张东

18301092802

准备: 微信小程序: WEB问题速查 小恐龙图标

1. 历届师兄师姐在今天课程中曾经问过的高频的经典的问题清单和解答: 小程序->首页->JSCORE->day01

自己可能没有发现问题，但是想知道别的同学都问过哪些问题

2. 今日课程对应的重点案例的视频以及笔试面试所需的扩展知识视频：小程序->在线->JSCORE->day01

如果因为某些原因错过直播讲解，不用着急！直接看小程序中视频的回看即可。

如果tmooc不能用了，立刻看小程序视频，小程序视频比tmooc要全，甚至有扩展的新知识。

如果学有余力，将来有能力拿更高薪资，小程序中所有视频必须都要看。但是不必现在都看完。只要在笔试面试前都看完即可！

问题: 个人版小程序有功能限制：禁止在线播放视频

解决: 长按视频连接地址，完整复制视频连接地址和验证码到浏览器打开。通过百度网盘下载后观看。

3. 今日课程中包含的英语单词的翻译：小程序->单词->JSCORE->day01

英文不太好的同学，可以提前看一下第二天需要用到哪些英文单词。

第三阶段要讲哪些东西:

1. JS高级: JS语言最核心的原理、ES5和ES6新的用法——8天

问题: 无法操作网页内容，无法实现各种页面交互效果。

2. DOM: 专门操作网页内容，实现各种页面交互效果的一套对象和函数——4天

问题: 对象名、函数名、步骤太长！太繁琐！

3. jQuery: 对DOM的简化——4天（了解）

问题: 只简化了对象名和函数名，没有简化步骤！

4. vue: 前端大项目开发的终极简化！——5天

第三阶段和前两个阶段的关系:

1. 第一阶段: 只需要js基础9天内容即可。~~nodejs和sql~~没学好一点关系都没有。

2. 第二阶段: 只需要css基础(选择器)和transition。~~boot和ajax~~没学好一点关系没有。

目录

[Jscore 27](#_Toc54854675)

[**Day01正课**: 27](#_Toc54854676)

[1. 正则表达式 27](#_Toc54854677)

[2. String正则相关的函数 27](#_Toc54854678)

[一. 正则表达式: PPT —— 下载: ftp 27](#_Toc54854679)

[二. String正则相关的函数: 3件事儿: (高频面试题) 27](#_Toc54854680)

[1. 查找敏感词: 4种情况: 27](#_Toc54854681)

[2. 替换敏感词: 29](#_Toc54854682)

[3. 切割字符串: 29](#_Toc54854683)

[**Day02**正课: 30](#_Toc54854684)

[一. String提供的正则相关的函数: 3件事: 30](#_Toc54854685)

[1. 查找关键词: 4种情况 30](#_Toc54854686)

[(1). 查找一个固定的关键词出现的位置: 30](#_Toc54854687)

[(2). 用正则模糊查找多种关键词的位置: 30](#_Toc54854688)

[(3). 用正则模糊查找多种关键词的内容: 30](#_Toc54854689)

[(4). 既查找每个敏感词的内容，又查找每个敏感词的位置: 32](#_Toc54854690)

[2. 替换关键词: 2种: 32](#_Toc54854691)

[(1). 简单替换: 将所有找到的敏感词都替换为统一的新值 32](#_Toc54854692)

[(2). 高级替换: 根据每次找到的敏感词不同，动态替换成不同的新值 33](#_Toc54854693)

[(3). 衍生操作：删除关键词，将关键词替换为""，就算删除了 34](#_Toc54854694)

[3. 切割字符串: 35](#_Toc54854695)

[(1). 简单切割: 切割符是固定不变的 35](#_Toc54854696)

[(2). 复杂切割: 切割符不是固定不变的，但是可以找到规律 35](#_Toc54854697)

[(3). 固定套路: 打散一个字符串为字符数组: 35](#_Toc54854698)

[(4). 示例: 按规定切割字符串 35](#_Toc54854699)

[二. RegExp对象: 36](#_Toc54854700)

[4. 如何创建正则表达式对象: 2种方式: 37](#_Toc54854701)

[(1). 标准写法: var reg=new RegExp("正则表达式", "ig") 37](#_Toc54854702)

[(2). 简写: var reg=/正则/ig 38](#_Toc54854703)

[5. RegExp提供的函数: 2个 38](#_Toc54854704)

[**Day03**正课: Function 42](#_Toc54854705)

[一. 回顾函数： 42](#_Toc54854706)

[(1). 声明方式: 43](#_Toc54854707)

[(2). 赋值方式: 43](#_Toc54854708)

[a. var函数名=function (形参变量列表){ 43](#_Toc54854709)

[b. 优点: 不会被声明提前，保持了程序正常的执行顺序，不会产生歧义 43](#_Toc54854710)

[c. 揭示了js中函数的本质: 43](#_Toc54854711)

[d. 示例: 声明提前笔试题 43](#_Toc54854712)

[(3). 用new创建: (因为格式极其晦涩，极少使用，了解即可) 44](#_Toc54854713)

[二. 重载(overload): 44](#_Toc54854714)

[a. 什么是arguments对象: 45](#_Toc54854715)

[b. 如何实现重载: 45](#_Toc54854716)

[三. 匿名函数: 46](#_Toc54854717)

[(1). 回调函数: 46](#_Toc54854718)

[(2). 匿名函数自调: 46](#_Toc54854719)

[四. 作用域(scope)和作用域链: 47](#_Toc54854720)

[1. 什么是作用域: 2种解释: 47](#_Toc54854721)

[2. 为什么: 避免不同范围的变量之间互相干扰 47](#_Toc54854722)

[3. js中包括2种作用域: 47](#_Toc54854723)

[(1). 全局作用域： 47](#_Toc54854724)

[(2). 函数作用域： 48](#_Toc54854725)

[4. 作用域链: 48](#_Toc54854726)

[(1). 什么是作用域链: 48](#_Toc54854727)

[(2). 函数从定义到调用，再到调用完成所经历的过程: 48](#_Toc54854728)

[(3). 以上过程中函数对象的"好友列表"，学名就叫作用域链: 50](#_Toc54854729)

[五. 闭包(closure): 50](#_Toc54854730)

[4. 如何: 3步 —— 判断闭包的三个特点 51](#_Toc54854731)

[**Day04**正课: OOP 53](#_Toc54854732)

[一. 什么是面向对象: 53](#_Toc54854733)

[二. 封装: 53](#_Toc54854734)

[6. this: 54](#_Toc54854735)

[7. 示例: 定义对象描述一个学生lilei的属性和会做的事儿: 54](#_Toc54854736)

[8. 创建对象可以有: 3种方式: 55](#_Toc54854737)

[(1). 用{}创建一个对象: 55](#_Toc54854738)

[(2). 用new Object()创建一个对象: 55](#_Toc54854739)

[(3). 使用构造函数反复创建多个相同结构的对象: 59](#_Toc54854740)

[三. 继承: 61](#_Toc54854741)

[5. 原型对象: 61](#_Toc54854742)

[**Day05**正课: 64](#_Toc54854743)

[一. 继承 64](#_Toc54854744)

[1. 自有属性和共有属性 64](#_Toc54854745)

[(1). 什么是: 64](#_Toc54854746)

[(2). 何时： 64](#_Toc54854747)

[(3). 如何: 64](#_Toc54854748)

[(4). 坑: 64](#_Toc54854749)

[(5). 示例: 测试获取和修改自有属性和共有属性 65](#_Toc54854750)

[2. 内置类型的原型对象 66](#_Toc54854751)

[a. 构造函数: 67](#_Toc54854752)

[b. 原型对象: 替该类型下所有子对象保存共有的方法 67](#_Toc54854753)

[3. 原型链: 68](#_Toc54854754)

[二. 多态： 68](#_Toc54854755)

[1. 什么是: 一个函数在不同的情况下表现出不同的状态 68](#_Toc54854756)

[2. 包括: 2种: 69](#_Toc54854757)

[3. 重写: 69](#_Toc54854758)

[三. 自定义继承: 70](#_Toc54854759)

[2. 如何: 2种: 70](#_Toc54854760)

[一. 严格模式: 73](#_Toc54854761)

[i. 总结: 不要走极端！ 76](#_Toc54854762)

[**Day06**正课: \*\*\*\*ES5\*\*\*\* 77](#_Toc54854763)

[一. 保护对象: 77](#_Toc54854764)

[4. 保护属性: 77](#_Toc54854765)

[i. 想获取访问器属性值时: 84](#_Toc54854766)

[ii. 想修改访问器属性值时: 84](#_Toc54854767)

[5. 保护对象的结构: 3级别 86](#_Toc54854768)

[(1). 防扩展: 禁止给对象随意添加新属性 86](#_Toc54854769)

[(2). 密封: 既禁止添加新属性，又禁止删除现有属性 86](#_Toc54854770)

[(3). 冻结: 既禁止添加新属性，又禁止删除现有属性，甚至连属性值都不能修改了 88](#_Toc54854771)

[a. 如何: Object.freeze(对象) 88](#_Toc54854772)

[b. 原理: 悄悄做了三件事: 88](#_Toc54854773)

[c. 何时: 88](#_Toc54854774)

[d. 示例: 冻结一个对象，禁止修改其属性值 88](#_Toc54854775)

[二.Object.create() 88](#_Toc54854776)

[三. 替换this: 90](#_Toc54854777)

[(1). 在一次调用函数时，临时替换一次this为指定的对象 90](#_Toc54854778)

[(2). 基于原函数创建一个新函数副本，并永久替换其中的this 92](#_Toc54854779)

[a. 问题: 如果一个函数每次调用时都要用call()替换this——麻烦 92](#_Toc54854780)

[b. 解决: 创建一个新函数副本，并永久绑定新函数中的this为指定对象 92](#_Toc54854781)

[c. 如何: var 新函数=原函数.bind(替换this的对象, 要固定的实参值) 92](#_Toc54854782)

[d. 原理: bind共做了3件事: 92](#_Toc54854783)

[e. 示例: 基于原函数创建一个新函数副本，并永久绑定this 92](#_Toc54854784)

[四. 数组新增函数: 93](#_Toc54854785)

[i. 将当前正在遍历的一个元素值交给第一个形参elem 93](#_Toc54854786)

[ii. 将当前正在遍历到的下标位置交给第二个形参i 93](#_Toc54854787)

[iii. 将当前every前的.前的数组对象交给第三个形参arr 93](#_Toc54854788)

[i. 将当前正在遍历的一个元素值交给第一个形参elem 94](#_Toc54854789)

[ii. 将当前正在遍历到的下标位置交给第二个形参i 95](#_Toc54854790)

[iii. 将当前every前的.前的数组对象交给第三个形参arr 95](#_Toc54854791)

[**Day07**正课: 97](#_Toc54854792)

[一. 数组新增函数: 97](#_Toc54854793)

[1. 遍历: 97](#_Toc54854794)

[i. 当前元素值给elem 97](#_Toc54854795)

[ii. 当前元素下标位置给i 97](#_Toc54854796)

[iii. 当前数组对象给arr 97](#_Toc54854797)

[i. 当前元素值给elem 98](#_Toc54854798)

[ii. 当前元素下标位置给i 98](#_Toc54854799)

[iii. 当前数组对象给arr 98](#_Toc54854800)

[2. 过滤: 99](#_Toc54854801)

[i. 当前元素值给elem 99](#_Toc54854802)

[ii. 当前元素下标位置给i 99](#_Toc54854803)

[iii. 当前数组对象给arr 99](#_Toc54854804)

[3. 汇总: 100](#_Toc54854805)

[i. 捐款箱: 当前reduce函数内保存的捐款箱变量中的临时汇总值 100](#_Toc54854806)

[ii. elem: 当前新元素值 101](#_Toc54854807)

[iii. i: 当前遍历到的位置 101](#_Toc54854808)

[iv. arr: 当前数组对象 101](#_Toc54854809)

[一. 模板字符串 101](#_Toc54854810)

[二. let 103](#_Toc54854811)

[1. 什么是let: ES6中专门代替var，用来声明变量新关键字 103](#_Toc54854812)

[2. 为什么: var的缺点: 103](#_Toc54854813)

[3. 示例: var的缺点: 103](#_Toc54854814)

[4. 何时: 今后新的框架程序中都是用let代替var的 104](#_Toc54854815)

[5. 优点: 2个: 104](#_Toc54854816)

[6. 原理: let做了两件事: 104](#_Toc54854817)

[7. 示例: 使用let代替var 104](#_Toc54854818)

[8. let的三个小脾气: 105](#_Toc54854819)

[9. 示例: let的小脾气 105](#_Toc54854820)

[三. 箭头函数 106](#_Toc54854821)

[1. 什么是箭头函数: 对旧js中function的简写 106](#_Toc54854822)

[2. 为什么: 简单！ 106](#_Toc54854823)

[3. 何时: 今后几乎所有的function都可用箭头函数简化——在将来框架开发中几乎看不到任何function了！ 106](#_Toc54854824)

[4. 如何: 3句话口诀 106](#_Toc54854825)

[5. 示例: 使用箭头函数简写各种function 106](#_Toc54854826)

[6. 箭头函数的特征: 箭头函数内的this与箭头函数外的this始终保持一致. 108](#_Toc54854827)

[7. 总结: 何时可以使用箭头函数，何时不能用箭头函数: 108](#_Toc54854828)

[8. 示例: 使用箭头函数简写回调函数和对象中的方法 108](#_Toc54854829)

[四. for of 111](#_Toc54854830)

[一. 参数增强: 114](#_Toc54854831)

[1. 参数默认值 114](#_Toc54854832)

[2. 剩余参数： 115](#_Toc54854833)

[3. 打散数组和对象 116](#_Toc54854834)

[二. 解构: 119](#_Toc54854835)

[4. 如何: 3种情况: 119](#_Toc54854836)

[(1). 数组解构: 119](#_Toc54854837)

[(2). 对象解构 120](#_Toc54854838)

[(3). 参数解构 121](#_Toc54854839)

[1). 定义函数时，先将形参列表装扮为一个对象的样子 122](#_Toc54854840)

[2). 调用函数时，所有实参值也必须放入一个相同的对象结构中传入 122](#_Toc54854841)

[三. class 125](#_Toc54854842)

[7. 如何设置共有属性值: 127](#_Toc54854843)

[8. 两种类型间的继承: extends 127](#_Toc54854844)

[a. 先再多定义一个父级class，替两个子级class集中保存相同的属性和方法 127](#_Toc54854845)

[b. 子级class中不需要包含重复的属性和方法 128](#_Toc54854846)

[c. 子级class继承父级class: 128](#_Toc54854847)

[**Day08** 130](#_Toc54854848)

[一. Promise: 130](#_Toc54854849)

[1. 什么是: 专门实现多个异步任务顺序执行的技术，专门用来避免回调地狱问题 130](#_Toc54854850)

[2. 问题: 旧js中如何使用回调函数保证多个异步任务顺序执行 130](#_Toc54854851)

[a. 定义前一项任务时: 130](#_Toc54854852)

[b. 调用前一项任务时: 130](#_Toc54854853)

[c. 结果: 130](#_Toc54854854)

[d. 示例: 使用回调函数方式，保证多个异步函数必须顺序执行 131](#_Toc54854855)

[总结: 135](#_Toc54854856)

[1. 正则表达式: 135](#_Toc54854857)

[(1). 最简单的规则就是要找的关键词正文 135](#_Toc54854858)

[(2). 如果规则中一位字符上有多种备选字时: [备选字] 135](#_Toc54854859)

[(3). 如果[]中部分备选字符是连续的: [x-x] 135](#_Toc54854860)

[(4). 预定义字符集: 针对4种最常用[]的情况提供的更简化写法 135](#_Toc54854861)

[(5). 如果规定一个字符集出现的次数，可以用量词。 136](#_Toc54854862)

[(6). 如果两个规则，二选一匹配即可: 规则1 | 规则2 136](#_Toc54854863)

[(7). 如果希望多个子规则作为一个整体先匹配，再和其它规则联合匹配: (子规则) 136](#_Toc54854864)

[(8). 如果想匹配特殊位置的关键词: 136](#_Toc54854865)

[2. String正则相关的函数: 3件事 136](#_Toc54854866)

[(1). 查找关键词: 4种情况 136](#_Toc54854867)

[(2). 替换关键词: 2种情况: 137](#_Toc54854868)

[(3). 切割字符串: 2种情况: 137](#_Toc54854869)

[3. RegExp对象: 137](#_Toc54854870)

[(1). 创建: 137](#_Toc54854871)

[(2). 验证格式: 137](#_Toc54854872)

[(3). 既查找每个敏感词内容，又查找每个敏感词位置: 137](#_Toc54854873)

[**Day04总结**: 137](#_Toc54854874)

[1. Function 137](#_Toc54854875)

[(1). 创建函数: 3种: 137](#_Toc54854876)

[(2). 重载: 138](#_Toc54854877)

[(3). 匿名函数: 138](#_Toc54854878)

[(4). 作用域和作用域链: 139](#_Toc54854879)

[(5). 闭包: 139](#_Toc54854880)

[4. 面向对象: 140](#_Toc54854881)

[(1). 封装: 3种: 140](#_Toc54854882)

[(2). 继承: 141](#_Toc54854883)

[(3). 多态: （重写） 141](#_Toc54854884)

[(4). 自定义继承: 2种: 141](#_Toc54854885)

[总结: this 4种 141](#_Toc54854886)

[new做了4件事: 142](#_Toc54854887)

[ES5: 142](#_Toc54854888)

[1. 严格模式: "use strict" 142](#_Toc54854889)

[2. 保护对象: 142](#_Toc54854890)

[(1). 保护对象属性: 2种: 142](#_Toc54854891)

[(2). 保护对象的结构: 3个级别: 144](#_Toc54854892)

[4. 替换this: 3种情况: 144](#_Toc54854893)

[(1). 只在一次调用函数时，临时替换一次函数中的this: call 或 apply 144](#_Toc54854894)

[(2). 创建一个新函数并永久绑定this为一个指定的对象: bind() 144](#_Toc54854895)

[5. 数组新增函数: 145](#_Toc54854896)

[(1). 判断: 145](#_Toc54854897)

[(2). 遍历: 2个: 146](#_Toc54854898)

[(3). 过滤: 146](#_Toc54854899)

[(4). 汇总: 146](#_Toc54854900)

[\*\*\*\*ES6\*\*\*\* 146](#_Toc54854901)

[Dom 149](#_Toc54854902)

[一. 什么是DOM: Document Object Model 149](#_Toc54854903)

[**1.** **什么是:** 149](#_Toc54854904)

[**2.** **为什么:** 149](#_Toc54854905)

[**3.** **何时:** 149](#_Toc54854906)

[4. DOM标准: 149](#_Toc54854907)

[5. DOM都能做哪些事儿: 5件事: 149](#_Toc54854908)

[二. DOM树: 149](#_Toc54854909)

[三. 查找元素: 今后，只要想修改元素，都要先找到元素。4大类查找方式: 150](#_Toc54854910)

[1. 不需要查找就可直接获得的元素对象: 4种: 150](#_Toc54854911)

[2. 根据节点间关系查找: 150](#_Toc54854912)

[a. 父子关系: 4种: 150](#_Toc54854913)

[b. 兄弟关系: 2个 151](#_Toc54854914)

[3. 按HTML特征查找: 4种: 153](#_Toc54854915)

[(1). 按元素的id名查找一个元素: 153](#_Toc54854916)

[(2). 按元素的标签名查找多个元素: 153](#_Toc54854917)

[(3). 按元素的class名查找多个元素: 154](#_Toc54854918)

[(4). 按表单元素的name名查找多个表单元素: 154](#_Toc54854919)

[(5). 示例: 按不同HTML特征查找指定的元素 154](#_Toc54854920)

[四. 事件: 156](#_Toc54854921)

[五. 综合示例: 纯客户端购物车（未完，明日待续...） 157](#_Toc54854922)

[Day02 161](#_Toc54854923)

[一. 查找元素: 4种: 161](#_Toc54854924)

[1. 不需要查找就可直接获得的元素: 4个 161](#_Toc54854925)

[2. 按节点间关系查找: 2大类关系，6个属性 161](#_Toc54854926)

[3. 按HTML特征查找: 4个函数 161](#_Toc54854927)

[4. 按选择器查找: 161](#_Toc54854928)

[a. 如果只查找一个符合条件的元素： 161](#_Toc54854929)

[b. 如果查找多个符合条件的元素: 161](#_Toc54854930)

[5. 示例：纯客户端版购物车 161](#_Toc54854931)

[二. 修改: 166](#_Toc54854932)

[1. 获取或修改元素的内容: 3种 166](#_Toc54854933)

[(1). 获取或修改元素开始标签到结束标签之间的原始的HTML代码内容: 166](#_Toc54854934)

[(2). 获取或修改元素开始标签到结束标签之间的纯文本内容: 166](#_Toc54854935)

[(3). 获取或修改表单元素的内容: 167](#_Toc54854936)

[(4). 示例: 开关门效果 167](#_Toc54854937)

[2. 属性: 3种: 168](#_Toc54854938)

[(1). 字符串类型的HTML标准属性 168](#_Toc54854939)

[(2). bool类型的HTML标准属性 174](#_Toc54854940)

[(3). 自定义扩展属性 177](#_Toc54854941)

[3. 样式: 179](#_Toc54854942)

[Day03 180](#_Toc54854943)

[一. 修改元素: 3种 180](#_Toc54854944)

[1. 内容: 3种 180](#_Toc54854945)

[2. 属性: 3种: 180](#_Toc54854946)

[(1). 字符串类型的HTML标准属性: 2种: 180](#_Toc54854947)

[(2). bool类型的HTML标准属性: 1种: 180](#_Toc54854948)

[(3). 自定义扩展属性: 2种方式 180](#_Toc54854949)

[(4). 示例: 标签页: 180](#_Toc54854950)

[3. 样式: 184](#_Toc54854951)

[i. 先获得包含计算后的所有样式的对象: 184](#_Toc54854952)

[ii. 再用.方式从style中提取出想用的个别css属性值 184](#_Toc54854953)

[二. 添加删除元素: 190](#_Toc54854954)

[1. 添加一个新元素到网页:3步 190](#_Toc54854955)

[(1). 创建一个新的空元素对象: 190](#_Toc54854956)

[a. var 一个新元素=document.createElement("标签名") 190](#_Toc54854957)

[b. 比如: 想创建一个空的a元素: 190](#_Toc54854958)

[(2). 为新元素对象添加必要属性: 191](#_Toc54854959)

[(3). 将新元素添加到DOM树上指定父元素下: 3种: 191](#_Toc54854960)

[a. 追加到一个指定父元素下所有直接子元素之后: 191](#_Toc54854961)

[b. 插入到一个指定父元素下的一个现有子元素之前: 191](#_Toc54854962)

[c. 替换一个指定父元素下的一个现有子元素: 191](#_Toc54854963)

[i. 创建一个空的文档片段对象： 194](#_Toc54854964)

[ii. 先将所有平级子元素临时加入到文档片段对象中暂存 194](#_Toc54854965)

[iii. 最后再将整个文档片段对象，一次性添加到DOM树上指定父元素下 194](#_Toc54854966)

[三. HTML DOM常用对象（了解）: 197](#_Toc54854967)

[(1). table管着行分组: 197](#_Toc54854968)

[a. 添加行分组: 197](#_Toc54854969)

[b. 删除行分组: 197](#_Toc54854970)

[c. 获取行分组: 197](#_Toc54854971)

[(2). 行分组管着行 198](#_Toc54854972)

[a. 添加行: 198](#_Toc54854973)

[b. 删除行: 198](#_Toc54854974)

[c. 获取行: 行分组.rows[i] //获取当前行分组中i位置的行对象 199](#_Toc54854975)

[(3). 行管着格: 199](#_Toc54854976)

[(4). 示例: 动态创建表格，带删除行功能 199](#_Toc54854977)

[补: 浏览器三大对话框: 202](#_Toc54854978)

[Day04 203](#_Toc54854979)

[一. HTML DOM: 203](#_Toc54854980)

[一. 什么是BOM: 207](#_Toc54854981)

[二. window： 3个角色: 207](#_Toc54854982)

[1. 代替ECMAScript中的global充当js语言的全局作用域对象 207](#_Toc54854983)

[2. 集中保存了所有原生的直接可用的对象和函数： 207](#_Toc54854984)

[3. window代表当前浏览器窗口对象: 207](#_Toc54854985)

[(1). 获取当前窗口的大小: 2组: 207](#_Toc54854986)

[(2). 打开和关闭窗口: 207](#_Toc54854987)

[三. 打开新链接共有几种方式：4种: 208](#_Toc54854988)

[1. 在当前窗口打开新链接，可以后退 208](#_Toc54854989)

[2. 在当前窗口打开新链接，禁止后退 208](#_Toc54854990)

[3. 在新窗口打开新链接，可以同时打开多个 208](#_Toc54854991)

[4. 在新窗口打开新链接，同时只能打开一个！ 208](#_Toc54854992)

[5. 原理: 208](#_Toc54854993)

[6. 示例: 以四种不同的情况打开新链接 209](#_Toc54854994)

[四. history: 210](#_Toc54854995)

[(1). 前进一步: history.go(1) 210](#_Toc54854996)

[(2). 后退一步: history.go(-1) 210](#_Toc54854997)

[(3). 刷新: history.go(0) 210](#_Toc54854998)

[9-1\_history.html 210](#_Toc54854999)

[9-2\_history.html 211](#_Toc54855000)

[9-3\_history.html 212](#_Toc54855001)

[五. location 213](#_Toc54855002)

[1. 什么是: 保存当前地址栏中的地址信息，并提供页面跳转操作的函数 的对象 213](#_Toc54855003)

[2. 何时: 2种: 213](#_Toc54855004)

[3.. 如何: 213](#_Toc54855005)

[(1). 分段获得地址栏中url中各个部分的内容 213](#_Toc54855006)

[(2). 提供了一些函数: 213](#_Toc54855007)

[4. 示例: 获取location各个部分的内容 213](#_Toc54855008)

[六. navigator 215](#_Toc54855009)

[七. 事件: 215](#_Toc54855010)

[(1). 在HTML中绑定: ——几乎不用 215](#_Toc54855011)

[(2). 在js中用赋值方式绑定: 215](#_Toc54855012)

[总结: 230](#_Toc54855013)

[0.DOM操作万能4步: 230](#_Toc54855014)

[1. 查找元素: 4种: 230](#_Toc54855015)

[2. 修改: 3种 231](#_Toc54855016)

[3. 添加元素: 232](#_Toc54855017)

[4. 删除元素: 父元素.removeChild(要删除的元素) 232](#_Toc54855018)

[5. 事件: 232](#_Toc54855019)

[总结: this 5种: 233](#_Toc54855020)

[JQery 234](#_Toc54855021)

[一. 什么是jQuery: 234](#_Toc54855022)

[二. 如何使用: 235](#_Toc54855023)

[三. jQuery原理: 239](#_Toc54855024)

[四. 查找元素: 243](#_Toc54855025)

[Day02 250](#_Toc54855026)

[一. 查找: jQuery新增选择器 250](#_Toc54855027)

[1. 内容过滤选择器: 250](#_Toc54855028)

[(1). 什么是: 根据元素的内容中包含的关键词来匹配元素 250](#_Toc54855029)

[(2). 何时: 如果用其它选择器很难区分元素时，就可以尝试用元素的内容不同来选择元素 250](#_Toc54855030)

[(3). 包含: 4种: 250](#_Toc54855031)

[(4). 示例: 使用内容过滤选择器选择指定元素 250](#_Toc54855032)

[2. 可见性过滤: 252](#_Toc54855033)

[3. 表单元素过滤选择器: 253](#_Toc54855034)

[二. 修改: 3种东西 255](#_Toc54855035)

[1. 内容: 3种: 255](#_Toc54855036)

[(1). 原始的HTML内容: 255](#_Toc54855037)

[(2). 纯文本内容: 255](#_Toc54855038)

[(3). 表单元素的值: 255](#_Toc54855039)

[(4). 示例: 通过修改元素内容，实现表单验证 255](#_Toc54855040)

[2. 属性: 3种: 257](#_Toc54855041)

[3. 样式: 261](#_Toc54855042)

[(1). DOM中: 261](#_Toc54855043)

[(2). jq中: 261](#_Toc54855044)

[i. $元素.addClass("class名") 261](#_Toc54855045)

[ii. $元素.removeClass("class名") 261](#_Toc54855046)

[iii. var bool=$元素.hasClass("class名") 261](#_Toc54855047)

[三. 按节点间关系查找： 264](#_Toc54855048)

[1. 父子关系: 264](#_Toc54855049)

[a. 问题: children默认只能选择所有直接子元素，无法只选择个别想要的子元素 264](#_Toc54855050)

[b. 问题: children默认只能在直接子元素中查找！无法在所有后代中查找 264](#_Toc54855051)

[2. 兄弟关系: 264](#_Toc54855052)

[(1). 前一个兄弟: 264](#_Toc54855053)

[(2). 下一个兄弟: 264](#_Toc54855054)

[(3). 问题: 有时，我们希望获得除当前元素之外其余所有兄弟元素（不论前后）！ 264](#_Toc54855055)

[3. 示例: 使用按节点间关系查找，选择指定的元素 265](#_Toc54855056)

[4. 示例: 标签页效果 266](#_Toc54855057)

[四. 添加/删除/替换/克隆: 268](#_Toc54855058)

[1. 添加&替换: 268](#_Toc54855059)

[(1). 回顾: DOM中: 3步: 268](#_Toc54855060)

[(2). jq中: 2步: 268](#_Toc54855061)

[2. 删除: $元素.remove() 270](#_Toc54855062)

[3. 示例: 点按钮，添加方块，点x删除方块 270](#_Toc54855063)

[Day03 273](#_Toc54855064)

[3. 克隆: 273](#_Toc54855065)

[二. 事件: 275](#_Toc54855066)

[1. 绑定事件: 275](#_Toc54855067)

[(1). 回顾: DOM中: js中2种: 275](#_Toc54855068)

[(2). jq中: 1种 276](#_Toc54855069)

[(3). 示例: 点击按钮 发射普通子弹，点击获得奖励按钮，获得跟踪导弹，还可移除导弹 276](#_Toc54855070)

[2.事件委托： 278](#_Toc54855071)

[(1). 何时: 多个平级子元素要绑定相同的事件时 278](#_Toc54855072)

[(2). DOM中: 3步 —— 对于今后框架开发才是重要的！ 278](#_Toc54855073)

[(3). jq中: 其实也可以用DOM版的事件委托，但是jQuery提供了事件委托的简化 278](#_Toc54855074)

[(4). 示例: 选择飞机 278](#_Toc54855075)

[3. window.onload事件，页面加载后自动执行: 280](#_Toc54855076)

[4. 鼠标事件: 283](#_Toc54855077)

[a. mouseover: 当鼠标进入某个元素范围内时自动触发 283](#_Toc54855078)

[b. mouseout: 当鼠标离开某个元素范围时自动触发 283](#_Toc54855079)

[5. 模拟触发: 287](#_Toc54855080)

[(1). 什么是: 不点击元素，也能触发元素上的事件处理函数 287](#_Toc54855081)

[(2). 比如: 搜索时: 287](#_Toc54855082)

[(3). 如何: $元素.trigger("事件名") 287](#_Toc54855083)

[(4). 意为: 找到指定元素，并触发元素上指定事件上绑定的事件处理函数 287](#_Toc54855084)

[(5). 简写: 如果要触发的事件，属于十几种常用事件之一，可简写为: 287](#_Toc54855085)

[(6). 示例: 按回车也能搜索event/17\_trigger.html 287](#_Toc54855086)

[三. 动画: 2大类 289](#_Toc54855087)

[1. 简单动画: 固定写死的三种动画效果，9个函数 —— 几乎不用！ 289](#_Toc54855088)

[2. 万能动画: 292](#_Toc54855089)

[3. 排队和并发: 293](#_Toc54855090)

[4. 停止动画: 293](#_Toc54855091)

[5. jQuery新增选择器: :animated 专门选择正在播放动画的元素 293](#_Toc54855092)

[6. 动画结束后自动执行: 293](#_Toc54855093)

[7. 示例: 四颗小星星，点每个小星星播放规定的动画效果 293](#_Toc54855094)

[Day04 297](#_Toc54855095)

[一. 类数组对象操作: 297](#_Toc54855096)

[(1). 回顾: 数组家的forEach 297](#_Toc54855097)

[(2). jq中: 新增each() 297](#_Toc54855098)

[(3). 示例: 遍历ul下每li，并按要求修改符合条件的li 297](#_Toc54855099)

[(1). 回顾数组家indexOf() 299](#_Toc54855100)

[(2). jq中: 新增index() 299](#_Toc54855101)

[(3). 示例: 五星评价 299](#_Toc54855102)

[二. 添加自定义函数： 301](#_Toc54855103)

[三. 封装自定义插件: 303](#_Toc54855104)

[四. ajax 308](#_Toc54855105)

[1. jquery中已经封装好了简化版的一句话发送ajax请求的函数 308](#_Toc54855106)

[2. 固定用法: 308](#_Toc54855107)

[3. 示例: 使用ajax函数发送多种请求 308](#_Toc54855108)

[五. \*\*\*\*\*跨域: 310](#_Toc54855109)

[1. 什么是: 一个网站下的网页中请求了别的网站的资源！ 310](#_Toc54855110)

[2. 比如: 310](#_Toc54855111)

[3. 包括: 310](#_Toc54855112)

[4. 问题: 浏览器禁止ajax发送跨域请求！ 311](#_Toc54855113)

[5. 示例: 抛出跨域错误: 311](#_Toc54855114)

[6. 跨域错误的原因: 浏览器都有一个同源策略 312](#_Toc54855115)

[7. 解决: 骗 313](#_Toc54855116)

[**今日北京: 晴 21~31度** 314](#_Toc54855117)

[总结: 315](#_Toc54855118)

[$的原理: new jQuery() 4种用法: 315](#_Toc54855119)

[总结: jQuery API 三大特点: 315](#_Toc54855120)

[1. 查找元素: 316](#_Toc54855121)

[2. 修改: 3种 316](#_Toc54855122)

[8. 动画: 320](#_Toc54855123)

[9. 类数组对象操作: 320](#_Toc54855124)

[10. 添加自定义函数: 321](#_Toc54855125)

[11. 封装自定义插件: 321](#_Toc54855126)

[12. ajax 321](#_Toc54855127)

[13. 跨域: 321](#_Toc54855128)

[总结: this 6种: 322](#_Toc54855129)

[Vue 323](#_Toc54855130)

[VUE->day01 323](#_Toc54855131)

[一. 什么是VUE: 323](#_Toc54855132)

[1. 什么是: 是第三方开发的基于MVVM设计模式的渐进式的纯前端js框架 323](#_Toc54855133)

[2. 为什么: 简洁！避免大量重复的编码！ 324](#_Toc54855134)

[3. 何时: 今后只要以数据操作(增删改查)为主的项目，都可用vue开发 324](#_Toc54855135)

[二. 如何使用VUE: 324](#_Toc54855136)

[1. 2种: 324](#_Toc54855137)

[(1). 将vue.js下载到项目本地引入网页中使用: —— 前3天 学vue基础知识和原理 324](#_Toc54855138)

[(2). 公司中都是用vue脚手架代码来开发项目: ——后2天 做学子商城项目 324](#_Toc54855139)

[2. 示例: 我的第一个vue程序，分别用jquery和vue实现点击按钮修改数量功能 325](#_Toc54855140)

[(1). jQuery版: 325](#_Toc54855141)

[(2). VUE版 326](#_Toc54855142)

[3. 总结: Vue开发一个功能的基本步骤: 3步 328](#_Toc54855143)

[(1). 定义界面: 328](#_Toc54855144)

[(2). 定义仓库对象: 2个 ，data和methods 328](#_Toc54855145)

[(3). 创建Vue对象，充当快递员: 328](#_Toc54855146)

[4. 简写: 329](#_Toc54855147)

[5. 示例: 简写以上案例: 329](#_Toc54855148)

[6. 简写后的步骤: 3步 330](#_Toc54855149)

[7. 结果: 331](#_Toc54855150)

[三. MVVM: (Model, View, ViewModel) 331](#_Toc54855151)

[1. 什么是MVVM设计模式: 对前端代码的重新划分 331](#_Toc54855152)

[2. 旧前端代码，分为三部分: 331](#_Toc54855153)

[3. 问题: 331](#_Toc54855154)

[4. 新MVVM，也将代码分为3部分: 332](#_Toc54855155)

[(1). 界面（View）: 332](#_Toc54855156)

[(2). 模型对象(Model): 332](#_Toc54855157)

[(3). 视图模型(ViewModel): 快递员 332](#_Toc54855158)

[5. new Vue()如何实现MVVM设计模式: Vue的绑定原理: 访问器属性+虚拟DOM树 332](#_Toc54855159)

[(1). 访问器属性: 复习： 小程序->在线->JSCORE->day05 5. 访问器属性 get set 332](#_Toc54855160)

[(2). 虚拟DOM树: 333](#_Toc54855161)

[四. 绑定语法: 学名插值语法, Interpolation 334](#_Toc54855162)

[1. 什么是绑定语法: 334](#_Toc54855163)

[2. 为什么: 335](#_Toc54855164)

[3. 何时: 335](#_Toc54855165)

[4. 如何: 335](#_Toc54855166)

[5. 后续用法: 335](#_Toc54855167)

[6. 其实: {{}}中除了可以放单个变量之外，还能放任何有返回值的合法的js表达式！——同ES6模板字符串${}的规则 335](#_Toc54855168)

[7. 示例: 在网页中用各种语法绑定变量 335](#_Toc54855169)

[8. 问题: {{}}只支持元素内容变化，不支持属性值变化——如果元素的属性值想根据js中的变量不断变化，不能用{{}}语法，报错 336](#_Toc54855170)

[五. 指令: 337](#_Toc54855171)

[1. 什么是: 337](#_Toc54855172)

[2. 为什么: 337](#_Toc54855173)

[3. 何时: 337](#_Toc54855174)

[4. 包括: 337](#_Toc54855175)

[(1). v-bind元素的属性值可能随变量自动变化 337](#_Toc54855176)

[(2). v-show根据js中的变量值控制一个元素的显示隐藏 338](#_Toc54855177)

[(3). v-if v-else根据js中的变量值，控制两个元素二选一显示隐藏 341](#_Toc54855178)

[(4). v-if v-else-if v-else多个元素多选一显示 343](#_Toc54855179)

[5. 根据数组内容，反复创建多个相同结构的元素: v-for 345](#_Toc54855180)

[(1). 语法： 345](#_Toc54855181)

[(2). 原理: 346](#_Toc54855182)

[(3). 强调: 346](#_Toc54855183)

[(4). 其实vue中的v-for of，统一了js中的for of和for in的功能！既可遍历数字下标，又可遍历自定义下标。 346](#_Toc54855184)

[(5) 坑: Vue中禁止使用下标，修改数组中的元素值。 346](#_Toc54855185)

[(6) :key="下标"属性 347](#_Toc54855186)

[(7). 总结: 鄙视: 为什么v-for必须加:key？ 347](#_Toc54855187)

[(8). 示例: 使用v-for遍历数组和对象 347](#_Toc54855188)

[(9). 其实: v-for还会数数: 仅根据一个数字，就可反复生成指定数量的HTML元素副本，且从1开始数数，一直数到给的数字为止 349](#_Toc54855189)

[(10). 示例: 使用v-for生成分页按钮 349](#_Toc54855190)

[2. 绑定事件: v-on 350](#_Toc54855191)

[a. 无需传其它实参值，只希望获得事件对象时： 352](#_Toc54855192)

[b. $event 354](#_Toc54855193)

[c. 示例: 鄙视题: 如果同时传入自定义实参值和事件对象: 354](#_Toc54855194)

[3. v-cloak防止用户短暂看到{{}}: 356](#_Toc54855195)

[4. v-html要绑定的内容是HTML片段 358](#_Toc54855196)

[5. v-once只在首次加载时绑定一次: 359](#_Toc54855197)

[6. 防止内容中{{}}被编译: v-pre 360](#_Toc54855198)

[二. \*\*\*\*\*双向绑定: 361](#_Toc54855199)

[1. 问题: 以上12种绑定方式都无法获得用户在页面上文本框中输入的新值！ 361](#_Toc54855200)

[2. 原因: 因为前12中绑定或指令都是单向绑定: 361](#_Toc54855201)

[3. 解决: 今后只要希望随时获得用户在页面上表单元素中所做的修改时，都要用双向绑定 361](#_Toc54855202)

[4. 如何: 通常用于绑定表单元素，因为只有表单元素，用户才能在页面上修改！ 362](#_Toc54855203)

[5. 示例: 点按钮，获得文本框中输入的关键词，执行搜索操作 362](#_Toc54855204)

[6. 双向绑定原理: 363](#_Toc54855205)

[7. 示例: 使用事件模拟v-model的原理: 363](#_Toc54855206)

[8. 实现按回车搜索，边输入边搜索: 365](#_Toc54855207)

[(1). 按回车搜索: 365](#_Toc54855208)

[(2). 实现边输入边搜索: 监听函数 365](#_Toc54855209)

[(3). 示例: 实现边输入边搜索，按回车也能搜索 366](#_Toc54855210)

[9. 问题: 有些表单元素用户在修改时value是固定不变的！改的是其他属性: 367](#_Toc54855211)

[10. 解决: 分析，用户操作这些元素时，到底改的是哪个属性？ 367](#_Toc54855212)

[1). 比如: 用户选中或不选中radio，改的是radio的checked属性！ 367](#_Toc54855213)

[2). 所以: v-model应该绑定在checked属性上 367](#_Toc54855214)

[3). 原理: 367](#_Toc54855215)

[11. 示例: 在vue中选择性别: 368](#_Toc54855216)

[12. 总结: 369](#_Toc54855217)

[三. 绑定样式: 2种样式 370](#_Toc54855218)

[1. 内联样式： 370](#_Toc54855219)

[(1). 不好的做法: 把style属性当做一个巨大的字符串，仅用一个变量来绑定！ 370](#_Toc54855220)

[(2). 好的做法: vue支持用对象语法绑定style属性: 2种: 370](#_Toc54855221)

[(3). 问题: 万一有一些不需要变化的内联样式: 371](#_Toc54855222)

[(4). 示例: 分别使用两种方法绑定小飞机的style属性，通过键盘方向键操控飞机飞行 371](#_Toc54855223)

[2. 绑定class属性: 375](#_Toc54855224)

[(1). 何时: 批量修改多个css属性值时 375](#_Toc54855225)

[(2). 不好的方式: 将整个class看做一个普通的字符串属性绑定 375](#_Toc54855226)

[(3). 好的方式: 2种: 375](#_Toc54855227)

[(4). 如果一个元素上既有不变的class，又有动态改变的class: 382](#_Toc54855228)

[二. 自定义指令: 382](#_Toc54855229)

[1. 何时: 希望在页面加载时就对元素做一些初始化的DOM操作: 382](#_Toc54855230)

[2. 如何: 2步 382](#_Toc54855231)

[(1). 向Vue家里添加一个自定义指令 382](#_Toc54855232)

[(2). 使用自定义指令: 383](#_Toc54855233)

[3. 结果: 383](#_Toc54855234)

[4. 示例: 使用自定义指令，让文本框自动获得焦点: 383](#_Toc54855235)

[三. 计算属性: 384](#_Toc54855236)

[1. 什么是: 384](#_Toc54855237)

[2. 何时: 384](#_Toc54855238)

[3. 如何: 385](#_Toc54855239)

[(1). 在new Vue()中添加新成员: 385](#_Toc54855240)

[(2). 使用计算属性: 385](#_Toc54855241)

[4. 问题: 既然是一个函数，为什么不放在methods中？ 385](#_Toc54855242)

[5. 计算属性的优点: 385](#_Toc54855243)

[6. 总结: 385](#_Toc54855244)

[7. 示例: 使用计算属性实现购物车总价: 385](#_Toc54855245)

[四. 过滤器: 387](#_Toc54855246)

[1. 什么是: 387](#_Toc54855247)

[2. 为什么: 387](#_Toc54855248)

[3. 何时: 387](#_Toc54855249)

[4. 如何: 2步 388](#_Toc54855250)

[5. 示例: 使用过滤器加工性别0和1后再显示 388](#_Toc54855251)

[6. 过滤器可以带参数，根据不同的参数值，过滤出不同的结果: 389](#_Toc54855252)

[7. 多个过滤器还可以连用: <元素>{{ 变量 | 过滤器1 | 过滤器2 | ... }}</元素> 390](#_Toc54855253)

[8. 示例: 定义带参数过滤器过滤出不同语言种类的男和女，并且为性别添加图标 390](#_Toc54855254)

[五. axios: 392](#_Toc54855255)

[六. vue对象生命周期: 394](#_Toc54855256)

[1. 什么是: 394](#_Toc54855257)

[2. 包括: 4个阶段: 394](#_Toc54855258)

[3. 为什么要学习生命周期: 395](#_Toc54855259)

[4. 何时: 今后只要希望在vue对象创建和挂载过程中，自动执行某些操作时，都要用生命周期 395](#_Toc54855260)

[5. 如何: vue对象有四个生命周期，每个生命周期前后都有一对儿（2个）钩子函数(回调函数) 395](#_Toc54855261)

[(1). 创建阶段 create 395](#_Toc54855262)

[(2). 挂载阶段 mount 395](#_Toc54855263)

[(3). 更新阶段 update 396](#_Toc54855264)

[(4). 销毁阶段 destroy 396](#_Toc54855265)

[6. 示例: 演示一个vue组件的4个生命周期和8个钩子函数 396](#_Toc54855266)

[总结: 398](#_Toc54855267)

[总结: 今后所有VUE功能三步: 398](#_Toc54855268)

[七. 组件: 399](#_Toc54855269)

[1. 什么是: 拥有专属的HTML+CSS+JS+数据的独立的可重用的页面功能区域 399](#_Toc54855270)

[2. 为什么: 重用 399](#_Toc54855271)

[3. 何时: 今后只要一个功能可能会被反复使用时，都用组件 399](#_Toc54855272)

[4. 如何: 399](#_Toc54855273)

[(1). 创建组件: 399](#_Toc54855274)

[(2). 在页面上反复使用组件: 399](#_Toc54855275)

[5. 组件的加载过程: 400](#_Toc54855276)

[6. 示例: 定义计数器组件，并反复使用 401](#_Toc54855277)

[八. 组件化开发: 403](#_Toc54855278)

[1. 什么是: 拿到一个页面后，先划分组件。然后再把每个组件分给不同的人去开发！最后运行时，再将所有组件合并为一个页面运行！ 403](#_Toc54855279)

[2. 为什么: 2个原因: 403](#_Toc54855280)

[3. 何时: 今后所有的前端框架项目都是用组件化开发实现的。 403](#_Toc54855281)

[4. 如何: 403](#_Toc54855282)

[5. 问题: Vue.component()创建的组件，称为全局组件。可放在任何位置！没有限制！但是，有些子组件，离开父组件单独使用，是没有意义的！所以不应该随意在父组件外部使用子组件！ 403](#_Toc54855283)

[6. 原理: 其实vue中共有三大类组件: 403](#_Toc54855284)

[7. 解决: 今后只要限制在指定父组件内才能使用的组件都要创建为子组件，3步: 404](#_Toc54855285)

[(1). 不要用Vue.component()创建，而应该将子组件创建为一个普通的js对象。但是对象的内容，要符合vue组件的格式要求: 404](#_Toc54855286)

[(2). 在父组件对象中添加一个新的成员: 404](#_Toc54855287)

[(3). 在父组件的template中，使用组件标签引入子组件: 404](#_Toc54855288)

[8. 坑: 一旦使用子组件，则引入时，子组件必须在父组件之前先引入！ 405](#_Toc54855289)

[9. 示例: 使用组件化开发，实现todo案例的界面部分: 405](#_Toc54855290)

[10. 组件间传参: 407](#_Toc54855291)

[11. 示例: 实现todo案例中父组件给子组件传值 407](#_Toc54855292)

[九. SPA: Single Page Application 409](#_Toc54855293)

[1. 什么是单页面应用: 整个应用程序只有一个唯一完整的HTML页面。其它所谓的"页面"，其实都是组件而已。所谓的"页面跳转"，其实只是在一个HTML中切换不同的组件显示而已。 410](#_Toc54855294)

[2. 为什么: 410](#_Toc54855295)

[3. 何时: 今后，只要使用框架开发项目，用的都是单页面应用！ 411](#_Toc54855296)

[4. 缺点: 首屏加载极慢。 411](#_Toc54855297)

[5. 如何: 3个主要的步骤: 411](#_Toc54855298)

[(1). 创建一个唯一完整的HTML文件 411](#_Toc54855299)

[(2). 创建多个页面组件 411](#_Toc54855300)

[(3). 在独立的router.js文件中创建路由器对象和路由字典列表： 411](#_Toc54855301)

[6. 结果: 只要地址栏相对路径一变，router对象会自动获得新的相对地址。自动去routes中查找对应的组件对象。最后用找到的组件对象代替HTML文件中<router-view>的位置 412](#_Toc54855302)

[7. 强调: Vue的路由器采用锚点地址，作为客户端导航的相对地址: 412](#_Toc54855303)

[8. 示例: 实现包含两个页面的基本的SPA应用 412](#_Toc54855304)

[9. 404页面： 414](#_Toc54855305)

[10. 页头: 414](#_Toc54855306)

[11. 路由跳转: 2种: 414](#_Toc54855307)

[12. 路由传参: 3步: 414](#_Toc54855308)

[13. 问题: 那么多js文件都放在同一个js文件夹下，非常乱 415](#_Toc54855309)

[总结: 416](#_Toc54855310)

[总结: 420](#_Toc54855311)

[1. 绑定语法(1种)和指令(13种): 420](#_Toc54855312)

[2. 绑定样式: 420](#_Toc54855313)

[3. 自定义指令: 2步 421](#_Toc54855314)

[4. 计算属性: 421](#_Toc54855315)

[5. 过滤器: 421](#_Toc54855316)

[6. axios: 422](#_Toc54855317)

[7. vue生命周期: 4个阶段 8个钩子函数 422](#_Toc54855318)

# Jscore

**Day01正课**:

1. 正则表达式

2. String正则相关的函数

# 一. 正则表达式: PPT —— 下载: ftp

<http://code.tarena.com.cn/WEBCode/wfd2003/10_JSCORE/day01/>1\_regexp.pptx

# 二. String正则相关的函数: 3件事儿: (高频面试题)

## 1. 查找敏感词: 4种情况:

## (1). 查找一个固定不变的敏感词出现的位置:

a. var i=str.indexOf("敏感词", fromi)

位置 的 "敏感词"

b. 意为: 在字符串str中，从fromi位置开始向后，查找第一个敏感词出现的位置

c. 简写: 省略第二个参数，默认从0位置开始找。

d. 返回值:

1). 如果在字符串中找到敏感词，则返回敏感词第一个字所在的下标位置i

2). 如果没找到敏感词，则返回-1

e. 问题: 不支持正则表达式，只能查找一种固定不变的敏感词——查找太弱！

f. 示例: 使用indexOf查找一个敏感词的位置:

|  |
| --- |
| <script>  //请用户输入一条消息内容  var msg=prompt("请输入消息内容");  // console.log(`用户输入了: ${msg}`);  //在消息内容中查找是否包含敏感词"我草"  var i=msg.indexOf("我草");  //如果找到敏感词（找到敏感词的位置i不是-1）  if(i!=-1){  //就在网页上显示"在位置i发现敏感词"  //向网页中 写 (`一条html代码`)  document.write(`<h1 style="color:red">在位置${i}发现敏感词，禁止发送</h1>`)  }else{//否则如果没找到敏感词  //就在网页上显示"然哥说：消息内容"  document.write(`<h1 style="color:green">然哥说:${msg}</h1>`)  }  </script>  运行效果: |

## (2). 用正则表达式模糊查找多种敏感词出现的位置:

a. var i=str.search(/正则表达式/i);

b. 意为: 在字符串str中，查找第一个符合正则表达式要求的敏感词的下标位置i

c. 返回值:

1). 如果在字符串中找到敏感词，则返回敏感词第一个字所在的下标位置i

2). 如果没找到敏感词，则返回-1

d. 问题: js中正则表达式默认区分大小写的。如果正则中写的是小写字母，则只能匹配小写字母。无法匹配大写字母。

e. 解决: 只要在正则表达式后的/之后加一个后缀: i —— ignore 忽略大小写

f. 所以，起始将来在js中写正则表达式，根本不用考虑大小写的差别。只要加上i，轻松实现既能防住大写字母，又能防住小写字母。

g. 示例: 使用search和正则表达式查找敏感词的位置：

|  |
| --- |
| <script>  //请用户输入一条消息内容  var msg=prompt("请输入消息内容");  // console.log(`用户输入了: ${msg}`);  //在消息内容中查找符合正则表达式要求的敏感词的位置  var i=msg.search(/([我卧]|wo)\s\*([艹槽草操]|cao)/i);  //如果找到敏感词（找到敏感词的位置i不是-1）  if(i!=-1){  //就在网页上显示"在位置i发现敏感词"  //向网页中 写 (`一条html代码`)  document.write(`<h1 style="color:red">在位置${i}发现敏感词，禁止发送</h1>`)  }else{//否则如果没找到敏感词  //就在网页上显示"然哥说：消息内容"  document.write(`<h1 style="color:green">然哥说:${msg}</h1>`)  }  </script>  运行效果: 同上 |

2. 替换敏感词:

3. 切割字符串:

**Day02**正课:

1. String提供的正则相关的函数

2. RegExp对象

# 一. String提供的正则相关的函数: 3件事:

## 1. 查找关键词: 4种情况

## (1). 查找一个固定的关键词出现的位置:

var i=str.indexOf("关键词",fromi)

问题: 只能查找一种固定的关键词，只要关键词一变，就查不出来了。

## (2). 用正则模糊查找多种关键词的位置:

var i=str.search(/正则/i)

问题: 只能返回关键词的位置，无法返回关键词的内容！

## (3). 用正则模糊查找多种关键词的内容:

a. var arr=str.match(/正则/i)

b. 意为: 在str中查找第一个符合正则表达式要求的关键词的内容和位置

c. 返回值:

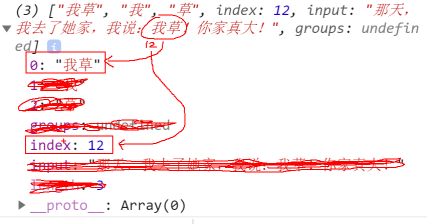
1). 如果match找到关键词，就返回一个数组:

arr : [ "0": "关键词内容", "index":"关键词的下标位置i" ]

如果想获得本次找到的敏感词的内容: 标准: arr["0"] 可简写为 arr[0]

如果想获得本次找到的敏感词的位置: 标准: arr["index"] 可简写为 arr.index

0和index禁止改名！因为match()是内置的函数，返回的结果格式也是固定的！不是人为可以修改的！所以我们只能遵照match()的要求使用！而无法自定义返回值的结构。



2). 如果match没找到关键词，返回null —— 造成错误最多的根源！

强调: 将来如果一个函数有可能返回null时，在使用这个函数的返回值之前，都必须先判断返回值是不是null，再决定如何使用返回值！必须为null的情况提供备选方案，保证程序在任何情况下都不会报错！

d. 示例: 既查找敏感词的内容，又查找敏感词的位置

|  |
| --- |
| <script>  //请用户输入一条消息内容  var msg=prompt("请输入消息内容");  // console.log(`用户输入了: ${msg}`);  //在消息内容中查找符合正则表达式要求的敏感词的位置  var arr=msg.match(/([我卧]|wo)\s\*([艹槽草操]|cao)/i);  //console.log(arr);  //arr : ["0":"我草", ..., "index":12]  //如果找到敏感词（找到敏感词的位置i不是-1）  if(arr!=null){  //就在网页上显示"在位置i发现敏感词"  //向网页中 写 (`一条html代码`)  // document.write(`<h1 style="color:red">在位置${arr["index"]}发现敏感词${arr["0"]}，禁止发送</h1>`);  document.write(`<h1 style="color:red">在位置${arr.index}发现敏感词${arr[0]}，禁止发送</h1>`);  }else{//否则如果没找到敏感词  //就在网页上显示"然哥说：消息内容"  document.write(`<h1 style="color:green">然哥说:${msg}</h1>`)  }  </script>  运行效果： |

e. 问题: 默认js中的正则表达式只要找到第一个符合要求的敏感词就下班了！不再继续往后找！

f. 解决: 在正则表达式第二个/后加g，意为global，全部

g. 新问题: match()一旦加上g之后，虽然可以返回所有关键词的内容，但是无法返回每个关键词的位置了！

h. 示例: 查找一个字符串中所有关键词的内容

|  |
| --- |
| <script>  var str="老师:请用小红 我的 朋友造句。小亮:小红是我的朋友。小然:朋友！小红是我的！";  //用正则表达式找出str中\*所有\*以小字开头的人名  var arr=str.match(/小[\u4e00-\u9fa5]/g);  console.log(arr);  </script>  运行效果： |

## (4). 既查找每个敏感词的内容，又查找每个敏感词的位置:

String家的函数，无法实现如此复杂的查找功能。

只能依靠reg.exec()

|  |
| --- |
| 总结:函数返回值有什么规律:  1. 如果原函数返回下标位置i，则如果找不到返回: -1  2. 如果原函数返回一个数组或一个对象，则如果找不到返回: null |

## 2. 替换关键词: 2种:

(1). 简单替换: 将所有找到的敏感词都替换为统一的新值

a. str=str.replace(/正则/ig, "新值");

b. 意为: 将str字符串中所有符合正则要求的关键词都替换为统一的新值！

c. 强调: 要向获得替换后的新字符串，必须用=接住replace的返回值新字符串才行

d. 示例: 将字符串中所有关键词都替换为统一的新值

|  |
| --- |
| <script>  //如果找到关键词  var str = "老师:请用小红 我的 朋友造句。小亮:小红是我的朋友。小然:朋友！小红是我的！";  //如果找不到关键词:  // var str="you can you up,no can no bibi";  //用正则表达式找出str中\*所有\*以小字开头的人名  var arr = str.match(/小[\u4e00-\u9fa5]/g);  //arr=null  console.log(arr);  //希望将所有以小字开头的人名都替换为\*\*  str = str.replace(/小[\u4e00-\u9fa5]/g, "\*\*");  //不报错，而是返回原字符串，什么也不干！  console.log(str);  //如果match找到了敏感词  if (arr != null) {  //才输出共替换了几处？  console.log(`共替换了${arr.length}处`);  // null.length 报错: 无法读取null的length属性  } else {//否则如果match没找到敏感词  //就输出共替换0处  console.log(`共替换了0处`)  }  </script>  运行结果： |

(2). 高级替换: 根据每次找到的敏感词不同，动态替换成不同的新值

a. str=str.replace(/正则/ig, function(keyword){

return 根据keyword收到的关键词不同，动态返回不同的新值替换到字符串中

})

b. 意为: 将str中每个符合正则要求的敏感词都交给一个回调函数中去判断应该替换为何种新值。

c. 原理:

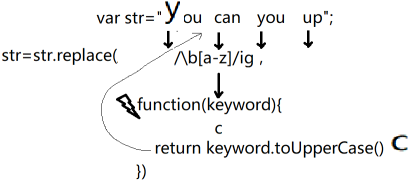
1). replace会用正则查找str中每个关键词

2). replace每找到一个关键词，就自动调用一次回调函数，并自动将本次找到的关键词传递给回调函数的keyword形参变量。

3). 回调函数内部，根据每次传入的关键词不同，动态选择不同的新值返回给replace

4). replace会将回调函数返回的新值，替换进字符串中本次找到的关键词位置。

5). 最后replace会将替换后的新字符串返回，并保存在=左边的变量中



d. 示例: 替换一句话中每个单词首字母为大写

|  |
| --- |
| <script>  //鄙视题: 将每个单词首字母改为大写  var str="you can you up";  //每找到一个敏感词就自动调用一次回调函数  str=str.replace(/\b[a-z]/ig,function(keyword){  //打桩: 输出回调函数调用次数、关键变量的值、函数的返回值  console.log(`自动调用了一次回调函数,keyword收到${keyword},返回新值:${keyword.toUpperCase()}`);  //keyword依次接到: y, c, y, u  return keyword.toUpperCase()  //return 依次返回: Y, C, Y, U  })//replace会将回调函数返回的值替换到字符串中关键词的位置上。  console.log(str);  </script>  运行效果:  自动调用了一次回调函数,keyword收到y,返回新值:Y  自动调用了一次回调函数,keyword收到c,返回新值:C  自动调用了一次回调函数,keyword收到y,返回新值:Y  自动调用了一次回调函数,keyword收到u,返回新值:U  You Can You Up |

(3). 衍生操作：删除关键词，将关键词替换为""，就算删除了

a. str=str.replace(/正则/ig, "")

b. 示例: 去掉字符串开头和结尾的空字符

|  |
| --- |
| <script>  //鄙视: 定义函数去掉字符串开头的和结尾的空字符  // 修剪 字符串开头的空字符  function trimLeft(str){  return str.replace(/^\s+/,"");  }  // 修剪 字符串结尾的空字符  function trimRight(str){  return str.replace(/\s+$/,"");  }  // 修剪 字符串开头和结尾的空字符  function trim(str){  return str.replace(/^\s+|\s+$/g,"")  }  //测试:  var str=" zhang dong ";  console.log(trimLeft(str));//"zhang dong "  console.log(trimRight(str));//" zhang dong"  console.log(trim(str));//"zhang dong"  //ES5/ES6: 内置了,不用自定义可直接使用！  console.log(str.trimLeft());//"zhang dong "  console.log(str.trimRight());//" zhang dong"  console.log(str.trim());//"zhang dong"  </script>  运行结果:  zhang dong  zhang dong  zhang dong  zhang dong  zhang dong  zhang dong |

## 切割字符串:

将一个字符串，切割为多段子字符串，2种:

(1). 简单切割: 切割符是固定不变的

a. var arr=str.split("切割符")

b. 意为: 将字符串str，按指定的切割符，切割成多段子字符串，保存在一个数组中返回

c. 强调: 切割后的结果中，不包含切割符的！

(2). 复杂切割: 切割符不是固定不变的，但是可以找到规律

a. var arr=str.split(/正则/i)

b. 意为: 将字符串str，按正则匹配到的切割符，切割成多段子字符串，保存在一个数组中返回

c. 强调:

1). 切割后的结果中，不包含切割符的！

2). 不用加g，split也能找到所有符合条件的切割符！

(3). 固定套路: 打散一个字符串为字符数组:

var arr=str.split("")

(4). 示例: 按规定切割字符串

|  |
| --- |
| <script>  var email="zhangdong@tedu.cn";  // 用户名 | 域名  //希望获得电子邮件中的用户名和域名  //按@切割电子邮件  var arr=email.split("@");  console.log(arr);  var uname=arr[0];  var domain=arr[1];  var str="you can you up";  //获得一个英文句子中单词的列表  //按空格切割字符串str  var words=str.split(" ");  console.log(words);  var str="you can you up";  //按一个以上的连续空格来切割  var words=str.split(/\s+/);  console.log(words);  //回顾:  var arr=[1,2,3];  arr.reverse();  console.log(arr);  //鄙视: 翻转一个字符串:  var str="hello";  //str.reverse();//报错: str.reverse不是一个函数  //先把str打散为字符数组  var arr=str.split("");  console.log(arr);  //调用数组家的函数翻转数组  arr.reverse();  //要将翻转后的字符数组再拼接回字符串  //str=arr.toString();//错误: toString()只能用逗号连接每个字  str=arr.join("");//无缝拼接  console.log(str);//olleh  </script>  运行结果：  ["you", "can", "you", "up"]  ["you", "can", "you", "up"]  [3, 2, 1]  ["h", "e", "l", "l", "o"]  olleh |

# 二. RegExp对象:

1. 什么是: 专门在内存中保存一条正则表达式，并提供用正则表达式执行验证和查找功能的函数 的对象

2. 为什么: 因为正则表达式是一种独立于编程语言之外的全新的语法，js语言不认识正则表达式，就需要一种专门的对象能将正则表达式翻译为js能理解的方式。让js语言也能使用正则表达式了。

3. 何时: 只要在js中使用正则表达式，都要先创建正则表达式对象

## 4. 如何创建正则表达式对象: 2种方式:

(1). 标准写法: var reg=new RegExp("正则表达式", "ig")

a. 强调: ""中不要再写//了！

b. 何时: 少数情况下，如果正则表达式需要动态拼接生成，就被迫选择new RegExp()方式。

c. 因为: js中字符串可以随意拼接！

d. 示例: 根据数组内容，动态生成正则表达式

|  |
| --- |
| <script>  //假设每次都要从服务器端动态获取了一个新的包含关键词的数组  //希望，数组中有什么关键词，咱们就要防住什么关键词  var arr=["青天","明月","紫烟"];//服务器上关键词列表可能发生变化  console.log(arr.join("|"));  //目的: 也不需要修改下方的js程序，就可屏蔽新的关键词列表。  //请用户输入一条消息  var msg=prompt("请输入消息内容");  //定义正则表达式防住arr数组中包含的关键词  //var reg=/arr.join("|")/; 错误://之间不认识js语句  var reg=new RegExp(arr.join("|"));  // "明月|青天|紫烟"  // new RegExp( "正则" )  console.log(reg);  //用正则表达式查找msg中是否包含关键词  var i=msg.search(reg);  //如果msg中包含关键词  if(i!=-1){  //则在网页中显示: 包含关键词禁止发送  document.write(`<h1 style="color:red">发现敏感词，禁止发送！</h1>`)  }else{//否则如果msg中不包含关键词  //则在网页中显示: 消息内容  document.write(`<h1 style="color:green">然哥说: ${msg}</h1>`)  }  </script>  运行效果: |

(2). 简写: var reg=/正则/ig

a. 何时: 绝大多数情况下，如果正则表达式是固定不变的，比如:手机号规则、电子邮件规则等，首选//创建。

b. 因为: //中是正则表达式的地盘，不认识js，不支持动态拼接字符串的语句。

## 5. RegExp提供的函数: 2个

(1). 验证字符串格式是否符合正则表达式要求:

a. var bool=reg.test(str)

b. 意为: 用这规则表达式reg去检查字符串str是否符合正则表达式格式要求

c. 何时: 今后只要网页中格式的验证，都用reg.test()

比如: 验证手机号格式、身份证号格式、电子邮件格式等

d. 返回值:

1). 如果字符串str的格式符合正则表达式reg的要求，则返回true

2). 如果字符串str的格式不符合正则表达式reg的要求，则返回false

e. 问题: 默认只要str中包含符合reg要求的部分内容，则test就返回true

比如: var reg=/\d{6}/

reg.test("abc123456") 返回true

reg.test("123456abc") 返回true

f. 解决: 今后凡是验证格式的正则表达式必须同时前加^后加$

表示，从头到尾必须和规则完全匹配才行！

比如: var reg=/^\d{6}$/ 意为从头到尾中间只能有6位数字！

reg.test("abc123456") 返回false

reg.test("123456abc") 返回false

g. 示例: 验证密码格式:

|  |
| --- |
| <script>  //请用户输入一个密码:  var pwd=prompt("请输入密码");  //定义一个正则表达式描述密码的格式: 6位数字  var reg=/^\d{6}$/; //从头到尾只能是6位数字  //用reg检查用户输入的密码pwd是否符合格式要求:  var bool=reg.test(pwd);  //如果检查通过  if(bool==true){  //就在页面输出: 密码可用  document.write(`<h1 style="color:green">密码可用</h1>`)  }else{//否则如果检查未通过  //就在页面输出: 密码格式不正确  document.write(`<h1 style="color:red">密码格式不正确</h1>`)  }  </script>  运行结果: |

(2). 既查找每个敏感词的内容，又查找每个敏感词的位置:

a. var arr=reg.exec(str);

b. 意为: 在str中查找下一个符合reg要求的关键词的内容和位置

c. 返回值: 每次查找的返回值和match不加g的情况完全一样

1). 如果找到关键词，就返回一个数组

arr : [ "0":"本次找到的关键词", ... , "index":"本次找到的关键词的下标位置" ]

2). 如果找不到关键词，就返回null

d. reg.exec() vs str.match()

1). str.match()不加g时，永远只能返回第一个关键词的内容和位置。无法自动找下一个关键词

2). reg.exec()在加g时，每找到一个关键词，再次调用reg.exec()时，可以自动继续找下一个关键词！直到返回null，说明找完了.

e. 固定套路: 循环+reg.exec()可找出一句话中所有关键词的内容和位置

//不管有没有关键词，都要先查找一次！才知道有没有关键词，有没有必要继续

var reg=/正则/g;

do{

var arr=reg.exec(str);

if(arr!=null){//如果本次找到了关键词，才有必要输出关键词的内容和位置

//console.log(arr);

console.log(`在位置${arr.index}发现敏感词${arr[0]}`)

}//否则如果没找到关键词，就什么也不输出。

}while(arr!=null);//如果arr!=null，说明本次找到关键词了，才有必要继续找下一个

f. 强调: 想让reg.exec()找所有，必须加g

g. 示例: 查找一句话中所有关键词的内容和位置

|  |
| --- |
| <script>  var str = "老师:请用小红 我的 朋友造句。小亮:小红是我的朋友。小然:朋友！小红是我的！";  //定义正则描述小字开头的人名  //用正则表达式找出str中\*所有\*以小字开头的人名和位置  var reg=/小[\u4e00-\u9fa5]/;  //用match()做: 无法实现找所有的要求  var arr = str.match(reg);  console.log(arr);//返回第一个人名和位置  var arr = str.match(reg);  console.log(arr);//只能返回第一个人名和位置  var reg=/小[\u4e00-\u9fa5]/g;  //do while  //不管有没有关键词，都要先查找一次！才知道有没有关键词，有没有必要继续  do{  var arr=reg.exec(str);  if(arr!=null){//如果本次找到了关键词，才有必要输出关键词的内容和位置  //console.log(arr);  console.log(`在位置${arr.index}发现敏感词${arr[0]}`)  }//否则如果没找到关键词，就什么也不输出。  }while(arr!=null);//如果arr!=null，说明本次找到关键词了，才有必要继续找下一个  </script>  运行效果:  ["小红", index: 5...]  ["小红", index: 5...]  在位置5发现敏感词小红  在位置16发现敏感词小亮  在位置19发现敏感词小红  在位置27发现敏感词小然  在位置33发现敏感词小红 |

**补**: **字符串是不可变类型:**

1. 字符串一旦创建，内容不可改变！只能整体替换！

2. 所以字符串家的函数，都无权直接修改原字符串，只能返回加工后的新字符串。原字符串始终保持不变！

3. 所以要想获得字符串家函数返回的处理结果，必须用=接住返回值！否则，处理结果没人要！就自动扔了！



**补: 数组本质**: js中根本就没有索引数组。js中一切数组底层都是关联数组。所有数组元素的下标都是字符串类型！

1. 其实，访问所有数组元素的标准写法，都可写成: 数组名["下标名"]

比如: var arr=[1,2,3]; arr["0"] 访问数组中"0"位置的元素

再比如: var ym=[]; ym["math"]=89; ym["math"]访问数组中"math"位置的元素

2. 简写: 自动翻译

1). 自定义名称的下标，可简写为: 数组名.下标名

比如: ym["math"] 可简写为 ym.math 会被自动翻译回 ym["math"]

2). 数字下标，~~不能简写为.下标名~~，因为和小数的写法冲突。应该只去掉""，保留[]。数组名[下标]

比如: arr.1 —— 错误, 和小数点冲突

arr[1] —— 正确

**Day03**正课: Function

1. 回顾函数:

2. 重载

3. 匿名函数

4. 作用域和作用域链

5. \*\*\*\*\*闭包:

# 一. 回顾函数：

1. 什么是: 程序中保存一段可重用的代码段的程序结构，再起一个名字

2. 何时: 只要一项任务的代码，可能被反复使用，都要封装为函数。

3. 如何定义: 声明方式:

function 函数名(形参变量列表){

要反复执行的代码段

return 返回值

}

4. 调用函数:

var 变量=函数名(实参值列表)

强调: 一个函数！只定义，不调用！是不执行的！

5. 示例: 定义一个做菜的函数，可以传入食材，并返回做好的饭菜

|  |
| --- |
| <script>  //想定义一个会做菜的函数  function cook(cai,dan,mian){ //只会做固定一种菜: 西红柿鸡蛋面  console.log(`炒${cai}`);  console.log(`炒${dan}`);  console.log(`${cai}和${dan}一起炒`);  console.log(`煮${mian}`);  return `香喷喷的${cai}${dan}${mian}`  };  //周一:  var wan=cook("西红柿","鸡蛋","面");  console.log(`一碗${wan}`);  //周二:  var wan=cook("辣条","螺蛳","粉儿");  console.log(`一碗${wan}`);  </script>  运行结果:  炒西红柿  炒鸡蛋  西红柿和鸡蛋一起炒  煮面  一碗香喷喷的西红柿鸡蛋面  炒辣条  炒螺蛳  辣条和螺蛳一起炒  煮粉儿  一碗香喷喷的辣条螺蛳粉儿 |

6. **鄙视**: js中定义函数共有几种方式，差别是什么？3种:

对应小视频: 小程序->在线->JSCORE->day03 2. 3. 两个视频 三遍起

(1). 声明方式:

function 函数名(形参变量列表){

要反复执行的代码段

return 返回值

}

问题: 会被声明提前:

在程序开始执行前

js引擎会先将var声明的变量和function声明的函数，

提前到当前作用域的顶部，优先集中创建。

但是，赋值留在原地

缺点: 打乱程序的执行顺序，产生歧义

(2). 赋值方式:

a. var函数名=function (形参变量列表){

要反复执行的代码段

return 返回值

}

b. 优点: 不会被声明提前，保持了程序正常的执行顺序，不会产生歧义

c. 揭示了js中函数的本质:

1). js中函数名也只是一个普通的变量名而已（几乎和var 变量没差别）

2). 函数其实是一个对象

3). 函数名变量保存着函数对象

d. 示例: 声明提前笔试题

|  |
| --- |
| <script>  // function fun(){ console.log(1) }  // fun(); //2  // function fun(){ console.log(2) }  // fun(); //2  //就想输出1和2  var fun=function(){ console.log(1) }  fun(); //1  var fun=function(){ console.log(2) }  fun(); //2  </script>  运行效果：  1  2 |

(3). 用new创建: (因为格式极其晦涩，极少使用，了解即可)

a. var 函数名=new Function("形参", ..., "函数体; return 返回值")

b. 更揭示了函数的本质: 函数其实也是一个对象，函数名仅仅是一个变量。

c. 总结: function其实是new Function()的简写！(等效于)

d. 示例: 验证函数的本质也是一个对象

|  |
| --- |
| <script>  //function add(a,b){//除了会被声明提前之外，和var fun=function()无差别  //var add=function(a,b){ return a+b; } //function其实是new Function()的简写  var add=new Function("a","b","return a+b");//最底层本质  console.log(add(3,5));//8  var sum=add;  //sum=function(a,b){return a+b};  console.log(sum(3,5));//8  var a=function(){ console.log(1) };//new Function() 被分配内存地址  var b=function(){ console.log(1) };//new Function() 被分配另一个新的内存地址  console.log(a==b);//false  //==比较两个对象时，不再隐式转换，而直接比较两个对象的地址是否相同。  </script>  运行效果：  8  8  false |

# 二. 重载(overload):

对应小视频: 小程序->在线->JSCORE->day03 4. 重载 overload pay

1. 什么是: 相同函数名，不同形参列表的多个函数，在调用时，可根据传入的实参值不同，自动选择匹配的函数版本来执行不同的逻辑。

2. 为什么: 减少函数名的个数，减轻调用者的负担！

3. 何时: 只要一件事，希望根据传入实参的不同，执行不同的逻辑时，都首选重载方式实现

4. 如何:

(1). 问题: js不支持传统的重载写法：js中不允许多个同名函数同时存在。如果同时存在，最后定义的同名函数会覆盖之前定义的所有同名函数

(2). 解决: 借助arguments对象来变通实现重载的效果

a. 什么是arguments对象:

1). 每个函数中自带的 —— 不用我们手工创建，就可直接使用！

2). 专门接收传入函数的所有实参值的 —— 作用/内容

3). 类数组对象 —— 长的像数组的对象

i. 相同点: ①下标, ②.length, ③遍历

ii. 不同点: 类型不同。类数组对象是Object家的孩子，类数组对象无法使用数组家的函数。

b. 如何实现重载:

### 1). 只定义一个函数，其中包含多种要执行的逻辑

### 2). 但是不要写任何形参变量，将来js中的函数即使一个形参也不写，arguments对象也可以自动替我们接住传入的所有实参值。

### 3). 可通过判断arguments中实参值的个数和实参值的内容不同来动态决定本次调用应该选择执行哪种逻辑

### 4). 可使用arguments[下标]方式获得本次调用时传入的第i个实参值

## 5. 示例: 定义一个支付函数pay，支持三种支付方式

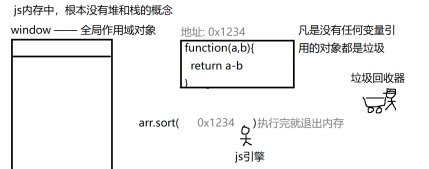
|  |
| --- |
| <script>  //定义一组支付的函数: 3种可能  //1. 手机支付: 不需要给收款员提供任何东西，就可手机支付  //2. 现金支付: 需要给收款员一笔现金  //3. 刷卡支付: 需要提供卡号和密码  //pay("6553 1234","123456");  //pay( 100 );  //pay( ↓ );  function pay( ){  // ↓ ↓ ↓  //arguments{ }.length  // 0 1 2...  //如果没有传入参数，就手机支付  if(arguments.length==0){  console.log(`手机支付...`);  }else if(arguments.length==1){//如果传入一个参数，就现金支付  console.log(`现金支付...收您${arguments[0]}元`)  }else{//否则就刷卡支付  console.log(`刷卡支付...从您卡号${arguments[0]}，扣款成功!`)  } //本例: 只考虑最多2个实参值的情况。  }  pay();//想手机支付:  pay(100); //想现金支付  pay("6553 1234","123456");//想刷卡支付  </script>  运行效果：  手机支付...  现金支付...收您100元  刷卡支付...从您卡号6553 1234，扣款成功! |

# 三. 匿名函数:

对应小程序视频: 小程序->在线->JSCORE->day03 5 匿名函数 防止全局污染

1. 什么是: 在定义函数时，不指定函数名的函数

2. 为什么: 匿名函数比有名称的函数更节约内存 —— 没有函数名，就没有变量引用，用完，内存就释放了！



3. 何时: 如果一个函数只使用一次时，都用匿名函数

4. 如何: 2种情况

(1). 回调函数:

a. 什么是: 我们定义的函数，但是我们自己不调用，而是交给另一个函数，由另一个函数根据需要自动调用。

b. 比如:

1). 数组排序: var arr=[12,3,123,23,1,2]; arr.sort(function(a,b){return a-b});

2). 高级替换:

str=str.replace(/\b[a-z]/ig, function(keyword){ return keyword.toUpperCase()})

c. 总结: 今后绝大多数回调函数，都优先使用匿名函数来定义

(2). 匿名函数自调:

a. 什么是匿名函数自调: 定义一个匿名函数后，立刻调用该函数，调用后立刻释放！

b. 为什么: 避免使用全局变量，造成全局污染！

c. 何时: 今后，所有js程序，都要放在一个匿名函数自调中包裹起来！

d. 如何: 标准写法: (function(){ ... })()

e. 说明: 用不用匿名函数自调，不影响功能的执行！只改变变量的性质而已。

f. 示例: 使用匿名函数自调避免使用全局变量

|  |
| --- |
| <script>  (function(){  //在页面开始加载时，弹出提示框: 开始加载页面内容,at:时间  //start变量就变成了局部变量  var start=new Date();//获得当前系统时间  alert(`开始加载页面内容,at:${start.toLocaleString()}`);  //复习第一阶段Date相关内容  })();//所有局部变量在匿名函数自调后，会被自动释放！因为局部变量没有保存在全局。（稍后讲）  </script>  <script>  //在页面加载完成是，弹出提示框：页面加载完成,at:时间  var end=new Date();//获得当前系统时间  alert(`页面加载完成,at:${end.toLocaleString()}`)  //复习第一阶段Date相关内容  //按理说: start变量和end变量在页面加载完之后，使命就结束了！  //但是: end变量却还遗留在window中，不会被释放！  </script>  运行效果： |

# 四. 作用域(scope)和作用域链:

对应小程序视频: 小程序->在线->JSCORE->day03 7. 原理 作用域和作用域链

1. 什么是作用域: 2种解释:

(1). 用法: 作用域是一个变量的可用范围

(2). 本质: 作用域是保存变量的特殊对象

2. 为什么: 避免不同范围的变量之间互相干扰

## 3. js中包括2种作用域:

(1). 全局作用域：

a. 什么是: 专门保存全局变量的特殊对象——window

b. 特点:

1). 在程序的任何一个角落都可以随时访问到全局作用域。

2). 全局作用域不会自动释放！除非关闭浏览器窗口。

c. 什么是全局变量: 保存在全局作用域中，可被程序任何角落随时访问到的变量

d. 全局变量特点: 2个

1). 随处可用！——缺点: 极易被污染！

2). 可反复使用 ——优点

(2). 函数作用域：

a. 什么是: 专门保存局部变量的特殊对象——？

b. 特点:

1). 当函数调用时才临时创建函数作用域对象，函数调用后函数作用域对象被自动释放！下次再调用同一个函数，也会重新创建一个全新的函数作用域对象.

2). 只有函数调用时，函数内的代码才能使用函数作用域对象。

c. 什么是局部变量: 保存在函数作用域当中，仅函数内的代码可以访问的变量

d. 局部变量特点:

1). 仅函数内可用！——优点: 不会被污染！

2). 不可重用 ——缺点

## 4. 作用域链:

### (1). 什么是作用域链:

由多级作用域，逐级引用，形成的链式结构

### (2). 函数从定义到调用，再到调用完成所经历的过程:

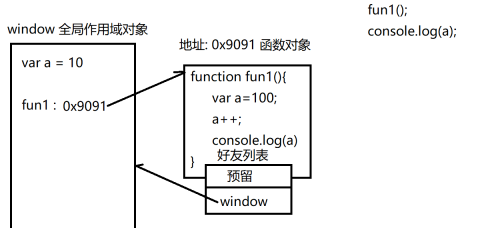
#### a. 定义函数时:

1). 创建函数对象保存函数内的代码片段

2). 每个函数对象身上都有一个"好友列表"，记录了每个函数可用的所有作用域对象，普通函数的好友列表包含2个格子:

i. 离函数自己近的格子，暂时预留

ii. 离函数自己稍远的格子存window。



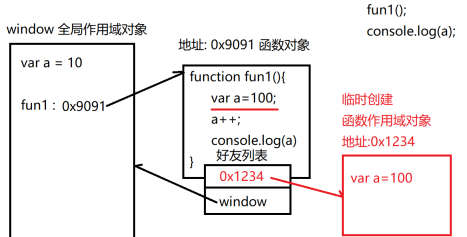
#### b. 调用函数时:

1). 临时创建本次函数调用的函数作用域对象

2). 将函数体中定义的局部变量临时创建到函数作用域对象中保存

强调: 局部变量有2种: ①在函数内var出的变量; ②形参也属于局部变量

3). 将临时创建的函数作用域对象的地址，临时保存到函数的好友列表中离函数自己近的格子中.



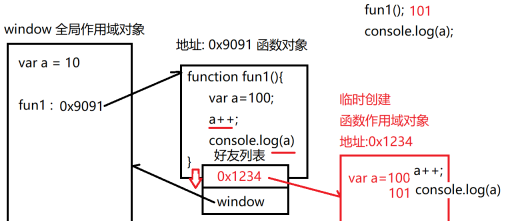
#### c. 执行函数过程中:

1). 如果函数体中用到了某个变量，优先找离自己近的函数作用域对象中的局部变量使用

2). 按顺序: 就近原则: 先局部，后全局。

i. 如果局部有该变量，则优先使用局部变量

ii. 除非函数作用域中找不到所需的变量，才被迫去window中查找全局变量使用。



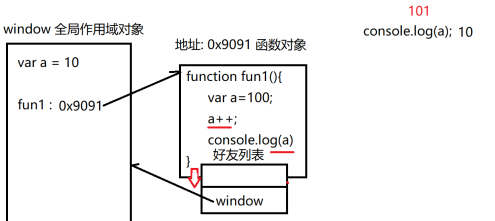
#### d. 函数调用后:

1). 好友列表中离自己近的格子会清空z

2). 导致临时创建的函数作用域对象被释放

但是保存着函数定义的函数对象是不会被释放的！

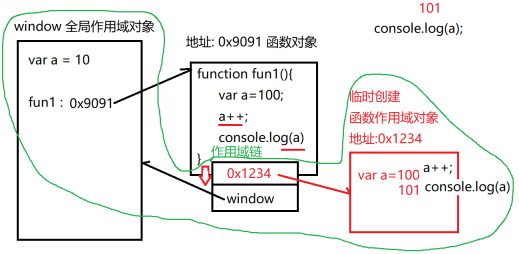
3). 导致函数本次使用的所有局部变量一同被释放——局部变量不可重用！



### (3). 以上过程中函数对象的"好友列表"，学名就叫作用域链:

a. 保存着: 一个函数可用的所有的作用域以及变量

b. 控制着: 变量的使用顺序: 先局部后全局——就近原则



# 五. 闭包(closure):

对应小程序视频: 小程序->在线->JSCORE->day03 8. 闭包 closure 压岁钱

1. 什么是: 2个概念

(1). 专门重用一个变量，又保护变量不被篡改一种编程方法

(2). 本质: 外层函数调用后，外层函数的作用域对象被内层函数引用着，无法释放，形成了闭包对象

2. 为什么: 全局变量和局部变量都有不可兼得的优缺点:

(1). 全局变量: 优点: 可重用, 缺点: 随处可用，极易被污染

(2). 局部变量: 优点: 只能在函数内使用，不会被污染, 缺点: 不可重用

3. 何时: 今后只要想重用一个变量，但是又要保护变量不被篡改，都要用闭包！

## 4. 如何: 3步 —— 判断闭包的三个特点

(1). 用外层函数包裹要保护的变量和使用该变量的函数

(2). 外层函数将内层函数return到外层函数外部，让外部可以使用

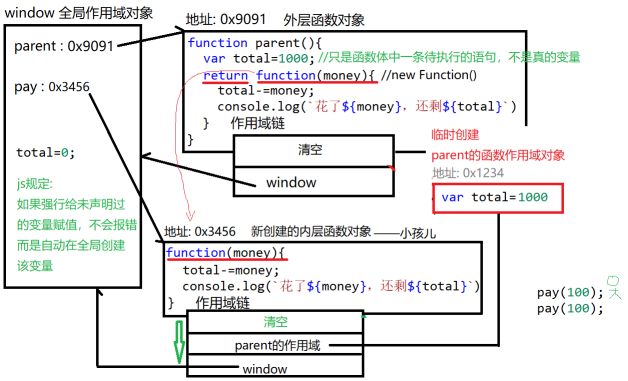
说明: 因为内层函数被返回到外部之后，一定会被外边的变量接住。所以，内层函数没必要起名！就算起了名字，将来外部也不一定使用这个名字！

(3). 外部要想使用内层函数，必须先调用外层函数，来获得返回的内层函数保存到变量中，才能反复使用！

5. 示例: 定义一个管理压岁钱的函数，要求压岁钱总数不能被外界随意篡改

|  |
| --- |
| <script>  //需求: 定义一个函数帮小孩儿管理压岁钱  //假设小孩儿有1000块钱压岁钱  //1. 用外层函数包裹要保护的变量和内层函数  function parent(){//父母帮小孩儿保管压岁钱  var total=1000;//——压岁钱  //2. 外层函数将内层函数return到外层函数外部，让外部可以使用  return function(money){//——小孩儿  //从总钱数中-本次花的钱  total-=money;  //输出花了多少，还剩多少余额  console.log(`花了${money}，还剩${total}`)  }  }  //3. 外部要想使用内层函数，必须先调用外层函数，来获得返回的内层函数保存到变量中，才能反复使用！  var pay=parent();  //pay:function(money){ ... }  //测试:  pay(100);  total=0;//尝试篡改total  pay(100);  pay(100);  </script>  运行结果:  花了100，还剩900  花了100，还剩800  花了100，还剩700 |

6. 一句话闭包形成的原因: 外层函数调用后，外层函数的作用域对象被内层函数引用着，无法释放，就形成了闭包！



7. 闭包缺点: 比一般的函数多占用了一块内存空间——外层函数的函数作用域对象

8. 解决: 一旦一个闭包不再使用，应该立刻释放:

保存内层函数的变量=null

导致内层函数对象被释放

导致外层函数的作用域对象(闭包)一同被释放

约定: 看视频时，必须跟着亲手画图(在操作系统的画图工具里)！至少画三遍以上！

**Day04**正课: OOP

1. 什么是面向对象:

2. 封装:

3. 继承:

4. 多态:

# 一. 什么是面向对象:

1. 什么是: 程序中都是先用对象结构来集中描述现实中一种事物的属性和功能。然后再按需访问对象中保存的属性和功能。

2. 为什么: 便于大量数据的管理和维护

3. 何时: 今后所有程序都要使用面向对象的思想开发

4. 如何: 三步/面向对象三特点: 封装，继承，多态

# 二. 封装:

对应小程序视频: 小程序->在线->JSCORE->day04 1. 用{}创建对象 this

1. 什么是: 将一个事物的属性和功能集中定义在一个对象内

2. 为什么: 便于大量数据的管理和维护

3. 何时: 今后只要使用面向对象的思想开发，第一步都是要先封装对象。然后才是按需使用不同对象中的属性和功能。

4. 如何创建一个对象:

(1). var 对象名={

属性名: 属性值,

... : ... ,

方法名: function(){

... ...

}

}

(2). 一个事物的属性值，会成为程序中对象里的一个属性。事物的一个功能(会做的一件事)，就会成为对象中的一个方法(function(){ ... })。——{}中的一切属性和方法统称为这个对象的成员.

5. 如何访问对象中的成员:

(1). ~~错误做法: 不能直接将对象中的属性当做普通全局变量访问:~~

~~console.log(属性名) //报错~~

(2). 正确做法: 必须先用对象名变量找到对象，再用.操作符，进入对象内访问对象的成员：

a. 访问对象中的属性: 对象名.属性名

b. 访问对象中的方法: 对象名.方法名()

## 6. this:

(1). 问题: 对象自己的方法中，想使用对象自己的属性，竟然报错，说属性名未定义！

(2). 原因: 因为对象不是作用域，所以对象中的方法的作用域链里不包含当前对象本身。所以，方法中不加任何.前缀的变量只能在方法内和window中查找。无法自动进入当前对象中查找。所以，即使对象自己的方法，想访问对象自己的属性，也不能直接使用！

(3). 解决:

a. 不好的解决方法: 方法中如果想用当前对象的属性，可加上"对象名.属性名"

b. 问题: 方法内写死外部对象的对象名——紧耦合。一旦将来外部的对象名发生变化，我们被迫要记得手动修改内部方法中写死的对象名——极其不便于维护！——应该极力避免！

c. 好的解决方法: 今后只要对象自己的方法想使用对象自己的属性，必须写成"this.属性名"。

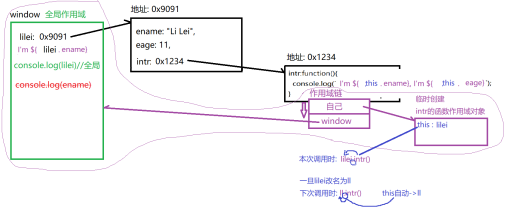
1). 什么是this: this是专门自动引用正在调用当前函数的.前的对象 的关键词

2). this关键词保存在函数临时创建的函数作用域中

3). 每次调用函数时，this自动先找"."，然后再自动指向.前的对象

比如: lilei.intr() this->.前的lilei， ll.intr() this->.前的ll

4). 好处: 方法内即使不写死对象名，也可以通过this自动获得.前当前对象现在的名字。执行时，和写死对象名效果完全一样！



(4). 总结: 今后只要对象自己的方法想使用对象自己的属性，必须加"this.属性名"。

## 7. 示例: 定义对象描述一个学生lilei的属性和会做的事儿:

|  |
| --- |
| <script>  //创建一个学生对象，来描述现实中一个学生的属性值和会做的事儿  //学生姓名: Li Lei  //学生年龄: 11  //学生会自我介绍: 李磊会说: I'm Li Lei, I'm 11  var lilei={ //对象的{}不是作用域  //属性名 : 属性值  ename: "Li Lei",  eage: 11,  //功能/方法: 因为功能是指会做一件事，所以功能被定义为一个函数  intr:function(){  console.log(`I'm ${this.ename}, I'm ${this.eage}`); //报错: ename未定义  }  }  //想输出李磊的年龄：  console.log(`李磊的年龄: ${lilei.eage}`);  //想请李磊做自我介绍:  lilei.intr();  //过了一年，李磊长了一岁  lilei.eage++;  //再输出李磊的年龄  console.log(`李磊的年龄: ${lilei.eage}`);  //再请李磊做自我介绍  lilei.intr();  </script>  运行效果：  李磊的年龄: 11  I'm Li Lei, I'm 11  李磊的年龄: 12  I'm Li Lei, I'm 12 |

## 8. 创建对象可以有: 3种方式:

### (1). 用{}创建一个对象:

var 对象名={

属性名: 属性值,

... : ... ,

方法名:function(){

... this.属性名

}

}

### (2). 用new Object()创建一个对象:

对应小程序视频: 小程序->在线->JSCORE->day04 2.3. 对象底层是关联数组...

a. 如何: 2步

1). 先创建一个空对象

var 对象名=new Object()

2). 给空对象中强行添加新属性和方法

对象名.属性名=属性值;

对象名.方法名=function(){ ... }

b. 揭示了一个本质:

1). {}其实就是new Object()的简写

2). js中对象底层也是关联数组

i. 都保存的是名值对儿的组合

ii. 都可用["下标名/属性名"]访问自己的成员

也都可以简写为"对象.属性名"方式

大部分情况不用在傻傻的区分该用.还是该用[""]，因为.就是[""]

特例: 如果属性名不是写死的，而是来自于一个变量，就不能用.，只能用[变量名]，且还不能加""

iii. 都可以随时往不存在的位置添加新成员：通过强行赋值的方式！

所以将来要想给对象或关联数组添加新成员，唯一的方式只有：强行赋值

iv. 强行访问对象或数组中一个不存在的位置，不会报错！会返回undefined

所以将来要判断一个对象或关联数组中是否包含某个成员：

对象.成员名!==undefined 返回true，说明包含该成员

返回false，说明不包含该成员！

v. 遍历对象或关联数组中的成员，都可用: for in循环

回顾: for(var 变量名 in 对象或关联数组){

in会自动依次获得对象或关联数组中每个属性的属性名，保存到in前的变量中

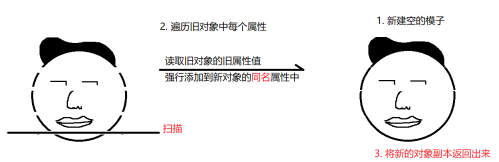
然后，我们就可以利用"对象[变量名]"读取出对象或关联数组中每个属性的属性值。

}

c. 示例: 比较对象和关联数组

|  |
| --- |
| <script>  var lilei={ //一个对象lilei  ename:"Li Lei",  eage:11  };  var ym=[]; //一个关联数组ym  ym["ename"]="杨幂";  ym["eage"]=30;  //输出对象和关联数组在内存中的存储结构，比较相似点  console.log(lilei);  console.log(ym);  //想访问lilei的ename属性  console.log(lilei["ename"], lilei.ename)  //想访问ym的ename属性  console.log(ym["ename"], ym.ename)  //想给李磊和ym都添加一个新属性:班级className  lilei["className"]="初一2班";  ym["className"]="表演系2班";  //再次输出两个对象，观察新属性是否强行添加成功  console.log(lilei);  console.log(ym);  //想获取李磊和ym的className属性  console.log(lilei.className, ym.className);  //想获得李磊和ym的不存在的salary属性  console.log(lilei.salary, ym.salary);  //分别遍历lilei和ym中都有哪些属性，属性值分别是什么:  for(var a in lilei){  console.log(`属性名:${a}`)  console.log(`属性值: ${lilei[a]}`)  //console.log(`属性值:${lilei.a}`) //错误  //a是一个循环变量，每一轮的值都不一样，都在不停变化  //lilei.a会被翻译为lilei["a"]，如果变量a被包在""中，就不再是变量了。而是一个普通的写死的属性名"a"了！也就无法变化了。  //如果lilei[a]，每一轮循环中，lilei[a]，分别会变成lilei["ename"]、lilei["eage"]、lilei["className"]，因为不加""的a是变量，可以不断变化。  }  for(var b in ym){  console.log(`属性名:${b}`);  console.log(`属性值: ${ym[b]}`)  //console.log(`属性值:${ym.b}`) //错误  //b也是一个循环变量，每一轮的值都不一样，都在不停变化  //ym.b会被翻译为ym["b"]，如果变量b被""包裹，就不再是变量了。而是一个普通的写死的属性名"b"，也就无法发生变化了  //如果ym[b]，每一轮循环， ym[b]，分别会变为ym["ename"]、ym["eage"]、ym["className"]。因为不加""的b是变量，可以不断变化。  }  </script>  运行效果：  {ename: "Li Lei", eage: 11}  [ename: "杨幂", eage: 30]  Li Lei Li Lei  杨幂 杨幂  {ename: "Li Lei", eage: 11, className: "初一2班"}  [ename: "杨幂", eage: 30, className: "表演系2班"]  初一2班 表演系2班  undefined undefined  属性名:ename  属性值: Li Lei  属性名:eage  属性值: 11  属性名:className  属性值: 初一2班  属性名:ename  属性值: 杨幂  属性名:eage  属性值: 30  属性名:className  属性值: 表演系2班 |

d. 示例: 克隆一个对象:



|  |
| --- |
| <script>  var lilei={  sname:"Li Lei",  sage:11  };  //定义一个函数clone()  //接收一个原对象  //经过加工和处理  //返回一个一模一样的新对象  function clone(oldObj){  //1. 先创建一个空对象等着！  var newObj={};  //2. 遍历旧对象中每个属性  for(var key in oldObj){  //每遍历旧对象中一个属性，都要为新对象强行一个同性的新属性，属性值等于旧对象中当前同名属性值  newObj[key]=oldObj[key];  }  //3. 返回克隆好的新对象  return newObj;  }  //测试:  var lilei2=clone(lilei);  console.log(lilei);//长的一模一样  console.log(lilei2);//长的一模一样  console.log(lilei==lilei2); //false 说明两个对象地址不同！是两个独立的存储空间  var hmm={  sname:"Han Meimei",  sage:12  }  var hmm2=clone(hmm);  console.log(hmm);//长的一模一样  console.log(hmm2);//长的一模一样  console.log(hmm==hmm2); //false  </script>  输出结果:  {sname: "Li Lei", sage: 11}  {sname: "Li Lei", sage: 11}  false  {sname: "Han Meimei", sage: 12}  {sname: "Han Meimei", sage: 12}  false |

e. 问题: 前两种创建对象的方式，一次只能创建一个对象。如果反复创建多个相同结构的对象时，代码很繁琐的！

### (3). 使用构造函数反复创建多个相同结构的对象:

对应小程序视频: 小程序->在线->JSCORE->day04 5. 构造函数 constructor new

a. 什么是构造函数: 描述同一类型所有对象的统一结构的函数

b. 为什么: 为了重用对象结构

c. 何时: 只要反复创建多个相同结构的对象时，都要先定义构造函数，来描述一类对象的统一结构。

d. 如何: 2步:

1). 定义构造函数:

function 类型名(形参变量列表){

this.属性名=形参变量1;

this.属性名=形参变量2;

~~this.方法名=function(){ ... }~~

}

2). 用new调用构造函数反复创建同一类型的多个相同结构的对象

var 对象名=new 类型名(实参值列表)

e. new做了4件事（构造函数的原理）

1). new先创建一个新的空对象

2). 设置新对象"\_ \_proto\_ \_"属性自动继承构造函数的原型对象

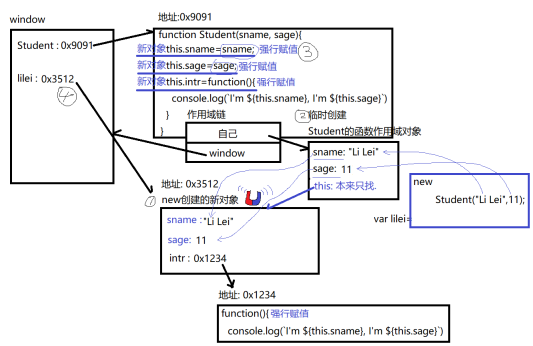
3). 用new调用构造函数:

i. 将实参值传给构造函数的形参变量

ii. 自动将构造函数中所有this都吸引到new创建的新对象来

结果，每一句【this.属性名=形参】都强行给对象赋值新属性名（添加一个新属性）

4). new会返回新创建的对象地址保存给=左边的变量



f. 示例: 创建构造函数Student()描述所有学生对象统一结构，并创建两个具体的学生对象lilei和hmm

|  |
| --- |
| <script>  //想创建两个学生:  //lilei: sname: Lilei, sage: 11, 会自我介绍: I'm Li Lei, I'm 11  //hmm: sname: Han Meimei, sage: 12, 会自我介绍: I'm Han Meimei, I'm 12  //总结: 所有学生都有sname和sage两个属性，所有学生都有intr()方法做自我介绍  //所以:  function Student(sname, sage){  this.sname=sname;  this.sage=sage;  this.intr=function(){  console.log(`I'm ${this.sname}, I'm ${this.sage}`)  }  }  //用new调用Student()构造函数，创建lilei对象，并将李磊的属性值保存进将要创建的lilei对象中  var lilei=new Student("Li Lei",11);  //用new调用Student()构造函数，创建hmm对象，并将韩梅梅的属性值保存进将要创建的hmm对象中  var hmm=new Student("Han meimei",12);  console.log(lilei);  console.log(hmm);  </script>  运行结果:  Student {sname: "Li Lei", sage: 11, intr: ƒ}  Student {sname: "Han meimei", sage: 12, intr: ƒ} |

g. 为什么构造函数中所有规定的属性和方法前都要有this.?

为了让this和将来的new配合，将构造函数中所有规定的属性和方法通过强行赋值的方式，添加到将来new创建的新对象中。

h. 构造函数的优点: 重用代码！

i. 构造函数有问题: 浪费内存——如果把方法定义包含在构造函数中。那么每调用一次构造函数，都会重复创建一次方法函数对象的副本。

# 三. 继承:

对应小程序视频: 小程序->在线->JSCORE->day04 6. 继承 原型对象 prototype ...

1. 什么是: 父对象的成员子对象无需重复创建就可直接使用！

2. 为什么: 既可重用代码，又可节约内存！

3. 何时: 今后只要同一类型多个子对象都需要同一个方法时，这个方法不要定义在构造函数中，应该定义在所有孩子的父对象中一份即可！所有孩子，就可通过继承关系，直接使用！

4. 如何:

(1). 从此构造函数中不能包含任何方法的定义，只包含属性的定义。

(2). 所有子对象都需要的方法，应该集中保存在这个类型的原型对象中

## 5. 原型对象:

(1). 什么是: 专门为同一类型下所有子对象集中保存共有方法的父对象

(2). 如何创建: 原型对象不是我们自己手工创建的，而是买一增一：

当我们定义构造函数时，其实悄悄的赠送了我们一个空的原型对象

(3). 何时/如何建立继承关系: new的第2步:

a. new为每个子对象自动添加了一个"\_ \_proto\_ \_"，指向当前子对象的父对象

b. new将子对象的"\_ \_proto\_ \_"，自动指向构造函数的原型对象

(4). 结果: 从此，凡是被new出的子对象，都默认继承自构造函数的原型对象。凡是保存在构造函数的原型对象中的共有方法，所有子对象可无条件无需创建就可直接使用。

(5). 当子对象调用一个自己没有，但是却保存在原型对象中的共有方法时，程序首先在子对象自己身上找。如果在自己身上找不到，就会自动延"\_ \_proto\_ \_"关系去父对象查找！

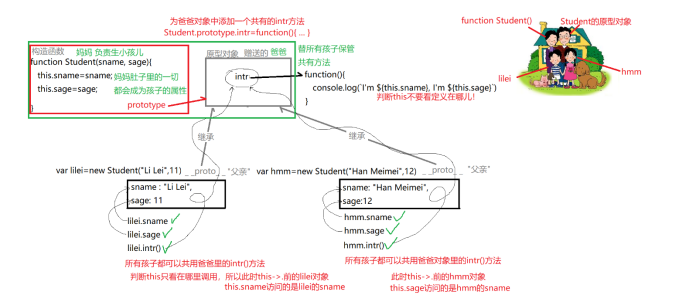
(6). 总结: 今后所有子对象都需要的共有方法应该集中添加到构造函数的原型对象中保存。

(7). 如何将共有方法添加到构造函数的原型对象中: 唯一正确且稳妥的办法:

构造函数都有一个prototype属性，默认指向赠送的原型对象

所以: 构造函数.prototype.共有方法=function(){ ... }

原型对象 强行赋值



6. 示例: 为Student的原型对象添加共有方法，并让子对象调用共有方法

|  |
| --- |
| <script>  //先定义构造函数妈妈，但是不要包含方法定义！  function Student(sname, sage){  this.sname=sname;  this.sage=sage;  }  //输出附赠的空的原型对象  console.log(Student.prototype);//看到一个空对象  //向原型对象中添加intr方法: 强行赋值  Student.prototype.intr=function(){  console.log(`I'm ${this.sname}, I'm ${this.sage}`)  }  //妈妈生了两个孩子  var lilei=new Student("Li Lei",11);  var hmm=new Student("Han meimei",12);  console.log(lilei);  console.log(hmm);  // 李磊 的 爹 是不是 Student的原型对象  console.log(lilei.\_\_proto\_\_==Student.prototype); //true  // 韩梅梅 的 爹 是不是 Student的原型对象  console.log(hmm.\_\_proto\_\_==Student.prototype); //true  lilei.intr();  hmm.intr();  </script>  运行结果：  {constructor: ƒ}  Student {  sname: "Li Lei",  sage: 11,  \_\_proto\_\_:  intr: ƒ ()  }  Student {  sname: "Han meimei",  sage: 12,  \_\_proto\_\_:  intr: ƒ ()  }  true  true  I'm Li Lei, I'm 11  I'm Han meimei, I'm 12 |

**Day05**正课:

\*\*\*\*面向对象\*\*\*\*

1. 继承

2. 多态

3. 自定义继承

\*\*\*\*ES5\*\*\*\*

1. 严格模式

\*\*\*\*面向对象\*\*\*\*

# 一. 继承

对应小程序视频: 小程序->在线->JSCORE->day04 7.8.9

## 1. 自有属性和共有属性

(1). 什么是:

a. 自有属性: 保存在当前子对象内，只归当前子对象自己独有的属性

b. 共有属性: 保存在原型对象中，归该类型下所有子对象共有的属性

(2). 何时：

a. 如果希望每个子对象都不相同的属性值，应该定义为自有属性

b. 如果希望多个子对象共用相同的属性值时，应该定义为共有属性

(3). 如何:

a. 所有构造函数中的属性或通过.强行添加给对象的属性都是自有属性

b. 构造函数.prototype.共有属性=属性值 —— 共有属性

(4). 坑:

a. 想获取属性值则自有属性和共有属性都可用“对象.属性名”

b. 修改属性值:

1). 修改自有属性，必须用子对象.属性名=新值

2). 修改共有属性，必须用原型对象.属性名=新值

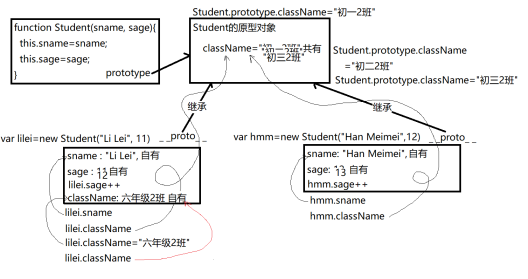
i. 不能用子对象修改共有属性 —— 不要尝试这样做！no zuo no die!

ii. 如果用子对象强行修改共有属性: 不会报错

①js绝不允许子对象直接修改共有属性，所以共有属性保持不变

②js会自动给当前子对象添加一个和要修改的共有属性同名的自有属性

结果: 当前子对象在这个共有属性的使用上和其它子对象分道扬镳，再无统一之日。



(5). 示例: 测试获取和修改自有属性和共有属性

|  |
| --- |
| <script>  //定义一个构造函数描述所有学生的统一属性结构  function Student(sname,sage){  this.sname=sname;  this.sage=sage;  }  //希望将来的所有学生都是同一个班级的学生  Student.prototype.className="初一2班"  var lilei=new Student("Li Lei",11);  var hmm=new Student("Han Meimei",12);  console.log(lilei);  console.log(hmm);  //想获得lilei和hmm的sage //自有  console.log(lilei.sage, hmm.sage);  //想获取lilei和hmm的className //共有  console.log(lilei.className, hmm.className);  //修改lilei和hmm的年龄+1 //自有  lilei.sage++;  hmm.sage++;  //再次获得lilei和hmm的sage //自有  console.log(lilei.sage, hmm.sage);  //lilei和hmm都升了一级  //正确:  Student.prototype.className="初二2班";  //再次获取lilei和hmm的className //共有  console.log(lilei.className, hmm.className);  //错误: 李磊想拖着全班一起留级！  lilei.className="六年级2班";  //再次获取lilei和hmm的className //共有  console.log(lilei.className, hmm.className);  // 六年级2班 初二2班  console.log(lilei);  console.log(hmm);  //又过了一年，大家一起升到初三2班  Student.prototype.className="初三2班";  //再次获取lilei和hmm的className //共有  console.log(lilei.className, hmm.className);  // 六年级2班 初三2班  </script>  运行效果: |

## 2. 内置类型的原型对象

(1). 什么是内置类型: ES标准中规定的浏览器自带的类型

(2). 包括: 11种:

String Number Boolean

Array Date Math RegExp

Error

Function Object

global(在浏览器中被window代替)

(3). 什么是类型: 由一个构造函数和一个原型对象组成的一个整体，称为类型，用于描述一类对象的统一结构和方法定义。

(4). 每种类型: 都包含两部分:

a. 构造函数:

1). 描述该类型下将来所有子对象的统一结构

2). 反复创建该类型的子对象

b. 原型对象: 替该类型下所有子对象保存共有的方法

(5). 比如: Array就是一个中类型，包括2部分:

a. 构造函数: function Array(){ ... ... }

用于反复创建数组类型的子对象: var arr=new Array(1,2,3) 可简写为 var arr=[1,2,3]

b. 原型对象: Array.prototype保存了所有数组家孩子共用的方法:

.sort() .push() .reverse() .slice() .indexOf() ... ...

将来随着ES标准不断更新，Array.prototype一定会继续加入很多更好用的共有方法。

(6). 问题: 如果我们经常对数组做一种操作，但是数组的原型对象中没有提供现成的共用方法，怎么办？——可以自定义一个共有方法，强行添加到数组原型对象中。

a. 强行添加自定义共有方法:

Array.prototype.自定义共有方法=function(形参列表){

... 想获得将来调用这个方法的.前的某个具体的数组，应该用: this

}

b. 调用自定义共有方法: 和调用内置共有方法完全一样！

数组.自定义共有方法(实参值列表)

(7). 示例: 为数组家添加数组元素求和的sum函数，并调用

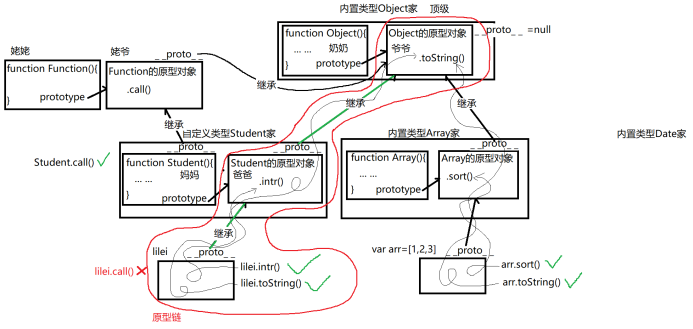
|  |
| --- |
| <script>  //如果我的项目中经常需要对数组中的元素值求和  //但是假设数组的原型对象中并没有提供求和的函数sum  //我们就可以想数组原型对象中强行添加一个自定义的sum函数  Array.prototype.sum=function(){  console.log(`调用了一次Array.prototype中我们自定义的sum()函数`)  var total=0;  //遍历将来要求和的数组中每个元素  //this->将来调用.sum()函数的.前的某一个数组家孩子对象  //比如: arr1.sum() this->arr1  //比如: arr2.sum() this->arr2  for(var i=0;i<this.length;i++){  total+=this[i];  }  return total;  }  console.log(Array.prototype);  //测试:  //两个数组家孩子  var arr1=[1,2,3]  var arr2=[1,2,3,4,5];  //可共用数组原型对象中我们自定义的sum()函数，谁调用，this就指谁  console.log(arr1.sum());//6  console.log(arr2.sum());//15  </script>  运行效果:  [sum: ƒ, constructor: ƒ, concat: ƒ, copyWithin: ƒ, fill: ƒ, …]  调用了一次Array.prototype中我们自定义的sum()函数  6  调用了一次Array.prototype中我们自定义的sum()函数  15 |

## 3. 原型链:

(1). 什么是: 由多级父对象逐级继承形成的链式结构

(2). 保存着: 子对象可以使用的所有共有的属性和方法

(3). 控制着: 属性和方法的使用顺序: 先自有，再共有 —— 就近



# 二. 多态：

对应小程序视频: 小程序->在线->JSCORE->day05 1. 多态 override toString()

1. 什么是: 一个函数在不同的情况下表现出不同的状态

## 2. 包括: 2种:

(1). 重载(overload): 一个函数根据传入的实参值不同，动态选择执行不同的逻辑

(2). 重写(override): ?

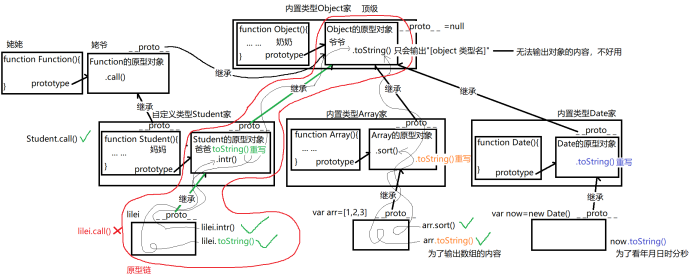
推翻

## 3. 重写:

(1). 什么是: 在子对象中定义一个和父对象中的成员同名的自有成员

(2). 为什么: 从父对象继承来的东西不一定总是好用的！

(3). 何时: 只要子对象觉得从父对象继承来的东西不好用！都可以在子对象内定义同名成员，重写父对象中的同名成员。

(4). 如何: 在子对象内定义和父对象中的成员同名的自有成员  


(5). 示例: 为自定义类型或自定义对象重写好用的toString()方法

|  |
| --- |
| <script>  // var lilei={//new Object()  // sname: "Li Lei",  // sage:11,  // //只给李磊重写toString()方法  // toString:function(){  // return `{ sname: ${this.sname}, sage: ${this.sage} }`  // }  // }  function Student(sname, sage){  this.sname=sname;  this.sage=sage;  }  //如果希望将来所有学生都有好用的toString(),则toString应该放在Student类型的原型对象中  Student.prototype.toString=function(){  return `{ sname: ${this.sname}, sage: ${this.sage} }`;  //格式可以自己随便改！  }  var lilei=new Student("Li Lei", 11);  var hmm=new Student("Han Meimei",12);  console.log(lilei.toString());  console.log(hmm.toString());  var arr=[1,2,3]; //new Array()  var now=new Date(); //new Date()  //对三个对象分别调用toString()——本意为了输出对象的内容  console.log(lilei.toString());  console.log(arr.toString());  console.log(now.toString())  </script>  运行效果：  { sname: Li Lei, sage: 11 }  { sname: Han Meimei, sage: 12 }  { sname: Li Lei, sage: 11 }  1,2,3  Tue Jun 02 2020 14:55:20 GMT+0800 (中国标准时间) |

# 三. 自定义继承:

对应小程序视频: 小程序->在线->JSCORE->day05 2. 自定义继承 Object.setPrototypeOf()

1. 何时: 如果整个父对象都不想要！都想换掉！就可以自定义继承

## 2. 如何: 2种:

## (1). 只更换一个子对象的父对象:

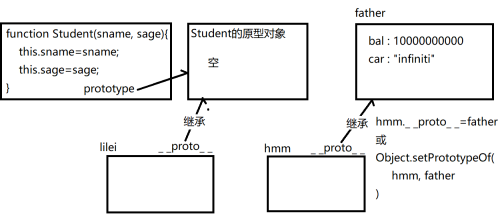
a. 子对象.\_ \_proto\_ \_=新父对象

b. 问题: 有些浏览器可能不允许直接使用\_ \_proto\_ \_

c. Object.setPrototypeOf(子对象, 新父对象)

设置原型对象的子对象 为新父对象

d. 示例: 只更换一个对象的爹:



|  |
| --- |
| <script>  function Student(sname, sage){  this.sname=sname;  this.sage=sage;  }//prototype=>{ } //旧爹 空的  var lilei=new Student("Li Lei",18);  var hmm=new Student("Han Meimei",19);  //输出李雷和韩梅梅的爹  console.log(lilei.\_\_proto\_\_);  console.log(hmm.\_\_proto\_\_);  //新爹 诱惑  var father={  bal:100000000000000,  car:"infiniti"  }  //hmm想认father当爹  //hmm.\_\_proto\_\_=father;//有时有的浏览器不允许直接使用\_\_proto\_\_  Object.setPrototypeOf(hmm, father);  //输出李雷和韩梅梅的爹  console.log(lilei.\_\_proto\_\_);  console.log(hmm.\_\_proto\_\_);  //输出李雷和韩梅梅的银行存款和car  console.log(hmm.bal, hmm.car);  console.log(lilei.bal, lilei.car);  </script>  运行结果:  {constructor: ƒ}  {constructor: ƒ}  {constructor: ƒ}  {bal: 100000000000000, car: "infiniti"}  100000000000000 "infiniti"  undefined undefined |

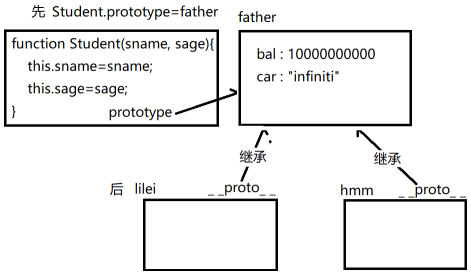
## (2).更换多个子对象的爹:

a. 修改构造函数的原型对象: 构造函数.prototype=新对象

b. 问题: 如果在已经创建子对象之后，再换构造函数的原型对象，已经创建的子对象的原型对象是不会自动跟着改变的——无法达到更换所有子对象的原型对象的效果

c. 时机: 如果想修改多个子对象的原型对象，必须在创建子对象之前就要更换！

d. 示例: 同时更换lilei和hmm的爹



|  |
| --- |
| <script>  function Student(sname, sage){  this.sname=sname;  this.sage=sage;  }//prototype=>{ } //旧爹 空的  //输出构造函数旧的原型对象  console.log(Student.prototype);  //新爹 诱惑  var father={  bal:100000000000000,  car:"infiniti"  }  //修改构造函数的原型对象为father  Student.prototype=father;  //输出构造函数新的原型对象  console.log(Student.prototype);  //创建子对象  var lilei=new Student("Li Lei",18);  var hmm=new Student("Han Meimei",19);  //输出李雷和韩梅梅的爹  console.log(lilei.\_\_proto\_\_);  console.log(hmm.\_\_proto\_\_);  //输出李雷和韩梅梅的银行存款和car  console.log(hmm.bal, hmm.car);  console.log(lilei.bal, lilei.car);  </script>  运行结果:  {constructor: ƒ}  {bal: 100000000000000, car: "infiniti"}  {bal: 100000000000000, car: "infiniti"}  {bal: 100000000000000, car: "infiniti"}  100000000000000 "infiniti"  100000000000000 "infiniti" |

\*\*\*\*ES5\*\*\*\*

ECMAScript 第五版 对旧js打补丁 —— 各个击破！没有联系

# 一. 严格模式:

对应小程序视频: 小程序->在线->JSCORE->day05 3. ES5 严格模式 use strict...

1. 什么是: 比旧的js运行机制要求更严格的模式

2. 为什么: 因为旧的js有很多广受诟病的缺陷

3. 何时: 今后所有js程序都要运行在严格模式下！

4. 如何启用严格模式: 在当前<script>标签内顶部或js文件的顶部添加: "use strict"

启用 严格模式

5. 新规定:

## (1). 禁止给未声明的变量赋值:

a. 旧js中：强行给未声明过的变量赋值，不报错，自动在全局创建该变量

b. 缺点: 极易造成全局污染！

c. 严格模式：强行给未声明过的变量赋值，报错！xxx变量未定义！

d. 好处: 极大的减少了全局污染的可能！也避免了程序调试中的歧义。

e. 示例: 使用严格模式禁止给未声明过的变量赋值

|  |
| --- |
| <script>  "use strict";//启用严格模式  //想定义一个函数可以给女朋友发送一条消息  function send(){  var gf; //女朋友  //我想告诉她: 今晚308,w84u  //但是不小心变量名拼错了  //js没有变量检查机制，即使写错变量名，变量不存在，也会在全局创建该变量  pyq="今晚308,w84u";//严格模式: 报错！阻止了一次误操作的发生  console.log(`我女朋友收到:${gf}`);  }  send();  //在全局输出变量pyq的内容  console.log(`我的朋友圈:${pyq}`);  </script>  运行结果:  Uncaught ReferenceError: pyq is not defined  at send (6\_use\_strict.html:18)  at 6\_use\_strict.html:21 |

## (2). 静默失败升级为错误：

a. 什么是静默失败: 执行不成功也不报错

b. 缺点: 极其不便于程序的调试

c. 严格模式: 所有静默失败都被升级为错误！

d. 优点: 极其便于程序的调试，避免歧义

e. 示例: 使用严格模式让静默失败升级为错误

|  |
| --- |
| <script>  "use strict";  //一会儿要讲的新知识，先不要问为什么  //一个员工eric  var eric={  eid:1001,  ename:"埃里克"  }  //公司规定员工编号不能随意修改——只读  // 定义 属性 eric的eid属性为不可修改  Object.defineProperty(eric,"eid",{writable:false});  //试图修改eric的eid属性值  eric.eid=1002;//严格模式: 报错:  //Cannot assign to read only property 'eid' of object  //不能 赋值 给 只读 属性 eid  //不能给只读属性eid赋值——23行  console.log(eric);//没改成  </script>  运行结果:  Uncaught TypeError: Cannot assign to read only property 'eid' of object '#<Object>'  at 7\_use\_strict.html:23 |

## (3). 普通函数和匿名函数自调中的this不再指向window，而是undefined

a. 旧js中: fun() 或 (function(){ ... })() this->window

b. 缺点: 极易造成全局污染

c. 严格模式: fun() 或 (function(){ ... })() this=undefined

d. 优点: 杜绝了因为错误的使用this而造成的全局污染！

e. 示例: 使用严格模式杜绝因为错误使用构造函数而导致的全局污染

|  |
| --- |
| <script>  "use strict";  function Student(sname,sage){  this.sname=sname;  this.sage=sage;  //没有return，所以没有返回值，相当于undefined  }  //本来构造函数应该用new调用  var lilei=new Student("Li Lei",11);  console.log(lilei);  //但是不小心忘记写new了  var hmm=Student("Han Meimei",12);  //undefined.sname=sname; //报错:  //Cannot set property 'sname' of undefined  //不能 设置 属性 sname 的 undefined  //不能设置undefined的sname属性  //undefined.sage=sage; //报错  console.log(hmm);//undefined  //console.log(sname, sage);//在全局直接输出本该保存到对象内的属性，却可以直接使用！——全局污染！  </script>  运行结果:  Student {sname: "Li Lei", sage: 11}  Uncaught TypeError: Cannot set property 'sname' of undefined  at Student (8\_use\_strict.html:13)  at 8\_use\_strict.html:21 |

## (4). 不推荐使用arguments.callee了

a. 什么是arguments.callee: 在函数内部自动获得当前函数本身 的关键词

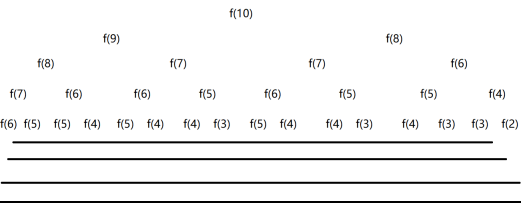
b. 何时: 专门用于递归调用时

c. 问题: 如果在递归中写死函数名，则一旦外部函数名发生变化，程序极容易出错——紧耦合

d. 解决: 在递归调用的函数内部用arguments.callee代替写死的函数名

e. 优点: arguments.callee在运行时，可自动获得当前正在执行的函数对象本身——即使外部函数改名了，也能照样获得当前函数本身——松耦合！

f. 问题: 递归算法本身效率极低——重复计算量太大！



g. 严格模式: 禁用arguments.callee就暗示着，不推荐使用递归算法

h. 解决: 绝大多数递归算法都可用循环方式代替——难度极高！

（扩展）小程序->在线->JSCORE->day05 3.ES5严格模式 ... 视频中有循环实现的菲波那契数列算法的详细讲解。——忘了菲波那契数列的同学，晚上可以看我讲的视频。

i. 总结: 不要走极端！

1). 如果采用递归算法并没有严重影响效率，依然可以采用递归算法，只不过只能在递归函数中写死函数名——退而求其次

2). 除非采用的递归算法严重影响了效率，才有必要费劲将递归转为循环。

j. 示例: 使用arguments.callee和递归实现菲波那契数列

|  |
| --- |
| <script>  "use strict";  //小程序->在线->JSCORE->day05 3.ES5严格模式 ...  //复习第一阶段讲过的斐波那契数列  //1 1 2 3 5 8 13 21 34 55 ...  //1 2 3 4 5 6 7 8 9 10  //总结:  //第一项是1,  //第二项还是1,  //从第三项开始，当前项的值等于相邻的前两项值的和  //定义一个函数求出斐波那契数列中第n项时的值是多少  function f(n){  if(n<3){  return 1;  }else{  // 递归 递归  return arguments.callee(n-1)+arguments.callee(n-2);//报错  //自动获得当前函数名 f f  }  }    console.log(f(10));//55  </script>  运行结果：  Uncaught TypeError: 'caller', 'callee', and 'arguments' properties may not be accessed on strict mode functions or the arguments objects for calls to them  at f (9\_argument.callee.html:26)  at 9\_argument.callee.html:31 |

**Day06**正课: \*\*\*\*ES5\*\*\*\*

1. 保护对象

2. Object.create()

3. 替换this

4. 数组函数

# 一. 保护对象:

1. 问题: js中的对象自己毫无自保能力。别人可以随意修改对象的属性值和对象结构。对象对此毫无抵抗力。但是企业中一个对象的数据和结构往往不能随意修改为无意义的值。

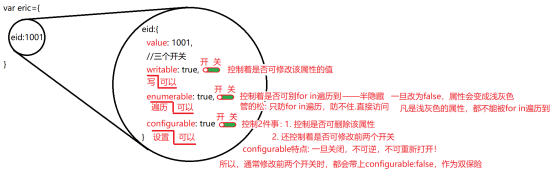
2. 解决: 在ES5标准中，其实已经给每个对象内置了自我保护的办法！

3. 如何保护对象：保护对象中的单个属性 + 保护整个对象的结构

## 4. 保护属性:

对应小程序视频: 小程序->在线->JSCORE->day05 4. 保护对象属性 开关...

(1). ES5标准中规定，对象中的一个属性，已经不再是一个普通的变量了。而是一个缩微的小对象



(2). 强调: 虽然对象的每个属性在底层也是一个缩微的小对象。但是小对象中的开关~~不能用.直接访问~~到！必须用专门的函数！

(3). 查看一个对象的属性的小对象中的开关值: （用的少，了解即可）

var obj=Object.getOwnPropertyDescriptor(对象, "属性名");

获得自己的属性的描述信息

(4). 修改一个对象中一个属性中开关的状态: ~~也不能用.直接修改~~

a. 如果只修改一个属性的开关:

(重新)定义属性

Object.defineProperty(对象名, "属性名", {

开关名: true或false,

开关名: true 或false

... ...

})

b. 示例: 使用defineProperty()修改属性的开关，保护属性

|  |
| --- |
| <script>  "use strict";  //一个员工对象eric  var eric={  eid:1001, //员工编号  /\*eid:{  value:1001,  writable:true, //控制着是否可修改属性值  enumerable:true, //控制着是否可被for in遍历到，但是防不住.直接访问  configurable:true //控制着: 1. 是否可删除该属性  // 2. 是否可修改前两个开关  }\*/  ename:"埃里克", //员工姓名  salary:12000 //薪资  }  //错误: 尝试用.访问每个属性小对象内的开关值  // console.log(  // eric.eid.value,  // eric.eid.writable,  // eric.eid.enumerable,  // eric.eid.configurable  // ); //undefined undefined undefined undefined  //正确：用专门的函数查看一个对象中一个属性的开关值  var obj=Object.getOwnPropertyDescriptor(eric,"eid");  console.log(obj);  //企业中有三个规定:  //1. 员工编号禁止随意修改——只读 eid的writable改为false  //2. 员工不能没有姓名属性——不能删除 ename的configurable改为false  //3. 员工的薪资不能随意被for in遍历 salary的enumerable改为false  //让eric的eid属性只读  //重新定义eric的eid属性的writable为false  Object.defineProperty(eric,"eid",{  writable:false,  configurable:false //双保险: 禁止修改前两个开关 —— 一旦关闭不可逆！  })  //让eric的ename禁止删除  //重新定义eric的ename属性的configurable为false  Object.defineProperty(eric,"ename",{  configurable:false  })  //不想让eric的salary随意被for in遍历  //重新定义eric的salary属性的enumerable:false  Object.defineProperty(eric,"salary",{  enumerable:false,  configurable:false //双保险  })  //屌丝: 尝试将已经关闭的eid的writable重新打开，再修改  // Object.defineProperty(eric,"eid",{  // writable:true,  // configurable:true  // })//Cannot redefine property: eid  // // 不能 重新定义 属性 eid  //试图修改eid  // eric.eid=-2; //报错:  // //Cannot assign to read only property 'eid'  // //不能 赋值 给 只读 属性 eid  //试图删除ename  //删除 对象的属性——危险  //delete eric.ename; //报错: Cannot delete property 'ename'  //试图遍历eric所有属性，包括工资！  for(var key in eric){  console.log(`属性名:${key}, 属性值: ${eric[key]}`)  }  console.log(eric);  //尝试用.直接访问eric的salary属性  console.log(`薪资:${eric.salary}`) //依然可以正常输出薪资  </script>  运行效果:  {value: 1001, writable: true, enumerable: true, configurable: true}  属性名:eid, 属性值: 1001  属性名:ename, 属性值: 埃里克  {eid: 1001, ename: "埃里克", salary: 12000}  薪资:12000 |

c. 问题: 因为一个对象中肯定有多个属性都需要保护，但是defineProperty一次只能修改一个属性内的开关。如果想修改多个属性中的开关，就要反复写很多遍——麻烦！

d. 解决: 用defineProperties()一次修改多个属性内的开关

Object.defineProperties(对象名,{

属性名:{

开关: true或false,

... : ... ,

},

属性名:{

开关: true或false,

... : ... ,

},

... ...

})

e. 示例: 使用defineProperties()简化defineProperty()

|  |
| --- |
| <script>  "use strict";  //一个员工对象eric  var eric={  eid:1001, //员工编号  /\*eid:{  value:1001,  writable:true, //控制着是否可修改属性值  enumerable:true, //控制着是否可被for in遍历到，但是防不住.直接访问  configurable:true //控制着: 1. 是否可删除该属性  // 2. 是否可修改前两个开关  }\*/  ename:"埃里克", //员工姓名  salary:12000 //薪资  }  //错误: 尝试用.访问每个属性小对象内的开关值  // console.log(  // eric.eid.value,  // eric.eid.writable,  // eric.eid.enumerable,  // eric.eid.configurable  // ); //undefined undefined undefined undefined  //正确：用专门的函数查看一个对象中一个属性的开关值  var obj=Object.getOwnPropertyDescriptor(eric,"eid");  console.log(obj);  //企业中有三个规定:  //1. 员工编号禁止随意修改——只读 eid的writable改为false  //2. 员工不能没有姓名属性——不能删除 ename的configurable改为false  //3. 员工的薪资不能随意被for in遍历 salary的enumerable改为false  //让eric的eid属性只读  //重新定义eric的eid属性的writable为false  //让eric的ename禁止删除  //重新定义eric的ename属性的configurable为false  //不想让eric的salary随意被for in遍历  //重新定义eric的salary属性的enumerable:false  Object.defineProperties(eric,{  eid:{  writable:false,  configurable:false  },  ename:{  configurable:false  },  salary:{  enumerable:false,  configurable:false  }  })  //屌丝: 尝试将已经关闭的eid的writable重新打开，再修改  // Object.defineProperty(eric,"eid",{  // writable:true,  // configurable:true  // })//Cannot redefine property: eid  // // 不能 重新定义 属性 eid  //试图修改eid  // eric.eid=-2; //报错:  // //Cannot assign to read only property 'eid'  // //不能 赋值 给 只读 属性 eid  //试图删除ename  //删除 对象的属性——危险  //delete eric.ename; //报错: Cannot delete property 'ename'  //试图遍历eric所有属性，包括工资！  for(var key in eric){  console.log(`属性名:${key}, 属性值: ${eric[key]}`)  }  console.log(eric);  //尝试用.直接访问eric的salary属性  console.log(`薪资:${eric.salary}`) //依然可以正常输出薪资  </script>  运行结果:  {value: 1001, writable: true, enumerable: true, configurable: true}  属性名:eid, 属性值: 1001  属性名:ename, 属性值: 埃里克  {eid: 1001, ename: "埃里克", salary: 12000}  薪资:12000 |

(5). 访问器属性:

对应小程序视频: 小程序->在线->JSCORE->day05 5. 访问器属性 get set ...

a. 问题: 仅使用开关无法用自定义规则灵活保护属性

b. 解决: 给对象中的一个属性请保镖——访问器属性

c. 什么是访问器属性: 属性自己不实际保存属性值，仅提供对另一个属性的保护功能，这样的特殊属性，称为访问器属性，俗称保镖

d. 何时: 今后只要想用自定义规则灵活保护属性时，都要用访问器属性

e. 如何: 3步: ~~访问器属性不能在对象{}内直接写！~~

只能通过Object.defineProperty()或Object.defineProperties()添加

Object.defineProperties(对象, {

1). 先定义一个隐姓埋名且半隐藏的属性准备实际保存属性值

\_属性名:{

value: 属性值,

writable:true, //这种属性值往往可以改，只不过需要验证通过才能改

enumerable:false, //不想让受保护的属性随意被for in遍历到

configurable:false, //以上设置禁止随意修改

},

2). 为要保护的属性: \_属性名 请保镖：

//保镖要冒名顶替\_属性名

对外公开属性名:{ //保镖一请就是一对儿

//专门负责从受保护的\_属性中获取属性值给外部

get:function(){

return this.\_属性名; //从受保护的\_属性中读取属性值

},

//专门负责将要修改的新属性值验证后保存到受保护的\_属性中

set:function(value){

//形参value: 自动接住要修改给属性的新值

if(value符合条件){

this.\_eage=value; //将value新值真正保存到受保护的\_属性中

}else{

throw Error("报错")

}

},

enumerable:true,//保镖要代替\_属性抛头露面，所以应该for in可看到

configurable:false, //以上设置禁止随意修改

//因为保镖不实际存值，所以没有value属性

//因为writable不好用，我们才用的访问器属性，所以一旦用get()和set()就不再需要使用writable开关了。等于用灵活的get()和set()代替了不灵活的writable。

}

})

3). 对象外或其他人使用对象内的访问器属性和使用普通属性完全一样！因为外界根本就不知道他们访问的是一个冒名顶替的访问器属性。蒙在鼓里。

i. 想获取访问器属性值时:

console.log(对象名.访问器属性名)

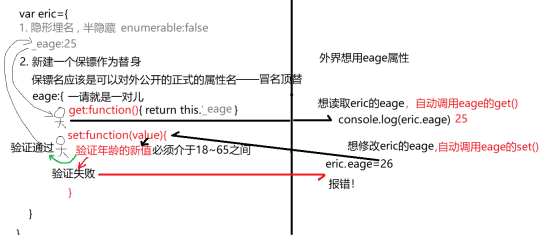
自动调用访问器属性中的get()，从\_属性中获取属性值返回给外部

ii. 想修改访问器属性值时:

对象名.访问器属性名=新属性值

自动调用访问器属性中的set()，并自动将=右侧的新属性值传给set()的形参value。

set()内验证value是否符合要求。如果value符合要求，才将value实际保存到受保护的\_属性中。如果value不符合要求，就报错，并且不会把错误的value保存到\_属性中。

s

f. 示例: 使用访问器属性保护年龄属性: 年龄必须介于18~65之间

|  |
| --- |
| <script>  var eric={  eid:1001,  ename:"埃里克",  eage:25  };  //年龄eage可以修改，但是年龄必须介于18~65之间！  //1. 先将eage属性值保存到一个半隐藏的\_eage属性中  Object.defineProperties(eric,{  \_eage:{//受保护的实际保存数据的属性  value:eric.eage, //将旧eric的eage属性值转义到半隐藏的新的\_eage属性中  writable:true,//年龄可以改  enumerable:false, //不想让受保护的属性随意被for in遍历到  configurable:false, //以上设置禁止随意修改  },  //2. 为\_eage请保镖:  //保镖要冒名顶替eage  eage:{ //保镖一请就是一对儿  //专门负责从受保护的\_eage属性中获取属性值给外部  get:function(){  console.log(`当试图读取eage的属性值时，自动调用一次eage的get()`)  //this->当前eric对象  return this.\_eage;  },  //专门负责将要修改的新属性值验证后保存到受保护的属性中  set:function(value){  console.log(`当试图修改eage的属性值时，自动调用一次eage的set(value=${value})` );  if(value>=18&&value<=65){  this.\_eage=value;  }else{  throw Error("您的年龄超范围，无法保存！")  }  },  enumerable:true,//保镖要代替\_eage属性抛头露面  configurable:false, //以上设置禁止随意修改  //因为保镖不实际存值，所以没有value属性  //因为writable不好用，我们才用的访问器属性，所以一旦用get()和set()就不再需要使用writable开关  }  });  //外界: 使用访问器属性和使用普通属性完全一样！因为外界根本不知道自己访问的是访问器属性。  //想获取eric的年龄  console.log(eric.eage);//  //先修改eric的年龄  eric.eage=26;//  console.log(eric);  //试图修改错误的年龄  eric.eage=-2;  </script>  运行结果:  当试图读取eage的属性值时，自动调用一次eage的get()  25  当试图修改eage的属性值时，自动调用一次eage的set(value=26)  {eid: 1001, ename: "埃里克", \_eage: 26}  当试图修改eage的属性值时，自动调用一次eage的set(value=-2)  Uncaught Error: 您的年龄超范围，无法保存！  at Object.set [as eage] (3\_get\_set.html:39)  at 3\_get\_set.html:56 |

## 5. 保护对象的结构: 3级别

对应小程序视频: 小程序->在线->JSCORE->day06 1. 保护对象结构

(1). 防扩展: 禁止给对象随意添加新属性

a. 旧js中: 对象即使创建完之后，依然可以通过强行赋值方式为对象随意添加新属性

b. 如何防扩展: Object.preventExtensions(对象)

c. 意为: 阻止(添加)扩展(给)对象

d. 示例: 阻止为eric对象添加新属性

|  |
| --- |
| <script>  "use strict";  var eric={  eid:1001, //只读  ename:"埃里克"  };  //设置eid只读  Object.defineProperty(eric,"eid",{  writable:false,  configurable:false  });  //禁止给eric对象扩展新属性  Object.preventExtensions(eric);  //尝试修改eid  //eric.eid=-2; //报错:Cannot assign to read only property 'eid'  // 不能 赋值 给 只读 属性 eid  //但是，因为eid管的太严，屌丝觉得不好用，于是自己另起炉灶，添加了一个甚至多个eid  //eric.Eid=-2; //报错:  //Cannot add property Eid, object is not extensible  //不能 添加 属性 Eid（因为）对象是不 可扩展  console.log(eric);  </script>  运行结果:  Uncaught TypeError: Cannot add property Eid, object is not extensible  at 4\_preventExtensions.html:28 |

(2). 密封: 既禁止添加新属性，又禁止删除现有属性

a. 问题: 对象中几乎所有属性都应该是禁止删除的，那就意味着每个属性都要添加configurable:false开关。——麻烦！

b. 解决: 只要密封对象即可

c. 如何: Object.seal(对象)

密封

d. 意为: 既禁止给对象添加新属性，又禁止删除对象中现有属性

e. 原理: seal()内部自动做了两件事:

1). 自动调用了Object.preventExtensions()——防扩展

2). 自动将所有属性的configurable都改为了false

f. 好处: 从此所有属性的configurable:false，都可省略！

g. 何时: 今后几乎所有对象都应该密封起来！

h. 示例: 密封eric对象

|  |
| --- |
| <script>  "use strict";  var eric={  eid:1001, //只读  ename:"埃里克", //禁止删除  salary:12000 //禁止for in遍历  }  Object.defineProperties(eric,{  eid:{writable:false},  salary:{enumerable:false}  });  //密封eric对象  Object.seal(eric);  //尝试重新打开eid的writable开关  // Object.defineProperty(eric,"eid",{  // writable:true,  // configurable:true  // });//报错: Cannot redefine property: eid  //尝试修改eid的值  // eric.eid=-2;//报错: Cannot assign to read only property 'eid'  //尝试删除ename属性  // delete eric.ename;//报错: Cannot delete property 'ename'  //尝试遍历eric中所有属性  for(var key in eric){  console.log(`属性名:${key},属性值:${eric[key]}`)  }  </script>  运行结果:  Uncaught TypeError: Cannot redefine property: eid  at Function.defineProperty (<anonymous>)  at 5\_seal.html:26 |

(3). 冻结: 既禁止添加新属性，又禁止删除现有属性，甚至连属性值都不能修改了

a. 如何: Object.freeze(对象)

b. 原理: 悄悄做了三件事:

1). 自动调用了Object.preventExtensions()——防扩展

2). 自动将所有属性的configurable都改为false——密封

3). 自动将所有属性的writable都改为false

c. 何时:

1). 大部分对象不用冻结  
 2). 少量多个模块共用的对象，不想因为某一个模块篡改了对象中的内容，而导致牵一发而动全身时，都要将这个共用的对象冻结！（连属性值都不能个修改！）

d. 示例: 冻结一个对象，禁止修改其属性值

|  |
| --- |
| <script>  "use strict"  var pool={  host:"127.0.0.1",//writable:false  port:3306,//writable:false  db:"xz"//writable:false  }  Object.freeze(pool);  pool.host="192.168.0.100";//报错: Cannot assign to read only property 'host'  </script>  运行结果:  Uncaught TypeError: Cannot assign to read only property 'host' of object '#<Object>'  at 6\_freeze.html:19 |

# 二.Object.create()

1. 什么是Object.create(): 专门用于根据一个现有父对象，临时创建一个新的子对象继承当前父对象。

2. 问题: 通常创建子对象都是new 构造函数()，但是如果没有构造函数，只有父对象时，也想创建子对象，继承父对象，就无法用new了！

3. 解决: 用Object.create()临时创建子对象

4. 何时: 即使没有构造函数，只有一个父对象，也想创建子对象继承父对象时，就可用Object.create()

5. 如何:

var 子对象名=Object.create(父对象,{

属性名:{

value:属性值,

开关: true 或 false

},

属性名:{

value:属性值,

开关: true 或 false

},

... ...

})

6. Object.create()一共做了3件事:

(1). 创建一个新的空对象

(2). 让新对象继承()中第一个参数对象father

(3). 将()中第二个对象中的属性添加到新对象中成为新对象的自有属性

7. 示例: 没有构造函数，只有父对象，也想创建子对象继承父对象

|  |
| --- |
| <script>  //只有父对象，没有构造函数  var father={  bal:100000000000,  car:"infiniti"  }  //也想临时创建子对象继承father  var lilei=Object.create(father,{  //不能直接写属性值，必须按照defineProperties的格式写  sname:{  value:"Li Lei",  writable:true,  enumerable:true  },  sage:{  value:11,  writable:true,  enumerable:true  }  });//3件事  Object.seal(lilei);//密封  //1. 创建一个新的空对象  //2. 让新的空对象继承father对象  //3. 为新对象添加自有属性  console.log(lilei);  console.log(lilei.sname, lilei.sage, lilei.bal, lilei.car);  </script>  运行结果:  {sname: "Li Lei", sage: 11}  sage: 11  sname: "Li Lei"  \_ \_proto\_ \_:  bal: 100000000000  car: "infiniti"  \_\_proto\_\_: Object  Li Lei 11 100000000000 infiniti |

# 三. 替换this:

对应小程序视频: 小程序->在线->JSCORE->day06 2. 替换this call,apply,bind

1. 其实如果一个函数中的this不是想要的！是可以随意更换的！

2. 2种情况:

(1). 在一次调用函数时，临时替换一次this为指定的对象

a. 如何: 要调用的函数.call(替换this的对象, 实参值1, 实参值2,...)

b. 结果:

1). 在本次调用函数时，函数中的this会被临时替换为()中指定的任意对象

2). 将从第二个实参开始往后的所有实参值自动传给要调用的函数的形参变量

c. 示例: 使用call替换薪资计算函数中的this，计算不同员工的薪资

|  |
| --- |
| <script>  //有一个公共的计算薪资的函数  //三个形参 底薪 奖金1 奖金2  function jisuan(base, bonus1, bonus2){  console.log(`${this.ename} 的总工资是: ${base+bonus1+bonus2}`)  }  //有两个员工:  var lilei={ ename:"Li Lei" };  var hmm={ ename:"Han Meimei" };    //李磊这个月底薪10000，奖金1:1000, 奖金2:2000，李磊想用jisuan()计算自己的总工资  //错误做法1:  //jisuan() //this->window window.ename->undefined  //错误做法2: lilei能调用全局函数jisuan()  //console.log(lilei); //查看李磊的原型链都没有jisuan()  //因为jisuan()函数保存在全局window中，不属于任何一个对象的原型链，只属于作用域链。  //lilei.jisuan() //报错: lilei.jisuan is not a function  //正确做法:  jisuan.call(lilei,10000,1000,2000)  // | ↓ ↓ ↓  //jisuan( ↓ base,bonus1,bonus2)  // this.ename ....  //call()会临时将jisuan()中的this->lilei  //this.ename -> lilei.ename -> "Li Lei"  //又能将10000,1000,2000依次传给jisuan()函数的base,bonus1,bonus2  //如果hmm也想用jisuan()计算自己的薪资:底薪4000,奖金1:5000,奖金2:6000  jisuan.call(hmm,4000, 5000, 6000)  // | ↓ ↓ ↓  // jisuan( ↓ base,bonus1,bonus2)  // this.ename...  </script>  运行结果：  Li Lei 的总工资是: 13000  Han Meimei 的总工资是: 15000 |

d. 特殊情况: 如果要调用的函数需要传入多个实参值，但是给定的实参值却是放在一个数组中给的。出现了不一致！

e. 解决: 用apply()代替call，将实参值数组直接传入apply()

要调用的函数.apply(替换this的对象, 保存实参值的数组)

f. 原理: apply比call多一项功能，可自动打散数组为多个实参值，再依次分别传给函数的形参变量。

g. 示例: 使用apply()代替call()，先打散数组，再传参

|  |
| --- |
| <script>  //有一个公共的计算薪资的函数  //三个形参 底薪 奖金1 奖金2  function jisuan(base, bonus1, bonus2){  console.log(`${this.ename} 的总工资是: ${base+bonus1+bonus2}`)  }  //一个员工lilei  var lilei={ ename:"Li Lei" };  //李磊想用jisuan()计算自己的薪资  //但是李磊的薪资是放在一个数组中的  var arr=[10000,1000,2000];  //jisuan.call(lilei,arr[0],arr[1],arr[2])  jisuan.apply(lilei,arr)  // | 打散数组为多个值  // | 10000 1000 2000  // | ↓ ↓ ↓  // jisuan( ↓ base,bonus1,bonus2)  // this.ename  </script>  运行结果：  Li Lei 的总工资是: 13000 |

(2). 基于原函数创建一个新函数副本，并永久替换其中的this

a. 问题: 如果一个函数每次调用时都要用call()替换this——麻烦

b. 解决: 创建一个新函数副本，并永久绑定新函数中的this为指定对象

c. 如何: var 新函数=原函数.bind(替换this的对象, 要固定的实参值)

d. 原理: bind共做了3件事:

1). 基于原函数创建一个一模一样的函数副本

2). 永久替换新函数副本中的this为指定对象

3). 永久绑定个别固定的实参值

e. 示例: 基于原函数创建一个新函数副本，并永久绑定this

|  |
| --- |
| <script>  //有一个公共的计算薪资的函数  //三个形参 底薪 奖金1 奖金2  function jisuan(base, bonus1, bonus2){  console.log(`${this.ename} 的总工资是: ${base+bonus1+bonus2}`)  }  //一个员工lilei  var lilei={ ename:"Li Lei" };  //每个月开工资时都要计算自己的薪资  //李磊买一个自己专属的新计算器呢！  var jisuan2=jisuan.bind(lilei,10000);  //2件事:  //创建了一个和jisuan()一模一样的新函数副本  //将新函数jisuan2中的this永久替换为了lilei  //将将函数中第一个形参base永久固定为10000  // 底薪几乎不变  jisuan2(1000,2000);  jisuan2(500,1000);  jisuan2(2000,2000);  </script>  运行结果:  Li Lei 的总工资是: 13000  Li Lei 的总工资是: 11500  Li Lei 的总工资是: 14000 |

# 四. 数组新增函数:

1. 判断: 2个

对应小程序视频: 小程序->在线->JSCORE->day06 3. every原理 判断数组...

(1). 判断数组中是否所有元素都符合要求

// 每个/所有

a. var bool=arr.every(function(elem, i, arr){

return 判断结果

})

b. 原理:

1). every内自带for循环遍历数组中每个元素——信任

2). 每遍历到一个元素就自动调用一次回调函数——信任

3). 每次调用回调函数时都自动传入三个实参值:

i. 将当前正在遍历的一个元素值交给第一个形参elem

ii. 将当前正在遍历到的下标位置交给第二个形参i

iii. 将当前every前的.前的数组对象交给第三个形参arr

4). 在回调函数内部，利用传入的三个实参值判断当前元素是否符合要求，并返回当前元素的判断结果

5). every要求:

i. 所有元素的判断结果都返回true时，整个every才返回true，说明整个数组都是符合要求的。

ii. 反之，如果遍历过程中，碰到一个元素的判断结果返回false，则没必要再向后执行。而是整个every直接返回false，表示当前数组中不是所有元素都符合要求！

c. 示例: 判断数组是否全由奇数组成:

|  |
| --- |
| <script>  var arr1=[1,2,3,4,5]; //false  var arr2=[1,3,5,3,1]; //true  //想判断哪个数组全由奇数组成:  var result1=arr1.every(function(elem,i,arr){  console.log(`arr1.every()自动执行一次回调函数: elem=${elem},i=${i},arr=${arr},本次判断结果: ${elem%2==1}`);  //每个元素都要是奇数才行  return elem%2==1 //只判断当前一个元素是不是奇数  });  var result2=arr2.every(function(elem,i,arr){  console.log(`arr2.every()自动执行一次回调函数: elem=${elem},i=${i},arr=${arr},本次判断结果: ${elem%2==1}`);  return elem%2==1  });  console.log(result1,result2);  </script>  运行结果: |

(2). 判断数组中是否包含符合要求的元素

// 有一些

a. var bool=arr.some(function(elem,i,arr){

return 判断条件

})

b. 原理:

1). some内自带for循环遍历数组中每个元素——信任

2). 每遍历到一个元素就自动调用一次回调函数——信任

3). 每次调用回调函数时都自动传入三个实参值:

i. 将当前正在遍历的一个元素值交给第一个形参elem

ii. 将当前正在遍历到的下标位置交给第二个形参i

iii. 将当前every前的.前的数组对象交给第三个形参arr

4). 在回调函数内部，利用传入的三个实参值判断当前元素是否符合要求，并返回当前元素的判断结果

5). some要求:

i. 只要碰到一个元素检查结果为true，就不必再继续，整个some就返回true，说明数组中包含符合要求的元素。但是包含几个，不知道！

ii. 反之，如果some遍历结束都没有碰到一个检查结果为true的元素，整个some才被迫返回false，说明整个数组中都不包含任何符合要求的元素。

c. 示例: 判断哪个数组中包含偶数:

|  |
| --- |
| <script>  var arr1=[1,2,3,4,5]; //true  var arr2=[1,3,5,3,1]; //false  //想判断哪个数组包含偶数  var result1=arr1.some(function(elem,i,arr){  console.log(`arr1.some()自动执行一次回调函数: elem=${elem},i=${i},arr=${arr},本次判断结果: ${elem%2==0}`);  //如果当前元素是偶数就返回true  return elem%2==0  });  var result2=arr2.some(function(elem,i,arr){  console.log(`arr2.some()自动执行一次回调函数: elem=${elem},i=${i},arr=${arr},本次判断结果: ${elem%2==0}`);  //如果当前元素是偶数就返回true  return elem%2==0  });  console.log(result1,result2);  </script>  运行结果: |

2. 遍历

3. 过滤

4. 汇总

**Day07**正课:

# 一. 数组新增函数:

## 1. 遍历:

依次取出数组中每个元素，执行相同的操作。2个

## (1). forEach: 单纯为了简化普通for循环遍历索引数组

对应小程序视频: 小程序->在线->JSCORE->day06 4. forEach

a. arr.forEach(function(elem,i,arr){

对每个元素值要执行的操作

})

b. 原理:

1). forEach也自带for循环，**自动**遍历数组中每个元素

2). 每遍历一个元素，就自动调用一次回调函数

3). 每次调用回调函数时，都会自动传入三个实参值:

#### i. 当前元素值给elem

#### ii. 当前元素下标位置给i

#### iii. 当前数组对象给arr

c. 示例: 点名:

1\_forEach.html

|  |
| --- |
| <script>  var arr=["亮亮","然然","东东"];  //依次点名，点到谁，谁喊到！  //用for: 无进一步简化的可能了  // for(var i=0;i<arr.length;i++){  // alert(`${arr[i]} - 到！`);  // }  //用arr.forEach():  // arr.forEach(function(elem,i,arr){  // console.log(`forEach自动调用了一次回调函数, elem=${elem}, i=${i}, arr=${arr}`);  // alert(`${elem} - 到！`);  // })  //arr.forEach(function(elem,i,arr){  //回调函数内容中本次不需要i和arr，可省略  //arr.forEach(function(elem){  //提前尝鲜: 箭头函数=>  // arr.forEach((elem)=>{  // alert(`${elem} - 到！`);  // })  //箭头函数可以更简化: 如果只有一个形参可省略()，如果只有一句话，可省略{}  arr.forEach(elem=>alert(`${elem} - 到！`))  //总结: forEach好处是可以和箭头函数配合有进一步简化的可能！  //关于箭头函数，先不要问为什么！后边再详细讲！  </script>  运行结果: |

## (2). map: 依次取出原数组中每个元素值，执行相同的操作后，将修改后的新值放入一个新数组中返回。原数组保持不变。

对应小程序视频: 小程序->在线->JSCORE->day06 6. map原理 数组元素\*2...

a. var 新数组=arr.map(function(elem, i, arr){

return 将当前元素值加工后的新值

})

b. 原理:

1). map内部先自动创建一个空的新数组，等待接收加工后的新元素值

2). map也自带for循环，自动遍历数组中每个元素

3). 每遍历一个元素就自动调用一次回调函数

4). 每次调用回调函数时，都自动传入三个实参值:

#### i. 当前元素值给elem

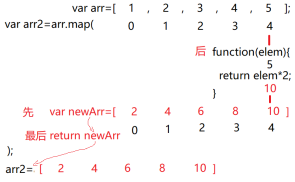
#### ii. 当前元素下标位置给i

#### iii. 当前数组对象给arr

5). 回调函数内部会对本次取出的当前元素值进行加工，并返回加工后个一个新元素值

6). map会将回调函数本次返回的新值，保存到新数组中相同的位置上。

7). 遍历结束后，map会将包含修改后的新元素值的新数组返回



c. 示例: 将原数组中每个元素值\*2，并放入新数组中返回2\_map.html

|  |
| --- |
| <script>  var arr=[1,2,3,4,5];  //想对数组中每个元素\*2后，放入一个新数组返回，保护原数组不变  var arr2=arr.map(function(elem){  return elem\*2;  });  console.log(arr);  console.log(arr2);  </script>  运行结果:  [1, 2, 3, 4, 5]  [2, 4, 6, 8, 10] |

## 过滤:

复制出原数组中符合条件的元素值，放入新数组中返回，原数组中的元素保持不变，不会少元素！

对应小程序视频: 小程序->在线->JSCORE->day06 7. filter

a. var 新数组=arr.filter(function(elem,i,arr){

return 判断条件

})

b. 原理:

1). filter内部先自动创建一个空的新数组，等待接收加工后的新元素值

2). filter也自带for循环，自动遍历数组中每个元素

3). 每遍历一个元素就自动调用一次回调函数

4). 每次调用回调函数时，都自动传入三个实参值:

i. 当前元素值给elem

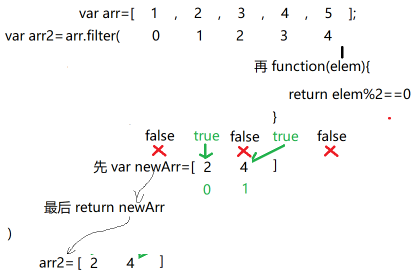
ii. 当前元素下标位置给i

iii. 当前数组对象给arr

5). 回调函数内部会判断当前元素值是否符合要求，并返回判断结果

6). 如果本次判断结果为true，则filter自动将当前元素追加到新数组中。如果本次判断结果为false，则filter什么也不干！

7). 遍历结束后，filter会将包含符合条件的元素的新数组返回



c. 示例: 复制出数组中的偶数，放入新数组中返回 3\_filter.html

|  |
| --- |
| <script>  var arr=[1,2,3,4,5];  //想复制出数组中的偶数组成新数组  var arr2=arr.filter(function(elem){  return elem%2==0  });  console.log(arr);  console.log(arr2);  </script>  运行结果:  [1, 2, 3, 4, 5]  [2, 4] |

## 3. 汇总:

对数组中所有元素求和

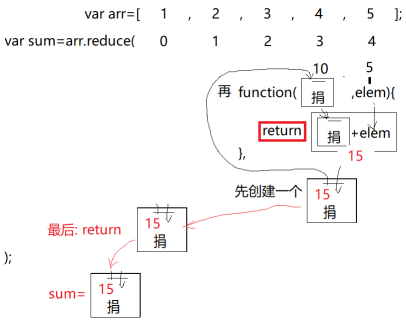
对应小程序视频: 小程序->在线->JSCORE->day06 8. reduce原理 简单 数组元素求和

a. var 汇总结果=arr.reduce(function(捐款箱, elem, i, arr){

return 旧捐款箱中的值加上当前新元素值获得的新汇总值，自动放入捐款箱中

},捐款箱中的起始值)

b. 原理:



1). reduce函数内，先创建一个变量，暂且称为捐款箱，保存reduce函数的第二个实参值——汇总的起始值。

2). reduce函数内也自带for循环，自动遍历数组中每个元素

3). 每遍历到一个元素就自动调用一次回调函数

4). 每次调用回调函数时，都会自动传入四个值:

i. 捐款箱: 当前reduce函数内保存的捐款箱变量中的临时汇总值

ii. elem: 当前新元素值

iii. i: 当前遍历到的位置

iv. arr: 当前数组对象

5). 在回调函数内部，要将新遍历到的当前元素值和捐款箱变量中的临时汇总值相加得到新的临时汇总值，并将新汇总值返回给reduce函数

6). reduce函数每次收到回调函数返回的新临时汇总值，都重新保存回捐款箱变量中，为继续累加下一个元素值做准备。

7). 当所有元素都遍历完，reduce就返回最后捐款箱变量中的汇总值，就是这个数组所有元素的和。

c. 示例: 对一个数组中的元素求和

4\_reduce.html

|  |
| --- |
| <script>  var arr=[1,2,3,4,5];  //相对数组中的元素值求和  var sum=arr.reduce(function(捐款箱,elem){  console.log(`自动调用一次回调函数:捐款箱中现有${捐款箱}元, 当前新元素值${elem}元，现在捐款箱中的钱+当前新元素值=${捐款箱+elem}元，还放回捐款箱，为继续累加下一个元素值做准备.`)  return 捐款箱+elem  },0);//捐款箱中的钱数从0开始累加  console.log(sum);  </script>  运行结果:  自动调用一次回调函数:捐款箱中现有0元, 当前新元素值1元，现在捐款箱中的钱+当前新元素值=1元，还放回捐款箱，为继续累加下一个元素值做准备.  自动调用一次回调函数:捐款箱中现有1元, 当前新元素值2元，现在捐款箱中的钱+当前新元素值=3元，还放回捐款箱，为继续累加下一个元素值做准备.  自动调用一次回调函数:捐款箱中现有3元, 当前新元素值3元，现在捐款箱中的钱+当前新元素值=6元，还放回捐款箱，为继续累加下一个元素值做准备.  自动调用一次回调函数:捐款箱中现有6元, 当前新元素值4元，现在捐款箱中的钱+当前新元素值=10元，还放回捐款箱，为继续累加下一个元素值做准备.  自动调用一次回调函数:捐款箱中现有10元, 当前新元素值5元，现在捐款箱中的钱+当前新元素值=15元，还放回捐款箱，为继续累加下一个元素值做准备.  15 |

\*\*\*\*ES6\*\*\*\*

# 一. 模板字符串

对应小程序视频: 小程序->在线->JSCORE->day07 1. 模板字符串

1. 什么是: 支持换行以及动态生成内容的特殊字符串

2. 为什么: 旧js中，用+拼接字符串，极容易和算术计算的+法计算混淆

3. 何时: 今后，只要动态拼接字符串内容时，都用模板字符串代替传统的用+拼接

4. 如何:

(1). 整个模板字符串必须用一对儿反引号`包裹——键盘左上角ESC键正下方那个键

(2). 在模板字符串中支持换行

(3). 在模板字符串如果包含用js表达式动态生成的内容，必须用${}包裹

5. 结果: js程序会先将${}中的js表达式结果计算出一个子字符串，然后再将子字符串拼接到原字符串中${}位置代替。结果，模板字符串其实还会变为一个普通的字符串。

6. 总结: ${}内都可以放哪些东西？不能放哪些东西？和Vue框架中的{{}}要求是一样的！

(1). 可以放: 变量，运算，三目运算，有返回值的函数调用，创建对象，访问数组元素

(2). 不能放:

a. 程序结构(分支和循环)

b. 没有返回值的函数调用或js表达式

7. 示例: 使用模板字符串，动态生成字符串内容:

5\_templateString.html

|  |
| --- |
| <script>  var price=12.5; //单价  var count=5; //数量  //打印一张简单的超市小票  console.log(`  单价: ¥${price.toFixed(2)}  数量: ${count}  ------------------------------  小计: ¥${(price\*count).toFixed(2)}  `);  var sex=1;//性别: 男  var orderTime=1591338064518;//下单时间:2020/6/5 14:21  var week=["日","一","二","三","四","五","六"];  // 0 1 2 3 4 5 6  console.log(`  性别:${sex==1?"男":"女"},  下单时间:${new Date(orderTime).toLocaleString()},  今天星期${week[new Date().getDay()]}  `)// 5  // week[5]  // 五  </script>  运行结果：  单价: ¥12.50  数量: 5  ------------------------------  小计: ¥62.50    性别:男,  下单时间:2020/6/5 下午2:21:04,  今天星期五 |

# 二. let

对应小程序视频: 小程序->在线->JSCORE->day07 2. let

1. 什么是let: ES6中专门代替var，用来声明变量新关键字

2. 为什么: var的缺点:

(1). var会被声明提前——破坏了程序的执行顺序

(2). var声明的变量没有块级作用域——导致块内的变量会提升到块外，影响外部代码

3. 示例: var的缺点:

6\_var.html

|  |
| --- |
| <script>  var t=0;//定义一个全局变量，用来累加以下两个函数的执行时间  function fun1(){  console.log(`fun1耗时0.3s`);  t+=0.3;  }  function fun2(){//是作用域  //var t; //undefined  //从此: 因为fun2中有了局部变量t，就不会再用全局的t了  console.log(`fun2耗时0.8s`);  t+=0.8;//只是加在局部变量t上  //后期维护过程中，又新增了一段代码:  if(false){//不是作用域 //根本不会执行  var t=new Date();  console.log(`当前时间: ${t.toLocaleString()}`)  }//虽然这段代码没有执行，但是却让原本正常执行的其它程序，发生错误！  }  fun1();  fun2();  console.log(`共耗时${t}s`);//1.1  </script>  运行结果:  fun1耗时0.3s  fun2耗时0.8s  共耗时0.3s |

4. 何时: 今后新的框架程序中都是用let代替var的

5. 优点: 2个:

(1). let不会被声明提前——保证程序顺序执行

(2). let为js添加了块级作用域的效果——防止块内的变量，影响到块外部。

6. 原理: let做了两件事:

(1). let相当于匿名函数自调

(2). let还可能自动修改变量名

7. 示例: 使用let代替var

7\_let.html

|  |
| --- |
| <script>  var t=0;//定义一个全局变量，用来累加以下两个函数的执行时间  function fun1(){  console.log(`fun1耗时0.3s`);  t+=0.3;  }  function fun2(){//是作用域  console.log(`fun2耗时0.8s`);  t+=0.8;//只是加在局部变量t上  //后期维护过程中，又新增了一段代码:  if(true){//变成作用域了  //let 让if后{也变成了作用域  let t=new Date();  console.log(`当前时间: ${t.toLocaleString()}`);  //在底层已经被翻译为以下代码:  // (function(){  // var \_t=new Date();//成了if内的局部变量，不会影响外部  // console.log(`当前时间: ${\_t.toLocaleString()}`)  // })();  }//就算if执行，里边的let t和外边的t，不会互相影响  }  fun1();  fun2();  console.log(`共耗时${t}s`);//1.1  </script>  运行结果:  fun1耗时0.3s  fun2耗时0.8s  当前时间: 2020/6/5 下午3:23:21  共耗时1.1s |

8. let的三个小脾气:

(1). 即使在全局声明的let，也不会保存到window中！

(2). 在let之前禁止提前使用该变量！

(3). 在同一作用域内，不允许反复let同名的变量

9. 示例: let的小脾气

8\_let2.html

|  |
| --- |
| <script>  var a=10;  console.log(window.a);//10  //(function(){  let b=10;  console.log(window.b);//undefined  //})()  console.log(c);//undefined  var c=10;  console.log(c);//10  //console.log(d);//报错: Cannot access 'd' before initialization  // 不能 访问 的 在 初始化 之前  //let d=10;  //console.log(d);//10  var e=10;  var e=100;  console.log(e);//100  //let f=10;  //let f=100; //报错：Identifier 'f' has already been declared  // 标识符/变量名 f 已经 被 声明过了  </script>  运行结果:  10  undefined  undefined  10  100 |

# 三. 箭头函数

对应小程序视频: 小程序->在线->JSCORE->day07 3. 箭头函数

1. 什么是箭头函数: 对旧js中function的简写

2. 为什么: 简单！

3. 何时: 今后几乎所有的function都可用箭头函数简化——在将来框架开发中几乎看不到任何function了！

4. 如何: 3句话口诀

(1). 去掉function在()和{}之间加=>

(2). 如果()中只有一个形参，可省略()，但是如果()中没有形参，则()不能省略

(3). 如果函数体只有一句话，可省略{}和这条语句结尾的;

强调: 如果仅有的一句话还是return，则必须省略return——语法规定，没有为什么

5. 示例: 使用箭头函数简写各种function

9\_arrow.html

|  |
| --- |
| <script>  //function add(a,b){  //var add=function(a,b){  var add=(a,b)=>//{  console.log(`${a}+${b}=${a+b}`)  //}  add(3,5);  var arr=[12,3,123,32,2,1];  //arr.sort(function(a,b){ return a-b; });  arr.sort((a,b)=>a-b)  console.log(arr);  var str="you can you up";  // str=str.replace(/\b[a-z]/g, function(keyword){  // return keyword.toUpperCase();  // })  str=str.replace(/\b[a-z]/g,kw=>kw.toUpperCase())  console.log(str);  var arr=["亮亮","然然","东东"]  //arr.forEach(function(elem){  arr.forEach(elem=>/\*{\*/  console.log(`${elem} - 到`)  /\*}\*/);  var arr=[1,2,3,4,5];  // var arr2=arr.map(function(elem){  // return elem\*2;  // });  var arr2=arr.map(elem=>elem\*2)  console.log(arr);  console.log(arr2);  var arr=[1,2,3,4,5];  // var sum=arr.reduce(function(box,elem){  // return box+elem  // },0);//从0开始累加  var sum=arr.reduce((box,elem)=>box+elem,0)  console.log(sum);  //(function(){  (()=>{  console.log("我是匿名函数自调");  console.log(`当前系统时间:${new Date().toLocaleString()}`)  })();  var t=5;  //var timer=setInterval(function(){  var timer=setInterval(()=>{  if(t>0){//如果倒计时没结束  console.log(t);//就输出倒计时  t--;//倒计时-1  }else{//否则如果倒计时已经结束  console.log(`boom!!!`);//就爆炸  clearInterval(timer);//并停止定时器  }  },1000)  </script>  运行结果:  3+5=8  [1, 2, 3, 12, 32, 123]  You Can You Up  亮亮 - 到  然然 - 到  东东 - 到  [1, 2, 3, 4, 5]  [2, 4, 6, 8, 10]  15  我是匿名函数自调  当前系统时间:2020/6/5 下午4:12:28  5  4  3  2  1  boom!!! |

6. 箭头函数的特征: 箭头函数内的this与箭头函数外的this始终保持一致.

强调: 这个特点仅作用域this，其他变量的作用域保持不变！

7. 总结: 何时可以使用箭头函数，何时不能用箭头函数:

(1). 可以使用箭头函数:

a. 如果函数中根本就没有用到this，可随意用箭头函数简写

b. 如果函数中用到了this，但恰好希望函数内的this与函数外的this保持一致，则可以用箭头函数简写

(2). 不能用箭头函数简写:

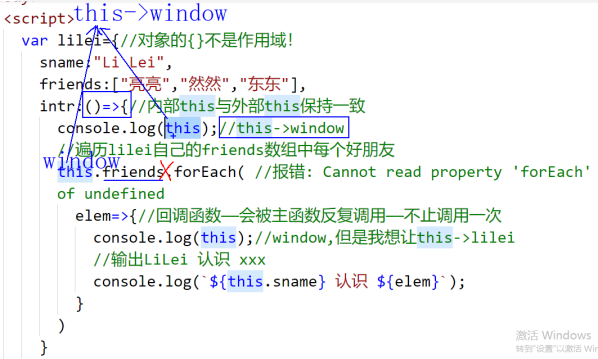
c. 如果函数中用到了this，但是不希望函数中的this与函数外的this保持一致时，则不能用箭头函数简写，典型场景: 对象中的方法不能用箭头函数简写

8. 示例: 使用箭头函数简写回调函数和对象中的方法

10\_arrow\_this.html







|  |
| --- |
| <script>  var lilei={//对象的{}不是作用域！  sname:"Li Lei",  friends:["亮亮","然然","东东"],  intr:function(){  //遍历lilei自己的friends数组中每个好朋友  this.friends.forEach(  elem=>{//回调函数——会被主函数反复调用——不止调用一次  console.log(this);//window,但是我想让this->lilei  //输出LiLei 认识 xxx  console.log(`${this.sname} 认识 ${elem}`);  }  )  }  }  lilei.intr();  function liang(帮别人做一件事){  console.log(`亮哥自己先吃完饭`);  /\* \*/帮别人做一件事();//因为回调函数在主函数内调用时，前边既没有.，又没有new，所以通常回调函数中的this只能默认指window  }  //然哥想让liang帮忙带一屉包子  liang(function(){  console.log(this);  console.log(`帮然哥带包子`)  });  </script>  运行结果：  {sname: "Li Lei", friends: Array(3), intr: ƒ}  Li Lei 认识 亮亮  {sname: "Li Lei", friends: Array(3), intr: ƒ}  Li Lei 认识 然然  {sname: "Li Lei", friends: Array(3), intr: ƒ}  Li Lei 认识 东东  亮哥自己先吃完饭  Window {parent: Window, opener: null, top: Window, length: 0, frames: Window, …}  帮然哥带包子 |

# 四. for of

对应小程序视频: 小程序->在线->JSCORE->day07 4. for of

1. 什么是for of: 对大多数for循环遍历索引数组的简写

2. 何时: 今后绝大多数情况下从头到尾的遍历索引数组和类数组对象，中每个元素都可用for of简化普通for循环

3. 如何:

for(var 变量 of 数组/类数组对象){

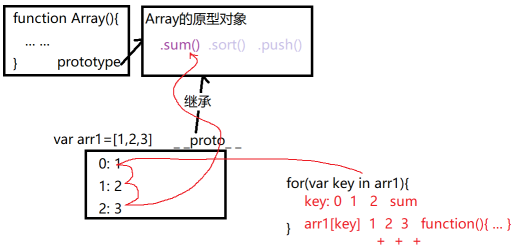
//of会依次取出数组中每个元素值，自动保存到of前的变量中

对of前变量执行操作

}

4. 总结:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | for | arr.forEach | for of | for in |
| 数字下标 | 索引数组 | √ | √ | √ | × |
| 类数组对象 | √ | × | √ | × |
| 字符串 | √ | × | √ | × |
| 自定义名称下标 | 关联数组 | × | × | × | √ |
| 对象 | × | × | × | √ |



5. for of的局限: vs for循环

(1). for(var i=0;i<arr.length;i++)

a. 可改变遍历的顺序: for(var i=arr.length-1;i>=0;i--){ ... }

b. 可改变遍历的步调: for(var i=0;i<arr.length;i+=2)

c. 既可以获得元素的内容arr[i]，又可以获得元素的位置i

(2). for(var elem of arr)

a. 不可改变遍历的顺序，只能从头到尾

b. 不可改变遍历的步调，只能依次挨个遍历

c. 只能获得元素值elem，无法获得元素位置i

6. 总结: 因为绝大多数循环，不需要改变顺序，步调，只关心元素值，不关心位置，所以反而for of用的绝对多！

7. 示例: 使用for of遍历索引数组和类数组对象

11\_for\_of.html

|  |
| --- |
| <script>  var arr=["亮亮","然然","东东"];  //点名  for(var i=0;i<arr.length;i++){  console.log(`${arr[i]} - 到！`)  }  arr.forEach(elem=>console.log(`${elem} - 到！`));  for(var elem of arr){console.log(`${elem} - 到！`)}  //定义函数对任意个数字求和  function add(){  var sum=0;  //for(var i=0;i<arguments.length;i++){  for(var value of arguments){  sum+=value  }  return sum;  }  console.log(add(1,2,3))//6  console.log(add(1,2,3,4,5))//15  //为数组家添加一个对数组元素求和的共有方法sum  Array.prototype.sum=function(){  //this->将来的arr1, arr2  var total=0;  // for(var i=0;i<this.length;i++){  // total+=this[i]  // }  //this.forEach(elem=>total+=elem);  for(var elem of this){  total+=elem;  }  //错误: in不止遍历当前对象的成员，还会自动沿着原型链向上继续遍历父对象中可用的成员  //in会依次取出in后对象中每个属性名保存到in前的变量key中  // for(var key in this){  // total+=this[key]  // }  return total;  }  var arr1=[1,2,3];  var arr2=[1,2,3,4,5]  console.log(arr1.sum());  console.log(arr2.sum());  </script>  运行结果:  亮亮 - 到！  然然 - 到！  东东 - 到！  亮亮 - 到！  然然 - 到！  东东 - 到！  亮亮 - 到！  然然 - 到！  东东 - 到！  6  15  6  15 |

正课: ES6 —— 难

1. 参数增强

2. 解构

3. class

4. Promise

# 一. 参数增强:

对应小程序视频: 小程序->在线->JSCORE->day07 5. 参数增强 ...

## 1. 参数默认值

(1). 什么是: 定义函数时，希望将来就算没有给实参值，形参变量也有一个默认值可以使用。

(2). 如何: function 函数名(形参变量=默认值){ ... }

(3). 原理:

a. 如果调用函数时只要传了实参值（!==undefined），都优先使用调用者传入的自定义实参值

b. 只有调用函数时没有传入实参值时，才被迫使用=后的默认值作为参数变量的默认值使用。

(4). 示例: 为函数形参变量指定参数默认值

1\_default.html

|  |
| --- |
| <script>  //定义一个函数可以输出我的自我介绍  // 默认值  function intr(intrSelf="主人很懒，什么也没留下"){  document.write(`<h1>我的自我介绍: ${intrSelf}</h1>`)  }  //调用者传入了自我介绍的内容  intr("you can you up, no can no bibi");  //调用者没有传入自我介绍的内容，也想有默认值占位  intr();  //如果恶意传入""  intr(""); //因为""不等于undefiend，所以不启用默认值！而使用""  //如果恶意传入undefined  intr(undefined);//因为传入的值也是undefined，所以认为什么都没传入，所以启用默认值！  </script>  运行结果:  我的自我介绍: you can you up, no can no bibi  我的自我介绍: 主人很懒，什么也没留下  我的自我介绍:  我的自我介绍: 主人很懒，什么也没留下 |

## 2. 剩余参数：

(1). 什么是剩余参数: 专门用于代替arguments接收不确定个数的实参值的新语法

(2). 为什么: arguments的缺点:

a. 箭头函数中不能使用arguments

b. 不是纯正的数组，无法使用数组家的函数

c. 只能接收所有实参值，无法有选择的接收部分实参值

d. 名字太长，且没有意义！

(3). 何时: 今后只要在ES6中获取不确定个数的实参值，都用剩余参数语法代替arguments

(4). 如何: 定义函数时

function 函数名(形参变量, 形参变量, ...数组名){ }

(5). 结果: ...运算符专门收集除前几个形参之外剩余的实参值，放入一个自定义数组中返回.

(6). 示例: 使用剩余参数语法计算不确定个数的数字的和

2\_rest.html

|  |
| --- |
| <script>  //错误:  // var add=()=>{  // var sum=0;  // for(var elem of arguments){//报错: arguments is not defined  // sum+=elem  // }  // return sum;  // }  //正确:  // var add=(...arr)=>{  // //arr:[1,2,3]  // var sum=0;  // for(var value of arr){  // sum+=value  // }  // return sum;  // }  //正确: 证明arr是一个纯正的数组  // var add=(...arr)=>{  // 数组家的  // var result=arr.reduce(function(捐款箱,elem){  // return 捐款箱+elem  // },0);  // return result;  // }  //正确: 终极简写，时髦！—— 学有余力的同学  var add=(...arr)=>arr.reduce((捐款箱,elem)=>捐款箱+elem,0);  console.log(add(1,2,3));//6  console.log(add(1,2,3,4,5));//15  运行结果:  6  15 |

(7). 示例: 使用剩余参数计算薪资

3\_rest2.html

|  |
| --- |
| <script>  //定义一个计算薪资的函数  // 收集除第一个ename之外剩余的其它实参值  function jisuan(ename,...arr){  //因为arr是一个纯正的数组  var total=arr.reduce(function(捐款箱,elem){  return 捐款箱+elem  },0);  console.log(`${ename}的总工资是：${total}`)  }  jisuan("Li Lei",10000,2000);  jisuan("Han Meimei",3000,4000,5000,1000)  </script>  运行结果:  Li Lei的总工资是：12000  Han Meimei的总工资是：13000 |

(8).优点:

a. 箭头函数支持

b. 是一个纯正的数组，可使用数组家函数

c. 可有选择的获取部分实参值，不一定非要获取全部实参值

d. 可自定义数组名，简单，见名知意

## 3. 打散数组和对象

(1). 什么是: 将数组或对象打散为多个实参值，再传入函数中

(2). 何时: 如果要调用的函数需要多个实参值，但是多个实参值却是放在一个数组中给的，出现不一致，就可先打散，再传参

(3). 如何: 调用函数时:

要调用的函数(...数组名)

(4). 原理: 先打散数组为多个值，再分别传给要调用的函数的形参变量

(5). 总结:

a. 定义函数时，形参列表中的...表示收集剩余参数

b. 调用函数时，实参值中的...表示打散数组再传参

(6). 示例: 查找数组中的最大值和最小值

4\_spread.html

|  |
| --- |
| <script>  //回顾: Math.max() Math.min()  console.log(`最大值: ${Math.max(2,5,7,1)}`);  console.log(`最小值: ${Math.min(2,5,7,1)}`);  //通常程序中保存多个值都是放在数组中保存  var arr=[2,5,7,1];  //想获得数组中多个元素值中的最大值和最小值  //错误:  console.log(`最大值: ${Math.max(arr)}`);//NaN  console.log(`最小值: ${Math.min(arr)}`);//NaN  //说明Math.max()和Math.min()不支持数组中查找最大值和最小值  //解决: 先打散数组再传参  //正确:  console.log(`最大值: ${Math.max(...arr)}`);//7  console.log(`最小值: ${Math.min(...arr)}`);//1  </script>  运行结果:  最大值: 7  最小值: 1  最大值: NaN  最小值: NaN  最大值: 7  最小值: 1 |

(7). 时髦: 其实都是先打散再粘起来

a. 复制一个数组: var arr2=[...arr1];

b. 复制一个对象: var obj2={...obj1};

c. 合并两个数组为一个新数组: var arr2=[...arr1,...arr2];

d. 合并两个对象为一个新对象: var obj3={...obj1, ...obj2};

(8). 示例: 使用...简写复制数组和对象，以及合并数组和对象

5\_spread2.html

|  |
| --- |
| <script>  //复制一个数组  var arr1=[1,2,3];  var arr2=[...arr1];  console.log(arr2);  console.log(arr1==arr2);//false  //复制一个对象  var lilei={  sname:"Li Lei",  sage:11  }  var lilei2={...lilei};  console.log(lilei2)  console.log(lilei==lilei2);//false    //合并两个数组  var arr1=[1,2,3];  var arr2=[6,7,8];  var arr3=[...arr1,4,5,...arr2];  console.log(arr3);  //合并两个对象:  var lilei={  sname:"Li Lei",  sage:11  };  var lilei2={  className:"初一2班",  intr:function(){  console.log(`I'm Li Lei, I'm 11`)  }  };  var lilei3={  ...lilei,  phone:"iPhoneX",  ...lilei2  }  console.log(lilei3);  </script>  运行结果:  [1, 2, 3]  false  {sname: "Li Lei", sage: 11}  false  [1, 2, 3, 4, 5, 6, 7, 8]  {sname: "Li Lei", sage: 11, phone: "iPhoneX", className: "初一2班", intr: ƒ} |

# 二. 解构:

1. 什么是: 从一个大的数组或对象中只提取出个别想用的成员保存在变量中，单独使用

2. 为什么: 在每次使用数组或对象成员时，总要arr[i]或对象.属性名——麻烦

3. 何时: 今后只要希望从一个大的数组或对象中只提取出个别成员单独使用时，就可用解构

## 4. 如何: 3种情况:

(1). 数组解构:

对应小程序视频: 小程序->在线->JSCORE->day07 6. 解构(上) ...

a. var [变量1, 变量2,...]=数组

0 1

b. 结果: =右边数组中对应位置的元素值会自动赋值给=左边[]中相同位置的变量

变量1=数组[0]

变量2=数组[1]

c. 强调: =左边的[]不是创建数组的意思！仅仅是将=左边的变量们装扮成数组的样子，骗取=右侧数组的信任，以便获得=右侧数组中对应位置的值。

d. 固定套路: 不声明第三个变量，交换两个变量的值: [b,a]=[a,b]

e. 示例: 使用解构提取数组中保存的年月日，以及交换两个变量的值

6\_destruct\_arr.html

|  |
| --- |
| <script>  var arr=[2020,6,25];  // 年 月 日  //var [y,m,d]=arr;  //console.log(`今年端午节放假:${y}年${m}月${d}日`)  //错误:  //var [m,d]=arr; //m=arr[0]=2020, d=arr[1]=6  //正确:  var [ ,m,d]=arr; //m=arr[1]=6, d=arr[2]=25  // 0 1 2  console.log(`今年端午节放假:${m}月${d}日`)  //不能声明第三个变量，交换两个变量的值  var a=3;  var b=5;  // 先将a和b放入一个数组中  [b,a]=[a,b]  //再用b和a解构出来对应位置的元素值  console.log(a, b);//5 3  </script>  运行结果:  今年端午节放假:6月25日  5 3 |

(2). 对象解构

对应小程序视频: 小程序->在线->JSCORE->day07 6. 解构(上) ...

a. 什么是: 提取出对象中个别属性值或方法，保存到单独变量中简化使用

b. 如何: var {属性名1: 变量1, 属性名2:变量2, ...:... }=对象

c. 结果: =左边对应属性名后的变量会自动收到=右边对象中同名属性的属性值

变量1=对象.属性名1

变量2=对象.属性名2

d. 其中:

1). {}不是创建对象的意思，仅仅是将等号左边的多个变量装扮成对象的样子，来骗取等号右边的对象的信任，从而自动提取出相同属性名的属性值保存到对应变量中

2). =左边{}里的属性名专门用于配对

e. 问题: 如果被解构出来的方法中用到this，this就是会变成window！

所以: 如果对象方法中用到this，不应该用这种方式解构！

f. 简写: 多数情况，对象中的成员名已经很简洁，很见名知意了，不需要改名。但是我们又不想把相同的属性名和变量名写两遍，其实是需要写一遍成员名即可:

1). var {成员名1, 成员名2,...}=对象

2). 相当于: var {成员名1:同名变量1, 成员名2:同名变量2,...}=对象

3). 结果: 一个名字两用: 既当做配对时的属性名，又当做将来单独使用时的变量名。

g. 总结: 今后对象结构时，=左边的{}中都使用简写，只写一个成员即可！

h. 示例: 从对象中解构出个别属性和方法单独使用

7\_destruct\_obj.html

|  |
| --- |
| <script>  var user={  uname:"dingding",  phone:"18301092802",  sex:1,  age:20,  intr:function(){  console.log(`I'm ${this.uname}, I'm ${this.age}`);  },  login:function(){  console.log("登录...")  },  logout:function(){  console.log("注销...")  },  changePwd:function(){  console.log("修改密码...")  }  }  //在当前页面中只想使用uname和changePwd两个成员  // 属性名:变量名, 属性名:变量名  // var {uname:u, changePwd:change}=user;  // //u=user.uname  // //change=user.changePwd  // console.log(`欢迎 ${u}`);  // change();  //如果我们觉得对象中的属性名已经很简洁，且见名知意了，不想改名  //var {uname:uname, changePwd:changePwd}=user;  //简写:一个名字两用,既当做配对的属性名，又当做将来单独使用时的变量名  var {uname, changePwd, intr}=user; //今后对象解构基本都只写一个名字  console.log(`欢迎 ${uname}`);//全局变量  changePwd();  //用旧方法调用  user.intr();//this->user this.uname=user.uname=dingding  //解构后再调用  intr();//this->window this.uname=window.uname=dingding  //this.age=window.age=undefined  </script>  运行结果:  欢迎 dingding  修改密码...  I'm dingding, I'm 20  I'm dingding, I'm undefined |

(3). 参数解构

对应小程序视频: 小程序->在线->JSCORE->day08 1. 解构(下) ...

a. 问题: 如果一个函数多个实参值都不确定有没有，但是又要求传入的实参值必须固定传给指定的形参变量才行

b. 错误解决: 单靠默认值无法解决所有需求！

c. 正确解决: 参数解构: 2步:

1). 定义函数时，先将形参列表装扮为一个对象的样子

function 函数名({

配对儿

属性名1: 形参1,

属性名2: 形参2,

属性名3: 形参3

}){

}

2). 调用函数时，所有实参值也必须放入一个相同的对象结构中传入

函数名({

属性名1:实参值1,

属性名2:实参值2,

属性名3:实参值3

})

d.结果: 形参1=实参值1, 形参2=实参值2, 形参3=实参值3

e. 简写: 定义函数时，通常用于配对的属性名和形参变量名几乎都是相同的，所以，形参列表对象中的名称只要写一个，就可以一个名字两用。而且，一旦改为一个名字，就可以继续支持参数默认值语法了。

function 函数名({

//一个名字两用: 既用作将来配对的属性名，又用作接收实参值的形参变量名

形参名1=默认值,

形参名2=默认值,

... ...

})

f. 示例: 定义一个点套餐的函数，支持各种需求的客户随意更换套餐中的内容

8\_destruct\_arguments.html

|  |
| --- |
| <script>  //要定义一个点套餐的函数  //一个套餐中包含三种商品:  //主食: zhushi 默认: 香辣鸡腿堡  //小吃: xiaochi 默认: 薯条  //饮料: yinliao 默认: 可乐  // function order({  // //配对儿 接收实参值  // //属性名 : 形参  // zhushi:zhushi, //不支持默认值  // xiaochi:xiaochi, //不支持默认值  // yinliao:yinliao //不支持默认值  // }){  function order({  //一个名字两用: 既当做配对的属性名，又当做接收实参值的形参变量名  zhushi="香辣鸡腿堡",  xiaochi="薯条",  yinliao="可乐",  }){  console.log(`  您点的套餐是:  主食:${zhushi},  小吃:${xiaochi},  饮料:${yinliao}  `)  }  //第一个用户: 香辣鸡腿堡， 薯条， 可乐  order({  zhushi:"香辣鸡腿堡",  xiaochi:"薯条",  yinliao:"可乐"  })  //第二个用户着急赶火车，只想要默认的套餐  order({});  //第三个用户: 只想换主食: 奥尔良烤腿堡，小吃和饮料保持  order({ zhushi:"奥尔良烤腿堡"});  //第四个用户: 想换主食和小吃: 巨无霸和鸡块, 饮料保持默认  order({  zhushi:"巨无霸",  xiaochi:"鸡块"  });    //第五个用户: 只想换小吃为辣翅，主食和饮料保持默认！  order({  xiaochi:"辣翅"  });    //第六个用户: 只想换饮料为奶茶，主食和小吃保持默认！  order({  yinliao:"奶茶"  });    //第七个用户: 想换小吃和饮料为土豆泥和红茶，主食保持不变！  order({  xiaochi:"土豆泥",  yinliao:"红茶"  })  </script>  运行结果:  您点的套餐是:  主食:香辣鸡腿堡,  小吃:薯条,  饮料:可乐  您点的套餐是:  主食:香辣鸡腿堡,  小吃:薯条,  饮料:可乐  您点的套餐是:  主食:奥尔良烤腿堡,  小吃:薯条,  饮料:可乐    您点的套餐是:  主食:巨无霸,  小吃:鸡块,  饮料:可乐    您点的套餐是:  主食:香辣鸡腿堡,  小吃:辣翅,  饮料:可乐    您点的套餐是:  主食:香辣鸡腿堡,  小吃:薯条,  饮料:奶茶    您点的套餐是:  主食:香辣鸡腿堡,  小吃:土豆泥,  饮料:红茶 |

# 三. class

对应小程序视频: 小程序->在线->JSCORE->day08 2. class

1. 什么是: 集中保存一个类型的构造函数和原型对象方法的程序结构

2. 为什么: 旧js中，同属于一个类型的构造函数和原型对象方法虽然是一家人，但是却分着写！！！不符合封装的要求！

3. 何时: 今后只要创建一种类型，包含构造函数和原型对象方法时，都要用class包裹构造函数和原型对象方法

4. 如何定义class: 3句话

(1). 用class{}包裹构造函数和原型对象方法

(2). 类型名不应该是属于构造函数，应该属于整个class，所以要将构造函数名/类型名提升到class后，作为整个class的名称。且所有构造函数从此统一更名为"constructor"

(3). 直接定义在class下的函数，默认就会保存到原型对象中成为共有方法，且不用再加"类型名.prototype"前缀，也不用在写"=function"

class 类型名{

constructor(形参列表){

this.属性名=形参;

... = ...

}

共有方法(){

... ...

}

}

5. 如何使用class: class就是新瓶装旧酒，底层还是以前的构造函数和原型对象，所以class的用法和构造函数的用法完全一样！

var 对象名=new 类型名(属性值列表)

6. 示例: 定义一个学生类型的class，并创建一个学生对象lilei

9\_class.html

|  |
| --- |
| <script>  //定义一种学生类型，规定所有学生的统一属性结构，要求所有学生都会做自我介绍  class Student{  //回顾: 一种类型都包括: 构造函数和原型对象  //先定义构造函数:  constructor(sname, sage){  this.sname=sname;  this.sage=sage;  }  //再向原型对象中添加共有方法  intr(){  console.log(`I'm ${this.sname},I'm ${this.sage}`)  }  }  //class就是新瓶装旧酒，底层还是以前的构造函数和原型对象，所以class的用法和构造函数的用法完全一样！  //想创建一个学生对象lilei  var lilei=new Student("Li Lei",11);  lilei.intr();  console.log(lilei);  </script>  运行结果:  I'm Li Lei,I'm 11  Student {sname: "Li Lei", sage: 11}  sage: 11  sname: "Li Lei"  \_ \_proto\_ \_:  constructor: class Student  intr: ƒ intr()  \_ \_proto\_ \_: Object |

## 7. 如何设置共有属性值:

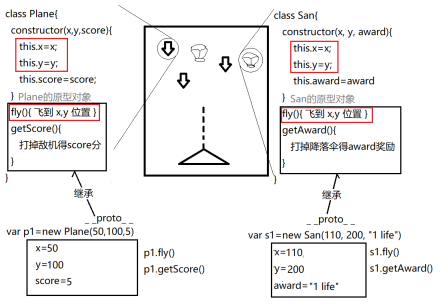
(1). 问题: 直接放在class中的属性值会自动复制到每个子对象中成为自有属性，所以class中就不应该出现属性值的定义，只包含构造函数和方法名

(2). 如何为所有子对象设置共有属性:

类型名.prototype.共有属性=属性值

## 8. 两种类型间的继承: extends

(1). 问题: 多个class中包含部分相同的属性结构和方法定义



(2). 解决: 再多定义一个父级class，集中保存两个子级class中相同部分的属性和方法。最后再让两个子级class继承父级class。

(3). 如何:

a. 先再多定义一个父级class，替两个子级class集中保存相同的属性和方法

b. 子级class中不需要包含重复的属性和方法

c. 子级class继承父级class:

class 子级 extends 父级class{

constructor(xxx){

super(xxx) //借用父级class中构造函数的意思

this.属性名=形参

}

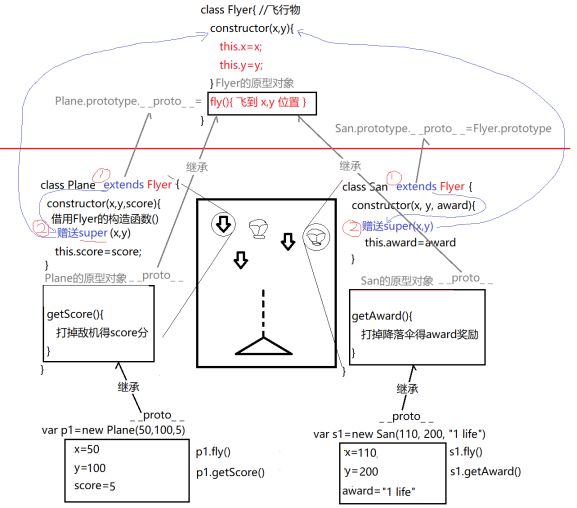
}

其中: extends简化了"子类型.prototype.\_\_proto\_\_=父类型.prototype"

并且还赠送了一个super()函数，专门代表父类型中没名的构造函数

调用super()等效于调用父级class的构造函数中的代码。

(4). 结果: 创建子级class的子对象，子对象既可使用父级class的属性和方法，又可使用爷爷级class的属性和方法。



(5). 示例: 定义飞机大战中飞机和降落伞类型，并使用继承避免重复代码

|  |
| --- |
| <script>      //先定义父类型Flyer      class Flyer{        constructor(x,y){          this.x=x;          this.y=y;        }        fly(){          console.log(`飞到x:${this.x},y:${this.y}位置`)        }      }      //再定义子类型继承父类型      class Plane extends Flyer{        constructor(x,y,score){          super(x,y);//必须写在子类型属性之前          this.score=score;        }        getScore(){          console.log(`击落敌机得${this.score}分`)        }      }      var p1=new Plane(50,100,5);      console.log(p1);      p1.fly();      p1.getScore();      //模仿Plane，自己定义一个San类型继承Flyer    </script>  运行结果:  Plane {x: 50, y: 100, score: 5}  score: 5  x: 50  y: 100  \_ \_proto\_ \_: Flyer  constructor: class Plane  getScore: ƒ getScore()  \_ \_proto\_ \_:  constructor: class Flyer  fly: ƒ fly()  \_ \_proto\_ \_: Object  飞到x:50,y:100位置  击落敌机得5分 |

**Day08**

正课:

1. Promise

# 一. Promise:

对应小程序视频: 小程序->在线->JSCORE->day08 3. 回调地狱 callback hell 和promise...

1. 什么是: 专门实现多个异步任务顺序执行的技术，专门用来避免回调地狱问题

callback hell

2. 问题: 旧js中如何使用回调函数保证多个异步任务顺序执行

(1). 错误: 单纯顺序调用多个异步函数，无法保证多个异步任务必须顺序执行。

(2). 正确: 可以用回调函数: 2步:

a. 定义前一项任务时:

function 前一项任务(小推车){

...异步任务

异步任务的最后一句话之后

自动调用 小推车()

}

b. 调用前一项任务时:

前一项任务(

function(){

下一项任务的执行语句

}

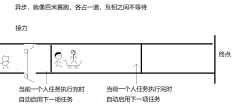
)

c. 结果:

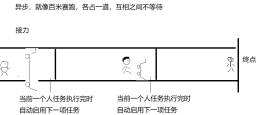
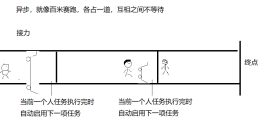
1). 前一项任务的“小推车”在开始执行时，就提前保存住了包含下一项任务的匿名函数

2). 下一项任务只有在前一项任务最后一句话执行完，自动调用“小推车”时，才能开始执行。此时已经是几s之后了！

1 2



3 4

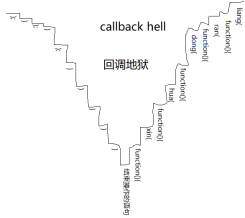
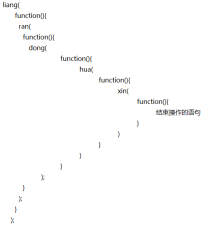


d. 示例: 使用回调函数方式，保证多个异步函数必须顺序执行

1\_callback.html

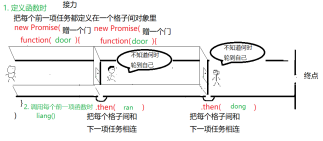
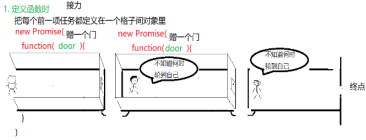
|  |
| --- |
| <script>  //亮可以跑步，但是要提前准备一个小推车，装上下一项要执行的任务  function liang(小推车){  console.log(`亮起跑...`);  setTimeout(function(){ //异步  console.log(`亮到达终点!`);  //当亮的最后一步执行完之后，自动倒出小推车中的东西开始执行  小推车();  },6000)//亮需要6s跑完全程  }  //然可以跑步  function ran(小推车){  console.log(`然起跑...`);  setTimeout(function(){ //异步  console.log(`然到达终点!`);  //当然的最后一步执行完时，然才调用小推车里的东西  小推车();  },4000)//然需要4s跑完全程  }  //东可以跑步  function dong(){  console.log(`东起跑...`);  setTimeout(function(){ //异步  console.log(`东到达终点!`)  },2000)//东需要2s跑完全程  }  //必须让亮、然、东按顺序依次起跑，跑接力  //错误  // liang();//亮起跑...  // ran();//然起跑...  // dong();//东起跑...  //正确:  liang(  //只创建一个新函数，暂不执行！  function(){//用匿名函数包裹下一步要做的事儿  ran(  function(){//只创建一个新匿名函数，包含然的下一项任务，但暂不执行  dong();  }  );//然的小推车=function(){ dong() }  }  );//亮的小推车=function(){ ran(); }  //亮内部，当亮最后一步执行完，自动调用小推车()，匿名函数才开始执行，那已经是6s后了！  //然内部，当然最后一步执行完，自动调用然的小推车()，最内层匿名函数才开始执行，已经又是4s后  //所以轮到内层函数执行时，已经等了6s+4s=10s;  // 亮 然  </script>  运行结果:  亮起跑...  亮到达终点!  然起跑...  然到达终点!  东起跑...  东到达终点! |

3. 新问题: 有时企业中需要5个异步函数顺序执行！就会产生非常深的嵌套！酷似传说中的18层地狱，所以这个新问题，这种特别难看的深层嵌套的书写格式，被称为回调地狱问题

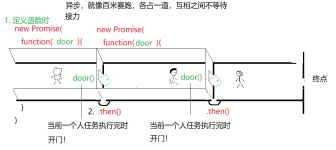
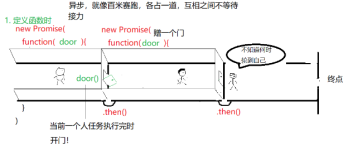


1. 新解决: Promise，2步:

1 2



3 4



(1). 定义前一项任务时:

function 前一项任务(){

//要让外界知道这个函数中有一个格子间

return new Promise( //创建一个格子间对象

//形参永远可以随意改名

function(door){//附赠了一个门door

原异步任务

原异步任务最后一步执行完，一定记得开门:

door() //只有开门，下一项任务才能开始执行

}

)

}

(2). 调用前一项任务时:

//.then()否则连接前后两项任务

前一项任务().then(下一项任务)//千万不要加()，因为不是立刻调用执行！是稍后才执行

(3). 结果: 当调用前一项任务结束后，前一项任务调用door()开门，通知下一项任务。.then()中的下一项任务才开始执行！

5. 优点: 即使有很多任务必须顺序执行，也不会出现深层嵌套的回调地狱

任务1.then(任务2).then(任务3).then(任务4).then(...)...

6. 示例: 使用Promise优化多个异步任务顺序执行

2\_promise.html

|  |
| --- |
| <script>  //亮先跑  function liang(){  //让外部知道liang中有一个格子间  return new Promise(  // 赠了一个门  function(door){  //原来的异步任务！该怎么写就怎么写  console.log(`亮起跑...`);  setTimeout(function(){ //异步  console.log(`亮到达终点!`);  //亮执行完之后，要开门！下一项任务才知道该轮到自己执行了  door();  },6000)//亮需要6s跑完全程  }  )  }  //然可以跑步  function ran(){  //让外界知道ran里有一个格子间  return new Promise(  function(door){  console.log(`然起跑...`);  setTimeout(function(){ //异步  console.log(`然到达终点!`);  //然最后一步执行完，一定记着开门！  door();  },4000)//然需要4s跑完全程  }  )  }  //东可以跑步  function dong(){  console.log(`东起跑...`);  setTimeout(function(){ //异步  console.log(`东到达终点!`)  },2000)//东需要2s跑完全程  }  // //让亮先执行，但是亮之后接着然继续执行  // liang()  // //return一个new Promise对象（格子间）  // //所有格子间都有.then()方法用来连接下一项任务或下一个格子间  // .then(ran)//千万不要加()，因为ran不是立刻调用，仅仅是和前一个任务连接起来而已  // //当亮中调用door()后.then中的ran()开始执行  // //ran结束后，又return一个new Promise对象(格子间)  // //所有格子间都可以继续.then()  // .then(dong)//千万不要加()，因为dong不是立刻调用，仅仅是和前一个任务连接起来而已  //所以:  liang().then(ran).then(dong);  </script>  运行结果:  亮起跑...  亮到达终点!  然起跑...  然到达终点!  东起跑...  东到达终点! |

# 总结:

## 1. 正则表达式:

(1). 最简单的规则就是要找的关键词正文

(2). 如果规则中一位字符上有多种备选字时: [备选字]

(3). 如果[]中部分备选字符是连续的: [x-x]

a. [a-z] 一位小写字母

b. [A-Z] 一位大写字母

c. [A-Za-z] 一位字母，不区分大小写

d. [0-9A-Za-z] 一位字母或数字，不区分大小写

e. [\u4e00-\u9fa5] 一位汉字

(4). 预定义字符集: 针对4种最常用[]的情况提供的更简化写法

a. \d 一位数字 [0-9]

b. \w 一位字母数字或\_都行 [0-9A-Za-z\_]

c. \s 一位空字符 空格、Tab、换行等

d. 通配符 .

(5). 如果规定一个字符集出现的次数，可以用量词。

a. 有明确数量边界的量词

1). {n} 必须n个，不能多也不能少

2). {n,m} 至少n个，最多m个

n<= <=m

3). {n,} 至少n个，多了不限

n<=

b. 没有明确数量边界的量词

1). \* 可有可无，多了不限 >=0

2). ? 可有可无，最多一个 0或1

3). + 至少一个，多了不限 >=1

(6). 如果两个规则，二选一匹配即可: 规则1 | 规则2

(7). 如果希望多个子规则作为一个整体先匹配，再和其它规则联合匹配: (子规则)

(8). 如果想匹配特殊位置的关键词:

a. ^规则 匹配紧挨着开头的一个关键词

b. 规则$ 匹配紧挨着结尾的一个关键词

c. \b规则 匹配紧挨着单词边界之后的一个关键词

d. 规则\b 匹配紧挨着单词边界之前的一个关键词

Day02总结:

## 2. String正则相关的函数: 3件事

### (1). 查找关键词: 4种情况

a. 查找一个固定的关键词出现的位置:

var i=str.indexOf("关键词", fromi)

b. 用正则模糊查找多种敏感词的位置:

var i=str.search(/正则/i)

c. 查找关键词的内容:

1). 如果只查找一个关键词的内容和位置：

var arr=str.match(/正则/i)

2). 如果查找所有关键词的内容，不关心位置:

var arr=str.match(/正则/ig)

d. 既查找每个关键词的内容，又查找每个关键词的位置:

reg.exec()

### (2). 替换关键词: 2种情况:

a. 简单替换: str=str.replace(/正则/ig, "新值")

b. 高级替换: str=str.replace(/正则/ig, function(keyword){

return 根据keyword不同，动态选择不同的新值返回并替换到字符串

})

c. 删除关键词: str=str.replace(/正则/ig, "")

### (3). 切割字符串: 2种情况:

a. 简单切割: var arr=str.split("切割符")

b. 复杂切割: var arr=str.split(/正则/i)

c. 固定套路: var arr=str.split("")//将字符串打散为字符数组

## 3. RegExp对象:

(1). 创建:

a. 如果正则表达式是动态拼接而成的: var reg=new RegExp("正则","ig")

b. 如果正则表达式固定不变的: var reg=/正则/ig

(2). 验证格式:

var reg=/^正则$/;

var bool=reg.test(str)

(3). 既查找每个敏感词内容，又查找每个敏感词位置:

var reg=/正则/g

do{

var arr=reg.exec(str);

...

}while(arr!=null); //只有本次找到了敏感词才有必要继续执行下一次查找

Day04总结:

## 1. Function

### (1). 创建函数: 3种:

#### a. 用声明方式: 会被声明提前

function 函数名(形参变量列表){

函数体;

return 返回值

}

#### b. 用赋值方式: 不会被声明提前

var函数名=function (形参变量列表){

函数体;

return 返回值

}

#### c. 用new:

var 函数名=new Function("形参变量1","形参变量2",...,"函数体")

揭示了js中函数的本质:

function 其实是new Function()的简写

函数其实也是一个对象

函数名只是一个普通的变量而已

### (2). 重载:

a. 何时: 一件事可能根据传入实参值的不同选择执行不同的逻辑时就要用重载

b. 如何:

1). 只定义一个函数，不要定义任何形参变量

2). 调用函数时，在函数内arguments对象会自动接住所有传入函数的实参值列表（类数组对象:有下标，有length，可遍历，但是不是数组家孩子）

3). 在函数内通过判断arguments的length属性不同或arguments中实参值不同来选择不同的逻辑执行

4). 如果想获得本次传入的实参值: arguments[下标]

### (3). 匿名函数:

a. 何时: 如果一个函数只用一次时，不会重复使用，就要用匿名函数

b. 如何: 2种情况:

1). 回调函数: 主函数(function(形参变量){ ... })

2). 匿名函数自调: (function(){ ... })()

何时: 今后所有js代码都应该放在一个匿名函数自调中——避免使用全局变量

### (4). 作用域和作用域链:

a. 作用域: js中包含两种作用域:

1). 全局作用域: window 始终在内存中，保存所有全局变量。全局变量特点：

i. 优点: 可重用

ii. 缺点: 随处可用——极易被污染

2). 函数作用域: 只在函数调用时才临时创建，函数调用后，函数作用域就释放。保存着函数内所有局部变量(2种: 形参和函数内var)。局部变量特点:

i. 优点: 仅函数内可用，不会被污染

ii. 缺点: 不可重用

b. 作用域链: 其实每个函数对象在创建时，都有一个属于自己的作用域链:

1). 保存了一个函数可用的所有作用域和变量

2). 控制了变量的使用顺序: 先函数内后全局——就近

### (5). 闭包:

a. 何时: 想重用一个变量，但是又想保护变量不被篡改——其实就是给一个函数保护一个专属的变量，仅函数自己可反复用，别人无法使用。

b. 如何: 3步:

1). 用外层函数包裹内层函数和要保护的变量

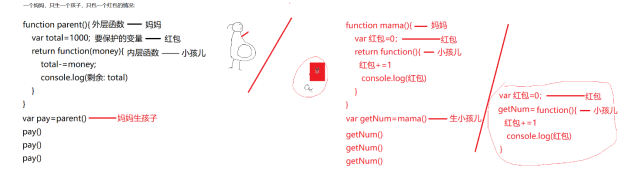
2). 外层函数将内层函数return到外层函数外部

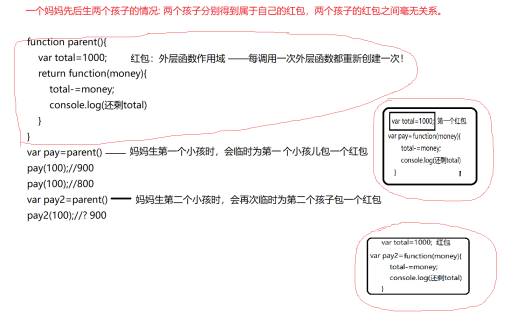
3). 外部的使用者需要调用外层函数，获得返回的内层函数对象，并将内层函数对象保存在变量中，反复使用。

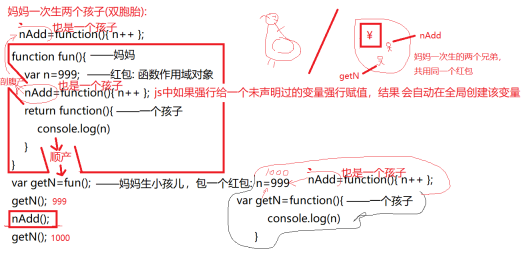
c. 闭包形成的原因: 外层函数调用后，外层函数的作用域对象被内层函数的作用域链引用着，无法释放，成为内层函数的私有财产，就形成了闭包。

d. 关于闭包的三个小故事：

对应小程序视频: 小程序->在线->JSCORE->day03 9. 关于闭包的比喻(三个小故事)







## 4. 面向对象:

### (1). 封装: 3种:

a. 用{}创建一个对象:

var 对象名={

属性名: 属性值,

... : ...

方法名: function(){

... this.属性名

}

}

b. 用new Object()：2步 揭示: js对象底层都是关联数组

1). var 对象名=new Object()

2). 对象名.新属性=新值;

对象名.方法名=function(){ ... this.属性名 ...}

c. 用构造函数反复创建多个相同结构的对象: 2步

1). function 类型名(形参变量列表){

this.属性名=形参;

... = ... ;

~~this.方法名=function(){ ... this.属性名 ... }~~

}

2). var 对象名=new 类型名(实参值列表)

### (2). 继承:

a. 何时：只要同一类型多个子对象都要使用的共有方法，应该集中定义在构造函数的原型对象中:

b. 如何: 构造函数.prototype.共有方法=function(){ ... ... }

原型对象 强行赋值

c. 结果: 凡是被这个构造函数new出来的所有子对象，都自动继承当前构造函数的原型对象。所有子对象无需重复创建方法，就可直接使用原型对象中的方法。

### (3). 多态: （重写）

a. 何时: 如果子对象觉得从父对象继承来的东西不好用！就可以重写！

b. 如何: 只要在子对象中定义和父对象中成员同名的自有属性。

c. 结果: 子对象再调用这个成员时，只使用自有的成员，不会再舍近求远使用父对象中不好用的成员了。

### (4). 自定义继承: 2种:

(1). 何时: 如果整个父对象都不是想要的，就可以整体更换为想要的新的父对象

(2). 如何: 2种:

a. 只修改一个对象的父对象: 2种:

1). 子对象.\_ \_proto\_ \_=新父对象

2). Object.setPrototypeOf(子对象, 新父对象)

b. 同时修改将来多个子对象的父对象:

1). 其实就是修改构造函数的原型对象

构造函数.prototype=新对象

2). 时机: 必须在创建子对象之前更换

### 总结: this 4种

判断this指谁，一定不要看this定义在哪儿！一定只看调用时.前是谁或有没有new.

1. obj.fun() this->.前的obj

2. new Fun() this->new本次正在创建的一个新对象

3. 原型对象.prototype.共有方法=function(){ ... } this->将来调用这个共有方法的.前的某个子对象

4. fun()或(function(){ ... })() this->默认指window ——记住，规定，没有为什么

|  |
| --- |
| new做了4件事:  1. 创建新对象  2. 继承: 让新子对象自动继承构造函数的原型对象  3. 调用构造函数，替换构造函数中的this为new新创建的子对象，通过强行赋值将构造函数中规定的属性添加到新的子对象中  4. 返回新对象的地址保存到等号左边的变量里 |

## ES5:

## 1. 严格模式: "use strict"

(1). 禁止给未声明的变量赋值

(2). 静默失败升级为错误

(3). fun() 或 (function(){ ... })() this->undefined

(4). 禁用了arguments.callee

## 2. 保护对象:

### (1). 保护对象属性: 2种:

a. 用开关保护对象的属性:

1). 每个属性有3个开关:

i. writable: true或false 控制是否可修改当前属性值

ii. enumerable: true 或false 控制是否可用for in遍历到该属性

半隐藏(浅紫色)，只防for in 不防.

iii. configurable: true或false 控制2件事:

①是否可删除该属性；②是否可修改前两个特性

一旦改为false 不可逆

2). 修改开关: 2种:

i. 只修改一个属性中的开关:

Object.defineProperty(对象,"属性名",{

开关: true或false,

configurable:false //双保险

})

ii. 同时修改多个属性中的开关:

Object.defineProperties(对象,{

属性名:{

开关: true或false,

configurable:false //双保险

},

属性名:{

开关: true或false,

configurable:false //双保险

},

... ...

})

b. 用自定义规则保护对象的属性: 访问器属性 3步:

Object.defineProperties(对象,{

1). 定义一个隐形埋名且半隐藏的普通属性实际保存属性值

\_属性名:{

value:属性值,

writable: true, //通常受保护的属性都可以修改，只不过满足条件才能改

enumerable:false, //半隐藏，不想被外人轻易发现

configurable:false //以上设置禁止随意修改

},

2). 请保镖: 冒名顶替原属性，一请就是一对儿

对外公开的属性名:{

get:function(){ //负责从受保护的\_属性中获取属性值

return this.\_属性名

},

set:function(value){ //负责将新属性值经过验证后更新到受保护的\_属性中

if(value符合要求){

this.\_属性名=value

}else{

throw Error("错误提示")

}

},

enumerable:true, //保镖要代替被保护的\_属性名抛头露面

configurable:false, //以上配置禁止随意修改

}

})

3). 使用访问器属性: 和使用普通属性是完全一样的。其它程序不知道自己用的是访问器属性——蒙在鼓里。

i. 想获取访问器属性值: 对象.访问器属性

程序会自动调用访问器属性内部的get()，返回受保护的\_属性名

ii. 想修改访问器属性值: 对象.访问器属性=新值

程序会自动调用访问器属性内部的set()，将新值自动传给value形参，如果value形参验证通过，才将新属性值value，保存到受保护的\_属性名中

### (2). 保护对象的结构: 3个级别:

a. 防扩展: Object.preventExtensions(对象)

b. 密封: Object.seal(对象)

c. 冻结: Object.freeze(对象)

3. Object.create()

(1). 何时: 即使没有构造函数，只有一个父对象，也想创建子对象继承父对象时都用Object.create()

(2). 如何:

var 子对象名=Object.create(父对象,{

自有属性名:{

value:属性值,

writable: true 或false,

enumerable:true 或 false,

},

自有属性名:{

value:属性值,

writable: true 或false,

enumerable:true 或 false,

},

... ...

})

Object.seal(子对象);

### 4. 替换this: 3种情况:

(1). 只在一次调用函数时，临时替换一次函数中的this: call 或 apply

a. 多数情况下首先call()

b. 只有在多个实参值被放在一个数组中给定时，需要先打散数组，再传参：apply()

(2). 创建一个新函数并永久绑定this为一个指定的对象: bind()

## 5. 数组新增函数:

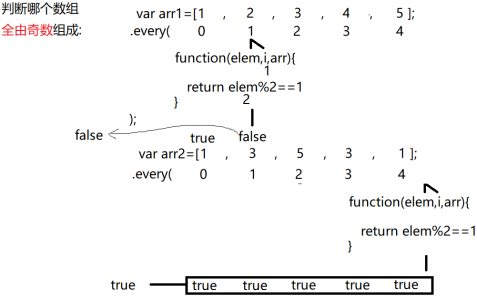
### (1). 判断:

a. 判断数组中是否所有元素都符合要求:

arr.every(function(elem,i,arr){

return 判断条件

})

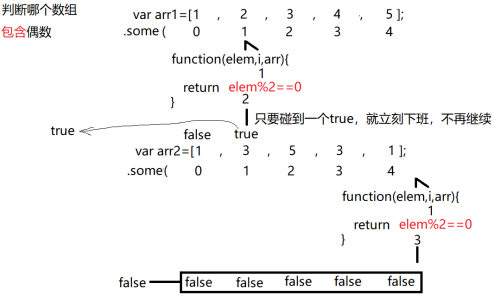


b. 判断数组中是否包含符合要求的元素:

arr.some(function(elem,i,arr){

return 判断条件

})



### (2). 遍历: 2个:

a. 如果只是单纯遍历原数组中的值，代替普通for循环: forEach

arr.forEach(function(elem,i,arr){

对当前元素值执行的操作

})

b. 如果希望保护原数组，经过修改原数组中的元素值后，返回新数组: map

var 新数组=arr.map(function(elem,i,arr){

return 当前元素值经过加工后返回的新值

})

### (3). 过滤:

var 新数组=arr.filter(function(elem, i, arr){

return 判断条件

})

### (4). 汇总:

var 汇总结果=arr.reduce(function(捐款箱, elem,i,arr){

return 捐款箱+elem

}, 捐款箱中汇总开始时的起始值)

## \*\*\*\*ES6\*\*\*\*

1. 今后，拼接字符串都用模板字符串代替+号拼接

2. 今后，凡是声明变量都用let代替var。(不会被声明提前，不会超出代码块{})

3. 今后，几乎所有function都可用=>简写

特例: 只有不希望函数内this与函数外this保持一致时，才不能用箭头函数简写

4. 今后，凡是从头到尾遍历索引数组全部元素时，且只关心元素内容不关心元素位置的遍历都可用for of简写普通for循环

5. 今后，只要希望调用者没有传入实参值时，形参也有默认值可用就用：形参=参数默认值

6. 今后，只要希望定义函数时，可收集多个不确定个数的实参值时，都用...数组名，代替arguments。

7. 今后，只要希望调用函数时，先打散数组为多个值，再分别传参给函数时，都用...数组名

8. 今后，只要希望提取出数组或对象中个别成员值放在变量中单独简化使用时，就用解构。

var [变量1, 变量2, ...]=数组

var {属性名1, 属性名2, ...}=对象

1. 今后，只要多个实参值不确定有没有，但是又要求实参值必须对应的传给指定的形参变量时，就用参数解构:

(1). 定义函数时:

function 函数名({

形参名1=默认值1,

形参名2=默认值2,

... = ...

}){

... 函数体 ...

}

(2). 调用函数时:

函数名({

形参名1: 实参值1,

形参名2: 实参值2,

... : ...

})

10. 今后，只要希望定义一种类型，包含构造函数和原型对象方法，反复创建多个相同结果的对象，都要用class包裹

class 类型名{

constructor(形参列表){

this.属性名=形参;

... = ...

}

共有方法(){

... ...

}

}

11. 今后，只要两种class中包含部分相同的属性和方法定义，都可用两种类型间的继承;

class 父类型{

constructor(形参列表){

this.相同部分的属性=形参

... = ...

}

相同部分的方法(){ ... }

}

class 子类型 extends 父类型{

constructor(形参列表){

super(形参列表);

this.子类型独有属性=形参

... = ...

}

子类型独有方法(){ ... }

}

12. 今后，只要希望一项任务必须在一个异步任务之后执行，或者多个异步任务必须顺序执行，都必须用Promise技术，2步

(1). 定义前一项任务时:

function 前一项任务(){

//要让外界知道这个函数中有一个格子间

return new Promise( //创建一个格子间对象

//形参永远可以随意改名

function(door){//附赠了一个门door

原异步任务

原异步任务最后一步执行完，一定记得开门:

door() //只有开门，下一项任务才能开始执行

}

)

}

(2). 调用前一项任务时:

//.then()否则连接前后两项任务

前一项任务().then(下一项任务)

# Dom

# 一. 什么是DOM: Document Object Model

文档 对象 模型

1. 什么是:

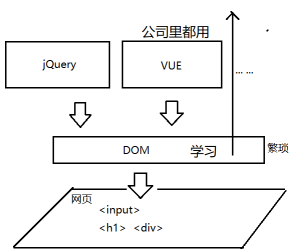
专门操作网页内容的一套对象和函数的整体，统称为DOM

1. 为什么:

因为ECMAScript，只规定了js语言的核心语法和内存原理，没有规定如何操作网页的内容。

1. 何时:

今后只要想操作网页的内容，实现页面交互效果，都要用DOM提供的对象和函数——唯一的办法。



## 4. DOM标准:

(1). 问题: 自打有浏览器那天开始，各个浏览器厂商就都有自己的一套DOM对象和函数。只不过，各个厂商之间的DOM对象和函数不兼容

(2). 解决: W3C组织出面制定了DOM对象和函数的国际标准

(3). 现在: 使用DOM标准的对象和方法来操作网页内容，几乎所有浏览器100%兼容。

## 5. DOM都能做哪些事儿: 5件事:

增删改查+事件绑定

# 二. DOM树:

对应小程序视频: 小程序->在线->DOM->day01 0 DOM Tree 树 元素 节点 标签

1. 什么是: 浏览器在内存中使用一个树型结构来保存网页中的所有内容

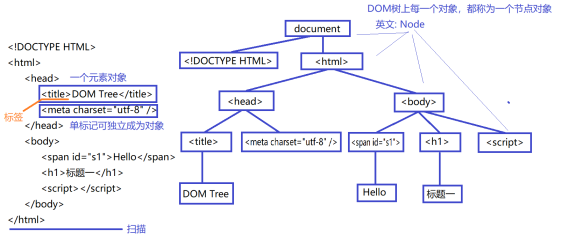
2. 为什么: 树形结构可以最直观的展现上下级包含关系。而网页中的元素及其内容，刚好也都是上下级包含关系的。

3. 何时: 只要浏览器读取到一个HTML文件，就会扫描HTML文件的内容，并将HTML文件的内容，保存到内存中一棵新创建的树结构上。

4. 如何:

(1). 只要浏览器读取到一个HTML文件，就会先在内存中创建一个唯一的树根对象document。

(2). 从上向下扫描网页的内容，每扫描到一项内容，就会在内存中创建一个对象来保存这项内容，并将对象挂在到document对象下对应位置。



# 三. 查找元素: 今后，只要想修改元素，都要先找到元素。4大类查找方式:

1. 不需要查找就可直接获得的元素对象: 4种:

(1). document 树根对象

(2). document.documentElement <html>元素对象

(3). document.head <head>元素对象

(4). document.body <body>元素对象

2. 根据节点间关系查找:

对应小程序视频: 小程序->在线->DOM->day01 1. 按节点间关系查找 节点树 元素树...

(1). 节点树: 包含网页中一切内容的完整树结构

a. 父子关系: 4种:

1). 想获得当前元素对象的父对象: 元素对象.parentNode

父 节点

2). 想获得当前元素对象下的所有直接子节点对象们: 元素对象.childNodes

孩子 节点 们

3). 想获得当前元素对象下的第一个直接子节点对象: 元素对象.firstChild

第一个孩子

4). 想获得当前元素对象下的最后一个直接子节点对象: 元素对象.lastChild

最后一个孩子

b. 兄弟关系: 2个

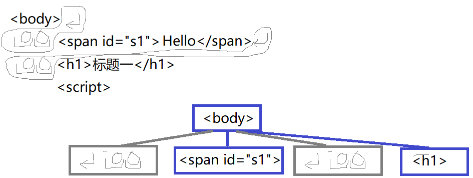
1). 获得当前元素对象的前一个兄弟节点对象: 元素对象.previousSibling

前一个 兄弟

2). 获得当前元素对象的后一个兄弟节点对象: 元素对象.nextSibling

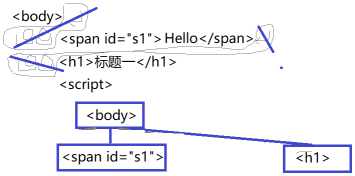
下一个兄弟

(2). 节点树的问题: 连看不见的回车和空字符都创建为DOM树上的节点，但是程序员从来不关心这些回车和空格。——严重干扰查找结果！——今后几乎不用



(3). 新树: 元素树: 仅包含元素节点对象，不包含其它文字节点 的新树结构

(4). 优点: 仅包含元素对象，所以不会受到看不见的回车或空字符干扰！



## (5). 2大类关系: （根本不用背！用到时，能查笔记找到就行！）

### a. 父子关系: 4个属性:

1). 获得当前元素的父元素对象: 元素对象.parentElement

父 元素

2). 获得当前元素的所有直接子元素对象: 元素对象.children

孩子们

3). 获得当前元素下第一个直接子元素对象: 元素对象.firstElementChild

第一个 元素 孩子

4). 获得当前元素下最后一个直接子元素对象: 元素对象.lastElementChild

最后一个元素 孩子

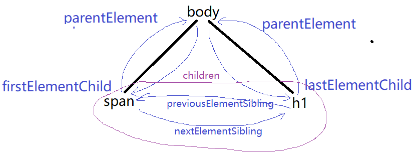
### b. 兄弟关系: 2个属性

1). 获得当前元素的前一个兄弟元素: 元素对象.previousElementSibling

前一个 元素 兄弟

2). 获得当前元素的下一个兄弟元素: 元素对象.nextElementSibling

下一个元素 兄弟



1. . 何时使用: 如果已经获得一个元素，找周围附近的元素时，才用节点间关系查找

(7). 示例: 使用节点间关系查找指定元素，并验证节点间关系

1\_domTree.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>DOM Tree</title>  <meta charset="utf-8" />  </head>  <body>  <span id="s1">Hello</span>  <h1>标题一</h1>  <script>  //前四个不需要查找，就可直接获得  console.log(document);  console.log(document.documentElement);//<html>  console.log(document.head);//<head>  console.log(document.body);//<body>    //想获得body下第一个直接子元素span  //错误: firstChild会受看不见的空字符干扰  //var span=document.body.firstChild;  //正确:  var span=document.body.firstElementChild;  console.log(span);  //想访问span的下一个兄弟元素h1  var h1=span.nextElementSibling  console.log(h1);  //h1的前一个兄弟是不是span  console.log(h1.previousElementSibling==span);//true  //访问body下所有直接子元素:  console.log(document.body.children);  //验证h1的爹是不是body  console.log(h1.parentElement==document.body);//true  </script>  </body>  </html>  运行结果: |

3. 按HTML特征查找: 4种:

对应小程序视频: 小程序->在线->DOM->day01 2. 按HTML特征查找 按id查找...

(1). 按元素的id名查找一个元素:

a. var 一个元素对象=document.getElementById("id名")

b. 在整个网页中获得一个元素按id名

c. 强调:

1). .前的主语必须是document

2). 因为只找一个元素，所以函数名中的Element是单数，没有s结尾

3). 如果网页中碰巧两个元素id相同了！永远只返回先找到的一个元素对象。

d. 返回值:

1). 如果找到符合条件的元素对象，则只返回一个元素对象

2). 如果没找到，则返回null

(2). 按元素的标签名查找多个元素:

a. var 类数组对象=任意父元素.getElementsByTagName("标签名")

b. 在指定父元素内获得多个元素按标签名

c. 强调:

1). .前的主语可以是任意父元素。查找时，只在.前的父元素下查找符合条件的子元素

2). 不仅查找直接子元素，而是在所有后代元素中查找所有符合要求的元素.

3). 因为很可能获得多个元素对象，所以函数名中Elements是复数，s结尾

4). 因为很可能获得多个元素，所以函数的返回值是一个类数组对象

5). 即使只找到一个符合条件的元素对象，也会放入一个类数组对象中返回！不会只返回一个DOM元素对象！

i. 问题: 如果想获得找到的唯一的DOM元素对象.

ii. 解决: 类数组对象[0] 取出类数组对象中保存在0位置的一个元素对象

d. 返回值:

1). 如果找到符合要求的子元素，则返回一个类数组对象包含多个找到的子元素

2). 如果没找到，返回内容为空的类数组对象 { length:0 }

(3). 按元素的class名查找多个元素:

a. var 类数组对象=任意父元素.getElementsByClassName("class名")

b. 在指定父元素下获取多个元素按class名

c. 强调:

1). .前的主语可以是任意父元素。查找时，只在.前的父元素下查找符合条件的子元素

2). 不仅查找直接子元素，而是在所有后代元素中查找所有符合要求的元素.

3). 因为很可能获得多个元素对象，所以函数名中Elements是复数，s结尾

4). 因为很可能获得多个元素，所以函数的返回值是一个类数组对象

5). 即使只找到一个符合条件的元素对象，也会放入一个类数组对象中返回！不会只返回一个DOM元素对象！如果想获得本次找到的唯一的DOM元素对象，必须在查找结果结尾加[0]

6). 如果一个元素上可能同时被多个class名修饰，那么只要使用其中任意一个class名就能找到该元素

d. 返回值:

1). 如果找到符合要求的子元素，则返回一个类数组对象包含多个找到的子元素

2). 如果没找到，返回内容为空的类数组对象 { length:0 }

如果返回的是下标位置，如果没找到，则返回-1；

(4). 按表单元素的name名查找多个表单元素:

a. var 类数组对象=document.getElementsByName("name名")

b. 在整个网页内获得多个元素按name名

c. 强调:

1). .前的主语必须是document，表示在整个页面范围查找

2). 因为很可能获得多个元素对象，所以函数名中Elements是复数，s结尾

3). 因为很可能获得多个元素，所以函数的返回值是一个类数组对象

4). 即使只找到一个符合条件的元素对象，也会放入一个类数组对象中返回！不会只返回一个DOM元素对象！如果想获得本次找到的唯一的DOM元素对象，必须在查找结果结尾加[0]

d. 返回值:

1). 如果找到符合要求的子元素，则返回一个类数组对象包含多个找到的子元素

2). 如果没找到，返回内容为空的类数组对象 { length:0 }

(5). 示例: 按不同HTML特征查找指定的元素

2\_iterator.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>遍历节点树</title>  <meta charset="utf-8"/>  </head>  <body>  <span>Hello World !</span>  <form>  用户名: <input type="text" name="uname"><br/>  性别:<label><input type="radio" name="sex" value="1">男</label>  <label><input type="radio" name="sex" value="0">女</label>  </form>  <ul id="nav">  <li class="menu-item parent">电影</li>  <li class="menu-item parent">综艺  <ul class="menu-sub">  <li class="menu-item child">跑男</li>  <li class="menu-item child">爸爸</li>  <li class="menu-item child">极限</li>  </ul>  </li>  <li class="menu-item parent">剧集</li>  </ul>  <script>  //先查找id为nav的元素对象  var ul=document.getElementById("nav");  console.log(ul);  //想在id为nav的ul下查找所有li元素  var lis=ul.getElementsByTagName("li");  console.log(lis)  //想获得id为nav的ul下第二个li下的唯一一个ul元素对象  var ul2=ul.getElementsByTagName("ul")[0];  console.log(ul2);  //只想查找ul2下的li元素  var lis=ul2.getElementsByTagName("li");  console.log(lis);  //在id为nav的ul下查找所有class为menu-item的元素  var items=ul.getElementsByClassName("menu-item");  console.log(items);  //在id为nav的ul下查找所有class为parent的元素  var parents=ul.getElementsByClassName("parent");  console.log(parents);  //在id为nav的ul下查找唯一一个class为menu-sub的ul元素对象  var ul2=ul.getElementsByClassName("menu-sub")[0];  console.log(ul2);  //想查找name名为sex的所有radio  var radios=document.getElementsByName("sex");  console.log(radios);  //只想查找唯一的name为uname的姓名文本框对象  var input=document.getElementsByName("uname")[0];  console.log(input);  </script>  </body>  </html>  运行结果: |

# 四. 事件:

1. 什么是事件: 浏览器自动触发的或用户手动触发的页面中内容或元素状态的改变

比如: 当单击一个按钮时会触发单击事件click

当鼠标进入一个元素范围内时会触发鼠标进入事件mouseover

当用户在文本框中输入内容时会触发内容改变事件change

... ...

2. 问题: 默认当单击一个按钮触发事件时，浏览器默认是什么也不会做的！但是，往往在事件发生时，我们都希望能自动执行一项任务。

3. 原因:

(1). 其实每个元素对象身上都有一批"on事件名"的特殊属性

(2). 当浏览器侦测到这个元素身上发生了某个事件时，就会找到元素身上对应的"on事件名"属性，尝试自动执行这个属性上提前保存的函数。

(3). 但是，所有on开头的属性的默认值都为null，所以默认情况下，就算触发了事件，浏览器也什么都不会做！

4. 解决: 今后，只要希望当一个元素上触发某个事件时，能自动执行一项任务，都要提前为这个元素身上对应的on开头的事件属性赋值一个函数——事件处理函数

5. 如何: 元素对象.on事件名=function(){ ... } ——绑定一个事件处理函数

6. 结果: 当这个元素身上发生了这个事件时，就会找到这个on开头的事件属性，自动调用其中提前保存的函数执行操作。

7. 问题: 事件处理函数中如何自动获得当前正在触发事件的这个元素对象？

(1). 错误: 使用事件处理函数外的全局变量。全局变量会被篡改或污染！当你用一个全局变量时，可能这个变量的值早就不是你想要的了！

(2). 正确: 今后只要在事件处理函数中想自动获得当前正在触发事件的元素对象，都要用this！因为浏览器自动执行事件处理函数时: 当前触发事件的元素对象.on事件名()

this->.前的当前正在触发事件的这一个元素对象

# 五. 综合示例: 纯客户端购物车（未完，明日待续...）

3\_shoppingCart.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>使用Selector API实现购物车客户端计算</title>  <meta charset="utf-8" />  <style>  table{width:600px; text-align:center;  border-collapse:collapse;  }  td,th{border:1px solid black}  td[colspan="3"]{text-align:right;}  </style>  </head>  <body>  <table id="data">  <thead>  <tr>  <th>商品名称</th>  <th>单价</th>  <th>数量</th>  <th>小计</th>  </tr>  </thead>  <tbody>  <tr>  <td>iPhone6</td>  <td>¥4488.00</td>  <td>  <button>-</button>  <span>1</span>  <button>+</button>  </td>  <td>¥4488.00</td>  </tr>  <tr>  <td>iPhone6 plus</td>  <td>¥5288.00</td>  <td>  <button>-</button>  <span>1</span>  <button>+</button>  </td>  <td>¥5288.00</td>  </tr>  <tr>  <td>iPad Air 2</td>  <td>¥4288.00</td>  <td>  <button>-</button>  <span>1</span>  <button>+</button>  </td>  <td>¥4288.00</td>  </tr>  </tbody>  <tfoot>  <tr>  <td colspan="3">Total: </td>  <td>¥14064.00</td>  </tr>  </tfoot>  </table>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: 用户点击table中每个button,页面会发生变化,所以，应该先查找table，然后再查找table下所有button元素  //先找id为data的table  var table=document.getElementById("data");  console.log(table);  //再在table下查找button元素  var btns=table.getElementsByTagName("button");  console.log(btns);  //2. 绑定事件处理函数  //本例中: 因为6个按钮都可单击，所以，应该给每个按钮都绑定一个单击事件处理函数  //遍历类数组对象btns中每个按钮元素对象  for(var btn of btns){  //每遍历一个按钮，就给当前按钮绑定单击事件的处理函数  btn.onclick=function(){  //alert("疼!");  //希望点哪个按钮，就让当前按钮的内容变成❀  //错误: 循环结束后，全局变量btn已经停留在最后一个按钮上了，今后执行这个函数，所有❀，都会给最后一个按钮，而不是当前按钮  //btn.innerHTML="❀";  //正确:  this.innerHTML="❀"; //this->当前正在触发事件的这个元素对象  }  }    //3. 查找要修改的元素  //4. 修改元素  </script>  </body>  </html>  运行结果: |

# Day02

正课:

1. 查找元素

2. 修改元素

3. 添加/删除元素

# 一. 查找元素: 4种:

1. 不需要查找就可直接获得的元素: 4个

2. 按节点间关系查找: 2大类关系，6个属性

3. 按HTML特征查找: 4个函数

4. 按选择器查找:

(1). 问题: 如果一个元素藏的很深，查找条件比较复杂，则使用按节点间关系查找或按HTML特征查找，步骤都会很繁琐！

(2). 解决: 因为CSS选择器在查找复杂元素时的语法非常简洁，所以DOM也支持按选择器查找元素

(3). 如何: 2个函数

a. 如果只查找一个符合条件的元素：

var 一个元素对象=任意父元素.querySelector("任意复杂的选择器")

b. 如果查找多个符合条件的元素:

var 类数组对象=任意父元素. ("任意复杂的选择器")

5. 示例：纯客户端版购物车

对应小程序视频: 小程序->在线->DOM->day01 3. 购物车 shoppingcart

day01 购物车/3\_shoppingCart.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>使用Selector API实现购物车客户端计算</title>  <meta charset="utf-8" />  <style>  table{width:600px; text-align:center;  border-collapse:collapse;  }  td,th{border:1px solid black}  td[colspan="3"]{text-align:right;}  /\*想让table下tfoot下最后一个td背景变为粉色\*/  table>tfoot td:last-child{  background-color:pink;  }  /\*想让table下tbody下每行最后一个td背景变为黄色\*/  table>tbody td:last-child{  background-color:yellow;  }  </style>  </head>  <body>  <table id="data">  <thead>  <tr>  <th>商品名称</th>  <th>单价</th>  <th>数量</th>  <th>小计</th>  </tr>  </thead>  <tbody>  <tr>  <td>iPhone6</td>  <td>¥4488.00</td>  <td>  <button>-</button>  <span>1</span>  <button>+</button>  </td>  <td>¥4488.00</td>  </tr>  <tr>  <td>iPhone6 plus</td>  <td>¥5288.00</td>  <td>  <button>-</button>  <span>1</span>  <button>+</button>  </td>  <td>¥5288.00</td>  </tr>  <tr>  <td>iPad Air 2</td>  <td>¥4288.00</td>  <td>  <button>-</button>  <span>1</span>  <button>+</button>  </td>  <td>¥4288.00</td>  </tr>  </tbody>  <tfoot>  <tr>  <td colspan="3">Total: </td>  <td>¥14064.00</td>  </tr>  </tfoot>  </table>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: 用户点击table中每个button,页面会发生变化,所以，应该先查找table，然后再查找table下所有button元素  //先找id为data的table  var table=document.getElementById("data");  console.log(table);  //再在table下查找button元素  var btns=table.getElementsByTagName("button");  console.log(btns);  //2. 绑定事件处理函数  //本例中: 因为6个按钮都可单击，所以，应该给每个按钮都绑定一个单击事件处理函数  //遍历类数组对象btns中每个按钮元素对象  for(var btn of btns){  //每遍历一个按钮，就给当前按钮绑定单击事件的处理函数  btn.onclick=function(){  //alert("疼!");  //希望点哪个按钮，就让当前按钮的内容变成❀  //正确:  //this.innerHTML="❀"; //this->当前正在触发事件的这个元素对象  /\*功能一. 实现点击按钮，修改数量\*/  //3. 查找要修改的元素  //本例中: 点击按钮，要修改当前按钮旁边的span的内容，所以，这里应该查找当前按钮旁边的span元素对象    //可以先查找当前按钮的爹td，然后查找td下所有直接子元素中第二个元素  var span=this.parentElement.children[1];  //console.log(span);  //4. 修改元素  //本例中: 如果当前按钮的内容是+,就对span的内容+1，否则如果当前按钮的内容是-，就对span的内容-1，但是购物车中数量不能小于1，到1就不能再减了。  //要想修改一个元素的内容，必须先获得元素的内容，修改后，再放回去  //4.1 先获得span的内容，因为之后要做算术计算，保险起见转为数字  //\*\*\*凡是从页面获得的一切，都是字符串类型。算术计算之前都要转为数字  var n=parseInt(span.innerHTML);//发生了按值传递,n只是span内容的副本  //4.2 根据点的按钮不同，修改n的值  //如果当前按钮的内容是+  if(this.innerHTML=="+"){  n++;//就n+1  }else if(n>1){//否则如果当前按钮的内容是-，且n还>1时  n--;//才能n-1  }//在程序中修改n，是不影响页面显示的  //4.3 要想将修改后的新值显示在页面上，必须将修改后的n再放回元素内容中  span.innerHTML=n;  /\*功能二. 数量修改同时，修改小计\*/  //3. 查找要修改的元素  //本例中: 数量修改，同时要修改当前行最后一个td中的小计内容  //查找当前按钮的爹的下一个兄弟  var lastTd=this.parentElement.nextElementSibling;  //4. 修改元素  //本例中: 先用单价\*数量n，计算出小计，再将小计放入lastTd的内容中    //4.1. 获得当前按钮的爹的前一个兄弟元素的内容中的单价  //特殊加工: 还要截取1位置到结尾的子字符串，可以去掉开头的¥  // 因为钱可能有小数，所以用parseFloat将截取后的字符串，转为数字  var price=parseFloat(  this.parentElement //td  .previousElementSibling //前一个td  .innerHTML //"¥4488.00"  .slice(1) //"4488.00"  );//4488  //console.log(price);  //4.2 计算小计=单价\*数量n  var subtotal=price\*n;  //4.3 将计算后的小计放入本行最后一个td的内容中，前加¥，后保留两位小数  lastTd.innerHTML=`¥${subtotal.toFixed(2)}`;  /\*功能三. 计算总价\*/  //3. 查找要修改的元素  //本例中: 总价应该修改到table下tfoot下最后一个td的内容中，所以应该查找table下tfoot下最后一个td    //var lastTd=table.querySelector("table>tfoot td:last-child");  // 已经在table下查找了 选择器只要从table下开始写即可  var lastTd=table.querySelector("tfoot td:last-child");  //console.log(lastTd);  //4. 修改元素  //本例中: 应该先获得tbody中每行最后一个td中的小计内容，去¥转为小数后，累加起来计算出总计。最后将总计放入tfoot中最后一个td中  //4.1 先查找table下tbody中每行最后一个td  var tds=table.querySelectorAll("tbody td:last-child");  //console.log(tds);  var total=0;//声明变量准备累加所有小计  //4.2 遍历tds中每个td  for(var td of tds){  //每遍历一个td，就获取当前td的内容，去¥转为小数后，累加到变量total中  total+=parseFloat(td.innerHTML.slice(1));  }  //4.3 将计算好的total总价，放入tfoot最后一个td中  lastTd.innerHTML=`¥${total.toFixed(2)}`;  }  }  </script>  </body>  </html>  运行结果: |

# 修改:

当我们找到一个元素后，可以改元素的三样东西:

1. 获取或修改元素的内容: 3种

对应小程序视频: 小程序->在线->DOM->day02 0.1 修改元素的内容...

(1). 获取或修改元素开始标签到结束标签之间的原始的HTML代码内容:

元素.innerHTML=`新HTML片段`

内部 html

(2). 获取或修改元素开始标签到结束标签之间的纯文本内容:

元素.textContent=`纯文本内容`

文本 内容

|  |
| --- |
| .innerHTML vs .textConent  1. 获取元素内容时:  .innerHTML: 获取的是原始的HTML代码内容  .textContent: 多做了两件事: 1. 去掉内嵌标签，2. 特殊符号翻译为正文  2. 修改元素内容时:  .innerHTML:会将新的内容先交给浏览器解析，将解析后的结果给人看  .textContent: 不会将新内容交给浏览器解析，努力保持内容的原样输入给人看 |

(3). 获取或修改表单元素的内容:

a. 问题: 大多数input表单元素是单标记！比如: <input/>，没有innerHTML或textContent

b. 解决: 今后只要想获取或修改表单元素的内容都要用：表单元素.value=新值

(4). 示例: 开关门效果

对应小程序视频: 小程序->在线->DOM->day02 1 动画 开关门效果

0\_door.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>读取并修改元素的内容</title>  <meta charset="utf-8" />  <style>  div{float:left; height: 100px; line-height: 100px; }  #d1,#d3{ background-color: #ccff00; }  #d2{ cursor: pointer; background-color: #ffcc00; }  </style>  </head>  <body>  <div id="d1">树形列表</div>  <div id="d2">&lt;&lt;</div>  <div id="d3">内容的主体</div>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: 用户点d2，触发开关门  var d2=document.getElementById("d2")  //2. 绑定事件处理函数  //本例中: 用户点击d2，触发开关门  d2.onclick=function(){  //3. 查找要修改的元素  //本例中: 点d2，要来回切换d1的显示和隐藏  var d1=document.getElementById("d1");  //4. 修改元素  //本例中: 第一次点d2，隐藏d1，第二次再点d2，则显示d1  //如果d1的style的display不是none  if(d1.style.display!=="none"){  d1.style.display="none";//就变成none  //当d1关上时，就修改当前点击的d2的内容为>>  this.innerHTML="&gt;&gt;";  }else{//否则如果d1的style的display是none  d1.style.display="";//就清除none，元素默认都是显示的  //当d1打开时，就修改当前点击的d2的内容为<<  this.innerHTML="&lt;&lt;";  }  }  </script>  </body>  </html>  运行结果: |

2. 属性: 3种:

(1). 字符串类型的HTML标准属性

对应小程序视频: 小程序->在线->DOM->day02 0.2 修改元素属性 字符串类型...

a. 什么是: HTML标准中规定的，值为字符串类型的HTML元素属性

b. 比如: id="xxx", name="xxx", title="xxx", href="xxx", src="xxx" ...

c. 如何: 2种:

1). 旧DOM标准提供的核心DOM4个函数:

i. 获得一个元素上一个属性的属性值: var 属性值=元素.getAttribute("属性名")

ii. 修改一个元素上一个属性的属性值: 元素.setAttribute("属性名","属性值")

iii. 判断一个元素上是否包含一个属性: var bool=元素.hasAttribute("属性名")

iv. 移除一个元素上的一个属性: 元素.removeAttribute("属性名")

v. 示例: 使用核心DOM 4个函数操作元素的字符串类型HTML标准属性

1\_attribute.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <a id="a1" href="http://tmooc.cn" target="\_blank">go to tmooc</a>  <script>  var a1=document.getElementById("a1");  //想获得a1的href属性值  var href=a1.getAttribute("href");  console.log(href); //"http://tmooc.cn"  //想修改a1的title属性值  a1.setAttribute("title","欢迎访问tmooc");  //想判断a1是否包含target属性  var bool=a1.hasAttribute("target");  console.log(bool);//true  //移除a1的target属性  a1.removeAttribute("target");  //想判断a1是否包含target属性  var bool=a1.hasAttribute("target");  console.log(bool);//false  </script>  </body>  </html>  运行结果: |

2). 新HTML DOM标准提供的简写: 元素.属性名

i. 因为浏览器在扫描元素时，就已经将元素所有HTML规定的属性都提前定义在了元素对象身上。所以，所有的HTML标准属性，都可以用"元素.属性名"方式来访问

ii. 比如:

①获取一个元素上一个属性的属性值: 元素.属性名

②修改一个元素上一个属性的属性值: 元素.属性名="属性值"

③判断一个元素上是否包含一个属性: 元素.属性名!==""

④移除一个元素上的一个属性: 元素.属性名=""

iii. 示例: 使用.属性名简写上一个例子，操作元素的属性  
1\_property.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <a id="a1" href="http://tmooc.cn" target="\_blank">go to tmooc</a>  <script>  var a1=document.getElementById("a1");  console.dir(a1);  //想获得a1的href属性  console.log(a1.href);  //设置a1的title属性  a1.title="欢迎访问tmooc"  //判断a1是否设置target属性  console.log(a1.target!=="");//true  //移除a1的target属性  a1.target="";//清空target属性值，失效了！  //判断a1是否设置target属性  console.log(a1.target!=="");//false  </script>  </body>  </html>  运行结果: |

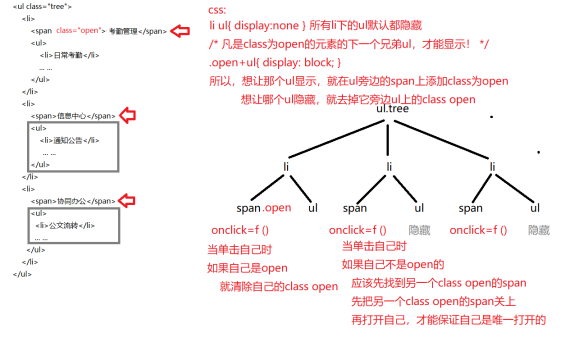
iv. 特例: class属性

①问题: 因为ECMAScript中已经将class定义为js语言的关键字，专门用于创建一种新类型。所以DOM标准中就不能再使用class作为属性名

②解决: 今后只要用.操作元素的class属性，都要被迫更名为className。操作.className，就等效于操作元素的class属性值。

v. 示例: 下拉菜单 手风琴效果

对应小程序视频: 小程序->在线->DOM->day02 2. 下拉菜单 手风琴效果...



1\_menu.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>1. 实现伸缩二级菜单</title>  <meta charset="utf-8" />  <style>  /\* 清空li前的小圆点 \*/  li{  list-style:none;  }  /\* 定义一级菜单span的默认样式 \*/  li span{  padding-left: 20px;  cursor: pointer;  background: url("../images/add.png") no-repeat center left;  }  /\* 所有li下的ul，默认都是隐藏的！ \*/  li ul{  display: none;  }  /\* 仅将span前的+号背景变成-号背景 \*/  .open{  background: url("../images/minus.png") no-repeat center left;  }  /\* 凡是class为open的元素的下一个兄弟ul，才能显示！ \*/  .open+ul{  display: block;  }  </style>  </head>  <body>  <ul class="tree">  <li>  <span class="open">考勤管理</span>  <ul>  <li>日常考勤</li>  <li>请假申请</li>  <li>加班/出差</li>  </ul>  </li>  <li>  <span>信息中心</span>  <ul>  <li>通知公告</li>  <li>公司新闻</li>  <li>规章制度</li>  </ul>  </li>  <li>  <span>协同办公</span>  <ul>  <li>公文流转</li>  <li>文件中心</li>  <li>内部邮件</li>  <li>即时通信</li>  <li>短信提醒</li>  </ul>  </li>  </ul>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: 点击每个li中的span触发切换不同的菜单显示  //查找class为tree的ul下的所有span元素  var spans=document.querySelectorAll("ul.tree span");  //2. 绑定事件处理函数  //本例中: 因为每个span都可单击  //遍历spans中每个span，绑定单击事件  for(var span of spans){  span.onclick=function(){  //this->当前正在点击的这一个span元素  //测试: 点哪个span，哪个span就变成❀  //this.innerHTML="❀";  //3. 查找要修改的元素  //本例中: 不用查找，因为点一个span时，只要修改自己的class就行了！  //4. 修改元素  //本例中: 当单击一个span时，当前span应该在开关之间来回切换  //如果当前span的class是open，说明当前span旁边的ul是开着的，那么我们只要清除span的class open，就把当前span旁边的ul关上了  if(this.className=="open"){  this.className="";  }else{//否则如果当前span的class不是open，说明当前span旁边的ul是隐藏的。那么只要修改当前span的class为open，当前span旁边的ul就打开了！  //打开自己之前，为了防止别的span也开着，应该尝试去查找另一个开着的span。（根据需求，一次只可能有一个是开着的，所以只要找一个就可以了！不用找全部！）  var openSpan=document.querySelector("ul.tree span.open");  //如果找到另一个开着的span，应该先把另一个开着的span关上  if(openSpan!=null){  openSpan.className="";  }  //然后再打开自己  this.className="open";  //结果: 肯定这一次只有当前span自己是打开的！别的span都是关上的  }  }  }    </script>  </body>  </html>  运行结果: |

(2). bool类型的HTML标准属性

对应小程序视频: 小程序->在线->DOM->day02 0.3 修改元素的属性 bool类型...

a. 什么是: HTML标准中规定的，不写属性值也能直接发挥作用的特殊属性

b. 比如: disabled 禁用 checked 选中 selected 选中

c. 如何: 1种

~~1). 不能用旧的核心DOM4个函数操作。因为那四个函数只支持字符串类型的HTML标准属性。不支持bool值！~~

2). 只能用新HTML DOM 打.访问。且属性值必须是bool类型

d. CSS: 状态伪类:

1). 什么是: 专门根据元素的状态来匹配元素

2). 包括:

i. :disabled 专门匹配禁用的元素

ii. :checked 专门匹配选中的元素checkbox或radio

iii. :selected 专门匹配选中的元素select下的option

e. 示例:

对应小程序视频: 小程序->在线->DOM->day02 3. 全选 取消全选 selectAll

2\_selectAll.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8" />  <title>全选和取消全选</title>  </head>  <body>  <h2>管理员列表</h2>  <table border="1px" width="500px">  <thead>  <tr>  <th><input type="checkbox"/>全选</th>  <th>管理员ID</th>  <th>姓名</th>  <th>操作</th>  </tr>  </thead>  <tbody>  <tr>  <td><input type="checkbox"/></td>  <td>1</td>  <td>Tester</td>  <td>修改 删除</td>  </tr>  <tr>  <td><input type="checkbox"/></td>  <td>2</td>  <td>Manager</td>  <td>修改 删除</td>  </tr>  <tr>  <td><input type="checkbox"/></td>  <td>3</td>  <td>Analyst</td>  <td>修改 删除</td>  </tr>  <tr>  <td><input type="checkbox"/></td>  <td>4</td>  <td>Admin</td>  <td>修改 删除</td>  </tr>  </tbody>  </table>  <script>  /\* 功能一. 点全选，控制下方所有checkbox的选中状态 \*/  //DOM 4步  //1. 查找触发事件的元素  //本例中: 用户点thead下的input，触发了变化  var chbAll=document.querySelector("table>thead input");  //2. 绑定事件处理函数  //本例中: 用户单击input时，触发变化  chbAll.onclick=function(){  //3. 查找要修改的元素  //本例中: 点全选，要改tbody中所有input  var chbs=document.querySelectorAll("table>tbody input");  //4. 修改元素  //本例中: 修改tbody中所有input的checked属性值和当前点击的全选input的checked属性保持一致  for(var chb of chbs){  chb.checked=this.checked;  }  }  /\* 功能二. 点击tbody中每个input，也有可能影响thead中input的状态 \*/  //DOM 4步  //1. 查找触发事件的元素  //本例中: 用户点tbody中每个input都可能触发变化  var chbs=document.querySelectorAll("tbody input");  //2. 绑定事件处理函数  //本例中: tbody下每个input都可以点击  for(var chb of chbs){  chb.onclick=function(){  //3. 查找要修改的元素  //本例中: 点tbody中每个input，都可能影响thead中input  var chbAll=document.querySelector("thead input");  //4. 修改元素  //本例中: 如果tbody中所有input都选中，则上边的全选就选中，否则如果tbody中有未选中的input，则上边的全选就不选中  //4.1 尝试查找tbody下所有input中那个未选中的input  var unchecked=document.querySelector("tbody input:not(:checked)")  //4.2 如果找到一个未选中的input，则上方的全选就不选中  if(unchecked!=null){  chbAll.checked=false;  }else{//4.3 否则如果未找到未选中的input，则上方的全选就选中  chbAll.checked=true;  }  }  }  </script>  </body>  </html>  运行结果: |

(3). 自定义扩展属性

对应小程序视频: 小程序->在线->DOM->day02 4.1 自定义扩展属性 何时使用...

a. 什么是: HTML标准中没有规定的，程序员根据自己的需要自发添加的一些自定义属性

b. 何时: 2种:

1). 在HTML元素上临时缓存一些业务相关的数据：

比如: <ul>

<li score="89">亮亮</li>

<li score="61">然然</li>

<li score="91">东东</li>

2). 可以代替id、元素、class选择器作为查找触发事件的元素的条件

i. id、元素、class选择器为什么不适合作为查找触发事件的元素的条件

①id选择器: 一次只能选择一个元素

②元素选择器: 人家在界面上实现一种效果，不一定只针对一种元素

比如: 想实现一个按钮，可能使用: a button input span ...

③class选择器: class的本职工作为元素添加css样式的。不是为js服务的。维护时，一旦别人发现一个和样式无关的class，很可能就会被清除掉！

ii. 解决: 可以添加自定义属性，来专门为js服务

比如: <button data-click="btn">

var btn=document.querySelector("[data-click=btn]")

像bootstrap学习，所有触发事件的元素上，都应该用自定义属性作为条件！

c. 如何: //HTML5标准

1). 为元素添加/修改自定义属性:

i. HTML中: <元素 data-自定义属性名="属性值">

ii. js中:

~~①不能用".自定义属性名"访问，因为所有自定义属性都是后天添加的，HTML标准中没有，自然也就不包含在内存中元素对象身上！~~

②旧HTML标准中，只能用核心DOM 4个函数中的:

元素.setAttribute("data-自定义属性名","属性值")

③HTML5中: 只要加了data-前缀的自定义属性，还可以用dataset属性来操作

元素.dataset.自定义属性名=新值

2). 获取自定义属性值:

i. 旧DOM中: ~~不能用.~~ 只能用getAttribute: 元素.getAttribute("data-自定义属性名")

ii. HTML5中: 元素.dataset.自定义属性名

3). 查找带有自定义扩展属性的元素: CSS属性选择器

var 元素/类数组对象=document.querySelectorAll("[data-自定义属性名=属性值]")

d. 示例: 使用HTML5方式操作自定义扩展属性3\_data-.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <a href="javascript:;" data-click="btn" data-count="5">click me</a>  <script>  //用属性选择器查找到带有data-click属性且属性值为btn上按钮元素  var btn=document.querySelector("[data-click=btn]");  console.log(btn);  //想将btn上count属性值+1  var count=parseInt(  btn.dataset.count  //btn.getAttribute("data-count")  );  count++;  //btn.setAttribute("data-count",count);  btn.dataset.count=count;  </script>  </body>  </html>  运行结果: |

3. 样式:

(1). 修改内联样式:

a. HTML中: <元素 style="css属性: 属性值">

b. DOM中: 元素对象.style.css属性="属性值" 下边这句话会被自动翻译为上边那句话

## classList

add( String [, String] )

Dom.classList.add(classname)  
添加指定的类值。如果这些类已经存在于元素的属性中，那么它们将被忽略。

remove( String [,String] )

Dom.classList.remove(classname)  
删除指定的类值。

item ( Number )

Dom.classList.item (classname)  
按集合中的索引返回类值。

toggle ( String [, force] )

Dom.classList.toggle(classname)  
当只有一个参数时：切换 class value; 即如果类存在，则删除它并返回false，如果不存在，则添加它并返回true。   
当存在第二个参数时：如果第二个参数的计算结果为true，则添加指定的类值，如果计算结果为false，则删除它   
contains( String )   
检查元素的类属性中是否存在指定的类值。   
replace( oldClass, newClass )   
用一个新类替换已有类。

# Day03

正课:

1. 修改元素

2. 添加/删除元素

3. HTML DOM常用对象

4. BOM

# 一. 修改元素: 3种

1. 内容: 3种

2. 属性: 3种:

(1). 字符串类型的HTML标准属性: 2种:

(2). bool类型的HTML标准属性: 1种:

(3). 自定义扩展属性: 2种方式

a. 旧js: 元素.getAttribute("data-自定义属性名")

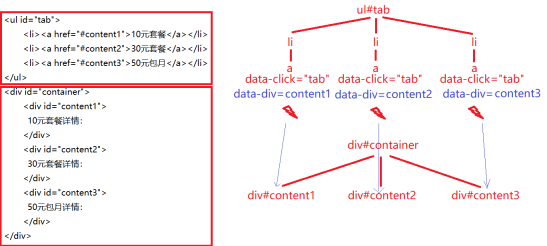
.setAttribute("data-自定义属性名","属性值")

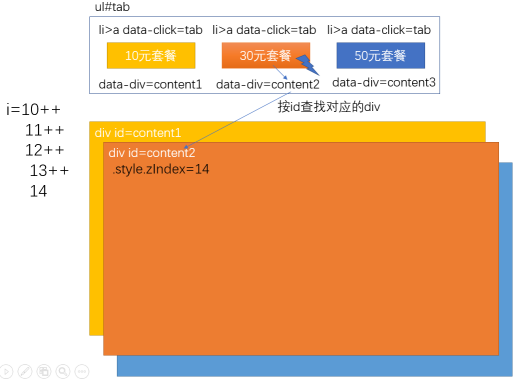
b. HTML5中: 元素.dataset.自定义属性名

c. 查找带有自定义扩展属性的元素: CSS 属性选择器: [data-自定义属性名=属性值]

(4). 示例: 标签页:

对应小程序视频: 小程序->在线->DOM->day02 4.2 标签页 tabs





day02 tabs/3\_tabs.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>读取并修改元素的属性</title>  <meta charset="utf-8" />  <style>  \*{  margin:0;  padding: 0;  }  #tab li{  float: left; list-style: none;  }  #tab li a{  display:inline-block;  text-decoration:none;  width: 80px; height: 40px;  line-height: 40px;  text-align: center;  color:#000;  }  #container{  position: relative;  }  #content1,#content2,#content3{  width: 300px;  height: 100px;  padding:30px;  position: absolute;  top: 40px;  left: 0;  }  #tab li:first-child,#content1{  background-color: #ffcc00;  }  #tab li:first-child+li,#content2{  background-color: #ff00cc;  }  #tab li:first-child+li+li,#content3{  background-color: #00ccff;  }  </style>  </head>  <body>  <h2>实现多标签页效果</h2>  <div class="tabs">  <ul id="tab">  <li><a data-click="tab" data-div="content1" href="#content1">10元套餐</a></li>  <li><a data-click="tab" data-div="content2" href="#content2">30元套餐</a></li>  <li><a data-click="tab" data-div="content3" href="#content3">50元包月</a></li>  </ul>  <div id="container">  <div id="content1">  10元套餐详情：<br />&nbsp;每月套餐内拨打100分钟，超出部分2毛/分钟  </div>  <div id="content2">  30元套餐详情：<br />&nbsp;每月套餐内拨打300分钟，超出部分1.5毛/分钟  </div>  <div id="content3">  50元包月详情：<br />&nbsp;每月无限量随心打  </div>  </div>  </div>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: 用户点标签中每个按钮触发变化  //但是，实现按钮效果，不一定只能用a，还可能换成span、button等  //所以，应该为每个按钮添加自定义扩展属性，来摆脱元素名的影响。  //比如: 为每个按钮上都添加一个自定义属性: data-click="tab"  // 可以改名 可以改名  //所以，应该查找所有带有data-click属性，且属性值为tab的元素  var tabs=document.querySelectorAll("[data-click=tab]")  //2. 绑定事件处理函数  var i=10; //提前定义一个变量，累加zIndex数值  //本例中: 每个标签页按钮都可以单击  for(var tab of tabs){  tab.onclick=function(){  //3. 查找要修改的元素  //本例中: 每点一个标签页按钮，要修改当前按钮对应的div  //问题: 如何让每个按钮自己记住自己对应的哪个div呢？  //解决: 3.1为每个按钮添加自定义扩展属性: data-div=对应div的id  //3.2当单击一个按钮时，就可以获得这个按钮上保存的自己对应的div的id  var id=//this.getAttribute("data-div");//旧js  this.dataset.div;//HTML5  //3.3用id查找本次点击的按钮对应的div  var div=document.getElementById(id);//不要加"",因为id是变量  //4. 修改元素  //本例中: 设置当前div的z-index属性最大！  //问题1: js中不能写z-index，-会和减法的-号冲突  //解决: z-index改为zIndex，去横线变驼峰  //问题2: 怎么保证本次这是的zIndex值最大？比之前的zIndex值都大！  //解决: 提前定义一个变量，保存一个数字，每使用一次，数字就+1，就可以保证最后一次使用的数字，一定比之前使用过的数字都大  i++;//先将i++  //再设置当前div的zIndex为新的i  div.style.zIndex=i;  }  }  </script>  </body>  </html>  运行结果: |

3. 样式:

对应小程序视频: 小程序->在线->DOM->day02 5 修改样式，计算属性

(1). 修改内联样式:

a. 元素.style.css属性="属性值"

b. 会被自动翻译为HTML中: <元素 style="css属性:属性值">

c. 问题: 有些css属性名中带-，但是js中不能随便写-，会和减法的-号冲突

d. 解决: 凡是带-的css属性名，在js中都要"去横线变驼峰"

比如: z-index -> zIndex

background-color -> backgroundColor

font-size -> fontSize

(2). 获取css属性值:

a. 坑: 因为.style属性，仅代表内联样式，所以获取时，也只能获取内联样式。但是实际开发中很少使用内联样式！尴尬: 明明知道这个元素上有很多css属性，但是用.style就是无法获得css属性值！

b. 解决: 今后如果想获得元素的所有css属性，应该使用计算后的样式

1). 什么是计算后的样式: 最终应用到这个元素上的所有css属性的总和

2). 如何: 2步

i. 先获得包含计算后的所有样式的对象:

var style=getComputedStyle(元素对象)

获得 计算后的 样式

可直接使用 不可改名

ii. 再用.方式从style中提取出想用的个别css属性值

style.css属性

c. 强调: 所有计算后的样式属性值都是只读的！不允许修改。因为计算后的样式属性来源不确定。很可能来自于外部样式表，同时被多个元素使用着。如果因为某一个元素擅自修改共用的css属性，会牵一发而动全身！——程序中最不希望看到的！

(3). 总结:

a. 今后只要获取一个元素的css属性值，都用getComputedStyle()

b. 今后只要修改一个元素的css属性值，都用.style

优点: 仅影响自己，不影响别人; 优先级最高，不会被其它属性覆盖！

(4). 示例: 使用getComputedStyle()获取元素的所有css属性值

0\_style\_getComputedStyle.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <style>  h1{  background-color:red;  }  </style>  </head>  <body>  <h1 id="h1" style="color:yellow">Welcome</h1>  <p>welcome to my web site</p>  <script>  var h1=document.getElementById("h1");  //想获得h1的字体颜色，背景颜色和字体大小  //错误：用h1.style方式  // console.log(h1.style.color);//内联  // console.log(h1.style.backgroundColor);//内部样式表  // console.log(h1.style.fontSize);//浏览器默认样式表  //正确: getComputedStyle(h1)  var style=getComputedStyle(h1);  console.log(style);//包含所有css属性,太大  console.log(style.color);//内联  console.log(style.backgroundColor);//内部样式表  console.log(style.fontSize);//浏览器默认样式表  //想通过计算后的样式对象style，修改某个css属性  style.fontSize="90px";//报错！  //Failed to set the 'font-size' property on 'CSSStyleDeclaration':  // 不能 设置 font-size 属性 在 xxx对象中  //These styles are computed, and therefore the 'font-size' property is read-only.  //这些 样式 是 计算后的 因此 font-size 属性  //是 只读的  </script>  </body>  </html>  运行结果:  CSSStyleDeclaration {0: "animation-delay", 1: "animation-direction", 2: "animation-duration", 3: "animation-fill-mode", 4: "animation-iteration-count", 5: "animation-name", …}  rgb(255, 255, 0)  rgb(255, 0, 0)  32px  Uncaught DOMException: Failed to set the 'font-size' property on 'CSSStyleDeclaration': These styles are computed, and therefore the 'font-size' property is read-only.  at http://127.0.0.1:5500/3\_style\_getComputedStyle.html:30:21 |

(5). 问题: .style.css属性，一句话只能修改一个css属性。而网页中很多效果，需要同时修改一个元素的多个css属性才能完整。用.stye.css属性做效果，代码会很繁琐！



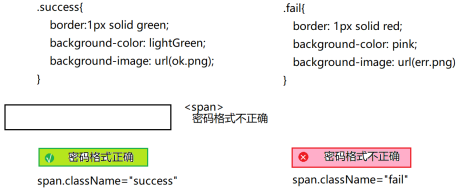
(6). 解决: 今后只要批量修改元素的多个css属性时，都用class代替.style方式

a. 先在css中定义好要使用的class，包含不同情况下的成套的css属性

b. 在js中，用程序根据不同的情况，判断该应用哪种class

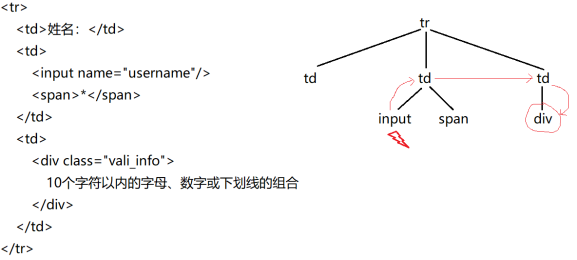
c. 优点: 无论要批量修改几个css属性，js语句永远只需要一句话:

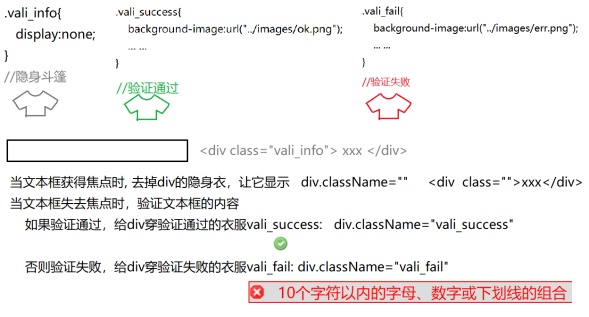
元素.className="class名"



(7). 示例: 带样式的表单验证:

对应小程序视频: 小程序->在线->DOM->day02 6. 带class样式的表单验证





0\_valiWithCss.html

|  |
| --- |
| <!doctype html>  <html>  <head>  <meta charset="UTF-8">  <title>实现带样式的表单验证</title>  <style>  table{width:700px}  td:first-child{width:60px}  td:nth-child(2){width:200px}  td:first-child+td{width:200px}  td span{color:red}  .vali\_info{  display:none;  }  .txt\_focus{  border-top:2px solid black;  border-left:2px solid black;  }  .vali\_success,.vali\_fail{  background-repeat:no-repeat;  background-position:left center;  display:block;  }  .vali\_success{  background-image:url("../images/ok.png");  padding-left:20px;  width:0px;height:20px;  overflow:hidden;  }  .vali\_fail{  background-image:url("../images/err.png");  border:1px solid red;  background-color:#ddd;  color:Red;  padding-left:30px;  }  </style>  </head>  <body>  <form id="form1">  <h2>增加管理员</h2>  <table>  <tr>  <td>姓名：</td>  <td>  <input name="username"/>  <span>\*</span>  </td>  <td>  <div class="vali\_info">  10个字符以内的字母、数字或下划线的组合  </div>  </td>  </tr>  <tr>  <td>密码：</td>  <td>  <input type="password" name="pwd"/>  <span>\*</span>  </td>  <td>  <div class="vali\_info">6位数字</div>  </td>  </tr>  <tr>  <td></td>  <td colspan="2">  <input type="submit" value="保存"/>  <input type="reset" value="重填"/>  </td>  </tr>  </table>  </form>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: 用户操作姓名文本框时会触发变化  var txtName=document.getElementsByName("username")[0];  //[0]: 从查询结果的类数组对象中取出找到的唯一的文本框元素对象  console.log(txtName);  //2. 绑定事件处理函数  //本例中: 当文本框获得焦点时，旁边的提示框，显示出来  txtName.onfocus=function(){  //3. 查找要修改的元素  //本例中: 当文本款获得焦点时，应该查找文本框的爹的下一个兄弟td中的div  var div=this.parentElement.nextElementSibling.children[0];  //4. 修改元素  //本例中: 当文本款获得焦点时，让旁边的div显示,脱掉div的隐身衣class  div.className="";  }  //2. 绑定事件处理函数  //本例中: 当文本框失去焦点时，就要验证文本框中用户输入的内容，并根据验证结果修改旁边提示框的样式类  txtName.onblur=function(){  //3. 查找要修改的元素  //本例中：当文本框失去焦点时，还是要修改文本框旁边的div的class  var div=this.parentElement.nextElementSibling.children[0];  //4. 修改元素  //先定义正则表达式来规定姓名的格式  var reg=/^\w{1,10}$/;  //验证当前文本框的内容是否符合格式要求  var bool=reg.test(this.value);  //如果验证通过  if(bool==true){  //就修改div的className为vali\_success  div.className="vali\_success";  }else{//否则如果验证不通过  //就修改div的className为vali\_fail  div.className="vali\_fail";  }  }  </script>  </body>  </html>  运行结果: |

# 二. 添加删除元素:

## 1. 添加一个新元素到网页:3步

(1). 创建一个新的空元素对象:

a. var 一个新元素=document.createElement("标签名")

创建 元素

b. 比如: 想创建一个空的a元素:

var a=document.createElement("a");

结果: a : <a></a>

(2). 为新元素对象添加必要属性:

a. 比如: 如果想让这个a可以当做超链接使用，至少得设置href属性和内容

a.href="http://tmooc.cn";

a.innerHTML="go to tmooc";

结果: <a href="http://tmooc.cn"> go to tmooc </a>

b. 问题: 这个a这是在内存中游离的a元素对象。还没有挂载到dom树上，所以浏览器不知道应该把这个a显示在具体哪个位置上。所以这个a，用户暂时是看不见的！

(3). 将新元素添加到DOM树上指定父元素下: 3种:

a. 追加到一个指定父元素下所有直接子元素之后:

父元素.appendChild(新元素)

追加 子元素

b. 插入到一个指定父元素下的一个现有子元素之前:

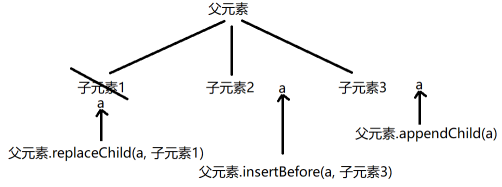
父元素.insertBefore(新元素, 现有子元素)

插入...之前

c. 替换一个指定父元素下的一个现有子元素:

父元素.replaceChild(新元素, 现有子元素)

替换 子元素



2. 示例: 向网页中添加一个新的a元素，再添加一个input元素，让input元素出现在a的指定位置上

1\_createElement.html

|  |
| --- |
| <script>  //想在body中添加一个a元素  var a=document.createElement("a");  a.href="http://tmooc.cn";  a.innerHTML="go to tmooc";  console.log(a);  document.body.appendChild(a);  //想创建一个input元素，放在a元素之前 / 之后 / 替换a元素  var input=document.createElement("input");  //input不需要设置任何属性，默认就是一个文本框  document.body.insertBefore(input,a);  //.appendChild(input);  //.replaceChild(input,a);  </script>  运行结果: |

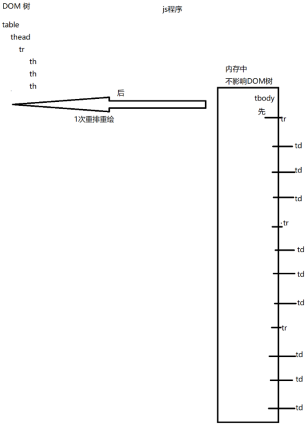
3. 优化:

(1). 问题: 浏览器显示网页的内容是经过先排版，再绘制的过程才显示出来的。只要js程序一修改DOM树，浏览器都被迫要重新对页面内容进行排版和绘制。过程非常的复杂和耗时。

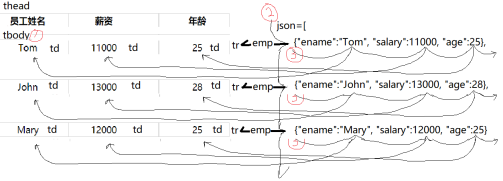
(2). 优化: 尽量减少操作DOM树的次数！

(3). 如何: 2种:

a. 如果同时添加父元素和子元素时，应该现在内存中将所有子元素添加到父元素中，最后再一次性将父元素添加到DOM树——一次重排重绘



1). 示例: 使用优化办法动态生成表格内容(tbody不在table上，也是临时创建的)



1\_createTable.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>动态创建表格</title>  <meta charset="utf-8" />  <style>  table{width:600px; border-collapse:collapse;  text-align:center;  }  td,th{border:1px solid #ccc}  </style>  </head>  <body>  <div id="data">  <table>  <thead>  <tr>  <th>员工姓名</th>  <th>薪资</th>  <th>员工年龄</th>  </tr>  </thead>  </table>  </div>  <script>  var json=[  {"ename":"Tom", "salary":11000, "age":25},  {"ename":"John", "salary":13000, "age":28},  {"ename":"Mary", "salary":12000, "age":25}  ];  //先创建一个tbody元素  var tbody=document.createElement("tbody");  //暂时不要将tbody元素追加到table中  //遍历json数组中每个员工对象  for(var emp of json){  //每遍历一个员工对象，就创建一个tr，将tr追加到tbody下  var tr=document.createElement("tr");  tbody.appendChild(tr);  //遍历当前emp对象中每个属性值  for(var key in emp){  //每遍历一个属性值，就创建一个td，并设置td的内容为当前属性值，最后将td追加到当前tr对象中  var td=document.createElement("td");  td.innerHTML=emp[key];//因为key是变量，所以不加""  tr.appendChild(td);  }  }  //等tbody中所有内容都加载完了，最后一次性将整个tbody追加到DOM树  //查找table对象  var table=document.querySelector("#data>table");  table.appendChild(tbody);  //浏览器只需要一次重排重绘  //对效果没有影响，却极大的优化了页面加载效率！  </script>  </body>  </html>  运行结果: |

b. 如果父元素已经在页面上了，要同时添加多个平级子元素，就要借助于一种新对象——文档片段对象

1). 什么是文档片段对象: 内存中临时保存多个平级子元素的虚拟父元素对象

2). 何时: 只要父元素已经在页面上了，要同时添加多个平级子元素，都要用文档片段对象

3). 如何: 3步:

i. 创建一个空的文档片段对象：

var frag=document.createDocumentFragment()

创建 文档 片段

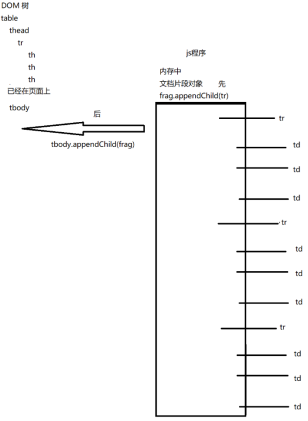
ii. 先将所有平级子元素临时加入到文档片段对象中暂存

frag.appendChild(子元素)

iii. 最后再将整个文档片段对象，一次性添加到DOM树上指定父元素下

父元素.appendChild(frag)

4). 结果: frag对象将平级子元素送到页面上指定父元素下之后，frag对象就释放了。不会成为页面上一级真正的元素！不影响页面元素的结构。



5). 示例: 使用文档片段动态生成表格内容(tbody已经在页面上了)

1\_createTable\_fragment.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>动态创建表格</title>  <meta charset="utf-8" />  <style>  table{width:600px; border-collapse:collapse;  text-align:center;  }  td,th{border:1px solid #ccc}  </style>  </head>  <body>  <div id="data">  <table>  <thead>  <tr>  <th>员工姓名</th>  <th>薪资</th>  <th>员工年龄</th>  </tr>  </thead>  <tbody>  </tbody>  </table>  </div>  <script>  var json=[  {"ename":"Tom", "salary":11000, "age":25},  {"ename":"John", "salary":13000, "age":28},  {"ename":"Mary", "salary":12000, "age":25}  ];  //因为tbody已经在页面上了  //所以，只能先创建一个临时的托盘——文档片段对象  var frag=document.createDocumentFragment();  //遍历json数组中每个员工对象  for(var emp of json){  //每遍历一个员工对象，就创建一个tr  var tr=document.createElement("tr");  //但是，不能将tr直接追加到tbody下  //而是，先将tr临时追加到文档片段对象托盘中暂存  frag.appendChild(tr);  //遍历当前emp对象中每个属性值  for(var key in emp){  //每遍历一个属性值，就创建一个td，并设置td的内容为当前属性值，最后将td追加到当前tr对象中  var td=document.createElement("td");  td.innerHTML=emp[key];//因为key是变量，所以不加""  tr.appendChild(td);  }  }  //最后,再一次性的将文档片段托盘添加到tbody中  var tbody=document.querySelector("#data>table>tbody");  tbody.appendChild(frag);  //查看以下tbody的孩子中有没有一个frag元素对象  console.log(tbody.children);  </script>  </body>  </html>  运行结果: |

4. 删除元素: 父元素.removeChild(要删除的元素)

# HTML DOM常用对象（了解）:

HTML DOM标准对个别常用的复杂的元素对象提供了简化的函数来操作这些对象。但是，不是所有元素都有这种简化。

1. <img>元素:

(1). 创建:

a. 旧DOM: var img=document.createElement("img")

b. HTML DOM: var img=new Image();

(2). 强调: 其实只有<img>和<option>两种元素可以new创建。其余元素都不能用new!

2. <table>元素: table是一个大家庭，从table到td，包含四级元素。所以，table采用逐级管理方式

(1). table管着行分组:

a. 添加行分组:

1). 旧DOM: 2步:

var 行分组=document.createElement("标签名")

table.appendChild(行分组);

2). HTML DOM: 1步，既创建行分组对象，又将行分组对象立刻添加到table上

var thead=table.createTHead()

var tbody=table.createTBody()

var tfoot=table.createTFoot();

3). 问题: 如果table已经在页面上了，就不应该用这种方式，过早的将空的行分组添加到table。结果，今后只要向行分组中添加任何东西都会导致重排重绘

b. 删除行分组:

table.deleteTHead()

table.deleteTFoot()

~~没有table.deleteTBody()~~

c. 获取行分组:

table.tHead 因为table中只能有一个thead，直接隶属于table

table.tFoot 因为table中只能有一个tfoot，直接隶属于table

table.tBodies[i] 因为HTML规定table下可以有多个tbody，所以多个tbody是放在一个集合中保存的。并不直接隶属于table

(2). 行分组管着行

a. 添加行:

1). 旧js中: 2步

var tr=document.createElement("tr");

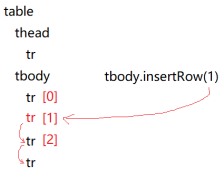
行分组.appendChild(tr)

2). HTML DOM中: 1步

i. var tr=行分组.insertRow(i)

ii. 意为: 先创建一个新的空行，然后再将新的空行插入到当前行分组中i位置。

iii. 强调: 如果i位置已经有行了，则新行不会替换i位置的行，而是将原i位置的行向后挤，新行占据i位置。



iv. 固定套路:

①在开头插入新行: var tr=行分组.insertRow(0)

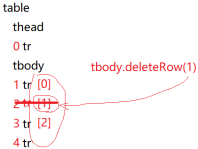
②在末尾追加新行: var tr=行分组.insertRow()

b. 删除行:

1). 行分组.deleteRow(i)

2). 意为: 删除行分组中i位置的行

3). 强调: 参数i，是在行分组内的相对下标位置，而不是行在整个表中的下标位置



4). 问题: 如果要删的行号，不确定，又不容易用肉眼看出行号来，deleteRow()行号参数应该写几?

5). 解决: 每个行对象都有一个tr.rowIndex属性，记录了当前行在整个表中的下标位置。所以，tbody.deleteRow(tr.rowIndex)，自动获得要删除的整个表行下标。

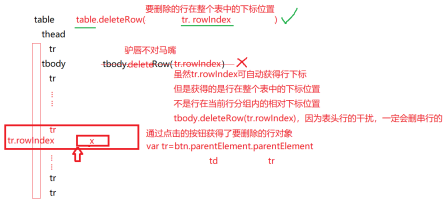
6). 问题: tbody要求行号是在tbody内的相对下标，但是tr.rowIndex自动获得只能是行在整个表中的下标位置。两者不一致！如果用tbody.deleteRow(tr.rowIndex)删除行，因为受到表头行的干扰，几乎一定会删串行/错位！

7). 终极删除行的最正确解决方案: table.deleteRow(tr.rowIndex)

因为table要的是行在整个表中的下标位置

tr.rowIndex给的刚好也是行在整个表中的下标位置

两者刚好配对！



c. 获取行: 行分组.rows[i] //获取当前行分组中i位置的行对象

(3). 行管着格:

a. 添加格: var td=tr.insertCell() //在行末尾追加一格

b. 删除格: tr.deleteCell(i) //删除i位置的一个格

c. 获取格: tr.cells[i] //获取行中i位置的格对象

(4). 示例: 动态创建表格，带删除行功能

对应小程序视频: 小程序->在线->DOM->day03 1 DOM添加新元素...

1\_createTable\_HTMLDOM.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>动态创建表格</title>  <meta charset="utf-8" />  <style>  table{width:600px; border-collapse:collapse;  text-align:center;  }  td,th{border:1px solid #ccc}  </style>  </head>  <body>  <div id="data">  <table>  <thead>  <tr>  <th>员工姓名</th>  <th>薪资</th>  <th>员工年龄</th>  <th>删除</th>  </tr>  </thead>  </table>  </div>  <script>  var json=[  {"ename":"Tom", "salary":11000, "age":25},  {"ename":"John", "salary":13000, "age":28},  {"ename":"Mary", "salary":12000, "age":25}  ];  //先创建一个tbody元素  var tbody=document.createElement("tbody");  //暂时不要将tbody元素追加到table中  //查找table对象  var table=document.querySelector("#data>table");  //遍历json数组中每个员工对象  for(var emp of json){  //每遍历一个员工对象，就创建一个tr，将tr追加到tbody下  // var tr=document.createElement("tr");  // tbody.appendChild(tr);  var tr=tbody.insertRow();//末尾追加新行  //tbody.insertRow(0); //始终在开头插入新行  //遍历当前emp对象中每个属性值  for(var key in emp){  //每遍历一个属性值，就创建一个td，并设置td的内容为当前属性值，最后将td追加到当前tr对象中  // var td=document.createElement("td");  // tr.appendChild(td);  var td=tr.insertCell();  td.innerHTML=emp[key];//因为key是变量，所以不加""  }  //当为当前行添加完数据格之后，再额外添加一个新格  var td=tr.insertCell();  //创建一个新按钮，并将新按钮追加到td内容中  var btn=document.createElement("button");  btn.innerHTML="×";  td.appendChild(btn);  //为按钮绑定单击事件处理函数  btn.onclick=function(){  //测试:点哪个按钮，让当前按钮变成❀  //this.innerHTML="❀";  //查找要修改的元素  //本例中: 点按钮要删除当前按钮所在的行  var tr=this.parentElement.parentElement;  // btn td tr  //修改元素  //删除这类操作，都是危险操作，所以在正式删除前都要再跟用户确认一下是否真的继续删除！  //如何跟用户确认？弹出确认框，请用户选择是否继续  //为了让提示更有针对性，对用户更有帮助，应该加入标志性的员工姓名  //先获得当前行中第一个格里保存的人名  var ename=tr.cells[0].innerHTML;  // td td的内容  var result=confirm(`是否继续删除 ${ename} 吗？`);  //如果用户点的是确认按钮，确认框整体返回true，说明用户想继续删除行  if(result==true){  //本例中: 最终要执行的  table.deleteRow(tr.rowIndex);  //因为前边已经查找过table元素了，所以table可以直接使用！  }//否则如果用户点的是取消，确认框整体返回false，条件不满足，什么也不干！  }  }  //等tbody中所有内容都加载完了，最后一次性将整个tbody追加到DOM树  table.appendChild(tbody);  //浏览器只需要一次重排重绘  //对效果没有影响，却极大的优化了页面加载效率！  </script>  </body>  </html>  运行结果: |

## 补: 浏览器三大对话框:

1. prompt: 输入框

2. alert: 警告框 只有一个确定按钮，没得选择！

3. confirm: 确认框 有两个按钮: 确认和取消。有的选择。

确认框有返回值

如果用户点确认按钮，则确认框返回true，说明用户同意继续

如果用户点取消按钮，则确认框返回false，说明用户不同意继续

# Day04

# 一. HTML DOM:

1. <form>元素:

(1). 获取页面中一个form元素: var form=document.forms[i]

a. 因为浏览器已经将一个网页中所有的<form>元素都保存在document的forms集合中

b. 可以用下标[i]，访问forms集合中保存的第i个form元素对象.

c. 如果网页中只有一个表单元素: var form=document.forms[0]

(2). 获取<form>元素内的表单元素时:

a. 标准: form.elements[id/i/name]

1). 因为form已经将表单中的所有表单元素都集中保存在了form的elements集合中。

2). 可通过下标[i]或[id]或[name]方式来获取<form>内的一个或多个表单元素

3). 返回值:

i. 如果要找的表单元素只有一个，则返回一个元素对象，不放在类数组对象中

ii. 如果要找的表单元素时多个，则放在一个类数组对象中返回。

b. 简写: 如果这个元素有name属性，则可以简写为:

form.name名

(3). 让表单元素自动获得焦点: 表单元素.focus()

(4). 示例: 简化带样式的表单验证案例，并让姓名文本框自动获得焦点

day03 valiWithCSS/0\_valiWithCss.html

|  |
| --- |
| <!doctype html>  <html>  <head>  <meta charset="UTF-8">  <title>实现带样式的表单验证</title>  <style>  table{width:700px}  td:first-child{width:60px}  td:nth-child(2){width:200px}  td:first-child+td{width:200px}  td span{color:red}  .vali\_info{  display:none;  }  .txt\_focus{  border-top:2px solid black;  border-left:2px solid black;  }  .vali\_success,.vali\_fail{  background-repeat:no-repeat;  background-position:left center;  display:block;  }  .vali\_success{  background-image:url("images/ok.png");  padding-left:20px;  width:0px;height:20px;  overflow:hidden;  }  .vali\_fail{  background-image:url("images/err.png");  border:1px solid red;  background-color:#ddd;  color:Red;  padding-left:30px;  }  </style>  </head>  <body>  <form id="form1">  <h2>增加管理员</h2>  <table>  <tr>  <td>姓名：</td>  <td>  <input name="username"/>  <span>\*</span>  </td>  <td>  <div class="vali\_info">  10个字符以内的字母、数字或下划线的组合  </div>  </td>  </tr>  <tr>  <td>密码：</td>  <td>  <input type="password" name="pwd"/>  <span>\*</span>  </td>  <td>  <div class="vali\_info">6位数字</div>  </td>  </tr>  <tr>  <td></td>  <td colspan="2">  <input type="submit" value="保存"/>  <input type="reset" value="重填"/>  </td>  </tr>  </table>  </form>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: 用户操作姓名文本框时会触发变化  // var txtName=document.getElementsByName("username")[0];  //[0]: 从查询结果的类数组对象中取出找到的唯一的文本框元素对象  // var txtPwd=document.getElementsByName("pwd")[0];  //先找<form>元素：  var form=document.forms[0];  //再在form元素中用简写查找表单元素  var txtName=form.username;  var txtPwd=form.pwd;  //希望开局就让姓名文本框自动获得焦点  txtName.focus();  console.log(txtName);  //2. 绑定事件处理函数  //本例中: 当文本框获得焦点时，旁边的提示框，显示出来  txtName.onfocus=function(){  //3. 查找要修改的元素  //本例中: 当文本款获得焦点时，应该查找文本框的爹的下一个兄弟td中的div  var div=this.parentElement.nextElementSibling.children[0];  //4. 修改元素  //本例中: 当文本款获得焦点时，让旁边的div显示,脱掉div的隐身衣class  div.className="";  }  //2. 绑定事件处理函数  //本例中: 当文本框失去焦点时，就要验证文本框中用户输入的内容，并根据验证结果修改旁边提示框的样式类  txtName.onblur=function(){  //3. 查找要修改的元素  //本例中：当文本框失去焦点时，还是要修改文本框旁边的div的class  var div=this.parentElement.nextElementSibling.children[0];  //4. 修改元素  //先定义正则表达式来规定姓名的格式  var reg=/^\w{1,10}$/;  //验证当前文本框的内容是否符合格式要求  var bool=reg.test(this.value);  //如果验证通过  if(bool==true){  //就修改div的className为vali\_success  div.className="vali\_success";  }else{//否则如果验证不通过  //就修改div的className为vali\_fail  div.className="vali\_fail";  }  }  </script>  </body>  </html>  运行结果: |

\*\*\*\*BOM\*\*\*\*Browser Object Model

浏览器 对象 模型

# 一. 什么是BOM:

1. 什么是BOM: 专门操作浏览器窗口的对象和函数

2. 何时: 今后只要希望操作浏览器窗口，或获取浏览器软件相关信息，都用BOM~~而不是DOM~~

3. 问题: BOM没有标准，兼容性极差！用的越来越少！

4. 包括:

(1). window: 3个角色

(2). history: 控制前进后退和刷新

(3). location: 控制页面跳转，也可以获取当前地址栏里的url信息

(4). navigator: 保存着所有浏览器的配置信息(对象)

(5). screen: 保存着当前浏览器所在的硬件设备的显示设备信息

(6). event: 保存着浏览器中触发的事件的相关信息

(7). document: 属于DOM，专门保存和操作网页的内容——前边已学过

# 二. window： 3个角色:

因为过于简单，所以小程序中未配视频，回看TMOOC或看笔记正文复习即可。

1. 代替ECMAScript中的global充当js语言的全局作用域对象

我们在全局声明的变量和函数默认都保存在window中

2. 集中保存了所有原生的直接可用的对象和函数：

(1). 原生: 所有浏览器自带的，不需要我们自己定义，就可直接使用的对象或函数

(2). 包括: ECMAScript+DOM+BOM=原生

3. window代表当前浏览器窗口对象:

(1). 获取当前窗口的大小: 2组:

a. 完整浏览器窗口大小: window.outerWidth window.outerHeight

b. 仅文档显示区部分的大小: window.innerWidth window.innerHeight



(2). 打开和关闭窗口:

a. 打开新窗口: window.open()

b. 关闭窗口: window.close()

# 三. 打开新链接共有几种方式：4种:

对应小程序视频: 小程序->在线->DOM->day03 4. 打开新链接4种方式

1. 在当前窗口打开新链接，可以后退

(1). HTML: <a href="url" target="\_self" >

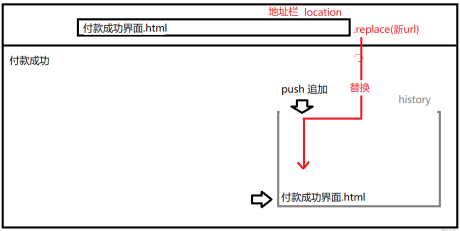
(2). JS: window.open("url", "\_self");

2. 在当前窗口打开新链接，禁止后退

~~(1). HTML:~~

(2). JS: location.replace("新url");

用新url代替history中旧的一条url，无法再后退到旧url了！



3. 在新窗口打开新链接，可以同时打开多个

(1). HTML: <a href="url" target="\_blank" >

(2). JS: window.open("url", "\_blank");

4. 在新窗口打开新链接，同时只能打开一个！

(1). HTML: <a href="url" target="自定义窗口名" > -> 新窗口的window.name值

(2). JS: window.open("url", "自定义窗口名"); -> 新窗口的window.name值

5. 原理:

(1). 每个窗口在内存中都有一个唯一的name属性，标记当前窗口的名字。

(2). 浏览器规定相同name属性值的窗口，在内存中同时只能打开一个

(3). 后打开的同名窗口，会覆盖先打开的同名窗口！

(4). 其实: <a>的target属性和window.open()的第二个实参值，都是在给新的窗口指定name名！

(5). 强调: 禁止使用"name"当做变量名或属性名！因为name是BOM中的保留字，有特殊用途！

(6). 预定义窗口名:

a. \_self，不自定义窗口名，而是自动将当前窗口自己的名字，作为新窗口的name名，

自己的

后果: 因为新窗口和旧窗口自己同名，所以新窗口覆盖旧窗口，不会额外打开新窗口了。

b. \_blank: 也不自定义窗口名，而是让窗口名空白！但是浏览器不允许窗口名空着，浏览器就会在底层自动为新窗口分配新随机名称（人看不见！）。

blank: 空白

后果: 因为每次打开的新窗口随机名称都不一样！所以，不会覆盖，而是可以反复打开多个！

6. 示例: 以四种不同的情况打开新链接

day03 BOM/4\_open.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8"/>  <title>打开新链接方式总结</title>  <script>  /\*打开新链接方式总结：  1. 在当前窗口打开，可后退  2. 在当前窗口打开，不可后退  3. 在新窗口打开，可打开多个  4. 在新窗口打开，只能打开一个  \*/  function open1(){  window.open("http://tmooc.cn","\_self");  }  function open2(){  location.replace("http://tmooc.cn");  }  function open3(){  window.open("http://tmooc.cn","\_blank");  }  function open4(){  window.open("http://tmooc.cn","tmooc");  }  </script>  </head>  <body>  <h3>1. 在当前窗口打开，可后退</h3>  <a href="http://tmooc.cn" target="\_self">欢迎访问tmooc</a><br>  <button onclick="open1()">欢迎访问tmooc</button>  <h3>2. 在当前窗口打开，禁止后退</h3>  <button onclick="open2()">欢迎访问tmooc</button>  <h3>3. 在新窗口打开，可同时打开多个</h3>  <a href="http://tmooc.cn" target="\_blank">欢迎访问tmooc</a><br>  <button onclick="open3()">欢迎访问tmooc</button>  <h3>4. 在新窗口打开，同时只能打开一个</h3>  <a href="http://tmooc.cn" target="tmooc">欢迎访问tmooc</a><br>  <button onclick="open4()">欢迎访问tmooc</button>  </body>  </html>  运行结果: |

# 四. history:

1. 什么是: 当前窗口中保存当前窗口打开后成功访问过得所有url地址的历史记录数组

2. 何时: history只给我们开放了前进后退和刷新的功能。其内容，我们既不可见，又不能修改！

3. 如何: 只能做三件事:

(1). 前进一步: history.go(1)

(2). 后退一步: history.go(-1)

有时如果后退一步不好用，还可以后退两步: history.go(-2)

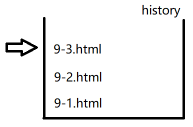
(3). 刷新: history.go(0)

4. 示例: 用js程序实现前进后退

步骤: 先用live server运行起来9-1.html

然后，依次点击超链接2和3跳转到9-2.html和9-3.html

结果: 让当前窗口的history中，有三个历史记录url，才可以测试前进后退功能



day03 BOM/

9-1\_history.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>使用 history 对象</title>  <meta charset="utf-8" />  </head>  <body>  <h2>9-1.html</h2>  <div>  <a href="9-1\_history.html">1</a>&nbsp;&nbsp;  <a href="9-2\_history.html">2</a>&nbsp;&nbsp;  <a href="9-3\_history.html">3</a>&nbsp;&nbsp;  </div><br/>  <div>  <a href="javascript: history.go(1)">前进一次</a>&nbsp;&nbsp;  <a href="javascript: history.go(2)">前进二次</a>&nbsp;&nbsp;  </div>  </body>  </html>  运行结果: |

9-2\_history.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>使用 history 对象</title>  <meta charset="utf-8" />  </head>  <body>  <h2>9-2.html</h2>  <div>  <a href="9-1\_history.html">1</a>&nbsp;&nbsp;  <a href="9-2\_history.html">2</a>&nbsp;&nbsp;  <a href="9-3\_history.html">3</a>&nbsp;&nbsp;  </div><br/>  <div>  <a href="javascript: history.go(-1)">后退一次</a>&nbsp;&nbsp;  <a href="javascript: history.go(1)">前进一次</a>&nbsp;&nbsp;  </div>  </body>  </html>  运行效果: |

9-3\_history.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>使用 history 对象</title>  <meta charset="utf-8" />  </head>  <body>  <h2>9-3.html</h2>  <div>  <a href="9-1\_history.html">1</a>&nbsp;&nbsp;  <a href="9-2\_history.html">2</a>&nbsp;&nbsp;  <a href="9-3\_history.html">3</a>&nbsp;&nbsp;  </div><br/>  <div>  <a href="javascript: history.go(-2)">后退二次</a>&nbsp;&nbsp;  <a href="javascript: history.go(-1)">后退一次</a>&nbsp;&nbsp;  </div>  </body>  </html>  运行结果 |

# 五. location

地址/位置

1. 什么是: 保存当前地址栏中的地址信息，并提供页面跳转操作的函数 的对象

2. 何时: 2种:

(1). 获得地址栏中的包含的相关信息时  
 (2). 执行页面跳转操作时

3.. 如何:

(1). 分段获得地址栏中url中各个部分的内容

a. location.href 获得完整url地址

b. location.protocol 获得协议部分

c. location.host 获得主机名+端口号

d. location.hostname 仅获得主机名

e. location.port 获得端口号

f. location.pathname 获得相对路径

g. location.search 获得?及其之后的查询字符串  
 h. location.hash 获得#及其之后的锚点地址

电话上的#

(2). 提供了一些函数:

a. location.assign("新url") 在当前窗口打开新链接，跳转到新url

等效于: window.open("新url","\_self")

b. location.replace("新url") 在当前窗口打开新链接，禁止后退

c. location.reload() 刷新页面

等效于: history.go(0)

4. 示例: 获取location各个部分的内容

day03 BOM/10\_location.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>事件处理</title>  <meta charset="utf-8" />  </head>  <body>  <form>  姓名:<input name="username"/><br>  密码:<input type="password" name="pwd"/><br>  爱好:<input type="checkbox" name="favs" value="running"/>跑步  <input type="checkbox" name="favs" value="swimming"/>游泳  <input type="checkbox" name="favs" value="basketball"/>篮球  <br>  <input type="submit"/>  </form>  <a href="#top">返回顶部</a>  <script>  console.log(location.href);  console.log(location.protocol);  console.log(location.host);  console.log(location.hostname);  console.log(location.port);  console.log(location.pathname); //在js程序中判断，根据pathname不同，动态改变主导航菜单中哪个菜单高亮显示，其余菜单恢复原样  console.log(location.search);  console.log(location.hash);  </script>  </body>  </html>  运行结果: |

# 六. navigator

1. 什么是: 保存浏览器软件和配置信息的对象

2. 何时: 只要希望获得浏览器配置信息时，都可用navigator

3. 如何:

(1). 查看浏览器已经安装的插件列表，比如判断是否安装了QQMusic插件或flash插件

navigator.plugins

navigator.plugins[“要查询的插件名称”]

(2). 查看浏览器的名称、内核和版本号

navigator.userAgent

# 七. 事件:

## 1. 什么是事件: 浏览器自动触发的或用户手动触发的页面内容和状态的变化

## 2. 什么是事件处理函数: 在事件发生前，提前保存在元素的事件属性上的函数。当事件发生时，浏览器会自动调用元素的事件属性上的事件处理函数。

## 3. 何时: 只要希望当事件发生时，自动执行一项任务时，都要提前绑定事件处理函数

## 4. 如何绑定事件处理函数: 3种:

对一个小程序视频: 小程序->在线->DOM->day04 1. 添加事件监听 移除事件监听

### (1). 在HTML中绑定: ——几乎不用

a. HTML中: <元素 on事件名="fun()">

b. js中: 定义一个全局函数:function fun(){ ... }

c. 问题: 不符合内容与行为分离的原则，极其不便于维护。

### (2). 在js中用赋值方式绑定:

a. 元素对象.on事件名=function(){ ... }

b. 问题: 一个事件属性上只能赋值一个事件处理函数，不灵活

无法当一个事件发生时，同时执行多个事件处理函数！

### (3). 在js中通过添加事件监听对象的方式:

a. 元素对象.addEventListener("事件名", 事件处理函数)

添加 事件 监听对象

b. 原理:

1). 在浏览器内存中有一个巨大的事件监听对象队列

2). 每次addEventListener()时，都会:2件事

i. 先创建一个事件监听对象:包含三项内容:

①哪个元素②什么事件③事件处理函数

ii. 将新创建的事件对象添加到浏览器的事件队列中保存

3). 当事件在一个元素上发生时，浏览器会遍历事件队列中每个事件监听对象，找到和当前元素当前事件名匹配的一个或多个监听对象，自动执行其中的事件处理函数。找到集合符合条件的监听对象，就执行几个事件处理函数。

c. 移除事件监听对象:

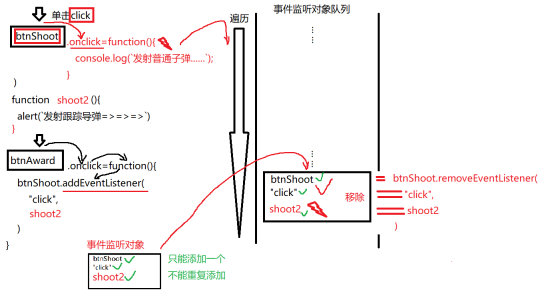
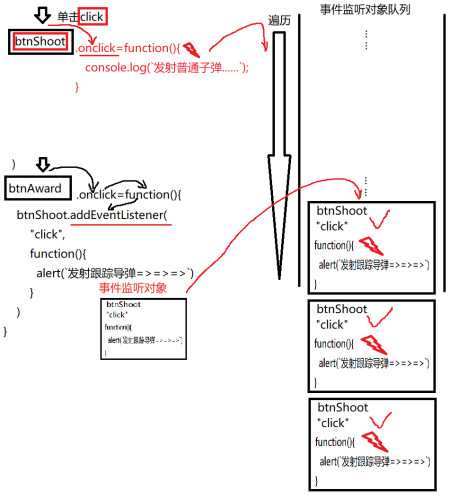
1) 元素对象.removeEventListener("事件名",原事件处理函数)

2). 问题: 移除事件处理函数时，如果只是把事件处理函数写的和原来事件处理函数一模一样，依然移除不掉！

3). 原因: 浏览器在移除事件监听对象时，会比较元素对象是否相同，事件名是否相同，事件处理函数是否是原事件处理函数对象（比的是地址！）。三个比较都相同，才能移除这个事件监听对象。如果绑定时和移除时都用匿名函数，则两次function的地址绝对不同！只要地址不同！就无法移除原事件监听对象。

4). 解决: 今后，只要一个事件处理函数有可能被移除，就不能用匿名函数。因为匿名函数创建后，就再也找不到了！应该用有名称的函数！移除时还可通过函数名获得原函数对象的地址!

5). 强调: 一旦使用有名称的函数添加事件监听，则完全相同的事件监听对象只能添加一个到队列中，不予许重复添加！



### (4). 总结:

a. 如果一个元素的一个事件上，只需要一个处理函数，不会增加新的处理函数，就依然可以使用.onclick=function(){ ... }

b. 只有一个元素的一个事件上，有可能同时执行多个事件处理函数时，才被迫改为用addEventListener()

(5). 示例: 点按钮发射多种子弹

day04 1\_addEventListener.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8"/>  <title>...</title>  </head>  <body>  <button id="btnShoot">shoot</button><br>  <button id="btnAward">获得跟踪导弹</button><br>  <button id="btnBreak">失去跟踪导弹</button><br>  <script>  var btnShoot=document.getElementById("btnShoot");  var btnAward=document.getElementById("btnAward");  var btnBreak=document.getElementById("btnBreak");  //希望开局点shoot按钮，只能发射一种普通子弹  btnShoot.onclick=function(){  console.log(`发射普通子弹......`);  }  //希望点获得跟踪导弹之后，为shoot按钮多添加一种新的跟踪导弹，希望再点shoot时，能同时发射两种子弹！  //因为跟踪导弹的事件处理函数有可能被移除  //所以必须定义为有名称的函数  function shoot2(){  alert(`发射跟踪导弹=>=>=>`)  }  btnAward.onclick=function(){  // btnShoot.onclick=function(){  // alert(`发射跟踪导弹=>=>=>`)  // }  //因为跟踪导弹的事件处理函数有可能被移除  //所以绑定时就要用函数名绑定  btnShoot.addEventListener("click",shoot2)  }  //希望点失去跟踪导弹，从shoot按钮上移除跟踪导弹的事件处理函数  btnBreak.onclick=function(){  //移除时必须用原函数名移除  btnShoot.removeEventListener("click",shoot2)  }  </script>  </body>  </html>  运行结果： |

# 5. 事件模型:

对应小程序视频: 小程序->在线->DOM->day04 2. 事件模型 event model ...

### (1). 什么是: 从触发事件到所有事件处理函数执行完所经历的过程

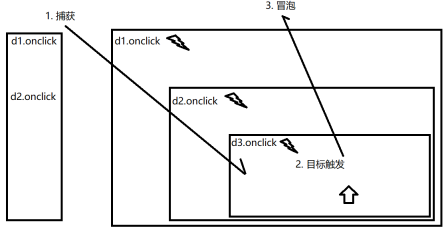
### (2). 包括: 3个阶段:

#### a. 捕获: 由外向内依次记录单击的元素的所有父元素上绑定的事件处理函数都有哪些。

——只记录不执行！

#### b. 目标触发: 优先触发当前点击的元素上的事件处理函数

#### c. 冒泡: 由内向外依次触发捕获阶段记录的所有父元素上的事件处理函数



# 6. 事件对象:

(1). 什么是: 事件发生时，浏览器自动创建的，保存事件信息的对象

(2). 何时: 2种:

a. 想获得事件信息时

b. 想修改事件的默认行为时

(3). 先获得事件对象，才能使用: 事件对象总是默认作为事件处理函数的第一个实参值悄悄传入！

在绑定事件处理函数时:

//当事件发生时自动创建event对象

↓ 信任！

元素.onclick = function( e ){

//e=event

}

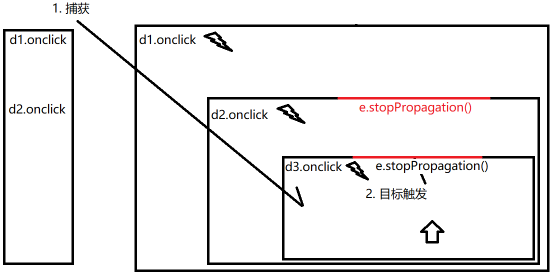
(4). 提供了:

### a. 取消冒泡/停止蔓延: e.stopPropagation()

停止 蔓延

1). 强调: 在当前事件处理函数内，e.stopPropagation()写前写后无所谓！

因为: e.stopPropagation()防的是父元素触发事件，不防当前元素自己！



2). 示例: 点哪个div，哪个div喊疼！没点的！不要喊！

2\_bubble.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>事件处理</title>  <meta charset="utf-8"/>  <style>  #d1 #d2 #d3{cursor:pointer}  #d1 {  background-color: green;  position: relative;  width: 150px;  height: 150px;  text-align: center;  cursor: pointer;  }  #d2 {  background-color: blue;  position: absolute;  top: 25px;  left: 25px;  width: 100px;  height: 100px;  }  #d3 {  background-color: red;  position: absolute;  top: 25px;  left: 25px;  width: 50px;  height: 50px;  line-height: 50px;  }  </style>  </head>  <body>  <div id="d1">  <div id="d2">  <div id="d3">  </div>  </div>  </div>  <script>  var d1=document.getElementById("d1");  var d2=document.getElementById("d2");  var d3=document.getElementById("d3");  //想点哪个div，哪个div喊疼！  d1.onclick=function(){  alert("d1疼！")  }  d2.onclick=function(e){  e.stopPropagation()  alert("d2疼！")  }  d3.onclick=function(e){  e.stopPropagation()  alert("d3疼！")  }  </script>  </body>  </html>  运行结果: |

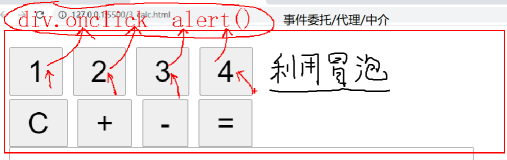
### b. 利用冒泡/事件委托:

对应小程序视频: 小程序->在线->DOM->day04 3. 利用冒泡...

1). 问题: 浏览器触发事件时，是通过遍历监听对象队列的方式，查找符合要求的监听对象。所以，监听对象队列中的对象数量多少，决定了查找的速度。队列中监听对象越多，查找越慢。反之，队列中监听对象越少，查找越快

2). 优化: 尽量减少整个页面中事件监听的个数！

3). 如何: 今后只要多个平级子元素都需要绑定相同的事件时，其实只需要把事件集中绑定在父元素上一份，所有子元素就可通过冒泡共用！因为任何情况只要点击子元素，一定会冒泡触发父元素上的事件处理函数。

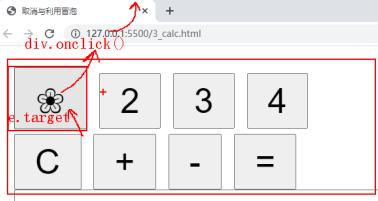


4). 问题: 当事件绑定在父元素上时，this不再指实际触发事件的子元素，而是指父元素了！事件绑定在哪个元素上，this就指谁！

5). 解决: 今后只要利用冒泡，都要用e.target代替this，用于获得最初触发事件的那个子元素！

e.target: 事件对象中自动保存的属性，专门用于保存最初触发事件的那个子元素。

vs this: e.target不会随冒泡而改变！this会随冒泡而改变！



6). 问题: 如果点的子元素不是想要的，我们就不想让它触发事件。只有点在希望的元素上才触发事件

7). 解决: 今后只要利用冒泡，都要先判断e.target是不是想要的子元素。只有e.target的特征是想要的子元素，才能继续执行后续操作。反之，如果e.target的特征不是想要的，则什么也不做！

# 补: 元素.nodeName 属性 记录元素的标签名！强调: 全大写！

8). 总结: 利用冒泡3步:

a. 事件只绑定在父元素上一份

b. e.target代替this，获得实际点击的子元素

c. 先判断e.target的特征符合要求，才能继续执行后续操作！

9). 示例: 实现计算器效果：

3\_calc.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>取消与利用冒泡</title>  <meta charset="utf-8"/>  </head>  <body>  <div id="keys">  <button>1</button><span>\*</span>  <button>2</button><span>\*</span>  <button>3</button><span>\*</span>  <button>4</button><span>\*</span><br>  <button>C</button><span>\*</span>  <button>+</button><span>\*</span>  <button>-</button><span>\*</span>  <button>=</button><span>\*</span>  </div>  <textarea id="sc" style="resize:none;width:200px; height:50px;" readonly></textarea>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: id为keys下的每个button都可点击  //因为每个按钮都能点，所以事件应该只绑定在父元素div上一份即可  var div=document.getElementById("keys")  //2. 绑定事件处理函数  //本例中: 绑定单击事件：  div.onclick=function(e){  //希望点每个按钮都能喊疼！  //alert("疼!");  //希望点哪个按钮，让哪个按钮的内容变成❀  //错误: 事件是绑定在父元素div上，所以this->父元素div  //this.innerHTML="❀";  //正确: e.target，代替this  //只希望button元素才能变成❀，别的元素即使点了，也不要变！  // 标签名  if(e.target.nodeName=="BUTTON"){  //e.target.innerHTML="❀"  //3. 查找要修改的元素  //本例中：每次点击按钮，都要修改下方的文本框  var sc=document.getElementById("sc");  //4. 修改元素  //先判断点击的按钮的内容  switch(e.target.innerHTML){  //如果是C，就清空显示屏文本框  case "C":  sc.value="";  break;  //如果是=，就将文本框内容，交给eval计算结果，再将结果放回显示屏中  case "=":  //错误处理: 复习第一阶段  try{//尝试执行  //eval: 可计算字符串类型的js表达式的值.——复习第一阶段  sc.value=eval(sc.value);  }catch(err){//如果出错  sc.value=err;//就把错误信息显示在文本框中  }  break;  //点击其余所有数字按钮和+ -号按钮，都只将按钮内容追加到文本框中算式结尾即可，不做计算  default:  sc.value+=e.target.innerHTML;  }  }    }  </script>  </body>  </html>  运行效果: |

c. 阻止元素自带的默认行为:

对应小程序视频: 小程序->在线->DOM->day04 4. 阻止元素自带默认行为

1). 问题: HTML中个别元素身上自带了一些自动执行的操作，但是，这些操作，有时我们不想用！

2). 比如:

i. <a href="#">xxx</a> 当点这个a时，会自动在地址栏结尾加#

ii. <input type="submit"/>和<button>放在<form>中时，只要点击，就自动提交！

3). 如何: e.preventDefault();

阻止 默认

4). 比如:

i. < a id="a1" href="#">xxx</a>

var a1=document.getElementById("a1");

a1.onclick=function (e){ e.preventDefault() ... }

ii. <input id="btnSubmit" type="submit">

<button id="btnSubmit">xxx</button>

var btnSubmit=document.getElementById("btnSubmit")

btnSubmit.onclick=function(e){ e.preventDefault() ... }

5). 简写:

i. <a href="javascript:;">xxx</a>

javascript: 让a点击时，不要跳转，而是执行一条js语句

; 什么也不做的空语句！

ii. <input type="button" value="提交"/>

<button type="button">提交</button>

6). 总结:

i. 今后只要想做一个什么都不做的死链接: <a href="javascript:;">

ii. 今后希望点表单中提交按钮时不要自动提交: <input type="button">

7). 示例: 验证a元素和input元素自带的默认行为，并阻止默认行为

4\_preventDefault.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <a id="a1" href="#">什么也不干！</a><br/>  <a href="javascript:;">什么也不干！</a>  <form>  用户名: <input name="uname"/><br/>  <!-- <input id="btnSubmit" type="submit"> -->  <!-- <button id="btnSubmit">提交</button> -->  <!-- <input type="button" value="提交"/> -->  <button type="button">提交</button>  </form>  <script>  var a1=document.getElementById("a1");  a1.onclick=function(e){  e.preventDefault();  alert("疼!")  }  var btnSubmit=document.getElementById("btnSubmit");  btnSubmit.onclick=function(e){  e.preventDefault();  alert("用ajax手动提交表单！")  }  </script>  </body>  </html>  运行结果: |

d. 获取鼠标位置: 当事件发生时，鼠标所在的位置有三组坐标

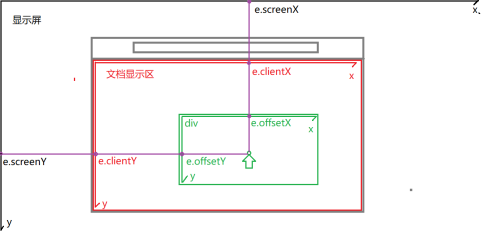
对应小程序视频: 小程序->在线->DOM->day04 5. 鼠标坐标 x y （拖拽效果选学）

(1). 鼠标位置相对屏幕左上角的距离: e.screenX e.screenY

(2). 鼠标位置相对浏览器文档显示区左上角的距离 e.clientX e.clientY

(3). 鼠标位置相对事件所在元素左上角的距离 e.offsetX e.offsetY

偏离 偏离



(4). 示例: 点击元素中某个位置，获得三组坐标:

5\_pop.html

|  |
| --- |
| <!doctype html>  <html>  <head>  <meta charset="UTF-8">  <title>在当前显示区范围内实现点不到的小方块</title>  <style>  div{  position:fixed;  top:50px;  left:100px;  width:100px;  height:100px;  background-image:url(images/xiaoxin.gif);  background-size:100%;  }  </style>  </head>  <body>  <div id="pop"></div>  <script>  var pop=document.getElementById("pop");  pop.onclick=function(e){  console.log(`鼠标距离:  屏幕左侧: ${e.screenX}, 屏幕顶部: ${e.screenY}  浏览器文档显示区左侧: ${e.clientX}, 文档显示区顶部: ${e.clientY}  div左侧: ${e.offsetX}, div顶部: ${e.offsetY}  `)  }  </script>  </body>  </html>  运行结果: |

### e. 窗口滚动事件:

对应小程序视频: 小程序->在线->DOM->day04 作业: 原生DOM页面滚动（动画效果选学）

1). 其实每当窗口滚动条发生滚动时，都会自动触发window.onscroll事件

2). window.onscroll=function(){ ... }中的代码，会在窗口滚动条滚动时自动触发

3). 获得滚动条滚动过的距离: 固定套路！——信任

var scrollTop=document.body.scrollTop||document.documentElement.scrollTop;

4). 还可以主动控制滚动到哪个位置: window.scrollTo(0, 纵向滚动到的位置)

说明: scrollTo第一个实参是水平方向的滚动位置，但是网页几乎很少水平滚动，所以第一个实参值几乎总是0。

5). 示例: 实现不带动画的电梯按钮

7\_scroll.html

|  |
| --- |
| <!doctype html>  <html>  <head>  <meta charset="UTF-8">  <title>根据页面滚动位置显示浮动框</title>  <style>  body {  height: 2000px;  }  #toTop {  position: fixed;  bottom: 100px;  right: 0;  display: none;  }  </style>  </head>  <body>  <div style="height:150px;"></div>  <div style="height:150px; background-color:pink">1楼</div>  <div style="height:150px; background-color:lightGreen">2楼</div>  <div style="height:150px; background-color:lightBlue">3楼</div>  <div style="height:150px; background-color:orange">4楼</div>  <div id="toTop">  <a id="a1" href="javascript:;">1楼</a>  <a id="a2" href="javascript:;">2楼</a>  <a id="a3" href="javascript:;">3楼</a>  <a id="a4" href="javascript:;">4楼</a>  <a href="javascript:;">返回顶部</a>  </div>  <script>  var div = document.getElementById("toTop");  //只要窗口的滚动条发生滚动时，就获取滚动条滚动过的距离  window.onscroll = function () {  var scrollTop = document.body.scrollTop || document.documentElement.scrollTop;  console.log(scrollTop);  //只要滚动距离>=500，就让div显示  if (scrollTop >= 500) {  div.style.display = "block";  } else {//否则如果滚动距离不足500，就让div隐藏  div.style.display = "none";  }  }  //点击a时回到顶部  var a = document.querySelector("#toTop>a:last-child")  a.onclick = function () {  window.scrollTo(0, 0);  }  //点哪个楼层按钮，跳到哪个楼层  var a1=document.getElementById("a1");  var a2=document.getElementById("a2");  var a3=document.getElementById("a3");  var a4=document.getElementById("a4");  a1.onclick=function(){  window.scrollTo(0, 150);  }  a2.onclick=function(){  window.scrollTo(0, 300);  }  a3.onclick=function(){  window.scrollTo(0, 450);  }  a4.onclick=function(){  window.scrollTo(0, 600);  }  </script>  </body>  </html>  运行结果: |

总结: 总结: DOM 不用背！！！用的时候来找到即可！

## 0.DOM操作万能4步:

1. 查找触发事件的元素

2. 为找到的元素绑定事件处理函数

3. 查找要修改的元素

4. 修改元素

## 1. 查找元素: 4种:

(1). 不需要查找就可直接获得的元素对象: 4个

a. document 根节点对象

b. document.documentElement <html>

c. document.head <head>

d. document.body <body>

(2). 按节点间关系查找: 如果已经获得一个元素，找周围附近的元素时，2大类关系，6个属性

a. 父子关系: 4个属性:

1). 元素.parentElement  
 2). 元素.children

3). 元素.firstElementChild

4). 元素.lastElementChild

b. 兄弟关系: 2个属性

1). 元素.previousElementSibling

2). 元素.nextElementSibling

(3). 按HTML特征查找元素: 只用一个条件就可以找到想要的元素时，4个函数

a. var 一个元素对象=document.getElementById("id名")

b. var 类数组对象=任意父元素.getElementsByTagName("标签名")

c. var 类数组对象=任意父元素.getElementsByClassName("class名")

d. var 类数组对象=document.getElementsByName("name名")

(4). 按选择器查找元素: 当查找条件复杂时，2个函数

a. 只查找一个符合要求的元素：

var 一个元素对象=任意父元素.querySelector("选择器");

b. 查找多个符合要求的元素

var 类数组对象=任意父元素.querySelectorAll("选择器");

|  |
| --- |
| 总结: 查找方法的返回值:  1. 如果原函数返回下标位置i，则如果找不到，返回-1  2. 如果原函数返回一个对象或数组，则如果找不到，返回null  3. 如果原函数返回一个类数组对象，则如果找不到，返回空类数组对象{ length:0 } |

## 2. 修改: 3种

(1). 内容: 3种

a. 绝大多数情况下，用.innerHTML就够了！

b. 除非在获取内容时，确实希望去掉内嵌标签，只要文本时，才用textContent

c. 获取或修改表单元素的值: 表单元素.value

(2). 属性: 3种:

a. 字符串类型的HTML标准属性: 2种:

1). 旧核心DOM 4个函数

i. 元素.getAttribute("属性名")

ii. 元素.setAttribute("属性名","属性值")

iii. 元素.hasAttrbiute("属性名")

iv. 元素.removeAttribute("属性名")

2). 新HTML DOM 简写:

i. 元素.属性名

ii. 元素.属性名="属性值"

iii. 元素.属性名!==""

iv. 元素.属性名=""

b. bool类型的HTML标准属性: 只能用 "元素.属性名" 操作，~~不能用核心DOM4个函数~~

c. 自定义扩展属性: HTML5标准

1). 添加/修改自定义扩展属性:

i. HTML中: <元素 data-自定义属性名="属性值"

ii. 旧js中: ~~不能用.,~~ 只能用 元素.setAttribute("data-自定义属性名","属性值")

iii. HTML5标准: 元素.dataset.自定义属性名=属性值

2). 获取自定义扩展属性值:

i. 旧js中: ~~不能用.,~~ 只能用 元素.getAttribute("data-自定义属性名")

ii. HTML5标准: 元素.dataset.自定义属性名

3). 查找带有自定义扩展属性的元素: 属性选择器 [data-自定义属性=属性值]

(3). 样式:

a. 只修改一个css属性时: 元素.style.css属性="属性值"

b. 批量修改多个css属性时: 元素.className="class名"

c. 获取css属性值时: var style=getComputedStyle(元素对象)

style.css属性名

## 3. 添加元素:

(1). 只添加一个元素: 3步

a. 创建一个空元素

var 一个元素=document.createElement("元素名")

b. 为元素设置关键属性

新元素.属性名=属性值

c. 将新元素添加到指定的父元素下: 3种:

1). 末尾追加: 父元素.appendChild(新元素)

2). 插入到一个元素前: 父元素.insertBefore(新元素, 现有元素)

3). 替换现有元素: 父元素.replaceChild(新元素, 现有元素)

(2). 优化: 如果添加多个元素，尽量减少操作DOM树的次数

a. 如果同时添加父元素和子元素，应该先将子元素添加到父元素，最后再一次性将父元素添加到DOM树

b. 如果父元素已经在页面上，需要同时添加多个平级子元素，应该用文档片段: 3步

1). 先创建文档片段对象

var frag=document.createDocumentFragment();

2). 先将多个平级子元素加入到文档片段中

frag.appendChild(子元素)

3). 最后将文档片段添加到DOM树

父元素.appendChild(frag)

## 4. 删除元素: 父元素.removeChild(要删除的元素)

## 5. 事件:

(1). 事件绑定: 3种:

a. <元素 on事件名="事件处理函数()">

function 事件处理函数(){ ... }

b. 一个元素的一个事件属性上，只绑定一个事件处理函数，不会动态增加或减少:

元素.on事件名=function(){ ... }

c. 一个元素的一个事件上，需要同时绑定多个事件处理函数，还可能动态添加或移除

元素.addEventListener("事件名", 事件处理函数)

元素.removeEventListener("事件名", 原事件处理函数)

(2). 事件模型: 3个阶段: 捕获、目标触发、冒泡

(3). 获得事件对象: 定义事件绑定时: 元素.on事件名=function(e){ //触发事件时e=event }

(4). 取消冒泡/停止蔓延: e.stopPropagation()

(5). 只要多个平级子元素都要绑定相同的事件时: 利用冒泡/事件委托: 3步

a. 事件绑定在父元素上一份

b. 处理函数中，用e.target代替this获得实际点击的那个子元素

c. 处理函数中，先验证e.target的特征是想要的元素，才继续执行后续操作。

(6). 阻止元素自带的默认行为: e.preventDefault();

(7). 获得鼠标位置：

a. 鼠标位置相对屏幕左上角的距离: e.screenX e.screenY

b. 鼠标位置相对浏览器文档显示区左上角的距离 e.clientX e.clientY

c. 鼠标位置相对事件所在元素左上角的距离e.offsetX e.offsetY

(8). 窗口滚动事件:

a. 当滚动条滚动时自动触发: window.onscroll=function(){ ... }

b. 获取滚动条滚动过的距离:

var scrollTop=document.body.scrollTop||document.documentElement.scrollTop;

c. 控制滚动条滚动到指定位置: window.scrollTo(0, 要滚动到的位置)

## 总结: this 5种:

1. obj.fun() this->obj

2. new Fun() this->new正在创建的子对象

3. 类型名.prototype.共有方法=function(){ ... } this->将来调用这个共有方法的.前的子对象

4. fun() 和 (function(){ })() 和回调函数中的this->window

5. 事件处理函数中的this->当前正在触发事件的这个元素对象

# JQery

**day01**

今日扩展视频**：**

小程序->在线->DOM->day04

5. 鼠标坐标x y 原生DOM拖拽效果

作业: 原生DOM页面滚动动画效果

作业: 学子商城DOM版 中 day02最后——放大镜

赠送：原生DOM轮播图

作业: 高频笔试题nodejs浏览器不同事件循环

正课:

1. 什么是jQuery

2. 如何使用jQuery  
3. jQuery的原理

4. 查找元素

# 一. 什么是jQuery:

对应小程序视频: 小程序->在线->JQUERY->day01 1. 什么是jQuery，如何使用jQuery，CDN

1. 什么是: 第三方开发的执行DOM操作的极简化的函数库

(1). 第三方: 需要先下载才能用！——不是原生，不能直接用！

(2). 执行DOM操作: jQuery还是执行DOM增删改查+事件绑定操作，学习jQuery还是在学习DOM。并不是一项全新的知识。

(3). 极简化: jQuery对DOM操作的每一步代码都进行了简化！

(4). 函数库: jQuery中用函数来解决一切问题！jQuery中没有属性的！所有jQuery操作都要加上()，因为都是函数调用！

2. 为什么: 2个原因:

(1). 简单

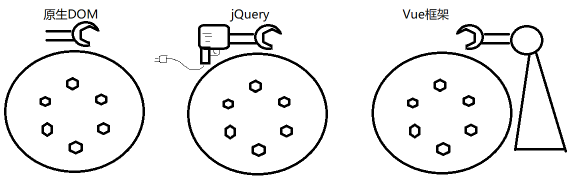
(2). 解决了大部分浏览器兼容性问题: 凡是jQuery让用的，都没有兼容性问题！

3. 何时: jQuery —— 用的越来越少了。除了旧项目维护还在用，新项目用的很少了！

4. 问题: 2个

(1). 只有PC端，没有移动端！

(2). 仅仅简化了DOM的每一步代码，但是没有从根本上简化开发流程，依然包含大量重复的低效率的代码！



# 二. 如何使用:

对应小程序视频: 小程序->在线->JQUERY->day01 2. 第一个jQuery程序 ...

1. 下载并引入页面: 2种:

(1). 去官网将jquery.js文件，下载到项目本地文件夹和项目源代码放在一起

a. 官网: jquery.com

b. 版本:

1.x 唯一兼容旧浏览器(IE8)

jquery-1.11.3.js

未压缩:

1). 拥有完备的注释，代码格式和见名知意的变量名——可读性好，但是体积大！

2). 适用于学习和开发之用。不适合于生产环境上线之用。因为体积大，客户端下载慢

压缩:

1). 删除了所有注释和代码格式，极简化了变量名——可读性差，但是体积小！

3). 不适用与学习和开发，更适用于生产环境上线之用。因为体积小，客户端下载快！

2.x 不再兼容旧浏览器，但是不支持ES6

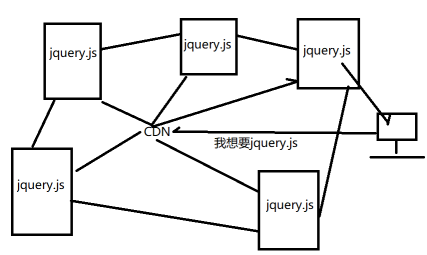
3.x不再兼容旧浏览器，开始支持ES6，比如for of ...

c. 引入页面: <script src="js/jquery-1.11.3.js">

d. 缺点: js文件放在我们自己的服务器上，因为我们自己的服务器始终只在一个地点，距离近的用户访问快！如果用户距离我们服务器很远，则访问就会慢。

(2). 用CDN网络中共享的js文件:

a. CDN网络: 全球共享的可根据客户端距离某一台服务器的远近自动为客户端分配最优的服务器下载资源的智能的内容分发网络。



b. 何时: 如果我们使用全球公认的广泛使用的第三方js时，都可用CDN网络中的js文件，而不用下载到本地！

c. 优点: 无论用户身处何地，都可以从离自己最近的服务器下载资源，保证了最快的下载速度

d. 如何:

1). 不用下载

2). 用官网提供的CDN网址，直接引入网页

<script

src="https://code.jquery.com/jquery-1.12.4.js"

integrity="sha256-Qw82+bXyGq6MydymqBxNPYTaUXXq7c8v3CwiYwLLNXU="

crossorigin="anonymous"></script>

2. 示例: 我的第一个jQuery程序: 记录按钮点击次数

0\_DOM.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  </head>  <body>  <h1>jQuery/$工厂函数</h1>  <button id="btn1">click me(0)</button>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: 点击id为btn1的按钮触发变化，所以查找id为btn1的元素  var btn = document.getElementById("btn1");  //2. 绑定事件处理函数  //本例中: 单击按钮触发事件，所以为按钮绑定单击事件  btn.onclick = function () {  //3. 查找要修改的元素  //本例中: 单击按钮时，就是要修改自己: this  //4. 修改元素  //4.1 先取出当前按钮的内容中()中的数量，转为整数  var n = parseInt(  this.innerHTML  // "click me(10000)"  // 012345678901234  // | |  .slice(9, -1) //复习第一阶段slice函数用法  // "10000"  );  //4.2 将数量+1  n++;  //4.3 将新数量拼到()中，放回当前按钮的内容中  this.innerHTML = `click me(${n})`;  }  </script>  </body>  </html>  运行结果: |

1\_jQuery.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  </head>  <body>  <h1>jQuery/$工厂函数</h1>  <button id="btn1">click me(0)</button>  <script src="js/jquery-1.11.3.js"></script>  <script>  //jQuery还是按照  //DOM 4步  //1. 查找触发事件的元素  //本例中: 点击id为btn1的按钮触发变化，所以查找id为btn1的元素  //var btn=document.getElementById("btn1");  // new jQuery("#btn1"); //2件事儿:创建新子对象和查找DOM元素  // 等效于 ↑  var $btn = $("#btn1");  //2. 绑定事件处理函数  //本例中: 单击按钮触发事件，所以为按钮绑定单击事件  //btn.onclick=function(){  //自动 ↑ 翻译为  $btn.click(function () {  //3. 查找要修改的元素  //本例中: 单击按钮时，就是要修改自己: this  //$btn.click()自动翻译为button.onclick=function(){}  //所以function中的this->当前点击的DOM按钮元素  //this不能直接调用jquery家函数  //只能$(this)，才能创建一个包含this的jquery家子对象，才能调用简化版函数  var $this = $(this);  //4. 修改元素  //4.1 先取出当前按钮的内容中()中的数量，转为整数  var n = parseInt(  //this.innerHTML  //自动 ↑ 翻译为  $this.html()  // "click me(10000)"  // 012345678901234  // | |  .slice(9, -1) //复习第一阶段slice函数用法  // "10000"  );  //4.2 将数量+1  n++;  //4.3 将新数量拼到()中，放回当前按钮的内容中  //this.innerHTML=`click me(${n})`;  //自动 ↑ 翻译为  $this.html(`click me(${n})`);  })  </script>  </body>  </html>  运行结果: |

# 三. jQuery原理:

1. 引入<script src="js/jquery-1.11.3.js">

其实引入了一种新的类型:

(1). 构造函数: function jQuery("选择器"){

...

}

(2). 原型对象: jQuery.prototype={ 保存着所有jQuery简化版函数 }

2. 问题1: 如果想调用jQuery原型对象中保存的简化版函数，必须先创建jQuery类型的子对象

3. 解决: var btn=new jQuery("... ...")

结果: btn.click() btn.html() ... ...

强调: 只有jQuery家子对象，才能调用jQuery家简化版函数

DOM元素无法调用jQuery家简化版函数

jQuery家孩子也无法直接调用DOM家原生函数

——不是一家人不进一家门

4. 问题2: 因为jQuery原型对象中所有简化版函数几乎都是要操作DOM元素的！

比如: click() 代替了 onclick= 专门为元素绑定单击事件

html() 代替了 innerHTML 专门获取或修改元素的内容

但是，jQuery子对象怎么知道自己要操作哪个DOM元素呢?

5. 解决: jQuery()构造函数有一个选择器参数

(1). 当new jQuery("选择器")时，可传入任意一个css选择器

(2). new jQuery(): 2件大事儿

a. 和普通new一样，创建子对象，继承父对象，添加自有属性...

b. 还会自动用传入的选择器去DOM树上查找符合条件的DOM元素对象，保存进jQuery子对象中

6. 结果: 对jquery子对象调用简化版函数，简化版函数都会自动翻译为原生DOM对应的操作，并作用于当前jquery子对象中保存的DOM元素对象上

7. 问题: 每次想调用jQuery家简化版函数时，都要写new jQuery("...")，太麻烦

8. 解决: 2件事:

(1). 将new jQuery()这句话，封装在了jQuery构造函数内部

结果: 创建jQuery子对象时，不写new，也等效于new jQuery()了！

(2). 祖师爷觉得每次都写jQuery()，也太麻烦了！

$=jQuery //给jQuery起一个外号/别名, $

(3). 结果: 从此$()=jQuery()=new jQuery()

9. 总结: 今后只要想使用jQuery家的简化版函数，都要先创建jQuery家子对象，并用选择器查找要操作的DOM元素。

var $jquery对象=$("选择器") —— 所有jQuery操作的第一步

10. 问题: 如果jQuery子对象和DOM元素对象采用相同的命名规则，则很难区分到底是谁家孩子！

11. 解决: 行规: 只要是jQuery家子的对象名，必须以$开头

12. 问题: 用jQuery绑定的事件处理函数中this->?

13. 原理: 因为$btn.click(function(){ ...this... }) ——表面

会被自动 ↓ 翻译为

button.onclick=function(){ —— 本质

...this->当前正在单击的DOM按钮元素对象

}

14. 问题: this->DOM按钮，不能继续使用jQuery家简化版函数

15. 解决: 其实jQuery构造函数除了支持选择器查找之外，还支持直接将一个DOM元素放入jQuery子对象中，无需查找。所以，将来如果一个DOM元素，也想用jQuery函数，可以用$(DOM元素)包裹以下就变成了jQuery家对象了！

function jQuery("选择器" 或 一个DOM元素){

... ...

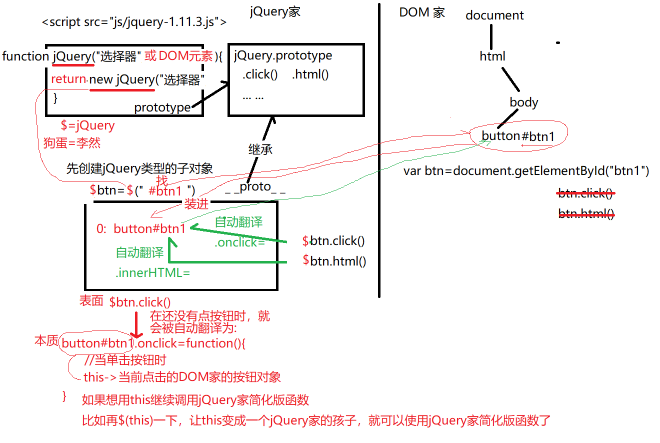
}

所以: var $btn=$("#btn1")

var $btn=$(this) this->点击的DOM按钮对象

16. 问题: 为什么必须把$(this)包裹以下用简化版函数？难道就不能this直接调用原生DOM函数吗？

17. 结论: 今后使用任何框架，最忌讳混搭！一定要专一！不同框架原理各不相同，返回值，参数，用法各不相同！如果混用！很容易懵逼！

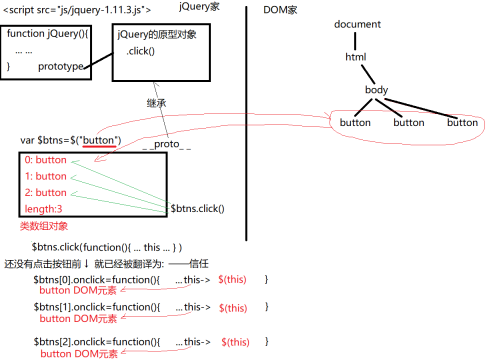


18. jQuery对象本质: 类数组对象，其中保存着所有找到的DOM元素对象

19. jQuery简化版函数三大特点:

(1). 自带for循环遍历:

a. 对整个jQuery对象调用一次简化版函数，等效于对其中每一个DOM元素对象分别调用一次简化版函数！



(2). 一个函数两用:

a. 凡是和修改(内容、属性、样式)有关的函数，都是一个函数两用

b. 一个函数，既能获取值，又能修改值

c. 通过重载方式判断本次该获取，还是该修改:

1). 如果调用函数时，没有传入新值，则函数默认执行获取旧值的操作

2). 如果调用函数时，传入了新值，则函数改为执行修改值的操作！

20. 示例: 多个按钮，点哪个按钮，就记录自己的点击次数

2\_jQueryAPI.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  </head>  <body>  <h1>jQueryAPI特点</h1>  <button id="btn1">click me(0)</button>  <button id="btn2">click me(0)</button>  <button id="btn3">click me(0)</button>  <script src="js/jquery-1.11.3.js"></script>  <script>  //为三个按钮绑定单击事件，记录单击次数  //DOM 4步  //1. 查找触发事件的元素  //本例中: 三个按钮都可以单击  var $btns = $("button");  console.log($btns);  //2. 绑定事件处理函数  //本例中: 每个按钮都能点，但是只需要对整个jQuery对象调用一次简化版函数即可  $btns.click(function () {  //alert("疼！");  //希望点那个按钮，让哪个按钮变成画  //this.innerHTML="❀";  //$(this).html("❀")  //3. 查找要修改的元素  //本例中: 单击按钮时，就是要修改自己: this  var $this = $(this);  //4. 修改元素  //4.1 先\*取出\*当前按钮的内容中()中的数量，转为整数  var n = parseInt(  $this.html() //()中没有传入新内容,是空的，所以html()本次执行获取  // "click me(10000)"  // 012345678901234  // | |  .slice( 9, -1) //复习第一阶段slice函数用法  // "10000"  );  //4.2 将数量+1  n++;  //4.3 将新数量拼到()中，\*修改\*回当前按钮的内容中  $this.html(`click me(${n})`);  //()中传入了新内容,不是空的，所以html(xxx)本次执行修改内容的操作  })  //button#btn1.onclick=function(){ this->button#btn1 }  //button#btn2.onclick=function(){ this->button#btn2 }  //button#btn3.onclick=function(){ this->button#btn3 }  </script>  </body>  </html>  运行结果: |

# 四. 查找元素:

1. jQuery支持所有CSS选择器: $("任意css选择器")

2. jQuery还新增了jQuery专属的个别选择器——仅jQuery可用！css不可用！

3. 基本过滤选择器:

(1). 回顾: CSS中的子元素过滤选择器:

a. 什么是: 按元素在其所在的父元素下的相对位置选择某个子元素

b. 包括: :first-child(可能选择多个) :last-child(可能选择多个) :nth-child(i) :only-child

c. 示例: 使用子元素过滤，选择指定元素

5\_child filter.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8" />  <title></title>  </head>  <body>  <h3>子元素过滤选择器.</h3>  <ul>  <li>child1-basic0</li>  <li>child2-basic1</li>  <li>child3-basic2</li>  </ul>  <ul>  <li>child1-basic3</li>  <li>child2-basic4</li>  <li>child3-basic5</li>  </ul>  <ul>  <li>child1-basic6</li>  </ul>  <script src="js/jquery-1.11.3.js"></script>  <script>  //查找每个ul中第一个li，让其边框变为红色  $("ul>li:first-child").css("border","1px solid red");  //查找每个ul中最后一个li，给其加绿色阴影  $("ul>li:last-child").css("box-shadow","0 0 5px green");  //查找每个ul中处于偶数位置的li,让其字体变为红色  $("ul>li:nth-child(2n)").css("color","red");  //查找每个ul中第2个li,修改其背景为黄色  $("ul>li:nth-child(2)").css("background-color","yellow")  //查找作为ul下唯一子元素的li  $("ul>li:only-child").css("padding","15px")  </script>  </body>  </html>  运行结果: |

(2). 基本过滤选择器:

对应小程序视频: 小程序->在线->JQUERY->day01 3. 子元素过滤child filter 。。。

a. 什么是: 先将所有符合条件的元素提取到一个大的集合中统一编号，然后按大排名的顺序，选择元素！——下标从0开始！

b. 包括:

1). :first 整个查询结果中第一个元素

2). :last 整个查询结果中最后一个元素

3). :even 整个查询结果中偶数下标位置的元素

偶数 4个字母

4). :odd 整个查询结果中奇数位置的元素

奇数 3个字母

5). :eq(i) text/plain素

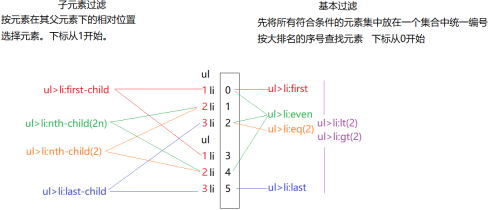
equal 等于

6). :gt(i) 整个查询结果中下标>i位置的元素

greater than > 大于

7). :lt(i) 整个查询结果中下标<i位置的元素

less than < 小于



c. 示例: 使用基本过滤，选择指定元素

6\_basic filer.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8" />  <title></title>  </head>  <body>  <h3>基本过滤选择器.</h3>  <ul>  <li>child1-basic0</li>  <li>child2-basic1</li>  <li>child3-basic2</li>  </ul>  <ul>  <li>child1-basic3</li>  <li>child2-basic4</li>  <li>child3-basic5</li>  </ul>  <ul>  <li>child1-basic6</li>  </ul>  <script src="js/jquery-1.11.3.js"></script>  <script>  //查找所有li中第一个li  $("ul>li:first").css("border","1px solid red")  //查找所有li中最后一个li  $("ul>li:last").css("box-shadow","0 0 5px green")  //查找处于正常人眼里的偶数位置的li  $("ul>li:odd").css("background-color","yellow");  //查找处于下标为偶数位置的li  $("ul>li:even").css("color","red")  //查找正常人眼里第2个li  $("ul>li:eq(1)").css("font-weight","bold")  //查找下标为2位置的li  $("ul>li:eq(2)").css("text-decoration","underline")  </script>  </body>  </html>  运行效果: |

(3). 示例: 按钮组

对应小程序视频: 小程序->在线->JQUERY->day01 作业:分别使用css和jquery实现按钮组效果

7\_basic filter2.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8" />  <title>...</title>  <style>  /\* 清除ul圆点，并所有li浮动 \*/  ul{  list-style:none  }  ul>li{  float:left;  padding: 5px 10px;  border:1px solid #555  }  /\* 纯css实现 \*/  /\* 设置第一个li的左上和左下圆角 \*/  /\* ul>li:first-child{  border-radius:5px 0 0 5px  } \*/  /\* 设置最后一个li的右上和右下圆角 \*/  /\* ul>li:last-child{  border-radius:0 5px 5px 0  } \*/  /\* 清除第一个li之后所有li兄弟的左边框 \*/  /\* ul>li~li{  border-left:0;  } \*/  </style>  </head>  <body>  <h1>实现按钮组效果</h1>  <ul>  <li>1</li>  <li>2</li>  <li>3</li>  <li>4</li>  </ul>  <script src="js/jquery-1.11.3.js"></script>  <script>  /\* 设置第一个li的左上和左下圆角 \*/  $("ul>li:first").css("border-radius","5px 0 0 5px")  /\* 设置最后一个li的右上和右下圆角 \*/  $("ul>li:last").css("border-radius","0 5px 5px 0")  /\* 清除第一个li之后所有li兄弟的左边框 \*/  $("ul>li:gt(0)").css("border-left",0)  </script>  <div style="clear:both"></div>  <h1>实现表格隔行变色效果</h1>  </body>  </html>  运行效果： |

|  |
| --- |
| **强调**: 如果将来一个效果即可用css做，又可以用js做！  1. 一定首选CSS:  (1). 效率高！  css是归计算机绘图引擎解析，因为绘图引擎只负责绘图，所以效率高！  js是归计算机程序引擎解析，因为计算机不止运行js程序！还同时运行着其他很多程序。js不得不需要跟其他程序共用计算机的资源——效率低  (2). 便于迁移和维护：  今后框架中，对js的修改极大！现在的js，DOM，jQuery，到了框架中几乎全都不能用！都需要按框架要求重新写！但是只有css，框架不管！移植时，只要将css拷贝到框架中，无需修改照样正常运行！  2. 当需要人为操作交互效果时，就只能用js做。 |

尝鲜: 修改css属性:   
1. DOM中: 元素.style.css属性="属性值"

2. jq中: $(...).css("css属性","属性值")

# Day02

今日扩展视频：

小程序->在线->JQUERY->day02

作业: jQuery单击下拉菜单

作业: jQuery带多种动画效果的对话框

直接在day02/5\_class2.html中做即可！

12. jQuery事件 事件委托

正课:

1. 查找

2. 修改

3. 按节点间关系查找

4. 添加/删除/替换/克隆

5. 事件绑定

# 一. 查找: jQuery新增选择器

对应小程序视频: 小程序->在线->JQUERY->day01 4. jQuery其它选择器...

1. 内容过滤选择器:

(1). 什么是: 根据元素的内容中包含的关键词来匹配元素

(2). 何时: 如果用其它选择器很难区分元素时，就可以尝试用元素的内容不同来选择元素

(3). 包含: 4种:

a. :contains(关键词) 匹配元素内容中包含指定"关键词"的元素

b. :has(选择器) 匹配子元素中包含符合选择器要求的元素 的父元素

根据孩子特征，选爹

c. :parent 匹配所有内容不为空的元素

d. :empty 匹配内容为空的元素

强调: 空元素: 连空字符内容都不能有！

(4). 示例: 使用内容过滤选择器选择指定元素

day01 剩余/10\_content filter.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  <style>  div {  margin: 10px auto;  }  </style>  </head>  <body>  <div class="container">  <h1>jQuery中的选择器——内容过滤选择器</h1>  <button>提交订单</button>  <button>Submit注册信息</button>  <button>马上提交</button>  <button>清空重填</button>  <hr />  <div class="alert" id="alert1">  第一个警告框  </div>  <div class="alert" id="alert2">  <span class="close">×</span>  第一个警告框  </div>  <div class="alert" style="height:30px"></div>  </div>  <script src="js/jquery-1.11.3.js"></script>  <script>  //选择包含"提交"二字的按钮，变为绿色按钮  $(".container>:contains(提交)")  .css("background-color", "lightGreen");  //选中包含.close按钮的.alert元素，让它们变为红色的警告框  $(".alert:has(.close)").css("background-color", "pink")  //选中不包含.close按钮的.alert元素，让它们变为绿色的警告框  $(".alert:not(:has(.close))")  .css("background-color", "lightGreen");  //希望给内容为空的div加红色边框  $(".alert:empty").css("border", "1px solid red");  //希望给有内容的div加绿色边框  $(".alert:parent").css("border", "1px solid green");  </script>  </body>  </html>  运行结果: |

2. 可见性过滤:

(1). 什么是: 根据元素是否可见来选择元素

(2). 包括:

a. :visible 专门匹配可见的元素

b. :hidden 专门匹配不可见的元素

坑: 只能选择display:none和type="hidden"隐藏的元素

~~无法选择visibility:hidden和opacity:0的元素~~

(4). 示例: 使用:hidden查找隐藏的元素

day01 剩余/12\_visibility filter.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  <style>  </style>  </head>  <body>  <div class="container">  <h1>可见性选择器</h1>  <div id="d1" style="display: none">lorem</div>  <div id="d2" style="visibility: hidden">lorem</div>  <div id="d3" style="opacity: 0">lorem</div>  <input id="d4" type="hidden" name="aa" value="bb" />  </div>  <script src="js/jquery-1.11.3.js"></script>  <script>  //想选择class为container下的隐藏的元素  var $elems=$(".container>:hidden");  console.log($elems);//4个元素  </script>  </body>  </html>  运行结果: |

3. 表单元素过滤选择器:

(1). **:**input 专门选择所有表单元素(input 、button、textarea 、select)

input 只是一个普通的元素选择器，只能选择input元素。不能选择其他类型的表单元素

(2). 问题: input元素就有很多种类，如何精确的只选择其中一种种类的input呢？

(3). 解决: 其实，input元素的每种type值，都对应着一个专门的选择器，比如:

a. :text 专门选择<input type="text"

b. :password 专门选择<input type="password"

c. :radio 专门选择<input type="radio"

... ...

(4). 示例: 点同意，启用表单元素；不同意，就禁用表单元素

day01 剩余/13\_form state selector.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>.....</title>  <meta charset="utf-8" />  <style>  </style>  </head>  <body>  <form>  用户名:<input disabled></br>  密码:<input type="password" disabled></br>  <input type="checkbox">我同意本站的使用条款<br>  <input type="submit" value="提交注册信息" disabled />  </form>  <script src="js/jquery-1.11.3.js"></script>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: 用户点type=checkbox的复选框触发启用禁用状态的改变  //应该查找type=checkbox的一个复选框  $(":checkbox")  //2. 绑定事件处理函数  //本例中:单击复选框触发变化  .click(function(){  //3. 查找要修改的元素  //本例中: 点了复选框要修改除当前复选框之外其余所有表单元素  var $others=$(":input:not(:checkbox)")  //4. 修改元素  //获得当前checkbox的选中状态  //错误: jQuery是函数库，不能用属性方式  //var checked=$(this).checked;  // 当前复选框 获取 checked属性的值  var checked=$(this).prop("checked");  //因为checked属性值只是一个bool值，不是jQuery家子对象  //所以变量名不加$  //如果checkbox是选中的  if(checked==true){  //其它表单元素启用  //修改其他表单元素的disabled属性值为false  $others.prop("disabled",false);  }else{//否则如果checkbox是未选中的  //其它表单元素禁用  //修改其他表单元素的disabled属性值为true  $others.prop("disabled",true);  }  //基础好的同学，必须改为1句话！  //如果选中(checked为true)，则其它元素启用(disabled为false)  //否则如果未选中(checked为false)，则其它元素禁用(disabled为true)  //所以，只要让其他元素的disabled值和checked的值相反就行！不用三目！  })  </script>  </body>  </html>  运行结果: |

# 二. 修改: 3种东西

## 1. 内容: 3种:

对应小程序视频: 小程序->在线->JQUERY->day02 1. jquery 修改元素的内容(有扩展)

(1). 原始的HTML内容:

a. DOM中: 元素.innerHTML

b. jq中: $元素.html("新内容")

(2). 纯文本内容:

a. DOM中: 元素.textContent

b. jq中: $元素.text("新内容")

(3). 表单元素的值:

a. DOM中: 元素.value

b. jq中: $元素.val("新值")

(4). 示例: 通过修改元素内容，实现表单验证

1\_html\_val.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title> new document </title>  <meta charset="utf-8">  </head>  <body>  <h1>操作元素的内容和值</h1>  <form action="">  用户名:<input type="text" name="uname">  <span></span><br>  密码:<input type="password" name="upwd">  <span></span><br>  <input type="submit" value="提交注册信息">  </form>  <script src="js/jquery-1.11.3.js"></script>  <script>  //正确时，使用图片:"<img src='img/ok.png'>"  //姓名错误时: "<img src='img/err.png'>用户名必须介于3~9位之间!"  //密码错误时: "<img src='img/err.png'>密码必须介于6~8位之间!"  //DOM 4步  //1. 查找触发事件的元素  //本例中: 当姓名文本框失去焦点时，触发验证和变化  //应该找文本框:  $(":text")  //2. 绑定事件处理函数  //本例中: 失去焦点触发验证和变化  .blur(function(){  var $txt=$(this);  //3. 查找要修改的元素  //本例中: 不管验证通过还是未通过，都要修改文本框旁边的下一个span  var $span=$txt.next();  //4. 修改元素  //获得文本框中内容去掉空字符后的长度  var length=$txt.val().trim().length;  //如果长度介于3~9位字符之间，就验证通过  if(length>=3&&length<=9){  //就设置span的内容为<img src='img/ok.png'>  $span.html(`<img src='img/ok.png'>`)  }else{//否则如果验证不通过  //就设置span的内容为<img src='img/err.png'>用户名必须介于3~9位之间!  $span.html(`<img src='img/err.png'>用户名必须介于3~9位之间!`);  }  });  //自己模仿我的姓名文本框，制作密码框验证  </script>  </body>  </html>  运行结果: |

## 2. 属性: 3种:

对应小程序视频: 小程序->在线->JQUERY->day02 2. jquery修改元素属性...

(1). 字符串类型的HTML标准属性:

a. DOM中: 2种:

1). 核心DOM 4个函数: 元素.getAttribute() setAttribute() ...

2). HTML DOM: 元素.属性名

b. jq中: 2种:

1). $元素.attr("属性名", "新值") 代替元素.getAttribute() setAttribute() ...

attribute指 HTML中开始标签中的属性

一个函数两用

2).问题: jQuery是函数库，不能用".属性名"方式访问

解决: jQuery中封装了一个新函数.prop("属性名", "新值")，专门代替".属性名"方式来获取或修改元素的属性值

property 指 内存中元素对象上的属性

一个函数两用

(2). bool类型的HTML标准属性

a. DOM中: 元素.属性名， ~~不能用核心DOM getAttribute()和setAttribute()~~

d. 如何: $元素.prop("属性名", bool) 代替".属性名"

~~c. 因为在DOM中不能用getAttribute()和setAttribute()，所以在jq中也不能用attr()~~

(3). 自定义扩展属性:

a. DOM中: 2种:

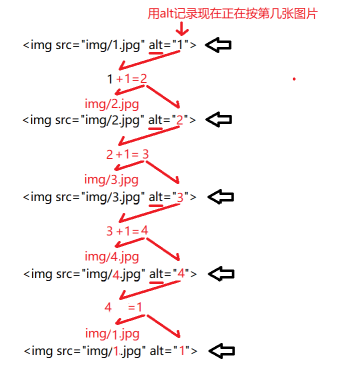
1). 核心DOM4个函数: 元素.getAttribute() 元素.setAttribute()

2). HTML5: 元素.dataset.自定义属性名

b. jq中: 1种: $元素.attr() 代替 元素.getAttribute() 元素.setAttribute()

~~c. 强调: 因为自定义属性在DOM中就不能用".属性名"方式访问，所以jq中自然也就不能用.prop()来访问！因为prop()代替的就是".属性名"~~

(4). 示例: 点击图片切换下一张



2\_attr.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title> new document </title>  <meta charset="utf-8">  </head>  <body>  <h1>操作元素的属性</h1>  <img src="img/1.jpg" alt="1">  <script src="js/jquery-1.11.3.js"></script>  <script>  //单击图片，切换下一张  //DOM 4步  //1. 查找触发事件的元素  //本例中: 用户点img触发变化  $("img")  //2. 绑定事件处理函数  //本例中: 单击触发变化  .click(function(){  //3. 查找要修改的元素  //本例中: 就是要修改当前img自己  var $img=$(this);  //4. 修改元素  //本例中:  //先获取当前img中alt属性的值转为整数  var i=parseInt($img.attr("alt"));  // .prop("alt")  if(i<4){//如果alt值<4，则alt+1  i++;  }else{//否则如果alt值==4，则alt=1  i=1;  }  //最后将alt的值拼成新的src路径，设置到img的src属性上，同时，将alt值再放回img的alt属性中，为下次点击做准备  //$img.attr("src",`img/${i}.jpg`);  // .prop()  //$img.attr("alt",i);  // .prop()  $img.attr({ src:`img/${i}.jpg`, alt:i })  // .prop()  })  </script>  </body>  </html>  运行结果: |

(5). 示例: 点击小图片，切换大图片

3\_attr2.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  <style>  body {  text-align: center;  }  </style>  </head>  <body>  <img src="img/1.jpg" data-target="img/1-l.jpg" class="my-small">  <img src="img/2.jpg" data-target="img/2-l.jpg" class="my-small">  <img src="img/3.jpg" data-target="img/3-l.jpg" class="my-small">  <img src="img/4.jpg" data-target="img/4-l.jpg" class="my-small">  <hr />  <img src="img/1-l.jpg" class="my-big">  <script src="js/jquery-1.11.3.js"></script>  <script>  //点击小图片，下方my-big中显示大图片  //DOM 4步  //1. 查找触发事件的元素  //本例中: 用户点class为my-small的img元素触发变化  $(".my-small")  //2. 绑定事件处理函数  //本例中: 单击触发变化  .click(function(){  //3. 查找要修改的元素  //本例中: 要修改class为my-big的img  $(".my-big")  //4. 修改元素  //本例中: 要修改my-big的src属性为当前点击的小图片上的data-target属性值！  // 当前小图片 获取 data-target属性值  .attr("src",$(this).attr("data-target"))  //.prop() 不能改.prop()  })  </script>  </body>  </html>  运行结果： |

## 3. 样式:

对应小程序视频: 小程序->在线->JQUERY->day02 3. jQuery修改样式 css()...

(1). DOM中:

a. 如果只修改一个css属性: 元素.style.css属性=值

b. 获取一个元素的css属性: var style=getComputedStyle(元素)

c. 如果批量修改一个元素的多个css属性: 元素.className="class名"

(2). jq中:

a. 将修改css属性和获取css属性统一为一个: $元素.css("css属性名","属性值")

一个函数两用:

如果没传入新属性值，则自动执行读取旧属性值的操作，相当于getComputedStyle

如果传入新属性值，则自动切换为执行修改属性值的操作，相当于元素.style.css属性

b. 如果批量修改一个元素的多个css属性，依然使用class的方式。

1). 问题: DOM中的className如果只想操作一个class名而不影响当前元素上其它class名，非常不方便！因为className是一个完整的字符串，只能整体替换所有class。

2). 解决: jq定义两个4个函数，专门对class执行不同的操作：

i. $元素.addClass("class名")

添加class

ii. $元素.removeClass("class名")

移除class

iii. var bool=$元素.hasClass("class名")

判断是否包含某个class

iv. $元素.toggleClass("class名")

切换有没有一个class

①等效于:

if(有这个class hasClass()){

就移除class removeClass()

}else{

就添加class addClass()

}

②问题: 能不能用toggleClass完全代替前边的addClass和removeClass

答: 不能！addClass永远是添加class， removeClass永远是移除class，但是toggleClass没准: 一次添加，一次移除，再一次才是添加，再一次又是移除！

③总结: 只有确实在有和没有一个class之间来回切换时，才用toggleClass

(3). 示例: 双态按钮

4\_class.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  <style>  .btn {  padding: 5px 10px;  border-radius: 3px;  border: 1px solid #aaa;  outline: none;  }  /\* 按钮抬起时的默认样子 \*/  .up {  background: #fff;  color: #333;  }  /\* 按钮按下时的样子 \*/  .down {  background: #ddd;  color: #fff;  }  </style>  </head>  <body>  <button class="btn up">双态按钮</button>  <script src="js/jquery-1.11.3.js"></script>  <script>  //双态按钮: 让按钮的class在up和down之间切换  //DOM 4步  //1. 查找触发事件的元素  //本例中: 用户点按钮触发变化  $("button")  //2. 绑定事件处理函数  //本例中: 单击触发变化  .click(function(){  //3. 查找要修改的元素  //本例中: 要修改的就是当前按钮自己  //var $btn=$(this);  //4. 修改元素  //本例中: 如果有down class，就移除down class，否则如果没有down class，就添加down class  // if($btn.hasClass("down")){  // $btn.removeClass("down")  // }else{  // $btn.addClass("down");  // }  //其实就是在有和没有down这个class之间来回切换！  $(this).toggleClass("down");  })  </script>  </body>  </html>  运行结果: |

|  |
| --- |
| 简写: .attr()/.prop()/.css() 其实都可以一句话修改多个属性值  $元素.attr或prop或css({  属性名: 属性值,  ... : ...  })  如果css中有-连接符，样式名使用驼峰 |

# 三. 按节点间关系查找：

对应小程序视频: 小程序->在线->JQUERY->day02 6. jQuery按节点间关系查找

## 1. 父子关系:

(1). 获得一个元素的父元素: $元素.parent() 代替 元素.parentElement

(2). 获得一个元素下所有直接子元素: $元素.children() 代替 元素.children

a. 问题: children默认只能选择所有直接子元素，无法只选择个别想要的子元素

解决: $元素.children(选择器)， 意为可以有选择的挑选部分符合条件的元素

b. 问题: children默认只能在直接子元素中查找！无法在所有后代中查找

解决: $元素.find(选择器) 在所有后代元素中查找符合要求的元素

(3). 获得一个元素下第一个直接子元素: ~~元素.firstElementChild~~

$元素.children(":first-child");

(4). 获得一个元素下最后一个直接子元素: ~~元素.lastElementChild~~

$元素.children(":last-child");

## 2. 兄弟关系:

(1). 前一个兄弟:

a. DOM中: 元素.

b. jq中: $元素.prev()

c. 问题: 有时，不止找前一个元素，想找之前所有元素

解决: $元素.prevAll(选择器)

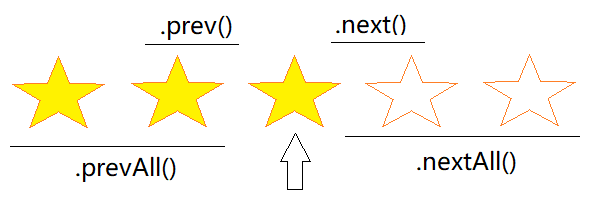
(2). 下一个兄弟:

a. DOM中： 元素.nextElementSibling

b. jq中: $元素.next();

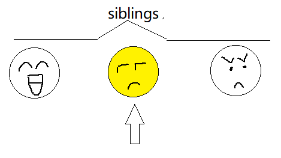
c. 问题: 有时，不止找下一个元素，想找之后所有元素

解决: $元素.nextAll(选择器)



(3). 问题: 有时，我们希望获得除当前元素之外其余所有兄弟元素（不论前后）！

解决: $元素.siblings("选择器")

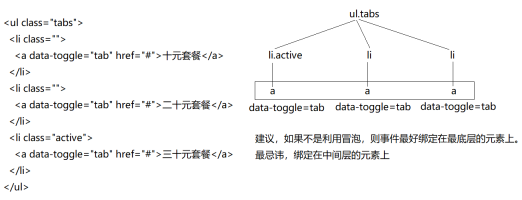


3. 示例: 使用按节点间关系查找，选择指定的元素

6\_traverse.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8" />  <title>...</title>  <script>  </script>  </head>  <body>  <!-- ul.top>(li.parent>ul>li.child\*3)\*2 -->  <ul class="top">  <li class="parent1">parent1  <ul>  <li class="child">child1</li>  <li class="child">child2</li>  <li class="child">child3</li>  </ul>  </li>  <li class="parent2">parent2  <ul>  <li class="child">child1</li>  <li class="child">child2</li>  <li class="child">child3</li>  </ul>  </li>  </ul>  <script src="js/jquery-1.11.3.js"></script>  <script>  //修改class为top的ul的所有直接子元素  $("ul.top").children().css("border","1px solid red")  //修改class为top的ul的所有后代li  $("ul.top").find("li").css("box-shadow","0 0 5px green")  //为class为child的li绑定单击事件  $("li.child").click(function(){  //选择当前元素的下一个元素/前一个元素/之前所有/之后所有/除自己之外所有  $(this).siblings()//.nextAll()//.prevAll()//.prev()//.next()  .css("background-color","yellow");  })  </script>  </body>  </html>  运行结果: |

4. 示例: 标签页效果



7\_tabs.html

|  |
| --- |
| <!doctype html>  <html lang="en">  <head>  <meta charset="UTF-8">  <style>  .tabs {  list-style: none;  padding: 0  }  .tabs a {  text-decoration: none;  color: #000;  padding: 6px 12px;  display: inline-block;  }  .tabs>li {  float: left;  border-bottom: 1px solid #000;  }  .tabs>.active {  border: 1px solid #000;  border-bottom: 0;  }  </style>  </head>  <body>  <h1>使用属性选择器实现标签页头的切换</h1>  <ul class="tabs">  <li class="active"><a data-toggle="tab" href="#">十元套餐</a></li>  <li><a data-toggle="tab" href="#">二十元套餐</a></li>  <li><a data-toggle="tab" href="#">三十元套餐</a></li>  </ul>  <script src="js/jquery-1.11.3.js"></script>  <script>  //DOM 4步  //1 查找触发事件的元素  //本例中: 用户点有data-toggle属性，且属性值为tab的元素触发变化  $("[data-toggle=tab]")  //2. 绑定事件处理函数  //本例中: 单击触发变化  .click(function(){  //3. 查找要修改的元素  //4. 修改元素  //本例中: 先给当前a的父元素li加active class，再去掉当前a的父元素里的兄弟身上的active class  $(this).parent()  // a 当前li  .addClass("active")//本来不需要返回值  //jQuery要求addClass必须返回.前的主语  //return $当前li  //addClass的返回值，刚巧和下一个操作的.前的主语一致！  .siblings().removeClass("active");  // 其它li  //jquery链式操作！  })  </script>  </body>  </html>  运行结果: |

# 四. 添加/删除/替换/克隆:

对应小程序视频: 小程序->在线->JQUERY->day02 7. jQuery添加删除元素

## 1. 添加&替换:

(1). 回顾: DOM中: 3步:

a. 创建新元素: var 新元素=document.createElement("标签名")

b. 设置关键属性: 元素.属性名=属性值

c. 将新元素添加到DOM树: 3种:

1). 父元素末尾追加新元素: 父元素.appendChild(新元素)

2). 插入到父元素下一个现有子元素之前: 父元素.insertBefore(新元素, 现有子元素)

3). 替换父元素下一个现有子元素: 父元素.replaceChild(新元素, 现有子元素)

(2). jq中: 2步:

a. 用一段HTML片段创建一个或多个元素:

var $新元素=$(`HTML代码片段`);

问题: 依然是在内存中游离的，没有在DOM树上！所以，页面上暂时看不到

解决: 挂载

b. 将新元素添加到DOM树: 5种情况，10个函数

**每个函数多出一个功能一模一样，因为主语不同，则函数返回的jquery对象不用。则后续可连接的链式操作也就不同！**

1). 父元素末尾追加新元素:

$父元素.append($新元素) 代替 父元素.appendChild(新元素)

兄弟函数: $新元素.appendTo($父元素)

2). 父元素开头插入新元素:

新增: $父元素.prepend($新元素)

兄弟函数: $新元素.prependTo($父元素)

3). 插入到一个现有子元素之前:

$现有元素.before($新元素) 代替 父元素.insertBefore(新元素, 现有子元素)

兄弟函数: $新元素.insertBefore($现有元素)

4). 插入到一个现有子元素之后:

新增: $现有元素.after($新元素)

兄弟函数: $新元素.insertAfter($现有元素)

5). 替换父元素下一个现有子元素:

$现有元素.replaceWith($新元素) 父元素.replaceChild(新元素, 现有子元素)

兄弟函数: $新元素.replaceAll($现有元素)

|  |
| --- |
| 为什么每个函数又多出一个功能一模一样，只是主语和宾语颠倒位置的新函数？  因为主语不同，则函数返回的jquery对象不用。则后续可连接的链式操作也就不同！  如果希望在追加一个新元素后，继续对新元素执行下一步操作：  不好的做法: $父元素.append($新元素)  return $父元素，下一步用不了！  $新元素.下一步操作  好的做法: $新元素.appendTo($父元素).下一步操作()  return $新元素 下一步可继续使用 |



## 2. 删除: $元素.remove()

3. 示例: 点按钮，添加方块，点x删除方块

9\_append.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title> new document </title>  <meta charset="utf-8">  <style>  .container {  border: 1px solid #aaa;  overflow: hidden;  }  .block {  float: left;  margin: 10px;  border: 1px solid #aaa;  background: #faa;  width: 150px;  height: 150px;  }  .block:hover {  box-shadow: 0 5px 6px #000;  }  .close {  float: right;  padding: 5px;  font-weight: bold;  opacity: .2;  cursor: pointer;  }  .close:hover {  opacity: .5;  }  </style>  </head>  <body>  <h1>添加/删除节点</h1>  <button id="add-block">添加区块</button>  <div class="container">  <!-- <div class="block">  <span class="close">×</span>  </div> -->  </div>  <script src="js/jquery-1.11.3.js"></script>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: 点击按钮添加方块  $("#add-block")  //2. 绑定事件处理函数  .click(function(){  // //3. 查找要修改的元素  // //本例中: 向class="container"的div开头插入一个新方块div元素  // var $container=$(".container");  // //4. 修改元素  // //本例中:  // //先创建一个新的方块元素对象  // var $block=$(`<div class="block">  // <span class="close">×</span>  // </div>`);  // //设置方块的随机背景色  // $block.css("background-color",`rgb(${  // parseInt(Math.random()\*256)  // },${  // parseInt(Math.random()\*256)  // },${  // parseInt(Math.random()\*256)  // })`);  // //最后将方块插入到$container开头  // $container.prepend($block);  //创建一个新方块  $(`<div class="block">  <span class="close">×</span>  </div>`)  //设置方块的随机背景色  .css("background-color",`rgb(${  parseInt(Math.random()\*256)  },${  parseInt(Math.random()\*256)  },${  parseInt(Math.random()\*256)  })`)  //return 新方块  //将方块插入到父元素开头  //.prependTo($(".container"))  //简写: 不用自己找.container，直接把选择器给prependTo，prependTo会自动查找选择器所指的父元素  .prependTo(".container");  })  //实现点x按钮删除当前方块  //DOM 4步  //1. 查找触发事件的元素  //本例中: 因为多个x都可单击，所以应该用事件委托优化——复习事件委托  //事件应该只绑定在父元素上一份即可  $(".container")  //2. 绑定事件处理函数  //本例中: 应该先获得事件对象e，才能用e.target获得目标元素x  .click(function(e){  //获得当前点击的元素  var $target=$(e.target);  //判断当前实际点的元素是否是想要的  //本例中: 只有点在class为close的span元素上，才能删除  if($target.hasClass("close")){ //说明我们点的是x  //3. 查找要修改的元素  //本例中: 点x，其实是要删除x的父元素div  $target.parent()  //4. 修改元素  //本例中: 删除父元素div  .remove();  }  })  </script>  </body>  </html>  运行结果: |

# Day03

今日扩展视频：

小程序->在线->JQUERY->day02

12. jQuery事件 事件委托

小程序->在线->JQUERY->day03

作业: jquery动画函数 手风琴效果

作业: 高频笔试题！防抖和节流

正课:

1. 克隆

2. 事件

3. 动画

## 3. 克隆:

(1). 什么是: 复制出一个一模一样的新的元素对象

(2). 如何: var $新元素副本=$原元素.clone()

(3). 示例: 选择主角飞机

day02 clone/10\_replace\_clone.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  <style>  body {  text-align: center;  }  </style>  </head>  <body>  <h1>替换节点</h1>  <div id="chosen">  <img src="img/p0.png">  </div>  <hr />  <div id="list">  <img src="img/p3.png">  <img src="img/p4.png">  <img src="img/p5.png">  </div>  <script src="js/jquery-1.11.3.js"></script>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: id为list下的所有img都可点击  //所以，应该用事件委托优化，事件应该只绑定在父元素上一份  $("#list")  //2. 绑定事件处理函数  //本例中: 单击图片触发变化  .click(function(e){  //用e.target代替this获得当前点击的img元素  var $target=$(e.target);  //验证$target是不是想要的  //本例中: 只有点在img元素上才能触发变化  //if($target是img){  //错误: jQuery是函数库，不能使用.属性  //if($target.nodeName=="IMG"){  //正确: jQuery用.prop()代替了.属性  //if($target.prop("nodeName")=="IMG"){  //新增: $元素.is("选择器")  // 元素选择器  if($target.is("img")){  // //3. 查找要修改的元素  // //本例中: 点击下方飞机图片，应该先将当前img复制一个副本，用副本去代替id为chosen的div下的img  // var $chosen=$("#chosen>img");  // //4. 修改元素  // $target.clone().replaceAll($chosen);  // //原img 副本img  //简写:  $target.clone().replaceAll("#chosen>img") //不用自己找，replaceAll自带查找功能！  }  })  </script>  </body>  </html>  运行结果: |

|  |
| --- |
| 新增:  var bool=$元素.is("选择器") 专门判断一个元素是否符合选择器的要求 |

# 二. 事件:

对应小程序视频: 小程序->在线->JQUERY->day02 8. jquery事件 事件委托 on off

## 1. 绑定事件:

(1). 回顾: DOM中: js中2种:

~~a. 赋值方式: 元素.on事件名=function(e){~~

this->当前正在触发事件的这个元素

}

b. 添加事件监听: 元素.addEventListener("事件名",事件处理函数)

移除事件监听: 元素.removeEventListener("事件名", 原事件处理函数)

(2). jq中: 1种

a. 标准写法: $元素.on("事件名", 事件处理函数)

$元素.off("事件名", 原事件处理函数)

b. 特殊:

1). DOM中: 有名的事件处理函数，只能添加一次，不能反复添加多次

2). jq中: 因为jQuery底层的特殊处理，所以$元素.on()即使使用有名称的函数，也可反复添加事件处理函数。移除时，用函数名可移除添加的多个同名事件处理函数

c. 简写: jQuery对十几种最常用事件，提供了更简化的简写方式

1). $元素.事件名(function(e){

this->当前正在触发事件的DOM元素

})

2). 比如: $元素.click(...) $元素.blur(...) ... ...

d. 总结: 哪些事件可简写:

|  |
| --- |
| 常用事件名:  blur 失去焦点  change 下拉列表选中项改变  click 单击  dblclick 双击  focus 获得焦点  keydown 键盘按键按下  keyup 键盘按键抬起  mousedown 鼠标按键按下  mouseenter 鼠标进入(jq)  mouseleave 鼠标移出(jq)  mousemove 鼠标移动  mouseout 鼠标移出(dom)  mouseover 鼠标进入(dom)  mouseup 鼠标按键抬起  resize 窗口大小改变  scroll 网页滚动 |

e. 其实:

~~$元素.click() 底层 自动翻译为 元素.onclick=function(){ ... }~~

$元素.click() 底层 自动翻译为 元素.addEventListener("click", ...)

(3). 示例: 点击按钮 发射普通子弹，点击获得奖励按钮，获得跟踪导弹，还可移除导弹

event/12\_bind\_unbind.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title> new document </title>  <meta charset="utf-8">  </head>  <body>  <h1>事件绑定</h1>  <button id="btn1">发射子弹</button>  <button id="btn2">获得奖励</button>  <button id="btn3">失去奖励</button>  <script src="js/jquery-1.11.3.js"></script>  <script>  //开局: 点击btn1，发射普通子弹  //btn1.addEventListener("click",function(){ ... })  // ||  //$("#btn1").on("click",function(){  // ||  $("#btn1").click(function(){  console.log(`发射普通子弹......`)  })  //因为发射导弹的处理函数有可能被移除，所以应该用有名称的函数  function shoot2(){  alert("发射跟踪导弹=>=>=>")  }  //点btn2，可以给btn1多添加一种跟踪导弹  //$("#btn2").on("click",function(){  // ||  $("#btn2").click(function(){  //btn1.addEventListener("click",shoot2)  // ||  //$("#btn1").on("click",shoot2)  // ||  $("#btn1").click(shoot2)  })  //点btn3, 可以从btn1上移除跟踪导弹  //$("#btn3").on("click",function(){  // ||  $("#btn3").click(function(){  //btn1.removeEventListener("click",shoot2)  // ||  $("#btn1").off("click",shoot2)  })  </script>  </body>  </html>  运行结果： |

## 2.事件委托：

(1). 何时: 多个平级子元素要绑定相同的事件时

(2). DOM中: 3步 —— 对于今后框架开发才是重要的！

a. 事件只绑定在父元素上一份即可

b. 用e.target代替this，来获得最初点击的目标元素

c. 判断e.target的特征是想要的，才继续执行后续操作

(3). jq中: 其实也可以用DOM版的事件委托，但是jQuery提供了事件委托的简化

a. 事件只绑定在父元素上一份即可，但是必须用.on()绑定

b. 不再用e.target代替this，jquery又让this重新指向了当初触发事件的那个子元素 ——又可以用this了

c. 不用自己写if来判断点击的元素是否想要的。而是在.on()中添加第二个选择器参数。由on()自动用选择器先验证当前点击的元素是否想要的，然后再决定是否执行事件处理函数

d. 总结:

$父元素.on("事件名", "选择器", function(){

//this->最初点击的那个子元素，而不再指父元素了！

})

e. 但是, 这种简写只有jQuery中能用！DOM中或今后的其它框架中都不能用！所以，这个简写了解即可！

(4). 示例: 选择飞机

day02 clone/10\_replace\_clone\_delegate.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  <style>  body {  text-align: center;  }  </style>  </head>  <body>  <h1>替换节点</h1>  <div id="chosen">  <img src="img/p0.png">  </div>  <hr />  <div id="list">  <img src="img/p3.png">  <img src="img/p4.png">  <img src="img/p5.png">  </div>  <script src="js/jquery-1.11.3.js"></script>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: id为list下的所有img都可点击  //1. 所以，应该用事件委托优化，事件应该只绑定在父元素上一份  $("#list")  //2. 绑定事件处理函数  //本例中: 单击图片触发变化  //只有点在img元素上才能触发变化  // 元素选择器  .on("click","img",function(){  //3. 查找要修改的元素  //4. 修改元素  //本例中: 先将当前点中的img克隆一个一模一样的副本，用副本代替id为chosen的div下的img。  $(this).clone().replaceAll("#chosen>img") //不用自己找，replaceAll自带查找功能！  })  </script>  </body>  </html>  运行结果: |

## 3. window.onload事件，页面加载后自动执行:

对应小程序视频: 小程序->在线->JQUERY->day03 1. 页面加载完成后自动执行...

(1). 问题1: 将来我们的js都是写在网页外部的独立的js文件中的。通过<script>标签引入到网页中。应该在网页开头的<head>中引入，还是在<body>结尾引入？

(2). 答: 如果希望页面一加载就自动执行的js，应该放在<body>结尾引入！因为js中通常都要操作HTML中的元素。必须先有HTML，才能执行js操作HTML。所以js必须在要操作的HTML之后才行！

(3). 问题2: 公司里什么人都有，有人偏偏就把js在<head>中引入！如何保证js文件无论在开头引入，还是在结尾引入，都能注定在页面加载完成后才自动执行？

(4). 解决

a. 其实整个网页加载完成时，浏览器会自动触发一个window.onload事件。

放在load事件中的代码，默认注定会在整个页面加载完成后，才自动执行！

b. 所以:

DOM中：window.onload=function(){

//这里的代码，无论何时定义，注定只会在整个页面所有内容加载完之后才执行！

}

JQuery：$(window).load(function(){

//这里的代码，注定只会在整个页面所有内容加载完之后才执行

});

(5). 问题3: window.onload=function(){}必须等待所有页面内容(HTML+JS+CSS+图片)全部下载完成，才能触发，才能绑定事件，让用户用上功能——晚。有些功能用户根本不关心界面的css和图片是否好看！一心只想第一时间赶紧用上功能！——抢票

(6). 解决:

a. 其实在window.onload之前，仅HTML和js下载完成后，就有一个事件触发:

DOMContentLoaded 仅DOM内容加载完成

放在这个事件中的代码，会比window.onload触发早的多

因为不用等大量的css和图片下载完，就可提前触发



b. 问题: DOMContentLoaded有兼容性问题

c. 解决: jQuery中提供了一个没有兼容性问题的统一做法代替DOMContentLoaded

$(document).ready(function(){ ... })

DOM内容 就绪

(7). 问题: $(document).ready(太长了！

(8). 解决: 今后绑定仅DOM内容加载完成可简写为：

$(function(){ ... })

ES6中：

$(()=>{ … })

(9). 总结: 今后所有不依赖于css和图片的大部分页面初始化jquery代码，都应该放在$(function(){ ... })中！

(10). $(function(){ ... }) 和(function(){ ... })()

a. (function(){ ... })() 匿名函数自调 立刻执行其中的代码，执行后立刻释放

b. $(function(){ ... }) 事件绑定 不是立刻执行，必须等到网页中HTML+JS都加载完成了，才自动执行

c. 为了避免使用全局变量，js代码应该包含在匿名函数自调中。但是进入jquery之后，$(function(){ ... })也可以将变量限制在function内部，也可以避免使用全局变量。所以，jquery中，写了$(function(){ ... })，就~~不用重复写(function(){ ... })()~~

(11). 示例: 仅DOM内容加载完成，就提前给按钮绑定单击事件

event/14\_ready.js

|  |
| --- |
| //希望仅DOM内容（HTML+JS）加载完就提前出发  //固定套路！  //DOMContentLoaded  //$(document).ready(function(){ //早  //||  $(function(){  alert("仅DOM内容（HTML+JS）加载完")  //自动给id为btn的按钮，绑定单击事件  $("#btn").click(function(){  alert("抢票！")  })  });  //希望当整个页面加载完成后，才  window.onload=function(){ //晚  alert("整个页面加载完成！")  } |

event/14\_ready.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title> new document </title>  <meta charset="utf-8">  <script src="js/jquery-1.11.3.js"></script>  <script src="14\_ready.js">  //$(function(){  //$("#btn").click(function(){ ... })  //});  //window.onload=function(){ ... }  </script>  </head>  <body>  <button id="btn">click me</button>  </body>  </html>  <!--仅DOM内容: HTML+JS 加载完先触发DOMContentLoaded，也就是$(function(){...}) 此时，页面中已经有HTML内容，所有事件可以正常绑定到元素上-->  <!--然后,当所有页面内容(HTML+JS+CSS+图片)加载完成后才自动执行window.onload() 稍晚一些-->  运行结果: |

## 4. 鼠标事件:

对应小程序视频: 小程序->在线->JQUERY->day-3 2. 鼠标事件 模拟触发

(1). 回顾: DOM中:

a. mouseover: 当鼠标进入某个元素范围内时自动触发

鼠标 进入

b. mouseout: 当鼠标离开某个元素范围时自动触发

鼠标 离开

(2). 问题: 反复进出子元素，也会反复冒泡触发父元素上的鼠标进出事件。和现实不符！

(3). 解决: jquery中或将来框架中都新增mouseenter(鼠标进入)和mouseleave(鼠标离开)代替mouseover和mouseout

(4). 优点: 反复进出子元素，不会反复触发父元素上的进出事件，和现实相符。

(5). 今后jquery中或框架中如果能用mouseenter和mouseleave，都首选这一组！

(6). 示例: 使用mouseenter和mouseleave代替mouseover和mouseout

event/15\_mouse.html

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <title>事件处理</title>  <meta charset="utf-8" />  <style>  #d1 #d2 #d3 {  cursor: pointer  }  #d1 {  background-color: green;  position: relative;  width: 150px;  height: 150px;  text-align: center;  cursor: pointer;  }  #d2 {  background-color: blue;  position: absolute;  top: 25px;  left: 25px;  width: 100px;  height: 100px;  }  #d3 {  background-color: red;  position: absolute;  top: 25px;  left: 25px;  width: 50px;  height: 50px;  line-height: 50px;  }  </style>  </head>  <body>  <div id="d1">  <div id="d2">  <div id="d3">  </div>  </div>  </div>  <script src="js/jquery-1.11.3.js"></script>  <script>  //当鼠标进入最外层的绿色d1范围内时，提示进入d1  //当鼠标离开最外层的绿色d1范围时，提示离开d1  $("#d1")  // //.mouseover(function(){  // .mouseenter(function(){  // console.log(`鼠标进入d1`)  // })//return $("#d1")  // //.mouseout(function(){  // .mouseleave(function(){  // console.log(`鼠标离开d1`)  // })  .hover(  function(){ console.log(`鼠标进入d1`) },//给mouseenter  function(){ console.log(`鼠标离开d1`) } //给mouseleave  )  </script>  </body>  </html>  运行结果： |

(7). 简写: 很多情况下都会同时绑定鼠标进入和鼠标移出事件

a. 今后，只要同时绑定鼠标进入和鼠标移出事件时，只要绑定一个hover()就够了。

b. 如何: $元素.hover( //=mouseenter+mouseleave

function(){ ... }, //给mouseenter

function(){ ... } //给mouseleave

)

c. 简写: 如果可以将两个function通过特殊的手段改成一样的函数，则只写一个function即可！——少见，特例

$元素.hover(//=mouseenter+mouseleave

function(){ ... } //既给mouseenter，又给mouseleave

)

d. 强调: .hover()中只写一个函数时，~~不是只绑定mouseenter的意思！~~而是同时绑定mouseenter和mouseleave的意思，且唯一的一个函数，既给mouseenter，又给mouseleave

e. 示例: 使用hover简化mouseenter和mouseleave

event/16\_hover.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title> new document </title>  <meta charset="utf-8">  <style>  #target {  border: 1px solid #eee;  border-radius: 6px;  padding: 10px;  transition: all .5s linear;  }  /\* 当鼠标进入div时，添加hover class，当鼠标移出div时，移除div的class hover \*/  #target.hover {  border: 1px solid #aaa;  box-shadow: 0 0 6px #aaa;  background-color:red;  color:#fff;  }  /\* 其实也可以用纯css做 \*/  /\* #target:hover {  border: 1px solid #aaa;  box-shadow: 0 0 6px #aaa;  background-color:red;  color:#fff;  } \*/  </style>  </head>  <body>  <h1>使用hover(fn,fn)</h1>  <h3>鼠标悬停在div上方，则突出显示；移出则取消突出显示</h3>  <div id="target">  <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Eligendi neque quae voluptatum ducimus culpa itaque maxime distinctio soluta cum cupiditate esse tenetur deserunt fuga perferendis sed veritatis asperiores. Numquam officia.</p>  </div>  <script src="js/jquery-1.11.3.js"></script>  <script>  // $("#target").hover( //=mouseenter+mouseleave  // function(){ $(this).addClass("hover") }, //给mouseenter  // function(){ $(this).removeClass("hover") } //给mouseleave  // )  $("#target").hover( //=mouseenter+mouseleave  function(){ $(this).toggleClass("hover") }, //既给mouseenter，又给mouseleave  )  </script>  </body>  </html>  运行结果: |

## 5. 模拟触发:

对应小程序视频: 小程序->在线->JQUERY->day-3 2. 鼠标事件 模拟触发

(1). 什么是: 不点击元素，也能触发元素上的事件处理函数

(2). 比如: 搜索时:

a. 点搜索按钮执行搜索

b. 在文本框中输入关键词，按回车也能实现和点搜索按钮完全一样的搜索功能

(3). 如何: $元素.trigger("事件名")

触发

(4). 意为: 找到指定元素，并触发元素上指定事件上绑定的事件处理函数

(5). 简写: 如果要触发的事件，属于十几种常用事件之一，可简写为:

$元素.事件名()

(6). 示例: 按回车也能搜索event/17\_trigger.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  <style>  </style>  </head>  <body>  <input><button>百度一下</button>  <script src="js/jquery-1.11.3.js"></script>  <script>  //希望点按钮获取文本款中的关键词，执行搜索操作  //DOM 4步  //1. 查找触发事件的元素  //本例中: 点按钮执行搜索  $("button")  //2. 绑定事件处理函数  //本例中: 绑定单击事件  .click(function(){  //先获取文本框中的内容  var value=$(":text").val().trim();  //如果文本框中内容不为""  if(value!==""){  //再执行搜索操作  console.log(`查找 ${value} 相关的内容...`)  }  });  //希望在文本框上按回车,可执行和点击按钮一样的搜索功能  // 键盘按下再抬起  $(":text").keyup(function(e){  //只有按的是回车键时才执行搜索  //事件对象e中保存了一个keyCode属性，专门获取本次按键的序号  //如果按的是13号键，说明本次按的是回车  if(e.keyCode==13){  //模拟触发按钮的单击事件处理函数  //$("button").trigger("click")  //因为click事件属于常用事件列表  $("button").click();  }  })  </script>  </body>  </html>  运行结果： |

# 三. 动画: 2大类

## 1. 简单动画: 固定写死的三种动画效果，9个函数 —— 几乎不用！

(1). 显示隐藏: $元素.show() $元素.hide() $元素.toggle()

显示 隐藏 在显示和隐藏之间来回切换

display:block display:none

a. 问题: show()/hide()/toggle()不加任何参数，默认瞬间显示隐藏，没有动画效果

b. 解决: 只要给show()/hide()/toggle()加上一个动画持续时间毫秒数参数，就能有过渡

(2). 上滑下滑: $元素.slideUp() $元素.slideDown() $元素.slideToggle()

(3). 淡入淡出: $元素.fadeIn() $元素.fadeOut() $元素.fadeToggle()

|  |
| --- |
| 简单动画9个函数两个致命缺点:  1. 效果好丑，但是如果想改，不知道改哪儿！——几乎不可维护！  2. 在F12 element中看到变化过程！—— 说明这组动画使用js程序+定时器模拟的。——效率低！远不如css transition  总结: 今后只要简单动画效果，首选css transition做！  优点: 1. 便于维护！2. 比js程序效率高！ |

(4). 示例: 简单测试简单动画效果

1\_show\_hide.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title> new document </title>  <meta charset="utf-8">  </head>  <body>  <h1>jQuery动画函数——显示隐藏动画</h1>  <button id="btn1">显示/隐藏div</button>  <div id="target">  <p><span>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Tempore debitis animi sint iste sequi sunt ad  excepturi error labore molestiae est expedita eos nisi placeat provident dolorem quos facilis!  Sapiente!</span><span>Accusamus neque id reprehenderit! Voluptatem in deleniti laboriosam commodi facere magnam  impedit minima corrupti distinctio culpa amet optio natus esse. Inventore incidunt ab id perspiciatis atque  minus magnam tempore harum.</span></p>  </div>  <script src="js/jquery-1.11.3.js"></script>  <script>  $("#btn1").click(function () {  var $div=$("#target");  //如果div是隐藏的，就显示div  // if($div.is(":hidden")){  // $div.show();  // }else{//否则如果div是显示的，就隐藏div  // $div.hide();  // }  //$div.toggle(2000);  //$div.slideToggle();  $div.fadeToggle();  });  </script>  </body>  </html>  运行结果： |

(5). 如果不考虑过渡效果，show()/hide()/toggle()还是非常好用的！大大的简化了display:none和display:block！——没有性能损失，也不用考虑可维护性！——建议使用的！

(6). 示例: 切换精简显示品牌列表和显示全部品牌

2\_show\_hide2.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  <style>  body {  text-align: center;  }  ul {  list-style: none;  }  li {  margin: 0;  padding: 0;  display: inline-block;  width: 30%;  }  </style>  </head>  <body>  <ul id="list">  <li>尼康(234)</li>  <li>佳能(22)</li>  <li>索尼(932)</li>  <li>宾得(11)</li>  <li>爱国者(3234)</li>  <li>欧巴(32)</li>  <li>海鸥(2334)</li>  <li>卡西欧(334)</li>  <li>三星(834)</li>  <li>松下(234)</li>  <li>其它品牌(2343)</li>  </ul>  <button data-toggle="brandlist">精简显示品牌</button>  <script src="js/jquery-1.11.3.js"></script>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: 点data-toggle="brandlist"的按钮  $("[data-toggle='brandlist']")  //2. 绑定事件处理函数  .click(function(){  //3. 查找要修改的元素  //本例中: 要修改ul下大于4位置且不是最后一个的li  $("ul>li:gt(4):not(:last)")  //4. 修改元素  //本例中: 切换这几个li的显示隐藏  .toggle();//在display:none和display:block之间来回切换  //修改当前按钮的文字:  var $btn=$(this);  //如果当前按钮的文字是: 精简显示品牌，就改为显示全部品牌  //否则如果当前按钮的文字是显示全部品牌，就改为精简显示品牌  if($btn.html()=="精简显示品牌"){  $btn.html("显示全部品牌");  }else{  $btn.html("精简显示品牌");  }  })  </script>  </body>  </html>  运行结果: |

## 2. 万能动画:

(1). 什么是: 能对大多数css属性应用动画效果的函数

(2). 如何: $元素.animate({

css属性: 目标值

}, 动画持续时间)

(3). 原理: animate函数会自动获得元素现在的状态和属性值，根据当前值与目标值之间的差距，自动计算动画变化的过程，并执行动画播放效果

(4). 问题: 也是用js程序+定时器模拟的——效率不如css

(5). 但是，animate毕竟比较灵活！所以，使用比较广泛！

(6). 问题: animate只支持单个数值的css属性。

不支持颜色动画和CSS3变换transform

但是CSS transition支持颜色动画和css3变换的！

## 3. 排队和并发:

(1). 并发: 放在一个animate中的多个css属性是同时变化的

(2). 排队: 多个css属性先后依次变化

a. 如何: 对一个元素先后调用多个animate()函数，多个animate()函数中的css属性是先后依次变化的！

b. 原理:

1). 每个元素身上都有一个动画队列

2). animate()其实并不是立刻执行动画的意思，仅仅是将动画加入到元素的动画队列中

3). 如果元素的动画队列中，没有正在播放的动画，则新加入的动画会立刻执行

4). 如果元素的动画队列中，有正在播放的动画，则新加入的动画，必须等待前一个动画播放完毕，才能开始播放！

## 4. 停止动画:

(1). $元素.stop()

(2). 坑: stop()默认只能停止队列中正在播放的一个动画。后续动画依然继续执行！

(3). 解决: $元素.stop(true) //既停止当前正在播放的动画，又清空后续队列

5. jQuery新增选择器: :animated 专门选择正在播放动画的元素

6. 动画结束后自动执行:

(1). 问题: animate()底层是定时器模拟的，所以animate()是异步的！放在animate()之后的代码不会等animate()执行完才执行，而是几乎和animate()同时执行。

(2). 解决: 所有动画函数都有最后一个回调函数参数，放在最后一个参数回调函数中的代码，注定会在动画播放结束后才执行

$元素.animate({ ... }, 持续时间, function(){ ... })

7. 示例: 四颗小星星，点每个小星星播放规定的动画效果

5\_stars.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  <style>  img {  position: relative;  }  </style>  </head>  <body>  <img id="s1" src="img/star.png"><br />  <img id="s2" src="img/star.png"><br />  <img id="s3" src="img/star.png"><br />  <img id="s4" src="img/star.png"><br />  <script src="js/jquery-1.11.3.js"></script>  <script>  /\*  s1在屏幕左上角的小星星， 点击后从左移动到屏幕右边  s2在屏幕左上角的小星星，点击后从左移动到屏幕右边，再移动到下边——走直角  s3在屏幕左上角的小星星，点击后从左上角移动到屏幕右下边，走斜线  s4点击小星星，变大、变淡.... 直至消失  \*/  $("#s1").click(function () {  var $s = $(this);  //希望点一次启动，再点一次停止  //如果当前星星正在播放动画，则停止动画  if ($s.is(":animated")) {  $s.stop();  } else {//否则如果当前星星没有播放动画，则启动动画  $s.animate({  left: 400  }, 3000)  }  })  $("#s2").click(function () {  var $s = $(this);  if($s.is(":animated")){  $s.stop(true)  }else{  $s.animate({  left: 400  }, 2000)  .animate({  top: 200  }, 1000)  }  })  $("#s3").click(function () {  var $s = $(this);  if($s.is(":animated")){  $s.stop()  }else{  $s.animate({  left: 400,  top: 200  }, 3000)  }  })  $("#s4").click(function(){  alert("疼！");  var $s=$(this)  $s.animate(  {//异步  width:256,  height:256,  opacity:0  },  2000,  function(){ //回调函数 注定会在动画播放结束后自动执行  //animate底层让this->animate.前的jq对象  $s.hide();//display:none  }  )  })  </script>  </body>  </html>  运行结果： |

# Day04

今日扩展视频：

小程序->在线->JQUERY->day04

作业: 高频笔试题 jsonp跨域

16. jquery跨域 cors 同源策略 express脚手架

项目: 学子商城项目 1~4

正课:

1. 类数组对象操作

2. 添加自定义函数

3. 封装自定义插件

4. ajax

5. \*\*\*\*\*跨域

# 一. 类数组对象操作:

对应小程序视频: 小程序->在线->JQUERY->day03 4. 类数组对象操作

1. 问题: jQuery中所有查找结果都是类数组对象，但是js语言非常歧视类数组对象。虽然长的像数组，但是无法使用数组家好的函数！就会限制jQuery查找结果发挥更大的作用.

2. 解决: jQuery中模仿着数组家最常用的两个函数forEach和indexOf，专门定义了两个新函数each()和index()，专门供jQuery查找结果的类数组对象使用！

3. 遍历查询结果中每个DOM元素: each():模仿数组家的forEach:

(1). 回顾: 数组家的forEach

1). arr.forEach(function(elem, i, arr){ ... })

2). 依次遍历arr数组中每个元素，每遍历一个元素，就自动执行回调函数，对每个元素执行相同的操作

(2). jq中: 新增each()

1). $(...).each(function(i, elem){

//i 获得当前正在遍历的下标位置

//elem 获得当前正在遍历的一个DOM元素

})

(3). 示例: 遍历ul下每li，并按要求修改符合条件的li

day03 each index/9\_each.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  <style>  </style>  </head>  <body>  <ul id="list">  <li>98</li>  <li>85</li>  <li>33</li>  <li>99</li>  <li>52</li>  </ul>  <script src="js/jquery-1.11.3.js"></script>  <script>  //回顾forEach  var arr=["亮亮","然然","东东"];  console.log(arr);  arr.forEach(function(elem){  console.log(`${elem} - 到！`)  })  //请给每个不足60分的成绩+10分，并将超过90分的成绩用绿色背景标识出来  console.log($("ul>li"));//爹是Object  //错误: 类数组对象不能用数组家的函数  //$("ul>li").forEach(function(elem){  //正确:  $("ul>li").each(function(i,elem){  //先将当前正在遍历的dom元素elem，转为jquery对象  var $elem=$(elem);  //再获得元素的内容，转为整数  var score=parseInt($elem.html());  // 副本 按值传递  //如果elem的内容<60，就+10分  if(score<60){  score+=10;//修改副本，页面是不变的  //必须将新值再放回去，页面才能改变  $elem.html(score);  }else if(score>=90){//否则如果elem的内容>=90，就让elem变绿  $elem.css("background-color","lightGreen")  }  })  </script>  </body>  </html>  运行结果: |

4. 获取元素在查找结果中的位置：模仿数组家的indexOf()

(1). 回顾数组家indexOf()

a. var i=arr.indexOf(要找的元素值)

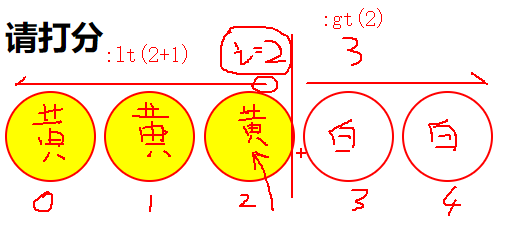
b. 在数组arr中查找“要找的元素值”出现的下标位置。如果找不到返回-1

(2). jq中: 新增index()

a. var i=$(...).index(DOM元素)

b. 查找一个指定DOM元素在$(...)查找结果中的下标位置

(3). 示例: 五星评价



day03 each index/10\_index.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  <style>  .score {  list-style: none;  margin: 0;  padding: 0;  }  .score li {  display: inline-block;  width: 50px;  height: 50px;  border: 1px solid #f00;  border-radius: 50%;  cursor: pointer;  }  </style>  </head>  <body>  <h3>请打分</h3>  <ul class="score">  <li></li>  <li></li>  <li></li>  <li></li>  <li></li>  </ul>  <script src="js/jquery-1.11.3.js"></script>  <script>  //回顾indexOf  var arr=["亮亮","然然","东东"];  //想查找然然的位置  var i=arr.indexOf("然然");  console.log(`然然在${i}位置`);  //获得当前单击的li在所有li中的位置i,i及其执行的都变为红色，i之后的都变为白色  //DOM 4步  //1. 查找触发事件的元素  //本例中: ul下的所有li都能单击  $("ul>li")  //2. 绑定事件处理函数  .click(function(){  //3. 查找要修改的元素  //4. 修改元素  //先获得当前点击的li在所有li中的下标位置i  var i=$("ul>li").index(this)  //再将<i+1位置的所有li背景变为黄色  //错误: i是变量！直接放在字符串中，就不是变量了！  //$(`ul>li:lt(i+1)`)  //正确: 用模板字符串将变量i拼接到选择器中  $(`ul>li:lt(${i+1})`).css("background-color","yellow");  //再见>i位置的所有li背景变为白色  $(`ul>li:gt(${i})`).css("background-color","#fff");  })  </script>  </body>  </html>  运行结果: |

对应小程序视频: 小程序->在线->JQUERY->day04 14. 添加jQuery自定义函数 封装jquery自定义插件

# 二. 添加自定义函数：

1. 问题: 我常用的功能，jquery家没有提供

2. 解决: 自定义一个函数，添加到jQuery家原型对象中

3. 如何:

jQuery.prototype.自定义函数=function(形参列表){

... ...

}

4. 问题: 如何获得将来调用这个公共函数的.前的jQuery查找结果对象？

5. 解决: this:

因为将来: $(...).自定义函数()

所以: this->.前的jquery查找结果对象$(...)

所以: jQuery原型对象方法中的this，不用再$(this)的。因为已经是jQuery子对象了。

6. 示例: 为jQuery家添加对查找结果中所有DOM元素内容求和的sum()函数

11\_sum.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title></title>  <script src="js/jquery-1.11.3.js"></script>  <script>  //在jQuery家原型对象中添加共用的函数sum  jQuery.prototype.sum=function(){  console.log(`调用了一次我自定义的sum函数`)  //遍历sum.前的查找结果中的每个dom元素的内容，并累加求和  var result=0;  //问题: 如何获得将来调用这个公共函数的.前的查找结果对象？！  //将来: $("ul>li").sum()  // this->$("ul>li")  for(var i=0;i<this.length;i++){  //因为this是$("ul>li")，是类数组对象，其中包含多个DOM元素li对象  //this[i]取出的应该是一个DOM的li对象  result+=parseInt(this[i].innerHTML)  }  return result;  }  </script>  </head>  <body>  <ul>  <li>85</li>  <li>91</li>  <li>73</li>  <li>59</li>  </ul>  <script>  //比如: 我经常需要对找到的所有元素内容求和！  // jquery家子对象  var result=$("ul>li").sum()  console.log(result);  </script>  </body>  </html>  运行结果: |

# 三. 封装自定义插件:

1. 什么是插件/组件: 包含专属的HTML，CSS，JS的独立的可重用的页面功能区域

2. 为什么: 重用！

3. 何时: 如果页面中一个独立的功能区域在项目中多个位置被反复用到，都要封装为插件，再反复使用插件！

4. 其实jquery官方提供了一套极好用的插件: jQuery UI

(1). jqueryui.com

(2). 如何使用: 仅手风琴效果举例:

a. 在页面中先引入jquery，再引入jqueryui的css和js

<script src="js/jquery-1.11.3.js">

<script src="js/jquery-ui.js">

<link rel="stylesheet" href="css/jquery-ui.css">

b. 按插件要求编写HTML内容(同bootstrap)

不同点: 使用jqueryUI时，HTML元素上不用加任何class！

c. 在自定义js中，用jquery找到插件的父元素，调用jqueryui提前定义好的插件函数即可！

(3). 问题: 只有pc端，没有移动端，且不支持响应式！——jQuery几乎退出历史舞台，被bootstrap替代了！

(4). 示例: 使用jqueryui插件实现手风琴效果

14\_widget\_accordion.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title> new document </title>  <meta charset="utf-8">  <script src="js/jquery-1.11.3.js"></script>  <script src="js/jquery-ui.js"></script>  <link rel="stylesheet" href="css/jquery-ui.css">  </head>  <body>  <h1>jQueryUI：Widgets —— Accordion</h1>  <!--以下HTML从14\_accordion.html中复制过来的，且去掉了所有的class-->  <div id="my-accordion">  <div>《西游记》简介</div>  <div>一个和尚和四个动物的故事: Lorem ipsum dolor sit amet, consectetur adipisicing elit. Similique nulla voluptas velit minus esse voluptatem illum quis magni nihil sint facilis cupiditate nobis quia ab neque. Modi veniam omnis nisi? </div>  <div>《水浒传》简介</div>  <div>105个男人和三个女人的故事: Lorem ipsum dolor sit amet, consectetur adipisicing elit. Omnis provident sapiente aperiam reprehenderit repellat rem magnam vel odio quia harum hic impedit dolorem similique ea est consequatur adipisci at nemo!  </div>  <div>《红楼梦》简介</div>  <div>一个男人和一群女人的故事: Lorem ipsum dolor sit amet, consectetur adipisicing elit. Delectus minima quidem aspernatur eligendi optio cupiditate minus nam expedita? Aliquid veritatis doloribus maxime vel dicta illo unde iusto qui quasi doloremque.</div>  </div>  <script>  $("#my-accordion").accordion();  </script>  </body>  </html>  运行结果： |

5. 模仿jqueryui风格，封装自定义jquery插件:

(0). 前提: 已经用传统的HTML，css和js实现了插件的功能，只不过和别的网页内容混在一起，没有独立出来，无法重用而已！

所以，封装插件的过程，其实只是一个提取HTML，CSS和js的过程。而不是从0开始制作一个功能！

(1). 先创建一个独立的css文件，将原页面中插件相关的css剪切到独立的css文件中保存。

(2). 再创建一个独立的js文件，在独立的js中向jquery原型对象中添加一个自定义插件函数，做2件事:

a. 为要应用插件的元素及其子元素自动添加必须的class

b. 自动绑定事件处理函数:

因为原页面中已经实现了插件的功能，所以只要单纯将原页面中的事件绑定代码剪切到插件函数中，放在自动添加class之后即可！

6. 如何在HTML页面中使用自定义jquery插件: 同使用jqueryui插件完全一样！

7. 总结: 从此如果其他项目也想用这个插件:

(1). 只要将包含插件的css和js的文件夹/压缩包拷贝到新项目

(2). 在新项目的网页中引入jquery、插件的js和css

(3). 调用一次插件函数即可！

8. 示例: 封装一个手风琴插件

my-ui/my-ui.css

|  |
| --- |
| .accordion{width:80%; margin:0 auto;}  .accordion>.title{  background:#eee; border:1px soild #aaa;  padding:6px; font-size:1.5em;  font-weight:bold; cursor:pointer;  }  .accordion>.content{  border-left:1px solid #eee;  border-right:1px solid #eee;  }  .accordion>:last-child{  border-bottom:1px solid #eee;  }  .fade{  height:0;  opacity:0;  overflow:hidden;  transition:all .5s linear;  }  .in{  height:84px;  opacity:1;  } |

my-ui/my-ui.js

|  |
| --- |
| jQuery.prototype.myAccordion=function(){  console.log(`调用了自定义的myAccordion()函数`)  //1. 为要应用插件的当前元素及其子元素自动添加class  //问题: 如何获得当前要应用插件的元素?  //解决：this->将来要应用插件的.前的元素  //因为this->$("#my-accordion")，已经是jQuery子对象，所以不用$(this);  var $parent=this;//$parent->id为my-accordion的父元素div  //为父元素div本身添加class accordion  $parent.addClass("accordion")  //为父元素下所有偶数位置的元素加title class  .children(":even").addClass("title")  //return 所有偶数位置的元素  //为每个偶数位置的元素的下一个兄弟元素加content和fade class  .next().addClass("content fade")  //return 所有奇数位置的元素  //为所有奇数位置的元素中第一个元素加class in  .first().addClass("in");  //2. 自动绑定事件  $(".accordion").on("click",".title",e=>  $(e.target).next(".content").toggleClass("in")  .siblings(".content").removeClass("in")  );  }  //将来: 比如: id为my-accordion的div想应用手风琴插件  //按jquery ui做法:  //$("#my-accordion").myAccordion()  //找到插件父元素 调用自定义插件函数 |

14\_accordion.html

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title> new document </title>  <meta charset="utf-8">  <script src="js/jquery-1.11.3.js"></script>  <script src="my-ui/my-ui.js"></script>  <link rel="stylesheet" href="my-ui/my-ui.css">  </head>  <body>  <h1>使用“高度动画”实现“手风琴”组件</h1>  <!--按插件规定编写HTML内容，一个class都不要加！-->  <div id="my-accordion">  <div>《西游记》简介</div>  <div>一个和尚和四个动物的故事: Lorem ipsum dolor sit amet, consectetur adipisicing elit. Similique nulla voluptas velit minus esse voluptatem illum quis magni nihil sint facilis cupiditate nobis quia ab neque. Modi veniam omnis nisi? </div>  <div>《水浒传》简介</div>  <div>105个男人和三个女人的故事: Lorem ipsum dolor sit amet, consectetur adipisicing elit. Omnis provident sapiente aperiam reprehenderit repellat rem magnam vel odio quia harum hic impedit dolorem similique ea est consequatur adipisci at nemo!</div>  <div>《红楼梦》简介</div>  <div>一个男人和一群女人的故事: Lorem ipsum dolor sit amet, consectetur adipisicing elit. Delectus minima quidem aspernatur eligendi optio cupiditate minus nam expedita? Aliquid veritatis doloribus maxime vel dicta illo unde iusto qui quasi doloremque.</div>  </div>  <script>  //查找要应用插件的父元素，只需要调用自定义的插件函数  $("#my-accordion").myAccordion();  //2件事:  //1. 自动添加class  //2. 自动绑定事件  </script>  </body>  </html>  运行结果: |

# 四. ajax

对应小程序视频: 小程序->在线->JQUERY->day04 15. jquery发送ajax请求$.ajax

1. jquery中已经封装好了简化版的一句话发送ajax请求的函数

2. 固定用法:

$.ajax({

url:"服务器端接口地址",

type:"get或post",

data:{ 变量: 值, 变量: 值 }, //发送到服务器端的参数变量和值

dataType:"json", //因为几乎所有服务器端返回的结果都是json字符串，都必须反复调用JSON.parse()将json字符串，转为js中的对象和数组，程序才能用。所以，发送ajax请求时，只要执行dataType参数值为json，$.ajax就会自动调用JSON.parse()将返回的json字符串，转化为对象或数组

.

. 延迟

.

//相当于onreadystatechange=function(){ ... }

success: function(result){ //只要请求响应成功，成功获得服务器端返回的结果时，自动调用success回调函数。并自动将服务器端返回的结果，经过JSON.parse()后翻译为对象，交给result变量。

//在success函数中，就可使用result来获得服务器端返回的结果执行后续操作

}

})

3. 示例: 使用ajax函数发送多种请求

15\_ajax.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/jquery-1.11.3.js"></script>  </head>  <body>  <script>  //想从东哥新浪云服务器的/index接口获得学子商城首页4个商品  //服务器端接口地址: http://xzserver.applinzi.com/index  $.ajax({  url:"http://xzserver.applinzi.com/index",  type:"get",  dataType:"json",  success:function(result){  console.log(result);  }  });  //想从东哥新浪云服务器的/details接口获得5号商品的详细信息  //服务器端接口地址:  // http://xzserver.applinzi.com/details  // 参数: lid=商品编号  $.ajax({  url:"http://xzserver.applinzi.com/details",  type:"get",  data:{ lid:5 },  dataType:"json",  success:function(result){  console.log(result);  }  });  //想利用东哥新浪云服务器的/signin接口登录学子商城  //服务器端接口地址:  //http://xzserver.applinzi.com/users/signin  //post类型:  //参数: uname=用户名&upwd=密码  // dingding 123456  $.ajax({  url:"http://xzserver.applinzi.com/users/signin",  type:"post",  data:{ uname:"dingding", upwd:"123456" },  dataType:"json",  success:function(result){  //result两种情况:  //如果用户名密码正确: {ok:1, uname:"dingding"}  //如果用户名密码错误: {ok:0, msg:"用户名或密码不正确"}  console.log(result);  if(result.ok==1){  document.write(`<h1>Welcome ${result.uname}</h1>`)  }else{  document.write(`<h1 style="color:red">${result.msg}</h1>`)  }  }  })  </script>  </body>  </html>  运行结果: |

# 五. \*\*\*\*\*跨域:

对应小程序视频: 小程序->在线->JQUERY->day04 16. jQuery跨域cors同源策略...

## 1. 什么是: 一个网站下的网页中请求了别的网站的资源！

## 2. 比如:

<link rel="stylesheet" href="别的网站的css">

<script src="别的网站的js">

<img src="别的网站的图片">

<a href="别的网站的网页">

<iframe src="别的网站的网页">

## 3. 包括:

网页来源 请求 要请求的资源所在的网址

(1). 域名不同: www.a.com -> www.b.com

(2). 子域名不同: oa.tedu.cn -> hr.tedu.cn

(3). 端口号不同: localhost:5500 -> localhost:3000

(4). 协议不同: http://12306.cn -> https://12306.cn

(5). 同一台主机: http://localhost -> http://127.0.0.1

域名 ip地址

## 4. 问题: 浏览器禁止ajax发送跨域请求！

报错: Access to XMLHttpRequest at 'http://localhost:3000/' from origin 'http://127.0.0.1:5500' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.

翻译: 从源头'http://127.0.0.1:5500'到目标'http://localhost:3000/'的xhr的访问被CORS策略阻止了。因为响应回来的资源中没有设置Access-Control-Allow-Origin响应头

## 5. 示例: 抛出跨域错误:

16\_server.js

|  |
| --- |
| const http=require("http");  http.createServer(function(req,res){  var weather=`晴 21~31度`;  res.writeHead(200,{  "Content-Type":"text/plain;charset=utf-8"  });  res.write(weather);  res.end();  }).listen(3000)  如何运行  1. 右键单击16\_server.js，选择"在终端中打开"或"在命令行中打开"  2. 等待打开的窗口中显示 路径> 输入node 16\_server.js 回车 启动服务器端  3. 打开浏览器，地址栏输入http://localhost:3000  看到晴 21~31度,说明服务器端启动成功！  如果启动服务器报: listen xxxxx :: 3000错误  原因: 同时开了多个命令行窗口，都监听3000端口，发生端口冲突  解决: 连续点命令行窗口右上角垃圾桶图标，关闭所有命令行窗口    然后再从第一步开始运行。始终保持命令行窗口只开一个 |

16\_client.html

右键单击16\_client.html，选择"open with live server"

F12->console看到错误！

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <script src="js/jquery-1.11.3.js"></script>  <script>  $.ajax({  url:"http://localhost:3000",  type:"get",  success:function(result){  document.write(`<h1>今日北京: ${result}</h1>`)  }  })  </script>  </body>  </html>  运行结果: |

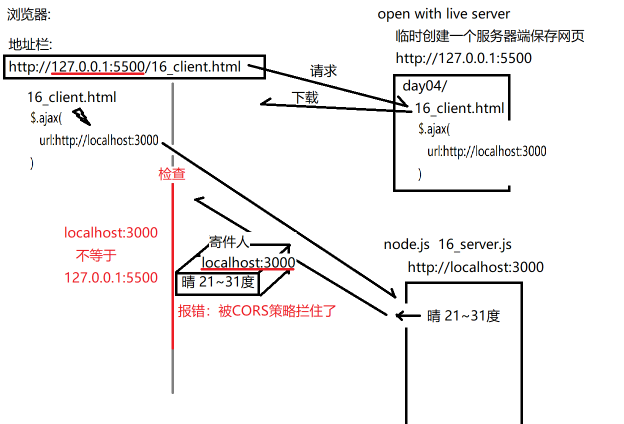
## 6. 跨域错误的原因: 浏览器都有一个同源策略

(1). 什么是同源策略: 当网页中发送ajax请求时，浏览器会检查ajax获得的响应结果和当前网页是否来自于**同一个域名**下的服务器。

a. 如果浏览器发现ajax获得响应结果和当前网页不是来自于同一个域名下，则不允许网页使用获得的返回结果

b. 只有浏览器发现ajax获得响应结果和当前网页来自于**同一个域名**下，才允许网页使用ajax返回的结果！

(2). 比如:



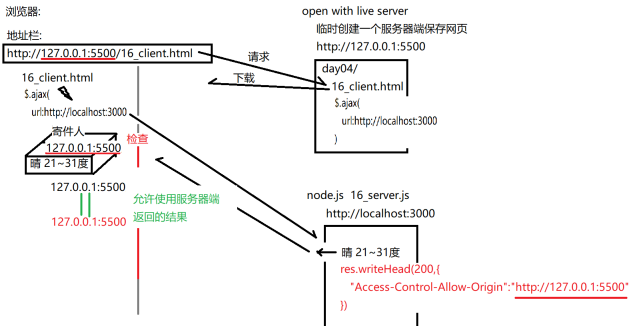
(3). 同源策略: Cross Origin Resources Sharing

跨 不同源头 资源 共享

只有网页来源和响应结果来源相同时才允许使用响应结果

7. 解决: 骗

修改服务器端代码的响应头中的Access-Control-Allow-Origin属性为客户端网页所在地址

8. 示例: 修改服务器端允许http://127.0.0.1:5500的网页跨域使用服务器端数据

修改16\_server.js

|  |
| --- |
| const http=require("http");  http.createServer(function(req,res){  var weather=`晴 21~31度`;  res.writeHead(200,{  "Content-Type":"text/plain;charset=utf-8",  "Access-Control-Allow-Origin":"http://127.0.0.1:5500"  });  res.write(weather);  res.end();  }).listen(3000)  运行结果:  重新运行服务器端  重新用live server打开16\_client.html  看到天气预报，说明成功！  **今日北京: 晴 21~31度** |

9. 问题: 一个服务器端同时包含很多个接口程序，如果每个接口程序都要重复写这么长一坨代码，才能跨域——极其不便于维护？

10. 解决: 各种服务器端语言nodejs, java,...都提供专门支持跨域的中间件程序，仅以nodejs举例:

(1). 服务器端项目安装cors模块:

npm i -save cors

结果: 服务器端项目/node\_modules/cors文件夹

(2). 在服务器端项目主程序app.js中，先引入cors模块，再配置cors中间件

const cors=require("cors");

//在app = express();之后

//配置cors中间件

app.use(cors({

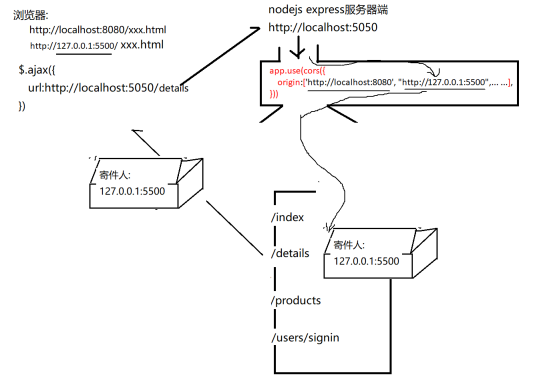
origin:['http://localhost:8080', "http://127.0.0.1:5500",... ...],

VUE脚手架 live server

//希望跨域的客户端网页地址

}))

(3). 结果: 凡是这台服务器的所有接口都可支持origin数组中所有客户端网页的跨域请求。



总结: 增删改查+事件绑定 不用背！！！用的时候来找到即可！

## $的原理: new jQuery() 4种用法:

1. $("选择器") 2件事:

(1). 创建jQuery子对象，

(2). 查找符合条件的DOM元素，保存进jQuery对象中

2. $(DOM元素) 创建jQuery子对象，并无需查找就可将已经获得的DOM元素保存进jQuery对象中.

3. $(`HTML片段`) 将HTML片段创建为新DOM元素对象

4. $(function(){ ... }) 绑定DOMContentLoaded事件，让代码在HTML+JS加载完就可提前执行。

## 总结: jQuery API 三大特点:

1. 自带遍历: 只对jQuery对象调用一次简化版函数，等效于自动对jQuery对象内保存的每个DOM元素都调用一次等效的原生函数或属性。

2. 一个函数两用: 和修改、事件绑定相关的函数

(1). 不传入新值时，默认执行获取旧值的操作

(2). 传入新值时，自动切换为执行修改操作

3. 凡是没有特定返回值的函数，几乎都返回.前正在操作的jQuery对象本身！

如果前一个函数正在操作的.前的属于和后一个操作想要的.前的主语，刚好一样！则可以使用链式操作，避免使用变量或重复代码！

## 1. 查找元素:

(1). 将所有的查找统一为按选择器查找: var $jquery对象=$("选择器")

a. jQuery支持所有CSS3选择器

b. 扩展:

1). 基本过滤:

:first :last :even :odd :eq(i) :gt(i) :lt(i)

2). 内容过滤: :contains(关键词) :has(选择器) :parent :empty

3). 可见性过滤: :visible :hidden

4). 表单元素过滤:

i. :input 选择所有表单元素(input select textarea button)

ii. :text :password :button :radio ... ...

(2). 不需要查找就可直接获得的元素对象: 1个

a. $(document) 根节点对象

(3). 按节点间关系查找:

a. 父子关系:

1). $元素.parent() 代替 元素.parentElement  
 2). $元素.children("选择器") 代替 元素.children ——只在直接子元素中查找

新增: $元素.find("选择器") 在所有后代中查找

3). $元素.children(":first-child") 代替 元素.firstElementChild

4). $元素.children(":last-child") 代替 元素.lastElementChild

b. 兄弟关系:

1). $元素.prev() 代替 元素.previousElementSibling

新增: $元素.prevAll("选择器")

2). $元素.next() 代替 元素.nextElementSibling

新增: $元素.nextAll("选择器")

3). 新增: 查找除当前元素之外其余所有兄弟: $元素.siblings("选择器")

## 2. 修改: 3种

(1). 内容: 3种

a. $元素.html() 代替了 .innerHTML

b. $元素.text() 代替了 .textContent

c. $元素.val() 代替了 .value

(2). 属性: 3种:

|  |  |  |
| --- | --- | --- |
|  | JQUERY | DOM |
| 字符串类型标准属性 | $元素.attr()  $元素.prop() | 核心DOM: getAttribute() setAttribute()  HTML DOM: 元素.属性名 |
| bool类型标准属性 | $元素.prop() | HTML DOM: 元素.属性名 |
| 自定义扩展属性 | $元素.attr()  没有对应！ | 核心DOM: getAttribute() setAttribute()  HTML 5: 元素.dataset.自定义属性 |

|  |  |  |
| --- | --- | --- |
|  | 凡是字符串类型的属性 | 凡是标准属性 |
|  | $元素.attr() | $元素.prop() |
| 字符串类型标准属性 | √ | √ |
| bool类型标准属性 | × | √ |
| 自定义扩展属性 | √ | × |

a. 字符串类型的HTML标准属性: 2种:

1). $元素.attr("属性名","属性值") 代替: getAttribute() setAttribute()

2). $元素.prop("属性名","属性值") 代替: 元素.属性名

b. bool类型的HTML标准属性: 只能用 $元素.prop("属性名",bool)

c. 自定义扩展属性: 1种

1). $元素.attr("data-自定义属性名","属性值")

代替 元素.getAttribute() 元素.setAttribute()

3). 查找带有自定义扩展属性的元素: 属性选择器 [data-自定义属性=属性值]

(3). 样式:

a. 只获取或修改一个css属性时: 统一为: $元素.css("css属性名", "属性值")

b. 批量修改多个css属性时: 用class

1). $元素.addClass()

2). $元素.removeClass()

3). $元素.hasClass()

4). $元素.toggleClass();

|  |
| --- |
| 简写: .attr()/.prop()/.css() 其实都可以一句话修改多个属性值  $元素.attr或prop或css({  属性名: 属性值,  ... : ...  }) |

**3. 添加元素: jquery只需要2步**

(1). 用HTML片段创建新元素

var $新元素=$(`HTML片段`)

说明: 元素所需的属性和样式已经包含在HTML片段里了，所以，无需再专门添加关键属性。

(2). 将新元素添加到DOM树: 5种 10个函数

a. 父元素末尾追加新元素:

$父元素.append($新元素) 代替 父元素.appendChild(新元素)

兄弟函数: $新元素.appendTo($父元素)

b. 父元素开头插入新元素:

新增: $父元素.prepend($新元素)

兄弟函数: $新元素.prependTo($父元素)

c. 插入到一个现有子元素之前:

$现有元素.before($新元素) 代替 父元素.insertBefore(新元素, 现有子元素)

兄弟函数: $新元素.insertBefore($现有元素)

d. 插入到一个现有子元素之后:

新增: $现有元素.after($新元素)

兄弟函数: $新元素.insertAfter($现有元素)

e. 替换父元素下一个现有子元素:

$现有元素.replaceWith($新元素) 父元素.replaceChild(新元素, 现有子元素)

兄弟函数: $新元素.replaceAll($现有元素)

(2). jQuery内已封装了尽量减少操作DOM树的优化(文档片段)，所以我们无需额外专门创建文档片段，jquery内部也已经自动使用了文档片段！

1. **删除元素:**

$元素.remove() 代替 父元素.removeChild(要删除的元素)

1. **新增:**

克隆: $元素.clone()

1. **新增:**

判断元素是否符合选择器要求: $元素.is("选择器")

1. **事件:**

(1). 事件绑定: 1种:

a. $元素.on("事件名", 事件处理函数) 代替 元素.addEventListener("事件名", 处理函数)

$元素.off("事件名", 原事件处理函数) 代替 元素.removeEventListener("事件名",原处理函数)

b. 简写: 只有以下列表中的事件才能: $元素.事件名(事件处理函数)

常用事件名:

|  |  |  |  |
| --- | --- | --- | --- |
| **blur** | 失去焦点 | **mouseenter** | 鼠标进入(jq) |
| **change** | 下拉列表选中项改变 | **mouseleave** | 鼠标移出(jq) |
| **click** | 单击 | **mousemove** | 鼠标移动 |
| **dblclick** | 双击 | **mouseout** | 鼠标移出(dom) |
| **focus** | 获得焦点 | **mouseover** | 鼠标进入(dom) |
| **keydown** | 键盘按键按下 | **mouseup** | 鼠标按键抬起 |
| **keyup** | 键盘按键抬起 | **resize** | 窗口大小改变 |
| **mousedown** | 鼠标按键按下 | **scroll** | 网页滚动 |

(2). 页面加载完成事件：2次

a. DOM中: 先DOMContentLoaded，再window.onload

b. jq中: 先$(function(){ ... })，再$(window).load(function(){ ... })

c. 今后，所有不依赖于css和图片的初始化页面的jquery代码（事件绑定，设置元素的初始内容、属性和样式等），都应该放在$(function(){ ... })中

(3). 事件模型: 3个阶段: 捕获、目标触发、冒泡

(4). 获得事件对象: 定义事件绑定时:

$元素.事件名(function(e){ //触发事件时e=event }

(5). 取消冒泡/停止蔓延: e.stopPropagation()

(6). 只要多个平级子元素都要绑定相同的事件时: 利用冒泡/事件委托: 3步

a. 事件绑定在父元素上一份，但是必须用.on()绑定

b. 不用e.target代替this，this指回了最初触发事件的子元素，又可以用了

c. 为.on()添加第二个选择器参数，.on()自动判断当前点击的元素是否想要的。无需自己写if了。

$父元素.on("事件名", "选择器", function(){

//this->指向最初触发事件的子元素，不再指父元素，this又能用了

//不用写if了

})

(7). 阻止元素自带的默认行为: e.preventDefault();

(8). 获得鼠标位置：

a. 鼠标位置相对屏幕左上角的距离: e.screenX e.screenY

b. 鼠标位置相对浏览器文档显示区左上角的距离 e.clientX e.clientY

c. 鼠标位置相对事件所在元素左上角的距离e.offsetX e.offsetY

(9). 鼠标事件:

a. DOM中: mouseover mouseout 反复进出子元素也会反复冒泡触发父元素的进出事件

b. jq中:

1). mouseenter和mouseleave 代替mouseover和mouseout

2). 如果同时绑定mouseenter和mouseleave，就可只绑定一个hover

$元素.hover( //=mouseenter+mouseleave

function(){ ... }, //给mouseenter

function(){ ... } //给mouseleave

)

3). 特例: 万一两个函数可以修改为一样的，也可以只写一个function

$元素.hover( //=mouseenter+mouseleave

function(){ ... } //给mouseenter和mouseleave

)

(10). 模拟触发: $元素.trigger("事件名")

如果要触发的事件属于常用事件列表，可简写为 $元素.事件名()

(11). 窗口滚动事件:

a. 当滚动条滚动时自动触发: window.onscroll=function(){ ... }

b. 获取滚动条滚动过的距离:

var scrollTop=document.body.scrollTop||document.documentElement.scrollTop;

c. 控制滚动条滚动到指定位置: window.scrollTo(0, 要滚动到的位置)

## 8. 动画:

(1). 简单动画: 3种固定效果，9个函数

a. 显示隐藏:

$元素.show(动画持续时间毫秒数, function(){ 动画结束后自动执行 })

$元素.hide(动画持续时间毫秒数, function(){ 动画结束后自动执行 })

$元素.toggle(动画持续时间毫秒数, function(){ 动画结束后自动执行 });

b. 上滑下滑:

$元素.slideUp(动画持续时间毫秒数, function(){ 动画结束后自动执行 })

$元素.slideDown(动画持续时间毫秒数, function(){ 动画结束后自动执行 })

$元素.slideToggle(动画持续时间毫秒数, function(){ 动画结束后自动执行 });

c. 淡入淡出:

$元素.slideUp(动画持续时间毫秒数, function(){ 动画结束后自动执行 })

$元素.slideDown(动画持续时间毫秒数, function(){ 动画结束后自动执行 })

$元素.slideToggle(动画持续时间毫秒数, function(){ 动画结束后自动执行 });

(2). 万能动画:

$元素.animate({

css属性: 目标值,

... : ...

}, 动画持续时间毫秒数, function(){动画结束后自动执行})

(3). 停止动画: $元素.stop()

(4). 查找正在播放动画的元素: jquery新增选择器: :animated

## 9. 类数组对象操作:

(1). 遍历查找结果中每个DOm元素: $(...).each(function(i, elem){ ... })

(2). 查找一个元素在查找结果中的下标位置: $(...).index(要找的DOm元素)

## 10. 添加自定义函数:

jQuery.prototype.自定义函数=function(形参列表){

this->将来调用这个自定义函数的.前的jQuery子对象

this不用$(this)

}

将来:

$(查找结果).自定义函数()

## 11. 封装自定义插件:

(1). 创建自定义插件:

a. 先创建独立css文件，将原网页中插件相关的css，剪切到独立的css文件中

b. 再创建独立js文件，为jquery原型对象添加一个自定义插件函数。插件函数中做2件事

1). 为找到的HTML元素自动添加class

2). 为找到的元素自定绑定事件处理函数

(2). 使用插件:

a. 将插件的css和js文件拷贝到项目中

b. 在页面中引入jquery的js和插件的js以及插件的css

c. 在页面中按插件要求编写HTML内容和结果，不需要加class

d. 在自定义js中，查找插件父元素，调用插件函数

## 12. ajax

$.ajax({

url:"服务器端接口地址",

type:"get或post",

data:{ 变量: 值, 变量: 值 }, //发送到服务器端的参数变量和值

dataType:"json",

success: function(result){

//可使用result来获得服务器端返回的结果执行后续操作

}

})

## 13. 跨域:

(1). 浏览器同源策略: 浏览器会检查ajax获取的服务器端响应结果的来源地址是否与当前网页来源地址相同。只有相应结果来源于当前网页来源相同，才允许使用响应结果。否则，不允许使用。

(2). 如何跨域:

a. 一个接口:

res.writeHead(200,{

... ...

"Access-Control-Allow-Origin":"要伪装的客户端地址"

})

res.write(要返回的数据)

res.end();

b. nodejs express项目所有接口都要跨域:

1). 安装cors: npm i -save cors

2). app.js中:

const cors=require("cors")

//在var app=express()之后

app.use(cors({

origin:['http://localhost:8080',"http://127.0.0.1:5500",...],

[]中放的是所有可能要伪装的客户端网页地址

}))

## 总结: this 6种:

1. obj.fun() this->obj

2. new Fun() this->new正在创建的子对象

3. 类型名.prototype.共有方法=function(){ ... } this->将来调用这个共有方法的.前的子对象

4. fun() 和 (function(){ })() 和回调函数中的this->window

5. 事件处理函数中的this->当前正在触发事件的这个元素对象

6. $jquery对象.click(function(){ this->将来点击的那个DOM元素对象})

# Vue

# VUE->day01

今日扩展视频：小程序->在线->VUE->day01

1.1 vscode部署vue网页模板code.snippet

作业: 为v-show和v-if添加过渡动画效果

作业: 使用vue实现标签页效果

高频笔试题: 观察者模式

基础比较弱！千万不要现在看！还是以课上内容为主。等到鄙视面试前再看！！

正课：

1. 什么是VUE

2. 如何使用VUE

3. MVVM

4. 绑定语法

5. 指令

对应小程序视频: 小程序->在线->VUE->day01 1. vue和第一个vue程序

# 一. 什么是VUE:

## 1. 什么是: 是第三方开发的基于MVVM设计模式的渐进式的纯前端js框架

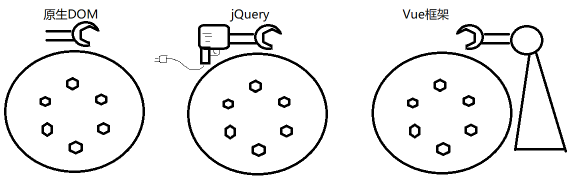
(1). 第三方开发: 下载才能用

(2). 基于MVVM: ?

(3). 渐进式: 可以在项目中逐步引入vue相关功能，很容易和其它技术混搭。但是强烈不建议

(4). 纯前端js: 不需要任何nodejs和后端的知识，单靠浏览器就可以运行和学习VUE

(5). 框架: 已经包含核心功能的半成品前端程序



## 2. 为什么: 简洁！避免大量重复的编码！

## 3. 何时: 今后只要以数据操作(增删改查)为主的项目，都可用vue开发

# 二. 如何使用VUE:

## 1. 2种:

### (1). 将vue.js下载到项目本地引入网页中使用: —— 前3天 学vue基础知识和原理

a. 官网: cn.vuejs.org —— 中文！ —— 学有余力才能去看！

b. <script src="js/vue.js">

c. 问题: 因为前端项目越来越庞大，文件夹结构和代码量越来越复杂，导致不同的团队和公司组织文件和文件夹结构时，各不相同！——混乱

### (2). 公司中都是用vue脚手架代码来开发项目: ——后2天 做学子商城项目

a. 什么是: 已经包含标准的文件夹结构和核心功能的半成品项目！

b. 优点: 标准！不同团队和不同公司发开出的项目结构几乎是完全相同的！

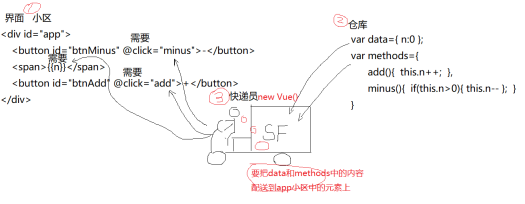
## 2. 示例: 我的第一个vue程序，分别用jquery和vue实现点击按钮修改数量功能

### (1). jQuery版:

1\_first\_jquery.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  </head>  <body>  <button id="btnMinus">-</button><span>0</span><button id="btnAdd">+</button>  <script src="js/jquery-1.11.3.js"></script>  <script>  //DOM 4步  //1. 查找触发事件的元素  //本例中: 点两个按钮修改数量  var $btnMinus=$("#btnMinus");  var $btnAdd=$("#btnAdd");  //2. 绑定事件处理函数  //本例中: 单击两个按钮  $btnAdd.click(function(){  //3. 查找要修改的元素  //本例中: 要修改span  var $span=$("span");  //4. 修改元素  //先取出span中的内容，转为整数  var n=parseInt($span.html())  //再将span的内容+1  n++;  //最后将新值放回span内容中  $span.html(n);  });  //2. 绑定事件处理函数  //本例中: 单击两个按钮  $btnMinus.click(function(){  //3. 查找要修改的元素  //本例中: 要修改span  var $span=$("span");  //4. 修改元素  //先取出span中的内容，转为整数  var n=parseInt($span.html())  //如果数量>0时，才将span的内容-1  if(n>0){ n-- }  //最后将新值放回span内容中  $span.html(n);  });  </script>  </body>  </html> |

### (2). VUE版



2\_first\_vue.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  </head>  <body>  <!--1. 编写界面:  要求: 整个界面必须包裹在一个唯一的父元素内,习惯上: <div id="app"></div>  额外: 2件事:  1.1 找到界面中将来可能发生变化的位置，用{{自定义变量名}}临时占位  本例中: span的内容，将来随点击按钮而变化，所以<span>{{n}}</span>  1.2 找到界面中可能触发事件的元素，用@事件名="自定义处理函数名"标记出来  本例中: 单击btnMinus时触发减法操作，单击btnAdd时触发加法操作，所以  -->  <div id="app">  <button id="btnMinus" @click="minus">-</button>  <span>{{n}}</span>  <button id="btnAdd" @click="add">+</button>  </div>  <!-- <div id="#app2">  </div> -->  <script>  //2. 定义两个仓库对象: 仓库名必须叫data和methods  //2.1 一个仓库叫data，专门保存界面上所需的所有变量和初始值  //本例中: 界面上一共需要一个变量n，初始值为0  var data={ n:0 };  //2.2 另一个仓库叫methods, 专门保存界面上所需的所有事件处理函数  //本例中: 界面上共需要两个事件处理函数,分别是add和minus  var methods={  //复习小程序->在线->JSCORE->day07 作业:ES6关于对象中属性和方法的简写  add(){  //虽然现在事件处理函数和data中的n分着写，但是将来data对象和methods对象会被vue合并为一个对象，事件处理函数data中的变量最终会放在一个大的对象中保存，所以,methdos中的函数可通过this.xx方式操作data中的变量  //本例中: 点btnAdd，触发add函数，只想给data中的数量n+1  this.n++;  //不用自己放回去！  },  minus(){  //本例中: 点btnMinus，触发minus函数，如果data中的数量n>0，才将数量n-1  if(this.n>0){ this.n-- };  }  }  //3. 创建一个VUE对象充当快递员，自动送货  new Vue({  el:"#app", //选择器: 告诉快递员，请将货物送到id为app的小区中的各个住户手中！——这个快递元素所负责的小区  //告诉快递员，需要配送的变量和函数都放在哪些仓库里。  //复习小程序->在线->JSCORE->day07 作业:ES6关于对象中属性和方法的简写  data, //data:data,  methods //methods:methods  });  // new Vue({  // el:"#app2" //这个快递员专门负责app2小区的自动送货任务  // })  </script>  </body>  </html>  运行结果： |

## 3. 总结: Vue开发一个功能的基本步骤: 3步

### (1). 定义界面:

a. 要求: 整个界面所有元素必须放在一个唯有的父元素下包裹

习惯上: <div id="app"> ... </div>

b. 找到界面内将来可能发生变化的位置，用{{自定义变量名}}临时占位

c. 找到界面中所有触发事件的元素，用@事件名="自定义处理函数名"标记

### (2). 定义仓库对象: 2个 ，data和methods

a. data: 专门保存界面中所需的所有变量及其初始值

b. methods: 专门保存界面中所需的所有事件处理函数

因为将来data对象和methods对象会被new Vue()合并为一个对象，所以methods中的事件处理函数，和data中的变量最终会保存在同一个对象中。所以，methods中的事件处理函数可以通过this.变量名操作data中的变量

### (3). 创建Vue对象，充当快递员:

new Vue({

el:"#app", //选择器: 告诉vue对象要将变量和函数送到页面中哪个大块区域中的元素上。

//告诉vue对象界面所需的一切变量和函数都保存在哪里——仓库位置

data, //data:data,

methods, //methods:methods,

})

## 4. 简写:

因为data对象和methods对象迟早都要被装进new Vue()对象中，所以实际开发中，我们不单独定义data和methods对象。而是直接在new Vue()中data属性和methods属性值对象中直接添加页面所需变量和事件处理函数:

new Vue({

el:"#app", //选择器: 告诉vue对象要将变量和函数送到页面中哪个大块区域中的元素上。

//告诉vue对象界面所需的一切变量和函数都保存在哪里——仓库位置

data:{

变量:初始值,

... : ...

},

methods:{

处理函数名(){

this.变量

},

...

}

})

## 5. 示例: 简写以上案例:

3\_first\_vue2.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  </head>  <body>  <!--1. 编写界面: -->  <div id="app">  <button id="btnMinus" @click="minus">-</button>  <span>{{n}}</span>  <button id="btnAdd" @click="add">+</button>  </div>  <script>  //2. 创建一个VUE对象充当快递员，自动送货  new Vue({  el:"#app",  //3. 在Vue对象内添加data属性和methods属性，分别保存页面所需的所有变量和事件处理函数  data:{  n:0  },  methods:{  add(){  this.n++;  },  minus(){  if(this.n>0){ this.n-- };  }  }  });  </script>  </body>  </html>  运行效果: 同上 |

## 6. 简写后的步骤: 3步

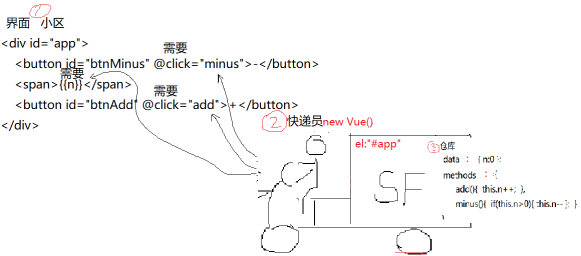
(1). 定义界面：

(2). 先定义new Vue()对象，用选择器找到自己负责的页面区域

(3). 在new Vue()对象内，添加data属性和methods属性，其值都是对象。

a. data属性对象中专门保存页面所需的所有变量,

b. methods属性对象中专门保存页面所需的所有函数。



## 7. 结果:

无论任何原因，只要data中的变量值发生变化！new Vue()快递员都会自动将新值重新送货到界面中对应的元素上。new Vue()会始终保持界面显示的内容和data中变量值同步！

# 三. MVVM: (Model, View, ViewModel)

对应小程序视频: 小程序->在线->VUE->day01 2. MVVM vue绑定原理 虚拟DOM树

## 1. 什么是MVVM设计模式: 对前端代码的重新划分

## 2. 旧前端代码，分为三部分:

(1). HTML: 专门保存网页的内容和结构

(2). CSS: 专门为网页中的元素添加样式

(3). JS: 专门为网页中的元素添加交互效果

## 3. 问题:

(1). 因为HTML和CSS都是静态的语言。缺少标准的程序语言必须的要素

a. 比如HTML和CSS中没有变量，就无法实现一次修改，多处变化

b. 比如HTML和CSS中缺少分支和循环，就无法实现基本的判断和反复生成的功能

总结: HTML和CSS生活不能自理！

(2). 页面上所有的琐碎的修改都要依赖js来实现。导致js代码极其冗余，且包含大量重复劳动！

## 4. 新MVVM，也将代码分为3部分:

### (1). 界面（View）:

包括以前的HTML+CSS，而且还增强了HTML的功能

a. 给HTML添加了变量的功能: <span>{{n}}</span> 如果n变化，则span自动变化，无需任何js代码。

b. 为HTML添加了分支和循环功能: v-if v-else 以及v-for

### (2). 模型对象(Model):

专门集中保存页面所需的变量和函数的对象

比如: data和methods都是模型对象

问题: 模型对象中的变量和方法不会自己长腿跑到界面中的元素上

### (3). 视图模型(ViewModel): 快递员

a. 什么是视图模型: 自动将data中的变量和methods中的函数送到界面中指定元素上，并能自动保持界面显示与data中变量值同步 一种特殊的对象

b. 比如: new Vue()就充当了视图模型自动配送的角色。

## 5. new Vue()如何实现MVVM设计模式: Vue的绑定原理: 访问器属性+虚拟DOM树

### (1). 访问器属性: 复习： 小程序->在线->JSCORE->day05 5. 访问器属性 get set

#### a. 回顾: 访问器属性:

1). 专门提供对一个普通属性的保护。

2). 每个访问器属性都包含一对儿, get和set函数

i. 当外界试图获取访问器属性的值时，自动调用访问器属性中的get

ii. 当外界试图修改访问器属性的值时，自动调用访问器属性中的set

#### b. new Vue()如何利用访问器属性:

1). data对象和methods对象进入new Vue()后会立刻被打散。data中的成员变量和methods中的成员函数，最终会直接隶属于new Vue()对象。且data中的变量和methods中的函数处于平级。所以methods中的函数才能用this.变量名方式访问data中的变量

所以，new Vue()中的几乎所有this，都指new Vue()对象！

Vue{

... ...



... ...

}

2). new Vue()隐藏data中原变量。然后为data中每个变量分别请保镖(访问器属性)。

结果: 任何情况下操作data中的变量n，早就不是普通的变量了。实际操作的都是访问器属性。且，只要试图修改data中的变量n，都会自动调用访问器属性n的set函数！

3). new Vue()在每个访问器属性的set函数中悄悄内置了一个通知函数。只要试图修改变量，都会调用访问器属性的set，都会制定向外发出通知: "xx变量的值变了！快来拿新值！"



### (2). 虚拟DOM树:

#### a. 什么是:

vue通过扫描真实DOM树，只提取出个别可能发生变化的元素，组成的一个简化版的新的DOM树。

#### b. 为什么: 优点：

1). 小, 只包含可能受影响的元素，不受影响的元素是不包含在内的

2). 遍历和查找快！

3). 更新效率高，因为每次只更新受影响的元素，不受影响的元素，不变的！

4). 避免重复代码。虚拟DOM树中已经提前封装了DOM增删改查操作！不用我们自己写！

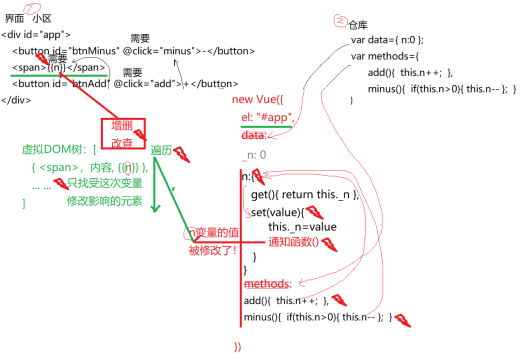
#### c. 虚拟DOM树在绑定过程中如何发挥作用：

1). new Vue中任何位置修改了data中的变量，其实修改的都是访问器属性

2). 都会自动触发这个访问器属性的set，都会自动发出通知

3). new Vue接到通知遍历虚拟DOM树，只找受本次变量修改影响的个别DOM元素

4). 虚拟DOM树利用已经封装好的增删改查操作，仅更新页面上受影响的个别元素。而不是大范围替换界面元素。—— 效率远高于jQuery！



对应小程序视频: 小程序->在线->VUE->day01 3. 绑定语法 {{}} 指令 v-bind... ...

# 四. 绑定语法: 学名插值语法, Interpolation

## 1. 什么是绑定语法:

vue给HTML中新增的一种可以让HTML直接使用 js中的变量 的特殊语法.

## 2. 为什么:

旧HTML不能使用js中的变量，哪怕很小的修改都要js先取出旧值，再修改后，最后放回来，才能更新界面——繁琐！

## 3. 何时:

今后只要界面中一个元素的内容，需要随js中变量自动变化，就要用绑定语法

## 4. 如何:

<元素>{{自定义变量名}}</元素>

## 5. 后续用法:

如果界面中定义了一个{{自定义变量名}}，则new Vue()的data中就必须添加一个同名的变量，来支持这个界面位置的变化！

## 6. 其实: {{}}中除了可以放单个变量之外，还能放任何有返回值的合法的js表达式！——同ES6模板字符串${}的规则

(1). 可以放: 变量，运算，三目，有返回值的函数调用，创建对象，访问数组元素

(2). 不可以放: ~~程序结构~~(分支和循环)以及~~没有返回值的~~函数调用。

## 7. 示例: 在网页中用各种语法绑定变量

4\_{{}}.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  </head>  <body>  <div id="app">  <!--复习： 小程序->在线->JSCORE->day07 1. 模板字符串-->  <h3>Welcome {{uname}}</h3>  <h3>性别: {{sex==1?"男":"女"}}</h3>  <h3>小计: ¥{{(price\*count).toFixed(2)}}</h3>  <h3>下单时间: {{new Date(orderTime).toLocaleString()}}</h3>  <h3>今天星期{{week[new Date().getDay()]}}</h3>  </div>  <script>  new Vue({  el:"#app",  data:{  uname:"dingding",  sex:1,  price:12.5,  count:5,  orderTime:1592549839941, //new Date().getTime()  week:["日","一","二","三","四","五","六"]  // 0 1 2 3 4 5 6  }  })  </script>  </body>  </html>  运行结果: |

## 8. 问题: {{}}只支持元素内容变化，不支持属性值变化——如果元素的属性值想根据js中的变量不断变化，不能用{{}}语法，报错

- src="{{pm25<100?'img/1.png':pm25<200?'img/2.png':pm25<300?'img/3.png':'img/4.png'}}":

Interpolation inside attributes has been removed.

属性中的插值语法已经被移除了

Use v-bind or the colon shorthand instead. For example,

请用v-bind或:简写代替. 例如，

instead of <div id="{{ val }}">, use <div :id="val">.

用<div :id="val"> 代替<div id="{{val}}">

解决: 指令:

# 五. 指令:

## 1. 什么是:

一批vue官方提供的，为HTML添加新功能的特殊自定义扩展属性

## 2. 为什么:

因为{{}}只支持元素内容随变量自动变化，不支持属性自动变化，甚至无法实现分支和循环等复杂结构

## 3. 何时:

今后只要元素的属性值想随变量自动变化或想给HTML添加更多新功能时，都用指令！

## 4. 包括:

### (1). v-bind元素的属性值可能随变量自动变化

a. <元素 v-bind:属性名="js变量或表达式">

b. 原理: 当new VUe()扫描到这里时，或者依赖的变量发生变化时，new Vue都会自动重新计算v-bind后的变量值或表达式结果，用变量值或表达式结果代替该属性的值！

c. 强调: 如果前边加了v-bind:，则""中千万不要再加{{}}，此时""就扮演了{{}}的角色。

d. 简写: 今后v-bind: 都省略 v-bind，只写:

<元素 :属性名="js变量或表达式">

e. 示例: 根据pm25数值改变表情图片

5\_v-bind.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  </head>  <body>  <!--需求: 希望img的src属性值随data中的pm25数值自动变化  pm25<100，src属性值为img/1.png  pm25<200, src属性值为img/2.png  pm25<300, src属性值为img/3.png  其余img/4.png  -->  <div id="app">  <h3>{{pm25<100?  ' img/1.png' : pm25<200?'img/2.png' : pm25<300 ? ' img/3.png ' :  'img/4.png'}}</h3>  <img :src="pm25<100?'img/1.png':pm25<200?'img/2.png':pm25<300?'img/3.png':'img/4.png'" alt="">  </div>  <script>  new Vue({  el:"#app",  data:{  pm25:350  }  })  </script>  </body>  </html>  运行结果： |

### (2). v-show根据js中的变量值控制一个元素的显示隐藏

a. <元素 v-show="bool类型的变量或js条件表达式">

b. 原理: 当new Vue首次扫描到这里或依赖的js变量发生变化时，new Vue都会自动计算""中的变量值或js表达式值。

1). 如果变量值或表达式值为false，则new Vue()自动给元素添加display:none，当前元素隐藏

2). 如果变量值或表达式值为true，则new Vue()自动去掉元素上的display:none，当前元素就显示

c. 示例: 打开和关闭对话框(极其常用)





6\_v-show.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  <style>  #alert{  width:300px;  height:100px;  background-color:#ffa;  position:fixed;  top:50%;  left:50%;  margin-left:-150px;  margin-top:-50px;  line-height:100px;  }  #alert>a{  float:right;  line-height:30px;  margin-right:10px;  }  </style>  </head>  <body>  <div id="app">  <!--希望div#alert随变量isShow的值不同而显示/隐藏  希望点click me，让div#alert显示（将isShow变量值改为true）  希望点×，让div#alert隐藏（将isShow变量值改为false）-->  <button @click="open">click me</button>  <div v-show="isShow" id="alert">  <a href="javascript:;" @click="close">×</a>  您的浏览器版本太低，请升级浏览器  </div>  </div>  <script>  new Vue({  el:"#app",  //因为页面上需要一个isShow变量，所以  data:{  isShow:false//定义一个变量控制对话框的显示和隐藏，开局默认隐藏  },  //因为页面上需要？个事件处理函数: open和close  methods:{  open(){  this.isShow=true;  },  close(){  this.isShow=false;  }  }  })  </script>  </body>  </html>  运行结果： |

### (3). v-if v-else根据js中的变量值，控制两个元素二选一显示隐藏

a. <元素1 v-if="bool变量或js条件表达式">

<元素2 v-else >

b. 原理: 当new Vue首次扫描到这里或依赖的js变量发生变化时，new Vue都会自动计算v-if后""中的变量值或js表达式值。

1). 如果v-if后的值为true，则保留v-if所在元素，删除v-else所在元素

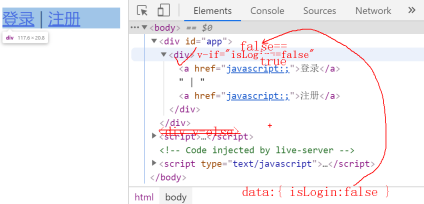
2). 如果v-if后的值为false，则删除v-if所在元素，保留v-else所在元素

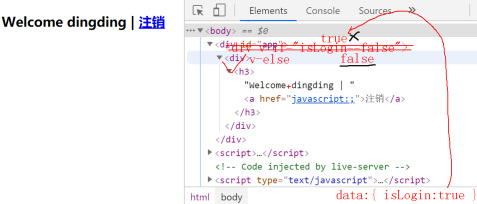
c. 强调:

1). 和程序中的if else一样，v-else后不要写="xxx"

2). v-if和v-else两个元素必须紧挨着，中间不能插入别的元素！

d. 示例: 切换登录和注销状态





7\_v-if\_v-else.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  </head>  <body>  <div id="app">  <!--根据变量值不同，显示不同的登录状态-->  <!--希望点击登录，切换到已登录状态  希望点击注销，切换回未登录状态-->  <!--希望未登录时只显示第一个div-->  <div v-if="isLogin==false">  <a href="javascript:;" @click="login">登录</a> | <a href="javascript:;">注册</a>  </div>  <!--已登录时只显示第二个div-->  <div v-else>  <h3>Welcome dingding | <a href="javascript:;" @click="logout">注销</a></h3>  </div>  </div>  <script>  new Vue({  el:"#app",  //因为页面上需要根据一个变量值显示不同的登录状态，所以data中要提供一个变量表示是否登录  data:{  isLogin:false,//开局，默认未登录  },  //因为页面上需要两个事件处理函数，所以methods中就要准备2个事件处理函数: login logout  methods:{  login(){  this.isLogin=true;  },  logout(){  this.isLogin=false;  }  }  })  </script>  </body>  </html>  运行结果： |

e. 鄙视: v-show 和 v-if的区别？

1). v-show通过添加属性display:none 实现元素隐藏 ——不改变DOM树结构，效率高

display:none时，所占空间消失

2). v-if 通过删除元素 实现隐藏 —— 改变DOM树，效率略低

### (4). v-if v-else-if v-else多个元素多选一显示

a. <元素1 v-if="bool变量或js条件表达式">

<元素2 v-else-if="另一个bool变量或js条件表达式">

... ...

<元素n v-else>

b. 原理: 当new Vue首次扫描到这里或依赖的js变量发生变化时，new Vue都会自动计算v-if后""中的变量值或js表达式值。

1). 如果v-if后的值为true，则保留v-if所在元素，删除其余所有v-else-if和v-else的元素

2). 如果v-if后的值为false，则删除v-if的元素，继续向后判断每个v-else-if后的条件。哪个v-else-if后的条件为true，就只保留这个v-else-if所在的元素。删除其余元素

3). 如果所有v-if和v-else-if中的条件都为false，则删除所有v-if和v-else-if，只保留v-else所在元素

c. 强调:

1). 和程序中的if else一样，v-else后不要写="条件"

2). v-if和v-else-if和v-else多个元素必须紧挨着，中间不能插入别的元素！

d. 示例: 根据PM25数值显示不同的表情

8\_v-else-if.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  </head>  <body>  <div id="app">  <!--希望根据pm25的数值，修改页面上的图片:  pm25<100，显示img/1.png  pm25<200, 显示img/2.png  pm25<300, 显示img/3.png  其余img/4.png  -->  <img v-if="pm25<100" src="img/1.png" alt="">  <img v-else-if="pm25<200" src="img/2.png" alt="">  <img v-else-if="pm25<300" src="img/3.png" alt="">  <img v-else src="img/4.png" alt="">  </div>  <script>  new Vue({  el:"#app",  //因为页面上需要根据pm25的数值控制显示哪张图片，所以:  data:{  pm25:360  }  })  </script>  </body>  </html>  运行结果: |

**VUE**

今日扩展视频：小程序->在线->VUE->day02

3. 双向绑定 v-model ... 防抖和节流——基础弱的同学不用看！

4. 双向绑定 v-model radio ... 选择飞机

5. 双向绑定 v-model select两级select级联选择省份和城市——基础弱的同学不用看！

6. 双向绑定 v-mode checkbox ... 多选 爱好——基础弱的同学不用看！

7. 双向绑定 v-model简写 监视函数 watch ——基础弱的同学不用看！

8. 绑定样式... 手机号验证 终极简写——基础弱的同学不用看！

作业: （扩展）高频笔试题 模拟VUE绑定原理

正课:

1. 指令:

2. 双向绑定:

3. 绑定样式:

4. 自定义指令:

五. 指令:

## 5. 根据数组内容，反复创建多个相同结构的元素: v-for

对应小程序视频: 小程序->在线->VUE->day01 4. v-for 遍历数组...

### (1). 语法：

<要反复创建的元素 v-for="(value, i ) of 数组/对象/字符串">

### (2). 原理:

当new Vue()首次扫描到这里时，或依赖的of后的数组内容发生变化时:

a. 自动遍历of后的数组或对象...

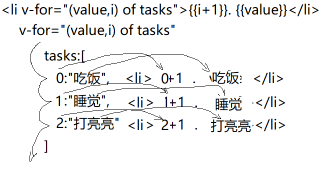
b. 每遍历一个成员，就

1). 将成员的值交给of前的value变量，

2) 将成员的下标位置交给of前的i变量

c. 反复创建v-for所在HTML元素，并将value和i的值替换到元素中可能发生变化的位置.

d. 在v-for所在的当前元素上以及当前元素的所有子元素中，都可以用value和i变量进行绑定。



### (3). 强调:

a. v-for必须写在要反复生成的元素上，而不能写在父元素上:

比如: 反复生成一个ul下的多个li元素，v-for应该写在子元素li上

b. of前的变量(value,i )可随意改名！但第一个变量始终接元素值，第二个变量始终接下标

### (4). 其实vue中的v-for of，统一了js中的for of和for in的功能！既可遍历数字下标，又可遍历自定义下标。

### (5) 坑: Vue中禁止使用下标，修改数组中的元素值。

原因: 数组的下标是数字，Vue只能监控自定义名称的属性，无法监控数字下标！

解决: 今后只要修改数组的内容，都必须用数组API修改。因为数组的所有API都是受到vue监控的。

比如: vm.数组[i]=新值 错误！页面不会改变

vm.数组.splice(i,1,新值) 正确

// 先删除i位置的1个旧元素，再在i位置添加一个新元素——替换

//复习第一阶段的数组函数

### (6) :key="下标"属性

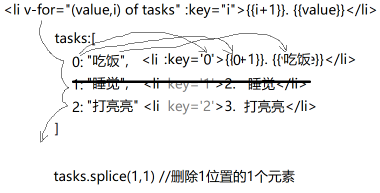
问题: v-for反复创建的多个元素，除了内容不同之外，默认没有任何差别！如果我们用程序修改了数组中某一个元素时，v-for不知道该对应修改哪个元素副本。于是，就会采用最笨的方法——删除所有旧元素，重新遍历整个数组，并重建整个列表。——效率极低

解决: 今后只要使用v-for，都必须同时绑定:key="i"属性。

结果:

1). 从此，v-for生成的每个元素副本身上，都多了一个唯一的标识key。值是不重复的。

2). 修改数组中任何一个元素值时，都只需要通过下标，找到这个元素值对应位置的DOM元素副本，只修改一个元素即可。不用重建整个列表。——效率高



### (7). 总结: 鄙视: 为什么v-for必须加:key？

a. 避免在修改某一个数组成员时，重建所有HTML元素副本

b. 如果加了:key，每次只需要修改一个数组元素对应的一个HTML元素副本即可——效率高

### (8). 示例: 使用v-for遍历数组和对象

1\_v-for.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  </head>  <body>  <div id="app">  <ul>  <!--因为要根据数组tasks的内容反复生成多个相同结构的li元素，所以应该在li元素上用v-for-->  <li v-for="(value,i) of tasks" :key="i">  <!--因为反复生成的内容中需要动态获得每个元素值和下标，所以，可以用of前的变量value和i用于绑定语法中动态生成内容-->  {{i+1}}. {{value}}  </li>  </ul>  <!--想遍历出lilei对象中每个属性的属性名和属性值-->  <ul>  <li v-for="(value,key) of lilei" :key="key">{{key}}: {{value}}</li>  </ul>  </div>  <script>  new Vue({  el:'#app',  data:{  //有一个任务列表，希望展现到页面上  tasks:["吃饭","睡觉","打亮亮"],//.length=3  // 0 1 2  lilei:{  sname:"Li Lei",  sage:11,  className:"初一2班"  }  }  })  </script>  </body>  </html>  运行结果: |

### (9). 其实: v-for还会数数: 仅根据一个数字，就可反复生成指定数量的HTML元素副本，且从1开始数数，一直数到给的数字为止

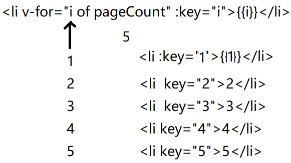
a. <要反复生成的元素 v-for="i of 数字">

b. 原理: 当new Vue()首次扫描到这里时，或依赖的数值发生变化时:

1). 数字是几，就会反复创建几个HTML元素副本

2). 每次创建副本时，都会将本次数到的数字保存到of前的变量中

3). 在当前元素及其子元素中变量i，可用于绑定。



### (10). 示例: 使用v-for生成分页按钮

2\_v-for.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  <style>  #pages{ list-style: none; }  #pages>li{  float:left;  padding:5px 10px;  border:1px solid #aaa;  margin:0 5px;  }  </style>  </head>  <body>  <div id="app">  <!--希望总页数是几，就反复生成几个分页分页按钮-->  <ul id="pages">  <li v-for="i of pageCount" :key="i">{{i}}</li>  </ul>  </div>  <script>  new Vue({  el:"#app",  data:{  pageCount:5,//分页时都会有一个变量保存总页数  }  })  </script>  </body>  </html>  运行结果: |

## 2. 绑定事件: v-on

对应小程序视频: 小程序->在线->VUE->day02 1. 事件绑定v-on ...

(1). 标准: <元素 v-on:事件名="事件处理函数(实参值列表)">

(2). 等效于: DOM中的<元素 on事件名="事件处理函数(实参值列表)">

(3). 重要差别: VUE中事件处理函数中的this，~~不再指向当前触发事件的元素~~。而是指向当前整个new Vue()对象！

(4). 简写:

a. "v-on:"可用"@"代替: <元素 @事件名="事件处理函数()">

b. 如果事件处理函数没有任何实参值，则可以省略():<元素 @事件名="事件处理函数">

(5). 其实: vue中的事件处理函数可以传参！

a. <元素 @事件名="事件处理函数(实参值列表)">

methods:{

事件处理函数(形参列表){ ... }

}

b. 示例: 点哪个div，哪个div喊谁疼！

3\_@.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  <style>  #app>div{  width:300px;  height:100px;  }  #app>div:hover{  box-shadow:0 0 5px red  }  </style>  </head>  <body>  <div id="app">  <!--因为点击两个div都能喊疼，只不过div的名字不同，所以两个div可共用同一个事件处理函数，只不过传入不同的div名字作为实参即可！-->  <div id="d1" style="background-color:#aaf" @click="say('d1')"></div>  <div id="d2" style="background-color:#ffa" @click="say('d2')"></div>  </div>  <script>  new Vue({  el:"#app",  data:{},  //因为界面中两个事件共用同一个事件处理函数，所以methods中只需要一个事件处理函数即可  methods:{  //但是，因为界面上的事件处理函数绑定时传入了一个实参值，所以methods中的事件处理函数就必须用一个形参变量接住实参值！  say(id){  alert(`${id} 疼! `)  }  }  })  </script>  </body>  </html>  运行结果： |

(6). 界面中，绑定时千万不要加（），因为加上空的（），表示什么参数都不传；浏览器一旦看到空的（），就什么

其实: 在vue中也可获得事件对象: 2种

a. 无需传其它实参值，只希望获得事件对象时：

1). 同DOM: 事件对象总是默认作为事件处理函数的第一个实参值自动传入

2). <元素 @事件名="事件处理函数">

methods:{

事件处理函数(e){ ... }

}

3). 问题: 如果需要同时传入实参值和获得事件对象，实参值和事件对象e传入的位置就会撞车！

<元素 @事件名="事件处理函数(实参值)">

event

methods:{

事件处理函数(e){ ... }

}

4). 错误的解决1: 在methods中在事件处理函数中e之前多加一个形参变量

<元素 @事件名="事件处理函数(实参值)">

event

methods:{

事件处理函数(形参1, e){ ... }

}

因为event对象，默认只能传给第一个形参变量，不会给之后的其它形参

5) 错误解决2: methods中交换事件处理函数的两个形参变量的顺序

<元素 @事件名="事件处理函数(实参值)">

event

methods:{

事件处理函数(e, 形参1){ ... }

}

因为第一个实参值，不会聪明到自动给第二个形参变量，依然给第一个形参，依然会和event发生冲突

6). 示例: 鄙视题: vue中如何获得鼠标位置: 4\_@\_e.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  <style>  #app>div{  width:300px;  height:100px;  }  #app>div:hover{  box-shadow:0 0 5px red  }  </style>  </head>  <body>  <div id="app">  <!--点哪个div的某个位置，就喊某个为疼！-->  <div id="d1" style="background-color:#aaf" @click="say"></div>  <div id="d2" style="background-color:#ffa" @click="say"></div>  </div>  <script>  new Vue({  el:"#app",  data:{},  methods:{  //因为想获得鼠标点击的位置，必须加形参e  //event  // ↓  say(e){  alert(`${e.offsetX},${e.offsetY}位置 疼! `)  }  }  })  </script>  </body>  </html>  运行结果： |

b. 关键词$event

如果同时传入实参值和事件对象：借助于vue一个关键词$event

DOM event

先↓

<元素 @事件名="事件处理函数(实参值, $event)">

e对应$event

methods:{

事件处理函数(形参1, e){ ... }

}

说明: $event和实参值可以交换位置。$event无论在实参列表中第几个位置，都可先获得事件对象event。不受位置影响！

但是如果html页面中实参和$event交换位置，则methods中的函数中的形参和e要对应交换，总之形参对应实参位置，e对应$event位置

c. 示例: 鄙视题: 如果同时传入自定义实参值和事件对象:

5\_@\_$event.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  <style>  #app>div{  width:300px;  height:100px;  }  #app>div:hover{  box-shadow:0 0 5px red  }  </style>  </head>  <body>  <div id="app">  <!--如果既需要传入实参值，又需要获得事件对象-->  <!-- DOM event-->  <!-- 先 ↓ -->  <div id="d1" @click="say($event,'d1')" style="background-color:#aaf" ></div>  <!-- DOM event-->  <!-- 先 ↓ -->  <div id="d2" @click="say($event,'d2')" style="background-color:#ffa" ></div>  </div>  <script>  new Vue({  el:"#app",  data:{},  methods:{  //因为界面上调用事件处理函数时，传入了两个实参，所以methods中定义事件处理函数时，也必须定义两个形参对应  say(e, id){  //鼠标位置相对事件所在元素左上角的距离  alert(`${id}的 ${e.offsetX},${e.offsetY}位置 疼! `)  }  }  })  </script>  </body>  </html>  运行结果: |

对应小程序视频: 小程序->在线->VUE->day02 2. 其它指令 绑定HTML内容

## 3. v-cloak防止用户短暂看到{{}}:

(1). 问题: 如果网速不好，js代码下载较慢，用户可能短暂看到页面上的{{}}语法！——尴尬！

(2). 解决: 2种方法

a. 用v-cloak属性:

1). 2步:

i. 先在页面上统一位置定义一个属性选择器:

[v-cloak]{ display:none } 意为: 凡是带有v-cloak属性的元素，都默认隐藏

斗篷/幕布

ii. 在网页中带有{{}}内容的元素上添加v-cloak属性，不需要任何属性值

2). 结果:

i. 如果网速慢, new Vue()所在的js暂时没有下载下来时，v-cloak选择器起作用，带有v-cloak和{{}}的元素暂时隐藏

ii. 当new Vue()所在的js文件，下载完成后，new Vue()自动找到所有v-cloak属性，自动移除这些元素上的v-cloak属性。这些带有{{}}的元素就显示出来了！

3). 强调: v-cloak属性是vue内置的指令名，不要随意修改！

b. 用v-text指令代替{{}}：

1). 1步: <元素 v-text="变量或js表达式"> </元素>

2). 原理: 当new Vue()首次扫描到这里或依赖的变量发生变化时

vue都会先计算""中js表达式的结果，然后用结果代替元素的内容

3). 为什么可以屏蔽{{}}，因为根本就没用{{}}。

4). 问题: {{}}的好处在于可以随意和其它写死的字符串拼接出一个新的字符串显示。但是用v-text，则无法将写死的字符串和变量或表达式随意拼接。

5). 解决: 在v-text中将写死的字符串和变量或表达式随意拼接，必须用模板字符串（反引号和${}）

<元素 v-text="`xxxx ${变量或js表达式}`"> </元素>

(3). 示例: 分别使用v-text和v-cloak解决短暂看到{{}}的问题:

6\_v-cloak\_v-text.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  <style>  /\* 凡是带有v-cloak属性的元素，开局默认暂时隐藏 \*/  [v-cloak]{  display:none  }  </style>  </head>  <body>  <div id="app">  <h3 v-cloak>用户名:{{uname}}</h3>  <h3 v-text="`积分: ${score}`"></h3>  </div>  <script>  //因为网速慢，new Vue()所在的js 2s后才到达页面  setTimeout(function(){  new Vue({  el:"#app",  data:{  uname:"dingding",  score:3000  }  })  },2000)  </script>  </body>  </html>  运行结果:    2s后才看到 |

## 4. v-html要绑定的内容是HTML片段

(1). 问题: 如果用{{}}或v-text绑定一段HTML片段，则不会将要绑定的内容交给浏览器解析，而是保持原样显示在页面上——不是我们想要的

(2). 为什么: {{}}和v-text底层其实相当于DOM中的.textContent

(3). 解决: 今后，只要绑定一段HTML片段，都要用v-html代替v-text和{{}}

(4). 为什么: v-html底层相当于DOM中的.innerHTML

<元素 v-html="变量或js表达式"> </元素>

(5). 原理: v-html会先将要绑定的内容交给浏览器解析，然后将解析后的可以给人看的内容替换元素的内容显示出来。在但文件组件里，样式不会应用在v-html内部

(6). 示例: 分别使用{{}}、v-text和v-html绑定HTML片段内容

7\_v-html.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  </head>  <body>  <div id="app">  <h3>{{html}}</h3>  <h3 v-text="html"></h3>  <h3 v-html="html"></h3>  </div>  <script>  new Vue({  el:"#app",  data:{  html:`来自&lt;&lt;<a href="javascript:;">新华社</a>&gt;&gt;的消息`  }  })  </script>  </body>  </html>  运行结果: |

## 5. v-once只在首次加载时绑定一次:

(1). 问题: 有些界面上显示的内容，只在首次加载时更新一次。之后几乎不会改变！如果这种元素也包含在虚拟DOM树中，无形中就增大了虚拟DOM树，可能影响遍历的速度

(2). 解决: 让这种元素只在首次加载时绑定一次，而且不加入虚拟DOM树中。

(3). 如何: <元素 v-once></元素>

(4). 原理: 凡是带有v-once的元素，只在首次加载时绑定一次，之后即使变量发生变化也不会改变。因为v-once的元素根本就没有加入到虚拟DOM树。

(5). 优点: 减少了虚拟DOM树中的内容，加快遍历的速度

(6). 示例: 使用v-once绑定页面加载完成时间

8\_v-once.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  </head>  <body>  <div id="app">  <h3 v-once>页面加载完成时间(一次性): {{time}}</h3>  <h3>当前系统时间(反复多次): {{time}}</h3>  </div>  <script>  var vm=new Vue({  el:"#app",  data:{  time:new Date().toLocaleString()  }  });  //希望每隔一秒重新修改new Vue()对象中time属性的值  setInterval(function(){  vm.time=new Date().toLocaleString();  },1000);  </script>  </body>  </html>  运行结果: |

## 6. 防止内容中{{}}被编译: v-pre

(1). 特殊情况：在元素内容中刚巧包含{{}}，但是不想被vue编译，只是想原样显示出来

(2). <元素 v-pre>xxx{{xxx}}xxx</元素>

(3). 结果: vue不会将内容中的{{}}解析为变量或js表达式，而是原样显示在页面上。

(4). 示例: 防止内容中的{{}}被编译:

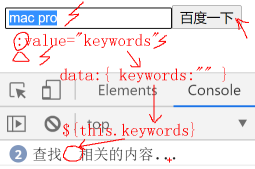
9\_v-pre.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  </head>  <body>  <div id="app">  <h3 v-pre>Vue框架用{{变量名}}方式标记页面中可能发生变化的位置</h3>  </div>  <script>  new Vue({  el:"#app"  })  </script>  </body>  </html>  运行结果: |

# 二. \*\*\*\*\*双向绑定:

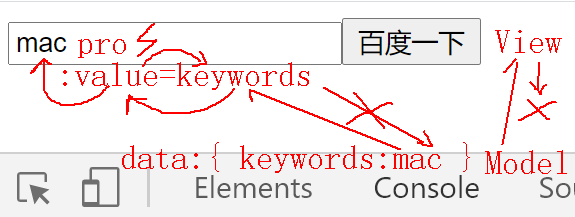
对应小程序视频: 小程序->在线->VUE->day02 3. 双向绑定v-model 绑定文本框

## 1. 问题: 以上12种绑定方式都无法获得用户在页面上文本框中输入的新值！



## 2. 原因: 因为前12中绑定或指令都是单向绑定:

(1). 单向绑定: 只能将程序中的变化，自动送到界面去，无法将界面上用户所做的修改，反向更新回程序中的变量中。（只能从Model->View，不能从View->Model）



## 3. 解决: 今后只要希望随时获得用户在页面上表单元素中所做的修改时，都要用双向绑定

(1). 双向绑定: 既能将程序中的变化，自动送到界面去，又能将界面上用户所做的修改，反向更新回程序中的变量中。（既能从Model->View，又能从View->Model）

## 4. 如何: 通常用于绑定表单元素，因为只有表单元素，用户才能在页面上修改！

<表单元素 v-model:value="自定义变量">

data:{

自定义变量: 初始值

}

## 5. 示例: 点按钮，获得文本框中输入的关键词，执行搜索操作

10\_v-mode.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  </head>  <body>  <div id="app">  <!--界面中哪个可能发生变化? 文本框的内容可能发生变化  因为文本框的内容是value属性，要绑定value属性值，应该用:绑定,自定义变量名为keywords-->  <!--但是，因为:value只能单向绑定(M->V)，不能双向绑定(V->M)，所以要改为v-model:value，才能双向绑定(V->M)-->  <!--点哪里可能触发查找事件? button-->  <input v-model~~:value~~="keywords"><button @click="search">百度一下</button>  </div>  <script>  new Vue({  el:"#app",  //既然页面上需要一个keywords变量，data中就应该定义一个keywords变量支持页面  data:{  keywords:"mac"//开局是空字符串  },  methods:{  //因为页面上需要一个事件处理函数search()，所以methods中就要定义一个search()函数  search(){  //如果keywords中收到的关键词不是空字符串，则执行查找操作  if(this.keywords.trim()!==""){  console.log(`查找 ${this.keywords.trim()} 相关的内容...`)  }  }  }  })  </script>  </body>  </html>  运行结果: |

## 6. 双向绑定原理:

在单向绑定原理（访问器属性+虚拟DOM树）基础上，又自动为表单元素绑定了onchange事件

onchange事件是DOM中常用事件，意为当内容发生改变时自动触发！

v-model会自动为当前表单元素绑定: @input或@change。当当前表单元素的value值发生变化时，自动触发事件，调用事件处理函数，自动修改data中的变量。

## 7. 示例: 使用事件模拟v-model的原理:

7\_v-model2.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>    <script src="js/vue.js"></script>  </head>  <body>    <!--1. 做界面-->    <div id="app">      <!--1.1 找可能发生变化的位置        本例中: 文本框的内容可能发生变化-->      <input type="text" @input="myInput" :value="keywords">      <!--1.2 找触发事件的元素        本例中: button触发事件-->      <button @click="search">百度一下</button>    </div>    <script>      //2. 创建new Vue()对象      var vm=new Vue({        el:"#app",        //3. 创建模型对象        //3.1 因为界面上只需要一个变量，所以        data:{          keywords:"" //保存用户在文本框中输入的内容，开局，用户什么都没输入，默认""        },        //3.2 因为界面上需要一个函数        methods:{          search(){            console.log(`查找 ${this.keywords} 相关的内容...`);          },          myInput(e){            //this不指当前触发事件的元素了            //vue中所有的this都指向当前new Vue()对象本身！            //获得当前文本框的内容            var value=e.target.value;            //        当前文本框 内容            //将文本框的内容自动修改到data中的keywords变量上            this.keywords=value;          }        }      })    </script>  </body>  </html>  运行结果: |

## 8. 实现按回车搜索，边输入边搜索:

### (1). 按回车搜索:

a. <input @keyup="事件处理函数">

b. 问题: 虽然可以实现边输入边搜索，但是，按任意键都执行搜索，不是我们想要的。我们希望只有按回车才执行搜索

c. 解决: vue中提供了一套事件修饰符:

1). 什么是: 对触发事件的条件或默认行为加以限制

2). 比如: @keyup.13，表示只有13号回车键才能触发事件

3). 扩展视频中:

.stop -> e.stopPropagation 停止冒泡

.prevent -> e.preventDefault 阻止默认行为

d. 问题: 如果只有13号键可以触发事件，那么，如何实现边输入边搜索呢？

e. 解决: 利用vue提供的"监听函数"

### (2). 实现边输入边搜索: 监听函数

a. 什么是"监听函数": 当data中某个变量一改变，就会立刻自动触发的特殊函数。

b. 何时: 今后，只要希望data中的一个变量一变，立刻自动执行一个操作时，都要用监听函数！

c. 如何:

new Vue({  
 el:"#app",

data:{ ... 变量: 值, ... },

methods:{

事件处理函数(){

}

},

watch:{ //专门包含所有监听函数的区域

变量名(){

//只有data中同名变量的值发生变化时自动触发！

}

}  
 })

### (3). 示例: 实现边输入边搜索，按回车也能搜索

8\_v-model3.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>    <script src="js/vue.js"></script>  </head>  <body>    <!--1. 做界面-->    <div id="app">      <!--1.1 找可能发生变化的位置        本例中: 文本框的内容可能发生变化-->      <!--1.2 找触发事件的元素        本例中:        button触发单击事件        input还可触发键盘抬起事件，但是，只允许13号键触发事件-->      <input type="text" v-model:value="keywords" @keyup.13="search">      <button @click="search">百度一下</button>    </div>    <script>      //2. 创建new Vue()对象      var vm=new Vue({        el:"#app",        //3. 创建模型对象        //3.1 因为界面上只需要一个变量，所以        data:{          keywords:"" //保存用户在文本框中输入的内容，开局，用户什么都没输入，默认""        },        //3.2 因为界面上需要一个函数        methods:{          search(){            console.log(`查找 ${this.keywords} 相关的内容...`);          }        },        watch:{//专门保存所有监听函数的区域          keywords(){//专门监听data中keywords变量的函数            //希望只要keywords变量发生改变，就自动搜索            this.search();          }        }      })    </script>  </body>  </html>  运行结果: |

## 9. 问题: 有些表单元素用户在修改时value是固定不变的！改的是其他属性:

比如: 性别: <input type="radio" value="1" name="sex">男

<input type="radio" value="0" name="sex">女

## 10. 解决: 分析，用户操作这些元素时，到底改的是哪个属性？

### 1). 比如: 用户选中或不选中radio，改的是radio的checked属性！

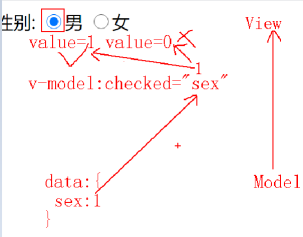
### 2). 所以: v-model应该绑定在checked属性上

<input type="radio" value="固定值" name="分组名" v-model:checked="变量名">

### 3). 原理:

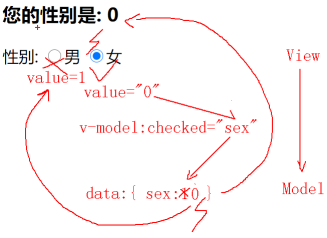
#### i. 开局: 将程序中的变量值显示到页面上: radio v-model:checked="变量名"

v-model会用变量值和当前radio的value做比较，如果变量值等于当前radio的value，则当前radio选中。否则，如果变量值不等于radio的value，则当前radio不选中



#### ii. 当用户切换了radio的选中状态时:

如果当前radio选中，则v-model自动将当前radio的value值传到程序中保存在data中的变量里。如果当前radio未选中，则v-model什么也不做！



## 11. 示例: 在vue中选择性别:

12\_v-model\_radio.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  </head>  <body>  <div id="app">  <h3>您的性别是: {{sex}}</h3>  <!--因为用户选择性别时同时改变两个radio的checked属性，所以两个radio上都要写v-model:checked=xxx  又因为无论用户点哪个input，修改的都是性别这一个属性值，所以两个radio绑定时都绑定sex一个变量-->  性别:  <label><input type="radio" value="1" name="sex" v-model~~:checked~~="sex">男</label>  <label><input type="radio" value="0" name="sex" v-model~~:checked~~="sex">女</label>  </div>  <script>  new Vue({  el:"#app",  //因为页面上只需要一个变量sex标记性别  data:{  sex:1, //开局为1 表示男  }  })  </script>  </body>  </html>  运行结果: |

## 12. 总结:

(1). 绑定value属性的表单元素:

<input type="text"> <textarea></textarea> <select>

(2). 绑定checked属性: <input type="radio"> <input type="checkbox">

13. 简写: 其实v-model后不用写任何属性名，v-model就可根据自己所在的元素自动判断该绑定哪个属性，比如:

(1). <input type="text" v-model~~:value~~="变量">

(2). <input type="radio" value="1" name="sex" v-model~~:checked~~="变量">

# 三. 绑定样式: 2种样式

对应小程序视频: 小程序->在线->VUE->day02 8. 绑定样式 style

## 1. 内联样式：

(1). 不好的做法: 把style属性当做一个巨大的字符串，仅用一个变量来绑定！

<元素 :style="xxxStyle">

data:{

xxxStyle:" bottom:50px; margin-left:-64px "

}

结果: 很难只修改其中一个css属性的值

**(2). 好的做法: vue支持用对象语法绑定style属性: 2种:**

a. 为每个可能发生变化的css属性分别指定变量:

<元素 :style="{ css属性名1: 变量1, css属性名2: 变量2 , .... }"

data:{

变量1: css属性值1

变量2: css属性值2

}

运行时自动翻译: <元素 style=" css属性名1:css属性值1; css属性名2: css属性值2; ..."



问题: 如果要修改的css属性比较多，data中就会产生很多变量，乱！

b. 其实：可以把整个style看做一个大的对象放在一个变量中。每个要修改的css属性都是这对象中的属性.

<元素 :style="xxStyle"

data:{

//先翻译为字符串: " css属性名1:css属性值1; css属性名2: css属性值2"

xxStyle:{

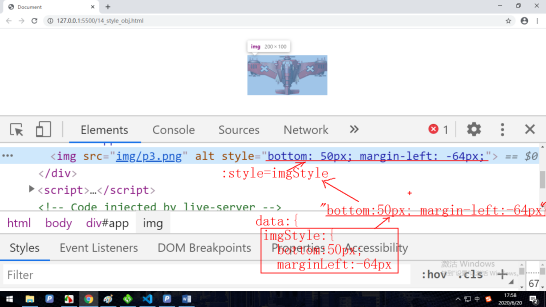
css属性名1: css属性值1

css属性名2: css属性值2

}

}

运行时自动翻译: <元素 style=" css属性名1:css属性值1; css属性名2: css属性值2; ..."



(3). 问题: 万一有一些不需要变化的内联样式:

a. 其实: 不带:的写死的style属性可以和带:动态生成的style属性共存！

b. 结果: 不带:的写死的style属性和带:动态生成的style属性合并为最终的一个style属性应用的元素上！



(4). 示例: 分别使用两种方法绑定小飞机的style属性，通过键盘方向键操控飞机飞行

13\_style.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  <style>  img{  position:fixed;  }  </style>  </head>  <body>  <div id="app">  <!--小飞机哪些属性可能发生变化  不好的做法: 整个style都会发生变化  好的做法: 两个css属性可能发生变化: bottom和marginLeft-->  <img style="left:50%;" :style="{bottom:imgBottom, marginLeft:imgML}" src="img/p3.png" alt="">  </div>  <script>  var vm=new Vue({  el:"#app",  data:{  //因为界面上需要两个变量: imgBottom和imgML  imgBottom:"50px", //必须加单位！  imgML:"-64px"  }  })  //当在浏览器窗口上按鼠标方向键时，控制飞机用  window.onkeydown=function(e){  if(e.keyCode==38){//↑  //增大bottom  vm.imgBottom=`${parseInt(vm.imgBottom)+10}px`  }else if(e.keyCode==37){//←  //减小margin-left  vm.imgML=`${parseInt(vm.imgML)-10}px`;  }else if(e.keyCode==39){//→  //增大margin-left  }else if(e.keyCode==40){//↓  //减小bottom    }  }  </script>  </body>  </html>  运行结果: |

14\_style\_obj.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  <style>  img{  position:fixed;  left:50%;  }  </style>  </head>  <body>  <div id="app">  <!--小飞机哪些属性可能发生变化  不好的做法: 整个style都会发生变化  好的做法: 两个css属性可能发生变化: bottom和marginLeft-->  <img style="left:50%;" :style="imgStyle" src="img/p3.png" alt="">  </div>  <script>  var vm=new Vue({  el:"#app",  data:{  imgStyle:{  bottom:"50px", //必须加单位！  marginLeft:"-64px"  }  }  })  //当在浏览器窗口上按鼠标方向键时，控制飞机用  window.onkeydown=function(e){  if(e.keyCode==38){//↑  //增大bottom  vm.imgStyle.bottom=  `${parseInt(vm.imgStyle.bottom)+10}px`  }else if(e.keyCode==37){//←  //减小margin-left  vm.imgStyle.marginLeft=  `${parseInt(vm.imgStyle.marginLeft)-10}px`;  }else if(e.keyCode==39){//→  //增大margin-left  }else if(e.keyCode==40){//↓  //减小bottom    }  }  </script>  </body>  </html>  运行结果: 同上 |

Day03

## 2. 绑定class属性:

### (1). 何时: 批量修改多个css属性值时

### (2). 不好的方式: 将整个class看做一个普通的字符串属性绑定

缺点: 极其不便于修改其中某一个class

### (3). 好的方式: 2种:

#### a. 对每个可能动态变化的class都执行一个变量

临时创建一个匿名对象的意思

绑定 ↓

#### 1). <元素 :class="{ class名1: bool表达式或变量, class名2: bool表达式或变量 }"

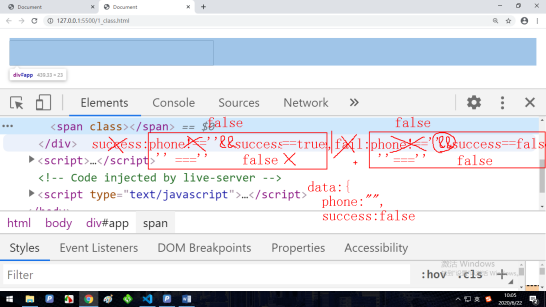
#### 2). 原理: new Vue()首次扫描到这里时，或依赖的变量发生变化时

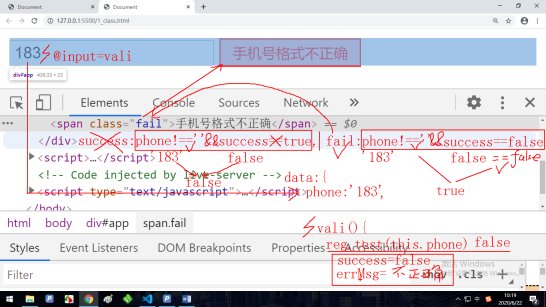
i. 计算每个class名之后的bool表达式或变量值

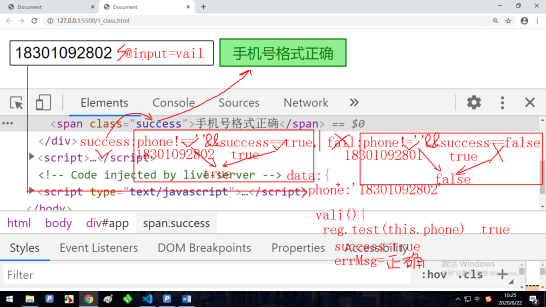
ii. 如果一个class名之后的bool表达式或变量值为true，则该class会出现在最终的元素上发挥作用——亮灯

iii. 如果一个class名之后的bool表达式或变量值为false，则该class不会出现在最终的元素上，不会发挥作用！——灭灯

#### 3). 示例: 验证手机号格式是否正确——带css样式







1\_class.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  <style>  /\* 验证通过时的样式 \*/  .success{  border:1px solid green;  background-color:lightGreen;  color:green;  padding:3px 10px;  font-size:12px;  }  /\* 验证失败的样式 \*/  .fail{  border:1px solid red;  background-color:pink;  color:red;  padding:3px 10px;  font-size:12px;  }  </style>  </head>  <body>  <div id="app">  <!--验证前后界面上哪里可能发生变化: 3处  1. input的内容 - phone 存储用户输入的手机号  开局: phone:""  2. span的class - success 存储一个bool值表示是否验证通过  开局: success:false  3. span的内容 - errMsg 存储span中要显示的提示信息  开局: errMsg:""-->  <!--只要用户在input中输入内容时，触发验证，所以  绑定oninput事件，执行vali函数，验证文本框中新内容-->  <input v-model="phone" @input="vali">  <!--如果用户没有在文本框中输入内容，则根本不用应用任何class-->  <!--只有用户在文本框中输入了内容且验证结果通过，就启用success class，禁用fail  否则如果用户在文本框中输入了内容且验证结果未通过，就启用fail class，禁用success-->  <span :class="{  success:phone.trim()!==''&&success==true,  fail:phone.trim()!==''&&success==false  }">{{errMsg}}</span>  </div>  <script>  new Vue({  el:"#app",  //因为界面上需要三个变量所以  data:{  phone:"",  success:false,  errMsg:""  },  //因为界面上需要一个函数  methods:{  vali(){  console.log(this.phone);  //用正则验证phone变量中手机号的格式是否正确  //先定义手机号正则  var reg=/^1[3-9]\d{9}$/;  //在用正则验证phone的内容是否符合格式要求  var result=reg.test(this.phone.trim());  //如果phone的内容为空  if(this.phone.trim()===""){  //就清除errMsg的内容，等于清除span的内容,什么也不显示  this.errMsg="";  }else if(result==true){//否则如果验证通过:  //就修改success变量为true，表示验证通过  this.success=true;  //就修改errMsg变量为手机号格式正确  this.errMsg="手机号格式正确";  }else{//否则如果验证未通过:  //就修改success变量为false，表示验证未通过  this.success=false;  //就修改errMsg变量为手机号格式不正确  this.errMsg="手机号格式不正确";  }  }  }  })  </script>  </body>  </html>  运行结果: |

#### b. 对整个class指定一个变量对象，一个变量对象中包含多个动态变化的class

#### 1). 怎么做

<元素 :class="变量名">

data:{

变量: {

class1:true或false,

class2:true或false

}

}

##### 2). 示例: 使用一个class变量修改上例:

2\_class\_obj.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  <style>  .msg{  padding:3px 10px;  font-size:12px;  }  /\* 验证通过时的样式 \*/  .success{  border:1px solid green;  background-color:lightGreen;  color:green;  }  /\* 验证失败的样式 \*/  .fail{  border:1px solid red;  background-color:pink;  color:red;    }  </style>  </head>  <body>  <div id="app">  <!--验证前后界面上哪里可能发生变化: 3处  1. input的内容 - phone 存储用户输入的手机号  开局: phone:""  2. span的class - spanClass 用对象结构存储两个class success fail的bool值  开局: { success:false, fail:false }  3. span的内容 - errMsg 存储span中要显示的提示信息  开局: errMsg:""-->  <!--只要用户在input中输入内容时，触发验证，所以  绑定oninput事件，执行vali函数，验证文本框中新内容-->  <input v-model="phone" @input="vali">  <!--新方法，不要在HTML中做任何判断条件-->  <span class="msg" :class="spanClass">{{errMsg}}</span>  </div>  <script>  new Vue({  el:"#app",  //因为界面上需要三个变量所以  data:{  phone:"",  spanClass:{  success:false, //都灭灯  fail:false, //都灭灯  },  errMsg:""  },  //因为界面上需要一个函数  methods:{  vali(){  //用正则验证phone变量中手机号的格式是否正确  //先定义手机号正则  var reg=/^1[3-9]\d{9}$/;  //在用正则验证phone的内容是否符合格式要求  var result=reg.test(this.phone.trim());  //如果phone的内容为空  if(this.phone.trim()===""){  //就清除errMsg的内容，等于清除span的内容,什么也不显示  this.errMsg="";  //清除success和fail两个class的残留  this.spanClass={ success:false, fail:false }  }else if(result==true){//否则如果验证通过:  //就修改spanClass内的success值为true，同时修改fail值为false  this.spanClass={ success:true, fail:false };  //就修改errMsg变量为手机号格式正确  this.errMsg="手机号格式正确";  }else{//否则如果验证未通过:  //就修改spanClass内的success值为false，同时修改fail值为true  this.spanClass={ success:false, fail:true };  //就修改errMsg变量为手机号格式不正确  this.errMsg="手机号格式不正确";  }  }  }  })  </script>  </body>  </html>  运行结果:  C:\xampp\htdocs\dd\VUE\day03\2_class_obj1.pngC:\xampp\htdocs\dd\VUE\day03\2_class_obj2.png |

### (4). 如果一个元素上既有不变的class，又有动态改变的class:

<元素 class="固定不变的class们" :class="可能动态改变的class们">

最终两个class会合并为一个class，作用到元素上。

# 二. 自定义指令:

对应小程序视频: 小程序->在线->VUE->day02 9. 自定义指令vue directive。。。

## 1. 何时: 希望在页面加载时就对元素做一些初始化的DOM操作:

比如: 自动获得焦点

## 2. 如何: 2步

### (1). 向Vue家里添加一个自定义指令

Vue.directive("指令名",{ //强调: 指令名千万不要带v-前缀！

//回调函数，当new Vue()将当前元素添加到页面上之后，自动执行

//当首次加载页面时，代用当前指令的元素被渲染到页面之后

插入后

inserted(domElem){

//domElem会自动接住带有当前指令的dom元素对象

对当前这个带有自定义指令的DOM元素，自动执行

对当前DOM元素对象domElem执行一些DOM相关的初始化操作，比如获得焦点

}

})

### (2). 使用自定义指令:

<元素 v-自定义指令名> //强调: 使用自定义指令时必须加v-前缀！

## 3. 结果:

(1). new Vue()首次扫描到这里时，会去Vue家里找有没有同名的自定义属性。

(2). 如果找到，就会自动调用自定义指令中的inserted()函数，并将当前带有自定义指令的DOM元素对象，传给inserted()第一个形参。

(3). 在inserted()回调函数内，对当前dom元素执行的操作，就会反应到页面上。

## 4. 示例: 使用自定义指令，让文本框自动获得焦点:

3\_directive.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  <script>  //向vue大家庭中添加一个新指令my-focus  Vue.directive("my-focus",{ //因为HTML语法不区分大小写，所以如果名字由多个单词组成，强烈推荐用-分割，而不用驼峰命名  //当当前元素被添加到页面上之后，当前元素自动获得焦点  inserted(domElem){  domElem.focus(); //复习DOM中四天上午  }  })  </script>  </head>  <body>  <div id="app">  <!--希望文本框input在页面加载时就自动获得焦点  1. vue官方没有定义让元素自动获得焦点的指令  2. 的确可以写文本框.focus()，但是vue中没地方写DOM代  唯一可以名正言顺写DOM代码的地方——自定义指令中-->  <input v-my-focus><button>百度一下</button>  </div>  <script>  new Vue({  el:"#app"  })  </script>  </body>  </html>  运行结果: |

# 三. 计算属性:

对应小程序视频: 小程序->在线->VUE->day02 10 计算属性

## 什么是:

自己不实际保存属性值，每次使用属性时，都临时根据所依赖的变量动态计算出结果应用到页面上。

## 2. 何时:

如果一个属性值不是现成的，需要经过复杂的计算过程才能获得，都要用计算属性

## 3. 如何:

### (1). 在new Vue()中添加新成员:

new Vue({

el:"#app",

data:{ ... },

methods:{ 事件处理函数 },

computed:{ 专门保存计算属性

属性名(){

return 根据其他变量动态计算出新属性值

}

}

})

### (2). 使用计算属性:

<元素>{{ 属性名 }}</元素> //计算属性使用时一定不要加()！

## 4. 问题: 既然是一个函数，为什么不放在methods中？

答: 因为放在methods中的普通函数，每调用一次，都会重新执行一次复杂计算，计算结果不会被vue缓存并反复使用。

## 计算属性的优点:

计算属性首次计算的结果会被vue缓存起来，即使反复使用计算属性也不会重复计算。除非依赖的变量发生了变化，才被迫重新计算。但是新值依然会被缓存，并重复使用！

## 6. 总结:

a. 今后如果侧重于做一件事儿，而不太关心返回值，就用methods中的普通函数！

b. 今后如果更侧重于使用函数返回的结果值，就用computed计算属性

## 7. 示例: 使用计算属性实现购物车总价:

4\_computed.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  </head>  <body>  <div id="app">  <!--希望显示购物车商品列表和购物车总价-->  <h3>总价: ¥{{total.toFixed(2)}}</h3>  <ul>  <li v-for="(p,i) of cart" :key="i">  {{p.pname}} | ¥{{p.price.toFixed(2)}} | {{p.count}} | ¥{{(p.price\*p.count).toFixed(2)}}  </li>  </ul>  <h3>总价: ¥{{total.toFixed(2)}}</h3>  </div>  <script>  new Vue({  el:"#app",  data:{  cart:[  {pname:"华为", price:5588, count:2},  {pname:"小米", price:3588, count:3},  {pname:"苹果", price:8588, count:1},  ]  },  methods:{    },  computed:{  //因为购物车总价不是现成的，需要经过复杂的遍历过程才能求出  //所以，用计算属性。  total(){  console.log(`调用了一次total()`)  var result=0;  for(var p of this.cart){  result+=p.price\*p.count;  }  return result;  }  }  })  </script>  </body>  </html>  运行结果: |

# 四. 过滤器:

对象小程序视频: 小程序->在线->VUE->day03 3. 过滤器

## 什么是:

专门将变量的原始值经过加工后再显示 的一种特殊的函数

## 为什么:

因为有些从服务器端拿到的变量值，不能直接给人看

比如: 日期和时间 性别

服务器端存的是: 毫秒数 1和0

人希望看到: 年月日时分秒 男和女

## 何时:

只要变量的原始值不能直接给人看，需要加工后才能给人看时，都用过滤器

## 如何: 2步

(1). 向vue大家庭中添加过滤器(先定义过滤器，后通过new Vue去扫描整个页面再调用过滤器执行功能，自定义指令同理)

进

↓

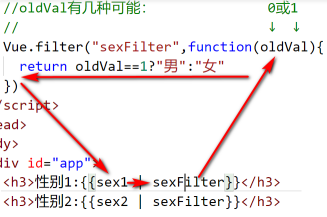
Vue.filter("过滤器名", function( oldVal ){

←出 return 根据oldVal加工后获得的新值

})

(2). 使用过滤器: <元素>{{ 变量 | 过滤器 }}</元素>

## 示例: 使用过滤器加工性别0和1后再显示



5\_filter.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  <script>  //因为项目中所有性别都需要先加工再显示  //将来 {{ sex | sexFilter }}  //oldVal有几种可能： 0或1  // ↓ ↓  Vue.filter("sexFilter",function(oldVal){  return oldVal==1?"男":"女"  })  </script>  </head>  <body>  <div id="app">  <h3>性别1:{{sex1}}</h3>  <h3>性别2:{{sex2}}</h3>  <h3>性别1:{{sex1==1?"男":"女"}}</h3>  <h3>性别2:{{sex2==1?"男":"女"}}</h3>  <h3>性别1:{{sex1 | sexFilter}}</h3>  <h3>性别2:{{sex2 | sexFilter}}</h3>  </div>  <script>  new Vue({  el:"#app",  data:{  sex1:1,  sex2:0  }  })  </script>  </body>  </html>  运行结果: |

## 6. 过滤器可以带参数，根据不同的参数值，过滤出不同的结果:

(1). 定义过滤器时:

Vue.filter("过滤器名", function(oldVal, 自定义形参){

//接住的是过滤器之前的变量的原始值

//自定义形参可接住调用过滤器时传入的实参值

})

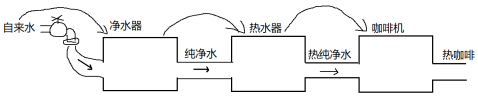
(2). 使用过滤器时:

<元素>{{ 变量 | 过滤器(自定义实参值) }}</元素>

## 7. 多个过滤器还可以连用: <元素>{{ 变量 | 过滤器1 | 过滤器2 | ... }}</元素>

强调: 只有第一个过滤器才能收到变量的原始值

之后其它过滤器收到的都不是原始值，而是上一个相邻的过滤器加工后的中间产物



## 8. 示例: 定义带参数过滤器过滤出不同语言种类的男和女，并且为性别添加图标

6\_filter.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  <script>  //希望根据输入的语言种类不同，加工出不同语言的男和女  // 预留 自定义  Vue.filter("sexFilter",function(oldVal, 语言种类){  //如果语言种类传入"英文",则加工出Male Female  if(语言种类=="英文"){  return oldVal==1?"Male":"Female"  }else{//否则如果传入的语言种类不是"英文"，则默认加工出男和女  return oldVal==1?"男":"女"  }  });  //为了能够给性别末尾追加图标:  //可能: {{sex | sexIcon }} 0和1  //也可能: {{ sex | sexFilter | sexIcon}} 男和女  //也可能: {{ sex | sexFilter("英文") | sexIcon}} Male和Female  // 预留  Vue.filter("sexIcon",function(oldVal){  //oldVal可能6种: 1和0 男和女 Male和Female  //如果oldVal是1或0，则直接返回♂和♀  if(oldVal==1){  return "♂"  }else if(oldVal==0){  return "♀"  }else if(oldVal=='男'||oldVal=="Male"){//否则如果oldVal不是0或1则需要将♂和♀拼接到现有oldVal结尾  return oldVal+"♂"  }else{  return oldVal+"♀"  }  })  </script>  </head>  <body>  <div id="app">  <h3>性别1:{{sex1 | sexIcon}}</h3>  <h3>性别2:{{sex2 | sexIcon}}</h3>  <h3>性别1:{{sex1 | sexFilter | sexIcon}}</h3>  <h3>性别2:{{sex2 | sexFilter | sexIcon}}</h3>  <!-- 自定义 -->  <h3>性别1:{{sex1 | sexFilter("英文") | sexIcon}}</h3>  <h3>性别2:{{sex2 | sexFilter("英文") | sexIcon}}</h3>  </div>  <script>  new Vue({  el:"#app",  data:{  sex1:1,  sex2:0  }  })  </script>  </body>  </html>  运行结果： |

# 五. axios:

对应小程序视频: 小程序->在线->VUE->day03 4. axios

1. 什么是:

专门在各种平台都能发送ajax的基于promise的函数库

1. 为什么: jQuery不是已经又$.ajax()

今后框架开发，几乎不用jquery的。仅仅为了使用一个函数$.ajax()发送ajax请求，就把包含几十个上百个函数的jQuery函数库引入进来，极其不划算！

今后其它非jquery的前端框架，急需要一种专门发送ajax请求的函数库。

1. 何时:

只要在vue中发送ajax请求，则都用axios。

但是其实axios绝不仅局限于vue中，普通网页中甚至nodejs中都可以使用

1. 如何:

(1). 发送get请求:

axios.get("服务器端接口地址", {

params:{ 变量名:值, ...: ..., }

}).then(function(result){ //.then会在成功接收到服务器端返回的结果时自动触发

//坑: result不是服务器端返回的数据

//result.data才是服务器端返回的数据！

})

(2). 发送post请求：

axios.post("服务器端接口地址","变量1=值1&变量2=值2&...")

.then(function(result){ //.then会在成功接收到服务器端返回的结果时自动触发

//坑: result不是服务器端返回的数据

//result.data才是服务器端返回的数据！

})

5. 问题: 项目中很多地方都要写ajax请求，如果每次都从http://开始写服务器端接口的完整路径，太麻烦了！

6. 解决: 其实一个项目中的绝大多数接口的域名部分是完全一样的。

(1). axios提供了一个位置，专门保存所有接口共用的相同部分的域名。

axios.defaults.baseURL="http://基础域名"

(2). 用axios发送请求时，只写接口的相对路径即可，运行时axios自动将保存的基础域名部分和相对接口名称拼接为完整的接口地址，再发送请求。

axios.get("/接口名",...).then(...)

7. 示例: 使用axios向东哥新浪云学子商城服务端接口发送三种请求:

7\_axios.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/axios.min.js"></script>  </head>  <body>  <script>  //配置所有接口相同的基础域名路径部分  axios.defaults.baseURL="http://xzserver.applinzi.com";  axios.get("/index")  .then(function(result){  console.log(result.data);  });  axios.get("/details",{  params:{ lid:5 } //自动翻译为?lid=5，添加到url结尾  }).then(function(result){  console.log(result.data);  });  axios.post(  "/users/signin",  "uname=dingding&upwd=123456"  ).then(function(result){  console.log(result.data);  });  </script>  </body>  </html>  运行结果： |

8. 问题: axios.get().then(function(){ ... })this->window，但是今后我们希望new Vue()中所有this都要指向当前new Vue()对象

9. 解决: 今后axios.get().then(function(){ ... })

↓

axios.get("/接口名",...).then(reuslt=>{ ... })

结果: .then()回调函数中的this和外部的this保持一致了！

# 六. vue对象生命周期:

对应小程序视频: 小程序->在线->VUE->day03 5. 生命周期

## 什么是:

一个VUe对象从创建到所有页面内容加载完成所经历的过程

## 2. 包括: 4个阶段:

(1). 创建 create 必经

a. 创建new Vue()对象，在new Vue()对象中创建data对象并请保镖

b. 暂时不扫描DOM树，暂时没有虚拟DOM树

(2). 挂载 mount 必经

a. 扫描DOM树，生成虚拟DOM树，并首次挂载数据到页面元素上

-------------------------------------------

(3). 更新 update 当data中的变量被修改时才触发

(4). 销毁 destroy 还有调用$destroy函数销毁当前组件时，才触发 —— 很少用

## 3. 为什么要学习生命周期:

(1). jquery中，在页面首次加载时有很多初始化操作，比如自动发送ajax请求，比如自动绑定首屏数据。这些操作都写在$(function(){ ... })中。凡是写在$(function(){ ... })中的代码，会在DOM内容加载完成后自动执行！

(2). 问题: vue框架中几乎不用jquery，自然也就没有$(function(){ ... })，比如自动发送ajax请求获取首屏数据的代码应该写在哪儿呢？

(3). 错误: 将axios请求的代码放在new Vue()外部。

因为将来vue采用组件化开发，规定一个组件的所有js代码必须放在这个组件的对象内部！组件对象之外不允许出现任何逻辑相关的代码

(4). 正确: 将要执行的页面初始化js代码，绑定到vue的某个生命周期阶段，自动执行！

## 4. 何时: 今后只要希望在vue对象创建和挂载过程中，自动执行某些操作时，都要用生命周期

## 5. 如何: vue对象有四个生命周期，每个生命周期前后都有一对儿（2个）钩子函数(回调函数)

beforeCreate() 在new Vue()创建前自动触发，没有data和变量呢，无法修改data中的变量——不适合发送首屏ajax请求

### (1). 创建阶段 create

created() 创建完new Vue()和data对象之后自动触发

a. 可以操作data中的变量

b. 但是暂时无法执行DOM相关的操作，因为还没有扫描DOM树

beforeMount() 在开始扫描DOM树之前自动触发

a. 也可以操作data中的变量

b. 但是暂时也无法执行DOM相关的操作，因为还没有扫描DOM树

### (2). 挂载阶段 mount

mounted() 在创建完虚拟DOM树，首次挂在页面内容完成之后自动触发

a. 已经有了data对象，可以操作data中的变量

b. 已经有了虚拟DOM树，且数据已经显示在页面上了，可以执行DOM相关操作

总结: mounted()才是最适合做页面初始化工作

beforeUpdate() 在开始修改data中的变量之前自动触发

### (3). 更新阶段 update

updated() 修改完data中的变量之后自动触发

beforeDestroy() 在开始销毁当前组件之前自动触发触发

### (4). 销毁阶段 destroy

destroyed() 在销毁当前组件之后自动触发



## 6. 示例: 演示一个vue组件的4个生命周期和8个钩子函数

强调: 页面上<div id="app">中必须绑定并使用了data中的products变量才能出发更新阶段

8\_lifecycle.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Document</title>  <script src="js/vue.js"></script>  <script src="js/axios.min.js"></script>  </head>  <body>  <!--希望页面加载时，就加载学子商场首页的六个商品-->  <div id="app">  <ul>  <li v-for="(p,i) of products" :key="i">  {{p.title}} | ¥{{p.price}} | {{p.details}}  </li>  </ul>  </div>  <script>  //配置  axios.defaults.baseURL="http://xzserver.applinzi.com";  var vm=new Vue({  el:"#app",  data:{  products:[]  },  methods:{  },  beforeCreate(){  console.log(`创建data:{}前自动触发`)  },  created(){  console.log(`创建data:{}后自动触发`)  },  beforeMount(){  console.log(`挂载页面元素和内容前自动触发`)  },  mounted(){  console.log(`挂载页面元素和内容后自动触发`)  //为了让.then()中的this和外部this保持一致，都指向当前new Vue()对象  axios.get("/index").then(result=>{  //想把请求回来的数据，保存到data中products变量上  this.products=result.data;  })  },  beforeUpdate(){  console.log(`修改data中的变量前自动触发`)  },  updated(){  console.log(`修改data中的变量后自动触发`)  },  beforeDestroy(){  console.log(`销毁当前组件前自动触发`)  },  destroyed(){  console.log(`销毁当前组件后自动触发`)  }  })  //在控制台中: vm.$destory()  </script>  </body>  </html>  运行结果: |

# 总结:

1. MVVM: 界面View+模型对象Model+视图模型ViewModel

2. Vue的绑定原理: 访问器属性+虚拟DOM树(+观察者模式)

3. 虚拟DOM树的优点:

(1). 小，仅包含可能变化的元素

(2). 遍历查找快

(3). 更新效率高，只更新受影响的元素

(4). 避免重复编码，已封装了DOM的增删改查操作

# 总结: 今后所有VUE功能三步:

1. 创建界面(增强)，界面中标记变量和事件处理函数

2. 创建new Vue()对象，用el:"选择器"，找到自己负责的页面区域的唯一父元素

3. 在new Vue()中添加data和methods，分别保存界面所需的所有变量和事件处理函数

# 七. 组件:

对应小程序视频: 小程序->在线->VUE->day03 6.1 创建单个组件

## 1. 什么是: 拥有专属的HTML+CSS+JS+数据的独立的可重用的页面功能区域

## 2. 为什么: 重用

## 3. 何时:

今后只要一个功能可能会被反复使用时，都用组件

## 4. 如何:

### (1). 创建组件:

组件

Vue.component("组件名",{

//模板/界面

template:`HTML片段`, //必须用唯一父元素包裹

data(){

return { //模型对象

数据/变量: 值,

... : ...

}

},

/\*\*\*\*\*\*以下内容就和new Vue()完全一样了\*\*\*\*\*\*\*/

methods:{ 函数 },

watch:{ 监视函数 },

computed:{ 计算属性 },

其它生命周期函数...

})

### (2). 在页面上反复使用组件:

VUE中一个组件，其实就是一个可重用的自定义标签而已！  
 <组件名></组件名>

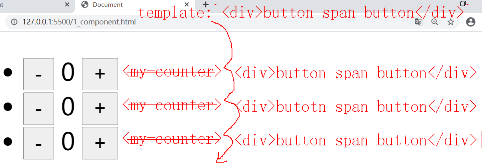
## 5. 组件的加载过程:

(1). Vue.component()创建一个组件对象，保存在内存中vue大家庭里备用

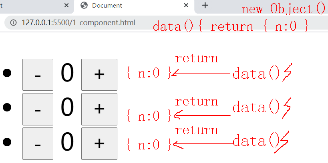
(2). new Vue()在扫描页面过程中，发现不是标准的HTML元素的标签，就会自动去内存中vue大家庭中找是否有同名的组件对象

(3). 一旦找到同名的组件对象，做3件事：

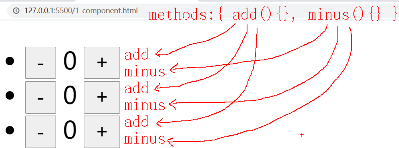
a. 用组件对象的template代替页面中<组件名>标签的位置



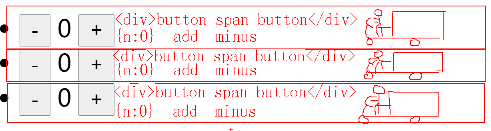
b. 自动调用组件对象的data()函数，在data函数内部先创建一个新的模型对象，包含新的模型变量。然后将新创建的模型对象返回给当前组件所在的位置

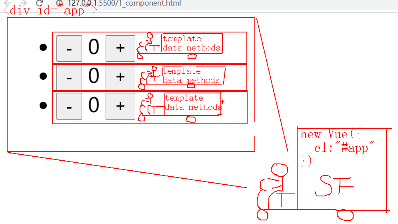


c. 将methods中的事件处理函数，分发给每个组件的副本



(4). 结果: 在页面中就会形成多个麻雀虽小五脏俱全的小组件副本。每个组件副本内部的原型原理和过程与new Vue()的原理是完全一样的！





## 6. 示例: 定义计数器组件，并反复使用

1\_component.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>    <script src="js/vue.js"></script>  </head>  <body>    <div id="app">      <ul>        <li>          <my-counter></my-counter>        </li>        <li>          <my-counter></my-counter>        </li>        <li>          <my-counter></my-counter>        </li>      </ul>    </div>    <script>      //因为界面中修改数量部分经常被重用，所以应该下封装为一个组件，再在界面中反复使用组件      //创建一个组件:      Vue.component("my-counter",{        //组件其实就是一个缩微的new Vue()        //麻雀虽小五脏俱全！        //1 做界面：和new Vue()做界面过程一样        //模板: 一次定义，反复生成一模一样的副本！        //1.1 也找可能发生变化的位置        //本例中: span的内容可能发生变化        //1.2 也找触发事件的元素        //本例中: 两个button可能触发事件        template:`<div>          <button @click="minus">-</button>          <span>{{n}}</span>          <button @click="add">+</button>        </div>`,        //2. 创建模型: 大致和new Vue()相同        data(){//变为一个函数          return {//相当于之前的data:{}            //因为界面中需要一个变量            n:0          }        },        methods:{ //和以前没有差别          //因为界面中需要2个函数          add(){            this.n++;          },          minus(){            if(this.n>0){              this.n--            }          }        }      })      //必须加new Vue()      new Vue({//扫描页面，识别vue的各种功能！        el:"#app"      })    </script>  </body>  </html>  运行结果： |

# 八. 组件化开发:

对应小程序视频: 小程序->在线->VUE->day03 6.2 Vue组件化开发 todo ...

## 1. 什么是: 拿到一个页面后，先划分组件。然后再把每个组件分给不同的人去开发！最后运行时，再将所有组件合并为一个页面运行！

## 2. 为什么: 2个原因:

(1). 极其便于多人协作开发！极大提高开发的效率

(2). 松耦合: 即使一个组件出错，不影响整个页面中其它组件的功能！

## 3. 何时: 今后所有的前端框架项目都是用组件化开发实现的。

## 4. 如何:

(1). 拿到功能/页面时，先划分组件:

a. 位置

b. 是否重用

(2). 为每个组件，在独立的js文件中创建组件对象

(3). 在父组件中，希望加载子组件的位置，用<子组件名>占位

(4). 在唯一完整的html页面中，先引入vue.js，再引入所有组件的独立js文件

(5). 创建new Vue()对象，在<div id="app">内，用<父组件>占位

## 5. 问题: Vue.component()创建的组件，称为全局组件。可放在任何位置！没有限制！但是，有些子组件，离开父组件单独使用，是没有意义的！所以不应该随意在父组件外部使用子组件！

## 6. 原理: 其实vue中共有三大类组件:

(1). new Vue() 根组件: 整个项目只有一个根组件，监控整个界面范围。

(2). Vue.component()全局组件: 可在界面中任何位置使用，没有限制

(3). 子组件: 被限制只能在一个指定的父组件内使用的组件。一旦超出指定的父组件范围就报错！

## 7. 解决: 今后只要限制在指定父组件内才能使用的组件都要创建为子组件，3步:

### (1). 不要用Vue.component()创建，而应该将子组件创建为一个普通的js对象。但是对象的内容，要符合vue组件的格式要求:

//子组件对象名，应该是将来组件标签名的驼峰命名形式

//比如: <todo-add>，则子组件对象名应该为: todoAdd

// <todo-list>，则子组件对象名应该为: todoList

var 对象名={

template:`HTML片段`,

data(){ return { ... } },

methods:{ ... },

... ...

}

### (2). 在父组件对象中添加一个新的成员:

父组件对象:{

template:xxx,

data(){ return { ... }},

... ...,

components:{ 子组件对象名, ... }

} todoAdd 或 todoList

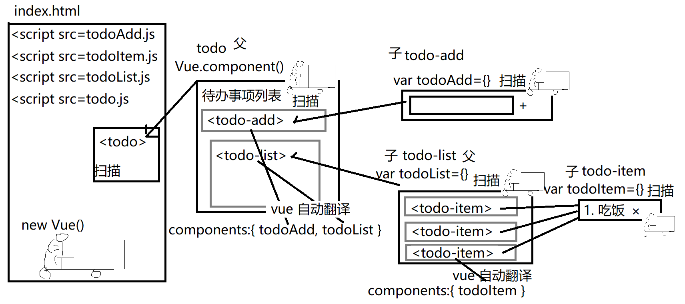
### (3). 在父组件的template中，使用组件标签引入子组件:

子组件的标签名，必须恢复成-分割

vue会自动将驼峰命名翻译为-分割！

比如： <todo-add> <todo-list>

## 8. 坑: 一旦使用子组件，则引入时，子组件必须在父组件之前先引入！



## 9. 示例: 使用组件化开发，实现todo案例的界面部分:

todo1/js/todoAdd.js

|  |
| --- |
| //创建todo-add组件  var todoAdd={    template:`<div>      <input><button>+</button>    </div>`  } |

todo1/js/todoItem.js

|  |
| --- |
| //创建todo-item组件  var todoItem={    template:`<div>      1. 吃饭 <a href="javascript:;">×</a>    </div>`  } |

todo1/js/todoList.js

|  |
| --- |
| //创建todo-list组件  var todoList={    template:`<ul>      <li><todo-item></todo-item></li>      <li><todo-item></todo-item></li>      <li><todo-item></todo-item></li>    </ul>`,    //todoItem是todoList的子组件    components:{ todoItem }  } |

todo1/js/todo.js

|  |
| --- |
| //创建todo组件  Vue.component("todo",{    template:`<div>      <h3>待办事项列表</h3>      <todo-add></todo-add>      <todo-list></todo-list>    </div>`,    //todoAdd和todoList是todo的子组件    components:{ todoAdd, todoList }  }) |

todo1/index.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>    <script src="js/vue.js"></script>    <script src="js/todoAdd.js">      //是todo的子    </script>    <script src="js/todoItem.js">      //是todoList的子    </script>    <script src="js/todoList.js">      //是todo的子    </script>    <script src="js/todo.js"></script>  </head>  <body>    <div id="app">      <todo></todo>    </div>    <script>      new Vue({        el:"#app"      });    </script>  </body>  </html>  运行结果: |

## 组件传参:

## （1）父传子

(1). 问题: Vue中父组件的成员，子组件也无权使用！

(2). 原理: Vue中每个组件的数据是专属的！

(3). 解决: 其实组件之间可以传递数据的！

(4). 父给子: 2步:

a. 父组件在子组件身上放一个数据:

父组件的template中:

<子组件 :自定义属性名="父组件变量"></子组件>

b. 子组件从父组件放变量的自定义属性中取出父组件传来的值

子组件对象中:

var 子组件={

template:xxx,

data(){ return { ... } }, //自己定义的

//属性，从自定义属性中获取变量值

props: [ "自定义属性名" , ... ] //父组件给的

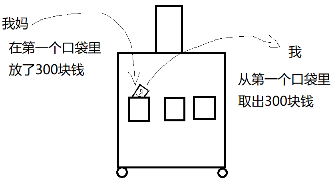
//结果: props中的自定义属性名，等效于当前子组件中的data中的变量。用法和data中的变量完全一样！

this.自定义属性名

或

{{自定义属性名}}

}



## 11. 示例: 实现todo案例中父组件给子组件传值

todo2/js/todoAdd.js

|  |
| --- |
| //创建todo-add组件  var todoAdd={    template:`<div>      <input><button>+</button>    </div>`  } |

todo2/js/todoItem.js

|  |
| --- |
| //创建todo-item组件  var todoItem={    template:`<div>      {{i+1}}. {{t}} <a href="javascript:;">×</a>    </div>`,    props:["t","i"]  } |

todo2/js/todoList.js

|  |
| --- |
| //创建todo-list组件  var todoList={    template:`<ul>      <li v-for="(t,i) of tasks" :key="i">        <todo-item :t="t" :i="i"></todo-item>      </li>    </ul>`,    props:["tasks"],    //todoItem是todoList的子组件    components:{ todoItem }  } |

todo2/js/todo.js

|  |
| --- |
| //创建todo组件  Vue.component("todo",{    template:`<div>      <h3>待办事项列表</h3>      <todo-add></todo-add>      <todo-list :tasks="tasks"></todo-list>    </div>`,    data(){      return {        tasks:[ "吃饭", "睡觉", "打亮亮"]      }    },    //todoAdd和todoList是todo的子组件    components:{ todoAdd, todoList }  }) |

todo2/index.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>    <script src="js/vue.js"></script>    <script src="js/todoAdd.js">      //是todo的子    </script>    <script src="js/todoItem.js">      //是todoList的子    </script>    <script src="js/todoList.js">      //是todo的子    </script>    <script src="js/todo.js"></script>  </head>  <body>    <div id="app">      <todo></todo>    </div>    <script>      new Vue({        el:"#app"      });    </script>  </body>  </html>  运行结果: |

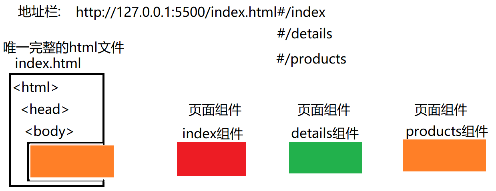
# 九. SPA: Single Page Application

单 页面 应用

对应小程序视频: 小程序->在线->VUE->day04 1. SPA单页面应用

## 什么是单页面应用:

整个应用程序只有一个唯一完整的HTML页面。其它所谓的"页面"，其实都是组件而已。所谓的"页面跳转"，其实只是在一个HTML中切换不同的组件显示而已。



## 2. 为什么:

|  |  |  |
| --- | --- | --- |
|  | 多页面 | 单页面 |
| 请求次数 | 每切换一次页面，都要向服务器发送请求，请求次数多 | 首次请求时，就把唯一完整的HTML页面和其他所有组件都下载到客户端本地。今后，无论怎样切换页面，都是在客户端切换不同的组件显示而已，不会像服务器端发送任何请求！请求次数少 |
| 公共资源的使用 | 每切换一次页面，都要重新请求公共的资源(bootstrap.css, jquery.js等)。请求次数多 | 单页面应用只在首次请求页面时，下载一次公共资源。之后切换页面时，除了组件部分之外，其余部分是保持把不变的！所以不会反复请求公共的资源！请求次数少 |
| 页面加载效率 | 每次切换页面都要废弃旧的DOM树，重建整棵DOM树，所以效率低。 | 只在加载首页时，建立完整DOM树。之后，切换页面时，因为唯一完整的HTML页面是不变的，只更新局部的组件内容。所以，DOM树也不会重建，而是更新部分节点。效率高！ |
| 页面切换动画 | 因为每次切换页面都是先清空前一个页面，再重新请求请求后一个页面。前后两个页面不可能并存在客户端。所以，几乎不可能实现页面切换动画。 | 因为所有的组件都在客户端了，所以完全有可能同时显示前后两个组件的部分内容，形成动画效果。 |

## 3. 何时: 今后，只要使用框架开发项目，用的都是单页面应用！

## 4. 缺点: 首屏加载极慢。

(1). 原因: 首屏就要把所有页面组件内容都下载下来！下载内容比较多！

(2). 解决: 懒加载！（明天讲）

## 5. 如何: 3个主要的步骤:

### (1). 创建一个唯一完整的HTML文件

a. 标准的支持vue的页面结构

b. 必须引入vue-router.js核心组件

c. 在唯一完整的页面内部添加<router-view>标签，为将来的页面组件占位。

### (2). 创建多个页面组件

a. 为每个页面组件创建独立的js文件

b. 每个页面组件都要创建为子组件的格式(~~不要用Vue.component()~~)

### (3). 在独立的router.js文件中创建路由器对象和路由字典列表：

a. 什么是路由字典: 专门保存地址栏中相对路径与组件对象之间对应关系的一个数组。

var routes=[

{ path:"/", component:Index },

{ path:"/details", component: Details}

... ...

]

b. 什么是路由器对象: 专门负责监控地址栏变化，并根据地址栏变化查找对应组件，替换页面中router-view的 核心对象

var router=new VueRouter({ routes })

c. 后续操作:

(1). 将所页面组件文件和router.js文件都引入到唯一完整的HTML页面中。

强调: 所有页面组件必须先于router.js引入

(2). 将router对象加入到new Vue()中！

new Vue({

el:"#app",

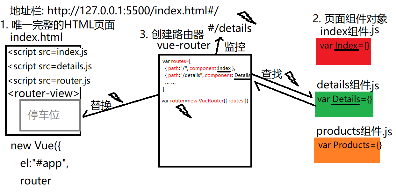
router

})

## 6. 结果: 只要地址栏相对路径一变，router对象会自动获得新的相对地址。自动去routes中查找对应的组件对象。最后用找到的组件对象代替HTML文件中<router-view>的位置

## 7. 强调: Vue的路由器采用锚点地址，作为客户端导航的相对地址:

http://127.0.0.1:5500/index.html#/相对路径



## 8. 示例: 实现包含两个页面的基本的SPA应用

SPA/js/index.js

|  |
| --- |
| var Index={    template:`<div>      <h3 style="color:red">这里是首页</h3>    </div>`  } |

SPA/details.js

|  |
| --- |
| var Details={    template:`<div>      <h3 style="color:green">这里是详情页</h3>    </div>`  } |

SPA/js/router.js

|  |
| --- |
| //路由字典: 包含多对儿相对路径与组件对象间对应关系  var routes=[    {path:"/", component:Index},    {path:"/details", component:Details}    //强调: component后的组件对象名必须和组件.js文件中的组件对象名一致！  ];  //路由器对象  var router=new VueRouter({    routes  }) |

SPA/index.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>    <script src="js/vue.js"></script>    <script src="js/vue-router.js"></script>    <script src="js/index.js">      //var Index={}    </script>    <script src="js/details.js">      //var Details={}    </script>    <script src="js/router.js">      //routes=[        //{ ..., component:Index},        //{ ..., component:Details}      //]    </script>  </head>  <body>    <div id="app">      <router-view></router-view>    </div>    <script>      new Vue({ //扫描        el:"#app",        router      })    </script>  </body>  </html>  运行结果: |

## 9. 404页面：

(1). 除了所有正确的路径之外，其余所有任意不正确的路径，都要跳转到404。

(2). 如何:

a. 在独立的js文件中创建一个NotFound组件

b. 在router.js中添加路由字典项: (列表的最后)

{ path:"\*", component:NotFound }

c. 在唯一完整的html页面中提前引入notFound.js文件。

## 10. 页头:

(1). 因为页头可以用在任何一个页面中，不受限制，所以应该是一个全局组件。

(2). 如何: 4步:

a. 在独立的js文件中，先创建一个普通的子组件，包含页头的部分内容

b. 在唯一完整的HTML页面中引入页头.js文件，同时就引入了页头组件对象——暂时还不是全局组件！

强调: 页头组件不用加入router.js中。因为页头组件绝对不会单独展示！总是和别的页面配合一起展示。所以页头没有专门的路径与之对应！

c. 在new Vue()之前，外部，使用Vue.component()将页头子组件，转化为全局组件。

d. 使用页头组件: 2种:

1). 所有页面都需要页头，无一例外:

应该将页头组件写在<router-view>外部的上方

2). 有的页面需要页头，有的页面不需要页头

i. 不能放在<router-view>外部的上方

ii. 哪个页面需要页头，就在哪个页面组件的内部添加页头组件。那些不需要页头的组件，就不写。

## 11. 路由跳转: 2种:

(1). html中: 不要用<a>

<router-link to="/相对地址">文本</router-link>

(2). js中: this.$router.push("/相对路径")

## 12. 路由传参: 3步:

(1). 修改router.js中路由字典中的路由地址，允许传参

{path:"/details/:变量名", component:Details, props:true},

(2). 跳转时，路径: /details/变量值

(3). 下个页面中如何获得地址栏中的参数值:

a. router.js中的props:true，意为让地址栏中的参数值自动变成当前页面组件的props中的一个属性值

b. 下一个页面中就可以用: props:[ "变量名" ]

(4). 坑: 一旦一个路径配置/:变量名，则必须携带参数才能进入该路径。如果不带参数，将被禁止进入！

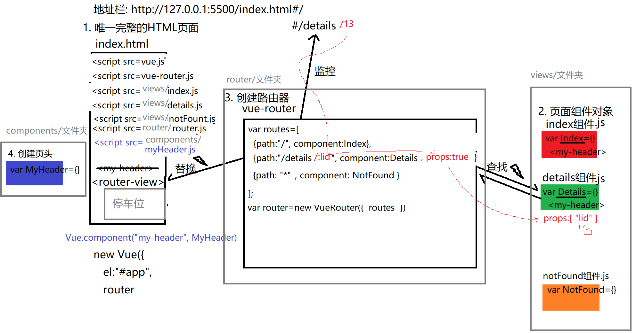
## 13. 问题: 那么多js文件都放在同一个js文件夹下，非常乱

解决: 建立专门的文件夹，分别保存不同的组件:

views/文件夹，专门保存所有页面组件

components/文件夹，专门保存所有全局组件

router/文件夹，专门保存router.js文件。



|  |
| --- |
| route, routes, router  1. router: 用new VueRouter()创建出的路由器对象  a. 监视地址栏变化  b. 还可以执行跳转动作！  2. routes: 一个路由字典数组，包含当前网站中所有路径与组件的对应关系列表。routes会被装入new VueRouter()中，和router对象一起发挥作用！  3. route: 一个路由地址，代表当前地址栏中的url信息，像BOM中的location。  地址栏：http://127.0.0.1:5500/index.html#/details |

# 总结:

3. 只要希望在页面加载时自动对元素执行一些初始化操作时就用自定义指令:

(1). 添加自定义指令:

Vue.directive("自定义指令名",{

inserted(domElem){  
 对domElem执行DOM操作

}

})

(2). 使用自定义指令:

<元素 v-自定义指令名>

4. 今后只要根据其他变量的值动态计算出一个属性值就用计算属性:

<元素>{{计算属性}}</元素>

new Vue({

el:"#app",

data:{...},

methods:{...},

computed:{  
 计算属性名(){

计算过程

return 计算结果

}  
 }

})

5. 希望将变量的原始值先加工后再显示给用户看时就用过滤器:

Vue.filter("过滤器名",function(oldVal, 自定义形参,...){

return 加工后的新值

})

<元素>{{ 变量 | 过滤器(实参值, ...) | ... }}</元素>

6. 只要在vue中发送ajax请求，就用axios

axios.defaults.baseURL="服务器端接口的公共基础地址部分"

axios.get(  
 "服务器端接口地址的相对路径",

{

params:{ 参数名: 参数值, ... }

}  
).then(result=>{

... result.data...

})

或

axios.post(  
 "服务器端接口地址的相对路径",

"参数名1=参数值1&参数名2=参数值2&..."  
).then(result=>{

... result.data...

})

强调: 在vue内使用axios，then中必须用箭头函数，保持then内this与外部this一致，都指向当前new Vue()对象

7. vue生命周期4个阶段 8个钩子函数

beforeCreate(){ ... }

(1). 创建(create)

created(){ ... }

beforeMount(){ ... }

(2). 挂载(mount)

mounted(){ ... 经常在这里发送ajax请求 ... }

beforeUpdate(){ ... }

(3). 更新(update)

updated(){ ... }

beforeDestroy(){ ... }

(4). 销毁(destroy)

destroyed(){ ... }

8. 只要希望重用一块独立的功能区域就用组件:

(1). 定义组件

Vue.component(`组件标签名`,{

template:`HTML内容片段`,

data(){ return { 变量 } },

//其余和new Vue()完全相同

})

(2). 在HTML中使用自定义组件

<组件标签名/>或双标记也行

(3). 原理: new Vue()扫描到自定义组件标签，

a.组件的template中的HTML内容代替页面中<组件标签>位置。

b. 并为这个小区域专门创建一个缩微版的vue类型对象。

1). 调用组件的data()函数为当前组件副本创建一个专属数据对象副本。

2). 引入组件对象中的methods等其他内容到当前组件对象副本中

9. 组件化开发:  
(1). 步骤:  
a. 拿到页面先划分功能区域

1). 从上到下，按功能不同划分区域

2). 按是否重用划分

b. 为每个组件创建独立的.js文件，其中包含一个组件对象及其内容

c. 将所有组件引入唯一完整的html页面中，并在<div id=”app”></div>中添加父组件标签。  
(2). 运行时:

a. new Vue()扫描<div id=”app”>，发现父组件标签，创建并替换父组件

b. 父组件扫描自己内部的template内容，创建并替换子组件

(3). 三种组件:

a. 根组件: new Vue()

b. 全局组件: Vue.component(...)

c. 子组件: 3步

1). var 子组件对象名={

内容必须符合组件的要求

}

子组件对象名必须是驼峰命名

2). 父组件对象中:{

... ...

components: { 子组件对象名, ... ,... }

}

子组件对象名必须是驼峰命名

3). 父组件template中用<子组件标签名/>引入子组件内容

components会将子组件对象名的驼峰命名自动翻译为-分隔

所以, 使用子组件标签时，要用-分隔多个单词

(4). 组件间传参: 父给子

a. 父组件给:

<子组件 :自定义属性名="父组件变量">

b. 子组件取:

props:["自定义属性名"]

结果: 在子组件内，props中的"自定义属性名"与子组件自己data中的变量用法完全相同！

10. SPA

(1). 3步:

a. 先创建唯一完整的HTML页面

1). 包含vue基本的页面结构

<div id="app"> new Vue({el:"#app"})

2). 引入所有必要的文件和组件

vue-router.js, 其它页面或组件文件, router.js路由器对象所在文件

3). <div id="app">中用<router-view/>为今后页面组件预留空位

b. 再为每个页面组件创建独立的文件。每个页面组件其实都是一个子组件

c. 创建router.js文件，创建路由器对象

1). 创建路由字典对象:

var routes=[

{path:"/", component:首页组件对象名},

{path:"/相对路径" , component: 其它页面组件对象名},

{path:"\*", component: 404页面组件对象 }

]

2). 创建路由器对象，并将路由字典对象转入路由器对象中  
 var router=new VueRouter({ routes })

3). 将router对象加入到new Vue()中

回到唯一完整的HTML页面中: new Vue({ el:"#app", router })

(2). 页头等全局组件：

a. 创建独立的文件保存页头组件的内容

b. 使用Vue.component("my-header",{ ... })将页头创建为全局组件

c. 在唯一完整的HTML页面中引入页头组件文件

d. 使用页头组件标签<my-header/>: 2种:

1). 如果所有页面都有统一的页头:

就放在唯一完整的html页面中<router-view>外部上方

2). 如果有的页面有页头，有的页面没有页头:

就只放在需要页头的组件中的template中

(3). 路由跳转: 2种:

a. html中: <router-link to="/相对路径">文本<router-link>

b. js中: this.$router.push("/相对路径")

(4). 路由传参:

a. 修改路由字典:

{path:"/相对路径/:自定义参数名", component:页面组件对象, props:true}

b. 跳转时:

<router-link to="/相对路径/参数值"

或

this.$router.push("/相对路径/参数值")

c. 下一个页面接:

1). props:[ '自定义参数名' ]

2). 可将"自定义参数名"用作绑定或程序中都行

11. 脚手架文件夹结构:

(1). 唯一完整的HTML页面: 一分为三:

a. public文件夹

1). 图片img文件夹放在public文件夹下

2). 第三方css的压缩版和第三方js的压缩版都放在public文件夹下

3). 唯一完整的HTML文件index.html中，head中引入第三方的css和js

b. src/App.vue

1). <template>下只包含公共的页头组件和<router-view>

2). <style>下包含所有网页都要用到的公共css样式，比如css重置代码

c. src/main.js

1). import引入App.vue，router，axios，以及其他全局组件

2). 将全局组件对象转为真正的全局组件: Vue.component( "组件标签名", 全局组件对象 )

3). 配置axios并放入原型对象中:

axios.defaults.baseURL="服务器端基础路径"

Vue.prototype.axios=axios;

(2). 为每个页面创建.vue组件文件，都放在src/views文件夹下。每个.vue文件中:

a. <template>标签内，包含这个页面的HTML内容

b. <script>export default{ ... }</script>中包含组件对象的js内容。

c. <style>标签内包含仅这个页面组件内才使用的css

d. <template>中的HTML内容以及<script>export default{...}</script>中的js内容，和前四天将的是完全一样的写法，绑定，指令，函数，生命周期，axios请求等都一样。前四天怎么用，这里就怎么用。

(3). 路由字典和路由器对象，在src/router/index.js文件中

a. 仅import首页组件对象，不要过早引入其它页面组件

b. 路由字典中首页组件: { path:"/", component:首页组件对象}

c. 其余页面组件都做成懒加载:

{

path: '/相对路径',

component: () => import(/\* webpackChunkName: "组件名" \*/ '../views/其它页面组件.vue')

}

(4). 全局组件都放在src/components文件夹下，每个全局组件.vue文件中。但是，全局组件必须在main.js引入，并用Vue.component()转化为真正的全局组件，才能在其它组件的HTML中使用。

(5). 运行时: 路由器router对象监视浏览器地址栏路径变化，并查找对应的页面组件内容，先代替App.vue中的<router-view>，然后main.js再将包含要加载的页面内容的App.vue组件内容，替换到唯一完整的index.html中空<div id="app">位置。

小程序: WEB问题速查

1. 问题清单: 小程序->首页->VUE->day04,05

2. 高清无码小视频: 小程序->在线->VUE->day04,05

今日扩展视频: 小程序->在线->VUE->day04、05中所有视频

不用今天都看完！！！！！！\*\*\*两个月后\*\*\*鄙视面试前\*\*\*看完就行\*\*\*！

3. 单词列表: 小程序->单词->VUE->day04,05

第三阶段上课约定:

1. 上课千万不要跟着记笔记和写代码！

2. 拼写错误！自己找！让我帮你找，你需要手抄20遍，给我拍照！

3. 现在正处在照猫画虎的阶段！每个VUE效果至少写3遍:

3.1 注释+代码抄一遍！

3.2 保留注释，删除代码，把代码再填回来！

3.3 删除注释和代码，试着用自己的话把注释写回来，根据自己的注释，把代码填回来

一个VUE效果，只有自己能写出人话注释，才算自己会！——标准！

如何提问:

1. 上课时，案例中，笔记中那句话哪行代码看不懂，立刻微信私信问我！

2. 小程序中问题清单里哪个问题和回答看不懂！立刻微信私信问我！

3. 个人项目中哪个功能不会做！立刻微信私信问我！

4. 刷笔试题时，遇到不会的笔试题！立刻微信私信问我！

下载源代码: 右键->链接另存为！

正课:

1. 脚手架:

2. 脚手架文件夹结构

3. ES6的模块化开发

4. 学子商城项目的制作过程:

5. 懒加载:

6. http-proxy:

# 十. 脚手架:

对应小程序视频: 小程序->在线->VUE->day04 2. 使用vue/cli工具创建学子商城项目

## 1. 问题: 现代的前端项目结构非常复杂！如果任由什么人随意定义项目的文件夹结构，后果，项目与项目之前结构差异极大！即不便于维护，也不便于开发人员快速适应！

## 2. 解决: VUE官方出了一套标准化的vue项目结构！所有使用vue框架开发的项目都要遵守这个标准化项目结构！——脚手架

## 3. 好处: 几乎所有用vue做到项目，文件夹结构几乎是完全一样的！开发人员可以非常快速的适应新项目！

## 4. 何时： 今后无论学习任何框架，先找脚手架。

## 5. 如何: 2步:

### (1).下载并安装一个可以反复生成脚手架代码的命令行工具: (老母鸡)

a. 说明: 只要在电脑上安装一次，就可反复为多个不同的项目分别创建脚手架项目结构

b. 如何: 2步:

##### 1). 先配置npm的国内淘宝镜像，从国内服务器下载和安装：快

2种选一种:

a. npm config set registry http://registry.npm.taobao.org

npm 配置 设置 仓库

执行完之后，确认是否设置成功: npm config get registry

npm 配置 获得 仓库

看到返回淘宝镜像，说明成功！

b. 备选方案: 如果上一步出错！可选择这一步:

npm i -g cnpm --registry=http://registry.npm.taobao.org

确认是否安装成功: cnpm -v

看到版本号说明成功！

##### 2). 下载并安装命令行工具： vue/cli， 2种选一种

a. 用npm安装: npm i -g @vue/cli

npm 安装 全局 command line interface

install global 命令 行 接口

//如果说FEXIST错误，可进入出错提示中的路径，默认为:

C:\Users\登录操作系统的用户名\AppData\Roaming\npm\node\_modules

删除@vue文件夹

b. 备选方案: 如果npm出错，可换成: cnpm i -g @vue/cli

看到: + @vue/cli@4.5.8，说明安装成功

### (2). 用脚手架为这个项目创建一套标准化的vue项目结构:

### a. 先找到要创建项目的文件夹：在操作系统中找到想要创建项目的文件夹目录，然后在地址栏中输入cmd，按回车

### b. 输入vue create 自定义项目名

#### 1). Vue CLI v4.5.8

? Please pick a preset: (Use arrow keys)

Default ([Vue 2] babel, eslint)

Default (Vue 3 Preview) ([Vue 3] babel, eslint)

> Manually select features

#### 2). ? Check the features needed for your project: (Press <space> to select, <a> to toggle all, <i> to invert selection)

() Choose Vue version

> (\*) Babel //翻译: 将时髦的ES6, ES7的语法，翻译为大多数浏览器都认识的ES5语法

( ) TypeScript

( ) Progressive Web App (PWA) Support

(\*) Router

( ) Vuex

( ) CSS Pre-processors

() Linter / Formatter

( ) Unit Testing

( ) E2E Testing

#### 3). ? Use history mode for router? (Requires proper server se

tup for index fallback in production) (Y/n) n

Vue的路由有两种模式:

hash模式：默认: #/相对路径 —— 不需要服务器端支持就可使用。

history模式：/相对路径 —— 需要服务器端配置首页重定向机制！

#### 4). ? Where do you prefer placing config for Babel, ESLint, etc.? (Use arrow keys)

In dedicated config files

> In package.json

#### 5). ? Save this as a preset for future projects? (y/N) N

c. 成功: 🎉 Successfully created project xzvue.

### (3). 运行脚手架项目:

#### a. 删除项目中的.git文件夹

#### b. vscode打开刚生成的脚手架项目文件夹

#### c. 在vscode中，右键点击package.json文件，选择在集成终端中打开

#### d. 在弹出的终端窗口中: 输入npm run serve

遇到问题: 去看小程序->首页->VUE->day04 第一个问题: npm或vue命令行...  
 遇到: 无法将“npm”项识别为 cmdlet...

可以右键点击我的电脑->属性->高级系统设置->环境变量->下方系统变量列表中->双击path->新建->C:\Users\ 用户名\AppData\Roaming\npm

其中用户名换成你当前操作系统登录时的用户名。

#### e. 成功: DONE Compiled successfully in 2949ms

#### f. npm run serve，启动了一个临时的服务器，保存vue编译后的所有网页。地址： App running at:

- Local: http://localhost:8080/

所以，在vscode中，按住ctrl点这个链接地址，打开浏览器

#### g. ftp上:

xzvue 30M 完整的xzvue脚手架代码

xzvue 88K 去掉node\_modules文件的简化版

解压，进入xzvue文件夹，在地址栏输入cmd, 再命令行输入npm i  
 就可以把去掉的node\_modules重新下载下来！

# 十一. 脚手架文件夹结构:

## 1. 唯一完整的HTML页面(被拆成三分，运行时再组合起来): public/index.html

(1). 问题: index.html中即没有<router-view>又没有new Vue()

(2). 原因:

a. vue的作者认为, 在index.html中不应该包含任何js代码！

b. 作者认为，vue中一切都是组件，连<div id="app">也是一个组件。所以，一定要将<div id="app">也放在一个独立的组件文件里。

(3). 解决:

a. 原来index.html中的new Vue()等js代码，都集中保存在src/main.js文件中

b. 原来index.html中的<div id="app">，都集中保存在src/App.vue文件中

c. 在用npm run serve编译整个项目时:

1). 用App.vue中的<div id="app">代替index.html中临时占位的空的<div id="app">

2). 将main.js中的new Vue()等js代码都重新引入回index.html页面中。

## 2. 页面组件(项目中共有几个页面，就要包含几个页面组件)

(1). src/views文件夹下

(2). 问题:

a. 在js文件中写组件的template时，没提示，很痛苦！

b. css写哪儿？

(3). 解决: Vue中独创了一种新的文件格式: .vue文件

a. Vue中每个组件，都要创建为一个.vue文件

b. 每个.vue文件包含三部分内容:

1). 专门编写HTML代码片段的区域： <template>

如果希望有提示，需要安装vscode的插件vetur

2). 专门编写组件对象的区域: <script>

export default {

//页面组件的内容

}

3). 专门编写CSS样式的区域: <style>

## 3. 路由器对象: src/router/index.js，也包含路由字典routes和路由器对象router。

## 4. 全局组件或子组件: src/components

# 十二. ES6的模块化开发:

## 1. 问题: 明明是router.js中需要用到index组件和details组件，但是我们却必须将index组件和details组件都引到index.html网页中！——乱

## 2. 解决: 脚手架中采用了ES6的模块化开发！可以让模块与模块之间直接引用！——直观，简洁

## 3. 如何: 2步

### (1). 将所有组件都变成模块对象：

a. 默认: 一个不带js的.vue文件，默认就是一个模块对象。

b. 如果一个.vue文件中带有js组件对象，则必须用export default {}抛出后，才能让别人引入！

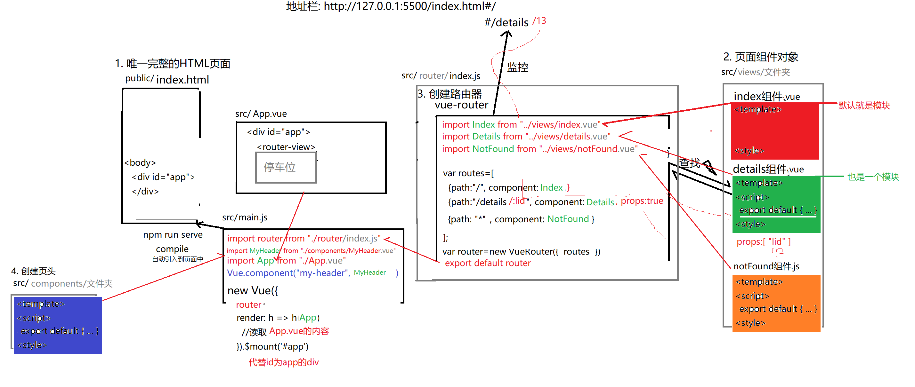
### (2). 如果一个组件/模块内，需要用到另一个模块的内容，就可用import引入其它模块，为自己所用

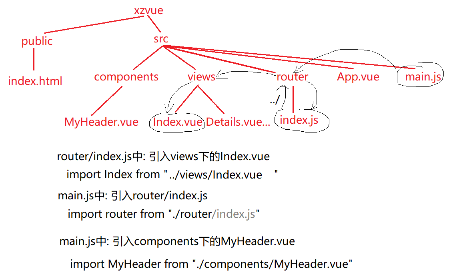
import 自定义变量名 from "相对路径"

从指定路径的文件中

引入一个模块对象

保存在自定义的变量名中





# 十三. 避免组件间样式冲突

## 1. 问题: 在不同的页面组件中使用相同的选择器规定了不同的css样式，结果，只有其中一个套css样式可以保留！发生了冲突！

## 2. 原理: 所有组件的css代码，最终会被打包压缩为一个css文件。将来所有组件的css，其实是放在一个文件中的。如果一个文件中出现完全相同的两个选择器，则只要后一个才能生效！

## 3. 解决: 2种:

### (1). 自动化(不是万能, 效率低):

a. <style scoped>

(当前组件)范围内的

b. 意为: 将以下css样式限制在仅当前组件内的元素可用！其它组件的元素不可用

c. 原理:

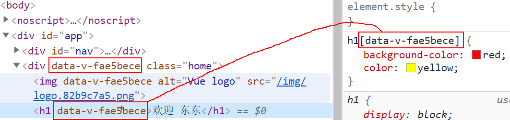
1). scoped会自动给当前组件的每个元素都添加一个相同的随机的自定义属性名。<元素 data-v-随机名>

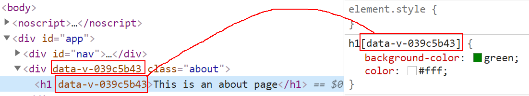
2). 自动将当前组件内的所有选择器结尾加一个附加条件:

其它选择器[data-v-随机名]

结果: 只有带有data-v-随机名的元素才能使用这个选择器

3). scoped可以保证每个组件的随机名是不一样的！就实现了一个组件，不能使用其它组件中的样式！





### (2). 手动解决(万能):

a. 如何: 2步:

1). 每创建一个组件，都在组件的唯一父元素上，定义一个专门的class名（尽量和组件名相同）

2). 在<style>中凡是这个组件中的选择器，必须以".这个组件名"开头

b. 示例:

Home.vue

|  |
| --- |
| <template>    <div class="home">      <img alt="Vue logo" src="../assets/logo.png">      <h1>欢迎 {{uname}}</h1>    </div>  </template>  <script>  export default {    data(){      return {        uname:"东东"      }    }  }  </script>  <style>    .home>h1{      background-color:red;      color:yellow;    }  </style>  运行结果: |

About.vue

|  |
| --- |
| <template>    <div class="about">      <h1>This is an about page</h1>    </div>  </template>  <style>    .about>h1{      background-color:green;      color:#fff;    }  </style>  运行结果: |

# 十四. 学子商城项目的实现步骤:

## 1. 安装和配置axios:

(1). 脚手架默认不带axios

(2). 安装: npm i -save axios

安装在当前项目内部

(3). 配置:

a. 在main.js中, new Vue()之前就要配置

b. 引入axios模块

import axios from "axios" //node\_modules中

c. 配置axios的基础路径:

axios.defaults.baseURL="http://服务器端接口的基础地址"

d. 问题: 几乎所有组件中都要用到axios

前提: 无论是new Vue()，还是Vue.component(), 还是export default {}，其实本质都是Vue类型的一种子对象

解决: 所以axios应该放在Vue类型的原型对象中！

Vue.prototype.axios=axios //复习JS高级第4天继承

(4). 使用: 在任何一个组件的任何一个位置，都可用:

this.axios.get/post(...).then(result=>{ ... result.data ... })

## 2. 复制旧项目public文件夹中img文件夹

粘贴到xzvue项目中public文件夹下！

复制旧项目public文件夹中css文件夹和js文件夹

粘贴到xzvue项目中public文件夹下！

删除xzvue/public/css文件夹中除bootstrap.min.css以外的其余所有css。

## 3. 在xzvue/public/index.html中引入bootstap.min.css和jquery.js以及bootstrap.min.js

<link rel="stylesheet" href="css/bootstrap.min.css">

<script src="js/jquery-3.4.1.min.js"></script>

<script src="js/bootstrap.min.js"></script>

## 4. 删除: src/views/Home.vue、About.vue，src/components/HelloWorld.vue

新建: src/views/Index.vue、Details.vue，src/components/MyHeader.vue

## 5. 调整路由: src/router/index.js中:

删除: import Home from '../views/Home.vue'

添加: import Index from "../views/Index.vue"

import Details from "../views/Details.vue"

注释掉: const routes=[]内的所有内容

在const routes=[ //内添加两条新的路由地址:

{path:"/", component:Index},

{path:"/details", component:Details}

]

## 6. 迁移首页：

(1). 复制: 旧项目public文件夹中的index.html文件中的<main>...</main>及其所以子内容

粘贴到: xzvue项目中src/views/Index.vue中<template>内部</template>

(2). 复制: 旧项目public文件夹中的css/index.css文件全部内容

粘贴到: xzvue项目中src/views/Index.vue中<style scoped>内部</style>

## 7. 修改App.vue:

(1). 将所有页面共用的css，应该复制到App.vue中的<style>里

复制旧项目public文件夹/css/base.css中所有内容

粘贴到xzvue/src/App.vue文件中<style>内

(2). 删除App.vue中<div id="app">内除<router-view/>外其余所有内容

## 8. 迁移页头:

(1). 复制旧项目public中header.html中的内容

粘贴到新项目xzvue中src/components/MyHeader.vue中的:

<template>

<header id="header" class="container">

里

</header>

</template>

(2). 复制旧项目public中css/header.css中的内容

粘贴到新项目xzvue中src/components/MyHeader.vue中<style scoped>中

(3). 在新项目xzvue中src/main.js中new Vue()之前：

a. 引入MyHeader.vue

import MyHeader from "./components/MyHeader.vue"

b. 用Vue.component()将MyHeader变成全局组件

Vue.component("my-header", MyHeader);

(4). 在App.vue中<router-view>上方加: <my-header>

## 9. 迁移详情页:

(1). 复制: 旧项目public文件夹中的product\_details.html文件中的<main>...</main>及其所以子内容

粘贴到: xzvue项目中src/views/Details.vue中<template>内部</template>

(2). 复制: 旧项目public文件夹中的css/product\_details.css文件全部内容

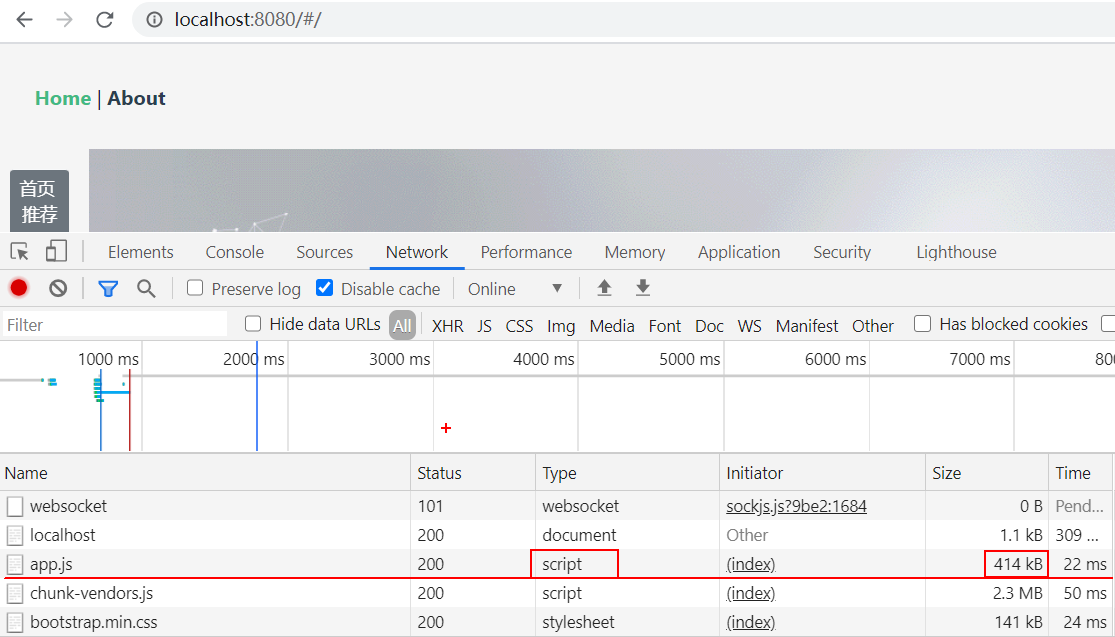
粘贴到: xzvue项目中src/views/Details.vue中<style scoped>内部</style>

# 十五. 懒加载:

## 问题: 单页面应用的致命问题: 首屏加载极慢。

**因为单页面应用默认必须在首次请求时，把所有页面组件都下载到客户端本地！**

## 原理: npm run serve时，vue脚手架会将所有.vue文件的代码都打包压缩到一个js文件中，再引入唯一完整的index.html文件中。所以，这个包含所有代码的js文件会很大！



## 解决: 懒加载，2种方案

**需要什么，才加载什么；不需要什么，就不加载！**

### (1). 异步延迟加载(脚手架默认):

#### a. 暂时不要把除首页之外的其它代码import进程序中！

#### b. 在路由字典中定义，只有当用户访问到某个组件时，才动态引入这个组件。

src/router/index.js

|  |
| --- |
| import Vue from 'vue'  import VueRouter from 'vue-router'  import Index from "../views/Index"  Vue.use(VueRouter)  const routes = [    {      path: '/',      component: Index    },    {      path: '/details/:lid',      //暂时不要把Details.vue引入项目      //只有当用户访问到details时，才将Details组件引入项目中      //而且webpackChunkName:是在指示，将Details.vue文件单独打包压缩为一个文件。文件名为details      component: () => import(/\* webpackChunkName: "details" \*/ '../views/Details.vue'),      props:true    }  ]  const router = new VueRouter({    base: process.env.BASE\_URL,    routes  })  export default router |

//该案例从网上copy下来

//第一步注释import导入的文件

//import About from '../components/About.vue';

//第二步将引入组件的方式以箭头函数的方式异步引入

const routes = [

{

path: '/about'，

component: () => import( /\* webpackChunkName: 'about' \*/ '../components/About.vue' )

}

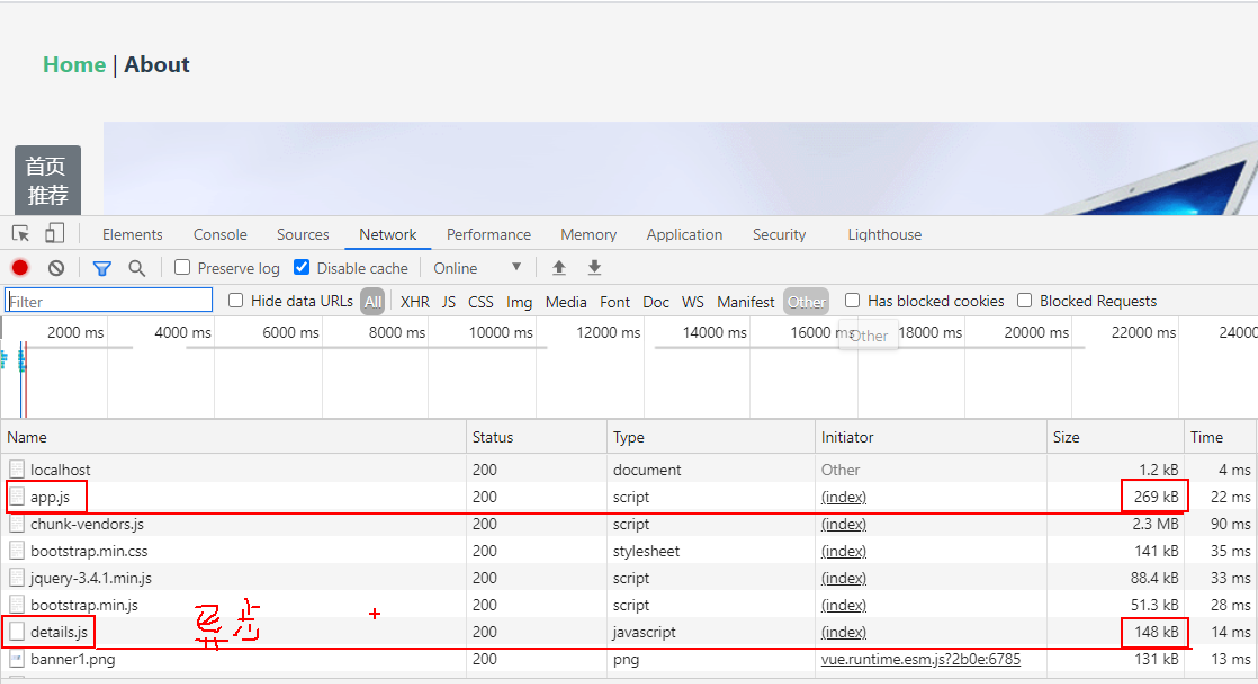
]

注意：import中 **/\* webpackChunkName: 'about' \*/** 这个注释必须有，这个为将来打包命名

结果：最后打包工具打包不会将 About.vue组件中的内容打包进app.js中，

它会单独打包成一个新的about.js文件，并在页面加载完成后异步加载，从而达到懒加载效果

#### c. 问题: 虽然不会在首屏下载details.js，但是却会在后台异步下载details.js，偷跑流量！



### (2). 彻底的懒加载:

#### a. 如果用户不访问其它页面，则根本一点儿都不下载其它页面的内容

#### b. 如何: 在vue脚手架中添加一个配置信息:

1). 在vue脚手架项目的根目录下创建vue.config.js

2). 在vue.config.js中添加配置信息，禁止提前下载其它组件

module.exports={

chainWebpack:config=>{

config.plugins.delete("prefetch")

//删除index.html开头的带有prefetch属性的link，不要异步下载暂时不需要的页面组件文件

},

}

vue.config.js

|  |
| --- |
| module.exports={    chainWebpack:config=>{      config.plugins.delete("prefetch")      //删除index.html开头的带有prefetch属性的link，不要异步下载暂时不需要的页面组件文件    },  } |

#### c. 结果: 除首页之外的其它页面组件内容，一点都不提前下载。只有当用户点连接，访问到一个页面时，才临时下载这个页面组件的内容

#### d. 问题: 因为当用户访问页面时，临时去下载页面的内容，加载速度稍慢。

# 十六. http-proxy代理跨域:

## 1. 问题:

(1). CORS: 优点: 仅服务器端修改即可; 缺点: 必须知道客户端的具体IP地址

(2). JSONP: 优点: 不需要知道客户端具体IP地址; 缺点: 需要客户端和服务器端都作出调整！

(3). 如果服务器端觉得CORS也不好用，JSONP也不好用，干脆就没有做跨域！单靠前端能否跨域？

## 2. 解决:

利用vue脚手架中自带的http-proxy代理程序来跨域访问。详细信息可查看官方文档: https://cli.vuejs.org/config/#devserver-proxy  
 今后，只要别人问你们公司的项目用的什么跨域，都说:http-procy（http代理）

## 3. 什么是HTTP代理(http-proxy):

在vue客户端项目中，用js临时创建一个小程序，发送http请求（不是ajax）。由这个底层的小程序负责发送请求，接收响应。而客户端vue项目只需要将自己的请求地址交给底层小程序，然后，等着从小程序获取响应结果即可！

## 4. 如何:

### (1). 在vue脚手架配置文件(vue.config.js)中，添加一个新的配置项

module.export={

... ... ,

devServer: {

proxy: {

'/api': { //为所有服务器端接口起一个别名前缀，为了和vue脚手架中其它页面的路由地址区分

target: `http://服务器端接口地址统一前缀`,

changeOrigin: true, //跨域

pathRewrite: {

//因为真实的服务器端地址中是不包含/api的，所以

'^/api' : '' //应该将程序中的/api删除(替换为空字符串)，再和target中的基础路径拼接起来作为发送到服务器的最终请求地址。

}

}

}

}

}

### (2). 在main.js中，axios的baseURL就不能写真实的服务器地址，应该写devServer中定义的所有服务器端地址的别名前缀:

axios.defaults.baseURL="/api"

### (3). 在组件内使用axios发送请求时，依然使用接口的相对地址就可以:

### this.axios.get("/index")

## 5. 原理:

this.axios.get("/index")

↓axios会将baseURL和"/index"拼接

this.axios.get("/api/index")

所有↓带有/api前缀的路径都会交给devServer程序处理

devServer —— 不使用ajax，也能发送http请求的程序

↓先将相对路径中的/api去掉

/index

↓再和提前配置好的这是服务器地址target拼接

http://localhost:5050/index

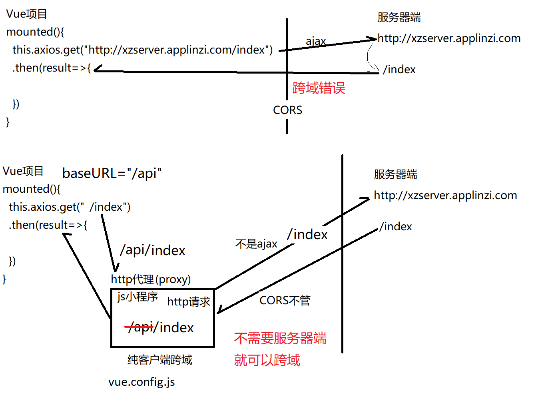
↓向真实的服务器接口地址发送请求

devServer接到响应结果

↓

返回给axios.then(result=>{})继续处理

整个过程因为没有用到ajax，所以不受浏览器同源策略的限制。



## 6. 强调: 只有访问没有启用跨域的服务器端时，才能使用http-proxy。不能和其它跨域方式并存的。

总结  
10. SPA

(1). 3步:

a. 先创建唯一完整的HTML页面

1). 包含vue基本的页面结构

<div id="app"> new Vue({el:"#app"})

2). 引入所有必要的文件和组件

vue-router.js, 其它页面或组件文件, router.js路由器对象所在文件

3). <div id="app">中用<router-view/>为今后页面组件预留空位

b. 再为每个页面组件创建独立的文件。每个页面组件其实都是一个子组件

c. 创建router.js文件，创建路由器对象

1). 创建路由字典对象:

var routes=[

{path:"/", component:首页组件对象名},

{path:"/相对路径" , component: 其它页面组件对象名},

{path:"\*", component: 404页面组件对象 }

]

2). 创建路由器对象，并将路由字典对象转入路由器对象中  
 var router=new VueRouter({ routes })

3). 将router对象加入到new Vue()中

回到唯一完整的HTML页面中: new Vue({ el:"#app", router })

(2). 页头等全局组件：

a. 创建独立的文件保存页头组件的内容

b. 使用Vue.component("my-header",{ ... })将页头创建为全局组件

c. 在唯一完整的HTML页面中引入页头组件文件

d. 使用页头组件标签<my-header/>: 2种:

1). 如果所有页面都有统一的页头:

就放在唯一完整的html页面中<router-view>外部上方

2). 如果有的页面有页头，有的页面没有页头:

就只放在需要页头的组件中的template中

(3). 路由跳转: 2种:

a. html中: <router-link to="/相对路径">文本<router-link>

b. js中: this.$router.push("/相对路径")

(4). 路由传参:

a. 修改路由字典:

{path:"/相对路径/:自定义参数名", component:页面组件对象, props:true}

b. 跳转时:

<router-link to="/相对路径/参数值"

或

this.$router.push("/相对路径/参数值")

c. 下一个页面接:

1). props:[ '自定义参数名' ]

2). 可将"自定义参数名"用作绑定或程序中都行

11. 脚手架文件夹结构:

(1). 唯一完整的HTML页面: 一分为三:

a. public文件夹

1). 图片img文件夹放在public文件夹下

2). 第三方css的压缩版和第三方js的压缩版都放在public文件夹下

3). 唯一完整的HTML文件index.html中，head中引入第三方的css和js

b. src/App.vue

1). <template>下只包含公共的页头组件和<router-view>

2). <style>下包含所有网页都要用到的公共css样式，比如css重置代码

c. src/main.js

1). import引入App.vue，router，axios，以及其他全局组件

2). 将全局组件对象转为真正的全局组件: Vue.component( "组件标签名", 全局组件对象 )

3). 配置axios并放入原型对象中:

axios.defaults.baseURL="服务器端基础路径"

Vue.prototype.axios=axios;

(2). 为每个页面创建.vue组件文件，都放在src/views文件夹下。每个.vue文件中:

a. <template>标签内，包含这个页面的HTML内容

b. <script>export default{ ... }</script>中包含组件对象的js内容。

c. <style>标签内包含仅这个页面组件内才使用的css

d. <template>中的HTML内容以及<script>export default{...}</script>中的js内容，和前四天将的是完全一样的写法，绑定，指令，函数，生命周期，axios请求等都一样。前四天怎么用，这里就怎么用。

(3). 路由字典和路由器对象，在src/router/index.js文件中

a. 仅import首页组件对象，不要过早引入其它页面组件

b. 路由字典中首页组件: { path:"/", component:首页组件对象}

c. 其余页面组件都做成懒加载:

{

path: '/相对路径',

component: () => import(/\* webpackChunkName: "组件名" \*/ '../views/其它页面组件.vue')

}

(4). 全局组件都放在src/components文件夹下，每个全局组件.vue文件中。但是，全局组件必须在main.js引入，并用Vue.component()转化为真正的全局组件，才能在其它组件的HTML中使用。

(5). 运行时: 路由器router对象监视浏览器地址栏路径变化，并查找对应的页面组件内容，先代替App.vue中的<router-view>，然后main.js再将包含要加载的页面内容的App.vue组件内容，替换到唯一完整的index.html中空<div id="app">位置。

12. 懒加载:

(1).异步延迟下载:

src/router/index.js

//不要import Details from "../views/Details"

const routes=[  
 ...

{

path:"/details/:lid",

component: () => import(

/\* webpackChunkName: "details" \*/

'../views/Details.vue'

),

props:true  
]

(2).彻底懒加载:

项目根目录创建vue.config.js

module.exports={

chainWebpack:config=>{

config.plugins.delete("prefetch")

//删除index.html开头的带有prefetch属性的link，不要异步下载暂时不需要的页面组件文件

},

}

13. http-proxy方式跨域:

在服务器端没有配置CORS或JSONP跨域的情况下:   
(1). vue.config.js中

module.exports={  
 ... ,

devServer: {

proxy: {

// detail: https://cli.vuejs.org/config/#devserver-proxy

'/api': {

target: `http://服务器端接口地址统一前缀部分`,

changeOrigin: true,

pathRewrite: {

'^/api' : '' //将程序中的/api，替换为空字符串，再和target中的基础路径拼接起来作为发送到服务器的最终请求地址。

}

}

}

}  
}  
(2). main.js中:   
...

axios.defaults.baseURL="/api"

14. 避免组件间样式冲突:

(1)<style scoped>  
(2)<template>  
 <组件父元素 class="组件名">  
 </template>  
 <style>  
 .组件名>子元素选择器{ ... }  
 </style>

15. watch+事件修饰符+防抖:

(1). 希望变量一变化就自动执行一项操作时:

data(){  
 return { 变量: 值 }  
},

watch:{  
 变量(){  
 要执行的操作

}  
}  
(2). 事件修饰符:

@事件.13 按回车时才执行

@事件.stop 阻止冒泡

@事件.prevent 阻止默认行为

(3). 防抖:

data(){

return {

... ,

timer:null

}

},

methods:{

查找方法(){

if(this.timer!=null){

clearTimeout(this.timer)

}

this.timer=setTimeout(()=>{

正式的查找操作

},等待时间)

}

}

今日对应小程序视频:

小程序->在线->VUE->day05  
 扩展0: Vue脚手架中使用http-proxy跨域

小程序->在线->VUE->day04

3\_vue脚手架文件夹结构(含组件间样式冲突)

小程序->在线->VUE->day02

3. 双向绑定v-model绑定文本框(含watch和防抖)

作业:

1. 复习今日问题清单: 小程序->在线->VUE->day04、day05

2. 看小程序视频使用vue脚手架完成学子商城首页、详情页

先下载: 小程序->在线->VUE->day05 0. VUE版学子商城 起始空项目 中

旧js项目: public.zip

vue脚手架起始项目: xzvue\_start\_with\_axios.zip

观看并跟着做: 小程序->在线->VUE->day05

1. 学子商城首页

2. 学子商城详情页 。。。（不带放大镜）

3. 学子商城详情页 。。。（放大镜效果）

家里环境xampp和nodejs能用，且第一阶段学的好的同学:

可下载0. VUE版学子商城 起始空项目 中xzserver.zip，用自己的服务器端代码

家里环境xampp和nodejs不能用，或者第一阶段学的不好的同学：

无需下载xzserver.zip

视频中所有axios请求的服务器端接口地址，都改为我的新浪云服务器地址，就不需要你再运行服务器端了。也不需要你了解任何服务器端知识了:

http://localhost:5050/index 改为 http://xzserver.applinzi.com/index

http://localhost:5050/details 改为 http://xzserver.applinzi.com/details?lid=xxx

# 总结:

## 1. 绑定语法(1种)和指令(13种):

(1). 只要元素的内容随变量自动变化: {{}} v-text

(2). 只要元素的属性值随变量自动变化: :

(3). 控制一个元素显示隐藏: v-show （display:none）

(4). 控制两个元素二选一显示: v-if v-else （删除元素）

(5). 控制多个元素多选一显示: v-if v-else-if v-else

(6). 反复创建多个相同结果的元素: v-for :key="下标"

(7). 绑定事件: @ $event

(8). 防止用户短暂看到{{}}: v-cloak v-text

(9). 只要绑定的内容是HTML片段: v-html

(10). 只要希望一个元素只在首次加载时绑定一次，之后都不会自动更新: v-once

(11). 只要不希望元素内容中的{{}}被vue解析，而是保持原样显示: v-pre

(12). 只要希望随时获得页面上用户在表单元素中修改的新值: v-model

## 2. 绑定样式:

(1). 如果希望精确修改某一个css属性的值时，绑定style内联样式:

a. 为每个要修改的css属性分别设置变量

<元素 style="固定不变的css属性 " :style="{ css属性1: 变量1, css属性2: 变量2, ... }"

data:{

变量1: css属性值1,

变量2: css属性值2

}

b. 为整个style属性设置一个变量对象，其中包含多个css属性和属性值

<元素 style="固定不变的css属性 " :style="xxxStyle"

data:{

xxxStyle:{

css属性1: css属性值1,

css属性2: css属性值2

}

}

(2). 如果希望批量修改一个元素的多个css属性的值时，就要绑定class属性:

a. 为每个class分别设置条件

<元素 class="不变的class " :class="{class1: bool表达式或变量, class2:bool表达式或变量 }"

bool表达式或变量的值为true就应用，false就不应用

b. 为整个class属性设置一个变量对象，其中包含多个备选的class名极其对应判断条件

<元素 class="不变的class " :class="xxxClass"

data:{

xxxClass:{

class1: true或false, //true就应用，false就不应用

class2: true或false //true就应用，false就不应用

}

}

## 3. 自定义指令: 2步

(1). 添加自定义指令:

Vue.directive("指令名",{

inserted(domElem){

对domElem执行DOM操作

}

})

(2). 使用自定义指令: <元素 v-自定义指令名>

## 4. 计算属性:

(1). 定义:

new Vue({

... ,

computed:{

自定义属性名(){

计算过程

return返回值

}

}

})

(2). 使用计算属性: <元素>{{自定义属性名}}</元素>

## 5. 过滤器:

(1)不带参过滤器:

a. 添加过滤器:

Vue.filter("过滤器名", function(oldVal){

return 根据不同的oldVal加工出不动的新值

})

b. 使用过滤器: <元素>{{ 变量 | 过滤器 }}</元素>

(2). 带参数过滤器：

a. 添加过滤器:

Vue.filter("过滤器名", function(oldVal, 自定义形参){

return 根据不同的oldVal加工出不动的新值

})

b. 使用过滤器: <元素>{{ 变量 | 过滤器(自定义实参) }}</元素>

(3). 多个过滤器连用: <元素>{{ 变量 | 过滤器1 | 过滤器2 | ...</元素>

## 6. axios:

(0). axios.default.baseURL="http://所有服务器端接口统一的域名部分"

(1). axios.get("/接口名", { params:{ 参数:值, 参数:值, ... } })

.then(function(result){

result.data ...

})

(2). axios.post("/接口名", "参数1=值1&参数2=值2&...")

.then(function(result){

result.data ...

})

## 7. vue生命周期: 4个阶段 8个钩子函数

beforeCreate()

(1). 创建阶段 create

created()

beforeMount()

(2). 挂载阶段 mount

mouted()

beforeUpdate()

(3). 更新阶段 update

updated()

beforeDestroy()

(4). 销毁阶段 destroy

destroyed()

# display:none 与 visibility:hidden 的区别是什么？

display: none; 与 visibility: hidden; 的共同点是：它们都能让元素隐藏（不可见）。

当然这两者对元素的隐藏处理也是有区别的，记录如下：

**空间占用方面**

display: none; 会让元素完全从渲染树（render tree）中消失，渲染的时候不占据任何空间，如同不存在（但其依然存在 DOM 树中）；

而 visibility: hidden; 不会让元素从渲染树消失，渲染时元素仍继续占据空间，只是内容不可见。

**是否继承**

display: none; 为非继承属性，子孙节点都会消失（株连式），由于元素从渲染树消失，通过修改子孙节点属性也无法使子孙节点显示；

而 visibility: hidden; 是继承属性，子孙节点由于继承了 visibility: hidden; 而消失，如果子孙节点设置了 visibility: visible;，则不继承，子孙节点又可以显示出来（伪株连式）。

**回流与重绘**

修改常规流中元素的 display 属性通常会造成回流（reflow）与重绘（repaint）；

而修改 visibility 属性只会造成本元素的重绘。

对这些区别，网上有个很**形象的对比**：

display:none; 是个相当惨无人道的声明，子孙后代全部搞死（株连性），而且连块安葬的地方都不留（不留空间），导致全体民众哗然（回流与重绘）。  
visibility:hidden; 则具有人道主义关怀，虽然不得已搞死子孙，但是子孙可以通过一定手段避免（伪株连性），而且死后全尸，墓地俱全（占据空间），国内民众比较淡然（重绘）。

# 导入&导出

由于HTML 中的特性名是大小写不敏感的，所以在JS中给Prop命名时用驼峰命名法（camelCase），对应在DOM中必须用短横线分隔命名（kebab-case）与Prop的驼峰命名对应**。**

# Element组件属性，事件，方法的使用

* 在使用组件的方法时需要在对应的组件中加入 ref = “组件别名”
* 在调用方法时直接使用 this.$refs.组件别名.方法名()

注意：在element-UI中所有组件 都存在 属性 事件 和方法

* 属性：直接写在对应的组件标签上，使用方法 :属性名 = 属性值 方式
* 事件：直接使用vue绑定事件方式写在对应的组件标签上，使用方式 @事件名 = vue中事件处理函数
* 方法： 1.在对应组件标签上使用 ref = “组件别名” ，2.通过使用this.$refs.组件别名.方法名() 进行调用