**: **''** }  } }, |

1. vue-cli3配置: vue.config.js文件中配置

|  |
| --- |
| **devServer: {**  **open: true,**  **proxy: {**  **// 处理以/api开头路径的请求**  **'/api': {**  **target: 'http://localhost:4000', // 转发的目标地址**  **pathRewrite: {**  **'^/api' : '' // 转发请求时去除路径前面的/api**  **},**  **changeOrigin: true, // 如果协议/主机也不相同, 必须加上**  **},**  **}**  **},** |

### 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

|  |
| --- |
| # 指向前台应用打包后文件位置  location / {  root D:\work\190418\video\day07\online\admin\_browser; # 前台应用的根路径  index index.html; # 默认index页面  # try\_files $uri $uri/ /index.html; # 总是返回首页  }  # 配置react项目的后台接口  location /api/ {  proxy\_pass http://127.0.0.1:5000/;  } |

## 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

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. 目标: 想办法在列表数据更新显示之后才去创建swiper
2. 方式一: 使用watch+$nextTick()
3. 方式二: 使用回调+$nextTick()

## Vue实例的响应式属性 VS 非响应式属性

### 问题剖析

1. 响应式式属性
   1. Vue实例初始化之前定义在data中的属性
   2. 修改响应式属性之后，会触发视图更新
2. 非响应式属性
   1. 直接通过this.xxx = value添加的属性是非响应式属性

|  |
| --- |
| data(){  this.age = 3 // age不是响应式的  **return** {  obj: { // obj是响应式属性  name: 'kobe' // name也是响应式属性}  } }, |

### 解决方案

1. 使用Vue.set() 或this.$set(target, propertyName, value)
2. 示例: this.obj.age = 30 --🡪 this.$set(this.obj, 'age', 30)

## 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

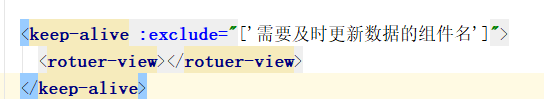
想要保留路由组件之前页面滚动的位置就要保留之前的状态

### 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include或者exclude属性，将需要及时刷新的路由组件排除



# 项目性能优化

## UI库按需加载(打包)问题

### 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

### 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

### 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

### 解决方案

|  |
| --- |
| 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'); |

## React性能优化值 shouldComponentUpdate

### 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, 而默认值是true

### 解决方案

1. 方式一: 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;
2. 方式二: 不实现Component, 而去实现PureComponent

## 图片懒加载

### 问题剖析

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

### 解决方案

1. npm install vue-lazyload
2. 声明使用: Vue.use(VueLazyLoad, {loading: loading图片})
3. 组件使用: <img v-lazy="item.primaryPicUrl" alt="新品">

## webpack打包优化

### 优化打包文件

1. **目标:**   
   兼容性 / 减小打包文件/ 懒加载 / 预加载 / 首屏加载优化
2. **技巧:**



### 优化打包

1. **目标:**   
   加快打包 / 提升开发调试体验
2. **技巧:**



## 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

* 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
* 尽量减少重绘重排的次数
* 选择器命名规范，通过id寻找更快
* 动画区域开启独立的图层
  + 定位： position: absolute/relative
  + Will-change: xxx;
* 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。
* 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

## 封装组件

### 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

### 解决方案

1. 抽取封装组件
2. 封装组件思想:
   1. 组件内部保留相同的内容
   2. 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
   3. 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
   4. 通过标签属性的形式传递，在组件内部通过props接收

# 经典问题解析

## MVC和MVVM的区别

### MVC



**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。

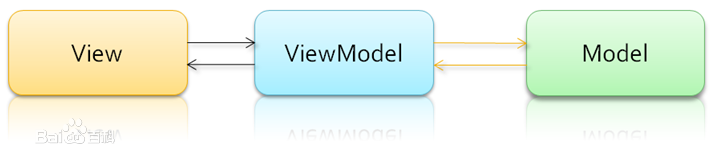
**View（视图）**是应用程序中处理数据显示的部分。  
　　通常视图是依据模型数据创建的。

**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



1. 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
2. 跨域资源嵌入： 默认允许, 比如: <script> / <link> / <img> / <video> / <audio> / <iframe>
3. 跨域读: 默认不允许, 比如ajax请求, 读取跨域的Cookie, localStorage, sessionStorage

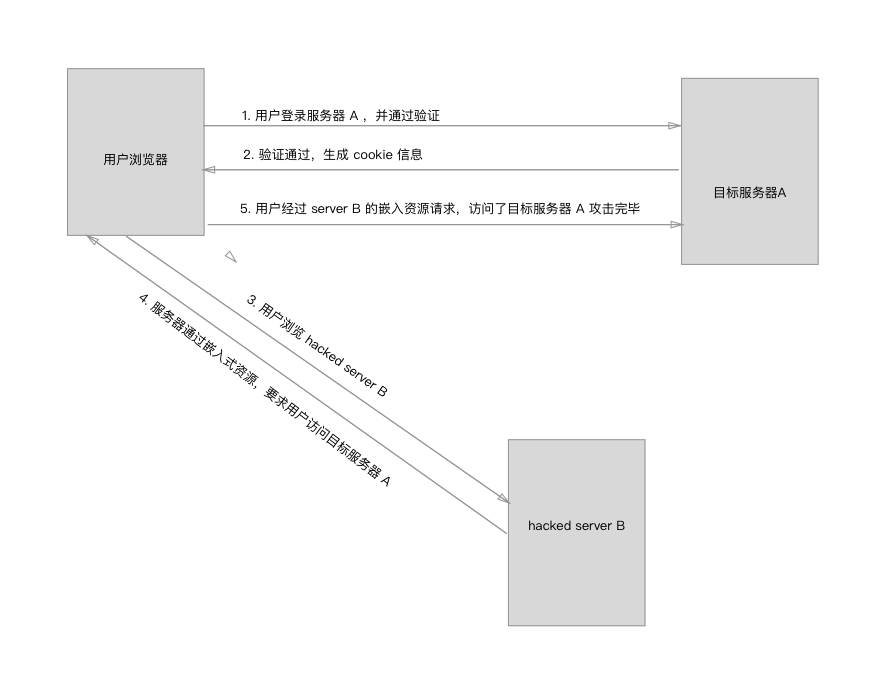
## XSRF/CSRF攻击

### [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

### 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

## 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

1. 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

2. 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

3. 验证请求的referer信息

4. 进行token验证

## 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: 给window绑定beforeupload事件监听

