目录

[方法部分很重要！ 29](#_Toc14775)

[词法结构 30](#_Toc12456)

[代码的解析 30](#_Toc27113)

[JS的流程 31](#_Toc29981)

[JS引用外联样式 31](#_Toc15111)

[JS注释 31](#_Toc19814)

[JS的背景 31](#_Toc23015)

[获取元素 32](#_Toc22967)

[静态方法document.getElementById('id') 32](#_Toc22968)

[动态方法.getElementsByClassName 33](#_Toc19732)

[动态方法.getElementsByTagName 33](#_Toc984)

[querySelector 35](#_Toc11321)

[querySelectorAll 36](#_Toc3964)

[classList选择器 37](#_Toc18736)

[语法 37](#_Toc18317)

[方法 37](#_Toc2104)

[toggle 38](#_Toc6241)

[remove清除类选择器的样式 40](#_Toc8476)

[add 添加类选择器的样式 40](#_Toc3125)

[批量设置行间样式cssText 40](#_Toc26685)

[a标签 41](#_Toc13467)

[声明和赋值 41](#_Toc17615)

[变量 42](#_Toc12345)

[常量 45](#_Toc26139)

[属性操作 46](#_Toc14212)

[window 46](#_Toc6091)

[属性 47](#_Toc12388)

[属性的读操作 47](#_Toc21888)

[获取行间样式 47](#_Toc2017)

[获取样式表/当前样式 48](#_Toc78)

[getComputedStyle与style的区别 48](#_Toc19015)

[不能拿来做判断的样式 49](#_Toc27851)

[属性的写操作 49](#_Toc30055)

[修改元素行间样式 49](#_Toc30413)

[修改元素外联样式 50](#_Toc7644)

[添加自定义属性 50](#_Toc28701)

[添加自定义索引 50](#_Toc1918)

[[] 中括号的用法 51](#_Toc10654)

[1.变量赋值 51](#_Toc4713)

[2.数组 52](#_Toc4787)

[3.(类)数组的下标 52](#_Toc10564)

[input中的value 52](#_Toc2293)

[逻辑性 53](#_Toc679)

[赋值为空 53](#_Toc627)

[字符串拼接 54](#_Toc22229)

[第一种写法 54](#_Toc20241)

[另一种写法：超级字符串（ES6） 54](#_Toc29732)

[复合样式的设置 55](#_Toc20932)

[面向过程 55](#_Toc12234)

[判断语句 55](#_Toc22711)

[if(条件){} 56](#_Toc24038)

[if(条件) 56](#_Toc13219)

[if(){}else if(){} 56](#_Toc4863)

[if(){}else if(){}else{} 56](#_Toc25056)

[if(){}else{} 58](#_Toc16977)

[不能作为判断条件的3个值 58](#_Toc15860)

[switch case defult break 58](#_Toc10138)

[try...catch语句 60](#_Toc31407)

[ECMAScript数据类型 61](#_Toc8267)

[为什么要有数据类型？ 61](#_Toc9749)

[复合数据类型和简单数据类型的区别？ 62](#_Toc30536)

[typeof的数据类型 62](#_Toc18597)

[对象数据类型 object 63](#_Toc32669)

[原始类型(基本类型) 63](#_Toc11193)

[1. Number（数字） 63](#_Toc32202)

[2. String（字符串） 66](#_Toc26648)

[3. 布尔类型 67](#_Toc7324)

[4. Undefined（未定义） 68](#_Toc9716)

[5. 空(Null): 69](#_Toc20913)

[6. 唯一symbol 70](#_Toc481)

[复杂（对象）数据类型不能运算 70](#_Toc30474)

[原始数据类型的运算 70](#_Toc26258)

[赋值运算符 70](#_Toc19045)

[关系运算符 71](#_Toc26966)

[逗号运算符 72](#_Toc18487)

[逻辑运算符 73](#_Toc26541)

[三元运算符（三目运算符）： 74](#_Toc18284)

[二元运算符: 74](#_Toc22136)

[一元运算符: 74](#_Toc9323)

[逻辑运算符的优先顺序 75](#_Toc30591)

[自增自减 75](#_Toc31977)

[运算顺序 76](#_Toc4205)

[执行顺序 76](#_Toc19336)

[取余 76](#_Toc22132)

[相加赋值 76](#_Toc26979)

[运算的优先级 77](#_Toc7216)

[数学方法Math 77](#_Toc27884)

[向下取整 77](#_Toc31619)

[向上取整 78](#_Toc22728)

[四舍五入 78](#_Toc17287)

[取绝对值 78](#_Toc25237)

[取几个数字中的最大值 78](#_Toc29375)

[取几个数字中的最小值 78](#_Toc6322)

[取随机数 78](#_Toc2101)

[幂运算 79](#_Toc10120)

[数字的二进制浮点数 80](#_Toc2601)

[字符串长度 81](#_Toc16872)

[原始数据类型的转换 82](#_Toc22385)

[字符串转数字 82](#_Toc3443)

[Number(); 82](#_Toc16585)

[parseInt(a,b) 83](#_Toc15144)

[parseFloat(a) 84](#_Toc4504)

[数字转字符串 84](#_Toc30097)

[String(a) 84](#_Toc11261)

[隐式转换 85](#_Toc22820)

[隐式转换 86](#_Toc14879)

[隐式转换成字符串 87](#_Toc4302)

[转换成布尔类型 88](#_Toc18192)

[数组 89](#_Toc16330)

[一维数组 89](#_Toc25547)

[二维数组 90](#_Toc27436)

[给数组添加元素push 90](#_Toc24846)

[类数组 90](#_Toc26824)

[如何将一个类数组转换成数组？ 91](#_Toc11556)

[1. slice 91](#_Toc18021)

[2. Array.from() 91](#_Toc7383)

[3. 扩展运算符 92](#_Toc29520)

[循环 92](#_Toc24871)

[Switch 92](#_Toc2002)

[while循环/for循环 93](#_Toc14195)

[while(条件){} 94](#_Toc8472)

[for循环 94](#_Toc4155)

[控制流程 98](#_Toc15755)

[continue和break 98](#_Toc2213)

[return 99](#_Toc9382)

[window 100](#_Toc13670)

[This 101](#_Toc32464)

[1.默认情况 101](#_Toc29123)

[2.函数和属性 102](#_Toc9418)

[3.方法和函数 103](#_Toc15297)

[无穷大 106](#_Toc11175)

[选项卡 107](#_Toc11972)

[1.大清洗 107](#_Toc6508)

[2.定点清除 107](#_Toc30761)

[数据类型 108](#_Toc20423)

[javascript的构成 108](#_Toc15771)

[ECMAScript 108](#_Toc25991)

[函数 109](#_Toc27448)

[什么是“函数”？ 109](#_Toc11445)

[什么是“方法”？ 109](#_Toc20420)

[函数声明 109](#_Toc21110)

[类型 111](#_Toc24435)

[匿名函数 111](#_Toc12077)

[有名函数 112](#_Toc31260)

[函数的调用方式 112](#_Toc20591)

[1.直接调用函数（打开网页即调用函数） 112](#_Toc14277)

[2.事件触发调用函数 113](#_Toc21443)

[3.定时器调用 114](#_Toc5877)

[4.回调函数(钩子函数callback) 115](#_Toc8269)

[函数传参 115](#_Toc557)

[形参的默认值（ES6） 117](#_Toc16074)

[作用 117](#_Toc3690)

[封装 117](#_Toc19090)

[fn() 函数名+括号做什么？ 118](#_Toc8883)

[fn()() 函数名+双括号做什么？ 118](#_Toc2933)

[函数的调用 118](#_Toc30960)

[函数传参 120](#_Toc1964)

[arguments 122](#_Toc10912)

[...arr 125](#_Toc5692)

[想要获取函数内值的方法 125](#_Toc20526)

[函数返回值 126](#_Toc32119)

[函数返回值的目的 129](#_Toc12270)

[函数使用注意事项 129](#_Toc18049)

[调试 130](#_Toc8719)

[函数的特性 131](#_Toc30212)

[函数自执行 131](#_Toc14590)

[域 132](#_Toc7028)

[作用域链 134](#_Toc31253)

[作用域和作用域链 136](#_Toc29593)

[什么是作用域？ 136](#_Toc4233)

[全局作用域与局部作用域 136](#_Toc52)

[JS特性 136](#_Toc29936)

[预解析 140](#_Toc17912)

[例子 140](#_Toc5593)

[什么是js预解析机制: 140](#_Toc11063)

[如何预解析（也叫变量提升） 140](#_Toc12879)

[预解析注意点 141](#_Toc1252)

[只有函数的预解析 141](#_Toc23865)

[只有变量的预解析 142](#_Toc3307)

[函数+变量的预解析 143](#_Toc6390)

[什么都没有的预解析 144](#_Toc5090)

[作用域链 144](#_Toc13482)

[逻辑顺序 145](#_Toc4509)

[多个script标签 145](#_Toc8464)

[如何解决报错 146](#_Toc4925)

[<script>标签可以放在html中 147](#_Toc24791)

[闭包 147](#_Toc15003)

[闭包是个啥？ 147](#_Toc6343)

[如何形成闭包环境？ 148](#_Toc25164)

[闭包的好处？ 150](#_Toc9803)

[应用 151](#_Toc21141)

[举例 152](#_Toc5907)

[闭包的应用价值 153](#_Toc5879)

[闭包的优缺点 154](#_Toc29410)

[优点 154](#_Toc16713)

[缺点 155](#_Toc4303)

[定时器 156](#_Toc3704)

[定时器概念 156](#_Toc15229)

[关闭定时器概念 157](#_Toc16992)

[函数的第3种调用方式:定时器调用 159](#_Toc17426)

[定时器的问题 160](#_Toc31583)

[setTimeout 160](#_Toc25082)

[计时器 161](#_Toc914)

[时间对象 161](#_Toc28092)

[获取已有（本机/服务器）时间 162](#_Toc5337)

[声明时间 162](#_Toc14180)

[设置时间 163](#_Toc28177)

[1.时间戳数字 163](#_Toc32079)

[2.也可以放字符串 163](#_Toc19249)

[3.还可以传入数字 164](#_Toc28262)

[4.数字版字符串 164](#_Toc7973)

[时间戳转时间公式 165](#_Toc15105)

[date的操作 165](#_Toc9623)

[运动函数tween 167](#_Toc4614)

[对象拷贝 167](#_Toc21523)

[浅拷贝/浅克隆 167](#_Toc5135)

[1.方法一Object.assign 167](#_Toc11553)

[2.方法二for in 169](#_Toc21648)

[深拷贝 170](#_Toc19780)

[1.方法一：JSON.parse(JSON.stringify数据(arr)); 170](#_Toc339)

[2.方法二：利用for in +递归+push 170](#_Toc2319)

[浅拷贝和深拷贝的区别 170](#_Toc19151)

[赋值/赋址 171](#_Toc27811)

[赋址小问题 173](#_Toc15385)

[字符串 176](#_Toc12371)

[超级字符串 176](#_Toc26499)

[字符串操作 177](#_Toc4935)

[字符串比较大小 177](#_Toc13615)

[长度length 177](#_Toc21585)

[截取类 177](#_Toc19216)

[查询类 183](#_Toc594)

[分割合成类 189](#_Toc571)

[编码转换类 191](#_Toc26851)

[大小写转换类 192](#_Toc15220)

[增删类 194](#_Toc17177)

[数组 200](#_Toc13649)

[数组属性 200](#_Toc28064)

[长度length 201](#_Toc22651)

[数组方法 202](#_Toc23173)

[截取类 202](#_Toc21189)

[查询类 205](#_Toc23646)

[实例 207](#_Toc24630)

[分割合成类 207](#_Toc19385)

[增删类 209](#_Toc22846)

[ES5类 213](#_Toc24821)

[ES6类 219](#_Toc6057)

[sort(function(a,b){}) 222](#_Toc11764)

[concat 227](#_Toc4108)

[reverse 228](#_Toc15942)

[reduce 228](#_Toc11204)

[Array.isArray（arr） 230](#_Toc22969)

[数组转字符串方法 230](#_Toc4028)

[对象 230](#_Toc1148)

[对象基础 230](#_Toc7829)

[对象的key值和value值相同 232](#_Toc28974)

[1.什么是对象？ 232](#_Toc14569)

[对象里面的第几个 233](#_Toc31890)

[2.什么是json? 233](#_Toc5307)

[3.‘Json’ 和 ‘对象’ 的区别 235](#_Toc18371)

[方法 236](#_Toc3413)

[json 转 对象 236](#_Toc27926)

[对象 转 json（字符串） 238](#_Toc5918)

[对象 转 数组、字符串（表单序列化） 239](#_Toc27923)

[toString 240](#_Toc7992)

[面向对象 242](#_Toc7212)

[什么是构造函数？ 243](#_Toc7389)

[1.内置构造函数和自定义构造函数 243](#_Toc24335)

[2.检测构造函数 245](#_Toc2775)

[创建对象 246](#_Toc13013)

[面向对象的规则 248](#_Toc29064)

[new操作符具体干了什么 249](#_Toc10073)

[一般面向对象的写法 250](#_Toc11865)

[原型和原型链 250](#_Toc25969)

[原型(显式原型) 251](#_Toc19344)

[原型链(隐式原型) 252](#_Toc26606)

[如何用面向对象写程序 254](#_Toc5078)

[hasOwnProperty 255](#_Toc1411)

[instanceof 255](#_Toc9979)

[包装对象 256](#_Toc6920)

[this指向 257](#_Toc16818)

[严格模式 259](#_Toc24757)

[改变this指向call/apply/bind 260](#_Toc14428)

[继承 263](#_Toc2627)

[什么是继承？ 263](#_Toc25897)

[继承了什么？ 264](#_Toc4380)

[继承的方法有哪些？ 264](#_Toc3075)

[多态 267](#_Toc16638)

[组件化开发 267](#_Toc2858)

[对象方法 268](#_Toc8892)

[Object.keys() (ES6) 268](#_Toc9585)

[Object.values() (ES6) 268](#_Toc32260)

[Object.entries() (ES6) 268](#_Toc23401)

[Object.assign()(ES6) 269](#_Toc13071)

[Object.is 270](#_Toc26732)

[使用[]代表变量 270](#_Toc2480)

[省略function(ES6) 270](#_Toc26946)

[类 271](#_Toc10783)

[class 271](#_Toc3005)

[类的继承extends 271](#_Toc29045)

[ES6语法 272](#_Toc31489)

[对象解构赋值 272](#_Toc24306)

[Set 274](#_Toc24406)

[Map 275](#_Toc4018)

[WeakMap 276](#_Toc1908)

[for in 276](#_Toc27995)

[作用 276](#_Toc23821)

[格式 276](#_Toc20209)

[in 运算符 277](#_Toc1737)

[注意 277](#_Toc1381)

[for of 278](#_Toc27415)

[扩展运算符... 279](#_Toc22950)

[将数组或者对象里面的值展开 280](#_Toc8821)

[将传入的类数组转为数组 281](#_Toc28737)

[属性名和属性值相同省略 281](#_Toc7464)

[使用匿名函数避免全局污染 281](#_Toc26306)

[省略function 283](#_Toc8053)

[箭头函数 283](#_Toc24962)

[什么是箭头函数？ 283](#_Toc4314)

[箭头函数的作用： this指向主的老爹 285](#_Toc13445)

[函数缩写 286](#_Toc15834)

[Promise 286](#_Toc26765)

[为什么需要promise 286](#_Toc26963)

[基本概念 287](#_Toc9620)

[promise的方法 289](#_Toc12676)

[模块化 294](#_Toc5706)

[import/ export 295](#_Toc11810)

[算法 296](#_Toc13387)

[递归 296](#_Toc6190)

[1.冒泡排序法 297](#_Toc11040)

[2.快速排序法 297](#_Toc4459)

[数组去重 298](#_Toc12372)

[方法一： 298](#_Toc4267)

[方法二： 299](#_Toc24865)

[方法三：（最简单） 300](#_Toc24900)

[方法四：（复杂数据类型的数组去重） 300](#_Toc31353)

[判断arr是不是数组 301](#_Toc22382)

[1.Array.isArray（arr） 301](#_Toc9857)

[2.arr instance Aarray 302](#_Toc10334)

[3.arr.constructor == Array 302](#_Toc12982)

[4.Object.prototype.toString.call(arr) 302](#_Toc3152)

[5.Object.prototype.toString.apply(arr) 302](#_Toc23629)

[数据类型的转换 302](#_Toc3219)

[避免帕金森的方法 303](#_Toc1422)

[git和github 304](#_Toc27993)

[下载远程仓库代码 304](#_Toc25776)

[上传的基本流程 305](#_Toc15793)

[后续上传的基本流程 306](#_Toc21521)

[pull远程更新代码 307](#_Toc20859)

[工作区和暂存区的添加撤销 307](#_Toc27020)

[撤销版本 308](#_Toc15010)

[delete删除参数 309](#_Toc23795)

[DOM 309](#_Toc20726)

[DOM加载顺序 309](#_Toc20728)

[childNodes[num] 310](#_Toc13821)

[nodeName 311](#_Toc11326)

[nodeValue 311](#_Toc7225)

[tagName 311](#_Toc19456)

[children 312](#_Toc5554)

[lastElementChild 312](#_Toc25847)

[firstElementChild 312](#_Toc3018)

[attributes 313](#_Toc19553)

[data-attrName /dataset. attrName 314](#_Toc21148)

[nodeType 315](#_Toc29275)

[parentNode 父节点 316](#_Toc7293)

[ElementSibling 兄弟节点 317](#_Toc24555)

[offsetParent父级属性操作 318](#_Toc30293)

[Attribute行间属性操作 318](#_Toc24168)

[元素的基点 320](#_Toc31721)

[获取指定元素的细节信息 320](#_Toc23337)

[ele.getBoundingClientRect().left/top/right/bottom 320](#_Toc11086)

[元素的宽高 321](#_Toc29087)

[HTMLElement.style  321](#_Toc14620)

[ele.scrollHeight/scrollWidth 321](#_Toc5956)

[ele.clientWidth/elem.clientHeight 322](#_Toc4895)

[ele.offsetWidth/elem.offsetHeight 323](#_Toc9)

[滚动距离 323](#_Toc28046)

[ele.scrollTop 323](#_Toc18213)

[ele.scrollLeft 323](#_Toc22299)

[document.documentElement.scrollTop 323](#_Toc1912)

[document.documentElement.scrollLeft 323](#_Toc29268)

[document.documentElement.style.setProperty（变量，新值） 323](#_Toc10202)

[元素的边距 324](#_Toc10024)

[ele.offsetLeft/offsetTop 324](#_Toc783)

[页面可视区的宽高clientWidth/clientHeight 325](#_Toc30102)

[document.documentElement.clientWidth/clientHeight 325](#_Toc6862)

[body.clientWidth/clientHeight 325](#_Toc6148)

[页面的实际尺寸 326](#_Toc1091)

[document.documentElement.scrollHeight/ scrollWidth 326](#_Toc9165)

[创建元素 326](#_Toc10381)

[插入元素方法一： 326](#_Toc32677)

[插入元素方法二： 327](#_Toc23555)

[删除元素 327](#_Toc11494)

[替换元素 327](#_Toc10100)

[动态/静态数组 328](#_Toc21145)

[克隆 328](#_Toc6510)

[offsetParent 328](#_Toc25106)

[表格 329](#_Toc3823)

[title网页标题 330](#_Toc6505)

[body网页正文 331](#_Toc5755)

[innerHTML 和 innerText 331](#_Toc6838)

[scrollIntoView() 332](#_Toc26849)

[Dom操作带来性能下降 332](#_Toc17983)

[BOM 333](#_Toc3387)

[文本输入流 334](#_Toc14067)

[关闭窗口 335](#_Toc10738)

[navigator/location/history 336](#_Toc31725)

[用户信息navigator 336](#_Toc21191)

[地址栏信息location 336](#_Toc16077)

[历史信息history 339](#_Toc25643)

[滚动距离window.pageYOffset/pageXOffset 339](#_Toc17968)

[页面可视区的宽高window.innerWidth/innerHeight 340](#_Toc15991)

[滚动事件 340](#_Toc31885)

[缩放事件 341](#_Toc1984)

[设置内容滚动到指定坐标 341](#_Toc10038)

[元素的位置信息Window.getComputedStyle() 342](#_Toc4436)

[事件 342](#_Toc5731)

[传统事件（DOM0事件） 342](#_Toc9772)

[监听事件（绑定事件、DOM2事件） 345](#_Toc7775)

[解绑事件 346](#_Toc29136)

[Event:事件对象 348](#_Toc27472)

[事件流（事件模型）、什么是事件冒泡和捕获？ 348](#_Toc27523)

[冒泡的应用：事件委托（事件源） 349](#_Toc3137)

[冒泡的弊端 351](#_Toc20405)

[如何解决冒泡 351](#_Toc14147)

[浏览器的默认行为 352](#_Toc16900)

[焦点事件 353](#_Toc6126)

[键盘事件 353](#_Toc17500)

[键盘事件注意点 354](#_Toc17027)

[鼠标事件 356](#_Toc21486)

[鼠标移动事件 357](#_Toc31097)

[弹出右键菜单 357](#_Toc17183)

[鼠标滚轮事件 357](#_Toc8268)

[input/textarea表单事件 360](#_Toc24470)

[t.focus(); 360](#_Toc19894)

[t.select(); 360](#_Toc11792)

[selectionStart 360](#_Toc16573)

[selectionEnd 360](#_Toc4533)

[setSelectionRange 360](#_Toc22338)

[t.oninput(); 361](#_Toc8940)

[键盘控制时的停顿问题 362](#_Toc23026)

[事件应用之拖拽 363](#_Toc23046)

[拖拽避免默认行为 364](#_Toc20675)

[解决办法1： 365](#_Toc5194)

[解决办法2： 365](#_Toc27757)

[解决办法3： 366](#_Toc13425)

[仿window拖拽 367](#_Toc792)

[回放 367](#_Toc26298)

[碰撞 369](#_Toc28225)

[鼠标画框 370](#_Toc7382)

[定义滚动条 371](#_Toc27720)

[正则表达式 372](#_Toc11346)

[什么是正则 372](#_Toc24893)

[正则关键字 373](#_Toc8913)

[test方法 376](#_Toc16071)

[search方法 377](#_Toc961)

[match方法 377](#_Toc31093)

[replace方法 378](#_Toc23777)

[子项（子串） 379](#_Toc28079)

[（$0,$1,$2,$3,$4,...） 380](#_Toc12292)

[没有子项的时候就是默认模式 380](#_Toc22779)

[有子项的时候$0后面的就是子项， 380](#_Toc25504)

[中括号 381](#_Toc29373)

[大括号 381](#_Toc14645)

[其他转义符 382](#_Toc18325)

[边界符 383](#_Toc32654)

[中文区间范围 383](#_Toc21765)

[UNICODE与ASCII的区别 384](#_Toc10346)

[Node简介 385](#_Toc3856)

[AJAX 386](#_Toc25616)

[AJAX的缺点 388](#_Toc2020)

[传统方式请求数据：表单 388](#_Toc15066)

[AJAX的交互模型: 389](#_Toc29143)

[一.打开浏览器 new XMLHttpRequest 389](#_Toc7097)

[二.在地址栏输入地址open 389](#_Toc13879)

[三.提交地址 394](#_Toc20545)

[四.等待服务器返回内容 394](#_Toc28612)

[Jquery的ajax获取数据 399](#_Toc2503)

[如何看接口 400](#_Toc19914)

[AJAX如何处理XML数据 400](#_Toc6278)

[ajax可能回出现的问题 401](#_Toc19765)

[AJAX的封装 401](#_Toc9457)

[最简单的封装 401](#_Toc24340)

[终极版封装 402](#_Toc17563)

[传统方式的上传数据 403](#_Toc4195)

[使用AJAX监控上传进度 404](#_Toc20320)

[跨域（跨源） 404](#_Toc9505)

[什么是跨域（跨源） 404](#_Toc16038)

[如何实现跨域（跨源） 406](#_Toc26863)

[cookie 409](#_Toc28667)

[什么是cookie? 409](#_Toc18895)

[作用 410](#_Toc11614)

[在哪里使用？ 410](#_Toc13758)

[设置cookis生命周期 410](#_Toc12519)

[封装形式：设置cookie 411](#_Toc14909)

[封装形式：删除cookie 412](#_Toc18103)

[获取cookie 412](#_Toc7431)

[封装形式：获取cookie 413](#_Toc1870)

[localStorage/sessionstorage本地存储 413](#_Toc8065)

[存储的方法：增删改查 414](#_Toc13761)

[storage与cookie的区别 414](#_Toc27306)

[Node 415](#_Toc26720)

[什么是Node? 415](#_Toc20377)

[阻塞和非阻塞 415](#_Toc6093)

[如何使用? 416](#_Toc10436)

[require 方法 416](#_Toc20376)

[listen方法 417](#_Toc19108)

[createServer方法 417](#_Toc21715)

[response.write( )/response.end( )方法 417](#_Toc9856)

[readFile 方法 418](#_Toc31297)

[fs.readFileSync(path[, options])方法 421](#_Toc6227)

[writeFile方法 422](#_Toc22288)

[unlink方法 422](#_Toc24880)

[querystring.parse(str[, sep[, eq[, options]]]) 422](#_Toc31800)

[注意 423](#_Toc6554)

[数据库 424](#_Toc26099)

[如何开启 424](#_Toc924)

[React/jquery/yarn 环境安装 425](#_Toc4595)

[安装插件npm i ... –S 记得要保存-S 428](#_Toc8875)

[React 428](#_Toc10298)

[虚拟(Virtual)DOM 429](#_Toc28569)

[index页面 430](#_Toc21442)

[在只有组件的index页面 430](#_Toc28964)

[没有组件的index页面 431](#_Toc4441)

[组件 431](#_Toc1502)

[组件的写法 432](#_Toc11861)

[标准类声明 433](#_Toc21127)

[组件 & Props 448](#_Toc15899)

[Props 448](#_Toc8212)

[组件通信 449](#_Toc17535)

[自动加载 452](#_Toc3820)

[为什么要避免DOM操作？ 453](#_Toc1163)

[React 路由 453](#_Toc28972)

[什么是路由？ 453](#_Toc7737)

[如何使用路由 455](#_Toc27634)

[index.js注意点 455](#_Toc15125)

[router.js路由 456](#_Toc16373)

[生命周期 461](#_Toc18666)

[触发render的方式 465](#_Toc23547)

[Refs & DOM 466](#_Toc31465)

[React 注意 467](#_Toc19686)

[Redux 467](#_Toc11673)

[Action 469](#_Toc24992)

[state 469](#_Toc3161)

[reducer 469](#_Toc7220)

[store 470](#_Toc20291)

[数据流 470](#_Toc5666)

[webpack 471](#_Toc18832)

[Vue 473](#_Toc24212)

[什么是vue? 473](#_Toc31565)

[引入vue.js 473](#_Toc5319)

[VUE实例 474](#_Toc12444)

[创建VUE实例 474](#_Toc16064)

[选项：挂载点el 474](#_Toc24599)

[选项：数据data 475](#_Toc1724)

[选项：methods方法 477](#_Toc26884)

[选项：生命周期(组件的钩子函数) 477](#_Toc7411)

[选项：模板语法template 480](#_Toc6276)

[选项：计算属性computed 480](#_Toc10409)

[选项：侦听属性watch 485](#_Toc21731)

[选项：template 486](#_Toc19112)

[全局钩子函数 488](#_Toc2221)

[指令 491](#_Toc1168)

[指令的作用 491](#_Toc29501)

[指令的参数 491](#_Toc13220)

[指令的修饰符 491](#_Toc29399)

[指令的缩写 492](#_Toc761)

[1.v-bind 492](#_Toc15507)

[2.v-for 499](#_Toc6198)

[3.v-text和v-html 507](#_Toc22691)

[4.v-if和v-show 507](#_Toc5964)

[5.v-on事件处理、改变数据 509](#_Toc11701)

[6.表单输入绑定v-model 515](#_Toc15550)

[特殊特性 519](#_Toc31877)

[ref 519](#_Toc2942)

[组件化 520](#_Toc29912)

[1.注册组件 521](#_Toc12908)

[2.进行声明式渲染 523](#_Toc6007)

[3.引用组件 524](#_Toc12015)

[组件的prop 525](#_Toc29928)

[混入 (mixins) 541](#_Toc7498)

[slot 541](#_Toc5871)

[使用slot分发内容 541](#_Toc27795)

[指定替换 542](#_Toc4896)

[<template></template>标签 543](#_Toc7037)

[编译作用域 544](#_Toc30478)

[动态组件 545](#_Toc3153)

[vue-cli脚手架 546](#_Toc727)

[vue-router 547](#_Toc12744)

[$route 547](#_Toc3354)

[router-link 548](#_Toc9128)

[query对象 548](#_Toc925)

[过渡效果css 549](#_Toc24913)

[过渡模式 550](#_Toc22062)

[编程式导航 551](#_Toc31128)

[导航钩子函数 551](#_Toc17798)

[vuex 555](#_Toc7728)

[字符串模板 555](#_Toc25024)

[Jquery 556](#_Toc26566)

[移动端 556](#_Toc517)

[Question 556](#_Toc4868)

[Q1 558](#_Toc21188)

[image相关语法 560](#_Toc17790)

[new Image() 560](#_Toc30929)

[image.onload 560](#_Toc6143)

[image.onerror 560](#_Toc15345)

# 方法部分很重要！

**小程序！**

**redux!**

**表单KEYS**

script标签与javascript标签的区别？

没有javascript这个标签。

**强类型语言**

对每一种数据类型都有定义，不同类型的变量存储不同类型的数据。

**弱类型语言**

忽略定义数据类型，一个变量可以同时存储不同类型的数据

**无类型语言**

window.onload是静态资源都加载完成之后

ready是

javascript:

ECMAscript

DOM

BOM

# 词法结构

**JavaScript是区分大小写的语言**

比如，关键字‘while’必须写成‘while’，而不能写成‘While’或者‘WHILE’.

比如，‘online’,‘Online’,‘OnLine’是3个不同的变量名。

**HTML并不区分大小写**

# 代码的解析

解析顺序：从上到下。

1.JavaScript是用unicode编写的。

2.JavaScript是一门区分大小写的语言。

3.JavaScript的转义符为 “\”

4.JS的注释为 “ // ” 或者 /\* \*/

5.JS的

# JS的流程

## JS引用外联样式

<script **src**="js/bootstrap.js"></script>

<script type= “text/JavaScript>中的type= “text/JavaScript是告诉浏览器要解析成JS代码，没有也行。

## JS注释

要养成写注释的习惯

//:单行注释

/\* 多行注释的内容 \*/

## JS的背景

script标签是专门用来解析js代码

一般写在**body结束标签的上面**,就超出body也就被解析到body中。

js -> javascript 发明出来的时候主要用来页面也用户进行交互的

**ECMAScript 国际标准** es6 es7

咱们当下学习的内容都是5.0左右的

## 获取元素

### 静态方法document.getElementById('id')

通过id的方式去获取元素。

<script>

var oBtn = document.getElementById('btn')

document.body.innerHTML = '<input id="btn" type="button" value="按钮"/> <input type="button" value="按钮"/> <input type="button" value="按钮"/>'

console.log(oBtn); //找不到，打印“null”

</script>

括号里面的id名可以用双引号（""）或者单引号（''）去包id

点击事件 -> onclick

鼠标移入事件 -> onmouseover

.....

“.” 类似于 的

### 动态方法.getElementsByClassName

通过属性名获取

**动态方法：是指随时根据页面的元素的变化，随时动态地获取。**

### 动态方法.getElementsByTagName

通过标签名获取

**动态方法：是指随时根据页面的元素的变化，随时动态地获取。**

<script>

var oBtn = document.getElementsByTagName('input')

document.body.innerHTML = '<input type="button" value="按钮"/> <input type="button" value="按钮"/> <input type="button" value="按钮"/>'

alert(oBtn.length);

</script>

获取标签名(一组，多个): 类数组（具有数组的一些特性(比如有length属性，可以使用下标)，但是它不是真正的数组，因为用不了数组的方法）。

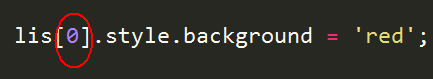
注意：

**就算页面中只有一个标签，如果要使用，也必须加下标**

要么在获取元素的时候加，



要么在使用的时候加



个人建议在获取的时候加

如果像下面这样不加下标就是错的！！！！

let lis = document.getElementsByTagName('li');

lis.style.background = 'red';

document.getElementById和getElementsByTagName的区别

1. **getElementById 前面只能放document**

**getElementsByTagName 前面能放document，还能放其他变量。**

代码块缓存，可以提高性能！



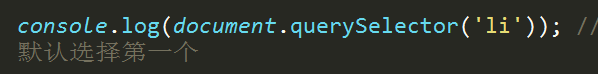
1. getElementById 不能找到一类元素

getElementsByTagName 可以找到一类元素

### querySelector

这个和querySelectorAll的区别就是**当用来获取一个元素的时候，不用加下标。**

querySelector() 用css选择器去获取一个元素



如果我要选择第二个呢？？？

用document. querySelectorAl(“li”)[1] 去取。

除了这种办法：？？？？？

let lis = document.getElementsByTagName('li');

lis[0].style.background = 'red';

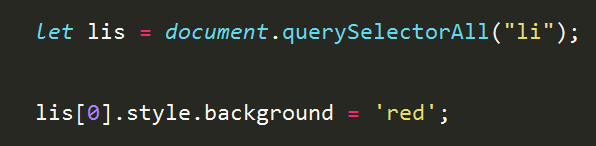
lis[1].style.background = 'red';

lis[2].style.background = 'red';

lis[3].style.background = 'red';

lis[4].style.background = 'red';

### querySelectorAll



let lis = document.querySelectorAll("li");

querySelectorAll() 用css选择器去获取\*一组\*元素

例如使用属性选择器



**当用来获取一个元素的时候，必须加下标”[]”**

getElementsByTagName通过标签名称类获取元素，

同时querySelectorAll和querySelector也可以通过标签名来获取元素！

## classList选择器

### 语法

element.classList

### 方法

|  |  |
| --- | --- |
| **方法** | **描述** |
| add(*class1, class2, ...*) | 在元素中添加一个或多个类名。  如果指定的类名已存在，则不会添加 |
| contains(*class*) | 返回布尔值，判断指定的类名是否存在。  可能值：   * true - 元素包已经包含了该类名 * false - 元素中不存在该类名 |
| item(*index*) | 返回类名在元素中的索引值。索引值从 0 开始。  如果索引值在区间范围外则返回 *null* |
| remove(*class1, class2, ...*) | 移除元素中一个或多个类名。  **注意：** 移除不存在的类名，不会报错。 |
| toggle(*class*) | 在元素中切换类名。 第一个参数为要在元素中移除的类名，并返回 false。  如果该类名不存在则会在元素中添加类名，并返回 true。 |

### toggle

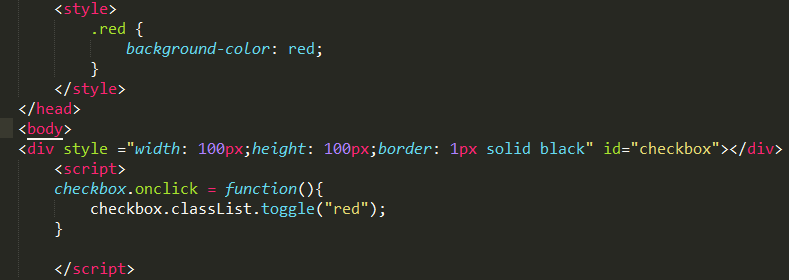
**反转CSS类的有无**

myDiv.classList.toggle('myCssClass'); //现在是增加

myDiv.classList.toggle('myCssClass'); //现在是删除

这个方法的作用就是，当myDiv元素上没有这个CSS类时，它就**新增**这个CSS类；

如果myDiv元素已经有了这个CSS类，它就是**删除**它。就是反转操作。



方法用于绑定两个或多个事件处理器函数，以响应被选元素的**轮流的click 事件**。

具体的说：

toggle() 方法切换元素的可见状态。

如果被选元素可见，则隐藏这些元素，如果被选元素隐藏，则显示这些元素。

**向toggle绑定一个参数的时候**

当指定元素被点击时，在绑定的参数和’空”之间切换.

例如：

ii.onclick = function(){

ii.classList.toggle('checked');

div.classList.toggle('hov');

}

**向 Toggle 事件绑定两个或更多函数**

当指定元素被点击时，在两个或多个函数之间轮流切换。

如果规定了两个以上的函数，则 toggle() 方法将切换所有函数。例如，如果存在三个函数，则第一次点击将调用第一个函数，第二次点击调用第二个函数，第三次点击调用第三个函数。第四次点击再次调用第一个函数，以此类推。

**语法**

$(*selector*).toggle(*function1()*,*function2()*,*functionN()*,...)

### remove清除类选择器的样式

元素名称.**classList**.**remove('**active**')**;

注意：这里一定要用classList.remove!!!

后面**用括号包起来**要清除的类名称，**类名称要加引号**

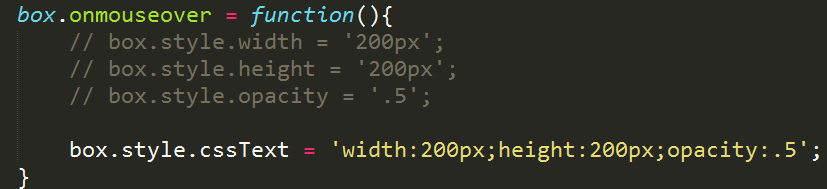
### add 添加类选择器的样式

元素名称.class**L**ist.**add**('Pu')

## 批量设置行间样式cssText

1.**只针对行间样式**style.cssText，不能修改外联样式。

2.能够对行间样式进行**批量设置**



3.多个style.cssText修改同一元素时，后面的cssText会全部覆盖前面cssText的所有设置。但是不会修改外联样式。

## a标签

在JS中，要让a标签不跳转，例如：

<a href="javascript:;" >缩小</a>

## 声明和赋值

1.声明和赋值是两码事。

var a; //这里是一个声明，只是一个声明,并没有把undifined**赋值**给a。

**只是预解析**var a= undifined,**并不是**把undifined**赋值**给a

2.声明：

一．**var i =0 , j = 0;**

//这里相当于

var i = 0;

var j = 0;

中间用“,”就可以var一次声明多个变量。

中间用“;”var一次声明一个变量。

二．var a = b = 5

在没有声明一个变量时，变量的THIS是什么？window?

例如1：

var a = b = 5; //相当于 var a = **window.b** = 5

这里的**b的主就是window**

例如2：

(function(){

var a = 5;

b = 666;// window.b = 666;

var c = 10;

})();

alert(b); //666

## 变量

#### 概念

可变的量,存储数据的 “容器” 以复用为主

var 变量的名字 = 要存储的数据

let 官方更推荐使用 let

#### 变量使用注意

**1.变量如果没有声明，访问变量就会出错，报错！**

2.变量名开头不能是数字,特殊的符号

3.也不能有关键字或者保留字

关键字:系统已经使用的名字 (undefined)

保留字:系统现在没有使用，但是将来会使用的名字 (class)

4.变量一般都是小驼峰命名法。首字母小写，单词与单词之间首字母大写。

例如：get Element By Id

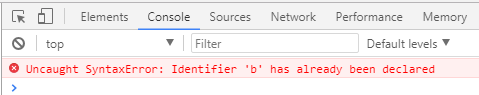
#### 变量的调用

**变量不能被调用**

#### var与let的区别

##### let声明不能重复，var可以重复

如果使用let，那么同名的变量不能重复！！！



如果使用var，那么同名的变量可以重复.

但是后面声明的同名变量会覆盖前面声明的同名变量

##### 未声明变量的处理

浏览器对var未声明的变量，解释为undifined

console.log(miao); //打印undifined

var miao = 123;

浏览器对let未声明的变量，会报错！

console.log(miao); // **Uncaught ReferenceError: miao is not defined**

let miao = 123;

##### 块级作用域

因为，let 是一个**块的作用**关键字。

也就是说let声明的变量**仅仅作用在所在的作用域内**。

下面的let会覆盖上面的let声明的值，如果是用let在全局声明一个变量，下面再在局部声明一个变量，那么下面局部声明的就会覆盖上面用let全局声明的。

let声明的循环：

for ( let = 0; i <arr.length; i++) {

}

因为var 不是一个块的作用关键字，是一个**全局作用**关键字。

也就是说var声明的变量能作用在全局。

var声明的循环：

**var i = 0**;

for ( ; i < arr.length ; ) {

}

i++;

var i 没有块级作用域

btn[i].onclick = function(){alert(i) }; //i是for循环的结果

let i 可以有块级作用域

btn[i].onclick = function(){alert(i); }//i就是本次循环的索引值

## 常量

不可变的量 const

作用： 如果使用const声明，改变了值就报错！！！

# 属性操作

**1.行间样式是元素的一个属性**

**2.class也是元素的一个属性**

**3.另外还能自定义添加元素的属性**

**任何属性的值都是要加“” （引号）的！！！！！**

## window

window 是一个对象，所有的js到在window中，**所以window是JS中的顶层**

任何一个函数的**直接调用**都是由window调用的。

<script>

function fn1(){

alert(this);

}

window.fn1();

</script>

如果这个直接调用函数，是发生在事件调用中，this也指向window

<button></button>

<script>

let btn = document.getElementsByTagName('button')[0]

function fn1(){

alert(this);

}

btn.onclick =()=>{

fn1(); //在这里执行this也是指向window

}

</script

## 属性

**对象如果没有的属性为undefined**

**当对象为多个元素的时候，那么它的属性为undefined**

**也就是说，属性只能针对某“一个”元素**

**类数组是没有属性的！**

## 属性的读操作

1. obj.attr  **使用小圆点**
2. obj[attr]  **使用中括号**

### 获取行间样式

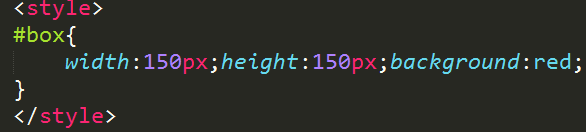
console.log(**box.style.width**)

box.style只能获取标签中行间的样式，非行间的（样式表）是获取不到的。

行间样式：



非行间样式：



### 获取样式表/当前样式

console.log(**getComputedStyle(div).height**)

getComputedStyle(obj).width : **计算后**的样式。

注意：

1、返回值是**带单位的字符串**(带单位就不能直接数学运算)。

2、获取CSS3的内容一般获取不准确

3、如果获取transform的时候，获取到的是一个矩阵。

这个getComputedStyle(div).height 和Style有什么区别？？？

getComputedStyle与style都是获取元素的属性值，

都是带单位的。与DOM操作中的属性不同！

### getComputedStyle与style的区别

我们使用element.style也可以获取元素的CSS样式声明对象，但是其与getComputedStyle方法还有有一些差异的。

1. **只读与可写**  
   正如上面提到的getComputedStyle方法是只读的，只能获取样式，不能设置；而element.style能读能写，能屈能伸。
2. **获取的对象范围**  
   getComputedStyle方法获取的是最终应用在元素上的所有CSS属性对象（即使没有CSS代码，也会把默认的祖宗八代都显示出来）；

而element.style只能获取元素style属性中的CSS样式。

因此对于一个光秃秃的元素<p>，getComputedStyle方法返回对象中length属性值（如果有）就是190+(据我测试FF:192, IE9:195, Chrome:253, 不同环境结果可能有差异), 而element.style就是0。

### 不能拿来做判断的样式

1.绝对路径

2.元素的颜色

3.innerHTML

## 属性的写操作

### 修改元素行间样式

**1. 一切CSS属性都是“.style”的属性**

**2.设置样式一般要加引号**，例如

box.style.width = '200px';

注意：这里修改是修改元素的行间样式，而**不是**改元素的**外联样式**。

### 修改元素外联样式

**一般**属性名怎么写，js操作就怎么写，**但是class为保留字**

要使用className

**className**

属性名

**tagName**

标签名

### 添加自定义属性

给元素添加的自定义属性，是在html结构中看不到的。

给元素添加的自定义属性，是属于这个元素的，所以要加” **this.** ”!!!

### 添加自定义索引

lis[i].num = i;

var arr = ['mm','tt','mimi']

for (var i=0;i<oBtn.length;i++){

oBtn[i].index = i;

oBtn[i].onclick = function(){

alert(arr[this.index]);

}

}

**作用：**

操作一个元素，控制另一个元素，用索引

当操作某件事的时候，还要对应另外一件事，就用索引

## [] 中括号的用法

### 1.变量赋值

**凡是能用.的地方都能用[];**

“[]”中**可以放变量** “.”后面**不可以放变量**

所以，能用“[]”的地方不一定能用“.”

比如:

box.style[kv] = vv;

换成

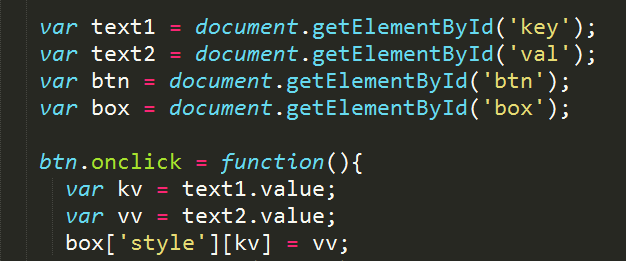
box.style.kv = vv;

style下面没有kv的属性，所以不能用.kv

注意：

在中括号中只要**不是字符串**，**就是变量**

**中括号应用：变量**



### 2.数组

['miao',1,true]

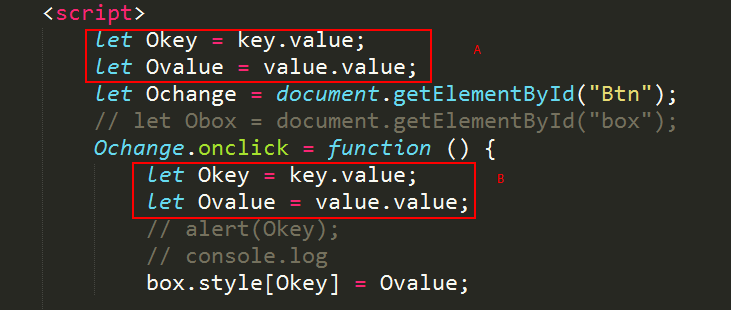
### 3.(类)数组的下标

[1,2,3,4,5][3] -> 4

## input中的value

一般操作（读和写）input的内容，使用value

## 逻辑性



A语句是在一打开页面的时候将值获取

B语句是在按了按钮之后将值获取

注意：

如果获取text1.value或者text2.value时，还没有点击，

那么会获取最开始的那个值。（不要在还没被事件触发时获取值，不然获取的值会不准确）

正确方式：

点击的时候再去获取input中的值

## 赋值为空

t1.value = ' '; //设置t1的value为空

## 字符串拼接

要么就用第一种，要么就用第二种，不可以混用！

### 第一种写法

**需要写加号**

字符串 + 任意数据 = 字符串拼接

如果字符串要加**变量**，就要**字符串拼接**

**格式为：**

****

**‘ + +’**

**例如：**

****

### 另一种写法：超级字符串（ES6）

**不需要写加号**

var 变量 = 值

**`${变量}`**

两边要加模板字符串

**``**



## 复合样式的设置

**驼峰式，不能加中线-**

background-color -> backgroundColor

如果非得background-color写，这个时候需要用中括号包一下

font-size -> fontSize

# 面向过程

需要先写思路

# 判断语句

所有括号里面的条件会被转成一个布尔值。

## if(条件){}

## if(条件)

if为什么可以单用？

因为根据需要，随时可以单用。

可以多次使用，不用else if

## if(){}else if(){}

else if 中必须带条件

else if 可以有无数个

## if(){}else if(){}else{}

if（条件1）{

代码块;

}

else if （条件2）{

代码块;

}

else if （条件2）{

代码块;

}

else if （条件3）{

代码块;

}

else if （条件4）{

代码块;

}

...

...

else{ **////注意这里不能指定条件**

代码块;

}

else if 中必须带条件

else if 可以有无数个

从上到下判断条件，只要找到一个满足条件的判断，就会执行对应的代码，其他不混再去理会。

所有条件都不满足，执行else中的代码

else中的**执行代码可以不写**，但是如果所有的条件都不满足，那么所有的代码都不会执行。

### if(){}else{}

这里else的条件是除了if成立的所有条件。

else{ **////注意这里不能指定条件**

代码块;

}

## 不能作为判断条件的3个值

1.读a标签的href是**绝对路径**，所以**不能**拿href去**进行判断**

2.读img标签的src是绝对路径，所以不能拿src去进行判断

3.获取background的时候，在不同的浏览器下显示是不同的，所以不能拿来进行判断。

## switch case defult break

var a = prompt(“请输入今天的星期对应的数字”);

switch (a) {

case “1”**:** //**case后面变量要加 “：”！！！**

alert (“您输入的星期一”);

**break**; //每一种情况一定要有中断！

case “2”**:**

alert (“您输入的星期二”);

break;

case “3”**:**

alert (“您输入的星期三”);

break;

case “4”**:**

alert (“您输入的星期四”);

break;

case “5”**:**

alert (“您输入的星期五”);

break;

case “6”:

alert (“您输入的星期六”);

break;

case “7”:

alert (“您输入的星期日”);

break;

**default : //可以写可以不写，如果不写程序直接结束**。

alert(“你输入的什么鬼？”);

break;

}

alert (“over”);

# try...catch语句

**try...catch**语句将能引发错误的代码放在try块中，并且对应一个响应，然后有异常被抛出。

**语法**

try {

try\_statements

}

[catch (exception\_var\_1 if condition\_1) { // non-standard

catch\_statements\_1

}]

...

[catch (exception\_var\_2) {

catch\_statements\_2

}]

[finally {

finally\_statements

}]

**try\_statements**

需要被执行的语句。如果有错误就会执行catch的代码

这里可以放**throw new** ...

throw new是抛出的意思，抛出后面跟什么，就把它作为实参，传给catch的形参**exception\_var\_1/ exception\_var\_2**

**catch\_statements\_1**, **catch\_statements\_2**

如果在try块里有异常被抛出时执行的语句。

**exception\_var\_1, exception\_var\_2**

用于保存关联catch子句的异常对象的标识符。**也就是try那里没有执行的错误信息**！

**condition\_1**

一个条件表达式。

**finally\_statements**

在try语句块之后执行的语句块。无论是否有异常抛出或捕获这些语句都将执行。

# ECMAScript数据类型

## 为什么要有数据类型？

在数据类型运算的时候，

因为数据类型的不同，

运算的结果也能够不一样。

每个数据类型的功能不一样，它们在js的意义也会不一样。

有些时候，要的就是结果不一样。

## 复合数据类型和简单数据类型的区别？

1.基本类型的值是不可变的；复合类型的值是可变的。

2.基本类型的比较是值的比较；复合类型的比较是引用（地址）的比较。

3.基本类型的变量是存放在栈区的（栈区指内存里的栈内存）；复合类型的值是同时保存在栈内存和堆内存中的对象：

**复合数据类型：**

**栈区内存保存**变量标识符和指向堆内存中该对象的指针，

**堆内存保存**对象的地址。

**简单数据类型：**

**栈区内存保存**变量的标识符和变量的值。

**堆内存保存**nothing.

## typeof的数据类型

1.typeof检查出来的值都是字符串并且是**\*\*小写\*\***。

2.比ECMAScript**多一个function少一个null**

string: box.value

number: number,NaN(注意NaN也算是number里面的)

object, {},[],null //typeof(数组)也是object,没有Array

undefined,

boolean,

symbol,

function

## 对象数据类型 object

对象数据类型：

function函数

global object全局对象

array数组

对象是属性的集合，每个属性都是由“名/值对”组成

## 原始类型(基本类型)

Number（数字）、Boolean（布尔）

Null（空对象）、Undefined（未定义）

Symbel（唯一）

### 1. Number（数字）

**数字类型包括”数字”和”NaN”**

#### 1.1 number类型

1.JS不区分整数值和浮点数值

#### 1.2 NaN 类型

NaN -> 非法的**数学运算！！！！**

**除了NaN，所有的数据自己都等于自己**

NaN比较特殊，特殊到自己不认识自己。

例如：

var a = NaN;

var b = NaN;

var c = NaN===NaN;

console.log(c);//false

如何检测数据是否为NaN：

**1. isNaN(X)**

**如果X是NaN就是true，否则false**

**isNaN(NaN) 这样就是true isNaN(其他) 这样就是false**

**!isNaN(num) 这样就是true !isNaN(其他) 这样就是false**

**! isNaN 可以理解为：不是非法的 ，也就是说是合法的。**

**2. a === a 通过数据自己等于自己，来判断是不是NaN.**

**当数据自己等于自己的时候，那么它就不是NaN。**

**当数据自己不等于自己的时候，那么他就是NaN。**

**isNaN() 内部是依靠一个Number()的方法来判断的，可以将字符串形式的数字转换为数字。**

<script>

alert(isNaN('250')); //**false**

</script>

如果 x 是特殊的非数字值 NaN（或者能被转换为这样的值），返回的值就是 true。

如果 x 是其他值,则返回 false。

例如：

document.write(isNaN(-1.23)+ "<br>"); **// false**

document.write(isNaN(5-2)+ "<br>"); **// false**

document.write(isNaN(0)+ "<br>"); **// false**

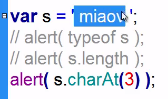
document.write(isNaN("Hello")+ "<br>"); **//true**

document.write(isNaN("2005/12/12")+ "<br>"); **//true**

### 2. String（字符串）

空格也是字符串，占一个字符串长度

某一个字符串的第几个字符串（从0开始）是charAt



页面中的所有内容都是字符串！！！

简写：

单引号或者双引号对就叫字符串

标准写法:

new String

但是直接调用系统内置的函数，性能不高

**字符串必须打引号，否则就会被浏览器识别为没有定义的变量，无论在哪里！**

每个字符通常来自于Unicode字符集。

### 3. 布尔类型

true\false

**真:非空字符串, 非0的数字, true, 能找到的元素ele，[],{},,**

**Symbol, function函数**

**假:空字符串 , 0 , false, null ,不能找到的元素ele**

**undefined, NaN**

**空对象≠空白对象**

空对象：null

空白对象：**[],{},fn**

boolean本质上是0（false）和1（true）

var a = true;

var b = 1;

var c = a == b;

console.log(c);//ture

**真:非空字符串,**

“” 空字符串是假的

“ “ 空格是真的

#### boolean 在内存中的存储格式

所有数据都存储在内存中，boolean在存储的时候有个特点，会将true转成1，会将false转成0；所以在内存中true是用1来表示的，而false是用0来表示的。

### 4. Undefined（未定义）

#### 什么情况出现undefined ?

**1.声明变量没有赋值.**

例如：

let s;

console.log(s);

当一个变量有名，但是没值的时候，默认值就是undefined

**2.当对象没有这个属性的时候（简单类型身上没有这个属性时）**

var b= {name :”小黑”}

console.log (b.index) / / undifined

console.log(b[“name”] / / 这样写是正确的

console.log(b[name]) // 这里的name不是一个变量所以不能不打引号

console.log (this.name) //这里的this是指window.

**3.当找不到数组中的值时**

**4.函数默认返回值**

**5.函数没有传实参，去访问形参的时候。**

**6.预解析的时候var的上面会出现undifined**

### 5. 空(Null):

**什么时候出现null?**

1. **找不到元素的时候,页面上没有这个元素的时候。**
2. **script放在元素的上面的时候并且没有加window.onload**

（正常情况下script是放在元素结构的下面的，这里是说放在结构上面的时候）

1. **let n= null**

null和‘’ （空字符串）有什么区别？？？

null即为“空”

### 6. 唯一symbol

唯一

# 复杂（对象）数据类型不能运算

# 原始数据类型的运算

**只有原始数据类型能进行运算，复杂（对象）数据类型不能进行运算**

## 赋值运算符

=、+=、-=、\*=、/=、%=

= 赋值

从右往左赋值

+=、-=、\*=、/=、%=

把符号两边的值进行运算，最终运算结果都赋值给左边

赋值运算符

a = b = c = 1

从右到左进行赋值

## 关系运算符

**=**

赋值

**==**

先左边后右边

会将两边的**数据类型统一,转成数字类型**，然后再比较

**（也就是说只比较绝对值）**

不用关心数据类型是否相同，直接比较

**===**（全等）:

不会去转换两边的数据类型，如果数据类型都不相等直接false

**（也就是说，先比较判断数据类型，再比较判断数值）**

> 、 < 、>= 、 <= 比较符号，与数学运算完全相同

‘two’ > ‘three’ //=> true ‘tw’在字母表中的索引大于’th’

!:

非，否，取反，默认强行转成布尔值

!=：

不等

!==

不全等（与===对应）

！= 不等于

不关心类型，比较的是内容，如果内容一致，返回false。如果内容不一致，返回ture。

！== 不等等于

即会关心内容，也会关心类型

只要是内容或者类型不一致，就会返回true。

内容和类型一致，返回false。

## 逗号运算符

作用：可以同时声明几个值。

var a , b , c;

a = b = c =1;

console.log(a)

## 逻辑运算符

| ： 会检查每一个 条件的真伪，再做“或”运算 (光棍的不怕死)  
||： 按照条件写的顺序，直到一个为true时，后面的条件则不再检查，直接进入条件   
& ： 会检查每一个 条件的真伪，再做“与”运算  (光棍的不怕死)  
&&： 按照条件写的顺序，直到一个为false时，后面的条件则不再检查，直接跳出

**&& 与**（谁1与谁2）

布尔值 && 布尔值

谁1成立 还会去看后面 谁2

如果谁1成立，谁2不成立，**返回谁2**

如果谁1不成立,直接返回谁1

80>pj >70 在代码中表达成

80>pj &&pj>70

**|| 或**

布尔值 || 布尔值

谁1 || 谁2

如果谁1成立就不会看谁2了，**返回谁1**

如果谁1不成立，还会看谁2，**返回谁2**

**( )** 有优先级

### 三元运算符（三目运算符）：

**“? :”**

非它即它

let a = 0 < 1 ? '呵呵' : '嘻嘻';

### 二元运算符:

a + b

### 一元运算符:

typeof a

## 逻辑运算符的优先顺序

! > && >||

很少用到这个优先顺序

## 自增自减

前自增 ++ num 先自增，后运算

前自减 -- num 先自减，后运算

后自增 num ++ 先运算，后自增

后自减 num-- 先运算，后自减

let num = 10;

console.log (num++ == num ++) 相当于 10 与 11比较

console.log (++num == ++num) 相当于11与12比较

**从左到右，从上到下**

## 运算顺序

**从左到右，从上到下**

## 执行顺序

**先执行括号里，再执行括号外**

## 取余

%

## 相加赋值

a = a + b

相当于 a += b //**注意 “+=” 这个符号千万中间不能加空格，否则就是非法运算符！！！**

## 运算的优先级

加“（）”的运算优先级就比较高！！！

!== 不全等

=== 全等

先乘除，后加减

## 数学方法Math

JS支持复杂的算术运算，这些复杂运算通过作为Math对象的属性定义的函数和常量来实现。

带Math的**返回值都为number**.

注意Math是对象，对象的首字母是大写的！

Math对象的实参是一个数字或者由逗号隔开的多个数字。

### 向下取整

**M**ath.floor ( ) 向下取整

### 向上取整

**M**ath.ceil () 向上取整

### 四舍五入

Math.round() 四舍五入

### 取绝对值

Math.abs()

### 取几个数字中的最大值

Math.max()

let a = Math.max(1,2,3);

console.log(a); //打印3

### 取几个数字中的最小值

Math.min()

### 取随机数

Math.random();

随机 0 - 1之间的无限不循环小数（伪随机数）

常用的取整公式

x ~ y : Math.round(Math.random()\*(y-x) + x)

0 ~ x : Math.round(Math.random()\*x)

1 ~ x : Math.ceil(Math.random()\*x)||1

0 ~ x-1 : Math.floor(Math.random()\*x)

### 幂运算

Math.pow()

let a = Math.pow(2,3);

console.log(a); //打印8。也就是2的3次幂。

**保留小数**

.toFixed(num) 保留num位小数,返回值是个string

例如：

3.1415926.toFixed(2)

**转换进制**

toString可以转除了10进制的进制数

变量.toString()

不能用 数字.toString() 会报错的。

**找到数组中最大的值**

let arr = [1,2,3]

let arr = [1,2,3];

console.log(Math.max.apply('',arr));

console.log(Math.max.apply(arr,arr));

为什么要用apply方法而不用call方法？

因为要把这个方法用到一个数组身上，所以用apply,不能用call。

call的第二个参数是

为什么不能console.log(Math.max(arr));

因为Math.max()是针对原始数字类型的函数方法。

## 数字的二进制浮点数

现代大多数语言都使用浮点数表示法，这是一种二进制浮点数表示法。它有一个问题如下：

let a = .3;

let b = .2;

console.log(a-b); // 0.09999999999999998

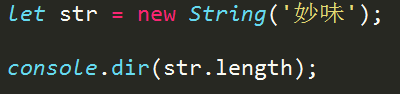
var c = .2 - .1;

console.log(c); //0.1

这种计算结果可以胜任大多数计算，这个问题只有在比较两个值是否相等的时候才会出现。

# 字符串长度

如何查询长度？



（注意：console.dir() 是指console.log()能实现的它都能实现，并且这个console.dir()能够看到更加详细的内容。）

字符串长了，就要加括号。

let val = 12345;

（parseInt(val)+””）.length

length 长度

页面中的所有内容都是字符串！！！

注意：

字符串的length只能读不能写



第0个是“1”！！！！

# 原始数据类型的转换

数据类型转换包含两种方式：

1.强制类型转换

强行转换（主动人为去转换）

2.隐式类型转换

系统偷偷的帮你转换的（非人为主动的转换）

## 字符串转数字

### Number();

number转空字符串为0

如果Number转不出来，比如：'20px'；那么同样会给个结果NaN

### parseInt(a,b)

1. 如果在内容中出现小数，那么小数不会保留，变成整数。

**向下取整**。

1. 在这里面不是四舍五入，而是去掉小数。
2. 如果第一个符号是数字，则继续解析直至遇到一个非数字符号为止。

**parseInt(a,b)**

**其中a表示要转的字符串**

**其中b表示要转的进制数**

parseInt(

作用1.字符串转整数

**作用2.第一个参数的进制数)**

比number多了一个取整功能

开头必须为数字，不然转NaN, **也就是说如果是字符串，那么就转成NaN** (空格也是字符串)

如果开头为0或空格，那么会舍弃0或空格

如果开头为数字中间不为数字了，那么就会把开头的数字转成数字返回。

如果添加第二个参数：

将第一个参数转成十进制

例如：

console.log(parseInt('110',2));

运行结果为：6 即：2^0+2^1+2^2=6

### parseFloat(a)

(字符串转小数)

相比parseInt可以转小数，不过没有第二个参数

例如：

console.log(parseFloat（’12a83’）) //12

## 数字转字符串

### String(a)

例如：var a = 123;

a = String(a );

console.log (a);

console.log (typeof a);

## 隐式转换

隐式类型转换：

系统偷偷的帮你转换的（非人为主动的转换）

运算符都会隐式类型转换

一.  **+ （加号）**

1.1如果字符串数字前面没有别的数据，那么转成number

1.2数字 + 数字 = 数字

2.**字符串 + 别的数据类型都是字符串**

总结：

字符串加数字,数字就会转成字符串。

二. **- （减号）\* （乘号）/（除号） %（取余）**

默认转成number类型。

如果运算不出来，就NaN。

总结：

数字减字符串，字符串转成数字。如果字符串不是纯数字就会转成NaN。字符串减数字也一样。两个字符串相减也先转成数字。

乘，除，大于，小于跟减的转换也是一样。

三. **alert()**

只要调用alert就会隐式转成string

 四.**再来看看一组 == 的。**

1.undefined等于null

2.字符串和数字比较时，字符串转数字

3.数字为布尔比较时，布尔转数字

4.字符串和布尔比较时，两者转数字

undefined == null; //true

'0' == 0; 　　 //true,字符串转数字

0 == false; //true,布尔转数字

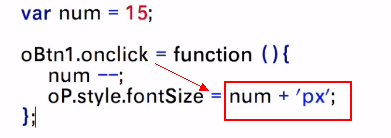
'0' == false; 　　　//true,两者转数字

null == false; 　 //false

undefined == false; 　//false

## 隐式转换

举例：

？

为什么没有给num加引号？？

因为num是一个变量。

### 隐式转换成字符串

typeof 一元运算符 专门用来检测数据类型的

**任何的数据类型 + 字符串 = 字符串**

转换成字符串：**数字 + 空字符串 = 字符串**

var a = 123;

a = a + “”; //**数字 + 空字符串 = 字符串**

console.log (a);//123

console.log (typeof a);//string

### 转换成布尔类型

**1.通过隐式转换**

var a = 123;

a =！ ！a;//===>a = ! ! Number (123)===>! false

console.log (a);//true

console.log (typeof a);//boolean

**2.强制转换**

Boolean()

把字符串类型转换为布尔类型

例如：var a = “true”;

a = Boolean(a );

console.log (a);//true

console.log (typeof a);// boolean

除了false、0、””、NaN、Undefined 在转换的时候会转成false以外，其他的都会转成true(),例如“false”

# 数组

## 一维数组

[]

let arr = [num,str,b]; //数组 简写

let arr2 = new Array('你'); //标准写法

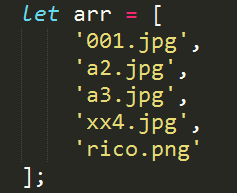
**注意： js计数是从0开始的**

[ ] 有两种含义，一种表示为数组，一种表示为下标

console.log(arr[2]); //这个[]代表下标

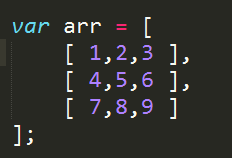
console.log(arr[arr.length-1]); //这表示是arr数组中的最后一个元素

多个数组的写法：



**注意：**最后一个元素后面不能有“，”（逗号）。否则就认为在逗号后面还有一个元素。

## 二维数组

****

遍历二维数组，就需要用for套for

## 给数组添加元素push

数组名.push (要添加的元素)

数组添加元素是默认在最后面添加元素？是的

如何在指定位置给数组添加元素？？？？

使用方法splice

# 类数组

**1.document.getElementsByTagName**

**2.document.querySelectorAll**

获取标签名(一组，多个): 类数组（具有数组的一些特性(比如有length属性，可以使用下标)，但是它不是真正的数组，

**因为用不了数组的方法**）。

例如：push 给数组添加元素。

其他的方法呢？

直接给数组的[i.index](索引值) 用 “=” 赋值。

## 如何将一个类数组转换成数组？

## 1. slice

最经典的方法，使用Array的slice方法，此方法如果不传参数的话会返回原数组的一个拷贝，因此可以用此方法转换类数组到数组；

var arr = Array.prototype.slice.call(arguments);

//等同于

var arr = [].slice.call(arguments)

## 2. Array.from()

是ES6中的方法，用于将类数组转换为数组。

var arr = Array.from(arguments);

只要有length属性的对象，都可以应用此方法转换成数组。

## 3. 扩展运算符

ES6中的扩展运算符...也能将某些数据结构转换成数组，这种数据结构必须有遍历器接口。

var args = [...arguments];

例如：

var list = [...'妙味课堂'];

这个将“妙味课堂”四个字符串转换成[‘妙’，‘味’，‘课’，‘堂’]

# 循环

## Switch

switch(变量){

case 是1 :

执行想要的效果;

break;

case 是2 :

执行想要的效果2;

break;

default:

执行都不是的效果

break;

}

default: 默认的条件 类似于 else

注意：

如果不加break那么会穿透,不但某个条件成立会执行，还会执行下面的代码。

## while循环/for循环

**while循环和for循环本质上都是一样的只是格式不同。**

<script>

var str = '31731638244927494902480242034'

var findTarget = '4';

var i = 0;

// for( ;str.indexOf(findTarget,i) != -1;){ //for带分号

// alert (str.indexOf(findTarget,i));

// i = str.indexOf(findTarget,i) + findTarget.length

// }

while( str.indexOf(findTarget,i) != -1){ //while不带分号

alert (str.indexOf(findTarget,i));

i = str.indexOf(findTarget,i) + findTarget.length

}

</script>

### while(条件){}

使用循环的时候一定要加上结束条件，要不然会永远都不会结束。

适合处理不知道有多少次循环的场景

<script>

var i = 0;

while (i<3){

alert(i);

i++;

}

</script>

### for循环

**一旦被for循环包住的东西就会重复执行！！！！**

**for(** 初始化变量;

判断循环终止条件;

变量的增加或减少 **)**

**{**执行代码**}**

循环

1.初始化变量（在整个循环中只会走一次）

2.判断(只要判断条件成立，就会循环,**不成立就不循环**)

**不成立才会跳出循环！！！！！**

**否则继续循环！**

**判断的条件如果有n多个，那么就按最后一个来判断。**

例如：

for(var i = 0,j =0 ,h =0;i<6,j<7,h<10;i++,j++,h++){

console.log(i) //打印9 ,**按最后一个判断条件来判断**

}

3.执行代码

4.变量的增加或者减少（最后一步）

使用if语句的逻辑理解为：

例一：

**var i = 0**; //初始化只会走一次

注意，这里一定要**声明变量**！

for(;**i<5**;){

**alert(i);**

**lis[i].style.background = 'red';**

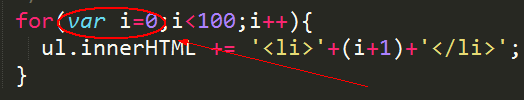
**i++;**

**alert(i);**

}

alert(i);

循环的遍历应用：



注意，这里一定要**声明变量**！

例二：

<script>

var str = '以去香港开户；2、可以通过互联网方式开户。但要说明的是，第二种方式目前还没'

var findTarget = '二种';

var i = 0; //初始化变量

for( ;str.indexOf(findTarget,i) != -1;){ //循环的终止条件

alert (str.indexOf(findTarget,i)); //执行代码

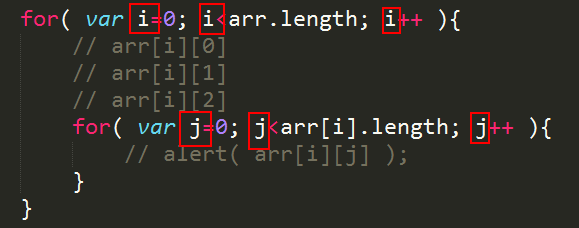
i = str.indexOf(findTarget,i) + findTarget.length

//变量的增加或者减少

}

</script>

#### for套for



需要有两个不同的变量



如果是两个并列的for就不用

**注意：**

单纯的for套for，变量名一定不能是一样的！！！

单纯

for(){

for(){}

}

不单纯的for套for，变量名可以一样（因为不在一个作用域中），但是不建议一样。

不单纯

for(){

li[i].onclick = function(){

for(){

}

}

}

object对象需要有特殊的结构才能使用for循环

**for套定时器**

循环几次定时器走几次

# 控制流程

## continue和break

continue是跳过

break是跳出,是终止**for循环**的

## return

函数return什么，执行之后就是什么

1.函数名+（） ==> return后面的值

<script>

alert (fn1());

function fn1 (){

return "123"

}

</script>

2.所有函数的返回值都是undifined

是终止**函数**后面的，后面的代码都不会执行了；

<script>

alert(fn1());

function fn1 (){

return '123';

alert (520)  **//这里永远都不会执行了**

}

</script>

必须是**有函数的情况下才能用return,**没有函数用return就会报错。

**无论何时，return后面要跟（）括号！！！！！**

return函数的执行

<script>

fn1()(); **//使用双括号**

function fn1 (){

return function (){

alert(123)

}

}

</script>

# window

是一个对象，不是块

变量 + 主

=属性

函数 + 主

方法

函数的this是什么？

是window

例如：

fn1(); //**这就相当于window.fn1();**

变量的THIS是什么？

是window

var a = b = 5; //相当于 var a = **window.b** = 5

这里的**b的主就是window**

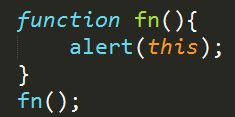
classlist.add

# This

## 1.默认情况

当this在浏览器中单独存在时，在浏览器中默认的this是window

例如：



网页弹出 [object Window]

执行一个函数fn(); 相当于window.fn();所以this 指向window

## 2.函数和属性

当this在一个变量中时，this指的是对象object。

属性：obj.a box.style btn.value 前面有主

变量：a 前面没主

例如：

let obj = {

name:'小明',

fn:function(){

//alert(this.name);

console.log(this); //这个this是指**obj**

}

}

**obj**.fn();

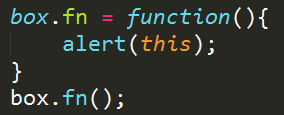
## 3.方法和函数

方法：前面有主，主是谁，this就是谁

函数：this默认为window 没有主

**所以说，方法和函数的最大区别是方法有主，函数没有主。**

例如



这个this是指[object HTMLDivElement]，也就是box这个对象

简单的说就是：

当某个事件触发的时候，被触发的这个元素就是this！！！！

**\*\*\*\*\*\*\* 非空对象才能加属性或者方法 ！！！！！**

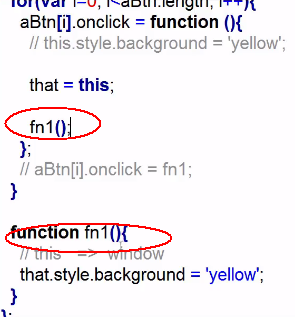
例如

var str = '';

str.a = 100;

alert(str.a);

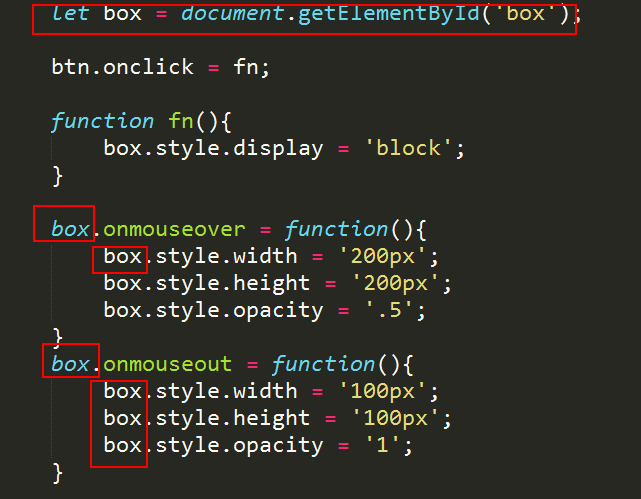
这时显示的就是undefined ,所以**非空对象才能加属性或者方法。**

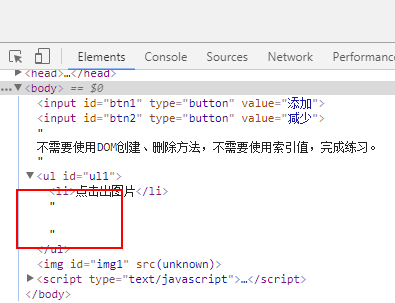


this 不就是自己吗？

不是

是指触发事件的元素



****???

# 无穷大

Infinity 正无穷大

-Infinity 负无穷大

# 选项卡

“大清洗”和”定点清除”的区别就是，大清洗是循环套循环，定点清除类似于递归...

## 1.大清洗

1.把所有的className全部清空

例如使用：.map(e=>e.className='')

2.把当前点击的按钮加上className

当操作某件事的时候，还要对应另外一件事，就用索引

lis[i].num = i;

## 2.定点清除

1.先把第一个按钮存起来

2.点击第二个按钮的时候，把第一个（上一个）按钮的className清掉

3.把当前的className加上active

4.更改上一个按钮prevLi的值，以便下一次用

# 数据类型

## javascript的构成

ECMAScript + DOM + BOM

## ECMAScript

规定了一些语法：

if(){}

for(;;){}

while(){}

运算符

数据类型

各种数据类型的方法

5.0之前，都叫ES5

ES2016以后的，都叫ES6（ES2017,ES2018...）

ES6泛指下一代js语法

# 函数

## 什么是“函数”？

实现什么目的的代码都放在函数内

**函数是这样一段JavaScipt代码，它只定义一次，但可能被执行会调用任意次。**

**函数是一种值可以赋值给变量**

var square = function(x) {

return x\*x

};

## 什么是“方法”？

**如果函数挂载在一个对象上，作为对象的一个属性，就称它为“对象的方法”！**

也就是说：**当函数赋值给对象的属性，我们称之为“方法”。**

## 函数声明

声明一个函数 **function**

fn() //这就是调用

function **fn**(){} //这里就是定义一个函数

调用在定义上面或者下面都可以。

函数表达式

let fn = function(){} //**这里只是声明一个函数，就是将函数的返回值赋值给fn**

**这里只是一个声明，没有调用**

调用在声明之后，不然会报错

函数可以套函数，直到内存占满~

在函数才被声明的时候，函数内部就有很多属性或者方法

比如:

name属性,length属性,arguments属性

prototype属性->原型->是给它的实例化对象使用的->它的值是一个\*\*对象\*\*

Scopes属性

apply方法

bind方法

call方法

## 类型

不管是匿名函数还是有名函数，都要有一个名字，这样才可以执行它。

**单纯的函数是无法执行的！代码从上到下执行到函数不会主动执行。**

### 匿名函数

function(){

alert('欢迎来到妙味课堂')

}

1. 匿名函数没有声明，就要加括号
2. 匿名函数的自执行要在后面加括号

配合var关键字，[匿名函数](https://www.baidu.com/s?wd=%E5%8C%BF%E5%90%8D%E5%87%BD%E6%95%B0&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Ydnhm3mW61m1RYn103mhwh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWfzPW6vP104nWb3PWn1nj0Y" \t "_blank)可以有效的保证在页面上写入Javascript，而不会造成全局变量的污染。

例如：

这里就是声明一个函数名，并把一个匿名函数赋值给它。

let move = function (){

el.scrollTop = 10000;

}

move();

### 有名函数

function **fn**(){}

有名函数带不带function都一样

例如：

function move(){

el.scrollTop = 10000;

}

move();

## 函数的调用方式

### 1.直接调用函数（打开网页即调用函数）

##### 有名函数的直接调用

调用格式：函数名 + (括号)

例如：

fn(); //函数调用，调用它的主是window,相当于window.fn();

##### 匿名函数的直接调用

调用方式：函数自执行（**(function(){}) ()**）

在一个表达式中调用两次函数。

<script>

function plus1(x) {

return x+1;

}

var square = function(x) {

return x\*x

}

var result = square(plus1(3)); //=>**在一个表达式中调用两个函数**

console.log(result);

</script>

### 2.事件触发调用函数

如果使用有名函数给事件调用，那么不要加()

**document.getElementById('box').onclick = fn; √**

**document.getElementById('box').onclick = fn(); ×**

function **fn**(){}

### 3.定时器调用

<script>

function fn1 (){

alert(999);

}

setInterval(fn1,1000); //这里fn1后不能加（），否则就自调用了。

</script>

例如：

let f = fn(); //1.调用函数fn(); 2.将fn()的返回值赋值给f

如果使用有名函数给事件调用，那么不要加()

document.getElementById('box').onclick = fn; √

document.getElementById('box').onclick = fn(); ×

### 4.回调函数(钩子函数callback)

又称钩子函数。

当一连串动作**完成之后**，再想执行一个函数，那么这个函数就叫“**回调函数**”。

通俗形象的解释：把函数f2当做一个参数传给函数f1，并且在f1里适当的时间执行f2

“回调”可以理解这是一个函数被调用的机制。

由于在调用函数时，实参中有回调函数，也有可能没有回调函数，

所以，在定义的函数中，对于回调函数，需要加一个判断。

if ( endFn ) {

endFn();

}

等价于

endFn && endFn (); //钩子函数，如果endFn这个函数存在或者是传了，那么执行endFn ();

## 函数传参

形参:

形参 -> 形式上的参数

例如：function(a,b){} 中 a,b就是形参

1.是可变的-> 直接理解为变量 var,let

2.形参是可以定义初始值的

function(a=1,b=[]){}

a默认值为1

b默认值为一个空白数组

如果没有初始值，就为undefined

实参:

实实在在的参数

divs[1].onclick = function(){

fn(2,5); //2,5是实参 -> 实实在在的参数

}

\*\*\* 实参的个数一定要对应形参的个数

function fn(a=1,b=[]){}

fn(5,6)

\*\*\* 实参可以传入任何数据

### 形参的默认值（ES6）

function fn(a=10){

console.log(a) //打印‘0’

}

fn(0)

在ES6中，可以给形参赋值。（但是不能再用let声明这个形参）

如果有实参，那么形参的赋值失效

如果没有实参，则形参的赋值为默认值。

## 作用

函数主要的用途是**复用！！！！**

## 封装

将一段代码放到一个 function name () { }中，

然后用name ();调用

对象是真的

## fn() 函数名+括号做什么？

**1.执行函数（函数调用）**

**2.拿到函数返回值**

例如：

var newfn = fn1(); //这个的意思就是：

1.将fn1这个函数执行

2.声明一个变量newfn.

3.将fn1这个函数的返回值赋值给newfn

## fn()() 函数名+双括号做什么？

**执行函数返回值**

### 函数的调用

#### 什么是函数调用

函数名+ （） 只要看到函数名加括号，就说明这个函数调用了。

fn();

#### 函数调用做什么

1.执行函数

2.执行返回值

#### 函数的第三种调用方式

function fn() {

alert(123)

}

fn.call();

call能够改变this指针

call能够改变this指针（this指向）

还有fn.apply(document)

applay能够改变this指针

applay能够改变this指针（this指向）

### 函数传参

#### 实参

fn (1,2)

实参 实实在在的一个参数

实参的个数一定要对应形参的个数

fn(100,1,function() ){

alert(1);

}

function fn(a,b){

console.log(a+b)

}

#### 形参

**1.什么是形参？**

形式上的参数

是可变的 直接理解为变量 var a,b

**2.形参初始值**

如果没有初始值，就为undefined.就相当于声明var/let一个变量。

形参是可以定义一个初始值的。

如果有初始值，例如function fn(**a=1,b=[]**){ }

function后面是形参

#### 实参会覆盖形参的初始值

例如：

function fn(arr){

alert(arr);//10

return function(**a=15**){ //这里面的形参a在执行的时候被实参覆盖

alert(a);//20

}

var arr = 5;

}

let f = fn(10);

f(**20**);

#### 实参和形参可以重名

实参和形参是可以重名的，但是个人建议在初期还是不要重名

这个更好去理解，形参和实参

let a = 10;

let b = 20;

function fn(a){

alert(a);

}

fn(b);

### arguments

**既能读，又能写的，实参集合（类数组）**

arguments 中保存了**实际传入**函数内的**所有参数**！

(因为形参的个数有时小于实参，所以实际传入了实参，但是并没有接收出来)

当函数的参数无法确定的时候使用，

因为arguments就是函数里面的一部分，肯定会得到所有的实参。

**即能读；**

例如

function fn(a,b,c,d,e,f,g){

console.log(arguments); //在这里读实参

let num = 0;

for(var i=0;i<arguments.length;i++){

num += arguments[i];

}

console.log(num);

}

fn(2,3,4,5,6,7,9,8,3,4,5,6,7,9,8,3,4,5,6,7,9,8,3,4,5,6,7,9,8,3,4,5,6,7,9,8,3,4,5,6,7,9,8,3,4,5,6,7,9,8,3,4,5,6,7,9,8,3,4,5,6,7,9,8,2,3,4,5,6,7,9,8,3,4,5,6,7,9,8,3,4,5,6,7,9,8,3,4,5,6,7,9,8,3,4,5,6,7,9,8,3,4,5,6,7,9,8,3,4,5,6,7,9,8,3,4,5,6,7,9,8,3,4,5,6,7,9);

**也能写**：修改argument的值，实参改变

function fn2(a,b){

/\*

修改arguments实参的值就发生变化了，

一般来说，arguments的值是不允许改变的。

\*/

arguments[0] = 10; **//在这里修改实参**

console.log(a); // 已经不是2了,显示是10

}

fn2(2,3);

#### JS重载

在js中不能直接实现函数的重载，但有没有办法实现类似重载效果的方法呢？

  可以，主要就是利用arguments

function test1()

{

var text="";

if(arguments.length==1)

{

//调用一个参数的方法

}

else if(arguments.length==2)

{

//调用两个参数的方法

}

else { //其他的方法

}

}

### ...arr

获取剩余参数

而且...arr必须是最后一个参数

        function fn( a,b, ...r ){

            // console.log( a,arguments );

            console.log( r );

        }

        fn(1,2,3,4,5)

### 想要获取函数内值的方法

1. 在函数外声明**全局变量var,**并在函数内将要找的值赋值给声明的全局变量。
2. 通过**局部的函数调用**+**传参**，来找到函数内的值

function fn1 () {

var **a** = “99999999”

fn2(**a**)

}

function fn2 (**ab**) {

alert(**ab**);

}

### 函数返回值

#### 1.return 返回函数的值

当有return的时候：“**函数名称+()**”就等于return后面的东西

当没有return的时候：函数的**默认**返回值为**undefined**

**例如：**

<script>

fn2(); //这里是函数fn2，这个大函数，

只会执行fn2,也就是**return** function fn (){alert(1);}.

但是执行的结果是什么都没有

fn2()(); //加了一个括号，

就是执行fn,也就是**alert(1);**

function fn2(){

return function fn (){

alert(1);

}

}

</script>

父级访问不到子级

子级**可以访问父级**,以及**可以访问爷爷级**，例如：

<script>

fn2(**10**)(**20**);

function fn2(**a**){

return function fn (**b**){

alert(**a+b**); //这里弹出来的就是30

}

}

</script>

**只要是函数，都有返回值。**

**例如：**

**function fun(n,o) { //n = 0**

**console.log(o); //undefined**

**return {**

**fun:function(m){ // m = 1,0**

**return fun(m,n);**

**}**

**};**

**}**

**var a = fun(0); //这里的意思是把fun(0)的返回值赋值给a**

**//上面例子中的红字为a.**

**a.fun(1); //这里的意思是执行a中的fun函数，实参为1**

\*\*函数\*\*返回值：（\*\*函数\*\*的返回值，return只能在函数内使用）

是函数的返回值，只能在函数内部使用 ？？？

这个的意思就是只有当用fn2();这样的调用函数调用的时候，才能用

#### 2.在作用域外面声明一个变量，返回函数的值

### 函数返回值的目的

函数名 + ()就等于 return后面的返回值结果;

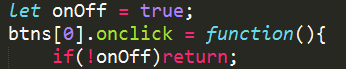
因为函数是一个域，在域中的变量或者运算结果，在外部是访问不到的（js的特性）

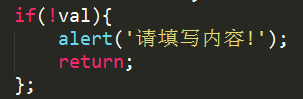
如果想让**外部也能够访问到函数内部的变量**或者运算结果，这个时候就用return

如果没有return默认返回值为undefined

什么时候用return?

1. 外部想访问内部的变量或者运算结果。
2. **在函数内部终止后面的代码执行。例如：**

****

****

### 函数使用注意事项

分号；

最好在每句话后面加上

等号 =

把右边的东西给左边

“”/ `` 双引号/单引号

只要是操作页面的东西，都要用''或者""

页面中的所有内容都是字符串

### 调试

1.alert,

2.console.log() 在浏览器的控制台打印内容。

3.console.dir() 是指console.log()能实现的它都能实现，并且这个console.dir()能够看到更加详细的内容。

console.log()只能看到HTML结构，而console.dir()能够看到HTML结构+CSS属性值。

**小技巧：**

多使用alert或者console.log,越快学会使用，越容易开窍。

### 函数的特性

#### 1.可套用性

也就是说函数套函数。

例如：

window.onload **= function(){**

var set = document.getElementById("set1");

set.onclick **= function () {**

Oset\_show.style.display = 'block';

};

}

#### 2.单向取值

函数的特性是，

**子函数能够访问到父函数**的参数或者变量

而**父函数访问不了子函数**的变量或者参数。

### 函数自执行

**(function(){**

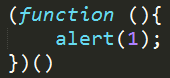
**}) ();**

**(**function(){

}**)** **()**

**也就是函数上面套一个括号，后面跟一个括号，就形成了函数自执行**

例如：



# 域

**1.全局域**

script就是一个域，里面的var 和function是全局的

**2.局部域**

函数也是一个域，是局部的域

1. {}
2. function () { }

域也就是块？

“块”是“域”的子集

“函数”也是“域”的子集

范围，函数可以套函数，子函数能够使用父函数的变量或者参数

如果父函数没有还会向父级的父级找，直到找到window为止。

子级会在父级的函数体内查找某个变量，如果没有会去父级的形参中找，直接走形参

函数自身有这个变量会先找自身的变量

在使用var的时候，会进行预解析，如果在声明之前打印这个值，那么会是一个undefined.

本质：

**var**声明的时候**会**在**window下注册**一个变量的属性。

**let** 声明的视乎**不会**在**window下注册**变量

**let声明上面会有暂存死区（不会域解析）**

如果在一个域中声明，那么在声明之前找这个变量就不会走域解析

有变量和函数，函数把变量覆盖

只要有函数就是一个域。function(){}

只要是一个块就是一个域 {}

## 作用域链

能够有规则的查找变量，这种规则叫做作用域链

函数可以套函数，子函数能够使用父函数的变量或者参数

例1：

function delFiledata (**context**,groupApi){ //这里是父级的参数（形参）

console.log(context);

var params = {

user: Cache.auth.id,

token: Cache.auth.utoken,

app\_code: 'ruixin\_dev',

device\_type: 'pc',

file\_id: context.file.file\_id

};

//下面的“子函数”并不需要传入参数，就能用父级的参数

groupApi.deleteSharedFile(params,function(errorCode,result){

if(errorCode){

alert('网络出错了')

}

**context**.$emit("delfiledata"); //这里就用到了父级的参数

} );

};

例2：

delFile:function(){

let that = this;

common.messagebox({

type:'confirm',

message: "确认删除此文件？删除后不可恢复。",

callback:function(){

//点击确定删除文件

delFiledata(that,GroupApi);

}

});

如果父函数没有还会向父级的父级找，直到找到window为止。

子级会在父级的函数体内查找某个变量，如果没有会去父级的形参中找，直接走形参

**小技巧：作用域链？**

只要是函数就是一个域，那么会先找域中的变量，

如果找不到还会去它的父级查找，直到window还是没有，就报错。

## 作用域和作用域链

## 什么是作用域？

作用域: 执行js的范围。域:范围。

function后面的{}才是一个作用域。

if判断、for循环后面的{}不是一个作用域。

## 全局作用域与局部作用域

当一个变量在**全局声明**的时候，那么在这个全局的**所有子作用域内**都**能读写**这个变量。（也就是是说在子作用域内同名的局部变量能够覆盖同名的全局变量）

如果一个变量**没有**在全局声明的时候，那么在这个全局的所有子作用域内**不能传递这个变量的值**。

## JS特性

1. js的一种特性,子级能够访问父级的参数或者变量，如果父级还是没有（形参也没有）那么会去父级的父级去中找，如果到了全局还是没有，那么就报错。
2. 子级先会去父级的函数体内找某个变量，如果没有那么还会去父级的形参中查找。

如果父级的形参中有，直接走形参。

var a = 1;

function fn1(a){ //**由于没有传实参，形参中的变量相当于 var a = undifined**

alert(a); // **undefined**

a = 2; //**由于fn1的预解析解析了var a = undifined,**

**所以赋值给函数内的声明值a,并不是函数外的a**

}

fn1();

alert(a); //**这里是函数外a 的值1**

1. 如果自身有这个变量会优先自身的变量

(访问变量的时候应该在声明之后访问（let）)

4.在使用var的时候会进行预解析，如果在声明之前打印这个值

那么会是一个undefined

本质：

var 声明的时候会在window下注册一个变量的属性(在全局的情况下会挂在window上，如果在函数内是不会挂在window上的)，赋值为undefined

let 声明（在全局下）的时候不会在window下注册这个变量

let声明的上面有暂存死区（不会预解析）

如果在一个域中声明，那么去声明之前找这个变量是不会走作用域链的(就报错)。

5.函数内有var声明的变量，参数也有一个和var声明变量一样的名字的形参

在函数一开始访问这个变量，那么会看参数是否有值，**如果没有值直接undefined**，

如果传入实参或者形参中有默认值，那么直接输出形参值。

注意：

在let声明的时候，一定不要和形参的名字重复。

如果重复，就会报错！！！

例如：

<script>

function fn(**a**){

let **a** = 10; //这里的声明的名字和形参名字相同，报错

let b = 20;

function fn2(b){

alert(b);

var b = 30;

}

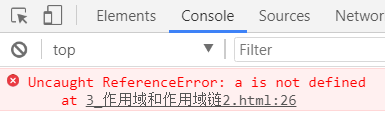
fn2(b);

}

fn(a);

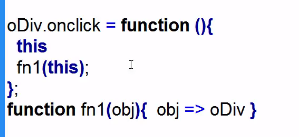
</script>

报错为：



## 预解析

### 例子

?

上面的怎么能执行下面的？

因为会预解析，

函数的预解：析就会让**整个函数**都被存在预解析的库里，从上到下逐行执行代码的时候随时调用！

变量的预解：析将变量预解析为**undefined，**被存在预解析的库里，从上到下逐行执行代码的时候随时调用

### 什么是js预解析机制:

在打开页面的时候会**提前执行的js顺序**，**这种规则叫预解析器**。

### 如何预解析（也叫变量提升）

1. **碰到var/let 碰到 function就会提前解析**

形参a的本质是var a = undifined.

解析之后会将解析的结果存到一个库中，供后续调用

1. 解析之后逐行解读代码。

（表达式是可以改变预解析的值！）

（参数也是一个表达式）

### 预解析注意点

**注意一：**

也就是说，如果一段js代码中既有函数也有var（同名的），那么直接走最后一个同名函数。

**注意二**

**就算var后面是个函数，也不算函数，只算变量**。

### 只有函数的预解析

1.在预解析过程中，如果**没有相同名字的函数**，那么就会从上到下依次解析。解析完成之后，从上到下依次执行。

2.在预解析的过程中，如果2个**函数的名字一样**，**那么后面的会把前面的覆盖**。等同于**前一个函数就从世界上消失了**。

例如：

alert(a);

function a (){

alert(1);

}

alert(a);

function a (){ **//这个函数把前面的弹alert(1)的函数覆盖了！！！**

alert(2);  **//所以结果弹三次“ function a (){alert(2); }”**

}

alert(a);

### 只有变量的预解析

只有变量的预解析，**将变量预解析为undefined**,然后再从上到下执行代码。

**就算var后面是个函数，也不算函数，只算变量。**

alert(a); //undefined

var a = 10;

alert(a); //10

var a = 20;

alert(a); //20;

### 函数+变量的预解析

1. 如果一段js代码中既有函数也有var（同名的），那么直接走最后一个同名函数。
2. 预解析完毕，然后逐行解析代码。逐行解读代码的时候，有=号就赋值，没有等号就过。如果逐行解析代码的时候，遇到同名函数，后面的覆盖前面的。
3. **就算var后面是个函数，也不算函数，只算变量。**

当有函数和变量的时候，如果是同名的，只走函数，变量的undifined略去。

如果不是同名的，从上到下执行

例如 ：

alert(a);

function a (){

alert(2);

}

var a = function(){

alert(5);

}

alert(a);

var a = 30;

function a (){

alert(a);

}

a(); //30() 报错

alert(a);

### 什么都没有的预解析

向下逐行解读代码。

## 作用域链

能够不段有规则的去查找变量，这种规则就是作用域链。

**触发条件：**

**在域内找不到变量时，就会触发作用域链机制**

当在域内找到变量时（包括undifined）

**执行顺序：**

子级先会去父级的函数体内找某个变量，

如果没有那么还会去父级的形参中查找（**由里到外**）。

## 逻辑顺序

一.全局预解析，逐行解读代码。

二．一旦遇到**函数调用**（例如:fn()）走作用域 。

1.预解析。

2.从上到下执行,逐行解读代码。也就是说，当遇到表达式的时候，不管是函数，还是变量，都会执行表达式。

三．走作用域链

# 多个script标签

**执行顺序：**

从上到下。

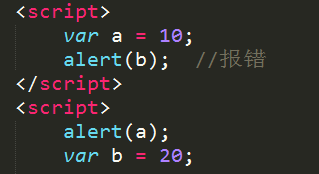
先将上面的script块执行，预解析、逐行解析表达式。

然后执行下面的script块。

**内部关联：**

如果有多个script标签，那么**只会向上去找**某个变量（函数）

如果有一个script标签报错，另外一个访问成功。**报错的在前，不会影响后面的访问成功。**



### 如何解决报错

1.在上面的script标签中使用window.onload

<script>

var a = 10;

window.onload = function(){

alert(b)

}

</script>

<script>

var b = 20;

</script>

2. 在上面的script标签中使用函数调用传参

<script>

var a = 10;

function fn(data){

console.log(data)

}

</script>

<script>

var b = 20;

fn(b);

</script>

### <script>标签可以放在html中

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Document</title>

<script src="lbrary.js"></script>

<!-- 引入一个javascript库 -->

</head>

<body>

<p>this is a paragraph of HTML</p>

**<script>**

**在这里编写嵌入到HTML文件中的javascript代码**

**</script>**

<p>Here is more HTML.</p>

</body>

</html>

# 闭包

## 闭包是个啥？

一个函数就是一个闭包。

function fn () {

var a = 10 ;

return function () {

a++;

console.log(a)

}

}

var a = fn();

a();

## 如何形成闭包环境？

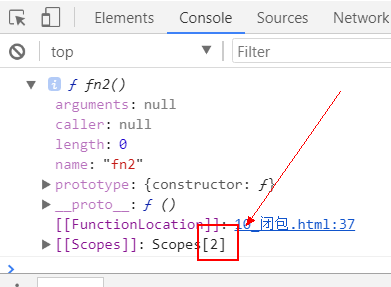
**条件一,外面:。父函数外使用着局部子函数的内容。**

也就是说，大函数外使用着局部的内容. 父函数中return局部的子函数,并在父函数外部调用

（如果父函数被存进一个变量中，就能在父函数变量不被回收的状态下无限次的使用！）

**条件二,内部:通过(**执行**)函数套函数，使子级要使用着父级的参数或者变量(**或者值**)**

形成闭包：



这个Scopes代表什么？？？代表形成闭包

**例如：**

**function fn(){**

**var a = 10;**

**function fn2(){**

**a ++; // 这里就是指“子级用着父级的变量”**

**alert(a); //这里弹出的是11，用了父级的变量**

**}**

**return fn2;**

**}**

**var f = fn(); //这里说的就是“外面：使用着局部的内容，”**

**f(); //这里说的就是“外面：使用着局部的内容，”**

function fn(){

var a = 10;

return function f(){

a ++;

alert(a);

}

}

var a = fn(); //这里是指将fn这个函数的返回值赋值给a

a(); //打印11

a(); //打印12

a(); //打印13

// console.dir(a);

console.log(fn() == fn());

这里为什么是false?

因为每一次执行fn的返回值是不同的，没有被垃圾回收机制回收

## 闭包的好处？

形成了闭包时,调用这个函数时，函数的变量就**不会**被浏览器的垃圾回收机制回收。

没有形成闭包时,调用这个函数时，函数的变量就**会**被浏览器的垃圾回收机制回收。

## 应用

1.子级能访问父级的参数或者变量，并且引用（使用这个参数或者变量）

2.父级的**参数或者变量**不被浏览器的垃圾回收机制回收。

**不被回收,**

闭包:

真正可以拿来用的闭包:**函数套函数**，

子级能访问父级的参数或者变量，并且引用（使用这个参数或者变量）

这个时候，父级的参数或者变量不被垃圾回收机制回收

**在没有形成闭包环境的时候：**

在函数名 + 括号执行的时候，函数内的参数或者变量会被浏览器垃圾回收机制给回收。

下一次再函数名 + 括号,运行完之后又会**被回收**。

## 举例

<script>

            function fn1(){

                var a = 0;

                function fn2(){

                    ++ a;

                    alert(a);

                }

                return fn2;

            }

            fn1()();//1 这里弹完1之后，因为fn1()没有被存起来，所以再次调用的时候，父函数的变量a==1被垃圾回收机制回收了

            fn1()();//1 因为刚才那个a==1被垃圾回收机制回收了，所以重新来执行fn1(),所以还是1。但是fn1()执行完之后，a==1又被垃圾回收机制回收了

            var newFn = fn1(); //因为外部引用了，而不是调用，所以形成闭包

            newFn();//1 这时执行完fn1()了，因为它被存起来到newFn中了，所以a==1没有被垃圾回收机制回收，a还是1

            newFn();//2  因为刚才a==1,所以在刚才的基础上再次执行，变成了2。

    </script>

var a=0;

function fn2(){

++ a;

console.log(a);

}

fn2();

fn2(); //这里形成闭包了吗？为什么同级能使用同级的参数？

**这里没有形成闭包，这里因为a是一个全局变量所以重复使用**

## 闭包的应用价值

let arr = [111,222,333];

const btns = document.getElementsByTagName('input');

**原来没有用闭包的时候是这么写的：**

for(var i=0;i<btns.length;i++){

btns[i].index = i;

btns[i].onclick = function(){

alert(arr[this.index]);

}

}

**用了闭包可以这么写：**

for(var i=0;i<btns.length;i++){

function fn(a){

btns[a].onclick = function(){

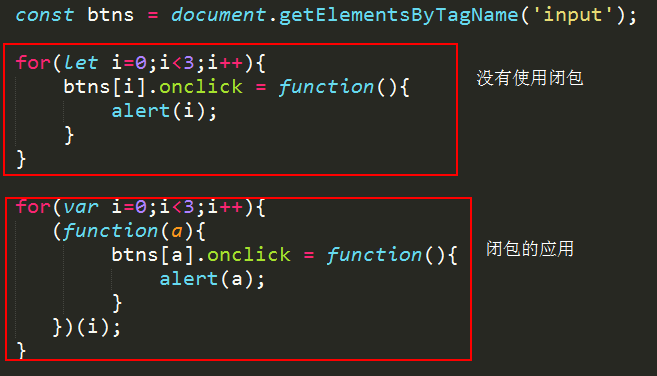
alert(arr[a]);

}

}

fn(i);

}



## 闭包的优缺点

### 优点

闭包就是一个函数引用另外一个函数的变量，**因为变量被引用着所以不会被回收，因此可以用来封装一个私有变量**。

这是优点也是缺点，不必要的闭包只会徒增内存消耗！

另外使用闭包也要注意变量的值是否符合你的要求，因为他就像一个静态私有变量一样。

闭包通常会跟很多东西混搭起来，接触多了才能加深理解，这里只是开个头说说基础性的东西。

内部一直使用着外部的值，那么就一直不被回收。

### 缺点

一直不回收，如果数量大了，会影响性能。

IE6版本浏览器会出现内存泄露。

# 定时器

## 定时器概念

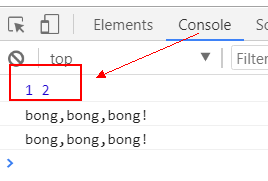
间隔型定时器: **setInterval** (函数,毫秒) 一直炸

延迟型定时器: **setTimeout**  (函数,毫秒) 只炸一次

1秒等于1000毫秒

返回值为数字，**每有一个定时器就添加一个数字（+1）**，只要开定时器这个数字就一直在。

**数字从1开始，整数递增。**



## 关闭定时器概念

关闭定时器：

clearInterval(要关闭的定时器);

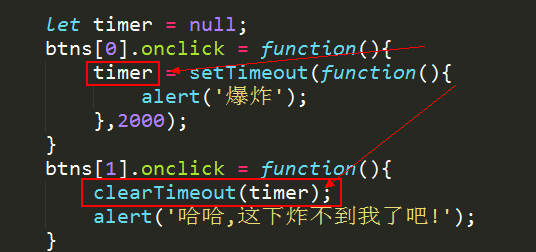
clearTimeout(要关闭的定时器);

**这2个关闭定时器都可以用。**

**也就是说，clearInterval可以清除setInterval也可以清除setTimeout**

用法：

1.原来的定时器是什么名字，如果要关闭就用clearTimeout(要关闭的定时器名字)



2.clearInterval后面的定时器。有就清除这个定时器。没有这个定时器，就不清除。都不会报错。

let timer = **null**; //为什么要声明一个空值？因为按钮1和按钮2都要通过定时器timer来执行函数，所以这里要声明，开启定时器和关闭定时器都需要！！

声明定时器还要用一个null!!

btns[0].onclick = function(){

timer = setTimeout(function(){ //声明定时器的

alert('爆炸');

},2000);

}

btns[1].onclick = function(){

**clearTimeout(timer);**

alert('哈哈,这下炸不到我了吧!');

}

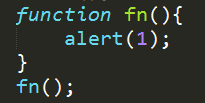
3.小技巧

当执行一个定时器的时候，为了不让它在还没执行完的时候再次执行，先使用clearInterval将这个定时器关闭。

## 函数的第3种调用方式:定时器调用

1. 直接调用

**格式为：函数名称+（）+；**



1. 事件调用



事件调用中包括匿名函数调用和有名函数调用，**有名函数调用就要用括号！！！**

1. 定时器调用

**定时器调用会将代码推到下一轮去执行，即使时间设置为0.**

**格式为：定时器函数+（有名函数/匿名函数,定时时间）**

set**I**nterval();

注意：这里如果加有名函数，那么不能加()括号，否则会立即执行，不会按照定时时间执行。

需要给定时器一个名字，因为在结束定时器的时候会用到这个名字。例如：

**var timer = setInterval ( fn1 , 1000 );**

**clearInterval(timer)**

## 定时器的问题

狂点开始按钮的时候，timer会一直赋值新的定时器，

好几个定时器开着，页面就乱了

当关闭定时器的时候，只关闭了最新赋值的定时器。

会导致关闭不了定时器。

解决：

1. 点完之后就不让点了。这种性能更高。

2. 再这次开启定时器**之前**，先把之前的定时器清了。

clearInterval(timer) 这里虽然是timer=null但是还是能清除掉

## setTimeout

setTimeout(fn1,1000);

这个说明是读到这一行的时候，先不执行，继续执行下面的代码。

再过过1000ms才执行这个定时器里面的事件。

transition

**定时器下的this指向window**

setTimeout(function(){console.log(this)},1000);

# 计时器

Console.time()

Console.timeEnd()

<div id='div1'></div>

<script>

var oDiv = document.getElementById("div1")

console.time("hello");

for(var i = 0;i<5000;i++){

oDiv.innerHTML += "a"

}

console.timeEnd('hello') //hello: 159.697021484375ms

</script>

# 时间对象

## 获取已有（本机/服务器）时间

new Date(); //**注意这个D是大写！！！！！**

获取到当前\*本机\*的时间(这个时间是可以任意修改的)

真正在开发中的使用服务器时间。

**new Date**; 这个是获取当前时间，以下面的格式：



**+new Date**; 这个是获取当前时间，以ms为单位的格式：



**new Date().toLocaleTimeString()** ; // 下午8:55:36

获取当前时间并转换为标准可读的时间格式

<script>

var Now = new Date();

console.log(Now); //这个是取现在的本机时间

</script>

## 声明时间

## 设置时间

setData

setDate() 方法用于设置一个月的某一天。

参数（目的是为了**设置**自己想要的时间）：

date = Sun Oct 07 2018 09:37:42 GMT+0800 (中国标准时间)

date.setDate(0); //当前日期从0开始算

date ==Sun Sep 30 2018 09:37:42 GMT+0800 (中国标准时间)

**某个月设置日期为0，那么就是得到上个月的最后一天**

**setMonth() 用于设置一年中的某个月**

### 1.时间戳数字

可以放**时间戳**数字

(**1970.1.1号0点0分0秒的时间到某个时间\*毫秒\*数字**)

### 2.也可以放字符串

new Date('December 17, 1995 03:24:00');

一 月：January 简写:Jan.

二 月：February 简写:Feb.

三 月：March 简写:Mar.

四 月：April 简写:Apr.

五 月：May 简写:May.

六 月：June 简写:Jun.

七 月：July 简写:Jul.

八 月：August 简写:Aug.

九 月：September 简写:Sep.

十 月：October 简写:Oct.

十一月：November 简写:Nov.

十二月：December 简写:Dec.

### 3.还可以传入数字

new Date(2017, 11, 31, 0, 0, 0); //月份需要减一（如果要设置12月，那么要写11）

### 4.数字版字符串

（这种比较好记！！！）

**let date = new Date('2017 12 7 00:00:00')**

以上的时间，如果设置的时间过大，那么会跳到下几个月或者下几天甚至下几年

<script>

var newTime = new Date(2018,8,15,15,22,10)

console.log(newTime); //这个是取设置的时间

</script>

## 时间戳转时间公式

？？？

## date的操作

**以下返回值均为number类型。**

// let iY = date.**getFullYear**(); 获取年

// let iMou = date.**getMonth**(); 获取月份

注意:获取出来的数字比想要的数字少1，所以要+1。

这里仅仅是获取出来的数字比想要的少1而已，

但是实质上new Date()出来的还是本机的真实时间。

// let iDate = date.**getDate**(); 获取天

// let iDay = date.**getDay**(); 周几 0-6 (0属于周日)

// let iH = date.**getHours**(); 获取小时

// let iM = date.**getMinutes**(); 获取分钟

// let iS = date.**getSeconds**(); 获取秒

// let Milliseconds = date.**getMilliseconds**(); 获取毫秒。

倒计时

let t = (newDate - nowDate) / 1000; //除以1000表示把获取到的毫秒转换为秒

let iDay = **Math.floor**(t/86400);

let iH = Math.floor(t%86400/3600);

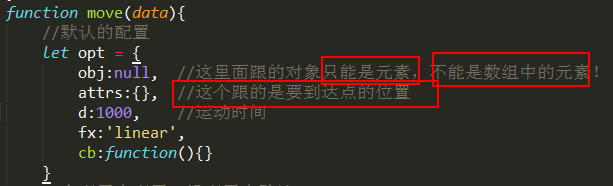
let iM = Math.floor(**t%86400%3600**/60);

let Se = Math.floor(t%60);

**Math.floor ( ) 向下取整**

**Math.ceil () 向上取整**

## 运动函数tween



# 对象拷贝

## 浅拷贝/浅克隆

### 1.方法一Object.assign

Object.assign方法用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。

Object.assign方法的**第一个参数是目标对象**，后面的参数**都是源对象**。

例如：

        const target = { a: 1 };

        const source1 = { b: 2 };

        const source2 = { c: 3 };

        Object.assign(target, source1, source2);

        target // {a:1, b:2, c:3}

注意，如果目标对象与源对象有

属性，或多个源对象有**同名属性**，则**后面的属性会覆盖前面的属性**。

const target = { a: 1, b: 1 };

const source1 = { b: 2, c: 2 };

const source2 = { c: 3 };

Object.assign(target, source1, source2);

target // {a:1, b:2, c:3}

如果只有一个参数，Object.assign会直接**返回**该参数。

const obj = {a: 1};

Object.assign(obj) === obj // true

此方法可以将...source中的属性复制到target中，同名属性会进行覆盖，并且在复制过程中实现了'伪'深拷贝

let foo = {

a: 1,

b: 2,

c: {

d: 1,

}

}

let bar = {};

Object.assign(bar, foo);

foo.a++;

foo.a === 2 //true

bar.a === 1 //true

乍一看，好像已经实现了深拷贝的效果，对foo.a进行的操作并没有体现在bar.a中,但是再往后看

foo.**c**.d++;

foo.c.d === 2 //true

bar.c.d === 1 //false

bar.c.d === 2 //true

Object.assign()的拷贝类型十分明显了,这是一种可以对非嵌套对象进行深拷贝的方法,如果对象中出现嵌套情况,那么其对被嵌套对象的行为就成了普通的浅拷贝.

### 2.方法二for in

for(var attr in obj2){

obj[attr] = obj2[attr];

}

//默认对象

const obj = {

name:'小黄',

age:18

}

/\*

\*\*\* 如果一个对象中有重名的key值，那么后面的会把前面的覆盖。

\*/

//配置的对象

const obj2 = {

job:'web前端',

name:'小黑',

name:'小白'

}

## 深拷贝

### 1.方法一：JSON.parse(JSON.stringify数据(arr));

arrCopy = JSON.parse(JSON.stringify(arr));

### 2.方法二：利用for in +递归+push

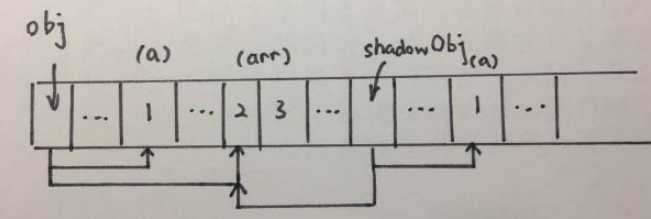
利用for in +递归+push来深度克隆

例如：

## 浅拷贝和深拷贝的区别

浅拷贝会导致 obj.arr 和 shallowObj.arr 指向同一块内存地址;

而深拷贝不会导致bj.arr 和 shallowObj.arr 指向同一块内存地址



<script>

        var obj = { a:1, arr: [2,3],c:1 };

        var shallowObj = shallowCopy(obj);

        function shallowCopy(src) {

        var dst = {};

        for (var prop in src) {

            if (src.hasOwnProperty(prop)) {

            dst[prop] = src[prop];

            }

        }

        return dst;

        }

        shallowObj.arr[1] = 5;

        console.log(obj.arr[1]); //因为是浅拷贝导致shallowObj和obj地址相同，所以改变拷贝对象的值就会改变原对象的值

    </script>

## 赋值/赋址

/\*

=

把右边的东西赋值给左边 （简单类型）

**复合类型就不是赋值，而是赋址。**

复合类型的等号是\*\*赋址\*\*。

\*/

简单类型就是 赋值

复合类型就是 赋址

// var a = 10;

// var b = a;

// b ++;

// alert(b); //11

// alert(a); //10

// var arr = [1,2,3];

// var arr2 = arr;

// arr2.push(4);

// console.log(arr);

var arr3 = [];

var arr4 = [];

console.log(arr3 == arr4); //?为什么不相等

因为地址是不一样的！

### 赋址小问题

#### 当实参/形参为对象类型的时候，

**有可能会修改域外面的对象。**

（参数为对象类型的时候，有可能会修改外面的对象）

形参a其实就是实参，改变了对象形参就会改变这个对象本身。

例一：

let arr = [1,2,3,4];

function fn(a){

a.push(5);

console.log(a);

}

fn(arr);

console.log(arr); //这里的数组arr变成了[1,2,3,4,5]\

例二:

function fn(a){

a.height = '100px';

console.log(a);

}

var o = {

width:'200px'

}

fn(o);

console.log(o);

#### 当实参/形参为简单类型的时候，

**不会影响域外面的变量。**

（简单类型在函数中更改是不会影响外面的变量的）

例如：

var a = 10;

function fn(a){

a ++;

console.log(a); //这里是11

}

fn(a);

console.log(a); //这里还是10，所以说不会改变域以外的变量

var obj1 = {

width:'200px'

}

var obj2 = {

height:'50px'

}

console.log(obj2); //浏览器有问题

obj1 = obj2; //赋址。

obj1.height = '30px';

#### 如何将复合类型data的值赋给变量x

let data = {a:20,b:10}

let x = JSON.**parse**(JSON.**stringify**(data))

# 字符串

页面中的都是字符串，可不不可以理解为html+css都是？？？？

## 超级字符串

` ` //数字键最左边，ESC键下面的

        var html = **`**

<ul>

    <li class="red"></li>

    <li class='green'></li>

    <li class='${classStr}'></li>

</ul>**`**

使用超级字符串拼接可以更加直观

   var n = 2;

        var m = 3;

        console.log( "n+m="+(n+m) )

        console.log( `n+m=${n+m}` )

## 字符串操作

### 字符串比较大小

字符串比较大小时，只比较每个字符串**第一位的值**！

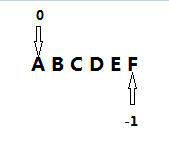
alert('11' > '2');//false

## 长度length

只可读**不可写**

### 截取类

#### 字符串定位



#### 1.slice

|  |  |  |  |
| --- | --- | --- | --- |
| slice | 英[slaɪs] | 美[slaɪs] | |
| vt. | 切成片; 切下; 划分; | |

|  |  |
| --- | --- |
| n. | 薄片; 一部分; （因失误而打出的） 曲线球; |

|  |  |
| --- | --- |
| vi. | 斜击; |

[例句]Try to eat at least four slices of bread a day.

每天尽量至少吃4片面包。

|  |  |
| --- | --- |
| [其他] | 第三人称单数：[slices](http://www.baidu.com/link?url=FhwAfaq5-O8_g9Cd7QGE_z4jsLNlxTdTegS7wpZdN5cSgHi62II-VoJ9cHUKhQI9EeOrylwdbzYF-cSqAwSzBbfHJ7e2Tu0_6oM31CEv71a" \t "_blank) 复数：[slices](http://www.baidu.com/link?url=FhwAfaq5-O8_g9Cd7QGE_z4jsLNlxTdTegS7wpZdN5cSgHi62II-VoJ9cHUKhQI9EeOrylwdbzYF-cSqAwSzBbfHJ7e2Tu0_6oM31CEv71a" \t "_blank) 现在分词：[slicing](http://www.baidu.com/link?url=8C1qd7yz7LiEBXFuruNWG52OVldNpDsn8hEVkDnzm5sTR0a5DthK1kBFd2Wa4SUiH69CRkpF4tqw2jZUN9J0Ny3pRh3OhzVmOvvi-jIoLce" \t "_blank) 过去式：[sliced](http://www.baidu.com/link?url=FhwAfaq5-O8_g9Cd7QGE_z4jsLNlxTdTegS7wpZdN5cSgHi62II-VoJ9cHUKhQI9EeOrylwdbzYF-cSqAwSzBlAY6T7_tZYL0QIZZkjsjAa" \t "_blank)过去分词：[sliced](http://www.baidu.com/link?url=FhwAfaq5-O8_g9Cd7QGE_z4jsLNlxTdTegS7wpZdN5cSgHi62II-VoJ9cHUKhQI9EeOrylwdbzYF-cSqAwSzBlAY6T7_tZYL0QIZZkjsjAa" \t "_blank) |

**返回值**

**返回提取字符串的某个部分**，并以**新的**字符串返回被提取的部分。

包含从 start 到 end （不包括该元素）的字符串。

**格式：**

stringObject.slice(start,end)

start 要抽取的片断的起始下标。如果是负数，则该参数规定的是从字符串的尾部开始算起的位置。也就是说，-1 指字符串的最后一个字符，-2 指倒数第二个字符，以此类推。

end 紧接着要抽取的片段的结尾的下标。

若未指定此参数，则要提取的子串包括 start 到原字符串结尾的字符串。

如果该参数是负数，那么它规定的是从字符串的尾部开始算起的位置。

|  |  |
| --- | --- |
| **参数** | **描述** |
| start | 要抽取的片断的起始下标。如果是负数，则该参数规定的是从字符串的尾部开始算起的位置。也就是说，-1 指字符串的最后一个字符，-2 指倒数第二个字符，以此类推。 |
| end | 紧接着要抽取的片段的结尾的下标。若未指定此参数，则要提取的子串包括 start 到原字符串结尾的字符串。如果该参数是负数，那么它规定的是从字符串的尾部开始算起的位置。 |

#### 2.substring

**返回值：**

一个**新的字符串**，该字符串值包含 stringObject 的一个子字符串，其内容是从 start 处到 end （**不包括该元素**）的字符串

**格式：**

substring() 方法用于提取字符串中介于两个指定下标之间的字符。

string.substring(from, to)

|  |  |
| --- | --- |
| **参数** | **描述** |
| from | 必需。一个**非负**的整数（负数也不报错），规定要提取的子串的第一个字符在 string Object 中的位置。 |
| to | 可选。一个**非负**的整数（负数也不报错），比要提取的子串的最后一个字符在 string Object 中的位置多 1。 **如果省略该参数，那么返回的子串会一直到字符串的结尾**。 |
| 无参数 | 当没有参数的时候，直接返回整个字符串 |

返回值：

substring() 方法返回的字符串**包括** 开始 处的字符，但**不包括** 结束 处的字符。

substring(起始位置,结束位置但不包括结束位置)：

截取指定位置的字符串的。

注意细节:

1如果第一个参数比第二个参数的值要大，**那么会颠倒顺序，再进行字符串操作**

2取值为一个**非负**的整数，如果参数为负数看成0即可，不报错。

let str = 'miaov';

console.log(str.substring(2,4)); //打印ao

console.log(str.substring(4,2)); //打印ao

如果参数为负数看成0即可。

let str = 'miaov';

console.log(str.substring(3,-1000)); //打印mia

#### slice和substring的区别

区别 ：1.slice第一个要比第二个参数大，不会改变位置

2.**slice可以使用负数，substring不能使用负数**

#### 3.substr

**返回值：**

一个**新的**字符串，包含从 stringObject 的 start（包括 start 所指的字符） 处开始的 length 个字符。如果没有指定 length，那么返回的字符串包含从 start 到 stringObject 的结尾的字符。

string.substr(从多少开始，截取几个)

**格式：**

*stringObject*.substr(*start*,*length*)

|  |  |
| --- | --- |
| **参数** | **描述** |
| *start* | 必需。要抽取的子串的起始下标。必须是数值。如果是负数，那么该参数声明从字符串的尾部开始算起的位置。也就是说，-1 指字符串中最后一个字符，-2 指倒数第二个字符，以此类推。 |
| *length* | 可选。子串中的字符数。必须是数值。如果省略了该参数，那么返回从 *stringObject* 的开始位置到结尾的字串。 |

string.substr(从多少开始，截取几个)

### 查询类

#### 1.charAt

**根据下标找到字符**

**indexOf的反义词**

**返回值：**

方法可返回指定位置的**字符**。

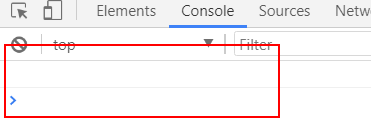
主要是为了兼容IE的低版本浏览器

**格式**

**string**.charAt(num)

字符串中第一个字符的下标是 0。

如果参数 index 不在 0 与 string.length 之间，该方法将返回一个**空字符串**。例如：



let c =”miaov”;

console.log(c.charAt**(0)**) //这里打印m

console.log(**c[0]**) //这里打印m,**目前基本都用这种方式打印字符串中的字符**

#### 2.1 indexOf

**从左往右**找到指定位置的**下标**

**返回值：**

可返回某个指定的字符串值在字符串中**首次**出现的位置。**即为找到元素的位置的下标（从0开始）。**

**类型是number**

**如果找不到就返回 -1**

格式：

string.indexOf(item,from)

item表示指定字符，item要加引号。

from表示起始的位置

如果说某个字符串中有这个值，那么就可以这些写

if(string.indexOf('') != -1){}

设置本身的位置只能找到本身。**如果要往后找，那么要在本身后 + +num**

**例如：**

let str2 = 'ababcabcababcababaabbcabababcaababbda';

let num = 0;

let index = [];

while(str2.indexOf('ab',num) != -1){

index.push( str2.indexOf('ab',num) );

num = str2.indexOf('ab',num) + 1;

}

console.log('出现的位置为'+index,'一共出现了'+index.length+'次');

#### 2.2 lastIndexOf

返回值：

lastIndexOf()

在一个字符串中的指定位置**从右向左**搜索。（这是和IndexOf唯一的不同）

方法可返回一个指定的字符串值**首次**出现位置（**注意是从后往前的首次**）的**下标**

返回值的类型为number.

格式：

stringObject.lastIndexOf(searchitem,from)

searchitem表示指定字符，searchvalue要加引号。

from表示起始的位置，如省略该参数，则将从字符串的最后一个字符处开始检索。

lastIndexOf() 方法对大小写敏感！

要检索的字符串值没有出现，则该方法返回 -1。

#### 3.includes(ES6)

**返回值：**

方法针对从第n个位置直到字符串结束。返回的都是布尔值。包含就true，不包含就false

格式：

(a:包含的字符串,b:指定开始查找的起始位置)

a:参数为必填

b:参数为选填，不填时默认为从左到右第一位。

#### 3.1 startsWith (ES6)

**返回值：**

**从左到右**判断是否是以指定的字符，在指定的位置开头，返回true或者false

**格式：**

(item:包含的字符串,from:起始指定位置**包含**这个指定位置):

item:参数为必填

from:参数为选填，表示开始搜索的位置。

当不填写时默认为字符串从左到右的第一位;

从左到右依次为0,1,2,3,4......(从0开始数起)

#### 3.2 endsWith(ES6)

**返回值：**

**从右到左**判断是否是以指定的字符，在指定的位置结束，返回true或者false

**格式：**

(item:包含的字符串,to:结束指定位置但**不包含**这个指定位置)

item参数为必填

to参数为选填，当不填写时默认为字符串的最后一位。也就是说，使用第二个参数n时，针对前n个字符。

例如：

var a = "badbb";

console.log(a.endsWith("a",1)); // 返回true

这三个方法都支持第二个参数，表示开始搜索的位置。

let s = 'Hello world!';

s.startsWith('world', 6) // true

s.endsWith('Hello', 5) // true

s.includes('Hello', 6) // false

上面代码表示，使用第二个参数n时，endsWith的行为与其他两个方法有所不同。它针对前n个字符，而其他两个方法针对从第n个位置直到字符串结束。

### 分割合成类

#### split

**返回值：**

一个字符串数组。该数组是通过在 separator 指定的边界处将字符串 stringObject 分割成子串创建的。返回的数组中的字串不包括 separator 自身。

但是，如果 separator 是包含子表达式的正则表达式，那么返回的数组中包括与这些子表达式匹配的字串（但不包括与整个正则表达式匹配的文本）

**格式：**

stringObject.split(*separator*,*howmany*)

|  |  |
| --- | --- |
| **参数** | **描述** |
| separator | 必需。字符串或正则表达式，从该参数指定的地方分割 stringObject。  如果没有，就讲字符串转为一个长度为1的数组。 |
| howmany | 可选。该参数可指定返回的数组的最大长度。如果设置了该参数，返回的子串不会多于这个参数指定的数组。如果没有设置该参数，整个字符串都会被分割，不考虑它的长度。 |

string.split(); //将字符串的string转换为数组的string

string.split(“”); //用一个空字符串分割string,即把字符串中的每个字符都分割为数组中的一项。

string.split('-');

**以某些字符为依据**，把**字符串分割**出来，**放到数组中。**

返回值是数组。

##### 1.空字符串为分割符

如果用空字符串为分割符，那么会把字符串的**每项**分割出来放到数组中。

str = 'miaov'

str.split('') -> ['m','i','a','o','v'];

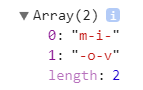
##### 2.字符串为分隔符

如果用字符串作为分隔符，那么会返回一个双字符串的数组

例如：

let str = 'm-i-a-o-v';

console.log(str.split('a'));

//打印一个数组

##### 3.空字符串分割空字符串

只有空字符串分割空字符串才会是个空白数组[]

##### 注意

一般会和数组的join合用

### 编码转换类

#### 1.charCodeAt

可以把指定的**字符转成unicode编码值**。

这个用的非常少。

string.charCodeAt(index);

第一种写法:

'm'.charCodeAt(); 109

第二种写法:

'miaov'.charCodeAt(0) 109

'miaov'.charCodeAt(1) 105

#### 2.fromCharCode

可以把指定的unicode编码值转成字符。

### 大小写转换类

#### 1.toLowerCase

**返回值：**

一个**新的**字符串，在其中 stringObject 的**所有大写字符**全部被转换为了小写字符。

**格式：**

stringObject.toLowerCase()

**注意！ 这个方法不带参数！！！！**

**谁要转换小写谁在这个方法前面放下并加‘· ’**

例如：

<script>

let str = 'MIAO';

console.log(str.charAt(0) + **str.slice(1).toLowerCase());**

//打印Miao

</script>

#### 2.toUpperCase

**返回值：**

一个**新的**字符串，在其中 stringObject 的所有小写字符全部被转换为了大写字符。

**格式：**

stringObject.toUpperCase()

**注意！ 这个方法不带参数！！**

**谁要转换大写谁在这个方法前面放下并加‘· ’**

### 增删类

#### 1.padStart

**返回值：**

从左到右（往前补全）返回一个补齐的字符串。

**格式：**

stringObject. padStart (length,what)

length: 补齐的长度（num）

what: '补全的内容'

第一个参数length用来指定字符串的**最小**长度。

如果原字符串的长度，**等于或大于**指定的最小长度，则返回原字符串。

第二个参数what是用来**补全的字符串**。

如果用来补全的字符串与原字符串，两者的长度之和超过了指定的最小长度，则会截去超出位数的**补全**字符串。

如果省略第二个参数，默认使用**空格**补全长度。

**用例：**

padStart的常见用途是为数值补全指定位数。下面代码生成 10 位的数值字符串。

'1'.padStart(10, '0') // "0000000001"

'12'.padStart(10, '0') // "0000000012"

'123456'.padStart(10, '0') // "0000123456"

另一个用途是提示字符串格式。

'12'.padStart(10, 'YYYY-MM-DD') // "YYYY-MM-12"

'09-12'.padStart(10, 'YYYY-MM-DD') // "YYYY-09-12"

#### 2.padEnd

**返回值：**

从右到左（往后补全）返回一个补齐的字符串。

**格式：**

stringObject. padEnd (length,what)

length: 补齐的长度（num）

what: '补全的内容'

第一个参数length用来指定字符串的**最小**长度。

如果原字符串的长度，**等于或大于**指定的最小长度，则返回原字符串。

第二个参数what是用来**补全的字符串**。

如果用来补全的字符串与原字符串，两者的长度之和超过了指定的最小长度，则会截去超出位数的**补全**字符串。

如果省略第二个参数，默认使用**空格**补全长度。

**举例：**

let str = '12';

str = str.padEnd(5,'.');

console.log(str); //打印12...

#### 3.trim

|  |  |  |  |
| --- | --- | --- | --- |
| trim | 英[trɪm] | 美[trɪm] | |
| vt. | 装饰; 修剪; 整理; | |

|  |  |
| --- | --- |
| adj. | 整齐的，整洁的; 修长的; 苗条的; |

|  |  |
| --- | --- |
| n. | 修剪; 整齐; 健康状态; 装束; |

[例句]The neighbours 'gardens were trim and neat.

邻居们的花园整洁美观。

|  |  |
| --- | --- |
| [其他] | 比较级：[trimmer](http://www.baidu.com/link?url=dMgHRf2TZZeJmDUT8vH8xQSr9bu2mJp5LKk-8QdK11PgEpr5yeD78dEElFvJrRsau2cWpo_6cj51p5k0s2eniSybnRKYc752bQdqYQ269oO" \t "_blank) 最高级：[trimmest](http://www.baidu.com/link?url=gJrdGEMDlCF_Hsg3fiNI9zQjpwdPWkLkTwGL7kmCQyFNRKYj30GcYrYgME3UvZOs908JRx17Dme92CQfmTVbzOS5EGWlTnCSLzVun4RggIi" \t "_blank) 第三人称单数：[trims](http://www.baidu.com/link?url=8lF9Mhs6RUKJPJijAbcaG7t_sgG8YBcGBaHs2dTWMIPnw1hSGshAXiekMNUjL-G_S0JotQH0BqKtd6H7DSm0f9fcAPrMEIBQyHx5sqdGJVS" \t "_blank)现在分词：[trimming](http://www.baidu.com/link?url=gJrdGEMDlCF_Hsg3fiNI9zQjpwdPWkLkTwGL7kmCQyFNRKYj30GcYrYgME3UvZOs908JRx17Dme92CQfmTVbzXdsCP69bwJ1klqk_rHGfoS" \t "_blank) 过去式：[trimmed](http://www.baidu.com/link?url=dMgHRf2TZZeJmDUT8vH8xQSr9bu2mJp5LKk-8QdK11PgEpr5yeD78dEElFvJrRsau2cWpo_6cj51p5k0s2eniK9zreuND60DCXf0r090EI_" \t "_blank) 过去分词：[trimmed](http://www.baidu.com/link?url=dMgHRf2TZZeJmDUT8vH8xQSr9bu2mJp5LKk-8QdK11PgEpr5yeD78dEElFvJrRsau2cWpo_6cj51p5k0s2eniK9zreuND60DCXf0r090EI_" \t "_blank) |

删除字符串开始和末尾的空格

**返回值：**

trim() 方法并不影响原字符串本身，它返回的是一个新的字符串

**格式：**

stringObject.trim()

stringObject.trim()函数会移除字符串开始和末尾处的所有换行符，空格(包括连续的空格)和制表符。

如果这些空白字符在字符串中间时，它们将被保留，不会被移除。

**例如：**

var str = ' 今天天气不错 ';

console.log(str.trim());

#### 4.repeat(ES6)

返回值：

返回一个新字符串，表示将原字符串重复n次。

格式：

string.repeat(num);

num:复制多少次

注意:

如果num传0，那么返回 '';

如果num是个小数，那么会向下取整 'na'.repeat(2.9) // "nana"

如果repeat的参数是负数或者Infinity，会报错。

如果参数是一个字符串型的数字，那么还是会复制，不然为'';

例如1：

let str = '<div>span</div>';

box.innerHTML = str.repeat('5');

例如2：

console.log( 'na'.repeat(2.9) );

#### 5.replace

replace() 方法用于在字符串中用一些字符替换另一些字符，或替换一个与正则表达式匹配的子串。

**返回值：**

一个新的字符串，是用 replacement 替换了 regexp 的第一次匹配或所有匹配之后得到的。

**格式：**

string.replace(字符串||正则,字符串||cb)

第一个参数：

替换谁

第二个参数:

替换成什么

**cb:默认的前3个参数**

$0 : 当前正则匹配到的字符（没有匹配到的就不显示）

$1 : 当前正则匹配到的字符索引

$2 : 整个字符串

$3 undefined

$4 undefined

****例子：var url = "http://localhost:64177/Home/AccordionIndex";****

****将斜杠转换成反斜杠：****

****url = url .replace("\/\/", "\\\\");  
url = url .replace("\/", "\\");  
url = url .replace("\/", "\\");****

****alert(url);****

****将会弹出：http:\\localhost:64177\Home\AccordionIndex****

****将反斜杠转换成斜杠：****

****var strurl = "http:\\localhost:64177\Home\AccordionIndex";****

****strurl = strurl .replace("\\\\", "\/\/");  
strurl = strurl .replace("\\", "\/");  
strurl = strurl .replace("\\", "\/");  
alert(strurl );****

****将会弹出：http://localhost:64177/Home/AccordionIndex****

# 数组

## 数组属性

Array：

简写: []

标准:new Array

一般数组的方法是直接操作本身

一般字符串是不会操作本身的，一般要使用返回值

## 长度length

**可读可写**

清空数组:

arr.length = 0;

通过下标去获取数组中的每个值。

可以使用for in循环 ，但是性能不高

数组为对象类型，里面可以放所有的数据类型。

一般数组的方法是直接操作数组本身，而字符串一般是不会操作

本身的，一般需要使用返回值（赋值返回值）。

## 数组方法

**改变原数组和返回值是两个概念！！！**

### 截取类

#### slice();

经常会用到！！！！！

数组的截取方法

**不会改变原数组**

**返回值：**

从已有的数组中返回选定的元素，组成的**新数组**。

包含从 start 到 end （不包括该元素）的 arrayObject 中的元素。

**格式：**

arrayObject.slice(start,end)

start： 必需。规定从何处开始选取。如果是负数，那么它规定从数组尾部开始算起的位置。也就是说，-1 指最后一个元素，-2 指倒数第二个元素，以此类推。

end 可选。规定从何处结束选取（**不包含**）。该参数是数组片断结束处的数组下标。

如果没有指定该参数，那么切分的数组包含从 start 到数组结束的所有元素。

如果这个参数是负数，那么它规定的是从数组尾部开始算起的元素。

slice(起始位置,结束位置但不包含结束位置):

#### splice();

|  |  |  |
| --- | --- | --- |
| splice | 英[splaɪs] | 美[splaɪs] |
| vt. | 绞接; 捻接（两段绳子）; 胶接; 粘接（胶片、磁带等）; | |

|  |  |
| --- | --- |
| n. | 胶接处，粘接处，铰接处; |

[例句]He taught me to edit and splice film

他教我电影剪接。

|  |  |
| --- | --- |
| [其他] | 第三人称单数：[splices](http://www.baidu.com/link?url=R8MWs3LHZZisD7G5yGAyWIuCCcDqwugmdxGCQadX8Lk0rC0_JoH2iLt9gHyePlPMxVUyXgDrhpZ4KPzlGJHvENAiYFZsE8-RejDIYVACG9y" \t "_blank) 现在分词：[splicing](http://www.baidu.com/link?url=h-0R4qaZwBodaQvVcYl5b4d0bfy0E2BxlaUJML2jJa3--9uPEwkQ7TQTeUEqjecx-KxZsZb2xX6wFLNVZOVBCX4-BSY7aIgRCupi5trTUmy" \t "_blank) 过去式：[spliced](http://www.baidu.com/link?url=R8MWs3LHZZisD7G5yGAyWIuCCcDqwugmdxGCQadX8Lk0rC0_JoH2iLt9gHyePlPMxVUyXgDrhpZ4KPzlGJHvEULOIERwocZJk9Q7hNvqzqi" \t "_blank)过去分词：[spliced](http://www.baidu.com/link?url=R8MWs3LHZZisD7G5yGAyWIuCCcDqwugmdxGCQadX8Lk0rC0_JoH2iLt9gHyePlPMxVUyXgDrhpZ4KPzlGJHvEULOIERwocZJk9Q7hNvqzqi" \t "_blank) |

概念：

添加，删除，替换数组中的元素。

**会改变原数组**

**返回值：**  
返回包含被删除项目的**新数组**。

如果从 arrayObject 中删除了元素，则返回的是含有被删除的元素的新数组。

如果没有删除元素，则返回一个空数组。

**格式：**

*array*.splice(from,howmany,item1,.....,itemX)

|  |  |
| --- | --- |
| **参数** | **描述** |
| *from* | 必需。规定从何处添加/删除元素。 该参数是开始插入和（或）删除的数组元素的下标，必须是数字。 |
| *howmany* | 必需。规定应该删除多少元素。必须是数字，但可以是 "0"。 如果未规定此参数，则删除从 index 开始到原数组结尾的所有元素。 |
| *item1*, ..., *itemX* | 可选。要添加到数组的新元素 |

splice(a,a-,a+)

参数解释：

第一个**a**

参数操作位置（包含被操作）

第二个**a-**

参数如果写非0的正数字，那么为删除,

数字是几就删除几个,返回值为删除的内容（数组）

如果为0,说明一个都不删，返回空白数组

第三个或三个之后的参数**a+**

添加的数据

#### splice与slice的区别

**slice** **不会直接改变原数组**的。返回的是被提取的新数组。

**splice 会改变原数组。** 返回的是删除元素组成的新数组。

### 查询类

#### includes

**格式：**

arr.includes(searchElement, fromIndex)

查看数组中是否包含某个数据。

**返回值：**

布尔类型

参数：

**searchElement**

需要查找的元素值

**fromIndex 可选**

从该索引处开始查找 searchElement。如果为负值，则按升序从 array.length + fromIndex 的索引开始搜索。默认为 0。

#### indexOf

查询数组中的下标

**返回值：**

返回某个指定的字符串值在数组中首次出现的位置。

**格式：**

ArrayObject.indexOf(searchvalue,fromindex)

indexOf() 方法可返回数组中某个指定的元素位置。

该方法将从头到尾地检索数组，看它是否含有对应的元素。开始检索的位置在数组 start 处或数组的开头（没有指定 start 参数时）。如果找到一个 item，则返回 item 的第一次出现的位置。开始位置的索引为 0。

如果在数组中没找到指定元素则返回 -1。

## 实例

查找数组中的 "Apple" 元素：

var fruits = ["Banana", "Orange", "Apple", "Mango"]; var a = fruits.indexOf("Apple");

a 结果输出：

2

### 分割合成类

#### join

**返回值：**

返回一个字符串。该字符串是通过把**数组**的每个元素转换为字符串，然后把这些字符串连接起来，在两个元素之间插入 separator 字符串而生成的。

**格式：**

arrayObject.join(*separator*)

|  |  |
| --- | --- |
| **参数** | **描述** |
| separator | 可选。指定要使用的分隔符。  **数组的每个元素合成字符串要用到的字符。**  **如果省略该参数，则默认使用逗号作为分隔符**。 |

##### 字符为链接符

以字符为链接符，链接数组的每项，并且转成字符串。

例如：

<script type="text/javascript">

var arr = new Array(3)

arr[0] = "George"

arr[1] = "John"

arr[2] = "Thomas"

document.write(arr.join("."))

</script>

输出：

George.John.Thomas

##### 不写连接符

如果不写连接符，就是把这个数组直接转成字符串并且带,号。

<script type="text/javascript">

var arr = new Array(3)

arr[0] = "George"

arr[1] = "John"

arr[2] = "Thomas"

document.write(arr.join())

</script>

输出：

George,John,Thomas

##### 注意

一般会和字符串的split合用。

**小技巧**：如果要转字符串的时候不带,号，在**join中就写空字符串 ''**。

### 增删类

**增加的**都是返回数组**长度number**,

**删减的**都是返回删除的那个**元素**

#### push();

**返回值：**

**数组的新长度，**类型为**number**类型

**格式：**

arrayObject.push(newelement1,newelement2,....,newelementX)

**参数 描述**

**newelement1 必需。要添加到数组的第一个元素。**

**newelement2 可选。要添加到数组的第二个元素。**

**newelementX 可选。可添加多个元素。**

push() 方法不创建新的创建，而是直接修改原有的数组。

#### pop();

**改变原数组，使原数组的长度减去1**

**返回值:**

返回数组的**最后一个元素**。

**格式：**

arrayObject.pop()

1括号里不需要放任何参数，不管写什么都是删除最后一个。

2如果数组已经为空，则 pop() 不改变数组，并返回 undefined 值。

#### shift();

往数组首位删除1个,并**改变原数组**以及它的长度

**返回值：**

方法用于把数组的第一个元素从其中删除，并**返回第一个元素的值**

|  |  |
| --- | --- |
| **类型** | **描述** |
| 任何类型（**\***） | 数组原来的第一个元素的值（移除的元素）。 |

如果数组是空的，那么 shift() 方法将不进行任何操作，返回 undefined 值。

**格式：**

array.shift()

括号里不需要放任何参数

#### unshift();

往数组首位添加1个或者多个

unshift() 方法不创建新的创建，而是直接修改原有的数组。

**返回值:**

**新数组的长度**。类型为**number**类型

**格式：**

arrayObject.unshift(newelement1,newelement2,....,newelementX)

**参数 描述**

**newelement1 必需。要添加到数组的第一个元素。**

**newelement2 可选。要添加到数组的第二个元素。**

**newelementX 可选。可添加多个元素。**

### ES5类

**因为函数可以套函数，所以方法可以套方法！**

#### every(function (e,i,all));

含义：

//查看数组中每个是不是都是统一的值，如果是返回true,否则false

格式：

arr.every(function(e,i,all){

})

返回一个布尔值。

#### some(callback(e,i,all));

含义：

//查看数组中有没有设置好的值，如果有就返回true，没有就返回false。

格式：

arr.some(function(e,i,all){

})

返回一个布尔值

#### forEach(callback(e,i,all));

**注意：forEach会改变原数组（而map不会改变原数组）**

**返回值**

**undefined !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!**

array.forEach() 数组的循环 ES5的方法

有2个参数:

第一个参数：

回调函数

回调函数中还有3个参数

1.循环中的每一个 arr[i]

2.循环中每一个的索引（下标）i

3.整个数组 arr

第二个参数:

改变this指向(写啥this就是啥)

默认this为window

**注意：**

没有break也没有continue

如果要有continue那么可以使用return，但是类似break的东西就没有了

**不能用return!!!**

**其他几个方法都要用return**

举例：

let mx = -Infinity;

arr.forEach(function(e){ // **for(var i=0;i<arr.length;i++){**

if(e > mx){ //  **if(arr[i] > mx){**

mx = e; // **mx = arr[i];**

}

});

#### map(callback(e,i,all));

map() 方法返回一个**新数组**，数组中的元素为原始数组元素**调用函数处理**后的值。

map() 方法按照原始数组元素顺序**依次处理元素**。(**也就是会遍历每一个元素**)。

**注意：** map() 不会对空数组进行检测。

**注意：** map() 不会改变原始数组。

注意：大小写！map ≠Map

**arr.map((e,i) = > {**

**return “123”;**

**})**

相等于：

**arr.map((e,i) = >**

**”123”**

**)**

#### filter(callback(e,i,all));

arr.filter(callback(e,i,all))

**不改变原数组,只会把过滤结果返回成一个新数组。**

**返回值**

**e的返回值是什么，filter的返回值就是什么。**

(过滤掉回调函数中条件不符合的（条件符合就留下）

**return后面跟布尔值**，是什么就留下什么

查看每条数据，某条数据是否条件成立，把不成立的过滤掉，返回一个新数组。)

例如：

let arr = [

{name:'衣服',num:20,checked:false},

{name:'裤子',num:30,checked:true},

{name:'袜子',num:5,checked:false}

];

console.log(arr.filter(function(e,i){

return !e.checked; //这里reurn后面跟的是checked:false的，那么就返回的是只有“衣服”，“袜子”这两个元素的数组。

}));

上面的例子，用箭头函数写就是：

console.log(arr.filter(e => !e.checked))

也可以写成：

console.log(arr.filter(e => { return !e.checked }))

filter()基本语法：

arr.filter(callback[, thisArg])

　　filter()参数介绍：

　　 参数名 说明

　　 callback 用来测试数组的每个元素的函数。调用时使用参数 (element, index, array)

**返回true表示保留该元素（通过测试），false则不保留。**

　　thisArg 可选。执行 callback 时的用于 this 的值。

### ES6类

#### find(callback(e,i,all))

含义：

find() 方法返回传入一个测试条件（函数）符合条件的数组**第一个元素**。

**返回值：**

find() 方法返回传入一个测试条件（函数）符合条件的数组**第一个元素**。

find() 并没有改变数组的原始值。而是把返回值放到一个新数组中。

find() 方法为数组中的每个元素都调用一次函数执行：

当数组中的元素在测试条件时返回 *true* 时, find() 返回符合条件的元素，之后的值不会再调用执行函数。（也就是说只有第一个复合条件的值会被返回）。

如果没有符合条件的元素返回 undefined。

查看数组中的name有没有裤子，有裤子就把这条数据取出来

        var arr = [10,20,60,30,40,50];

        var res = arr.find( function( item ){

            console.log(item)

            return item > 50

        } )

        console.log( res )

#### findIndex(callback(e,i,all))

在数组中找到条件成立数据的下标

        var arr = [10,20,60,30,40,50];

        var res = arr.findIndex( function( item ){

            console.log(item)

            return item > 50

        } )

        console.log( res )

#### Array.from();

把类数组转换成数组，例如：

<body>

<ul>

<li>1</li>

<li>2</li>

<li>3</li>

<li>4</li>

</ul>

<script>

let lis = document.getElementsByTagName('li');

lis = **Array.from(lis)**; //从类数组变成了数组

</script>

**把类数组转真数组的其他方法**

lis = [].slice.call(lis);

#### Array.of()

创建数组

        var arr = Array.of(10,20,30)

        var arr = Array.of(10)

### sort(function(a,b){})

#### 从小到大a-b/从大到小b-a

排序

不用管那么多，**a-b就是从小到大，b-a就是从大到小。**

**正数是变，负数是不变**

**格式：**

arr.sort(function(**a,b**){

return **a.text[0] – b.text[0];**

}

return后面的两个相减的值必须为数字类型。

如果排序的数组本身就是数字，那么就可以{return a-b}/{return b-a}排序。

如果潘旭的数组是对象等非数字类型，那么return后面的必须提取出对象中的数字来相减。例如：return **a.text[0] – b.text[0];** 也可以理解为用要排序的属性值来相减。

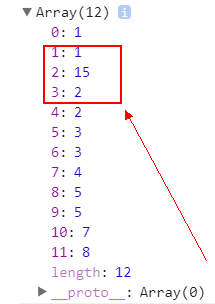
**排序方式：**

默认的排序方式是依据字符串比较的方式来排序的。

例如：

let arr = [3,5,5,2,1,8,7,4,3,1,2,15];

console.log(arr.sort()); //打印：

 //字符串比较大小时，只比较每个字符串**第一位的值**！例如：alert('11' > '2');//false

#### 从小到大

默认**从小到大**排列，

arr.sort(function(**a,b**){

return **a-b;**

}

这里的a是一个较小的值，b是一个较大的

let arr = [3,5,5,2,1,8,7,4,3,1,2,15];

console.log(arr.sort(function(a,b){

return a-b; } //从小到大

//这里的原理就是在数组里面抽出两个数字来，

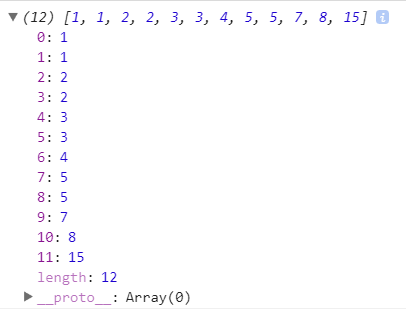
例如a=3 b=5.

因为a-b是一个**负数**，所以就不调换位置，

然后就对每两个数字进行这样的判断，

最终得到一个**从小到大**的排列

}));



#### 从大到小

arr.sort(这里面可以接收一个回调函数)，例如

let arr = [3,5,5,2,1,8,7,4,3,1,2,15];

console.log(arr.sort(function(**a,b**){

return **b - a;**

//这里的原理就是在数组里面抽出两个数字来，

例如a=3 b=5.

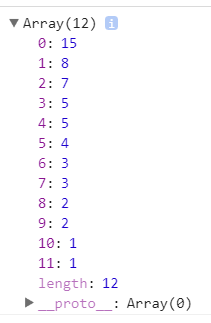
因为b-a是一个**正数**，所以就调换位置，

调换了位置之后，对两个数字进行循环，

最终得到一个**从大到小**的排列

}));

//从大到小排列打印



#### 随机排序

let arr2 = [1,2,3,4,5,6,7];

// //0.2 - 0.5 //随机排序

console.log(arr2.sort(function(a,b){

return Math.random() - 0.5 ; //这里面有可能是负数，有可能是正数，所以会随机排列

}))

#### 中文排序localeCompare方法

1.首先全部转换为大写或者小写字母

2. 使用localeCompare 比较大小

localeCompare方法格式

str.sort (function(a,b){return a.localeCompare(b)})

例如：

value.sort(function(a, b){

var letterA = utils.convertToABC(a.name).pinyin;

var letterB = utils.convertToABC(b.name).pinyin;

return letterA.localeCompare(letterB);

});

### concat

1.拼接两个数组

2.为数组中添加元素。

3.当参数为空时，可以复制数组。

**返回值**

返回一个新的数组。原来的数组不变

该数组是通过把所有 arrayX 参数添加到 arrayObject 中生成的。

如果要进行 concat() 操作的参数是数组，那么添加的是数组中的元素，而不是数组。

**语法**

arrayObject.concat(arrayX,arrayX,......,arrayX)

例如：

利用“返回一个新的数组。原来的数组不变”这个特性，可以复制数组，而原来的数组不变。

let arr2 = arr.concat();

### reverse

翻转数组

**返回值：**

该方法会改变原来的数组，而不会创建新的数组。

**语法**

arrayObject.reverse()

不包含任**何**参数。

### reduce

方法接收一个函数作为累加器（accumulator），数组中的每个值（从左到右）开始缩减，最终为一个值。

不改变原数组，返回一个callback的返回值

**语法：**

arr.reduce(callback, initialValue)

**参数**

**callback**

执行数组中每个值的函数，包含四个参数:

**previousValue**

上一次调用回调函数返回的值，或者是提供的初始值（initialValue）

**currentValue**

数组中当前被处理的元素

**currentIndex**

当前被处理元素在数组中的索引, 即currentValue的索引.如果有initialValue初始值, 从0开始.如果没有从1开始.

**array**

调用 reduce 的数组

**initialValue**

可选参数, 作为第一次调用 callback 的第一个参数。

**reduce是如何工作的**

例如执行下面的代码

[0, 1, 2, 3, 4].reduce(function(previousValue, currentValue, index, array){

return previousValue + currentValue;

});

**举例：**

        let arr = [1,2,3,4,5,6];

        function total (a,b){

            return a+b

        }

        console.log(arr.reduce(total));

### Array.isArray（arr）

判断是不是一个数组

let arr =[1,2,3];

Array.isArray(arr) ; //true

## 数组转字符串方法

1.stringify 转成JSON，也就是字符串。

2.join数组方法

# 对象

## 对象基础

ECMAScript:标准

数据类型：

1. ''或者"" 字符串

2. {} 对象 (存储数据的) Object

对象:

{

key:value

}

当key值和value值相同的时候，只用写一个key值即可表示。

例如data:{message} 就相当于 data:{message:message}

例如1：

box = {

style:xxx,

}

style是obj的key

xxx是obj的value

例如2：

let obj = {

style:{

width:'150px'

}

}

key值也是一个字符串,所以要套“”引号。（**key值必须为字符串类型**）

obj[‘style’]

另外一种写法： obj.style

## 对象的key值和value值相同

当key值和value值相同的时候，只用写一个key值即可表示。

例如data:{message} 就相当于 data:{message:message}

## 1.什么是对象？

1.数组[]，

2.函数function(){}，

3.对象{}

{} //大括号

[] //中括号

标准写法 {"name":"小青"}

js {'name':'小青'}

## 对象里面的第几个

let obj = {key1:value1,key2:value2,key3:value3};

obj[key1] 这表示对象的第一个

obj[key2] 这表示对象的第二个

obj[key3] 这表示对象的第三个

## 2.什么是json?

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。它使得人们很容易的进行阅读和编写。同时也方便了机器进行解析和生成

json是一种数据格式！

**1.有数组形式的json数据。较多用。**

**2.有对象形式的json数据。**

**这种格式里面的文件里面不能写注释！！！！**

'{}' //大括号加引号

'[]' //中括号加引号

标准写法 **'**{**"**name**"**:**"**小青**"**}**'**

例如1：



例如2：

list:[

{

ranknum:1,

name:"Google",

score:99,

},

{

ranknum:2,

name:"BaiDu",

score:98,

},

{

ranknum:3,

name:"YouTube",

score:97,

},

{

ranknum:4,

name:"新浪",

score:96,

},

{

ranknum:5,

name:"FaceBook",

score:95,

},

{

ranknum:6,

name:"网易",

score:94,

},

{

ranknum:7,

name:"twitter",

score:93,

},

{

ranknum:8,

name:"QQ",

score:92,

},

{

ranknum:9,

name:"XIAOMI",

score:91,

},

]

## 3.‘Json’ 和 ‘对象’ 的区别

Json是字符串类型的，是简单类型的。

对象是object类型的，是复杂类型的。

Json是可以进行运算的。

对象是不可以进行运算的。

所以要将Json的数据，转换为对象才能使用。

相反，在后台给前台数据的时候，或者是数据传输的时候，要将object类型的对象数据，转换为Json类型的数据。

## 方法

### json 转 对象

JSON.parse();

注意:

json必须要是标准格式的,如果不是标准格式就会报错。

里面是双引号，外面再包一个单引号

例如：

let json = '{"name":"小青"}';

console.log(JSON.parse(json)); //打印{name: "小青"}

let json = "{'name':'小青'}";

console.log(JSON.parse(json)); //这样就报错，因为**" "**。

如果要转换不标准的JSON就用**eval**,或者**newFunction**

#### eval

作用：一般使用的场景

**1.把一个不标准的json转成对象**

2.低版本浏览器用不了JSON.parse,JSON.stringify，所以用eval

eval(); 容易被注入病毒。

eval() 函数可计算某个字符串，并执行其中的的 JavaScript 代码。

尽可能把能够执行的js代码执行，把字符串转js

**语法**

eval(string)

string: 必需。要计算的字符串，其中含有要计算的 JavaScript 表达式或要执行的语句

**返回值**

通过计算 string 得到的值（如果有的话）。

**例如：**

<script type="text/javascript">

eval("x=10;y=20;document.write(x\*y)")

document.write(eval("2+2"))

var x=10

document.write(eval(x+17))

</script>

输出：

200

4

27

#### newFunction

把不标准的JSON转成对象

let fn = new Function('return' + "{code:0,msg:'有介个银!'}");

console.log(fn())

console.log(typeof(fn())); //object

打印：



### 对象 转 json（字符串）

JSON.stringify();

对象(也就是object)

注意:

stringify不能转函数和undefined的

例如：

let obj = {name:'小薛',fn:function(){},u:undefined,nn:null};

console.log(JSON.stringify(obj));

//打印{"name":"小薛","nn":null}，因为函数和undifined就是不能转json的

### 对象 转 数组、字符串（表单序列化）

let obj = {

name:'chaoxue',

pass:123

}

let arr = [];

for(let attr in obj){

arr.push( attr +”=”+obj[attr] )

}

[name=chaoxue,pass=123]

console.log(arr.join(“&”))

name=chaoxue&pass=123

#### 1.方法一：JSON.parse(JSON.stringify数据(arr));

arrCopy = JSON.parse(JSON.stringify(arr));

#### 2.方法二：利用for in +递归+push

利用for in +递归+push来深度克隆

### toString

只要尝试去alert，会默认调用该数据的toString方法.即转字符串。

toString -> 转成string

所有的数据自身都有toString方法

toString可以转除了10进制的进制数

变量.toString()

不能用 数字.toString() 会报错的。

arr -> 1,2,3

#### 数字转进制

可以转进制。

输入的参数为要转的进制数，**10不能作为参数**！

例如：var a = 123;

a = a.toString( );

console.log (a);

console.log (typeof a);

number.toString(进制数) -> 把number转成指定的进制数

let num = 116;

console.log(num.toString(2));

#### 对象.toString -> [object Object]

object.toString -> **[object Object]**  第二个Object是当前数据的内置对象为Object

Let Arr = [

a:1,

B:2,

C:3,

]

Let b = Arr.toString()

console.log(b); [**object Object**]

let a = JSON.stringify(Arr).toString()

console.log(a); [{"a":"1","B":"2","C":"3"}]

如上面的例子：

如果Arr存的是一个对象，那么内部会调用toString方法,就会变成[object Object],所以要先用JSON.stringify转一下

# 面向对象

方法挂在原型下面是为了节约性能

js面向对象:

把相同部分的代码提取出来封装成一个函数（类）

描述这个类型特性的**方法**挂在它的（类）原型上的一种编程方式

抽象：抽离出像的部分

抽象：封装函数

三大特征:

抽象

封装

继承

多态

之前我们写的代码叫 面向过程 （关注如何实现）

现在我们写的代码叫 面向对象 （谁来实现）

## 什么是构造函数？

构造函数，构造对象的函数,也就是“工厂函数”。

### 1.内置构造函数和自定义构造函数

#### 自定义构造函数（工厂函数）

下面这个function Person就是工厂函数

function Person(name,age){

var obj = {}; //原材料

obj.name = name; //加工

obj.age = age;

obj.say = function(){

alert(this.name);

}

return obj; //出厂

}

let p = new Person('一刀',8);

p.say();

#### 内置构造函数

var o = new Object(); //这里的Object()就是内置构造函数

var a = new Array(); //这里的Array()也是内置构造函数

#### 构造函数与普通函数的区别：

其一，实际上并不存在创建构造函数的特殊语法，其与普通函数唯一的区别在于调用方法。

**对于任意函数，使用new操作符调用，那么它就是构造函数；不使用new操作符调用，那么它就是普通函数。**

其二，按照惯例，我们约定构造函数名**以大写字母开头**，普通函数以小写字母开头，这样有利于显性区分二者。例如上面的new Array()，new Object()。

其三，使用new操作符调用构造函数时，会经历(1)创建一个新对象；(2)将构造函数作用域赋给新对象（使this指向该新对象）；(3)执行构造函数代码；(4)返回新对象；4个阶段。

以上的区别总结一下两句话：

面向对象的规则:1.**构造函数的首字母大写**

（仅仅是构造函数(类)名首字母大写，文件命名不用大写）

2.实例化的时候使用new

### 2.检测构造函数

#### constructor

通过实例化对象的constructor指向构造函数（意义是这样的，但是不一定准确）。

constructor:

obj.constructor = obj的构造函数

例如：

    <script>

        let a = [];

        console.log(a.constructor ) //ƒ Array() { [native code] }

console.log(a.constructor == Array) //true

    </script>

constructor容易被修改。

只要构造函数的原型被赋址，那么constructor就一定会被修改

如果被修改，需要手动去修正constructor指向

## 创建对象

创建对象也叫‘实例化对象’。

实例化对象就是创建对象的过程！

例如：

function teacher(name,age,t){

var name = name;

var age = age;

var t = t;

return {name,age,t,

say:function(){

alert(this.name);

}

}

}

let t1 = teacher('leo',80,'数学'); t1是teacher的实例化对象

t1.say();

#### 对象直接量

例如:

var obj = {};

var obj2 = {a:1,b:2}

var obj3 = {

"a 1":1,

"a-2":2

}

#### 通过new创建对象

var o = new Object(); //创建一个空对象，和{}一样

var a = new Array(); //创建一个空数组，和[]一样

var d = new Date();

var r = new RegExp('js');

#### 3.Object.create()

**该方法用于创建一个新的对象，第一个参数是这个对象的原型。**

var o1 = Object.create({x:1,y:2});

**创建一个不继承任何属性的对象**

var o2 = Object.create(null); //创建一个不继承任何属性的对象

**创建一个普通的空对象，和{}，new object()一样。**

var o3 = Object.create(Object.prototype)

## 面向对象的规则

构造函数的首字母大写

实例化的时候使用new

1.new 函数是一元运算符，**专门用来运算函数的**。

2.new的时候可以**不用加括号**，**加括号**的情况**为了传参。**

3.new完之后构造函数中的**this指向它的实例化对象**

例如：

let fn = function (){

console.log(this); //new完之后构造函数中的this指向它的实例化对象。

如果没有new,这个this就是window

}

let f = new fn; //这个是new完之后。

let f = fn(); //这个是没有new。

4.new完之后默认的返回值变成了实例化对象

如果有return，要看return后面是什么数据类型，

如果是简单类型，那么返回实例化对象

如果是复合类型，那么就返回这个复合类型

let fn = function (){

return function fn3(){}; //这里是复合类型

// return false; //这里是简单类型

}

let f = new fn;

log(f); //如果是复合类型，那么就返回这个复合类型fn3

//如果是简单类型，那么返回实例化对象fn

### new操作符具体干了什么

new操作符具体干了什么呢?其实很简单，就干了三件事情。

var obj = {};

obj.\_\_proto\_\_ = Base.prototype;

Base.call(obj);

第一行，我们创建了一个空对象obj  
第二行，我们将这个空对象的\_\_proto\_\_成员指向了Base函数对象prototype成员对象  
第三行，我们将Base函数对象的this指针替换成obj，然后再调用Base函数，于是我们就给obj对象赋值了一个id成员变量，这个成员变量的值是”base”

## 一般面向对象的写法

1.**属性**挂在构造函数中

2.**方法**挂在构造函数的原型下

3. var 对象 = **new** 构造函数();

4. **对象.方法(); //执行**

例如：

function 构造函数(){

this.属性

}

构造函数.原型.方法 = function(){};

**面向对象的使用：**

var 对象1 = new 构造函数();

对象1.方法();

## 原型和原型链

使用function定义的对象与使用new操作符生成的对象之间有一个重要的区别，这个区别就是function定义的对象有一个prototype属性，使用new生成的对象就没有这个prototype属性，存在\_\_proto\_\_。

所有的内置构造函数的（以及大部分自定义的构造函数）都具有一个继承自Object.prototype的原型

## 原型(显式原型)

每个函数function都有一个prototype,即显示原型

原型 prototype（等同于css中的class）

普通方法 （等同于css中的style）

当创建一个函数的时候，这个函数有很多的属性和方法

其中有一个属性叫prototype，它就是原型，

\*\*\*\*它的属性值为一个对象\*\*\*\*

原型的价值：

去改写对象下面公用的方法或者属性 , 让公用的方法或者属性在内存中存在一份 ( 提高性能 )

constructor -> 指向构造函数

!!! **构造函数的原型只给，该函数的实例化对象使用。**

Array.prototype.push = function(){

alert('真不是我要刀你，是ta');

}

let arr = [];

arr.push(5); //这里数组没有push5,而是弹“真不是我要刀你，是ta”

function fn(){}

就相当于：

let fn = new Function() + {} => new Object

## 原型链(隐式原型)

#### 什么是原型链？

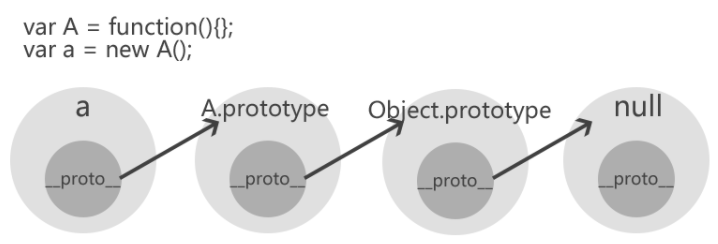
.每个实例对象都有一个\_\_proto\_\_,可称为隐式原型

对象上都有一个属性叫原型链

**\_\_proto\_\_**

原型 与 原型链的关系：

实例化对象的原型链 === 构造函数(constractor)的原型



对象是没有原型的，只有原型链

函数才有原型和原型链

看下实例化对象有没有say -> f.\_\_proto\_\_

Fn.prototype -> Fn.prototype.\_\_proto\_\_

Object.prototype

实例化对象的原型链 下面的构造函数的原型

#### 原型链用来做什么？

继承属性

## 如何用面向对象写程序

**原则**

-先写出普通的写法，然后改成面向对象写法 。

* + - * + 普通方法变型

尽量不要出现函数嵌套函数

可以有全局变量

把onload中不是赋值的语句放到单独函数中

* + - * + 改成面向对象

全局变量就是属性

函数就是方法

Onload中创建对象

改this指向问题

## hasOwnProperty

obj. hasOwnProperty(attr)

attr这个属性是不是obj\*\*自身\*\*的。

返回布尔值

prop是这个对象的属性 true,

否则false

## instanceof

**格式：**

A instanceof B:二元运算符

**含义：**

B是不是A的老爹（A必须要为一个’复合数据类型’），

(instanceof对’基本数据类型’检测不起作用，主要是因为**基本数据类型没有原型链**。)

也就是说：左值是不是右值构造出来的

**返回值：**

运算后的结果为 布尔值

一般的例子：

console.log(new String instanceof String); //true

    <script>

        let a = [];

        console.log(a instanceof Array) //true

    </script>

面向对象中比较特殊的地方（也算是bug吧）

console.log(Object instanceof Object);

## 包装对象

当去调用简单类型的属性或者方法的时候，系统会偷偷的把

这个简单类型转成对象类型，调用该对象下的属性或者方法

使用之后，自动销毁，这个过程就称为包装对象。

例如：

let str = 'hehe';

console.dir(typeof new String('hehe')); //object

## this指向

只要遇到有函数的情况下（也包括函数套函数），this非常容易被修改

**1.事件触发的元素**

2.函数直接调用 -> window

在全局环境中，this 永远指向 window。

console.log(this === window); //true

普通函数在调用时候(注意不是构造函数，前面不加 new)，其中的 this 也是指向 window。

            var x = 10;

            function foo(){

                console.log(this); //Window

                console.log(this.x); //10 ,这里指window下面的var x = 10;

            }

            foo();

3.对象下的this -> 就是对象

如果函数作为对象的方法时，方法中的 this 指向该对象。

            var obj = {

                x: 10,

                foo: function () {

                    console.log(this); //Object

                    console.log(this.x); //10

                }

            };

            obj.foo();

4.**定时器下的this指向window**

5.构造函数下的this -> 即将 new 出来的对象

所谓的构造函数就是由一个函数 new 出来的对象，一般构造函数的函数名首字母大写，例如像 Object，Function，Array 这些都属于构造函数。

            function Foo(){

                this.x = 10;

                console.log(this); //Foo {x:10}

            }

            var foo = new Foo();

            console.log(foo.x); //10

6.()=> 的this 指向主的老爹（看到箭头函数，this就指向主的老爹）

<script>

document.onclick = () => {

console.log(this); //window

}

</script>

箭头函数的this指向主的老爹

相比之下，普通函数的this就指向主

document.onclick = function(){

console.log(this); //document

}

7.undefined 严格模式下，例如：

let fn = function (){

"use strict" //这个就是所谓的严格模式

console.log(this); //undefined

}

fn();

## 严格模式

"严格模式"（strict mode）。顾名思义，这种模式使得Javascript在更严格的条件下运行。

**设立"严格模式"的目的，主要有以下几个：**

1. 消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为;

2. 消除代码运行的一些不安全之处，保证代码运行的安全；

3. 提高编译器效率，增加运行速度；

4. 为未来新版本的Javascript做好铺垫。

注：经过测试 IE6,7,8,9 均不支持严格模式。

**缺点：**

现在网站的 JS 都会进行压缩，一些文件用了严格模式，而另一些没有。这时这些本来是严格模式的文件，被 merge 后，这个串就到了文件的中间，不仅没有指示严格模式，反而在压缩后浪费了字节。

**用法:**

一般用于script标签的头部，或者函数的开始位置。

<script>

‘’use strict’’

</script>

## 改变this指向call/apply/bind

使用apply，call 与 bind 均可以改变函数对象中this的指向

1、都是用来改变函数的this对象的指向的。  
2、第一个参数都是this要指向的对象。  
3、都可以利用后续参数传参。

我们可以将call()和apply()看做是某个对象obj的方法。

某个对象obj，

通过调用这个call()或者apply()的形式,

来间接调用另外一个对象obj2的方法。

用于调用函数的时候，放在函数名之后，格式为”.call/.apply”

函数默认的this是window

使用了 call ，apply，bind之后，this指向就会发生改变

call (1,2,3,4,5)

第一个参数：

改变this指向

第二个以及后面的参数：

**不能放数组**，里面放实参

apply

第一个参数：

改变this指向

第二个以及后面的参数：

可以放**数组**，里面放实参

**call和apply的区别，call不能放数组，apply能放数组**

bind

第一个参数：

改变this指向 (bu

第二个以及后面的参数：

实参

function fn (a,b){

console.log(this,a,b)

}

fn.

let f =fn.bind(document);

f(1,2);

bind也能**改变this指向**，不过bind之后**不立即执行**，如果要执行就得执行它的**返回值**。

**例一：**

<script>

function fn (a,b){

console.log(this,a,b)

}

//fn.call(document); //这个时候是"执行"将this改为doucument

//fn.apply(document); //这个时候是"执行"将this改为doucument

fn.bind(document); //这个时候是将this改为doucument,但是不执行，只是将返回值变为document

let f =fn.bind(document); //当全局使用时，用f()这个函数接受返回值。

console.log(f());

// f(1,2);

</script>

**例二**

function fn1 (a , b) {

alert(a+b);

console.log(this); //默认这里面是window

}

fn1.call(document,1,2) //当使用了call,第一个参数就是this的指向。

call后面的实参是用属性值来表达的。

fn1.apply(document,[1,2]) //当使用了apply,第一个参数就是this的指向

apply后面的实参是用数组来表达的。

fn1.bind(document,1,2) //

this绑定之后不调用,如果要调用就是fn1.bind(document,1,2)();

也可以有多个参数，从第二个起就是实参的个数

## 继承

### 什么是继承？

子类继承父类特征，自己也有一套自己的特性。

### 继承了什么？

#### 1.属性的继承

子级对父级**属性**的继承。

#### 2.方法的继承

子级对父级**方法**的继承。

### 继承的方法有哪些？

#### 1.拷贝继承

##### 属性继承

调用父类，修改this指向。 其中修改this指向有三种方法：

ParentFunctionName. call(this,1,2,3,4,5)

ParentFunctionName. apply(this,[1,2,3,4,5])

##### 方法继承

很显然不能直接把父类的原型等于子类的原型，（这样相当于对象赋址，子级的方法一旦改变就会在子级和父级上都被用到，这就不是继承了）。

所以通过浅拷贝来继承（这样相当于对象赋值）：

高版本可以使用

**Object.assign**

例如：

Object.assign(DragSon.prototype,DragFather.prototype);

低版本可以使用

**for in**

例如：

for(let attr in DragFather.prototype){

if(DragFather.prototype.hasOwnProperty(attr)){

DragSon.prototype[attr] = DragFather.prototype[attr];

}

}

**拷贝继承中，有配置走配置，无配置走默认。**

例如：

//默认

let obj = {

name:'刘传超',

job:'前端工程师'

}

//配置

let obj2 = {

// name:'小可爱',

job:'暴民',

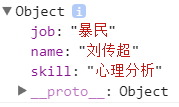
skill:'心理分析'

}

Object.assign(obj,obj2);

console.log(obj); // 默认的job就被配置的job覆盖了

显示如下：



#### 2.原型继承

##### 属性继承

调用父类，修改this指向。 其中修改this指向有三种方法：

ParentFunctionName. call(this,1,2,3,4,5)

ParentFunctionName. apply(this,[1,2,3,4,5])

##### 方法继承

利用这个函数实例化对象的地址与函数原型地址不同，

但是通过函数的实例化对象能找到原型的机制，进行方法的继承。

例如：

SonDrag.prototype = new FatherDrag ('box')

如果只用new FatherDrag,那么Drag会调用一次，但是功能是一样的。加了一个过渡函数yyd的目的就是为了少调用一次父类，就能提高性能！例如：

function yyd(){};

yyd.prototype = Drag.prototype; //把父级的原型赋值给新建函数的原型

//let yd = new yyd; //注意新建函数的实例化对象和它的原型

Drag2.prototype = new yyd;

#### 3.类式继承

##### 属性继承

##### 方法继承

## 多态

调用同一个的方法，却能得到不同的实现和结果，这种现象就是多态

## 组件化开发

## 对象方法

### **Object.keys() (ES6)**

**Object.keys()** 方法会返回一个由一个给定对象的自身可枚举属性组成的**数组**，数组中属性名的排列顺序和使用 [for...in](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/for...in" \o "for...in语句以任意顺序遍历一个对象的可枚举属性。对于每个不同的属性，语句都会被执行。) 循环遍历该对象时返回的顺序一致 。

### **Object.values() (ES6)**

### **Object.entries() (ES6)**

将对象的每一个键值对变成一个数组，多个键值对变为多个数组，最终形成一个二维数组。

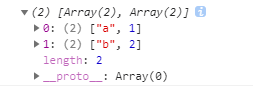
        var obj = {

            a: 1,

            b: 2

        }

        console.log( Object.entries(obj) )



### **Object.assign()(ES6)**

**Object.assign()** 方法用于将所有可枚举属性的值从一个或多个源对象复制到目标对象。它将返回目标对象。

key值相同，后面的对象覆盖前面对象的。

key值不相同，后面的增加到前面对象的后面。

        function move(obj){

            var defaultObj = {

                ease: "linear",

                duration: 1000

            }

            // var para = {

            //  ease: obj.ease || defaultObj.ease,

            //  duration: obj.duration || defaultObj.duration

            // }

            var para = {}

            Object.assign( para,defaultObj,obj ) //后面的对象覆盖前面的

            console.log( para )

        }

        move({

            duration: 2000

        })

### Object.is

判断是否相等

        console.log( Object.is( 1,1 ) )

        console.log( Object.is( 1,"1" ) )

        console.log( Object.is( NaN,NaN ) )

### 使用[]代表变量

        var attrname = "width";

        var obj = {

            [attrname]: 100,

        }

### 省略function(ES6)

非ES6写法

        var obj = {

            a: a,

            fn: function(){

                console.log( "fn" )

            }

        }

    obj.fn();

ES6写法

        var obj = {

            a,

            fn(){

                console.log( "fn执行了" )

            },

            b: 100

        };

        obj.fn();

## 类

### class

定义一个类的一种方法是使用一个**类声明。相当于声明一个函数而已。**

如果类要接收参数，在constructor中接收，相当于属性。

class Person {

constructor(name,age){

this.name = name;

this.age = age;

}

say(){

alert(this.name);

}

runing(){

alert(this.name + '会跑');

}

}

### 类的继承[extends](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Classes/extends)

[extends](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Classes/extends) 关键字在类声明或类表达式中用于创建一个类作为另一个类的一个子类。

在继承的时候，必须在constructor中写super(父类的参数)才能有this。

class Police extends Person {

constructor(name,age,job){

/\*

super写完之后才能写this

\*/

// this.job = job; //报错

super(name,age);

this.job = job;

}

say(){

alert(this.name+'是个铁狼');

}

kungfu(){

alert('会毒人,会救人,会查杀');

}

}

# ES6语法

## 对象解构赋值

1.使用对象解构赋值：

let obj = {

name:'一刀',

age:188,

job:'屠夫'

}

let {name:key,age,job} = obj;

console.log(key,age,job) //打印 ：一刀 188 屠夫

2.使用数组解构赋值：

let [a,b=2] = [1,3]; //用数组解构，必须右边也是一个数组，不然就报错。

console.log(b);

举例说明解构赋值（数组、对象）

let [b,a=6] = [a,b];

let {a:x=20} = {a:10}

注意：即便是数组或者对象中**只有一个值**，那么也要加[]或者{}.

因为使用结构赋值，左边和右边必须是相同的类型。

**2.将事件指定的元素中的属性结构赋值**

例如：

当事件的target是时，

就可以将这个<input type=’checkbox’></input>中的隐含属性 checked结构出来，并赋值。

let {checked} = ev.target;

所以checked 的值在点击checkbox时就是 => true/false ，动态变换。

## Set

这是一种数据结构。

var s = new Set([“a”,”b”,”c”])

方法1：

s.size 表示数组的长度

方法2：

.add

方法3：

.delete

方法4：

.has

方法5：

.clear

特性：

不会对其中的数组或者对象中的重复项进行添加。

## Map

这是一种数据结构。

var m = new Map( [ [“name”, ”miaov” ] ] )

console.log(m.size) ///这里是1，通过map这种数据结构将name作为key, miaov作为value

方法一：

.get

方法二;

.set

...其他方法和Set类似

## WeakMap

new WeakMap([iterable])

## for in

### 作用

遍历一个对象中的每个元素

通过for in是可以拿到**对象**中所有的key值。

### 格式

**for(var attr in obj){ }**

这里”attr”是指声明的一个变量

“obj”是指对象的名称。

对象需要特殊的格式情况下才能使用for循环

例如：

var obj = {

width:'100px', **//这里一定要用“,”逗号!!!**

height:'100px',

background:'red',

}

### in 运算符

可以运算对象的key值,查看对象中是否有这个key值

返回一个布尔值。

### 注意

数组与对象都能够使用for in循环，但是，for in的性能不高。

**对象{}目前来说只能用for in ！！！**

in不但能遍历当前对象上的属性····，原型链上的···

var obj = {

width:'100px', **//这里一定要用“,”逗号!!!**

height:'100px',

background:'red',

}

**for(var attr in obj)**{

console.log(attr); **// attr 即为对象obj的key值，即为“width/height/background”**

console.log(obj[attr]); **// obj[attr] 即为对象obj的value值**

box.style[attr] = obj[attr];

}

## for of

遍历所有数据结构的统一的方法。

for...of循环可以使用的范围包括

数组、Set 和 Map 结构、某些类似数组的对象（比如arguments对象、DOM NodeList 对象）、后文的 Generator 对象，以及字符串

for of:

keys() [key]

entries() -> [key,val]

Symbol.iterator 遍历器

只要有遍历器就可以使用for of循环

数组是没有values()

DOM对象是有values()的

## 扩展运算符...

扩展运算符以三个点的形式出现 ... 可以将数组或者对象里面的值展开。

### 将数组或者对象里面的值展开

**扩展语法**允许一个表达式在期望**多个参数**（用于函数调用）或**多个元素**（用于数组字面量）或多个变量（用于解构赋值）的位置扩展。

也就是说，将对象或者数组打散为参数的形式。

用于函数调用:

myFunction(...iterableObj);

用于数组字面量:

const [...iterableObj] = [1, 3, 5, 7, 9];

[...iterableObj, 0, 2, 4, 6, 8];

// [1, 3, 5, 7, 9, 0, 2, 4, 6, 8]

[0, 2, ...iterableObj, 4, 6, 8];

// [0, 2, 1, 3, 5, 7, 9, 4, 6, 8]

[...iterableObj, 0, 2, 4, 5, 6, 8, ...iterableObj];

// [1, 3, 5, 7, 9, 0, 2, 4, 5, 6, 8, 1, 3, 5, 7, 9]

用于对象字面量 (new in ECMAScript; stage 3 draft):

let objClone = { ...obj};

应用1：

        var arr = [1,23,4,5];

        console.log( Math.max( 1,23,4,5 ) );

        console.log( Math.max( ...arr ) );

### 将传入的类数组转为数组

function fn (...a){

console.log(a) //[1]

//[1, 2]

//[1, 2, 3]

//[1, 2, 3, 4, 5, 6, 6, 78]

}

fn(1)

fn(1,2)

fn(1,2,3)

fn(1,2,3,4,5,6,6,78)

## 属性名和属性值相同省略

{miaowei:miaowei} === {miaowei}

## 使用匿名函数避免全局污染

当一个js文件中有多个函数的时候，就相当于在全局声明了这些函数，会造成全局污染！

function add(){

};

function isFunction(){

};

function isArray(){

};

所以，使用匿名函数将这些函数包起来，就不会全局污染。

(function(){

function add(){

};

function isFunction(){

};

function isArray(){

};

})()；

当使用的时候，必须要暴露出这些函数来。

所以，将这个匿名函数声明一个名字，并用return将函数返回出去。

然后这些有名函数就可以作为声明的变量的属性来使用了

let unit = (function(){

function add(){

};

function isFunction(){

};

function isArray(){

};

return{

add:add,

isFunction:isFunction,

isArray:isArray

}

})();

unit.add();

unit.isFunction();

unit.isArray();

## 省略function

大括号里面的function可以省略。

例如：

var miaov = 10;

var o = {

miaov,

fn:function(){}, //带function和不带function是一样的。

fn2(){} //带function和不带function是一样的。

}

console.log(o); //{miaov: 10, fn: ƒ, fn2: ƒ}

## 箭头函数

### 什么是箭头函数？

function fn () {

}

//语法： () => {}

箭头后面带大括号，表示{......}.

这里这个大括号就表示一个函数体。在这里面就写函数方法

var add = (a) => {a+2}

console.log(add(10)) // 因为没有返回值，所以打印undifined

箭头后面不带大括号，表示{return ......}

var add = (a) => a+2

console.log(add(10)) //有返回值，所以打印12

箭头函数后面跟圆括号，表示{return( ) }

var add = (a) => ({a:a+2})

console.log(add(10)) //打印{a:12},表示返回一个对象是{a:12}

1. **var** f = v => v;
2. //上面的箭头函数等同于：
3. **var** f = **function**(v) {
4. **return** v;
5. };

如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分。

如果箭头函数只有一个参数，就不需要使用圆括号代表参数部分。

举例如下：

匿名函数，例1：

1. **var** f = () => 5;
2. //  等同于
3. **var** f = **function** () { **return** 5 };

匿名函数，例2：

1. **var** sum = (num1, num2) => num1 + num2;
2. //  等同于
3. **var** sum = **function**(num1, num2) {
4. **return** num1 + num2;
5. };

有名函数，例3

function fn(){

return 1;

}

相当于： let fn = () => 1;

有名函数，例4：

function fn(e){

return e;

}

//如果只有一个参数()是可以不带的

let fn = e => e;

### 箭头函数的作用： this指向主的老爹

**作用：**

使用箭头函数就不用改变this指向，this指向更加清晰。

document.onclick = function(){

/\*

var that = this;

setTimeout(function(){

console.log(that) //document

},1000)

\*/

setTimeout(()=>{

console.log(this); //document

},1000)

}

**原因：**

箭头函数中this绑定的是所定义的作用域中的this。

例如：

<script>

document.onclick = () => {

console.log(this); //window

}

</script>

箭头函数的this指向主的老爹

document.onclick = function(){

console.log(this); //document

}

一般函数的this指向主

## 函数缩写

reversedMessage: function (){ }

缩写为：

reversedMessage (){ }

## Promise

### 为什么需要promise

        function send(url,callback){

            callback()

        }

        send( "/url1" ,()=>{

            send( "/url2" ,()=>{

                send( "/url3" ,()=>{

                    send( "/url4" ,()=>{

                    })

                })

            })

        })

因为回调过多的时候，看起来很复杂，不利于维护，所以需要promise

异步编程解决方案

使用promise和then代替回调函数callback

使用promise和then使代码由纵向增长变为横向增长。

        var p = new Promise( (res,rej)=>{

            res("成功")

        } )

        var res = p.then( ()=>{

            console.log( 1 )

        } )

        .then( ()=>{

            console.log( 2 )

        } )

        .then( ()=>{

            console.log( 3 )

        } )

### 基本概念

\* Promise：是ES6中新增的异步编程解决方案，体现在代码中它是一个对象，

\* 可以通过 Promise 构造函数来实例化。

\* new Promise(cb) cb必须为一个函数，哪怕是一个空函数，否则报错

promise中的代码在调用promise的实例时会立即执行

三个状态Pending Resolved Rejected

        var p = new Promise( ( resolve,reject )=>{

            resolve() //成功

           reject(); //失败

          } );

console.log( p )

一旦进入成功或者失败，后面的成功或者失败的代码就没有用了，但是不会阻止后面代码的执行，所以要用return放在成功或者失败前面

#### 宏任务和微任务

在执行完所有的宏任务之后，会询问有没有微任务，有就执行微任务

        console.log( 1 ); //宏任务

        setTimeout( ()=>{ //setTimeout会将其中的任务推到下一轮去执行

            console.log( 3 );

        } )

        setTimeout( ()=>{ //setTimeout会将其中的任务推到下一轮去执行

            console.log( 4 );

        } )

        var p = new Promise( (res,rej)=>{ //宏任务

            console.log( 5 )

            res()

        } )

        p.then( ()=>{ //微任务

            console.log( 6 )

        } )

        console.log( 2 ); //宏任务

### promise的方法

\* > 两个原型方法：

\* - Promise.prototype.then() 成功就用这个

\* - Promise.prototype.catch() 失败就用这个

#### then：promise成功的方法

返回值是一个promise

        var **p** = new Promise( ( resolve,reject )=>{

            console.log( "实例化..." );

            return resolve("成功!");

        } );

        p.then( ( data )=>{

            console.log( "then: ",data )

        } )

then的链式调用

        p

        .then((data) => {

                console.log(data)

        })

        .then(() => {

                console.log('then')

        })

then方法调用之后，**返回的一个新的promise对象**

        var p = new Promise( (res,rej)=>{

            // res("成功")

            rej();

        } )

        var p2 = p.then( (data)=>{

            console.log( data )

        } )

        console.log( p2 === p )

#### catch：promise失败的方法

返回值是一个promise

then的一个变种

        var p = new Promise( (res,rej)=>{

          rej();

        } )

        p.catch( ()=>{

            console.log( "失败" )

        } )

捕获then出错的情况，好处是不会阻止后面代码的执行。

        var p = new Promise( (res,rej)=>{

            res("成功")

        } )

        var res = p.then( ()=>{

            console.log( 1 )

        } )

        .then( ()=>{

            console.log( 2 )

            a

        } )

        .then( ()=>{

            console.log( 3 )

            b

        } )

        .catch( (err)=>{

            console.log( err )

        } )

        .finally( ()=>{

            console.log( "执行完了..." )

        } )

#### finally

是指这个promise不管成功还是失败，最后都执行finally

        .finally( ()=>{

            console.log( "执行完了..." )

        } )

#### 静态方法Promise.resolve()

相当于new Promise((res,rej)=>{res()})

     var p = new Promise( (res,rej)=>{res()} )

     p

     .then( ()=>{ console.log(1) } )

     // .then( ()=>{ return new Promise( (res,rej)=>{res()} ) } )

     .then( ()=>{ return Promise.resolve( ) })

括号中接收一个参数，即成功返回的值，传递给下一个then.

        var p = new Promise( (res,rej)=>{res()} )

        p

        .then( ()=>{ console.log(1) } )

        .then( ()=>{ return Promise.resolve( 400 ) })

        .then( (data)=>{ console.log(data) } ) //打印400

#### 静态方法Promise.reject()

相当于new Promise((res,rej)=>{rej()})

由catch接收

        var p = new Promise( (res,rej)=>{res()} )

        p

        .then( ()=>{ console.log(1) } )

        // .then( ()=>{ return new Promise( (res,rej)=>{rej("失败!")} ) } )

        .then( ()=>{ return Promise.reject( "失败!" ) })

        .catch( (err)=>{ console.log( err ) } )

#### 静态方法Promise.all()

Promise.all 可以将多个Promise实例包装成一个新的Promise实例

\* - 当所有Promise实例的状态都变成resolved，Promise.all的状态才会变成resolved，此时返回值组成一个数组，传递给then中的resolve函数。

\* - 只要其中有一个被rejected，Promise.all的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。

        var p1 = new Promise( (res,rej)=>{

            setTimeout( ()=>{

                res("a")

            },1000 )

        } )

        var p2 = new Promise( (res,rej)=>{

            setTimeout( ()=>{

                res("b")

            },1500 )

        } )

        var p3 = new Promise( (res,rej)=>{

            setTimeout( ()=>{

                res("c")

            },500 )

        } )

        var p = Promise.all( [p1,p2,p3] );

        p.then( data=>console.log( data ) )//以数组的形式返回所有promise



#### 静态方法Promise.race()

比较哪一个promise先返回，先返回的接收

        var p1 = new Promise( (res,rej)=>{

            setTimeout( ()=>{

                res("a")

            },1000 )

        } )

        var p2 = new Promise( (res,rej)=>{

            setTimeout( ()=>{

                res("b")

            },1500 )

        } )

        var p3 = new Promise( (res,rej)=>{

            setTimeout( ()=>{

                res("c")

            },500 )

        } )

        var p = Promise.race( [p1,p2,p3] );

        p.then( data=>console.log( data ) )

p3先返回，所以只接收p3的返回值。



## 模块化

Script标签是按照从上到下的顺序加载

type=’module’即为模块，

模块中的东西只能在模块内部使用，不能被外部引用

<script src="./20-a.js" type="module" ></script>

即使加上defer也不能引用type=”module”的模块。

    <script src="./20-a.js" type="module" defer></script>

    <script src="./20-b.js" defer></script>

    defer: 在外链js上使用,会在其他的js,dom节点加载完成之后 再加载,多个defer的执行按照先后次序

    <script src="./20-2.js" defer></script>

    <script src="./20-1.js"></script>

### import/ export

任意两个JS文件都能导入导出互相引用。

import:

引入需要的文件

import浏览器不直接支持,需要通过babel去转

export:

导出

1.export {变量名} import {变量名} from '文件的地址'

(**组件中导出的时候要加括号，那么在主页面上导入的时候，就要跟着加括号**)

2.export default 变量名 import 变量名 from '文件的地址'

var n = 100;

var m = 20;

function add(x,y){

    return x+y;

}

export { n as newN ,m,add}

export { n,m,add}

export default 2000

import f from "./a.js"; //随便一个变量接收export default的值

console.log( f )

一个js文件只能默认导出一个值，否则会报错！

全部导入，并存为tool

import \* as tool from "./a.js";

console.log( tool.m,tool.newN,tool.default )

export function fn(){} import {变量名} from '文件的地址'

# 算法

### 递归

要想一件事情重复做，那么就要用递归。

递归就是自己调用自己。

### 1.冒泡排序法

也叫原始排序法。核心思想：

for套for,新建一个变量坑，让数组的后面一个和前面的比较

function bubbleSort(arr){

var i = 0,

j = 0;

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

for(j=0; j<=arr.length-i; j++){

var temp = 0;

// ">" 从小到大排序

// "<" 从大到小排序

if(arr[j] > arr[j+1]){

temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

}

return arr;

}

### 2.快速排序法

利用递归的方法

## 数组去重

### 方法一：

开一个循环，先把第0个取出来，

再开一个循环，从第0+1开始，和刚开始第0个取出来的比较，

增加从0+2开始，和刚开始第0个取出来的再比较，

所以是循环套循环

循环整个数组，每次循环都查看，当前循环的这个数据是否在整个数组中重复出现，出现就删除

为什么要 j-- ：

因为splice删除了数组中的某一个之后，修改了数组的长度

这个时候如果不j--,会跳过1位数据

let arr = [1,2,1,1,3,4,5,6,7,2,3]; //[1,2,3,4,5,6,7]

for(let i=0;i<arr.length;i++){

for(let j=i+1;j<arr.length;j++){

if(arr[i] == arr[j]){

console.log(j);

arr.splice(j,1);

j--;

}

}

}

### 方法二：

let arr = [1,2,'1',3,4,4,5,5,1,2,3];

let obj = {};

let arr2 = [];

for(let i=0;i<arr.length;i++){

//下面这样就会将数组中的，字符串格式的数字，与数字格式的数字，判断为相同的。

if (!obj[arr[i]]){ // 第一次出现设置为true了，那么！true就为false,就就进不去了

arr2.push(arr[i]);

obj[arr[i]]=true; //obj这个对象中的，key值为arr[i]的,第一次出现就设置为true

}

下面黄色的为能判断字符串格式的数字

// if(!obj[typeof arr[i] + arr[i]]){

// arr2.push(arr[i]);

// obj[typeof arr[i] + arr[i]] = true;

// }

}

console.log(obj);

console.log(arr2);

### 方法三：（最简单）

...new set(data) 数组去重最简单的方法

ES6下的Set去重***只能去重基本数据类型***

        var arr = [1,1,12,2,3,2,1,2];

        var res = [...new Set(arr)]; //打散之后要放到数组中，既加上[]方括号

        console.log( res );

### 方法四：（复杂数据类型的数组去重）

利用{}对象的KEY值唯一的特性。

<script>

let arr = [{

name: 2,

state: true,

output: 'Y',

}, {

name: 2,

state: true,

output: 'Y',

}, {

name: 5,

state: true,

output: 'S',

}, ];

let unique = {}; //首先定义一个空对象

arr.forEach(e=>{

unique[JSON.stringify(e)] = e;

//空对象的key值为JSON.stringify(e)的项的value值赋值为e

//因为对象的key值是唯一的，所以重复的项自然就没有了

})

arr = Object.keys(unique).map(e=>{

//使用Object.keys()提取对象中的key值，组成数组，并循环

return JSON.parse(e);

//把刚才转换成JSON格式的数据转换回来

})

</script>

**key值必须为字符串类型**

## 判断arr是不是数组

### 1.Array.isArray（arr）

判断是不是一个数组

let arr =[1,2,3];

Array.isArray(arr) ; //true

### 2.arr instance Aarray

//true

### 3.arr.constructor == Array

//true

### 4.Object.prototype.toString.call(arr)

        const a = ['Hello','Howard'];

        const b = {0:'Hello',1:'Howard'};

        const c = 'Hello Howard';

        Object.prototype.toString.call(a);//"[object Array]"

        Object.prototype.toString.call(b);//"[object Object]"

        Object.prototype.toString.call(c);//"[object String]"

### 5.Object.prototype.toString.apply(arr)

const a = ['Hello','Howard'];

const b = {0:'Hello',1:'Howard'};

const c = 'Hello Howard';

Object.prototype.toString.apply(a);//"[object Array]"

Object.prototype.toString.apply(b);//"[object Object]"

Object.prototype.toString.apply(c);//"[object String]"

## 数据类型的转换

**通过prompt(“ “)输入的不管是什么类型的数据，都判断为string（字符串）类型的数据。**

Number(字符串/boolean)

通过Number（输入的值），可以将字符串类型的值转换为数字类型的值。

1. 如果转换的内容可以转成数字，那么就直接返回这个内容对应的数字。
2. 如果不可以转换，那么返回NaN。
3. 如果在内容中出现浮点数（小数），那么小数会保留。
4. 如果内容为空，那么转换为0。

举例，将输入的一个字符串类型的数字转换成数字类型的数字。

（只有转换了才能算）

var a = prompt(“请输入”);

var b = Number(a);

console.log(b);

## 避免帕金森的方法

if(!onOff) return;

onOff = false;

.........

onOff = true;

# git和github

## 下载远程仓库代码

git init

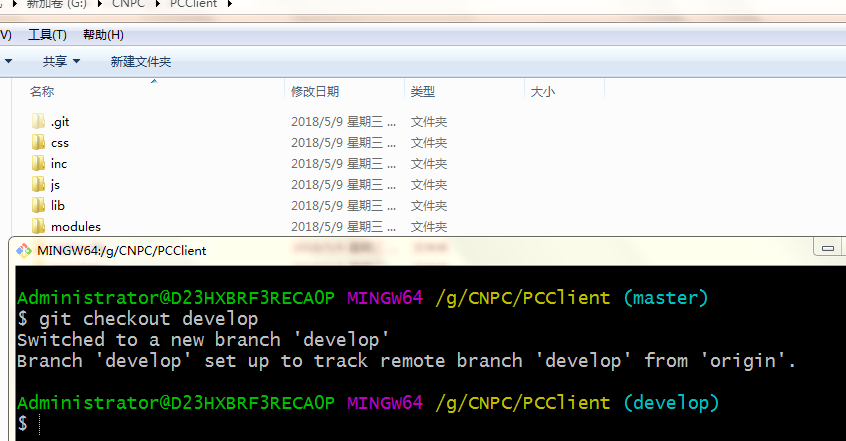
git clone 远程仓库的url地址

注意：有的时候git clone 要去输入密码是因为ssh秘钥的问题，在服务器上把秘钥删掉就好了

只有删完了ssh秘钥之后才会有下面的clone界面。这个才是真克隆。

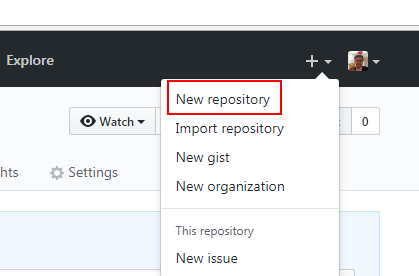
出现拉分支拉不出来的情况就要进一层目录再拉

git checkout 本地分支的名称。例如:

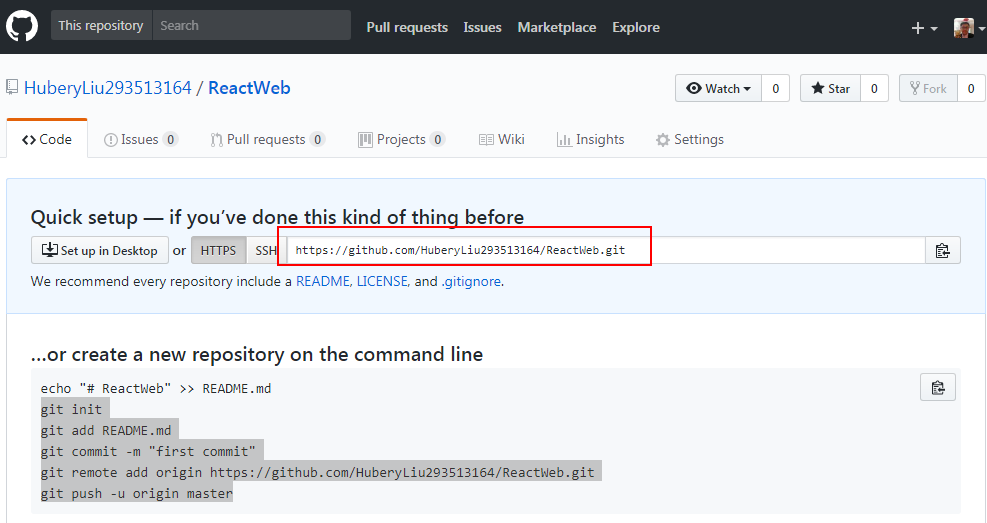


## 上传的基本流程

**1.首先在github上新建一个远程仓库**

****

**新建好是这样的：**

****

**2.在本地新建一个文件夹，放入自己的项目代码。**

git init // 在自己新建的文件中初始化

git add . //把自己的代码放到自己github软件的**缓存区**

git commit -m "first commit" //把自己的代码放到自己github软件的**工作区**

**3.将自己的仓库和刚才建立好的远程仓库关联起来**

git remote add origin <https://github.com/HuberyLiu293513164/ReactWeb.git>

**4.将本地仓库的内容上传到远程仓库**

git push -u origin master

## 后续上传的基本流程

1.git add . //把自己的代码放到自己github软件的**缓存区**

2. git commit -m "关于这次提交的注释" //把自己的代码放到自己github软件的**工作区**

3.git push //把自己的代码放到**github网站**上面

如果修改之前提交过的文件中的内容可以使用快捷提交版本区的方式提交

git commit -a -m "注释"

把已经提交生成版本的文件再次提交，这次提交中包括修改的内容

## pull远程更新代码

如果本地修改了，

再拉取远程代码，

只要本地修改的和要拉取的代码不是同一个文件，没有冲突，就会拉取成功。

## 工作区和暂存区的添加撤销

git add “文件名” （单引号双引号都可）

将文件从工作区添加到暂存区。

git reset HEAD “文件名” （单引号双引号都可）

将文件从暂存区撤销到工作区

## 撤销版本

git commit –m “注释” --amend

git rm –f <file.name> (删除工作区及暂存区的文件)

rm –rf “文件夹名称” （删除整个文件夹）

还原

还原指定的文件

git checkout 历史记录ID 文件名

把工作区的代码还原成暂存区

git checkout -- 文件名

还原整个历史激励

git reset –hard 历史记录

## delete删除参数

人为在window下设置的属性是可以被delete掉

var出来的属性是delete不掉的。

例如：

var a = 10;

function fn(){

var b = c = 20;

}

fn();

// window.a = 20;

// delete this.a; //这个时候是能够删掉的

// console.dir(this);

delete this.a;//10,undefined

alert(this.a);//10,因为没有删掉

alert(this.b);//undefined

alert(this.c);//20

# DOM

Document Object Model(文档对象模型)

通过document提供的接口，赋予开发者操作页面的能力

对于结构的关系非常清晰,任意操作它们

页面中所有的内容都叫节点。

## DOM加载顺序

DOM文档加载的步骤为

1. 解析HTML结构。
2. 加载外部脚本和样式表文件。
3. 解析并执行脚本代码。
4. DOM树构建完成。//DOMContentLoaded
5. 加载图片等外部文件。
6. 页面加载完毕。//load

## childNodes[num]

-- > 所有子节点(包含文本节点，注释节点)。

这是一个**类数组**，num是指第几个节点

div.childNodes.length //查看子节点个数的方法

回车就是**一个**子节点，空格也是**一个**子节点，标签也是**一个**子节点

当空格和回车在一起的时候，算是一个子节点。

空格+回车+文字在一起的时候，算是一个子节点。

## nodeName

**定义和用法**

nodeName 属性指定节点的节点名称。

如果节点是元素节点，则 nodeName 属性返回标签名。

如果节点是属性节点，则 nodeName 属性返回属性的名称。

对于其他节点类型，nodeName 属性返回不同节点类型的不同名称。

## nodeValue

**定义和用法**

nodeValue 属性设置或返回指定节点的节点值。

**注释：**如果您希望返回元素的文本，请记住文本始终位于文本节点中，并且您必须返回文本节点的值（element.childNodes[0].nodeValue）。

**提示：**nodeValue 属性的替代选择是 textContent 属性。

## tagName

**定义和用法**

tagName 属性返回元素的标签名。

在 HTML 中，tagName 属性的返回值始终是大写的。

Q3: 例如：(右键点击生成菜单的例子)

ul.onclick = function(ev){

if(ev.target.tagName === '**LI**'){ //这里的li要大写，why?

console.log(ev.target.innerText);

ev.cancelBubble = true;

}

}

**因为**tagName 属性的返回值始终是大写的！！！

也只有tagName是大写打印，className就是小写打印。

## children

所有元素节点

children:不是标准的属性，**但是所有浏览器都支持**。

找到某个元素下个所有**元素子节点**。

这个应用范围最广，强烈建议用！！！！！！！

### lastElementChild

### firstElementChild

var ul = document.getElementById('ul');

console.log(ul.lastElementChild);

console.log(ul.firstElementChild);

## attributes

DOM元素都有一个对应的attributes属性来存放所有的attribute节点，attributes是一个类数组的容器。

例如：

<div class="box" id="box" gameid="880">hello</div>

上面的div元素的HTML代码中有class、id还有自定义的gameid，这些特性都存放在attributes中，类似下面的形式：

[ class="box", id="box", gameid="880" ]

操作行间的属性(传统方法)：

获取属性：

box.getAttribute(属性名)

设置属性:

box.setAttribute(属性名,属性值)

例如：

删除属性:

box.removeAttribute(属性名)

应用实例：

// console.log(img.src); //不能直接拿来判断(绝对路径)

console.log(img.getAttribute('src'));//(可以拿来判断（相对路径）)

attribute节点都是在HTML代码中可见的

property只是一个普通的名值对属性

## data-attrName /dataset. attrName

操作行间的属性(较新的方法)：

ES5

**data-**attrName 设置行间属性的方法

dataset.attrName 获取行间添加的属性的方式

下面是元素应用data属性的一个例子：

<div id="day2-meal-expense"

data-drink="coffee"

data-food="sushi"

data-meal="lunch">¥20.12</div>

要想获取某个属性的值，可以像下面这样使用dataset对象：

var expenseday2 = document.getElementById('day2-meal-expense');

var typeOfDrink = expenseday2.dataset.drink;

假设上面的例子中现在有如下data属性，data-meal-time，则我们要获取相应的值可以使用：

expenseday2.dataset.mealTime

而使用dataset属性，我们根本不需要任何循环去获取你想要的那个值，直接秒杀

**什么地方使用dataset？**

每次你使用自定义data属性的时候，使用dataset去获取名-值对就是个不错的选择

## nodeType

查看节点的类型

例如：console.log(div.nodeType); //打印1

常用的类型:

document 9

**元素节点**  1

**文本节点** 3

空格和文字都是文本节点，在一个元素中既有空格又有文字，算一个文本节点。

**注释节点**  8

**属性节点** 2

ul

<ul>

<li>1</li>

<li>2</li>

</ul>

## parentNode 父节点

var alis = ul.children;

console.log(alis[0].parentNod);

//打印

parentNode.parentNode 爷爷节点

console.log(alis[0].parentNod.parentNode); //打印body

一个页面最大的是document

window是窗口

## ElementSibling 兄弟节点

找到除了上个和下个的兄弟节点用：

e.parentNode.children

当e！= e.parentNode.children时，也就是“找到除了上个和下个的兄弟节点”

找到某个元素的下个兄弟节点

找到某个元素的下个兄弟节点：

**next**ElementSibling

找到某个元素的上个兄弟节点：

**previous**ElementSibling

## offsetParent父级属性操作

最近的有定位属性(position:···;)的祖先节点

如果祖先节点都没有定位，那么默认为body

## Attribute行间属性操作

**获取元素行间的属性**

elem.getAttribute('key值')

例如：

<div id="div1" index ="2"></div>

alert(div.getAttribute('index')) //这里就获取了

**设置元素的行间属性**

elem.setAttribute('key值','value值')

**删除元素的行间属性**

elem.removeAttribute('key值')

## 元素的基点

**左上角的点**！

## 获取指定元素的细节信息

### ele.getBoundingClientRect().left/top/right/bottom

ele.getBoundingClientRect()

获取指定元素的细节信息

**返回值**:

{} (返回值为一个对象)

（高版本：left,top,bottom,rigit,width,height）

（低版本：left,top,right,bottom）

获取的是绝对位置（当前位置到底边的距离）：

DOMRect 对象包含了一组用于描述边框的只读属性——left、top、right和bottom，单位为像素。除了 width 和 height 外的属性都是相对于**视口的左上角位置**而言的。

获取的值是会**根据滚动条变化**的：

当计算边界矩形时，会考虑视口区域（或其他可滚动元素）内的滚动操作，也就是说，当滚动位置发生了改变，top和left属性值就会随之立即发生变化（因此，它们的值是相对于视口的，而不是绝对的）。如果不希望属性值随视口变化，那么只要给top、left属性值加上当前的滚动位置（通过window.scrollX和window.scrollY）

## 元素的宽高

### **HTMLElement.style**

 通过 style 可以访问的 CSS 属性列表，可以查看 [CSS Properties Reference](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Properties_Reference" \o "/en-US/docs/Web/CSS/CSS_Properties_Reference)。

在为特定的元素设置样式时很有用。

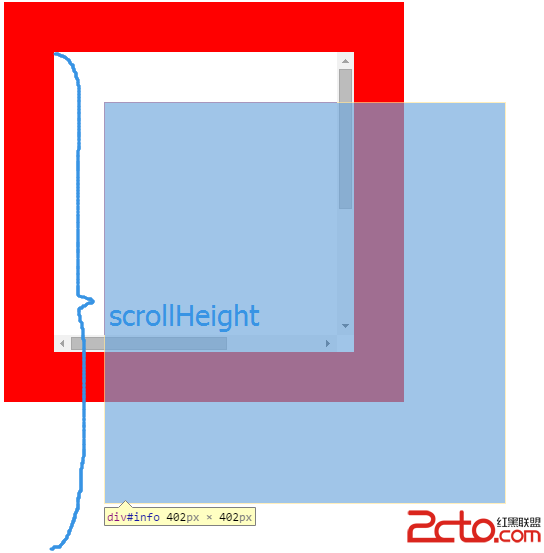
**可读可写**

通常，要了解元素样式的信息，仅仅使用 style 属性是不够的，这是因为它只包含了在元素内嵌 style 属性（attribute）上声明的的 CSS 属性，而不包括来自其他地方声明的样式，如 [<head>](https://developer.mozilla.org/zh-CN/docs/Web/HTML/Element/head" \o "HTML head 元素 规定文档相关的通用信息（元数据），包括文档的标题，文档的样式和脚本的链接（定义）等。) 部分的内嵌样式表，或外部样式表。要获取一个元素的所有 CSS 属性，你应该使用 [window.getComputedStyle()](https://developer.mozilla.org/zh-CN/docs/Web/API/Window/getComputedStyle" \o "Window.getComputedStyle() 方法给出应用活动样式表后的元素的所有CSS属性的值，并解析这些值可能包含的任何基本计算。)。

### ele.scrollHeight/scrollWidth

获取某个元素的被**内容撑开**的高度/宽度

\*obj.scroll**H**eight

  
scrollWidth=padding+包含内容的完全高度  
这里等于：50+402+50=502

### ele.clientWidth/elem.clientHeight

(获取某个元素的宽高)

注意：

**不**计算边框

加上padding

即width + padding

### ele.offsetWidth/elem.offsetHeight

(获取某个元素的宽高)

注意：

**计算边框**

加上padding

即width + padding + border

## 滚动距离

### ele.scrollTop

### ele.scrollLeft

### document.documentElement.scrollTop

(DOM提供)document.documentElement.scrollTop**能读能写**

### document.documentElement.scrollLeft

(DOM)document.documentElement.scrollLeft**能读能写**

元素的样式

### document.documentElement.style.setProperty（变量，新值）

## 元素的边距

### ele.offsetLeft/offsetTop

注意:从来都没有ele.offsetRight和ele.offsetBottom

**offsetLeft:**

**某元素**的**外边框** 到 这个元素的有定位父级的**内边框** 的距离。**不包括边框。**

**offsetTop**

上外边框到有定位父级的上内边框的距离。

offsetLeft:

从子级的左外边框到定位父级的左内边框的距离

number类型

offsetTop：

从子级的上外边框到定位父级的上内边框的距离

number类型

正常使用它

1.子级要有定位，定位父级也要有定位(子级没有定位，父级也就不需要定位)

2.子级和定位父级必须要触发haslayout(加宽高、zoom...)

3.清除默认样式

## 页面可视区的宽高clientWidth/clientHeight

### document.documentElement.clientWidth/clientHeight

/\*

document.documentElement.clientWidth(可视区的宽度)

document.documentElement.clientHeight(可视区的高度)

\*/

console.log(document.documentElement.clientHeight)

### body.clientWidth/clientHeight

另外一种获得可视区宽高的写法

## 页面的实际尺寸

### document.documentElement.scrollHeight/ scrollWidth

表示html元素内容的实际尺寸

## 创建元素

**document.createElement**('标签的名字'); //创建元素

(这里大多数还是document)

对于<img>标签要用：let img = new Image

## 插入元素方法一：

appendChild

（向父级中的末尾添加一个元素）

parentNode.appendChild(*node*)

|  |  |  |
| --- | --- | --- |
| **参数** | **类型** | **描述** |
| *node* | 节点对象 | 必须。你要添加的节点对象。 |

## 插入元素方法二：

（向父级中的某个元素前插入元素）

parentNode.insertBefore(新添加的元素,已有的元素)

特性：

如果第二个参数为假的，那么还是往前面加

也就是说，如果第二个参数是没有的，那么等同于appendChild

## 删除元素

parent.removeChild(删除的那个)

高版本浏览器自带

element.remove();

## 替换元素

replaceChild();

parent.replaceChild(newnode,oldnode);

DOM的操作都是剪切，不是复制

## 动态/静态数组

获取动态数组

getElement**s**By**T**agName()

children

getElement**s**By**C**lassName()

**前面只能加document**,不能加定量或者变量。

(驼峰命名：两边小写，中间大写)

获取静态数组：

querySelectorAll

## 克隆

node.cloneNode();

node.cloneNode(true); 参数true表示:还可以克隆子节点

## offsetParent

定位父级是谁

当某元素的祖先级没有position：relative/absolute定位的时候，默认定位父级为body

只要祖先级有定位position：relative/absolute，那么子级的定位父级为这个加了定位的元素

建议：

子级加定位父级也加定位。

## 表格

tBodies[0]

一个table可以有很多tBody,获取到的是一个集合

rows（一行,获取tr的）

也是一个集合，也是支持下标

cells(一个单元格,获取td)

tHead首行

tFoot最后一行

createTFoot()

createTHead()

删除单元格

deleteRow(index)

//按 HTML DOM 来获取表格的行数

alert(table.rows.length);

//按 HTML DOM 来获取表格主体里的行数

alert(table.tBodies[0].rows.length);

//按 HTML DOM 来获取表格主体内第一行的单元格数量(tr) alert(table.tBodies[0].rows[0].cells.length);//获取第一行单元格的数量

//按 HTML DOM 来获取表格主体内第一行第一个单元格的内容(td) alert(table.tBodies[0].rows[0].cells[0].innerHTML);//获取第一行第一个单元格的内容

## title网页标题

document.title = ABC

## body网页正文

document.body 指的就是HTML中的body.

<body>

<script>

console.log(document.body)

</script>

</body>

## innerHTML 和 innerText

innerHTML和innerText是放在document.body下面的标签。

可以直接用document.body引用，也可以用document.body下面的元素引用。

innerText:

操作双标签中的**文本**

innerHTML : 操作元素的内容（**包括操作结构**）

box.innerHTML = '<p>哈哈</p>';

这个时候box中就有了p标签

innerText只可以解析文本内容

innerHTML可以解析文本内容+标签

注意：只有cssText能够覆盖原有样式，**innerHTML标签不能覆盖原有样式**，只会在原有样式上进行修改。

## **scrollIntoView()**

该**Element.scrollIntoView()**方法将调用它的元素滚动到浏览器窗口的可见区域。

* 如果true，元素的顶部将对齐到可滚动祖先的可见区域的顶部。这是默认值。
* 如果false，元素的底部将与可滚动祖先的可见区域的底部对齐。

## Dom操作带来性能下降

任何DOM操作都会使页面性能下降，例如：

**每次进行DOM操作，即每次innerHTML插入的时候，都会引发全页面的渲染。**

下面的插入方式性能差。

for(var i=0;i<1000;i++){

box.**innerHTML**+=`<div style="width:100px;height:100px;background:red;">${i+1}</div>`;

}

**避免DOM操作就会使性能更高，例如下面的性能会更高**

for(var i=0;i<1000;i++){

str+=`<div style="width:100px;height:100px;background:red;">${i+1}</div>`;}

}

box.**innerHTML** = str;

# BOM

Browser Object Model

浏览器对象模型

**window**

通过window提供的api赋予开发者操作浏览器的能力。

基本上是不兼容的。

window.open(url,打开的方式,可以设置浏览器窗口)

第三个参数支持: width=420,height=230,resizable=yes(是否可以缩放),scrollbars=yes(带不带滚动条)

在Chrome中必须是在**用户主动触发**的情况下才会有用，不然会被拦截。

IE下是不会拦截的。

例如：

document.documentElement.onclick = function(){

window.open('http://www.miaov.com','\_blank','width=420,height=230,resizable=yes');

}

## 文本输入流

新窗口如果使用了document.write，那么等同于开了个水龙头，**开着一直不关闭就容易出错，一般的症状就是js代码不执行**

**解决:**

使用close()

例如：

const btn = document.getElementById('btn');

const t = document.getElementById('t');

btn.onclick = function(){

let s = window.open('','\_blank');

**s.document.write(t.value);**

**s.document.close();**

## 关闭窗口

close()

在使用该方法的时候，IE会弹出一个提示框

Chrome直接关闭

例如：

window.close();

## navigator/location/history

### 用户信息navigator

浏览器用户信息window.navigator.userAgent

返回值为字符串

IE10及以下都是MSIE xx

注意:

这个信息容易被修改,只能做参考，不能100%肯定

### 地址栏信息location

地址栏信息

window.location.href

查询信息:

window.location.search

?到#号之间的信息

#### 哈希（锚）信息

window.location.hash

#及之后信息

#### 哈希变化事件

window.location.onhashchange事件:

可以监听hash值的变化,只要hash值改变就会触发这个事件

\*\*\*改变hash值，页面是不会跳转的。

\*\*\*改变search值，页面是会跳转的。

hash，search都是可读可写的。

#### 哈希hash属性

**定义和用法**

hash 属性是一个**可读可写**的字符串，该字符串是 URL地址栏的锚部分（从 # 号开始的部分）。（**包括#这个符号**）。

**语法**

location.hash=anchorname

写操作:

window.loaction.hash = 'page=1';

读操作:

console.log(window.loaction.hash)

如果把值赋值给window.location.hash,就会加一个#

例如：

h = 'lx=sh';

window.location.hash = h;

console.log(window.location.hash) // # lx=sh

**注意：hash值不会随着页面刷新而变化。**

监听变化

window.onhashchange = function(){}

#### 搜索search属性

定义和用法

search 属性是一个可读可写的字符串，可设置或返回当前 URL 的查询部分（问号 ? 之后的部分）。

**语法**

location.search=path\_from\_questionmark

### 历史信息history

一刷新就没了

onpopstate

onpushstate

history.go(-1) 返回上一步

## 滚动距离window.pageYOffset/pageXOffset

上滚动距离

(DOM提供)document.documentElement.scrollTop**能读能写**

(BOM提供)window.pageYOffset**只能读，不能写**

左滚动距离:

(DOM)document.documentElement.scrollLeft**能读能写**

(BOM提供) window.pageXOffset**只能读，不能写**

注意：

只有左上，没有右下！！！

IE8-浏览器不支持pageYOffset/pageXOffset

## 页面可视区的宽高window.innerWidth/innerHeight

可视区的宽：

(DOM提供的可视区宽高):

document.documentElement.clientWidth/ clientWidth

(BOM提供的可视区宽高, 不计算滚动条的。):

window.innerWidth/ innerHeight

## 滚动事件

onscroll

当滚动条滚动的时候触发这个事件

window.onscroll = function(){

console.log(12324);

}

## 缩放事件

onresize

窗口缩放的时候触发

window.onresize = function(){

console.log('缩放');

}

## 设置内容滚动到指定坐标

可把内容滚动到指定的坐标

**定义和用法**

scrollTo() 方法可把内容滚动到指定的坐标。

**语法**

scrollTo(*xpos,ypos*)

|  |  |
| --- | --- |
| **参数** | **描述** |
| xpos | 必需。要在窗口文档显示区左上角显示的文档的 x 坐标。 |
| ypos | 必需。要在窗口文档显示区左上角显示的文档的 y 坐标。 |

## 元素的位置信息Window.getComputedStyle()

Window.getComputedStyle() 方法给出应用活动样式表后的元素的所有CSS属性的值，并解析这些值可能包含的任何基本计算。

**只可读，不可写**

**语法**

let style = window.getComputedStyle(element, [pseudoElt]);

**element**

 用于获取计算样式的[Element](https://developer.mozilla.org/zh-CN/docs/Web/API/Element" \o "Element（元素）接口是 Document的一个对象. 这个接口描述了所有相同种类的元素所普遍具有的方法和属性。 这些继承自Element并且增加了一些额外功能的接口描述了具体的行为. 例如,  HTMLElement 接口是所有HTML元素的基础接口， 而 SVGElement 接口是所有SVG元素的基本接口.)

从getComputedStyle返回的对象是只读的，可以用于检查元素的样式（包括由一个<style>元素或一个外部样式表设置的那些样式）

# 事件

## 传统事件（DOM0事件）

点击事件 -> onclick

鼠标移入事件 -> onmouseover

鼠标移出事件 -> onmouseout

**元素本身就有事件，如果没有事件函数，那么就是null !!!**

|  |  |
| --- | --- |
| **属性** | **当以下情况发生时，出现此事件** |
| onabort | 图像加载被中断 |
| onblur | 元素失去焦点 |
| onchange | 用户改变域的内容 |
| onclick | 鼠标点击某个对象 |
| **ondblclick** | 鼠标双击某个对象 |
| onerror | 当加载文档或图像时发生某个错误 |
| onfocus | 元素获得焦点 |
| onkeydown | 某个键盘的键被按下 |
| onkeypress | 某个键盘的键被按下或按住 |
| onkeyup | 某个键盘的键被松开 |
| onload | 某个页面或图像被完成加载 |
| onmousedown | 某个鼠标按键被按下 |
| onmousemove | 鼠标被移动 |
| onmouseout | 鼠标从某元素移开 |
| onmouseover | 鼠标被移到某元素之上 |
| onmouseup | 某个鼠标按键被松开 |
| onreset | 重置按钮被点击 |
| onresize | 窗口或框架被调整尺寸 |
| onselect | 文本被选定 |
| onsubmit | 提交按钮被点击 |
| onunload | 用户退出页面 |

传统事件:

onclick

onmouseover

onmouseout

onmouseenter

onmouseleave

onmousedown

onmouseup

onrezie

onscroll

onhashchange

onpopstate

## 监听事件（绑定事件、DOM2事件）

addEventListener() 方法用于向指定元素添加事件句柄

**语法**

e*lement*.addEventListener(*event*, *function*, *useCapture*)

**参数值**

|  |  |
| --- | --- |
| **参数** | **描述** |
| event | 必须。字符串，指定事件名。  **注意:** 不要使用 "on" 前缀。 例如，使用 "click" ,而不是使用 "onclick"。 |
| function | 必须。指定要事件触发时执行的函数。   当事件对象会**作为第一个参数**传入函数。 事件对象的类型取决于特定的事件。例如， "click" 事件属于 MouseEvent(鼠标事件) 对象。  **想传入其他参数，可以放在第二个位置，及其以后。这里好像是不能传入参数的额！！！？？？？** |
| useCapture | 可选。布尔值，指定事件是否在捕获或冒泡阶段执行。  可能值:   * **true - 事件句柄在捕获阶段执行** * false- false- 默认。事件句柄在冒泡阶段执行 |

语法：

obj.addEventListener('不带on的事件名',事件函数,是否捕获(默认是false，不捕获))

注意：

如果在事件套事件的时候使用了addEventlistener那么会出现重复绑定。

理解为：1.事件套事件就是“box的click”套“box2的mouseover”。

2.重复绑定就是：“box的click”的时候同时绑定了box2

box.addEventListener('click',fn);

function fn(){

alert(1);

box2.addEventListener('mouseover',fn);

function fn(){

alert('小盒子触发');

}

}

## 解绑事件

**语法**

*element*.removeEventListener(*event*, *function*, *useCapture*)

**参数值**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| event | 必须。要移除的事件名称。.  **注意:**不要使用 "on" 前缀。 例如，使用 "click" ,而不是使用 "onclick"。 |
| function | 必须。指定要移除的函数。 |
| useCapture | 可选。布尔值，指定移除事件句柄的阶段。  可能值：   * **true - 在捕获阶段移除事件句柄** * false- 默认。在冒泡阶段移除事件句柄   **注意:** 如果添加两次事件句柄，一次在捕获阶段，一次在冒泡阶段，你必须单独移除该事件。 |

格式：

obj.removeEventListener(要解除的事件名,要解除的函数名,boolean);

注意：

\*\*\* 解除的函数必须为有名函数，匿名函数解不了

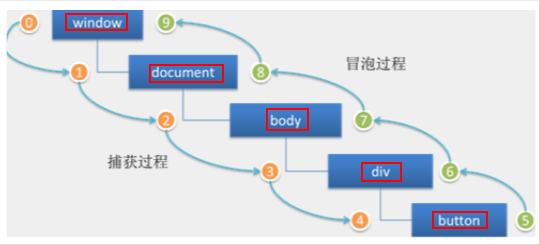
## Event:事件对象

当用户触发某个事件的时候，记录下用户操作的一些细节的信息。

1.事件函数的第一个参数，一定是事件对象。

2.在IE和chrome中 全局有个event属性，就是事件对象

## 事件流（事件模型）、什么是事件冒泡和捕获？



1.捕获阶段：

从（window）上往下到目标点的阶段

2.到达目标阶段

3.冒泡

从（目标点）下往上到（window）的阶段

传统的事件(带on的就是传统事件)是没有捕获的，只有冒泡。

非传统事件就即有捕获，也有冒泡。(先捕获，后冒泡)

目标阶段不用先走捕获，只走顺序绑定，同事件，同名函数的下面会把上面的覆盖。

这里是指非传统事件，例如：

### 冒泡的应用：事件委托（事件源）

什么是事件委托？

简单来说事件委托就是父元素监听子元素的冒泡事件。

    <div id="div">

        <ul id="ul" >

            <li data-key="北京">北京</li>

            <li data-key="上海">上海</li>

            <li data-key="杭州">杭州</li>

        </ul>

    </div>

    <script>

        document.getElementById('div').addEventListener('click',function(e){

        var value=e.target.attributes['data-key'].value; // 获取目标阶段元素的'data-key'属性的值

        console.log(value);

        });

    </script>

ev.target 可以通过触发父级事件直接找到触发这个事件的子级是谁。

**通过点击父级找到所有儿子！！！！！**

\*\* 只能监听父级及以下的元素，兄弟级是监听不到的。

冒泡机制

**例如：**

btn.onclick = function(){

num ++;

}

ul.onclick = function(ev){

alert(**ev.target**.innerText); //点哪个就显示是哪个

console.log(ev.target);}

例如：(右键点击生成菜单的例子)

ul.onclick = function(ev){

if(ev.target.tagName === '**LI**'){ //这里的li要大写，why?

console.log(ev.target.innerText);

ev.cancelBubble = true;

}

}

因为所有元素的标签通过“.tagName”都是以大写打印！

也只有tagName是大写打印，className就是小写打印。

### 冒泡的弊端

**问题:**

当目标点绑定了事件函数，祖先级元素也绑定了同样的事件

函数，在触发目标事件的时候通过冒泡也会调用祖先级的事件函数

### 如何解决冒泡

**解决办法：**

#### 1.阻止冒泡

在目标点上取消，不让父级触发。

ev.cancelBubble = **true**; //cancelBubble是一个属性。

ev.stopPropagation(); //stopPropagation是一种方法。

例如：

<body>

<button id="btn">点击打开</button>

<div id="div"></div>

<script>

const div = document.getElementById('div');

const btn = document.getElementById('btn');

btn.onclick = function(ev){

div.style.display = 'block';

ev.cancelBubble = true; //取消冒泡

ev.stopPropagation();

}

document.onclick = function(){

div.style.display = 'none';

}

</script>

#### 2.不绑同一事件名的函数

#### 3.使用不冒泡行为

onmouseenter

onmouseleave

\*\*\* 不冒泡

## 浏览器的默认行为

什么是浏览器的默认行为？

你不想触发某个行为，浏览器偷偷的帮你触发了

1.传统的事件(所谓传统事件也就是on...的事件)绑定阻止默认行为:

**return false;**

2.绑定事件的阻止默认行为:

**ev.preventDefault();**

## 焦点事件

onfocus:

聚焦

onblur:

失焦

## 键盘事件

onkeydown 键盘按下

onkeyup 键盘抬起

onkeypress 事件会在键盘按键被按下并释放一个键时发生。

onkeypress很少用，并且不管用，建议少用。用onkeydown代替。

**ev.keyCode 获取到键值**

**ev.which 也是获取键值**

常用的键值

左上右下

37 - 40

0-9

48 - 57

a-z

65 - 90

回车 13

空格 32

查看键值的方法：

document.onkeydown = function(ev){

console.log(ev.keyCode); }

### 键盘事件注意点

小细节：

当通过键盘去监控（获取）输入的内容时，请使用onkeyup

因为onkeydown会少监听一次。

addEventListener()

不管有没有写true，那么都有捕获的过程，只不过没监听而已

没监听不等于没有

shift -> 16

注意：

如果使用组合键的时候，后面的键会把前面的键给覆盖

特殊键:

ev.shiftKey

ev.altKey

ev.ctrlKey

布尔值 按着就是true,否则false

once:

只触发一次,掉用完之后就解除绑定。

ev.cancelBubble = true

ev.stopPropagation();

注意：

如果在事件套事件的时候使用了addEventlistener

那么会出现重复绑定。

## 鼠标事件

**ev.which 获取鼠标键值**

document.onclick = function(ev){

console.log(ev.which); //鼠标左键是1，右键没有，滚轮键也没有

}

### 鼠标移动事件

clientX/clientY 鼠标到可视区的距离（**不包含滚动条的**）

pageX/pageY **鼠标**到浏览器顶端的距离（**包含滚动条**）

以上得到的只是一个数字类型，但是实际单位是“px”。

例如，“跟着鼠标走的div”

const div = document.getElementsByTagName('div')[0];

document.onmousemove = function(ev){

div.style.left = ev.pageX - div.offsetWidth/2 + 'px';

div.style.top = ev.pageY - div.offsetHeight/2 + 'px';

}

### 弹出右键菜单

**oncontextmenu**：

弹出右键菜单

### 鼠标滚轮事件

**onscroll**

只有滚动条的情况下才会触发

**onmousewheel**：IE和chrome下

ev.wheelDelta

上：180 正数

下: -180 负数

**DOMMouseScroll**  FF

ev.detail

上: -3 负数

下: 3 正数

**例如：**

addWheel(window,

function(o){

if(o){

alert('上滚');

}else{

alert('下滚');

}

}

);

function addWheel(obj,fn){

obj.addEventListener('mousewheel',callback);

obj.addEventListener('DOMMouseScroll',callback);

function callback(ev){

let o = true; //向上

if(ev.wheelDelta){

o = ev.wheelDelta > 0?true:false;

}else{

o = ev.detail < 0? true:false;

}

fn && fn(o);

//这里是指，如果存在fn（即上面的橘黄色的函数），那么就执行fn(o),否则就不执行。这个程序里面写了fn这个函数，所以就是真的，所以就执行fn(o)。如果程序里面没有写fn这个函数，那么也就不执行了。

这里是个回调函数！这种写法是个典型！

//alert(o);

}

}

## input/textarea表单事件

### t.focus();

自动聚焦

### t.select();

选中文本,也可以拿来聚焦

### selectionStart

### selectionEnd

### setSelectionRange

其使用方式如下：

**inputElement.setSelectionRange(selectionStart, selectionEnd, [optional] selectionDirection);**

* **selectionStart**：第一个被选中的字符的序号（index），从0开始。
* **selectionEnd**：被选中的最后一个字符的前一个。换句换说，不包括index为selectionEnd的字符。
* **selectionDirection**：选择的方向。可选值为forward、backward或none。

### t.oninput();

**文本内容发生变化的时候触发**

**例如：**

t.focus();

btn.onclick = function(){

t.select();

}

t.oninput = function(){

span.innerText = (this.value.length)+'/266';

console.log(this.value.length);

}

## 键盘控制时的停顿问题

浏览器默认为了优化用户体验，设置了如下规则：

在按下某个键盘的时候，大概有430ms左右事件让用户抬起，如果430ms左右之后还没有送开手，就说明是长按。

在这种规则下就会产生**停顿效果**。

**解决方法:**

按下的时候开个定时器，把间隔范围缩小到30左右。

代码如下：

let timer = null;

document.onkeydown = function(ev){

clearInterval(timer);

timer = setInterval(function(){

if(ev.keyCode == 39){

div.style.left = div.offsetLeft + 5 +'px';

}

if(ev.keyCode == 40){

div.style.top = div.offsetTop + 5 +'px';

}

if(ev.keyCode == 38){

div.style.top = div.offsetTop - 5 +'px';

}

if(ev.keyCode == 37){

div.style.left = div.offsetLeft - 5 +'px';

}

},10);

}

document.onkeyup = function(){

clearInterval(timer);

}

## 事件应用之拖拽

1.onmousedown 针对元素进行点击

2.onmousemove 针对document进行，move是在document上move

3.onmouseup 针对document进行

一般这三个事件都是在一起用吧？

例如：

const box = document.getElementById('box');

let disX = 0;

let disY = 0;

box.onmousedown = function(ev){

/\*

拿到按下时的位置

\*/

disX = ev.pageX;

disY = ev.pageY;

let t = box.offsetTop; //这里将执行完一次拖拽之后的盒子位置存起来

let l = box.offsetLeft;

box.onmousemove = function(e){

/\*

拿到move时的位置

\*/

let mpageX = e.pageX;

let mpageY = e.pageY;

box.style.left = l + (mpageX - disX) + 'px'; //在执行下一次拖拽的时候，将第一次拖拽之后的盒子位置加上，这样盒子就不会在第二次拖拽的时候从(0,0)点出发。

box.style.top = t + (mpageY - disY) + 'px';

// box.style.left = (mpageX - disX) + 'px';

// box.style.top = (mpageY - disY) + 'px';

// 这里这种情况就会，在第二次拖拽的时候，让盒子从(0,0)点出发

}

box.onmouseup = function(){

box.onmousemove = null;

}

}

例外一种避免第二次拖拽从（0,0）点出发的方法是：

const box = document.getElementById('box');

let disX = 0;

let disY = 0;

box.onmousedown = function(ev){

/\*

拿到按下的位置到元素的边的距离

\*/

disX = ev.pageX - box.offsetLeft;

disY = ev.pageY - box.offsetTop;

box.onmousemove = function(e){

/\*

拿到move时的位置

\*/

let mpageX = e.pageX;

let mpageY = e.pageY;

box.style.left = (mpageX - disX) + 'px';

box.style.top = (mpageY - disY) + 'px';

}

box.onmouseup = function(){

box.onmousemove = null;

}

}

## 拖拽避免默认行为

move、up的时候绑在document身上就能解决快速移动和没在元素本身释放的bug，释放up的时候多次触发up事件中的代码。

### 解决办法1：

在up的时候也把up事件清除,这样就不会导致

在down的时候清除默认行为。

例如，黄色代码所示：

const box = document.getElementById('box');

let disX = 0;

let disY = 0;

box.onmousedown = function(ev){

/\*拿到按下的位置到元素的边的距离\*/

disX = ev.pageX - box.offsetLeft;

disY = ev.pageY - box.offsetTop;

document.onmousemove = function(e){

/\*拿到move时的位置\*/

let mpageX = e.pageX;

let mpageY = e.pageY;

box.style.left = (mpageX - disX) + 'px';

box.style.top = (mpageY - disY) + 'px';

}

document.onmouseup = function(){

document.onmousemove = null;

document.onmouseup = null;

}

return false;

}

### 解决办法2：

把匿名函数变成有名函数

const box = document.getElementById('box');

box.addEventListener('mousedown',function(ev){

let disX = ev.pageX - this.offsetLeft;

let disY = ev.pageY - this.offsetTop;

document.addEventListener('mousemove',move);

document.addEventListener('mouseup',up);

function move(e){

box.style.left = e.pageX - disX + 'px';

box.style.top = e.pageY - disY + 'px';

}

function up(){

document.removeEventListener('mousemove',move);

document.removeEventListener('mouseup',up);

}

});

### 解决办法3：

使用开关+有名函数

const box = document.getElementById('box');

let onOff = true;

box.addEventListener('mousedown',function(ev){

onOff = true;

let disX = ev.pageX - this.offsetLeft;

let disY = ev.pageY - this.offsetTop;

document.addEventListener('mousemove',move);

document.addEventListener('mouseup',up);

function move(e){

if(!onOff)return;

box.style.left = e.pageX - disX + 'px';

box.style.top = e.pageY - disY + 'px';

}

function up(){

onOff = false;

if(!onOff)return;

console.log(123);

}

});

## 仿window拖拽

const box = document.getElementsByClassName('box')[0];

let disX = 0;

let disY = 0;

let actX = 0;

let actY = 0;

box.onmousedown = function(ev){

disX = ev.pageX - box.offsetLeft;

disY = ev.pageY - box.offsetTop;

let cloneBox = box.cloneNode(true); //仿window桌面的思想就是，clone一个原来的元素，把这个元素变成半透明的，然后这个半透明元素在动。当鼠标抬起时，将克隆的元素remove(),将原来的元素展示。

cloneBox.style.opacity = .5;

body.appendChild(cloneBox);

document.onmousemove = function(e){

cloneBox.style.left = (e.pageX - disX) + 'px';

cloneBox.style.top = (e.pageY - disY) + 'px';

//存分身的位置，存起来给真身用。

actX = e.pageX - disX;

actY = e.pageY - disY;

}

document.onmouseup = function(){

document.onmousemove = document.onmouseup = null;

cloneBox.remove();

box.style.left = actX + 'px';

box.style.top = actY + 'px';

}

return false;

}

## 回放

回放的核心思想：

1.新建数组，用于保存运动元素每个位置（X轴一个数组，Y轴一个数组）。

2.按下mousedown和mousemove的时候，分别将盒子的每个位置push到数组中。

3.开一个连续炸的定时器setInterval,让每一次循环把数组中的每一个元素拿出来展示（可以选择从头到尾shift,或者pop从尾到头。

4.定时器在回放完一次之后，要记得关闭，否则就会不断的回放，一动就会回放！

const box = document.getElementById('box');

const btn = document.getElementById('btn');

let arrX = [];

let arrY = [];

let disX = 0;

let disY = 0;

box.onmousedown = function(ev){

/\*

拿到按下时的位置

\*/

disX = ev.pageX - box.offsetLeft;

disY = ev.pageY - box.offsetTop;

arrX.push(box.offsetLeft);

arrY.push(box.offsetTop);

document.onmousemove = function(e){

box.style.left = e.pageX - disX + 'px';

box.style.top = e.pageY - disY + 'px';

arrX.push((e.pageX - disX));

arrY.push((e.pageY - disY));

}

document.onmouseup = function(){

document.onmousemove = document.onmouseup = null;

}

return false;

}

let timer = null;

btn.onclick = function(){

timer = setInterval(function(){

if(!arrX.length && !arrY.length){ //

clearInterval(timer);

}

box.style.left = arrX.shift() + 'px';

box.style.top = arrY.shift() + 'px';

},50);

}

## 碰撞

const div1 = document.getElementById('div1');

const div2 = document.getElementById('div2');

Drag(div2);

// Drag(div2);

/\*

1.获取鼠标的点击位置，因为是在box上面点，所以就要获取在box上面的相对位置。

2.碰撞元素的四个方向获取（受鼠标移动事件影响），被碰撞元素四个方向的获取（受被碰元素影响）。

3.比较是不是碰到。

4.在mousemove时，展现碰撞元素的位置。

5.消除默认样式。

\*/

function Drag(box){

box.onmousedown = function(dev){

let disX = dev.pageX - this.offsetLeft;

let disY = dev.pageY - this.offsetTop;

document.onmousemove = function(mev){

/\*

B的四个方向

\*/

let bl = mev.pageX - disX; //

let bt = mev.pageY - disY;

let br = bl + box.offsetWidth;

let bb = bt + box.offsetHeight;

/\*

A的四个方向

\*/

let al = div1.offsetLeft;

let at = div1.offsetTop;

let ar = al + div1.offsetWidth;

let ab = at + div1.offsetHeight;

if(br < al || bb < at || bl > ar || bt > ab){

div1.className = 'skyblue';

}else{

console.log('碰到');

div1.className = 'green';

}

box.style.left = bl + 'px';

box.style.top = bt + 'px';

}

document.onmouseup = function(){

document.onmousemove = document.onmouseup = null;

}

return false;

}

}

## 鼠标画框

核心思想：

1.Dom创建元素，并appendChild插入到body中.预先布局框的样式。

2.这里展现框的位置只需要**1.宽高、2.顶点（x,y）**

const body = document.getElementById('body');

document.onmousedown = function(ev){

let disX = ev.pageX;

let disY = ev.pageY;

let div = document.createElement('div');

div.id = 'k';

body.appendChild(div);

document.onmousemove = function(ev){

let w = Math.abs(ev.pageX - disX); //框的宽高

let h = Math.abs(ev.pageY - disY);

div.style.width = w + 'px';

div.style.height = h + 'px';

// console.log('移动的:'+(ev.pageX - disX) , '按下的:'+disX)

let l = Math.min(disX,ev.pageX); //框的顶点位置

let t = Math.min(disY,ev.pageY);

div.style.left = l + 'px';

div.style.top = t + 'px';

}

document.onmouseup = function(){

document.onmousemove = document.onmouseup = null;

div.remove();

}

}

## 定义滚动条

div2.onmousedown = function(ev){

let disY = ev.pageY - div2.offsetTop;

document.onmousemove = function(ev){

let t = ev.pageY - disY;

if(t < 0){

t = 0;

}else if(t > div1.clientHeight - div2.clientHeight){

t = div1.clientHeight - div2.clientHeight;

}

/\*

核心就是这个比例

\*/

let scale = t / (div1.clientHeight - div2.clientHeight);

txt.style.top = scale \* (div3.offsetHeight - txt.scrollHeight) + 'px';

div2.style.top = t + 'px';

}

document.onmouseup = function(){

document.onmousemove = document.onmouseup = null;

}

return false;

# 正则表达式

## 什么是正则

/^(\\d{1,4})(-|\\/)(\\d{1,2})\\2(\\d{1,2})$/;

^(?=.\*\\d)(?=.\*[a-z])(?=.\*[A-Z]).{8,10}$

强大的字符串操作工具

专门用来检索（模糊范围）\*字符串\*的一种规则（工具） 擅长模糊范围字符串的检索

用更少的代码去操作字符串

简写:

//

标准写法:

new RegExp(''||正则,修饰符)

\ + 字母 会有特殊的意义

## 正则关键字

**\**  转义符

例如：

\. 把(点)“.”转义

**\d** 一个数字

**\D** 一个非数字

**修饰符:**

**i 忽略大小写**

**g 找全局**

**“ + ”** 量词:（用来修饰前面的规则的）

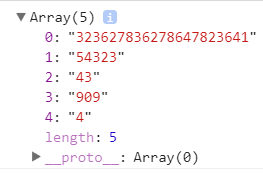
也就是说，\d带“+”就会开启匹配多个数字连在一起的，字符串中连在一起的数字有多少个就会匹配多少个。

例如：

let str = 'ab323627836278647823641c54323 n43kndsjk909fjkdb4';

// console.log(str.match(/a/)); //[a]

console.log(str.match(/\d+/g)); //找全局



**开头和结尾**

^ 这个是开头

$ 这个是结尾

例如：

btn.addEventListener('click',function(){

let tv = t.value;

if(/^(1[89])$|^([2-9][0-9])$|^(10[0-9])$|^110$/.test(tv)){

window.location.href = 'http://www.miaov.com';

}else{

alert('不好意思,未满18禁止入内');

}

});

**排除[^]**

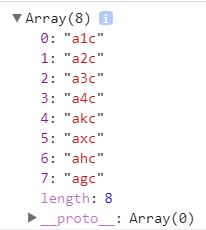
如果在[有^]代表排除

例如1:

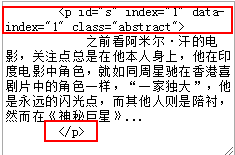
[^v] 排除v

let str = 'a1ca2ca3cavca4cakcaxcahcagc';

console.log(str.match(/a([^v])c/g));



例如2：

/<[^>]+>/ 匹配 => 

**重复子项**

\num 重复一次（注意仅仅是能查找重复一次的，重复多次的不能查找），

\num **+** 查找重复多次的，如果要重复多次加量词即可

子项为第几个子项num就是几

**\w 一个数字|字母|下滑线**

## test方法

检测字符串是否被正则匹配，**类似于includes**

**格式：**

正则.test(字符串);

**返回值（布尔值）：**

检测字符串是否被正则匹配，如果匹配true，否则false

修饰符:

i -> 忽略大小写

例如：

let str = 'Abc';

// console.dir(/x/.test(str));

console.log(/a/i.test(str));

## search方法

查找正则匹配到的下标，**类似于indexOf**

**格式：**

string.search(字符串||正则)

**返回值：**

查找正则匹配到的下标，找到返回下标，没找到返回-1

## match方法

把正则匹配到的字符**push到一个数组**中

**格式：**

string.match(正则):

**返回值:**

匹配上就为数组

没匹配上为null

如果只匹配一个，也就是说用”\d”的情况下，那么数组中有3个属性，

第一个：匹配到的字符

第二个：下标

第三个：整个字符串

length还是1

例如：

let str = 'ab323627836278647823641c54323 n43kndsjk909fjkdb4';

console.log(str.match(/\d/));



## replace方法

replace() 方法用于在字符串中用一些字符替换另一些字符，或替换一个与正则表达式匹配的子串。

**返回值：**

一个新的字符串，是用 replacement 替换了 regexp 的第一次匹配或所有匹配之后得到的。

**格式：**

string.replace(字符串||正则,字符串||cb)

第一个参数：

替换谁

第二个参数:

替换成什么

**cb:默认的前3个参数**

$0 : 当前正则匹配到的字符（没有匹配到的就不显示）

$1 : 当前正则匹配到的字符索引

$2 : 整个字符串

$3 undefined

$4 undefined

## 子项（子串）

从左往右数

(())

(((\d)+)\D+((\d+)\D+)\d+)

例如：

let str = '2018../-/01.----\_\_\_---dsja24';

console.log(str.replace(/(\d+)\D+(\d+)\D+(\d+)/,function($0,$1,$2,$3){

console.log($1,$2,$3); // **2018 01 24**

return $1 + '年' + $2 + '月' + $3 + '日';

})); // 2018年01月24日

## （$0,$1,$2,$3,$4,...）

### 没有子项的时候就是默认模式

即为：$0 : 当前正则匹配到的字符（没有匹配到的就不显示）

（

### 有子项的时候$0后面的就是子项，

**有几个子项就有几个$x**

$1 : 第一个子项

$2 : 第二个子项

$3 : 第三个子项

$4 : 第四个子项

......

$n -1: 第四个子项

）

$n : 当前正则匹配到的字符索引

$n+1 : 整个字符串

$n+2 undefined

$n+3 undefined

## 中括号

正则中指的是范围 || **指定的字符**

0-9 数字的范围

a-z 小写字母的范围

A-Z 大写字母的范围

A-Za-z 大写到小写字母：

## 大括号

量词:

{} 区间范围

{n,m} 最小n,最大为m

{n,} 最小n次,最大不限

{n} 最大最小都是n次

{1,} == +

{0,} == \*

{0,1} == ?

## 其他转义符

\w 一个字符、数字、下划线

\W 非一个字符、数字、下划线

\s 一个空格

\S 一个非空格

**\***

**.**

**?**

例如：

let str = 'a1h^正&\n则\_ca2cacabc';

console.log(str.match(/a\d**?**c/g)); //可以有（最多一个数字）也可以没有

// console.log(str.match(/a**.**+c/g)); //任意字符(除了\n以外)

// console.log(str.match(/a\d**\***c/g)); //可以有数字也可以没有数字

## 边界符

\b 一个边界符

\B 一个非边界符

字母加中文，中文是有边界的

中文与中文之间是没有边界符号。即便是有空格也没有边界。

英文与英文之间如果有空格才有边界。没有空格就没有边界。

例如1：

let str = 'miaov ke ta类似ng';

console.log(str.replace(/\b/g,'|'));

例如2：

<li class="li1 li2 li3"></li>

要用正则匹配其中的li1,就是

let re = new RegExp('\\b'+ ‘li1’ +'\\b');

## 中文区间范围

[\u4e00-\u9fa5] （有事e00，有酒罚5（我））

例如：

console.log(/[\u4e00-\u9fa5]/.test('鞫')) //true

# UNICODE与ASCII的区别

ASCII 是英文码，用来表示英文字符的一种编码规范。

UNICODE 是万国码，为了解决 ASCII码与其他语言编码冲突的问题。不能从扩展ASCII 的角度入手，UNICODE作为一个全新的编码系统应运而生，它可以将中文、法文、德文……等等所有的文字统一起来考虑，为每一个文字都分配一个单独的编码。

# Node简介

下载模块:

npm install 模块的名字 -> 缩写 npm i、

npm i jquery -S （等于下面这个）

npm install jquery --save

卸载模块:

npm uninstall 模块的名字

下载所有依赖:

下载package.json中写入的模块

npm i

安装nrm

npm i nrm -g

nrm test 去查看你的镜像速度

nrm use cnpm(最快的) 切换镜像

# AJAX

Asynchronous JavaScript And XML

异步的js和xml

前后端数据交互的技术

直白点说-ajax只是获取数据的技术

难点:

拿到数据之后怎么做

let arr = [1,2,3,4] 静态数据

被谷歌推广开的

XML：数据格式，类似于html（自定义的html）

不单单指XML还有json格式

**注意:**

在服务器环境中避免出现中文

1.必须要知道接口长什么样

优点:

不刷新页面，就能异步更新数据，给用户的体验非常好

前后端相分离，减少了服务器的压力

参考:

1、最大的一点是页面无刷新，在页面内与服务器通信，给用户的体验非常好。

　 2、使用异步方式与服务器通信，不需要打断用户的操作，具有更加迅速的响应能力。

　 3、可以把以前一些服务器负担的工作转嫁到客户端，利用客户端闲置的能力来处理，减轻服务器和带宽的负担，节约空间和宽带租用成本。并且减轻服务器的负担，ajax的原则是“按需取数据”，可以最大程度的减少冗余请求，和响应对服务器造成的负担。

4、基于标准化的并被广泛支持的技术，不需要下载插件或者小程序。

## AJAX的缺点

不能回退，不能使用历史记录，要使用history

## 传统方式请求数据：表单

<form action="/get"> **//这个指routes文件夹下面的get.js的数据文件**

用户名:<input type="text" name="name"/>

<br />

密码:<input type="password" name="pass"/>

<input type="submit" value="提交"/> **//这里必须提交之后才能获取数据**

</form>

**action:**提交数据的地址，默认是当前页面。

**method :** 数据提交的方式，默认是get方式（前端用什么方法发，后端就用什么方法去取）

1.get

把数据名称和数据值用=连接，如果有多个的话，那么他会把多个数据组合用&进行连接，然后把数据放到url?后面传到指定页面

2.post

post是通过请求头将数据传输给后端的

**enctype :** 提交的数据格式，默认application/x-www-form-urlencoded

例一：post请求

<form action="1.post.php" method="post">

    <input type="text" name="username" />

<input type="text" name="age" />

<input type="submit" value="提交" />

</form>

## AJAX的交互模型:

### 一.打开浏览器 new XMLHttpRequest

const ajax = new **XMLHttpRequest**; //创建一个ajax对象

### 二.在地址栏输入地址open

open有三个参数：

open（method,url,是否异步）

#### open参数1：打开方式method

请求方式分两种get和post

ajax.open('get','/get?name='+this.value);

ajax.open('post','/post');

ajax.setRequestHeader('Content-Type','application/x-www-form-urlencoded')

##### get:

**用途：**

通过url的方式进行请求-把数据放到url上。所以直接暴露出来的数据可以使用get方式。

1.体积有限制，每个浏览器都有最大限制url的规则，不同的浏览器体积大小也不一样。

2.安全方面相对不安全,通过记录是可以拿到用户的信息的

3.**可被浏览器缓存**。

##### post:

1.理论上是无限制大小（但是一般后端童鞋会进行限制）

2.相对get会安全一些

3.需要设置请求头；通过服务器进行传输-把数据放到send中

ajax.setRequestHeader('Content-Type','application/x-www-form-urlencoded')

//Content-Type 这里是指内容类型

// urlencoded这里是指把send里面的内容进行url编码，就像get的open的第二个参数

通过服务器进行传输-把数据放到send中

        xhr.open('post','2.post.php',true);

        //post方式，数据放在send()里面作为参数传递

        xhr.setRequestHeader('content-type', 'application/x-www-form-urlencoded');//申明发送的数据类型

        //post没有缓存问题

        //无需编码

        xhr.send('username=刘伟&age=30');

4.请求不会被缓存

##### post和get的区别

1.体积上，post可以携带无限制大小的信息，

get只能携带少量信息(1024个字节)。

2.安全上，post把信息放到请求头的消息体里，更安全。

get直接把信息放到地址栏里，不安全。

3.请求格式，post的url要分开写，1设置请求头，2数据放进send。

get的url不要分开写。

4.缓存上，post请求不会被缓存。

get请求会被缓存。

5.中文解析上，post需要设置xhr.setRequestHeader('content-type', 'application/x-www-form-urlencoded'); 请求头

get需要设置'+encodeURI('刘伟')+'

真正的验证只有后端来判断

#### open参数2：地址url

这里就是放请求数据的地址。**是放在’?’后面**。

例如：

请求一个图片，就放图片的地址。

##### get缓存问题

**什么是缓存问题？**

当使用get方法请求的时候，会有缓存问题。post没有缓存问题。

后端数据是动态变化的，但是ajax请求中的url地址是固定的，而且浏览器有一个缓存机制，如果每次请求的ajax的url是相同的，那么ajax就会只去读浏览器缓存的东西，这样就会造成获取不到最新数据。

(当一个url第一次被请求成功的时候，浏览器会把这个url缓存到本地，在这个url多次请求的时候，资源直接往本地取。)

**为什么会有缓存问题？**

因为get是获取的意思，前端向后端获取数据，就被浏览器存下。

而post是发送的意思，前端向后端发送数据，都发给后端了，怎么会被浏览器存下呢？

**如何解决？**

1.缓存 在url？后面连接一个随机数，时间戳

浏览器默认情况下具有缓存，其中get方式也会有缓存。

**如何清除默认情况下的get缓存？**

给每一次从服务器获取到的数据加上一个new data时间戳的标志，

因为每次获取到的数据有不同的标志，

所以每次读取的时候就读不到了，

从而达到了清除默认缓存的目的。例如：

cache:true

ajax.open('get',opt.url + '?' + opt.data + '&' + (opt.**cache**?+new Date:''),opt.asy);

##### get乱码问题

2.乱码 编码encodeURI

在ie下，中文需要转成uri编码格式

encodeURI(数据)

xhr.open('get','2.get.php?username='+encodeURI('刘伟')+'&age=30&' + new Date().getTime(),true)

post在IE下，使用中文是没问题的。

因为设置了请求头。

'application/x-www-form-urlencoded');//申明发送的数据类型

#### open参数3：是否异步

true就是异步非阻塞。也就是说这个AJAX没有完成前，后面的代码就可以运行了！

false 就是同步阻塞。也就是说要等这个AJAX请求完成了，后面的代码才会执行！

默认是什么呢？异步还是同步？

//阻塞 -> 同步

//非阻塞 - 异步

### 三.提交地址

这里就是在网络里面向后端请求数据，是肯定要花时间的。

ajax.**send**(); //提交地址

send()里面放给后端的数据，例如：

ajax.send('username=刘伟&age=30');

但是，这个数据需要设置请求头后端才能解码。所以要设置请求头：

xhr.setRequestHeader('content-type', 'application/x-www-form-urlencoded');//申明发送的数据类型

xhr.send('username=刘伟&age=30');

### 四.等待服务器返回内容

ajax.**onload** = function(){

let json = JSON.parse(ajax.**responseText**);

//ajax返回的responseText是一个JSON格式的数据,但是typeof只能显示是一个string类型

//对于JSON格式的数据需要“json转对象”，转成“对象形式”进行操作

//以下为操作数据

if(json.code == 0){

user.className = 'no';

}else if(json.code == 1){

user.className = 'ok';

}

span.innerText = json.msg;

console.log(ajax.responseText,json); //通话

}

JSON不能直接用，要转换成对象或者数组进行数据操作！！！

onload只支持IE9以上浏览器

onreadystatechange能监听ajax执行的步骤过程

#### 使用ajax.onreadystatechange接听

on readystate change : 当readyState改变的时候触发。

onreadystatechange包含 个属性值：

##### 1. readystate请求状态

其中readystate是指：

readyState属性：请求状态

0 （初始化）还没有调用open()方法

1 （载入）已调用send()方法，正在发送请求

2 （载入完成）send()方法完成，已收到全部响应内容

3 （解析）正在解析响应内容

4 （完成）响应内容解析完成，可以在客户端调用了

每完成一个步骤就会就会执行一次函数，每次执行

函数都给你一个数字

0-4 -> 5步,但是0是监听不到的

数字4为步骤全部走完

##### 2. status http状态码

**即：服务器(请求资源)的状态。**

**状态码**

当浏览者访问一个网页时，浏览者的浏览器会向网页所在服务器发出请求。当浏览器接收并显示网页前，此网页所在的服务器会返回一个包含HTTP状态码的信息头（server header）用以响应浏览器的请求。

HTTP状态码的英文为HTTP Status Code。

|  |  |  |
| --- | --- | --- |
| 200 | OK | 请求成功。一般用于GET与POST请求 |
| 201 | Created | 已创建。成功请求并创建了新的资源 |
| 202 | Accepted | 已接受。已经接受请求，但未处理完成 |
| 203 | Non-Authoritative Information | 非授权信息。请求成功。但返回的meta信息不在原始的服务器，而是一个副本 |
| 204 | No Content | 无内容。服务器成功处理，但未返回内容。在未更新网页的情况下，可确保浏览器继续显示当前文档 |
| 205 | Reset Content | 重置内容。服务器处理成功，用户终端（例如：浏览器）应重置文档视图。可通过此返回码清除浏览器的表单域 |
| 206 | Partial Content | 部分内容。服务器成功处理了部分GET请求 |

##### 3. responseText/responseXML返回的内容

**const doc = ajax.responseText**

**获取匹配到的数据，是一个针对数据格式为对象的固定的方法**

**这里的数据格式都是字符串**

**const doc = ajax.responseXML;**

**获取匹配到的数据，是一个针对数据格式为XML的固定的方法**

如果把onreadystatechange放在send之前，那么可以多监听一步

放在send之前是走1234

放在send之后是走234

ajax.onreadystatechange = ready;

function ready(){

alert(ajax.readyState);

if(ajax.readyState === 4){//只能说可以接收到服务器的信息，但是不保证成功还是失败

if(ajax.status >=200 && ajax.status <= 207){

opt.success(ajax.responseText);

}else{

opt.fail(ajax.status);

}

}

}

200-207之间就是后台返回数据了！

#### 使用ajax.onload接听

ajax.**onload** = function(){ // **onload只支持IE9以上浏览器**

const doc = ajax.**responseXML**;

**const person = doc.getElementsByTagName('person');**

**//通过获取JSON中的关键字，将JSON格式转换为对象格式！**

for(let v of person){

html += `<li>我的名字叫${v.children[0].innerHTML},

今年${v.children[1].innerHTML}岁，

性别${v.children[2].innerHTML},

我想说:${v.children[3].innerHTML}

</li>`;

}

ajax.onload = function(){

const doc = ajax.**responseXML**;

let html = '';

const person = doc.getElementsByTagName('person');

for(let v of person){

html += `<li>我的名字叫${v.children[0].innerHTML},

今年${v.children[1].innerHTML}岁，

性别${v.children[2].innerHTML},

我想说:${v.children[3].innerHTML}

</li>`;

}

ul.innerHTML = html;

console.log(person);

}

## Jquery的ajax获取数据

<script src="javascripts/jquery-3.2.1.min.js"></script>

<script>

let user = document.getElementById('user');

let span = document.getElementById('span');

user.onfocus = function(){

this.className = '';

}

user.onblur = function(){

$.ajax({

url:'/get',

data:{

name:user.value

},

dataType:'json',

success:function(data){

console.log(data);

}

});

}

</script>

## 如何看接口

接口:

http://localhost/get

https://sp0.baidu.com/5a1Fazu8AA54nxGko9WTAnF6hhy/su

上面这些从https到最后的都是接口，而不是那个get,或者su

字段:

**?**name**=**yangna

**?**wd**=**2022&cb=fn

1.注意字段的格式！！！

2.“name”, “wd”都是后端提供的字段

接口和字段都是由后端来提供的，一般都有开发文档

## AJAX如何处理XML数据

## ajax可能回出现的问题

在ie下，中文需要转成uri编码格式

encodeURI(数据)

post在IE下，使用中文是没问题的。

因为设置了请求头。

## AJAX的封装

### 最简单的封装

user.onblur = function(){

ajax(

'/get',

'get',

'name='+this.value,

function(data){

console.log(JSON.parse(data));

}

);

}

function ajax(url,method,data,success){

const ajax = new XMLHttpRequest;

ajax.open(method,url+'?'+data,true);

ajax.send();

ajax.onload = function(){

success(ajax.responseText);

}

}

### 终极版封装

function ajax(json){

//默认的配置，目的是为了不报错！

var opt = {

url:'',

data:{},

type:'get',

dataType:'json',

success:function(){},

fail:function(){}

}

//有配置走配置，没配置走默认

// Object.assign(opt,json); //这个只支持高版本，不支持低版本

for(var attr in json){ //for in 支持低版本

opt[attr] = json[attr];

}

var ajax = new XMLHttpRequest;

//把对象转成字符串

var arr = [];

for(var attr in opt.data){

arr.push(attr + '=' + opt.data[attr]);

}

opt.data = arr.join('&');

if(opt.type == 'get'){

/\*

get方式会有缓存,当一个url第一次被请求成功的时候，浏览器会把这个url缓存到本地，在这个url多次请求的时候，资源直接往本地取。

\*/

ajax.open('get',opt.url+'?'+opt.data+ '&' + (opt.cache?+new Date:'')

,true);

ajax.onreadystatechange = ready;

ajax.send();

}else if(opt.type == 'post'){

ajax.open('post',opt.url,true);

ajax.onreadystatechange = ready; ajax.setRequestHeader('Content-Type','application/x-www-form-urlencoded');

ajax.send(opt.data);

}

// ajax.onload = function(){

// opt.success(ajax.responseText);

// }

function ready(){

//alert(ajax.readyState);

if(ajax.readyState === 4){//只能说可以接收到服务器的信息，但是不保证成功还是失败

if(ajax.status >=200 && ajax.status <= 207){

if(opt.dataType === 'json'){

opt.success(JSON.parse(ajax.responseText));

}else if(opt.dataType === 'xml'){

opt.success(ajax.responseXML);

}else{

opt.success(ajax.responseText);

}

}else{

opt.fail(ajax.status);

}

}

}

}

## 传统方式的上传数据

不能监控上传进度

## 使用AJAX监控上传进度

## 跨域（跨源）

### 什么是跨域（跨源）

跨域（跨源）:

跨出某个范围。

源:

域名、协议、端口

同源策略:

同源策略（Same origin policy）是一种约定，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，则浏览器的正常功能可能都会受到影响。可以说Web是构建在同源策略基础之上的，浏览器只是针对同源策略的一种实现

同源：

相同域名、协议、端口

跨源(不同源):

不同域名、协议、端口

域名:

ip地址的一个别名，为了方便记忆

www.baidu.com

www.baidu.com/xxx

www.taobao.com

协议：

http/https 超文本传输协议

file

ftp (文件传输协议)

.....

端口:

类似柜台的服务人员

www.baidu.com:80

[www.baidu.com:88](http://www.baidu.com:88)

### 如何实现跨域（跨源）

#### 1.标准浏览器

利用 XMLHttpRequest() 与后端配合来解决。

前端使用：

XMLHttpRequest()

var ajax = new XMLHttpRequest;

后端设置请求头 ：

php : header('Access-Control-Allow-Origin:\*');

app.all('\*', function(req, res, next) {

res.header("Access-Control-Allow-Origin", "\*");

next();

});

问题 ：低版本的 XMLHttpRequest() 对象不支持与后端配合。

#### 2.后端代理

通过一个服务器文件访问第三方资源，并且这个服务器文件和前端文件同源，从而实现前端文件访问第三方资源的跨域问题。

后端去请求裸露的数据不会有跨域问题

后端先把数据取到当前服务器中，

然后前端通过 正常 的 ajax 请求来获取数据。

#### 3.jsonp

json + padding

##### 什么是jsonp?

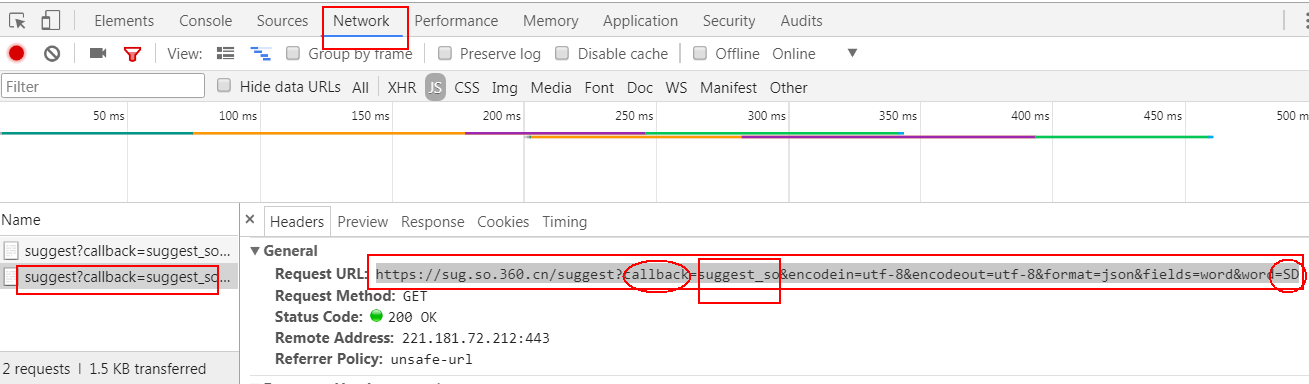
是一种前端获取数据的方式

##### 什么情况下使用JSONP?

首先要确定数据的格式：数据本身就是**函数名 + 括号，**

只有确定了数据格式，才能使用JSONP.

如果是ajax没有函数名 + 括号，就不能使用JSONP



##### 如何实现？

1. 需要在\*全局\*定义一个与数据函数名一样的函数

function fn(data){ //需要在\*全局\*定义一个与数据函数名一样的函数

console.log(data);

}

//如果不是全局的函数那么就可以像下面这样设置

window.fn = function(){}

2.当需要使用数据的时候，创建一个script标签

3.把接口放到script.src中

4.操作数据，就是操作刚才在全局生命的那个与数据函数名一样的函数

例如：

function fn(data){ //需要在\*全局\*定义一个与数据函数名一样的函数

console.log(data);

}

btn.onclick = function(){

// 下面的代码等同于fn([1,2,3,4]);把数据当做实参传了进去

let os = document.createElement('script');

/\*

callback=fn

后端提供了一个让前端配置的参数，传什么函数名

后端就调用这个函数名并且把数据当实参传入。

\*/

os.src = 'http://localhost/jsonp?callback=fn'; //这里fn就是数据函数名吧？

document.getElementsByTagName('head')[0].appendChild(os);

os.remove(); //删除head中添加的script标签，避免浏览器内存占用过多

}

function fn (data){

console.log(data);

let html = '';

data.result.forEach(e => {

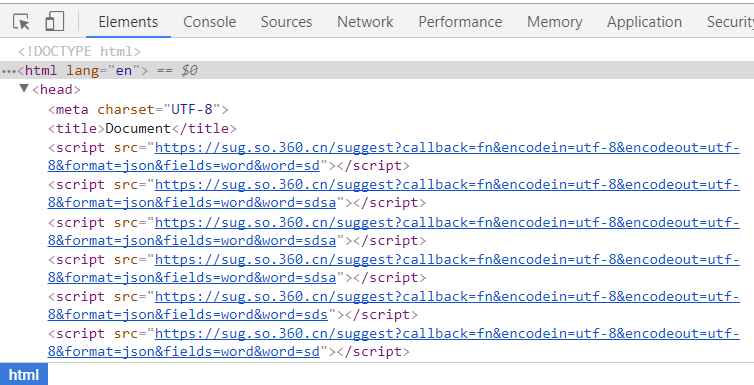
html += "<li>" + e.word + "</li>"

});

ul.innerHTML = html;

}

为什么要用os.remove()



AJAX获取到的数据格式是什么? 一种前端获取的数据格式

ajax没有函数名 + 括号

# cookie

## 什么是cookie?

其实它是一个后端的技术

指某些网站为了辨别用户身份、进行 session 跟踪而储存在用户本地终端上的数据

## 作用

前端使用cookie的主要用途：进行本地存储（前端的数据库）

例如： 猜你喜欢

在老版本浏览器中存储的空间是很有限的，每个域名只能有xx条

一个浏览器能创建的 Cookie 数量最多为 300 个，并且每个不能超过 4KB，每个 Web 站点能设置的 Cookie 总数不能超过 20 个

## 在哪里使用？

**必须在服务器端环境下才能使用**

设置:

document.cookie = 'key=val;'

## 设置cookis生命周期

cookie默认的生命周期:

在浏览器关闭之前

**设置生命周期:**

**document.cookie = 'key=val;expires=生命周期的时间'**

如果val存的是一个对象，那么内部会调用toString方法,就会变成[object Object],所以要用JSON.stringify转一下

### 封装形式：设置cookie

//设置cookie

function setCookie(key,val,time){

if(time){

let t = new Date();

t.setDate(t.getDate() + time);

document.cookie = key+'='+val+';expires='+t;

}else{

document.cookie = key+'='+val;

}

}

### 封装形式：删除cookie

function removeCookie(key,val){

setCookie(key,val,-1);

}

## 获取cookie

**document.cookie**

**一次性会把所有的cookie都获取出来。**

例如：

document.cookie = 'name=yidao;expires=7';

document.cookie = 'user=wangshuo;expires=7';

console.log(document.cookie); // 

**cookie与cookie之间是通过; （分号+空格）来分割的**

### 封装形式：获取cookie

function getCookie(key){

let c = document.cookie.split('; ');

console.log(typeof(c));

let data = c.find(e=>new RegExp('^' + key.trim() + '=').test(e));

return data?data.split('=')[1]:null;

}

# localStorage/sessionstorage本地存储

存储量 5M

提供了非常方便的api

还提供了一些监听事件（onstorage）:当localStorage中的数据发生变化的时候，\*\***所有的兄弟**\*\*页面触发

生命周期:永远存在

sessionstorage和localstorage的区别：

localstorage有永久的生命周期，不会随着页面关闭而关闭。

sessionstorage没有永久的生命周期，随着页面的关闭而关闭。

### 存储的方法：增删改查

**设置存储**

localStorage.setItem('users2','[1,2,3,4]');

可以直接存一个数组

**获取存储**

console.log(localStorage.getItem('users'));

**移除存储**

localStorage.removeItem('users');

**清空存储**

localStorage.clear(); //清空

# storage与cookie的区别

storage用于本地存储，不能用于与后台交互

cookie更加成熟，用于与后台交互。

# Node

## 什么是Node?

node.js -> 服务器语言

JavaScript

https://nodejs.org/en/

node的特性

Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型、单线程，使其轻量又高效。

Node.js 的包管理器 npm，是全球最大的开源库生态系统

都是基于ECMAscript的

### 阻塞和非阻塞

异步:非阻塞 前面的代码不会影响后面代码的执行

同步:阻塞 前面的代码会影响后面代码的执行

**阻塞：**前面的代码执行不出来，后面就没办法执行。例如：

上面一个script双标签;

下面一个script双标签;

如果上面的scrip没有执行出来，下面的script就不会执行，这就叫阻塞。

**非阻塞：**

前面的代码执行不影响后面的代码执行。

    setTimeout(function() {

            alert(1);

        }, 2000);

    alert(2);

## 如何使用?

### require 方法

用于引用Node.js的模块。

require('http');

require('fs');引自带的模块

require('./1');引自己的模块

### listen方法

server.listen(80);

server就代表createServer方法中创建的server.

### createServer方法

创建服务器文件

接收三个参数（request，response，next）

//请求:request （接收客户端的信息）

//响应:response（发送给客户端信息）

### response.write( )/response.end( )方法

write和end是一对,写了write必须写end

response.write('<div>hello,world</div>');

response.end();

### readFile 方法

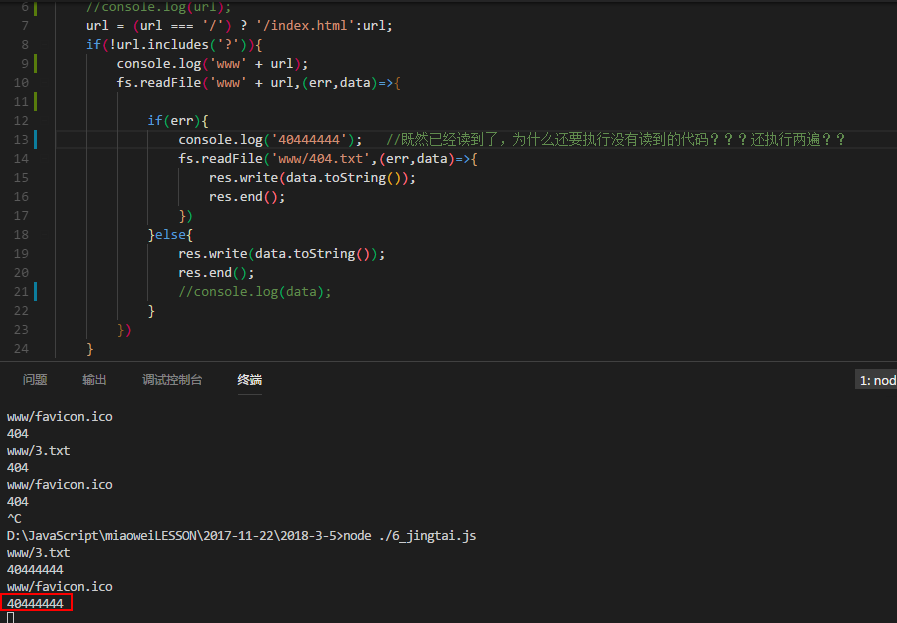
const fs = require('fs');

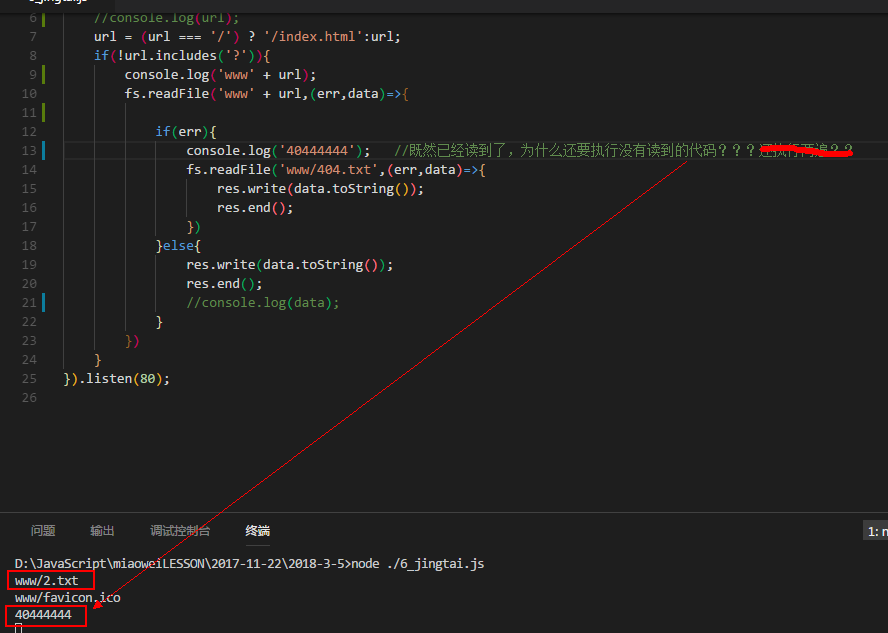
用于读取文件

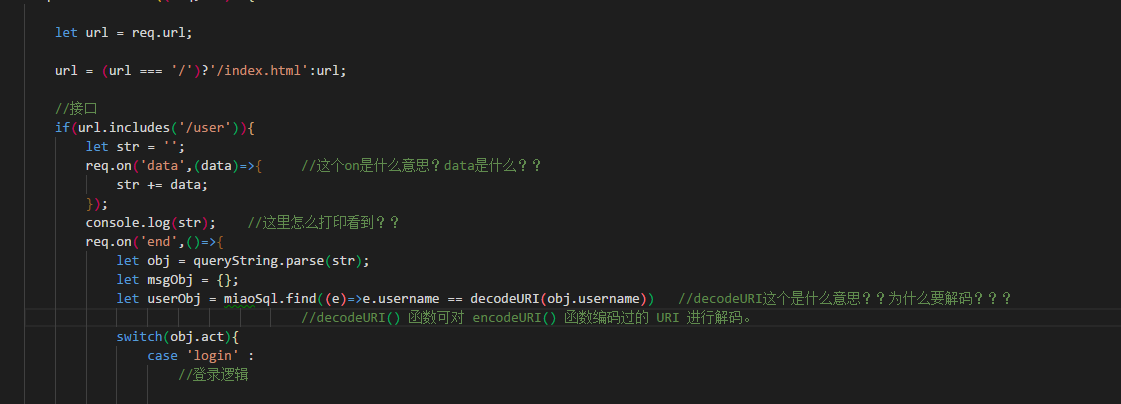
fs.readFile('路径',回调函数(err,data)) err就是失败，data你要的数据

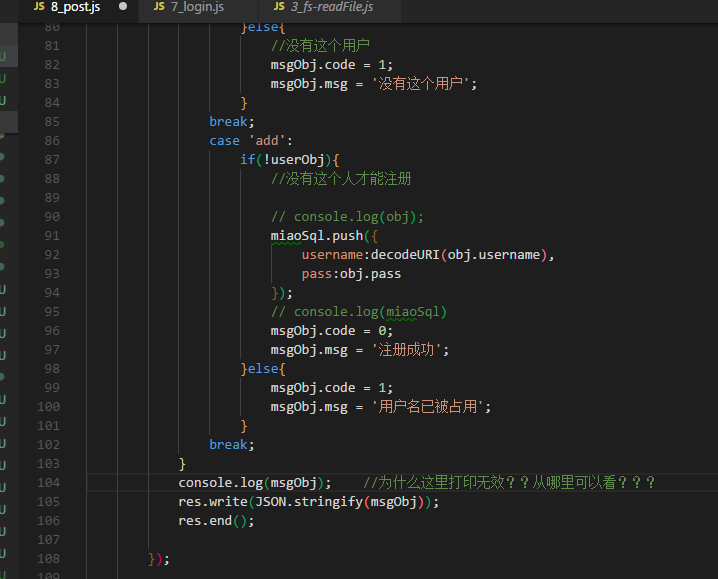
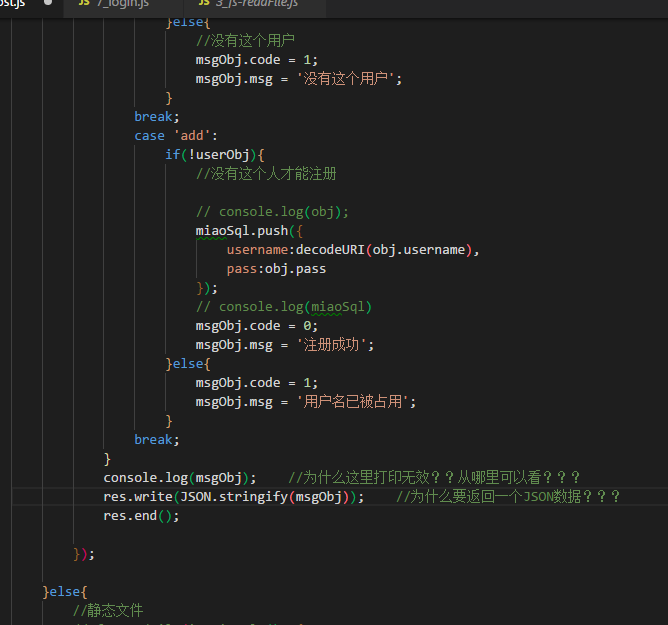
一般数据都是Buffer格式的，要用toString()转一下











### fs.readFileSync(path[, options])方法

try{}

catch{}

[fs.readFile()](http://nodejs.cn/api/fs.html" \l "fs_fs_readfile_path_options_callback) 的同步版本。 返回 path 的内容。

**try** {

let data = fs.**readFileSync**('www'+url);

res.write(data.toString());

res.end();

}

**catch** (error) {

let data = fs.readFileSync('www/404.txt');

res.write(data.toString());

res.end();

}

### writeFile方法

const fs = require('fs');

用于写文件

writeFile('写什么文件',内容,(err)=>{成功})

### unlink方法

删除文件

unlink(“删除哪个文件”,回调(err))

### querystring.parse(str[, sep[, eq[, options]]])

例子，查询字符串 'foo=bar&abc=xyz&abc=123' 被解析成：

{

foo: 'bar',

abc: ['xyz', '123']

}

### 注意

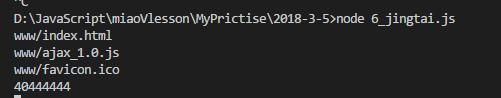
#### 1. events.js:182

events.js:182 这个报错说明这个80端口被node占着呢,忘关了

#### 2.

res.write(JSON.stringify(msgObj)); node没有alert,用这个代替

#### 3.chrome浏览器的bug—favicon.ico



POST用body

get 用 query

npm i mongoose

# 数据库

<https://www.mongodb.com/>

下载安装

## 如何开启

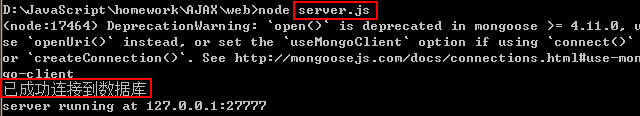


1.在mongodb的bin目录下开启命令行

2.新建一个存放数据的文件夹。

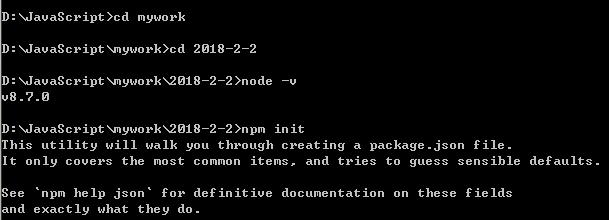
3.开启数据库命令格式：mongod - - dbpath=新建存放数据文件夹的路径 - - port=端口号

4.开启node后台命令：node 服务器文件，例如：

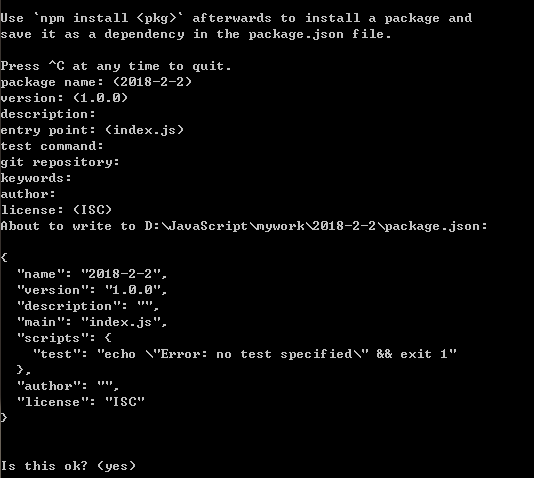


# React/jquery/yarn 环境安装

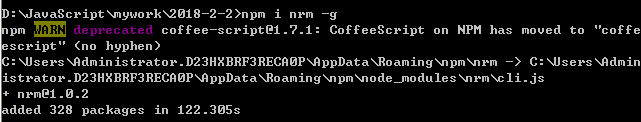
1.CMD 进入到要创建的文件夹,npm初始化



2.然后全部默认回车，进行初始化



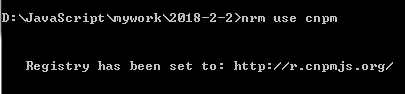
3.安装nrm(nrm就是npm的一个国内镜像节点)



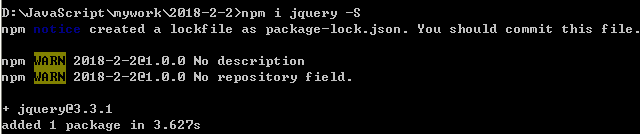
4.测试最快的节点



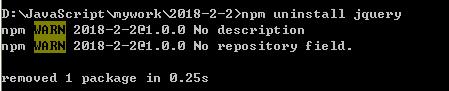
5.启用最快的节点



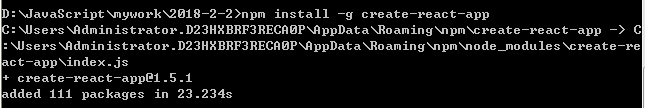
6.下载一个jquery.



7.卸载jquery.

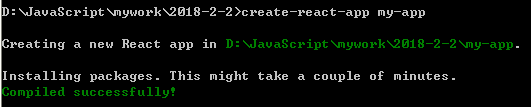


8.安装react(寻找 最快的镜像点进行安装)



9.创建一个自己的项目：（每个项目都要创建，无需联网创建）

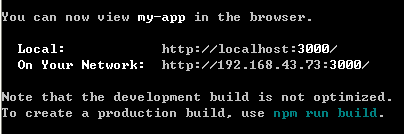
**create-react-app my-app**



9.创建完成my-app这个项目之后，进入创建的项目，执行cd my-app

（my-app是创建的项目名称，可以随意修改）。

10.启动项目,执行 npm start。启动成功，看到以下打印和新的页面。

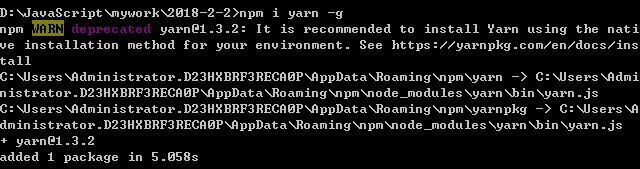


11.终止项目（如果不终止就会导致3000端口一直被占用。）

**ctrl+C**

12.安装yarn（）

npm i yarn -g



13.**npm i**

根据package.json这个文件，补齐环境中的组件。

## 安装插件npm i ... –S 记得要保存-S

# React

react:(facebook出的)

用于构建用户界面的 JavaScript 库

MVC不是框架（属于MVC的V层）

M: Model 模型 一般指的是数据

V: View 视图 html+css

C: control 控制器 状态（暂时当做事件）

## 虚拟(Virtual)DOM

<ul>

<li>1</li>

<li>2</li>

</ul>

虚拟(Virtual)DOM

ul = {

name:'ul',

children:[

{

name:'li',

txt:1

},

{

name:'li',

txt:2

},

{

name:'li',

txt:3

}

]

}

让开发者只用专注数据即可，DOM的东西不用再考虑

生态 -> react-native

## index页面

### 在只有组件的index页面

在只有组件的index页面，

只要把组件导入即可

import App from './reactDemo/App';

### 没有组件的index页面

1. import React from 'react';

2. import ReactDOM from 'react-dom';

//引入ReactDOM是为了后面的使用ReactDOM.render。

3. ReactDOM.render(

第一个参数:组件,

第二个参数:挂在点, document.getElementById('root')

第三个参数:回调函数（挂在完成）

)

## 组件

组件的名字首字母必须大写

继承Component === React.Component

Component是React里面的组件接口，所有自己写的组件都要继承Component。

### 组件的写法

1.import React from 'react';

2. import ReactDOM from 'react-dom';

(用就引用，不用就不引用)

(ReactDom是基于React，所以React必须写)

3. ReactDOM.render(

第一个参数:组件,

第二个参数:挂在点, document.getElementById('root')

第三个参数:回调函数（挂在完成）

)

例如：

import React from 'react';

import ReactDOM from 'react-dom';

ReactDOM.render(

<div>你好,世界！！！！！</div>,

document.getElementById('root')

);

### 标准类声明

遵守MVC规则。

例如：

class App extends Component {

constructor() {

super();

this.state = {

arr: [1,2,3,4,5,6]

}

}

click = (){ //控制器事件

this.setState({

arr: arr

});

}

render() {

return (

<div>

                //这里面写JSX格式的内容。

</div>

);

}

}

#### M模型数据

##### this.state

**这个就是模型数据**

可以用this.state 初始化数据的

只要this.state的数据发生变化，就会更新页面

以下为固定写法：

**constructor()**{ //**这里面是放数据的，所以要有这个constructor**

//写了constructor必须写super

**super();**

**this.state** = { }

}

如果有初始数据就写上面的**constructor，没有初始数据就不写，写了会报错！！！！**

##### setState()

setState() 用来接受一个函数而不是一个对象，例如

 该函数将接收先前的状态作为第一个参数，将此次更新被应用时的props做为第二个参数：

// Correct

this.setState((prevState, props) => ({

counter: prevState.counter + props.increment

}));

上方代码使用了[箭头函数](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions)，但它也适用于常规函数：

// Correct

this.setState(function(prevState, props) {

return {

counter: prevState.counter + props.increment

};

});

##### constructor

constructor只进一次，数据更新只进render，不进constructor

#### 事件、方法控制器相关

事件 = （**形参**）=> {

let {arr,...} = this.state // **赋值结构，因为要获取初始的数据**

**this.setState({ //改变数据**

**})**;

}

##### 作用

无论是事件还是方法控制器，**其目的都是为了修改数据**。

因为每个组件（无论是父组件还是子组件）初始数据是无法传递的，所以方法要挂在对应的数据的组件上。

事件触发的时候，(onClick,onChange,onDoubleClick.....)

必然要改变数据，如果不改变数据就不是react了！

##### 1.事件

React 元素的事件处理和 DOM元素的很相似。但是有一点语法上的不同:

* React事件绑定属性的命名采用**驼峰式写法**，而不是小写。
* 如果采用 JSX 的语法你需要传入一个**函数**作为事件处理函数，而不是一个字符串(DOM元素的写法)

##### 事件this指向问题

添加事件函数的时候，事件函数默认的this为undefined

click(){console.log(this);} //undefined

如果要让事件函数中的this指向类，方法有以下几种:

**1.如果使用create-react-app自带一种方式(默认开启初始化语法器)**

事件 = (形参) => {

this //指向类

}

可以使用属性初始化器来正确的绑定回调函数：

class LoggingButton extends React.Component {

// This syntax ensures `this` is bound within handleClick.

// Warning: this is \*experimental\* syntax.

handleClick = () => {

console.log('this is:', this);

}

render() {

return (

<button onClick={this.handleClick}>

Click me

</button>

);

}

}

**2.在构造函数中绑定。**

在绑定事件函数的时候使用bind(this)

onClick = {this.click.bind(this)}

class Toggle extends React.Component {

constructor(props) {

super(props);

this.state = {isToggleOn: true};

// This binding is necessary to make `this` work in the callback

this.handleClick = this.handleClick.bind(this);

}

handleClick() {

this.setState(prevState => ({

isToggleOn: !prevState.isToggleOn

}));

}

render() {

return (

<button onClick={this.handleClick}>

{this.state.isToggleOn ? 'ON' : 'OFF'}

</button>

);

}

}

ReactDOM.render(

<Toggle />,

document.getElementById('root')

);

**3.在回调函数中使用 [箭头函数](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions)：（避免使用）**

class LoggingButton extends React.Component {

handleClick() {

console.log('this is:', this);

}

render() {

// This syntax ensures `this` is bound within handleClick

return (

<button onClick={(e) => this.handleClick(e)}>

Click me

</button>

);

}

}

使用这个语法有个问题就是每次 LoggingButton 渲染的时候都会创建一个不同的回调函数。在大多数情况下，这没有问题。然而如果这个回调函数作为一个属性值传入低阶组件，这些组件可能会进行额外的重新渲染。**我们通常建议在构造函数中绑定或使用属性初始化器语法来避免这类性能问题。**

##### 阻止事件发生

在 React 中另一个不同是你不能使用返回 false 的方式阻止默认行为。你必须明确的使用 preventDefault。例如，传统的 HTML 中阻止链接默认打开一个新页面，你可以这样写：

<a href="#" onclick="console.log('The link was clicked.'); return false">

Click me

</a>

在 React，应该这样来写：

function ActionLink() {

function handleClick(e) {

e.preventDefault();

console.log('The link was clicked.');

}

return (

<a href="#" onClick={handleClick}>

Click me

</a>

);

}

在这里，e 是一个合成事件

##### 2.赋值解构

let {arr,...} = this.state // **赋值结构，因为要获取初始的数据**

##### 3.C方法控制器

事件 = （）=> {

**this.setState({**

**})**;

}

this.setState({ }) **这个是控制器，因为要更新数据**

每次进行事件触发的时候，就需要更新数据，没有更新的数据就可以不写。

4.回调函数

()=>{}为数据更新之后的回调函数。

db = () => {

this.setState({

dbck:true,

ctxt:this.props.txt

},()=>{

this.refs.nn.selectionStart = this.refs.nn.value.length;

this.refs.nn.focus();

})

};

#### V视图JSX

\* JSX

\* 属性

\* jsx标签也是可以支持属性设置的

\* 基本使用和html/XML类型，在标签上添加属性名 = 属性值，值必须使用""包含

\* 值是可以接收插值表达式的

\*

\* 注意：

\* 1. class：使用className属性来代替

\* 2. style：值必须使用对象

**固定结构：**

**render(){**

**return (**

**)**

**}**

##### renser(){}

框架自带的函数方法：render(){}

才打开页面的时候会进入一次，当**this.setState**数据更改的时候也会进render。

不管是子级还是父级，**只要数据修改了，就会全部render一遍**。、

**注意：**

验证数据的正确性，就在render(){}里面验证。

但是要在let {arr} = this.state;之后取数据:console.log(arr);

##### return( JS XML)

**JSX写法：**

先将静态页面写好，然后复制粘贴到JSX中。

再逐行进行修改。

JS XML**作用**：

1.输入要渲染的 JS XML

2.在花括号中可以写入JS代码，

例如：可以调用事件函数的方法，可以写字符串。

<div> {`${checked}`} </div>

JSX允许在大括号中[嵌入任何表达式](https://doc.react-china.org/docs/introduction-jsx.html%EF%BC%83JSX%E5%B5%8C%E5%A5%97)，所以我们可以在map()中这样使用：

function NumberList(props) {

const numbers = props.numbers;

return (

<ul>

{numbers.map((number) =>

<ListItem key={number.toString()}

value={number} />

)}

</ul>

);

}

JSX**语法**: 一种 JavaScript 的语法扩展

JS XML ->

1.顶层**只能有一个**标签,一般是<div></div>

2.class都要写成className

3.{}

1)**花括号中可以直接写入js代码**

**(基本上花括号里面的就是放JS代码的，例如变量)**

2)[1,2,3,4] ->... 1 2 3 4 (如果是个数组，花括号会扩展开)

4.单标签必须加反斜线 <input />

（正常来讲标签<p></p>是这样的，**而这个反斜线在后面**）

5.推荐在 JSX 代码的**外面扩上一个小括号**，这样可以防止分号自动插入 的bug.

例如：

<div>

<div>1</div>

<div className="active">

{/\* {alert(1)} \*/}

{

/\*arr\*/

}

<input type="text" />

</div>

</div>

##### input表单

**受控组件与非受控组件**

在表单元素中，如果**只设置**了value，无论value的值是常量还是变量，那么会认定为受控组件

输入内容的时候表单元素是不会改变的。

<input type="text"

value={val}

/>

解决方案:

1.给表单元素input加上onChange事件属性,通过“onChange事件函数的方法”去修改this.state的数据。

**在方法中**，通过事件对象ev的target找到元素的value值

**在渲染中**，把this.state数据的变量值直接调用。

**注意：**遇到input要点击事件的时候，就要用onChange,而不能用onClick,否则会报错！

Warning: Failed prop type: You provided a `checked` prop to a form field without an `onChange` handler. This will render a read-only field. If the field should be mutable use `defaultChecked`. Otherwise, set either `onChange` or `readOnly`.

2.使用defaultValue。

defaultValue={val}

这种办法只能改变使input里面的值改变，而无法操作数据，不建议使用。

##### key值

父组件和子组件进行组件通信的时候，必须要在传的属性值中包含key值，否则会报错。

Warning: Each child in an array or iterator should have a unique "key" prop.

Check the render method of `Index`.

Key可以在DOM中的某些元素被增加或删除的时候帮助React识别哪些元素发生了变化。因此你应当给数组中的每一个元素赋予一个确定的标识。

一个元素的key最好是这个元素在列表中拥有的一个独一无二的字符串。

例如：

<li key={i}>{e}</li>

key会作为给React的提示，但不会传递给你的组件。如果您的组件中需要使用和key相同的值，请将其作为属性传递：

const content = posts.map((post) =>

<Post

key={post.id}

id={post.id}

title={post.title} />

);

##### 事件

* React事件绑定属性的命名采用**驼峰式写法**，而不是小写。
* 如果采用 JSX 的语法你需要**传入一个函数**作为事件处理函数，而不是一个字符串(DOM元素的写法)

例如，传统的 HTML：

<button onclick="clickname ()">

Activate Lasers

</button>

React 中稍稍有点不同：

<button onClick={this.clickname}>

Activate Lasers

</button>

##### 向事件处理程序传递参数bind

常我们会为事件处理程序传递额外的参数。例如，若是 id 是一个内联 id，以下两种方式都可以向事件处理程序传递参数：

<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>

<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>

值得注意的是，通过 bind 方式向监听函数传参，在类组件中定义的监听函数，事件对象 e 要排在所传递参数的后面，例如:

class Popper extends React.Component{

constructor(){

super();

this.state = {name:'Hello world!'};

}

preventPop(name, e){ //事件对象e要放在最后

e.preventDefault();

alert(name);

}

render(){

return (

<div>

<p>hello</p>

{/\* Pass params via bind() method. \*/}

<a href="https://reactjs.org" onClick={this.preventPop.bind(this,this.state.name)}>Click</a>

</div>

);

}

}

##### 引用图片

本地图片

方法一：

<img src={require('../img/icon1.png')} alt="" />

require里只能写字符串，**不能写变量。**

方法二：

import search from '../img/search.png'

import user from '../img/user.png'

<img src={search} alt="" />

<img src={user} alt="" />

## 组件 & Props

**组件的作用：**

组件可以将UI切分成一些的独立的、可复用的部件，这样你就只需专注于构建每一个单独的部件。

**什么是组件：**

组件从概念上看就像是函数，它可以接收任意的输入值（称之为“props”），并返回一个需要在页面上展示的React元素。

### Props

根组件，用于使用其他组件的数据

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

const element = <Welcome name="Sara" />;

ReactDOM.render(

element,

document.getElementById('root')

);

1. 我们对<Welcome name="Sara" />元素调用了ReactDOM.render()方法。
2. React将{name: 'Sara'}作为props传入并调用Welcome组件。
3. Welcome组件将<h1>Hello, Sara</h1>元素作为结果返回。
4. React DOM将DOM更新为<h1>Hello, Sara</h1>。

### 组件通信

#### 标签数据的传递（单向数据流）

组件可以选择将其状态作为属性传递给其子组件：

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

这也适用于用户定义的组件：

<FormattedDate date={this.state.date} />

##### 父级向子组件传递数据：

在父级的**render()**文件中，

把引用进来的子级的类名添加属性，

（<PPa txt={e} key={i} id={i} /> 这个就是给类添加属性）

注意：

1.挂属性的标签一定要”< />”前后字符匹配，注意斜线。

2.例外一种格式：

<List {...{

key:i,

txt:e.txt,

checked:e.checked,

id:e.id,

changeChecked:this.changeChecked,

rm:this.rm

}}/>

##### 如何传递 props 到基础构造函数的：

constructor(props) {

super(props);

this.state = {date: new Date()};

}

##### 子级接受父级数据

**第一种接受数据的方法：**

子级的事件函数中通过this.props接受上面添加的属性数据、方法等，

（this是指子级的类名，props是那个类名的属性）

render(){

let {checked,txt} = this.props}

**第二种接受数据的方法：**

通过以下的写法，把父级的数据变成了子级的.

(这里所谓的数据，是指挂在render属性下的数据)

两个数据（子数据与父数据）并不干扰。

子级的constructor中的props只会走一次,

会接受父级传过来的第一次数据，就算父级数据更新

也不会再次更新props了

constructor(props){

super(props);

this.state = {

ondb:false,

ctxt:''

}

}

##### 子级向父级传递数据

接受了数据之后调用父级的方法。

**如何实现：**

**父级属性名(子级的实参)**

(但是代码中是写父级的属性名,其实就是调用这个属性名的函数)

例如：

obj={

function(){alert(1)}

}

调用function就是obj();

(代码中写的属性名是父级的，实参是子级的)。

**在哪里执行：**

执行是在父级的事件、方法控制器内执行。

注意：

**每次数据改变都会render() 无论是子级的数据还是父级的数据改变**

## 自动加载

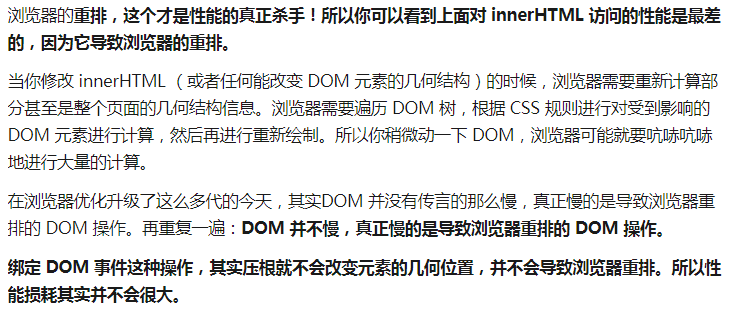
放在index.js中的最下面

if(module.hot){

module.hot.accept();

}

## 为什么要避免DOM操作？



## React 路由

## 什么是路由？

前端路由：

简单理解就是切换页面（组件）

hash

histroy

3.前端路由的使用场景？

前端路由更多用在单页应用上, 也就是SPA, 因为单页应用, 基本上都是前后端分离的, 后端自然也就不会给前端提供路由。

4.前端路由优缺点

优点：

1.从性能和用户体验的层面来比较的话，后端路由每次访问一个新页面的时候都要向服务器发送请求，然后服务器再响应请求，这个过程肯定会有延迟。而前端路由在访问一个新页面的时候仅仅是变换了一下路径而已，没有了网络延迟，对于用户体验来说会有相当大的提升。

2.在某些场合中，用ajax请求，可以让页面无刷新，页面变了但Url没有变化，用户就不能复制到想要的地址，用前端路由做单页面网页就很好的解决了这个问题

缺点：

使用浏览器的前进，后退键的时候会重新发送请求，没有合理地利用缓存

## 如何使用路由

### index.js注意点

引入react-router-dom



1. 导入组件的时候要注意！

import { Router, Route, Link } from 'react-router'

import { //导入工具的时候要注意！！！！

BrowserRouter as Router,

// HashRouter as Router,

Route,

Link

} from 'react-router-dom'

2.ReactDOM.render里面加<Router></Router>标记对！

给根组件包(也就是index.js)加上<Router>标签对（只需要一个Router即可）.

ReactDOM.render( //DOM渲染的时候要<Router></Router>,这里也要注意！

<Router>

<App />

</Router>

,

document.getElementById('root')

);

### router.js路由

**1.导入组件、样式（组件、样式有几个就导入几个）**

import './css.css';

import About from './about';

import Home from './home';

#### Link和Route的区别

**Link**

link可以改变url地址。相当于react版的a标签。

**Route**

Route中的一个属性path=’/about’，就是读取link改变的url地址。

读取之后，用Route后面的component={类名}去渲染。

**2.使用Link和Route**

#### Link

如果想改变url地址只能用link.不是写在路由中，而是写在每个小组件中。

相当于react版的a标签。

因为在被浏览器渲染的时候被解析成a标签，在被解析之后，Link就相当于a标签。

所以css中不用给Link添加样式，只需要给a添加样式。

**格式：**

Link 通过设置to="路径" ，改变URL地址

<Link to="/about" >详情页</Link>

1.第一种格式

to=string -> to="/about"

2.第二种格式，**这种格式可以和其他组件进行数据传递！这样可以传入数据。**

例如：

<Link to={{

pathname:'/about',

state:{

num:this.state.num

}

}} >详情页</Link>

**需要在对应的组件中的this.props的location中去获取to传递的数据，**例如：

let {**location**:{state:{num}}} = this.props;

to=object -> to={{

pathname:'/about',

search: '?sort=name',

hash: '#the-hash',

state: { fromDashboard: true }

}}

如果在Link中写上replace,是把之前的路由记录替换（点击浏览器回退键，不能回退）

<Link to="/about" replace>详情页</Link>

#### Route

Route 是用于声明路由映射到应用程序的组件层。渲染组件。

**格式：**

<Route path="/about" component={About} />

**path**后面的就是获取link改变的URL地址中的值，例如

<Route path="/about" component={About} />

这种情况下，

**component** = {类名} ，当匹配到 URL 时，单个的组件会被渲染。

在子组件中class的类名是什么，这里就放什么。

<Route exact path="/" component={Home}/>

这里的“/”是指浏览器中的根目录。

**精确模式**

exact:(精确模式)

没有写exact：

<Route path="/" component={Home}/>

<Route path="/about" component={About}/>

那么既会出现about页面，也会出现Home页面

如果你写exact

<Route **exact** path="/" component={Home}/>

<Route path="/about" component={About}/>

那么只会出现about或者Home页面

**另一种格式**

Route可以不写component

可以使用render

render={({mathch})=>{

return <Tab1 />

}}

render中只要匹配到就会进这个函数中，并且返回对应的组件

<Route render={()=>{

return <Home2/>

}}/>

**重定向Redirect**

没有重定向的Route

<Route path="/about" render={(props)=>{

return <About url={props}/>

}}/>

重定向的Route

<Route path="/about" render={(props)=>{

return <Redirect to="/about2" />;

//return <About url={props}/>

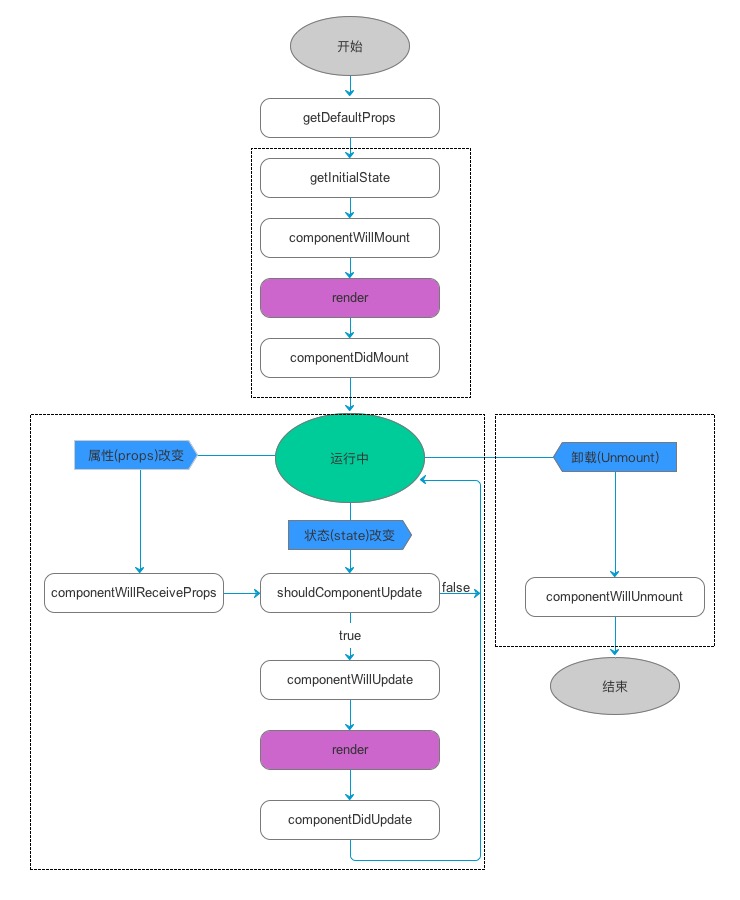
}}/>

<Route path="/about2" render={(props)=>{

return <About2 url={props}/>

}}/>

## 生命周期



* 第一阶段：是组件第一次绘制阶段，如图中的上面虚线框内，在这里完成了**组件的加载和初始化**；
* 第二阶段：是组件在运行和交互阶段，如图中左下角虚线框，这个阶段组件可以处理用户交互，或者接收事件**更新界面**；
* 第三阶段：是组件卸载消亡的阶段，如图中右下角的虚线框中，这里做一些**组件的清理**工作。

#### 1. getInitialState

首先调用 getInitialState()，来初始化组件的状态。

#### 2. componentWillMount

然后调用 componentWillMount()，在第一次绘制render 之前。可以在这里做一些业务初始化操作，也可以**设置组件状态**。这个函数在整个生命周期中**只被调用一次**。

#### 3. componentDidMount

然后调用 componentDidMount()，通知组件已经加载完成。这个函数调用的时候，其虚拟 DOM 已经构建完成。可以在这个函数开始获取？？**其中的元素或者子组件**了。需要注意的是，React-native 框架是先调用子组件的 componentDidMount()，然后调用父组件的函数。从这个函数开始，就可以**和 JS 其他框架交互**了，例如设置计时 setTimeout 或者 setInterval，或者发起网络请求。这个函数也是只被**调用一次**。这个函数之后，就进入了稳定运行状态，等待事件触发。

#### 4. componentWillReceiveProps

如果组件收到新的属性（props），就会调用 componentWillReceiveProps()，

输入参数 nextProps 是即将被设置的属性，旧的属性还是可以通过 this.props 来获取。在这个回调函数里面，你可以根据属性的变化，通过调用 **this.setState()**来**仅仅更新你的组件状态**.这里调用**更新状态并不会触发render() 调用**。

例如：

componentWillReceiveProps: function(**nextProps**) { // nextProps可以作为参数传进来，但是好像this.props在这里不能作为参数传

this.setState({

likesIncreasing: nextProps.likeCount > this.props.likeCount //this.props是随用随取

});

}

#### 5. shouldComponentUpdate

当组件接收到新的属性和状态改变的话，都会触发调用 shouldComponentUpdate(...)，

默认情况下，这个函数永远返回 true 用来保证数据变化的时候 UI 能够同步更新。在大型项目中，你可以自己重载这个函数，通过检查变化前后属性和状态，来决定 UI 是否需要更新，能有效提高应用性能。

shouldComponentUpdate() {

alert("shouldComponentUpdate");

return true; // 记得要返回true

}

#### 6. componentWillUpdate

如果 1.组件**状态或者属性改变**，

2.并且**上面的 shouldComponentUpdate(...) 返回为 true**，

就会调用 componentWillUpdate()。

这个回调中，可以做一些在更新界面之前要做的事情。

需要特别注意的是，在这个函数里面，你就**不能使用 this.setState 来修改状态**。

这个函数调用之后，就会把 nextProps 和 nextState 分别设置到 this.props 和 this.state 中。紧接着这个函数，就会**调用 render() 来更新界面了**。

#### 7. componentDidUpdate

调用了 render() 更新完成界面之后，会调用 componentDidUpdate() 来得到通知。

因为到这里已经完成了属性和状态的更新了，此函数的输入参数变成了 prevProps 和 prevState。

#### 8. componentWillUnmount

组件要被从界面上移除的时候，就会调用 componentWillUnmount()

在这个函数中，可以做一些组件相关的清理工作，例如取消计时器、网络请求等。

区别

componentWillMount、componentDidMount

和componentWillUpdate、componentDidUpdate可以对应起来。

区别在于，前者只有在挂载的时候会被调用；而后者在以后的每次更新渲染之后都会被调用。

### 触发render的方式

在react中，触发render的有4条路径。

**以下假设shouldComponentUpdate都是按照默认返回true的方式。**

1. 首次渲染Initial Render
2. 调用this.setState （并不是一次setState会触发一次render，React可能会合并操作，再一次性进行render）
3. 父组件发生更新（一般就是props发生改变，但是就算props没有改变或者父子组件之间没有数据交换也会触发render）
4. 调用this.forceUpdate

## Refs & DOM

ref:可以给组件或者元素加标识，方便快速查找并操作它

<List ref="name"> //这样就是给组件加标识

使用:

this.refs.name //这样就是去使用上面加标识的组件

**作用：**

某些情况下你需要在典型数据流外强制修改子代。要修改的子代可以是 React 组件实例，也可以是 DOM 元素。

**为DOM元素添加ref**

class CustomTextInput extends React.Component {

constructor(props) {

super(props);

this.focus = this.focus.bind(this);

}

focus() {

// 直接使用原生 API 使 text 输入框获得焦点

this.textInput.focus();

}

render() {

// 使用 `ref` 的回调将 text 输入框的 DOM 节点存储到 React

// 实例上（比如 this.textInput）

return (

<div>

<input

type="text"

ref={(input) => { this.textInput = input; }} />

<input

type="button"

value="Focus the text input"

onClick={this.focus}

/>

</div>

);

}

}

React支持一个可以附加到任何组件的特殊属性ref。ref属性可以是一个字符串或一个回调函数。当ref属性是一个回调函数时，函数接收底层DOM元素或类实例（取决于元素的类型）作为参数。这使你可以直接访问DOM元素或组件实例。

不要过度使用 Refs。如果你发现自己经常在应用程序中使用refs来“搞事情”，请考虑使用[状态提升](https://doc.react-china.org/docs/lifting-state-up.html)。

## React 注意

App.test.js是一个没有用的文件，不用管！

## Redux

什么是redux?

**处理复杂的state （状态），Redux 试图让 state 的变化变得可预测**

import { createStore } from 'redux';

function counter(state=0, action) { //这里是reducer

let {type} = action;

switch (type) {

case 'INCREMENT':

return state + action.value;

default:

return state;

}

}

let store = createStore(counter);

$(document).click(ev=>{ //这里是action

store.dispatch({ type: 'INCREMENT',value:6 });

});

store.subscribe( ()=>{

let state = store.getState();

console.log(state);

} );

思路：

1. 要想更新 state 中的数据，你需要发起一个 action。(**唯一改变 state 的方法就是触发**[action](http://www.redux.org.cn/docs/Glossary.html" \l "action)**)**

2. 为了把 action 和 state 串起来，开发一些函数，这就是 reducer。

reducer 只是一个接收 state 和 action，并返回新的 state 的函数。

3. **整个应用的 [state](http://www.redux.org.cn/docs/Glossary.html" \l "state) 被储存在一棵 object tree 中，并且这个 object tree 只存在于唯一一个 [store](http://www.redux.org.cn/docs/Glossary.html" \l "store) 中。**

## Action

1.一个action至少要有一个type字段！

2.在action里面声明数据，以便于在reducer里面操作数据！

## state

## reducer

1.reducer是一个纯函数，确定的输入得到确定的输出！

2.reducer需要两个值，即**state的初始值**和action

3.不要在里面调用非纯函数,例如：

function counter(state={value:0}, action) {

let {type} = action;

switch (type) {

case 'INCREMENT':

// return state + action.value;

// state.value+=6; //这里就是非纯函数，这样是错误的

// return state;

return Object.assign({},state,{value:state.value+6});

//这里就是纯函数，这样是正确的（注意顺序）

default:

return state;

}

}

如上所述，如果使用非纯函数，

就会使当前的state和过去的state相同，这样在react里面看不到数据的变化就不会触发组件的更新。

## store

1.store.getState()

获取state

2.store.subscribe

只要发起了新的action, store.subscribe就会执行。

3.store.dispatch

本质就是更新state

只要发起了新的action, store.dispatch就会执行。

store.dispatch({ type: 'INCREMENT',value:6 });

## 数据流

**严格的单向数据流**是 Redux 架构的设计核心

**1.调用** [store.dispatch(action)](http://www.redux.org.cn/docs/api/Store.html" \l "dispatch)。

**2.Redux store 调用传入的 reducer 函数。**

**3. 根 reducer 应该把多个子 reducer 输出合并成一个单一的 state 树。**

**4. Redux store 保存了根 reducer 返回的完整 state 树。**

# webpack

1. npm install webpack –g （npm）

2.在根目录下创建webpack.config.js

3.创建webpack配置文件

{

"name": "wp",

"version": "1.0.0",

"description": "",

"main": "es6.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

**"start": "webpack" //这里等会npm start**

},

"author": "",

"license": "ISC",

"dependencies": {

},

"devDependencies": {

"babel-core": "^6.26.0",

"babel-loader": "^7.1.4",

"babel-preset-env": "^1.6.1",

"webpack": "^3.5.1" //要这个版本才能运行，高版本暂时用不了

}

}

4.创建出入口文件

在index里面引入ppa

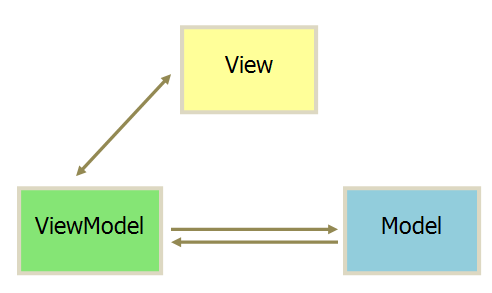
在···里面导出app

# Vue

## 什么是vue?

Vue 的核心库只关注视图层,是mvvm框架。

它采用双向绑定（data-binding）：View的变动，自动反映在 ViewModel，反之亦然。



**所以，变成人员只需要关注数据数据数据！！！数据的处理！！！！**

## 引入vue.js

**<script src="vue.js"></script>**

引入的VUE.js文件需要放在所有javascript文件之前，否则就无法使用

## VUE实例

### 创建VUE实例

每个 Vue 应用都是通过用 Vue 函数创建一个新的 **Vue 实例**开始的：

var vm = new Vue({

// 选项

})

一个 Vue 应用由一个通过 new Vue 创建的**根 Vue 实例**，以及可选的嵌套的、可复用的组件树组成。

### 选项：挂载点el

**使用选项el告诉程序这个实例挂载在哪里**

**el后面跟CSS的ID选择器 或者 类选择器**

#### 声明式渲染(VUE就是用这个)

只需要告诉程序where,what

**使用选项el告诉程序这个实例挂载在哪里**

**el后面跟CSS的ID选择器**

声明式地将数据渲染进 DOM 的系统

<div id="app">

{{ message }}

</div>

var app = new Vue({

**el: '#app',**

data: {

message: 'Hello Vue!'

}

})

#### 命令式渲染

需要告诉程序where,what,how。

### 选项：数据data

Vue 的**响应式系统**中加入了其 **data**对象中能找到的所有的属性。当这些**属性的值发生改变时**，视图将会产生“响应”，即匹配更新为新的值。

注意：数据中的每一项之间都要用”,” 逗号隔开！！养成好习惯，每一个数据都写写分割符，最后一项也要写！

设置属性也会影响到原始数据，反之亦然？？？

**data里面的属性可以直接渲染**

**如何禁止数据改变的响应渲染？**

**Object.freeze(obj)**

<body>

<div id="app">

<p>{{ foo }}</p>

<!-- 这里的 `foo` 不会更新！ -->

<button v-on:click="foo = 'baz'">Change it</button>

</div>

<script>

var obj = {

foo:'bar'

}

**Object.freeze(obj)**

new Vue({

el:'#app',

data:obj

});

</script>

</body>

**实例属性与方法？**

除了数据属性，Vue 实例还暴露了一些有用的实例属性与方法。它们都有前缀 $，以便与用户定义的属性区分开来

### 选项：methods方法

### 选项：生命周期(组件的钩子函数)

生命周期的函数就像数据data一样，写在实例中。

new Vue({

data: {

a: 1

},

created: function () {

console.log('a is: ' + this.a)

}

})

注意：

1.生命周期钩子的 this 上下文指向调用它的 Vue 实例。

2.不要在选项属性或回调上使用[箭头函数](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/Arrow_functions" \t "_blank)，比如 created: () => console.log(this.a) 或 vm.$watch('a', newValue => this.myMethod())。因为箭头函数是和父级上下文绑定在一起的，this 不会是如你所预期的 Vue 实例

#### 生命周期beforeCreate

#### 生命周期created

用来在一个实例被创建之后执行代码

new Vue({

data: {

a: 1

},

created: function () {

// `this` 指向 vm 实例

console.log('a is: ' + this.a)

}

})

渲染这个组件会调用一次这个生命周期函数。

复用这个组件，这个函数不会再次被调用了。也就是说只有在生成的时候调用一次。

#### 生命周期mounted

**可以在mounted这个生命周期中，查看data或者props的数据**

el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用该钩子。如果 root 实例挂载了一个文档内元素，当 mounted 被调用时 vm.$el 也在文档内。

注意 mounted **不会**承诺所有的子组件也都一起被挂载。如果你希望等到整个视图都渲染完毕，可以用 **[vm.$nextTick](https://cn.vuejs.org/v2/api/" \l "vm-nextTick)** 替换掉 mounted：

|  |
| --- |
| mounted: function () {  this.$nextTick(function () {  // Code that will run only after the  // entire view has been rendered  }) } |

**生命周期函数mounted可以调用methods中的方法**

例如：

生命周期函数mounted中可以放‘$on’监听函数，在监听函数中调用methods中的方法。

im.$on('imclick', function(event) {

close(context, event);

});

**生命周期的钩子函数中，可以修改data数据，并造成重新渲染。**

例如：

mounted:function(){

let context = this;

im.$on('imclick', function () {

context.showMan = false;

});

}

### 选项：模板语法template

### 选项：计算属性computed

**作用**

对数据的进一步处理，应该用计算属性。

**是选项data / 继承数据props的另外一种存在方式。**

（对于任何复杂逻辑，你都应当使用**计算属性**）

**格式：**

computed:{key:value }

**key值即代替data选项中的key值**的，放在双括号中的key值。

key 在computed选项中不能和data/props相同，否则会报错！

value则需要写成一个函数，function(){ return ...... },将返回值赋值给key值。

changeNum:function(){

return this.arr.filter(e=>!e.checked).length;

},

也可以写成：

changeNum (){

return this.arr.filter(e=>!e.checked).length;

},

例如：

当不使用computed选项时，放在双括号里面进行数据处理。会显得比较臃肿，不容易理解。所以诞生了computed属性。

<body>

<div id="app">

{{ message.split('').reverse().join('')}}

</div>

<script src="VUE.js"></script>

<script>

new Vue({

el:'#app',

data: {

**message**:'Hello1111'

}

})

</script>

</body>

把以上代码中的双大括号中（mustache）的复杂逻辑应该放在computed选项中写。例如：

<body>

<div id="app">

{{ message }}

{{ reversedMessage }}

</div>

<script src="VUE.js"></script>

<script>

var vm = new Vue({

el:'#app',

data: {

message:'Hello1111'

},

computed:{

**reversedMessage**: function (){ //这里名字不能和data相同

return this.message.split('').reverse().join('')

}

}

})

</script>

</body>

#### 计算属性的getter函数和setter函数

这个computed选项中的声明函数**就是实例化对象vm**中的**属性**

**使用“.”就可以直接getter到**

**<script>**

var vm = new Vue({

el:'#app',

data: {

message:'Hello222'

},

computed:{

reversedMessage: function (){

return this.message.split('').reverse().join('')

}

}

})

**console.log(vm.reversedMessage);**

**console.log(vm.message);**

</script>

**使用get这个api也能获取到并返回给{{ }} 中**

<body>

<div id="app">

{{ fullName }}

</div>

<script src="VUE.js"></script>

<script>

var vm = new Vue({

el:'#app',

data: {

firstName: 'Foo',

lastName: 'Bar',

// fullName: 'Foo Bar'

},

computed:{

fullName: {

// getter

**get**: function () {

return this.firstName + ' ' + this.lastName

},

// setter

**set**: function (newValue) {

var names = newValue.split(' ')

console.log(names); //["John", "Doe"]

this.firstName = names[0]

this.lastName = names[names.length - 1]

}

}

}

})

</script>

</body>

#### 通过方法实现计算属性

我们可以通过在表达式中调用方法来达到同样的效果。

<body>

<div id="app">

{{ **reversedMessage()** }}

</div>

<script src="VUE.js"></script>

<script>

var vm = new Vue({

el:'#app',

data: {

message:'Hello222'

},

**methods**:{

reversedMessage: function (){

return this.message.split('').reverse().join('')

}

}

})

</script>

</body>

#### 计算属性和方法的区别

计算属性是：只有在它的相**关依赖发生改变时**才会重新求值。

(相关依赖不好改，找不到，最好别动)！

也就是说，通过methods是无法改变computed的数据的。

方法是：每次**触发重新渲染（计算属性的数据）时**，调用方法将**总会**再次执行函数。

例如：在HTML结构中使用 {{ }} ，来触发重新渲染。

**计算属性是基于它们的依赖进行缓存的**

**计算属性具有缓存特性，而data没有缓存特性。**

我们为什么需要缓存？

假设我们有一个性能开销比较大的的计算属性 **A**，它需要遍历一个巨大的数组并做大量的计算。然后我们可能有其他的计算属性依赖于 **A** 。如果没有缓存，我们将不可避免的多次执行 **A** 的 getter！？？？

### 选项：侦听属性watch

监控data选项中的属性，当这个属性对应的值发生变化就会执行函数。

例如：

    watch:{

        list:function(){

            store.save("miaov-new-class",this.list);

        }

    },

一般监听时是不能监听到对象属性值的变化的，数组的值变化可以听到。

**深监控：**

要想监控list这个属性里面的属性的值变化，就要用深监控。

list必须为一个对象{},不能是一个数组。数组中的元素的属性名好像能监控到，但是属性值监控不到。

    watch:{

        list:{

**handler**:function(newValue,oldValue){

newValue是改变数据之**后**的数据，

OldValue是改变数据之**前**的数据，

                store.save("miaov-new-class",this.list);

            },

**deep:true**

        }

    },

监控路由信息的变化：

watch:{

$route(){

console.log(this.**$route**.params.userId)

}

},

### 选项：[template](https://cn.vuejs.org/v2/api/" \l "template" \o "template)

* **类型**：string
* **详细**：

一个字符串模板作为 Vue 实例的标识使用。模板将会 **替换** 挂载的元素。挂载元素的内容都将被忽略，除非模板的内容有分发插槽。

如果值以 # 开始，则它将被用作选择符，并使用匹配元素的 innerHTML 作为模板。常用的技巧是用 <script type="x-template"> 包含模板。

template作为选项时，**不可以放自定义标签**，

只能放**html标准标签**和**{{ }}**

**template只能放在组件中使用，而不能放在根实例中使用。**

**格式：**

template: **`** <div>...

</div> **`** , //这里面用上标点来把结构引起来。

如果你熟悉虚拟 DOM 并且偏爱 JavaScript 的原始力量，你也可以不用模板，**[直接写渲染 (render) 函数](https://cn.vuejs.org/v2/guide/render-function.html)**，使用可选的 JSX 语法。？？？

**例如:**

components.getFileMessage = function(resolve, reject) {

var options = {

name: 'groupfile', //name

// props: ['message', 'keyword'],

template: '#rong-template-file',

+

另外一个JS文件将HTML模板引入。

<script id="rong-template-group-files" type="text/x-template">

<!--#include file="conversation/group-files.html" -->

</script>

#### 文本

数据绑定最常见的形式就是使用“Mustache”语法 (双大括号) 的文本插值：

<span>Message: {{ msg }}</span>

**如何禁止数据改变的插值渲染？**

通过使用 **[v-once 指令](https://cn.vuejs.org/v2/api/" \l "v-once)**，你也能执行一次性地插值，当数据改变时，插值处的内容不会更新。

#### 特性

Mustache 语法不能作用在 HTML 特性上，遇到这种情况应该使用 [v-bind 指令](https://cn.vuejs.org/v2/api/" \l "v-bind)

### 全局钩子函数

 Vue.nextTick 和 vm.$nextTick

let el = document.getElementsByClassName('rong-conversation-content')[0];

let move = function (){

el.scrollTop = 10000;

}

Vue.nextTick(move)

#### Vue.nextTick

nextTick是全局vue的一个函数，在vue系统中，用于处理dom更新的操作。vue里面有一个watcher，用于观察数据的变化，然后更新dom，vue里面并不是每次数据改变都会触发更新dom，而是将这些操作都缓存在一个队列，在一个事件循环结束之后，刷新队列，统一执行dom更新操作。   
通常情况下，我们不需要关心这个问题，而如果想在DOM状态更新后做点什么，则需要用到nextTick。在vue生命周期的created()钩子函数进行的DOM操作要放在Vue.nextTick()的回调函数中，因为created()钩子函数执行的时候DOM并未进行任何渲染，而此时进行DOM操作是徒劳的，所以此处一定要将DOM操作的JS代码放进Vue.nextTick()的回调函数中。而与之对应的mounted钩子函数，该钩子函数执行时所有的DOM挂载和渲染都已完成，此时该钩子函数进行任何DOM操作都不会有个问题。

vue更新数据是异步的

1.0 中确实是通过 v-el:xxx 标记 dom 然后在组件里通过 this.$els.xxx 就可以获得这个 dom 对象了

$nextTick(() => {}) 与DOM相关操作写在该函数回调中，确保DOM已渲染

什么是Vue.nextTick()

官方文档解释如下：

在下次 DOM 更新循环结束之后执行延迟回调。在修改数据之后立即使用这个方法，获取更新后的 DOM。

所以就衍生出了这个获取更新后的DOM的Vue方法。所以放在Vue.nextTick()回调函数中的执行的应该是会对DOM进行操作的 js代码，

什么时候需要用的Vue.nextTick()

你在Vue生命周期的created()钩子函数进行的DOM操作一定要放在Vue.nextTick()的回调函数中。原因是什么呢，原因是在created()钩子函数执行的时候DOM 其实并未进行任何渲染，而此时进行DOM操作无异于徒劳，所以此处一定要将DOM操作的js代码放进Vue.nextTick()的回调函数中。与之对应的就是mounted钩子函数，因为该钩子函数执行时所有的DOM挂载已完成。使用时机：el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用该钩子函数，此时页面并未全部渲染。

在某个动作有可能改变DOM元素结构的时候，对DOM一系列的js操作都要放进Vue.nextTick()的回调函数中

#### vm.$nextTick

用vue 获取动态元素的宽度。要放在this.$nextTick里面！

this.$nextTick(function () {

let C = context.$refs.delDiv.clientWidth;

console.log(C)

})

这里的this是指根实例。

如果用DOM操作就获取不到vue中的元素宽度，会得到0.

let C = document.getElementsByClassName('delDiv')[0];

console.log(C) //可以获取到元素

console.log(C.clientWidth) //但是获取到的元素宽度就是0

setTimeout(console.log(C.clientWidth),1000) //即便延迟获取也获取不到

## 指令

### 指令的作用

指令的职责是，当表达式的值改变时，将其产生的连带影响，**响应式地作用于 DOM。**

**将数据和DOM做关联！！！**

也就是说，必须将指令添加到一个元素上。

### 指令的参数

一些指令能够接收一个“参数”，在指令名称之后以冒号表示。

例如：v-bind、v-on

<a v-bind:href="url">...</a>

### 指令的修饰符

修饰符 (Modifiers) 是以半角句号 . 指明的特殊后缀，用于指出一个指令应该以特殊方式绑定。

例如，.prevent 修饰符告诉 v-on 指令对于触发的事件调用 event.preventDefault()

<form v-on:submit.prevent="onSubmit">...</form>

### 指令的缩写

### 1.v-bind

作用：绑定行间属性（内联样式）和class列表。

在将 v-bind 用于 class 和 style 时，Vue.js 做了专门的增强。表达式结果的类型除了字符串之外，**还可以是对象或数组**。

格式1：

v-bind:class="r"

格式2：

:class="r"

#### v-bind用于绑定一个或多个特性或一个组件

**v-bind:属性名=“表达式”**

动态地绑定一个或多个特性，或一个组件 prop 到表达式。

<!-- 绑定一个属性 -->  
<img v-bind:src="imageSrc">

v-bind动态渲染key值

如果想要有，能够**解析属性的对应的值**，就用v-bind

如果不需要解析属性，也就是**不是动态渲染**，就不用v-bind

<transfer-list :data="originData" title="源列表"></transfer-list>

        <script>

            let data = {

                originData: [ // 源列表

                    {

                        id:1000,

                        title: 'js课程',

                        checked: false

                    },

                    {

                        id:1101,

                        title: 'css课程',

                        checked: false

                    },

                ]

            }

            new Vue({

                el: '#app',

                data: {

                    ...data

                }

}）

#### v-bind用于绑定class外联样式表

##### 对象语法

**1.动态切换class:**

我们可以传给 v-bind:class 一个对象，以**动态地**切换 class：

例如：

<div class="static"

v-bind:class="{ **'active'**: isActive, **'text-danger'**: hasError }">

</div>

在上边的例子中，v-bind:class 指令也可以与普通的 class 属性共存。

当创建实例中的data为：

data: {

isActive: true,

hasError: false

}

结果渲染为：

|  |
| --- |
| <div class="static active"></div> |

当 isActive 或者 hasError 变化时，class 列表将相应地更新。例如，如果 hasError 的值为 true，class 列表将变为 "static active text-danger"。

|  |
| --- |
| <div class="static active text-danger"></div> |

**2.另一种绑定数据对象的方法是：数据对象不必内联定义在模板里。**

<div v-bind:class="classObject"></div>

data: {

classObject: {

active: true,

'text-danger': false

}

}

**3.绑定的不仅仅可以是数据对象，还可以是返回对象的计算属性**

<div v-bind:class="classObject"></div>

data: {

isActive: true,

error: null

},

computed: {

classObject: function () {

return {

active: this.isActive && !this.error,

'text-danger': this.error && this.error.type === 'fatal'

}

}

}

##### 数组语法

我们可以把一个数组传给 v-bind:class，以应用一个 class 列表：

|  |
| --- |
|  |

<div v-bind:class="[activeClass, errorClass]"></div>

data: {

activeClass: 'active',

errorClass: 'text-danger'

}

渲染结果为：

|  |
| --- |
| <div class="active text-danger"></div> |

如果你也想根据条件切换列表中的 class，可以用三元表达式：

|  |
| --- |
| <div  v-bind:class="[isActive ? activeClass : '', errorClass]"  ></div> |

在数组语法中也可以使用对象语法：

|  |
| --- |
| <div  v-bind:class="[{ active: isActive }, errorClass]"  ></div> |

##### 用在组件上

使用v-bind,在一个自定义组件上绑定使用 class 属性时，这些类将被添加到该组件的根元素上面。这个元素上已经存在的类不会被覆盖。

在使用组件时，可以不用v-bind为组件添加class:

例如：如果你声明了这个组件：

Vue.component('my-component', {

template: '<p class="foo bar">Hi</p>'

})

然后在使用它的时候添加一些 class：

<my-component class="baz boo"></my-component>

HTML 将被渲染为:

<p class="foo bar baz boo">Hi</p>

在使用声明的组件时，对于**带数据绑定** class 也同样适用：

<my-component v-bind:class="{ active: isActive }"></my-component>

当 isActive 为 truthy时，HTML 将被渲染成为：

<p class="foo bar active">Hi</p>

#### v-bind用于绑定style内联样式

##### 对象语法

直接绑定到一个样式对象，并把这个对象的值放在选项data里面。 **\*\*这里有绑定变量，就要加v-bind,没有变量就不加v-bind**

<div v-bind:style="styleObject"></div>

data: {

styleObject: {

color: 'red',

fontSize: '13px'

}

}

**\*\*这里有绑定变量，就要加v-bind,没有变量就不加v-bind**

<div

class="pic"

style='background-image:url(css/skin/ruixin/avatar.png);'

></div>

<div style="float:left;display: inline-block;">{{item.name}}</div>

中括号不能随便加，有变量才能加，否则就会报错。





##### 数组语法

<div v-bind:style="[baseStyles, overridingStyles]"></div>

### 2.v-for

**作用：**

指令根据一组数组的选项列表进行渲染。

v-for 指令需要使用 item in items 形式的特殊语法，

items 是源数据数组或者对象。**是数据中的属性名**。

 item 是元素迭代的别名。**是写在双大括号里面的属性名。**

例如：

<body>

<div id="app">

<li v-for="item in items">

{{ item.message }}

</li>

</div>

<script src="VUE.js"></script>

<script>

new Vue ({

el:'#app',

data:{

items:[

{ message:'Foo'},

{ message:'Bar'}

]

}

})

</script>

</body>

你也可以用 of 替代 in 作为分隔符

#### 对象的v-for

当遍历的是对象时，有三个参数。(value, key, index) in object

第一个参数value是数组/object的元素值，

第二个参数key是数组/object的key值，

第三个参数index是数组/object的索引。

<div v-for="(value, key, index) in object">  
 {{ index }}. {{ key }}: {{ value }}  
</div>

例如：

<body>

<div id="app">

<li v-for='v in object'>

{{ v }}

</li>

</div>

<script src="VUE.js"></script>

<script>

new Vue ({

el:'#app',

data:{

object: {

firstName: 'jone',

lastName: 'Doe',

age:30

}

}

})

</script>

</body>

#### 数组的v-for

当遍历的是数组的时候，有两个参数。(item,key) in items

第一个参数就是数组中的每个元素。

第二个参数就是数组中每个元素的下标。

例如：

<body>

<div id="app">

<li v-for="(item,key) in items">

{{ item }}-{{key}}

</li>

</div>

<script src="VUE.js"></script>

<script>

new Vue ({

el:'#app',

data:{

items:[

{ message:'Foo'},

{ message:'Bar'}

]

}

})

</script>

</body>

#### key

建议尽可能在使用 v-for 时提供 key，除非遍历输出的 DOM 内容非常简单.

所以你需要用 v-bind 来绑定动态值

<li v-for='value in object' **:key = 'value.id'**>

{{ value }}

</li>

#### 数组更新检测

##### 变异方法

Vue 包含一组观察数组的变异方法，所以它们也将会触发视图更新

* push()
* pop()
* shift()
* unshift()
* splice()
* sort()
* reverse()

例如：

<body>

<div id="app">

<li v-for="item in items">

{{ item.message }}

</li>

</div>

<script src="VUE.js"></script>

<script>

**let a = new Vue** ({

el:'#app',

data:{

items: [

{ message: 'Foo' },

{ message: 'Bar' }

]

}

})

**a.items.push({ message: 'Baz' })**

</script>

</body>

##### 非变异方法

变异方法：会改变被这些方法调用的原始数组。

非变异方法：这些不会改变原始数组，但**总是返回一个新数组**。

例如：

<body>

<div id="app">

<li v-for="item in items">

{{ item.message }}

</li>

</div>

<script src="VUE.js"></script>

<script>

let a = new Vue ({

el:'#app',

data:{

items: [

{ message: 'Foo' },

{ message: 'Bar' }

]

}

})

a.items = a.items.filter(function (item) {

return item.message.match(/Foo/)

})

</script>

</body>

有些方法不能触发更新

1. 当你利用索引直接设置一个项时，例如：vm.items[indexOfItem] = newValue
2. 当你修改数组的长度时，例如：vm.items.length = newLength

利用索引直接设置一个项解决办法是：

<body>

<div id="app">

<li v-for="item in items">

{{ item.message }}

</li>

</div>

<script src="VUE.js"></script>

<script>

let a = new Vue ({

el:'#app',

data:{

items: [

{ message: 'Foo' },

{ message: 'Bar' },

]

}

})

当你利用索引直接设置一个项时

1.**Vue.set(a.items, 1, { message: 'Bar2' })**

**2.a.items.splice(1, 1, { message: 'Bar2' })**

**3.a.$set(a.items, 1, { message: 'Bar2' })**

</script>

</body>

对于修改数组的长度时，解决办法是：

实例名.items.splice(newLength)

<body>

<div id="app">

<li v-for="item in items">

{{ item.message }}

</li>

</div>

<script src="VUE.js"></script>

<script>

let a = new Vue ({

el:'#app',

data:{

items: [

{ message: 'Foo' },

{ message: 'Bar' },

{ message: 'Bar' },

{ message: 'Bar' },

]

}

})

**a.items.splice(3) //这时就会把数组截成只有三个元素的数组！**

</script>

</body>

向对象添加响应式属性

当只添加一个响应式属性的时候：

Vue.set(object, key, value)

<body>

<div id="app">

<li v-for="(item,key) in userProfile">

{{ item }}-{{key}}

</li>

</div>

<script src="VUE.js"></script>

<script>

let a = new Vue ({

el:'#app',

data:{

userProfile: {

name: 'Anika'

}

}

})

1. **Vue.set(a.userProfile, 'age', 27)**

**2.** **a.$set(a.userProfile, 'age', 27)**

</script>

</body>

当要添加多个响应式属性的时候：

<body>

<div id="app">

<li v-for="(item,key) in userProfile">

{{ item }}-{{key}}

</li>

</div>

<script src="VUE.js"></script>

<script>

let a = new Vue ({

el:'#app',

data:{

userProfile: {

name: 'Anika'

}

}

})

a.userProfile = Object.assign({}, a.userProfile, {

age: 27,

favoriteColor: 'Vue Green'

})

</script>

</body>

### 3.v-text和v-html

v-html就相当于原生JS中的innerHTML。

v-text 就相当于原生JS中的innerText

### 4.v-if和v-show

**v-if**

根据表达式的值的真假条件渲染元素

如果条件成立就显示这个元素，否则不加载某个（些）元素

v-if

v-else-if 类似于 v-else，v-else-if 也必须紧跟在带 v-if 或者 v-else-if 的元素之后。

v-else v-else 元素必须紧跟在带 v-if 或者 v-else-if 的元素的后面，否则它将不会被识别。

**用key管理可复用的元素**

**优先级：**

当 v-if 与 v-for 一起使用时，v-for 具有比 v-if 更高的优先级。

**v-show**

另一个用于根据条件展示元素的选项是 v-show 指令。**用法和v-if大致一样**.

不同的是带有 v-show 的元素**始终会被渲染并保留在 DOM 中**。

也就是说v-show 只是简单地切换元素的 CSS 属性 display。

**[v-if vs v-show](https://cn.vuejs.org/v2/guide/conditional.html" \l "v-if-vs-v-show" \o "v-if vs v-show)**

v-if 是“真正”的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。

v-if 也是**惰性的**：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。

相比之下，v-show 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。

一般来说，v-if 有更高的切换开销，而 v-show 有更高的初始渲染开销。因此，如果需要非常频繁地切换，则使用 v-show 较好；如果在运行时条件很少改变，则使用 v-if 较好。

### 5.v-on事件处理、改变数据

##### 事件处理

绑定事件监听器。

**缩写**：@ v-on: === @

**参数**：event

**格式：**

**v-on:event = “函数名(实参)”**

v-on:事件名 = “表达式”

对应声明式渲染中的

methods:**{**

函数名：function(形参){

}

**}**

直接把 JavaScript 代码写在 v-on 指令中是不可行的。因此 v-on 还可以接收一个需要调用的方法名称。方法名后面的参数默认是（event）。

然后把这个事件的方法名写在methods里面。

<body>

<div id="example-2">

<!-- `greet` 是在下面定义的方法名 -->

<button v-on:click="greet">Greet</button>

</div>

<script src="VUE.js"></script>

<script>

var example2 = new Vue({

el: '#example-2',

data: {

name: 'Vue.js'

},

// 在 `methods` 对象中定义方法

methods: {

greet: function (event) {

// `this` 在方法里指向当前 Vue 实例

alert('Hello ' + this.name + '!')

// `event` 是原生 DOM 事件

if (event) {

alert(event.target.tagName)

}

}

}

})

</script>

</body>

**方法名后面的括号里面可以传参，如果没有参数则默认是(event)**

<body>

<div id="example-3">

<button v-on:**click="say('hi')"**>Say hi</button>

<button v-on:**click="say('what')"**>Say what</button>

</div>

<script src="VUE.js"></script>

<script>

new Vue({

el: '#example-3',

methods: {

**say: function (message)** {

alert(message)

}

}

})

</script>

</body>

如果有参数，使用$event获取事件

@click.prevent="selectSkin(value,key,$event)"

###### 事件修饰符

Vue.js 为 v-on 提供了**事件修饰符**。之前提过，修饰符是由点开头的指令后缀来表示的。

使用修饰符时，顺序很重要；相应的代码会以同样的顺序产生。因此，用 v-on:click.prevent.self 会阻止**所有的点击**，而 v-on:click.self.prevent 只会阻止对元素自身的点击。

###### 按键修饰符

Vue 允许为 v-on 在监听键盘事件时添加按键修饰符：

<input v-on:keyup.13="submit">

记住所有的 keyCode 比较困难，所以 Vue 为最常用的按键提供了别名：

* .enter
* .tab
* .delete (捕获“删除”和“退格”键)
* .esc
* .space
* .up
* .down
* .left
* .right

修饰键：

* .ctrl
* .alt
* .shift
* .meta

<!-- Alt + C -->

<input @keyup.alt.67="clear">

<!-- Ctrl + Click -->

<div @click.ctrl="doSomething">Do something</div>

与按键别名不同的是，修饰键和 keyup 事件一起用时，事件引发时必须按下正常的按键。换一种说法：如果要引发 keyup.ctrl，必须按下 ctrl 时释放其他的按键；单单释放 ctrl 不会引发事件。

<!-- 按下Alt + 释放C触发 -->

<input @keyup.alt.67="clear">

<!-- 按下Alt + 释放任意键触发 -->

<input @keyup.alt="other">  
  
<!-- 按下Ctrl + enter时触发 -->  
<input @keydown.ctrl.13="submit">

**[exact 修饰符](https://cn.vuejs.org/v2/guide/events.html" \l "exact-%E4%BF%AE%E9%A5%B0%E7%AC%A6" \o ".exact 修饰符)**

**2.5.0 新增**

.exact 修饰符允许你控制由精确的系统修饰符组合触发的事件。

|  |
| --- |
| **<!-- 即使 Alt 或 Shift 被一同按下时也会触发 -->** <button @click.ctrl="onClick">A</button>  **<!-- 有且只有 Ctrl 被按下的时候才触发 -->** <button @click.ctrl.exact="onCtrlClick">A</button>  <!-- 没有任何系统修饰符被按下的时候才触发 --> <button @click.exact="onClick">A</button> |

**.stop阻止冒泡**

<div id="box" @contextmenu="fn1">

<div id="box2" @click.stop="fn2">2</div>

</div>

**.prevent 阻止默认行为**

例如，避免图片被拖拽。

<img :src="src" @mousedown.prevent=""/>

例如，避免右键的弹出对话框。

<div id="box" @contextmenu="fn1">

methods:{

fn1:function(ev){

console.log(ev);

alert('我是1');

ev.preventDefault();

},

}

###### [将原生事件绑定到组件](https://cn.vuejs.org/v2/guide/components-custom-events.html" \l "%E5%B0%86%E5%8E%9F%E7%94%9F%E4%BA%8B%E4%BB%B6%E7%BB%91%E5%AE%9A%E5%88%B0%E7%BB%84%E4%BB%B6" \o "将原生事件绑定到组件)

你可能有很多次想要在一个组件的根元素上直接监听一个原生事件。这时，你可以使用 v-on 的 .native 修饰符：

|  |
| --- |
| <base-input v-on:focus.native="onFocus"></base-input> |

##### 数据更改

<li><a href="#/all" **@click="active='all'"** :class="active=='all'?'selected':''">全部</a></li>

click事件用来将active这个数据的值改变为“all”

### 6.表单输入绑定v-model

碰到表单元素就要立马想到用

你可以用 v-model 指令在表单 <input> 及 <textarea> 元素上**创建双向数据绑定**。它会根据控件类型自动选取正确的方法来更新元素。

#### 当表单元素为文本时

<div id="app">

<input

type="text"

v-model.trim="val"

/>

<input

type="text"

v-model.trim="val"

/>

<div>{{val}}</div>

</div>

new Vue({

el:'#app',

data:{

val:'123'

},

});

#### 当表单元素为复选框（checkbox）时

如果有多个表单控件，要通过用户的操作去取值，**直接在元素身上绑v-model和添加value即可**.

<div id="app">

<input

type="checkbox"

value="刘春雨"

v-model="arr"

/>

<input

type="checkbox"

value="杨丽春"

v-model="arr"

/>

<input

type="checkbox"

value="李宇春"

v-model="arr"

/>

<div>{{arr}}</div>

<p style="white-space: pre-line;">{{ message }}</p>

<textarea v-model='message'></textarea>

</div>

new Vue({

el:'#app',

data:{

**arr:[],**

**message:''**

}

});

v-model 会忽略所有表单元素的 value、checked、selected 特性的初始值而总是将 Vue 实例的数据作为数据来源。你应该通过 JavaScript 在组件的 data 选项中声明初始值。

任何要在双括号{{ }}里面显示的数据都应该有初始值，否则就无法显示。如果没有初始值，那就是空字符串或者空数组。总之就是要显示什么，都要在data里面声明！如上面的例子所述。

#### 当表单为单选按钮（radio）时

<body>

<div id="example-4">

<input type="radio" id="one" value="One" v-model="**picked2**">

<label for="one">One</label>

<br>

<input type="radio" id="two" value="Two" v-model="**picked2**">

<label for="two">Two</label>

<br>

<span>Picked: {{ **picked2** }}</span>

</div>

<script src="VUE.js"></script>

<script>

new Vue({

el: '#example-4',

data: {

**picked2**: '',

}

})

</script>

</body>

当表单为选择框时

<body>

<div id="example-4">

<select v-model="**selected2**">

<option **disabled** value="">请选择</option>

<option>A</option>

<option>B</option>

<option>C</option>

</select>

<span>Selected: {{ selected2 }}</span>

<select v-model="**selected1**" multiple style="width: 50px;">

<option>A</option>

<option>B</option>

<option>C</option>

</select>

<br>

<span>Selected: {{ **selected1** }}</span>

</div>

<script src="VUE.js"></script>

<script>

new Vue({

el: '#example-4',

data: {

selected2:'' , //当为单选框时，就用空字符串

selected1:[], //当为复选框时，就用空数组

}

})

</script>

</body>

注意：这里的复选框和单选框不能共存，这是为什么呢？？？？因为在代码上你少写了一个闭合的</div>

有时我们可能想把值绑定到 Vue 实例的一个动态属性上，这时可以用 v-bind 实现，并且这个属性的值可以不是字符串。

<body>

<div id="example-4">

<select v-model="selected">

<!-- 内联对象字面量 -->

<option v-bind:value="{ number: 1234hahaha }">123</option>

</select>

</div>

<script src="VUE.js"></script>

<script>

let vm = new Vue({

el: '#example-4',

data: {

selected:{},

number:'',

}

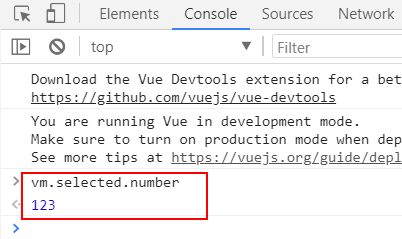
})

</script>

</body>

当没有选中的时候vm.selected.number就是undifined

当选中的时候，vm.selected.number就是1234hahaha



## 特殊特性

### ref

ref帮助我们快速定位某个\*\*组件\*\*或者某个\*\*元素\*\*

定义：

ref="名字" //定义的时候不带“s”

调用:

this.$refs.名字 //调用的时候带‘s’

例如：

<div id="app">

<p

ref="pp"

@click="click"

>pppppppppppp</p>

</div>

new Vue({

el:'#app',

methods:{

click(){

console.log(this.$refs.pp)

}

}

})

## 组件化

组件是可复用的 Vue 实例，且带有一个名字。

在 Vue 里，一个组件本质上是一个拥有预定义选项的一个 Vue 实例。

1.首先要把html结构写好。

2.

**完整的组件例子：**

<body>

<div id="components-demo">

<child-pa></child-pa>

</div>

<script src="VUE.js"></script>

<script>

Vue.component('child-pa', {

data: function () {

return {

count: 0

}

},

template: '<button v-on:click="count++">You clicked me {{ count }} times.</button>'

})

new Vue ({

el:'#components-demo'

})

</script>

</body>

### 1.注册组件

**格式一：**

Vue.component (组件的名字,{

template:`子组件的模板`

});

**格式二：**

const childPa = {

template:`<li>{{txt}}</li>`,

props:['txt']

}

Vue.component('child-pa',childPa);

**格式三：**

components.getFileMessage = function(resolve, reject) {

var options = {

name: 'groupfile', //name就是组件的名字

**注意:**

template顶层只能有一个元素。

#### 组件的命名（驼峰+烤串）

组件的命名可以是烤串命名法，也可以是驼峰命名法

#### 注册组件的选项

因为组件是可复用的 Vue 实例，所以它们与 new Vue 接收**相同的选项**，例如 data、computed、watch、methods 以及生命周期钩子等。**仅有的例外是像 el 这样根实例特有的选项。注册组件不需要el.**

##### [data 必须是一个函数](https://cn.vuejs.org/v2/guide/components.html" \l "data-%E5%BF%85%E9%A1%BB%E6%98%AF%E4%B8%80%E4%B8%AA%E5%87%BD%E6%95%B0" \o "data 必须是一个函数)

**一个组件的**data**选项必须是一个函数**，因此每个实例可以维护一份被返回对象的独立的拷贝（也就是说当引用组件的时候，每一次引用的组件都会创建自己的根实例，并进行计数count）：

data: function () {

return {

count: 0

}

},

#### 全局注册和局部注册

全局注册就是注册在new Vue外面，局部注册就是注册在new Vue里面。

全局注册：

  Vue.component("custom-select",{option})

局部注册：

            new Vue({

                el:"#app",

                components:{

                }

            })

如果使用局部注册，就可以**出现多个new Vue**

### 2.进行声明式渲染

// 创建根实例

<div id="example">

</div>

<script>

new Vue({

el: '#example'

})

</script>

### 3.引用组件

// 创建根实例

<div id="example">

**<child-pa></child-pa>**

**<child-pa></child-pa>**

**<child-pa></child-pa>**

</div>

<script>

new Vue({

el: '#example'

})

</script>

#### 引用组件的命名（烤串）

注意：**引用组件的时候，一定要用烤串命名法**

#### 组件的复用

你可以将组件进行任意次数的复用：

注意当点击按钮时，每个组件都会各自独立维护它的 count。因为你每用一次组件，就会有一个它的新**实例**被创建。

每个组件都是独立的，如果他们公用一个对象，在更改一个组件数据的时候，会影响其他组件，如果是函数的话，每个组件都有自己独立的数据，互相之间不会影响。

#### is扩展受限制元素

Vue是在浏览器解析和标准化HTML之后，才能获取模板的内容，所以有些元素限制了能被他包裹的元素。

例如，ul中只能放li;select中只能放option.某些元素放入了自定义元素，不符合W3C标准，最终会解析错误。

使用is对原生的HTML进行扩展,后面跟自定义组件。

例如：

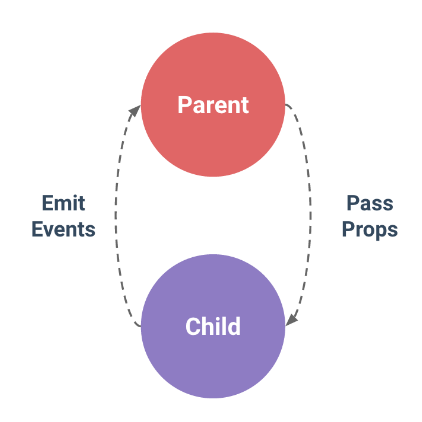
            <table>

                <tr is='custom-select'></tr>

            </table>

### 组件的prop

在 Vue 中，父子组件的关系可以总结为 **prop 向下传递，事件向上传递**。父组件通过 **prop**给子组件下发数据，子组件通过**事件**给父组件发送消息。



#### 1.父组件向子组件传递数据

早些时候，我们提到了创建一个博文组件的事情。问题是如果你不能向这个组件传递某一篇博文的标题或内容之类的我们想展示的数据的话，它是没有办法使用的。这也正是 prop 的由来。

组件的作用域是孤立的，那么组件是如何获取参数的呢？就是用prop特性。

当一个值传递给一个 prop 特性的时候，它就变成了那个组件实例的一个属性。

        <div id="app"> **//这个就是父组件**

            <h2>自定义的下拉框</h2> **//这是子组件**

            <custom-select></custom-select> //这是子组件

            <h2>自定义的下拉框2</h2> //这是子组件

            <custom-select></custom-select> //这是子组件

        </div>

如何将父组件的数据传递给子组件呢？

父组件向子组件传递数据举例如下：

**步骤一.**创建根实例，并将数据的源头放在data中。这个创建的根实例就是父组件。

**步骤二**.将数据传递给组件

在组件中需要显式的用props**显式声明自定义属性名**（作为形参），（这里的属性名**只可以用驼峰命名法**，在template模板中**也要用驼峰命名法**）。、

**这里所谓的显式声明自定义属性名，就是说可以自定义这个key值！**

第一步：用props接受

第二步：用v-bind在template的模板中去使用

例如：子组件要显式地用 [props 选项](https://cn.vuejs.org/v2/api/" \l "props)声明它预期的数据：

Vue.component('child', {

// 声明 props

props: ['message'], //这里注意格式是[ “ ” ]（中括号+里面的引号）

template: '<span>{{ message }}</span>'

})

然后我们可以这样向它传入一个普通字符串：

<child message="hello!"></child>

**步骤三**.自定义标签传递数据或者作为数据的源头。这个自定义标签就是子组件。

可以在自定义标签上使用自定义属性绑定数据，属性名**只可以用烤串命名法**，属性值可以绑定变量和常量。

当自定义标签用变量作为数据的传递者时，就是名副其实的父组件向子组件传递数据。也可以理解为当作为形参时，说明这个不是数据的源头，数据的源头在创建的根实例中

例如：

        <div id="app">

            <div style="float: left;">

                <h2>自定义的下拉框</h2>

                <custom-select

**btn="查询"** //这个绑定是是常量

**v-bind:list="list1"** //这个绑定是变量

></custom-select>

            </div>

            <div style="float:left;">

                <h2>自定义的下拉框2</h2>

                <custom-select **btn="搜索"** v-bind:list="list2"></custom-select>

            </div>

        </div>

当自定义标签用常量作为数据的源头时，就不是名副其实的父组件向子组件传递数据，是子组件自己传递数据。也可以理解为当作为实参时，说明这个就是数据的源头。

例如：

<body>

<div id="components-demo">

<child-pa tittle=' my jorney with 0Vue'></child-pa>

<child-pa tittle=' my jorney with 1Vue'></child-pa>

<child-pa tittle=' my jorney with 2Vue'></child-pa>

<child-pa tittle=' my jorney with 3Vue'></child-pa>

</div>

<script src="VUE.js"></script>

<script>

Vue.component('child-pa', {

props:['tittle'],

template: '<h3> {{ tittle }}</h3>'

})

new Vue ({

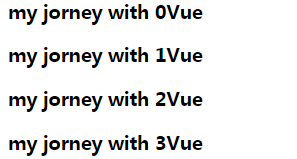
el:'#components-demo',

})

</script>

</body>

此时，屏幕打印以下信息，说明一个child-pa组件接受了多个参数。而这些个实参是组件的自定义属性。



##### 1.父级要有数据data

##### 2.组件接受数据并表现在模板中

组件要有props这个选项**接受数据。**

props以数组的形式接受key值。

**这里就是显式的声明key值！也就是显式的声明自定义属性名**

**这里就是使用props显式声明的自定义属性名！**

<app v-bind:key1='val1'></app>

Vue.component("app",{

props:['key1'],

template:`

<div>

<p>你好,{{key1}}</p>

</div>

`

});

new Vue({

el:'#app',

data:{

val1:"还是",

},

})

##### 3.将数据作为自定义属性绑定到子组件上

**在父级指代子组件的元素上用v-bind接收这个传递的数据，**

**其中key值是指子组件的props指定的名称**

**value值是指父级要传递的数据名称**

v-bind:属性名（key值）=‘根实例的数据key值（也可以理解为根实例的数据表达式）’

<app v-bind:key1='val1'></app>

#### 子组件向孙组件传递数据

##### 1.子级要有数据，

**也就是props显式声明自定义属性名**

##### 2.组件接受数据并表现在模板中

1.在孙级的component中使用props显式的声明自定义属性名

2.在template中使用上面一句话中显式声明的自定义属性名。

##### 3.将子级的数据渲染到孙级DOM

v-bind:属性名（这个就是孙级props显式声明的自定义属性名）=‘子级props显式声明的自定义属性名’

                        <custom-list

                            :listnum2="listnum1"

                        ></custom-list>

#### 子组件在调用的时候传值

1. 组件接受数据并表现在模板中

组件要有props这个选项**接受数据。**

props以数组的形式接受key值。

组件要有template这个选项**表现数据**。

template里面以双括号的形式表现数据

<app v-bind:key1='888'></app>

Vue.component("app",{

props:['key1'],

template:`

<div>

<p>你好,{{key1}}</p>

</div>

`

});

new Vue({

el:'#app',

})

2. 将数据渲染到DOM

v-bind:属性名（key值）=‘子组件要传入的值’

<app v-bind:key1='888'></app>

Vue.component("app",{

props:['key1'],

template:`

<div>

<p>你好,{{key1}}</p>

</div>

`

});

new Vue({

el:'#app',

data:{

val1:"还是",

},

})

##### 单向数据流

在子组件中不能直接改变父组件传来的数据：指的是在methods中直接调用父级出来的值。

**解决办法：用data转换后来改变父级传来的值**

在子组件中可以间接改变父组件传来的值，方法是将父组件传来的值在子组件的data中声明一个属于子组件的变量，然后把 父级传来的值赋值给这个子组件的变量。这时，再用methods方法改变值。

   data(){

       return {

        incrementCount:this.count //作为局部这个组件的data的初始值

               }

         },

   template:`

       <div>

            <h2>我是一个自定义的组件</h2>

            <input type="button" value="改变count的值" @click="changeCount" />

                 {{ incrementCount }}

       </div>

                `,

   methods:{

             changeCount(){

                  this.incrementCount++;

                    }

                }

#### 子组件向父组件传数据

**子级和父级的传递核心在于$emit(‘传递事件名’,要传递的实参)**

**步骤一**

在子组件上绑定一个传统事件：

     template:`<ul class="list">

              <li

                     v-for="item of list"

**@click="selectValueHandle(item)"**

                  >{{item}}</li>

                </ul>`,

在子组件的methods / **watch**实现方法，

将**事件**以及**传递的参数**传给监听器回调。

methods:通过方法动作事件触发emit

                methods:{

                    selectValueHandle:function(item){

                   //告知父级，改变val的值，需要触发一个自定义事件

                        this.$emit("**receive**",item);

                    }

                }

**watch:** 通过监听函数触发emit

watch: {

value: function() {

var context = this;

context.$emit('editBoxChange', context.getValue().text);

**$emit (event,[...args])**

参数：

{string} event

[...args]

触发当前实例上的事件。附加参数都会传给监听器回调。

**步骤二**

**在父级指代子组件的元素上**用v-on(‘**@**’)绑定$ emit传入的事件名。

**v-on后面跟的事件名称（receive）,来接受子级的$emit传来的事件和参数。**

**$emit后面等于的是 父组件方法的名称**

**这里只需要有‘父组件方法的名称’，就可以携带子组件传过来的参数。**

**展示参数在父组件的methods中的function中声明，即可在方法中获取**

**例如1：**

<li v-for="file in fileList">

<component

is='group-fileitem'

:file="file"

ref="fileItem"

@delfiledata="refresh"

>

</component>

</li>

**例如2：**

       <custom-list

           v-show="selectShow"

           :list="list"

**v-on:receive="changeValueHandle"** //这里不用携带参数

**也就是 @receive="changeValueHandle"**

        ></custom-list>

再在父级的methods中实现元素绑定的事件方法，并将接受到的参数存到data的声明的变量名中。用来渲染页面。

例1：

   methods:{

**changeValueHandle(value)**{ //**这里声明参数，即可接受到子级传过来的参数**

**this.val = value;**

         }

   }

例2：

upDown:function(bollo){ //**这里声明参数，即可接受到子级传过来的参数**

console.log(bollo)

}

综上所述，举例如下：

<body>

<div id="app">

<div>父级{{num}}</div>

<hr />

<child-pa **@changen="changeNum" 3**></child-pa>

</div>

<script src="vue.js"></script>

<script>

Vue.component('child-pa',{

template:`

<div>

<button

**@click="click" 1**

>子级点击修改</button>

<p>{{cnum}}</p>

</div>

`,

data(){

return {

cnum:0

}

},

methods:{

click(){

this.cnum ++;

**this.$emit('changen',this.cnum); 2**

}

}

});

new Vue({

el:'#app',

data:{

num:0

},

methods:{

**changeNum(sonnum){ 4**

this.num = sonnum; //这里就是第4步中数据的渲染了

}

}

});

</script>

</body>

#### $emit方法

**$emit (event,[...args])**

参数：

{string} event

[...args]

**触发当前实例上的事件**。附加参数都会传给监听器回调。

#### $on方法

我们也可以在某些特定情况或方法内采用**$on 方法来监听事件**，例如：

var vm = new Vue({

　 el : '#app',

　 data : {

　　 todo : []

　 },

　 methods : {

　　begin : function() {

　　　 this.$on('add', function(msg) {

　　　　　this.todo.push(msg);

　　　 });

　　}

　 }

});

#### $off 方法

取消监听

#### props验证

组件可以为props指定验证要求如果未指定验证要求，VUE会发出警告。

例如：

    props:{

        count**:{ //注意这个验证要用{}包起来**

           //type:Number, //**验证传过来的值是否是Number**

           //type:[Number,String], //**验证传过来的值是否是Number,或者是字符串。**

           //default:10 //**如果不传值过来，默认的值为10**

           //required:true //**如果不传值，就会在控制台里面发出警告**

**validator**:function (value){ // **自定义一套验证规则**

                   console.log(value);

**return** value > 10

                        }

**}**

       },

## 混入 (mixins)

混入 (mixins) 是一种分发 Vue 组件中可复用功能的非常灵活的方式。混入对象可以包含任意组件选项。当组件使用混入对象时，所有混入对象的选项将被混入该组件本身的选项。

## slot

### 使用slot分发内容

使用一种方式混合父组件的内容与子组件自己的模板。这个过程被称为‘内容分发’，在子组件中使用特殊的<slot>元素作为内容的插槽。

这样会使组件的可扩展性更强。

在自定义标签里面写入html内容，

并在temlate选项中插入<slot> html内容 </slot>标签。

自定义标签的html内容渲染优先级  **>**  slot标签的渲染优先级

    <body>

        <div id="app">

            <h2>自定义组件</h2>

            <custom>

             <!-- 当写在自定义标签之间的内容 要混合子组件中的模板 -->

                <div>我是html结构</div> **//优先级高**

            </custom>

        </div>

        <script>

            //定制模板 div

            Vue.component("custom",{

                template:`

                    <div>

                        <slot> **//这里面的优先级低**

                            <p>这是第1个提醒</p>

                            <p>这是第2个提醒</p>

                            <p>这是第3个提醒</p>

                        </slot>

                    </div>

                `

            })

            new Vue({

                el:"#app"

            });

        </script>

    </body

### 指定替换

自定义标签中加上 slot = “slot.name” ,

就可以指定地替换选项template中的<slot name = ‘’slot.name’’></slot>标签。

            <custom>

                <div slot="one">替换第1给提醒</div>

                <div slot="three">替换第3给提醒</div>

            </custom>

**替换**

           template:**`**

               <div>

                  <slot name="one"><p>这是第1个提醒</p></slot>

                  <slot name="three"><p>这是第3个提醒</p></slot>

                </div>

**`**

自定标签中 无名的

指定的替换选项template中无名的slot标签

            <custom>

               <div>替换无名的slot</div>

            </custom>

**替换**

           template:**`**

               <div>

                 <slot>无名的slot</slot>

               </div>

**`**

切换至英文状态  然后按左下角这个键

[https://gss0.baidu.com/94o3dSag_xI4khGko9WTAnF6hhy/zhidao/wh%3D600%2C800/sign=ef767b31462309f7e73aa514423e20cb/aec379310a55b3194bf8ecd043a98226cffc1715.jpg](https://gss0.baidu.com/94o3dSag_xI4khGko9WTAnF6hhy/zhidao/pic/item/aec379310a55b3194bf8ecd043a98226cffc1715.jpg)

### <template></template>标签

      <template slot="two">

                    <div>替换第1给提醒</div>

                    <div>替换第2给提醒</div>

                    <div>替换第3给提醒</div>

                    <div>替换第4给提醒</div>

                    <div>替换第5给提醒</div>

                    <div>替换第6给提醒</div>

        </template>

**等价于==**

                <div slot='two'>替换第1给提醒</div>

                <div slot='two'>替换第2给提醒</div>

                <div slot='two'>替换第3给提醒</div>

                <div slot='two'>替换第4给提醒</div>

                <div slot='two'>替换第5给提醒</div>

                <div slot='two'>替换第6给提醒</div>

### 编译作用域

父组件模板的内容在父组件作用域内编译；

子组件模板的内容在子组件作用域内编译

    <body>

        <div id="app"> //这里是父组件作用域

            <h2>自定义组件</h2>

            <custom>

                {{message}}

            </custom>

        </div>

        <script>

            Vue.component("custom",{

**data(){ //这里是子组件数据**

**return {**

**message:"我是子组件的数据"**

**}**

**},**

**template:`**

**<div> //这里是子组件作用域**

**{{message}}**

**<p>我是第1个提醒</p>**

**</div>**

**`**

            })

**new Vue({**

**el:"#app",**

**data:{ //这里是父组件数据**

**message:"我是父组件的数据"**

**},**

**});**

        </script>

    </body>

## 动态组件

多个组件可以使用同一个挂载点，动态地在他们之间切换。

使用保留的<component>元素，使用is特性进行动态绑定。

<component :is="current"></component>

可以使用**keep-alive**把切出去的组件保留在内存中，这样可以保留它的状态，避免重新渲染。

<keep-alive>

<component :is="current"></component>

</keep-alive>

## vue-cli脚手架

**如何安装脚手架**

1.首先要安装webpack. npm install webpack –g 的方式全局安装webpack.

2.然后要保证node的版本是最新的，只要不是太低就好。

3. npm install –g vue-cli 进行全局安装。

之后就可以使用命令 vue init <template-name> <project-name> 进行脚手架的安装。

**<template-name>**：模板名，比如 webpack, simple, browserify

**<project-name>**： 本地文件夹名称

**? Project name**  输入项目名称

**? Project description** 输入项目描述

**? Author** 作者

**? Vue build** 打包方式，回车就好了

**? Install vue-router?**  选择  Y 使用 vue-router，输入 N 不使用

**? Use ESLint to lint your code?** 代码规范，推荐 N

**? Setup unit tests with Karma + Mocha?** 单元测试，推荐 N

**? Setup e2e tests with Nightwatch?** E2E测试，N

## vue-router

Vue-router 是给Vue.js 提供路由管理的插件，利用hash 的变化控制动态组件的切换。

**好处：**

通过一次性加载所有的js和html资源，不需要再次请求加载。尤其用于移动端。

### $route

用来配置路由信息。

放在组件的事例中。

methods:{

getData(){

let id = **this.$route**.params.userId;

}

}

### router-link

两种形式，一种是匹配对象的，这个to前面就要加：

另外一种形式是匹配字符串的，这个to前面就不用加什么。

<router-link exact :to="{path:'',query:{info:'follow',a:1}}">我的关注 </router-link>

<router-link exact to="?info=share">我的分享 </router-link>

重定向

{

path: '\*',

// component: nofound

// redirect:'/home'

// redirect:{path:'/home'}

// redirect:{name:'About'}

redirect: (to) => {

if (to.path === '/123') {

return '/home'

} else if (to.path === '/456') {

return {path: '/document'}

} else {

return {name: 'About'}

}

// console.log(to)

// return '/home'

}

}

### query对象

query对象，URL查询参数

### 过渡效果css

1.首先用transition双标签将router-view双标签包住。

<transition mode="out-in">

<router-view class="center"></router-view>

</transition>

2.使用CSS控制过渡效果。

格式：

.name - enter/leave(进入/离开) -

v-enter: 定义进入过渡的开始状态

v-enter-active: 定义进入活动状态

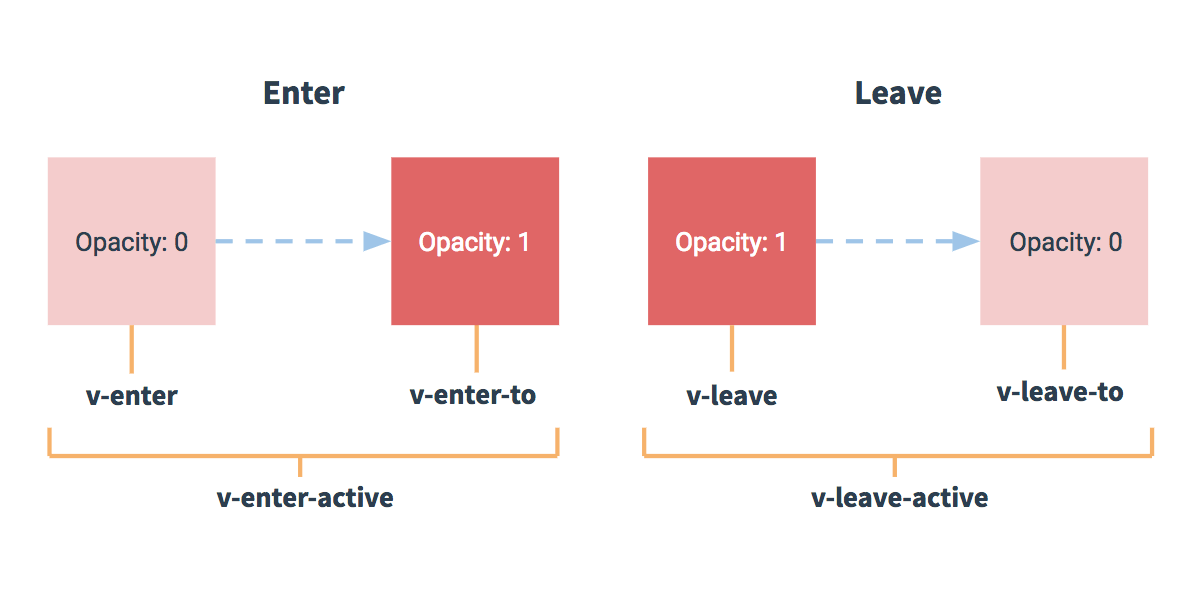
v-enter-to: 定义进入的结束状态

v-leave: 定义离开过渡的开始状态

v-leave-active: 定义离开活动状态

v-leave-to: 定义离开的结束状态

使用name属性改变类名前缀



<style>

.v-enter { //不带to的就是开始的效果

opacity: 0;

}

.v-enter-to { //带to的就是结束的效果

opacity: 1;

}

.v-enter-active { //带active的就是过渡需要的时间

transition: 1s;

}

.v-leave {

opacity: 1;

}

.v-leave-to {

opacity: 0;

}

.v-leave-active {

transition: 3s

}

</style>

### 过渡模式

in-out: 新元素先进行过渡，完成之后当前元素过渡离开

out-in: 当前元素先进行过渡，完成之后新元素过渡进入

在路由配置中meta可以配置一些数据，用在路由信息对象中

访问meta中数据：$route.meta

### 编程式导航

借助于 router 的实例方法，通过编写代码来实现导航的切换

* + back() 方法 回退一步
  + forward() 方法 前进一步
  + go() 方法 指定前进回退步数（括号内接收整数的实参。往后退N步就写-N,

往前前进N步就写N.

不能超过历史记录的记录最大值，否则无效。

实参为0则刷新页面。

）

* + push 导航到不同url，并向 history 栈**添加**一个新的记录

this.$router.push('/document')

this.$router.push({path:'/document'})

* + replace 导航到不同url，并**替换 history 栈中当前记录**

### 导航钩子函数

导航发生变化时，导航钩子用来拦截导航，让他完成跳转或者取消。

**执行钩子函数位置**

router全局

单个路由

组件中

#### router实例上的钩子函数

router实例上：**beforeEach**、**afterEach**

beforeEach是一个函数

router.beforeEach((to,from,next) =>{

console.log('beforeEach');

next();

})

接受三个参数，to就是目标导航的路由信息对象，from离开的导航的路由信息对象。

next是决定是否让其进入导航，是否被执行

next接收false作为参数的时候，不进入导航

**beforeEach**、**afterEach**是全局的钩子函数，只要触发路由就会触发这两个函数

#### 单个路由中的钩子函数

单个路由中：**beforeEnter**

beforeEnter是某一个具体路由导航的钩子函数，只有在触发这个路由的时候才会触发这个钩子函数。

{path: '/document',

name: 'Document',

beforeEnter (to, from, next) {

console.log('beforeEnter')

next()

},

meta: {

index: 2,

login: true,

tittle: 'document'

},

components: {

default: document,

slider: slider

}

}

#### 组件内的钩子

组件内的钩子：**beforeRouteEnter、beforeRouteUpdate 、beforeRouteLeave**

**如何使用：**

</div>

</template>

<script>

export default {

beforeRouteEnter(){

console.log('beforeRouterEnter')

}

}

</script>

**先执行路由的钩子函数**

**后执行组件生命周期函数**

**所以：**

**beforeRouteEnter (to,from,next ){}** 这个钩子函数在运行的时候，实例还没有被创建出来。所以这个函数里面的this是undifined

**如何在路由钩子函数中调用实例？**

**在next**

beforeRouteEnter(to,from,next){

next((vm2) =>{

vm2.test = '改变了'

});

}

**beforeRouteUpdate**

在进入了一级导航之后，二级导航的route要更新之前，执行这个钩子函数。它同样接受三个参数（to,from,next）.

**beforeRouteLeave**

当要离开一个导航时，有些数据还没有渲染完成，那么就要阻止这个导航变成另外一个。这时，就可以用**beforeRouteLeave**这个钩子函数。

它同样接受三个参数（to,from,next）。当不执行next()时，就不会往下执行切换路由。

#### 钩子函数接收的参数

to: **要进入的**目标 路由对象 到哪里去

from: 正要离开导航的路由对象 从哪里来

next: 用来决定跳转或取消导航

## vuex

Vuex是一个专为Vue.js应用程序开发的状态管理模式

## 字符串模板

模板-render函数

const **routes** = { //是routes不是Routers!!!!

}

Ready Detected Vue工具装到浏览器中

# Jquery

可以说，jquery就是对DOM和BOM方法的扩展版！！！

用法和DOM,BOM差不多。

# 移动端

# Question

string.lastIndexOf('字符')

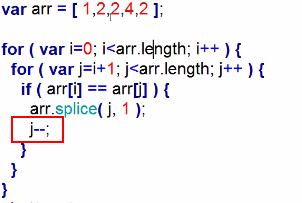
for( ; str.indexOf( s, i ) != -1 ; ){ //为什么没有++？？？？

alert( str.indexOf( s, i ) );

i = str.indexOf( s, i ) + s.length;

}

字符串比较只比较第一个字符串

???j- -是怎么回事？？？

var arr = [ 1,2,3,4,5,6,7,8 ];

arr.sort(function ( a, b ) { //怎么回事？？？？ pa

return Math.random() - 0.5;

});

alert( arr );

# Q1

first.onclick = function () { //这里红色的为什么不能封装，this为什么封装了之后就不起作用了？？？

if (onOff) {

this.nextElementSibling.className ='hide';

onOff = false;

}else {

this.nextElementSibling.className = "show";

onOff = true;

}

}

Q2：

JSON , object{} 数组[] 对象 这几个什么关系？？

‘{name:1}’

[],{name:1}

let obj = {a:1,b:2}

console.log(obj.b)

let obj = ‘{name:1}’;

数据格式的对象 =》 {}

数据类型的对象 =》 {}、[]

Q3：

数组如何去重？

Q4:

npm i 是什么意思装环境是给谁装？不是有一个create-react-app my-app了吗？为什么还要装环境？

Q5:

VScode如何快速切换页面和编辑器？？

Q5：

render(){

let {arr,num} = this.state;

let list = arr.map((e,i)=>{

let obj = {

key:i,

txt:e.txt,

checked:e.checked,

id:e.id,

cc:this.changeChecked

}

return <List {...obj}/>

//txt={e.txt} checked={e.checked} key={i}

})

为什么要有子组件？

constructor???

# image相关语法

## new Image()

## image.onload

图片加载成功

## image.onerror

图片加载失败

清缓存

