1. **正则表达式**

**Regular Expression**

概念: 规定一个字符串中内容出现规

律的规则

作用2种:

a. 格式验证；

b. 按规则查找/替换关键词；

使用:

|  |
| --- |
| 1. 最简单的正则表达式，是一个关键词的原文 |
| 2．字符集: 规定一位字符上多种备选字列表的集合，只要一位字符上有多种备选字时  **步骤：** [备选字列表]  强调: 一个字符集只能匹配一位字符  简写: 如果备选字符是连续的，可用-省略中间字符  比如: 1位数字: [0-9]  1位小写字母: [a-z]  1位大写字母: [A-Z]  1位字母: [A-Za-z]  1位字母或数字: [A-Za-z0-9]  1位汉字: [\u4e00-\u9fa5]  排除: [^47] 除了4和7之外都行  强调: 范围太大！ |

练习：

|  |
| --- |
| 车牌号: 1位汉字 1位大写字母 中点 五位大写字母或数字  [\u4e00-\u9fa5][A-Z]•[A-Z0-9] [A-Z0-9] [A-Z0-9] [A-Z0-9] [A-Z0-9] |

|  |
| --- |
| 3．预定义字符集: 4种:  ①1位数字: \d  等效于[1-9]  ②1位字母数字或下划线: \w  等效于[A-Z a-z 0-9 \_ ]  ③1位空字符: \s  包括: 空格tab…  ④匹配所有文字（通配符）：.  问题: 单靠字符集，无法灵活定义字符的个数  解决: 用量词 |
| 4.量词：规定一位字符出现次数的规则，只要定义一位字符出现次数，放在字符集之后，修饰相邻的前一个字符集  包含: 2大类:  ①有明确数量边界: 3种:  {n,m} 最少n个，最多m个  {n,} 最少n个，多了不限！  {n}必须n个，不能多也不能少  ②没有明确数量边界: 3种:  ? 可有可无，最多一个  \* 可有可无，多了不限  + 至少一个，多了不限  问题: 量词默认仅修饰相邻的前一个字符集 解决: 分组 |
| 5.分组：用( )将多个字符集包裹为一组，要希望将多个子规则视为一个整体，再和其它规则匹配时，就用分组  练习：  身份证号: 15位数字 2位数字 1位数字或Xx，后三位整体可有可无，最多1次  \d{15}(\d{2}[0-9Xx])? |
| 6.选择: 或  定义: 在多个条件中任选其一匹配，只要在多个条件中任选其一匹配即可  规则1|规则2  练习：定义完整手机号规则  ((\+86|0086)\s+)?1[3-8]\d{9} |
| 7. 指定匹配位置: 3个:  ①字符串开头 ^  ②字符串结尾 $  仅匹配开头或结尾的关键词时  仅匹配开头的空字符: ^\s+  仅匹配结尾的空字符: \s+$  同时匹配开头或结尾空字符:^\s+|\s+$  ③单词边界 \b  凡是能分割两个单词的字符，包括:  开头, 结尾，空格，标点…  例: 匹配每个单词首字母: \b[a-z] |
| 8.匹配相同内容: 2步  ①给第一个关键词加()  ②\1 代表和()中相同的内容  比如: 匹配连续的多个相同字母/字符 ([a-z])\1\* |

**二. String 的正则API:**

查找/替换/切割

**查找关键词: 4种:**

|  |  |
| --- | --- |
| 1. 查找一个固定的关键词出现的位置:  var i=str.indexOf("关键词",starti)  从str中starti位置开始，找下一个"关键词"的位置i  如果找到，返回关键词第一个字所在的位置i（下标）  如果找不到，返回-1  问题: 不支持正则，只能查找一个固定的关键词（但是效率高）   |  | | --- | | var smg=prompt("输入内容");  var i=smg.indexOf("卧槽"); if(i!=1){document.write("no")} else{document.write("YES")}; | |
| 2. 模糊查找一个关键词的位置:  var i=str.search(/正则/)  在str中从0位置开始找下一个符合"正则"要求的敏感词的位置i  如果找到，返回关键词第一个字的位置  如果没找到，返回-1  问题: 正则表达式默认区分大小写  解决: 在第二个/后加后缀i,表示ignore忽略大小写  更高要求: 既返回关键词内容，又返回关键词位置 |
| 3. 模糊查找关键词，并返回关键词内容: 两种:  ①只返回一个关键词的内容:  var arr=str.match(/正则/i)  在str中查找符合/正则/要求的关键词，并同时返回关键词的内容和位置  返回值: arr[ "0": 关键词, "index": i ]  数组本质: 所有数组底层其实都是关联数组  获取关键词内容: arr[0] arr.0  获取关键词位置: arr["index"] <==> arr.index  如果找不到，返回null  正则表达式第2个问题: 只找一个就结束了！  解决: 在第2个/后加g=>global 全部  ②查找所有关键词的内容(不关心位置):  var arr=str.match(/正则/g)  查找str中所有符合/正则/要求的关键词保存到arr中  返回值: 所有关键词组成的数组  找不到: 返回null  **强调**：如果一个函数可能返回null，都要先验证不是null  `替换了${names!=null?names.length:0}处`   |  | | --- | | var str=" 我是小红 小兰 小绿"  var i=str.match(/小[\u4e00-\u9fa5]/ig);  console.log(`共替换了${i!=null ? i.length :0}`); | |
| 4. 既查找每个关键词的内容，又查找每个关键词的位置:  match()不能实现:  match()不加g时，只能找1个关键词  match()加g后，不能获得位置  解决: RegExp对象的exec()方法 |

**替换关键词: 2种**

|  |
| --- |
| 1. 简单替换: 所有关键词都替换为统一的新值  str=str.replace(/正则/ig,"新值")  回顾: 所有字符串API都无权修改原字符串，只能返回新字符串！所以必须用变量才能借助修改后的新值。  将str中所有符合/正则/要求的关键词都替换为统一的"新值" |

|  |  |
| --- | --- |
| 2. 高级替换: 根据找到的每个关键词的不同，动态返回对应的新值替换。str=str.replace(/正则/ig, function(kword){  return 新值;  })   |  | | --- | | var str=" xu ting ";  function f1(str){return str.replace(/^\s+|\s+$/g,"");}  console.log(f1(str)); |   原理: a. replace查找str中符合条件的  每个关键词  b. 每找到一个关键词，就自动调  用回调函数  c. 每次调用函数都自动传入本次  找到的关键词  d. 回调函数根据本次传入的关键  词，加工后生成新替换值返回  e. replace将本地回调函数返回的  值，替换到正在遍历的位置。 |
| 衍生操作：删除关键字,替换为空字符串 |

**切割:**

|  |  |
| --- | --- |
| 将一个字符串，按照切割符，切割成多段子字符串  1.简单切割: 切割符是固定的  比如: [zhangdong@tedu.cn](mailto:zhangdong@tedu.cn)  no zuo no die  var arr=str.split(“固定的切割符”)  返回多段子字符串组成的数组  固定套路: 打散一个字符串为字符数组  var chars=str.split(“”)  2.复杂切割: 切割符不是固定的  var arr=str.split(/正则/);   |  | | --- | | var kword="lala sjsjsjs kkkk";  kword=kword.split(/\s+/ig);  console.log(kword); | |
| 作业:  1.编写正则，检查"微信"...  (wei|[w微])\s\*(xin|[信x])（有问题）  (微|w(ei)?)\s\*(信|x(in))  2.编写函数去开头和结尾的空字符 |

**三．RegExp对象:**

**(regular expression)**

|  |
| --- |
| **1.定义**：专门保存一条正则表达式，并提供用正则表达式执行验证和查找功能的API 的对象  **使用：** 只要在程序中保存一条正则表达式或使用正则表达式执行验证和查找时 |
| **2.创建: 2种:**  ①用//: **var reg=/正则/ig**  问题: 正则是写死的，无法再运行时动态生成正则表达式  ②用new: **var reg=new RegExp(“正则”,”ig”)**  好处: 参数是一个字符串格式的正则表达式，所以可以用任何程序的手段先动态生成字符串，再传给new RegExp变为正则表达式对象——动态生成正则 |

**二．API**

|  |
| --- |
| ①验证: 验证一个字符串的格式是否符合正则表达式的要求。  **var bool=reg.test(str)**  验证str是否符合reg的格式要求  问题: test其实是只要找到符合规则的，就返回true，不要求从头到尾完整匹配  如: /\d{6}/.test(“123456a”) -> true  解决: 今后只要验证，必须前加^，后加$，表示必须从头到尾完全匹配才行  如: /^\d{6}$/.test(“123456a”) -> false  其实: reg.test()也可判断一句话中是否包含敏感词，且更简单。  如: if(reg.test(msg)) 如果msg中包含符合reg的敏感词 |
| ②查找: 既查找每个关键词的内容，又查找每个关键词的位置  **var arr=reg.exec(str)**  让reg去str中找下一个符合规则的关键词的内容和位置  返回值: 和str.match()不加g的情况是一样的  arr[“0”: 关键词, “index”: i ]  如果找不到了，返回null  和str.match() 区别:  str.match()始终只能找第一个关键词  reg.exec()反复调用时，可自动向后找下一个   |  | | --- | | var kwords=["a","b","c"]  var msg=prompt("强输入内容")  var reg=new RegExp(kwords.join("|"));  //var i=msg.search(reg);  if(reg.text(msg)){ document.write("错")}  else{document.write("对")}  //if(i!=-1){ document.write("错")  //}else{document.write("对")} | |

|  |
| --- |
| 总结: 查找关键词: 5种  a.只判断是否包含:  var bool=reg.test(str)  b.查找一个固定关键词的位置:  var i=str.indexOf(“关键词”)  c.查找一个可变关键词的位置:  var i=str.search(/正则/i)  d.查找第一个关键词的内容和位置:  var arr=str.match(/正则/i)  arr[“0”:关键词, “index”: i ]  e.查找所有关键词的内容:  var arr=str.match(/正则/ig)  arr[关键词1, 关键词2, … ]  f.查找所有关键词的内容和位置:  do{  var arr=reg.exec(str);  if(arr!=null)  arr[“0”:关键词, “index”: i ]  else break;  }while(true); |

**四．Function**

|  |
| --- |
| **1.定义:**封装一段可重用的代码段的程序结构，再起一个名字  **使用:**如果一段代码可能被反复使用  程序的重要设计原则之一: DRY  Don’t Repeat Yourself |
| **2.创建**: 3种:  ①用声明方式:  function 函数名(形参列表){  函数体; return 返回值 }  问题: 会被声明提前(hoist)  回顾: 声明提前: 当程序开始执行前，先将var声明的变量和function声明的函数提前到当前作用域的顶部，集中创建，赋值留在原地，所有声明都集中放在当前作用域的顶部。  ②用赋值方式:  var函数名=function (形参列表){  函数体; return 返回值 }  *揭露了js中函数的本质: 函数其实是一个保存代码片段的对象，函数名其实就是一个普通的变量。*  *不会被声明提前！*  ③用new:  var 函数名=new Function("形参1",  ...,"函数体"; return 返回值) |

**专题（重组、匿名函数、作用域作用域链）**

|  |
| --- |
| **1.重载(overload):**  **定义:** 多个相同名称不同参数列表的函数，调用时可自动根据传入实参的不同，选择对应的函数执行。  **优点：**减少API的个数，减轻调用者的负担  **使用：**一件事，可能根据传入参数不同，执行不同的处理过程时  **步骤：**  问题: js语法默认不支持重载效果，js中不允许多个同名函数同时存在。若同时存在，最后一个同名函数会覆盖之前所有  解决: 变通实现，只定义一个函数，不定义参数列表  a.调用函数时，按需传入参数  b.所有参数都会被函数中的内置对象 arguments自动接收。  c.函数中根据参数列表的个数和内容，动态选择执行一种处理逻辑  *arguments: 每个函数中自动创建的可接受所有实参列表的类数组对象(长的像数组的对象)*  和数组比较:  相同：下标，.length，遍历  不同：不是数组类型，API不通用！ |
| **2.匿名函数**:  **定义**：函数时不指定函数名的函数  **特点:** 节约内存，匿名函数用完后，不再被任何变量引用着，就被垃圾回收！  **使用：**如果一个函数只用一次时  **步骤：**2个场景:  a.回调函数: 自己定义函数，却给别人去调用  b.自调函数:函数定义后，立刻自己调用自己执行一次，用匿名函数划分临时作用域，避免使用全局变量，避免使用全局变量,避免全局污染  **步骤：** (function(){  代码…  })()  *总结: 今后，禁止使用全局变量！今后，所有js代码一律放在一个大的匿名函数自调内！* |
| **3作用域(scope)和作用域链:**  作用域英文原文为scope，本意就是“范围”的意思  作用域是指一个数据的可用范围。  定义作用域是为了避免内外不同范围的数据间互相干扰  **JavaScript程序中，也包含两级作用域：**  a.全局作用域，指从任何位置都可访问的范围  b.函数作用域，指仅限于函数内才可访问的范围  **定义在不同范围的变量也有不同的名称。**  a.全局变量，是指保存在全局作用域中的变量。全局变量可在程序中的任何位置都能访问。  b.局部变量，是指保存在某一个函数内的变量。局部变量只能在函数内使用。  **函数执行时，也有两级范围的数据可用，分别是**  a.自己的函数作用域，其中包含自家带的面包；  b.全局作用域，其中包含商店和公园的面包  ***使用时，按照就近原则，优先使用自己函数作用域中的变量，自己没有，才使用全局范围内的。***  **作用域链，就是由多级作用域组成的链式结构**  当调用函数时，函数会将自己能用到的所有作用域都用“锁链”串联（引用）起来。  作用域链：  1. 串连着当前函数可使用的所有作用域范围，保存着当前函数可用的所有变量  2. 控制着变量的使用顺序：先局部，后全局；只要局部有，就不去全局找。 |
| **4. 闭包：**closure为一个函数保护一个专属的变量；  使用闭包原因：变量在函数外部很容易会造成全局污染，函数内部会有失忆情况，调用一次 重新定义一次;就是不可重用的意思，闭包，是既重用变量又保护变量不被篡改的一种编程方法。  闭包：1在函数的外部包裹一个函数；把要保护的变量包裹起来；  2想要获取到内层被包裹的函数 就要用到外层函数的返回值return;  3调用外层函数；获得内层函数，保存到全局变量中；由此返回的东西变成了变量的值，变成了函数名；  由于内层函数变成了外层函数的孩子，他们中间有total连接着，所以外层函数的变量 不会被重置  今后，只要希望既重用变量，又保护变量不被篡改时，就要用闭包为一个函数保管专属的变量。  如何: 3步  – 1. 外层函数包裹内层函数和要保护的变量  – 2. 外层函数返回内层函数对象到外部  – 3. 外部使用者调用外层函数，获得内层函数对象，并保存在外部变量中反复使用。  *创建内层函数时，内层函数的[[scopes]]属性，始终拴着自己可用的外层函数作用域。* |

**五．面向对象编程: OOP**

|  |
| --- |
| **定义:** 程序都是先用对象结构来描述现实中一个具体事物的属性和功能在按需使用对象的属性和功能  **优点:** 便于大量数据的管理和维护  **使用：**今后，所有程序都要用面向对象思想实现  三大特点（步骤）: 封装，继承，多态 |
| **1.封装:**  定义: 用对象结构集中存储一个事物的所有属性和功能  **优点：** 便于大量数据的管理和维护  **使用：** 今后，只要使用面向对象思想，都要先将零散的数据封装为对象  方式: 3种: {}、new、构造函数  **①用{}创建对象:**  var obj={  属性名:值,  … : …,  方法名:function(){  … this.属性名 …  }}  访问对象成员: 对象. 属性名  访问对象方法: 对象. 方法名()  问题: 对象自己的方法内，要使用当前对象自己的属性，不能直接写属性名——报错！未定义  原因: 对象不是作用域！所以不在方法的作用域链上，所以，即使自己的方法，也不能直接访问自己对象的属性  解决:  不好的解决: 写死“对象名.属性名”  问题: 如果对象名发生改变，被迫同时修改方法内的语句  好的解决: “this.属性名”  *this: 在函数中自动创建的关键词，自动引用正在调用当前方法的.前的对象*  **使用：**只要对象自己的方法，要访问自己的属性时，必须加this.  常犯错误: 以定义方法的位置来判断this的指向。  正确: this与定义的位置无关，只与调用时.前的对象有关。  2种:  a. obj.fun() this->obj  b. fun() this默认->window     |  | | --- | | var n=1;  var obj={  n:2,  fun:function(){  console.log(`${this.n}`)}}  obj.fun();  var fun=obj.fun;//这里为什么不加括号，表示不调用这个方法，  fun(); |   **② 用new:**  var obj=new Object();  obj.属性名=值;  obj.方法名=function(){ … }  *js中对象的本质: js中一切对象底层都是关联数组！*  a.访问对象的成员: 2种:  obj[“属性名”]=obj.属性名  如果属性名是动态变化的，必须用obj[变量]  b.访问对象中不存在的属性: 不报错！返回undefined  c.随时可给对象添加任何新属性: 不报错！而是自动创建  d.可以用for in遍历。  前两种方式问题: 一次只能创建一个对象，如果反复创建多个相同结构的对象时，代码大量冗余——不便于维护  **③用构造函数创建对象:**  **定义:** 描述一类对象相同结构的函数  **优点：**代码重用  **使用：**反复创建多个相同结构对象时  **步骤：**2步  1.定义构造函数:  function 类型名(){  this.属性名=值;  … = …;  This.方法名=function(){  … this.属性名 …}}  2.用构造函数反复创建多个相同结构的对象  var obj=new 类型名();  new: 4件事:  1.创建一个空对象  2.？（继承）  3.调用构造函数，并将构造函数中的所有this的指向自动替换为新的空对象，通过强行给新对象赋值的方式，向新对象中添加新属性  4.返回新对象地址  构造函数的问题：如果将方法定义放在构造函数中，会导致每次创建新对象时，都重复创建方法的副本——浪费内存  解决: 保存在一处，共用！ |
| **2.继承**:  **定义：**父对象的成员，子对象无需重复创建，就可直接使用  **优点**：代码重用，节约内存  **使用：** 所有子对象，需要相同的方法或属性时，只要在父对象中添加一次，所有子对象就可共用  步骤： js中继承，都是继承原型对象(prototype)  创建原型对象: 不用创建，每创建一个构造函数，都附赠一个空的原型对象  构造函数.prototype=自己的原型对象  何时继承:  new的第2步，自动设置继承关系，  子对象.\_ \_proto\_ \_=构造函数.prototytpe  向原型对象中添加共有成员: 强行赋值:  构造函数.prototype.共有成员=值/function(){...}  自有属性和共有属性:  获取属性值, 没有任何差别  差别在: 修改时: 自有属性，可用子对象修改；共有属性，不能用某一个子对象修改，只能用原型对象修改   |  | | --- | | 内置对象的构造函数和原型对象:  内置对象: ES标准中规定的，浏览器已经定义好的可直接使用的对象  包括: 11个  String Number Boolean  Array RegExp Date Math  Error  Function Object  global |   除了Math和global外，其余都是一种类型，都包含2部分组成:  a. 构造函数: 创建该类型的子对象  b. 原型对象: 保存该类型的所有子对象共有的成员  原型链:  定义： 由多级父元素逐级继承形成的链式结构  作用：  2个  a保存着所有对象可访问的属性和方法  b控制着属性和方法的使用顺序: 先用自己，自己没有才延原型链向上找（就近） |

|  |  |
| --- | --- |
| **3.多态:**  **定义：**一个函数在不同情况下表现出不同的状态  包括: 2种:   1. 重载: 2. 重写(override): 子对象觉得父对象的成员不好用，可在本地定义自有同名成员，覆盖父对象成员   **使用：** 只要觉得父对象的成员不好用，都可重写一个新的  **步骤：** 只要在子对象本地定义同名成员即可    **静态方法与实例方法:**  静态方法:  **什么是**: 不需要创建子对象，就可用构造函数直接调用的方法  **问题**: 原型对象中的方法，只能被当前类型的子对象使用  **何时**: 2种:  a.如果一个函数，不确定将来使用它的对象的类型时  b.如果一个函数在调用时，不需要提前创建子对象  步骤：将函数定义在构造函数对象上  构造函数.静态方法=function(){ ... }  调用时: 构造函数.静态方法()  强调: 调用静态方法之前不需要创建子对象  实例方法：  什么是实例方法: 给某个类型的子对象调用的方法  **使用：** 如果要求，方法执行时，必须先有一个具体的子对象，才能用。  **步骤：** 要么直接定义在对象上，要么定义在对象的原型对象上    **自定义继承关系:**  ①只修改一个对象的父对象:  子对象.\_\_proto\_\_=新父对象  或Object.setPrototypeOf(child,father)设置set child 的of爹prototype 为 father  问题: 一次只能改一个子对象的爹  ②同时修改所有子对象的原型对象  其实就是修改构造函数的prototype属性  构造函数.prototype=father  时机: 必须在定义完构造函数后，开始创建子对象之前，就要更换。  ③正式的两种类型间的继承:  问题: 两种类型间包含了部分相同的属性结构和方法定义  解决: 抽象父类型:  2步:   1. 定义抽象父类型:   a.在父类型构造函数中包含相同的属性定义  b.在父类型的原型对象中包含相同的方法定义  c.删除子类型中相同的内容   1. 让子类型继承抽象父类型:   a.让子类型构造函数继承父类型构造函数  Object.setPrototypeOf(  子类型.prototype, 父类型.prototype  )  b.让子类型构造函数借用父类型构造函数  错误: 直接调用父类型构造:父类型(参数值)  后果: 父类型构造函数中的this->window  解决: 函数内的this不是想要的，都可用.call()的方式，将正确的this注射进函数内，替换其中错误的this  正确的方法:  父类型构造.call(正确的this,参数值..)  结果: 执行时，父类型构造中的this，被外部正确的this给替换了！   |  | | --- | | function Enemy(fname,speed){  this.fname=fname;  this.speed=speed;  }  Enemy.prototype={  fly:function(){  console.log(`时速${this.speed}`)  }  }  Function plane(fname,speed,score){  Enemy.call(this,fname,speed);  this.score=score;  }  plane.prototype={  getScore:function(){console.log(`击落${this.fname},得到${this.score}分`)}}  plane.prototype.\_\_proto\_\_=Enemy.prototype;  var F16=new plane("tt",1000,20);  console.log(F16);  F16.getScore();  F16.fly(); | |

**六．ES5:**之前学的ES3.1

|  |  |
| --- | --- |
| 1.严格模式: 比普通js运行机制要求更严格的执行模式  **使用：**今后，所有项目代码，都必须运行在严格模式下！  **步骤：**在当前作用域顶部: **“use strict”**;  如果加在<script>元素或js文件顶部，则整段js代码都启用严格模式  如果只加在函数内的顶部，则仅函数内启用严格模式  **要求:**  a.禁止给未声明的变量赋值:  普通js中，可以给未声明的变量赋值  结果: 会自动在全局创建该变量  并且造成误会  （写错变量名不报错）！  b.静默失败升级为错误:  静默失败: 执行不成功,也不报错！  c.普通函数调用/匿名函数自调中的this，不再指向window，而是undefined  d.屏蔽了arguments.callee   |  | | --- | | **什么是arguments.callee**: 函数内自动创建的，引用当前函数本身的关键字。  **使用：**使用递归算法时，可在函数内避免写死当前函数名，变成松散耦合。屏蔽arguments.callee是在屏蔽递归算法，递归算法效率低，重复计算量太大！  **解决:** 几乎所有递归都可用循环代替！ |   斐波那契数列-递归：    斐波那契数列-循环： |
| 2.保护对象:   1. 保护对象的单个属性:   ES5将对象属性划分为两大类:  a.命名属性: 凡是可用.直接访问到的属性，又被分为两大类:  数据属性: 直接存储属性值的属性  访问器属性: 不直接存储属性值，  专门提供对另一个数据属性的  自定义保护  b.内部属性: 不能用.直接访问的属性  保护数据属性:  ES5将每个数据属性，都变成一个缩微的小对象：  { //四大特性:  value: 实际存储属性值,  writable: true/false, //控制能否修改  enumerable: true/false, //控制是否可  被for in遍历。但是用 . 依然可强  行访问该属性  configurable: true/false, //控制是否可  以删除该属性  //控制是否可修改另外两个特性  //一旦改为false，不可逆}  四大特性: value, writable, enumerable, configurable  获取四大特性（了解）: **var 属性对象=Object.getOwnPropertyDescriptor(obj,”属性名”)**  修改四大特性:  **Object.defineProperty(obj,”属性名”,{**  **开关: true/false,**  **… : …**  **})**  问题: 一次只能修改一个属性的四大特性  **Object.defineProperties(obj,{**  **属性名:{**  **开关: true/false,**  **… : …**  **},**  **属性名:{**  **开关: true/false,**  **… : …**  **}**  **})**  问题: 无法使用自定义规则保护属性  访问器属性: 访问器属性是不保存属性值，仅提供对另一个数据属性的保护——保镖  **使用：** 使用自定义规则保护属性时  **步骤：**  a.前提: 先有一个实际存储数据的隐藏的数据属性  b.定义访问器:  Object.defineProperties(obj,{  //访问器属性只能以此方式添加  \_数据属性:{  enumerable:false,  configurable:false  },  访问器属性:{  get:function(){  return this.\_数据属性  },  set:function(value){  if(value符合规则)  this.\_数据属性=value;  else throw new Error(“自定义错误信息”)},  enumerable:true,  configurable:false  }  })  问题：第一次传入的age未验证    问题： age验证函数执行重复写入 |
| 1. 保护对象结构:   保护对象的结构: 3个层次:  （1）防扩展: 禁止给对象添加新属性  Object.preventExtensions(obj)  阻止 扩展  设置obj对象，禁止扩展新属性  （2）密封: 在防扩展基础上，再禁止删除所有属性  本质:在设置防扩展同时，自动设置所  有属性的configurable为false  **步骤：** Object.seal(obj)  （3）冻结:在密封同时禁止修改属性值  本质: 在密封基础上，自动将所有属  性的writable设置为false！  **步骤：** Object.freeze(obj) |
| **小知识点：**   1. Object.create(): 没有构造函数，也想创建子对象   **步骤：** var child=Object.create(father,{  //语法相当于defineProperties的语法  属性:{ 四大特性 }  })  **原理：**创建新空对象child，然后让child继承father，并为新对象添加自有属性   1. 替换this:   .call/apply/bind  a.调用函数，并临时替换一次this为指定对象：.call/apply  fun.call(obj,实参…)  两件事: 调用一次函数执行；  在执行时临时替换其中的  this为obj。  .apply可打散数组为单个参数值后，  分别传入函数:  如果函数需要多个参数值，但给的  参数值却集中保存在一个数组中。  b.基于原函数创建一个完全相同的新函数，并永久绑定this为指定对象，bind  var newFun=fun.bind(obj,实参…)  创建一个和fun完全相同的新函数newFun，原函数不变。  永久将newFun中的this绑定为obj  永久将newFun中的部分参数绑定为实参…  *固定用法: 如果替换回调函数中的this，必须用.bind()。因为回调函数不是立刻执行！且不止执行一次！* |

**七．ES5数组API:**

|  |
| --- |
| 1.查找: 查找一个指定元素在数组中位置  var i=arr.indexOf(元素[, starti])  同字符串操作:str.indexOf() |
| 2.判断: 判断数组中的元素是否符合要求  两种:   1. 判断是否所有元素都符合要求：   var bool=arr.every(function(val,i,arr){  //val: 当前元素值  //i: 当前位置  //arr: 当前数组  return 判断条件;  })  every会拿着回调函数去每个元素上执行一次，每次执行时，都会将当前元素值，当前所在位置，当前数组对象，传入函数中  函数内用提供的参数值，判断当前元素是否符合条件，并返回判断结果  必须回调函数在每个元素上执行时，都返回true，整体判断结果才返回true！   1. 判断是否包含符合要求的元素:   var bool=arr.some(function(elem,i,arr){  //elem: 当前元素值  //i: 当前位置  //arr: 当前数组  return 判断条件;  }) |
| 3.遍历: 对每个元素都执行相同的操作  ①直接修改原数组  arr.forEach(function(elem,i,arr){  elem…  arr[i]…}) //没有return    forEach会拿着回调函数去每个元素上  执行一次  每次执行时，回调函数内都可用arr[i]  修改原数组中的每个元素值。  ②不直接修改原数组，而是返回新数组  var newArr=arr.map(function(elem,i,arr){  return 新值  })  a.map先创建一个新的空数组  b.然后拿着回调函数去每个元素上执行一次  c.每次执行时，都将回调函数返回的新值，放入新数组中相同位置  d.最后，将保存新值的新数组返回，原数组保持不变 |
| 4.过滤/汇总  过滤: 复制出原数组中符合条件的元素，组成新数组  var subArr=arr.filter(function(elem,i,arr){  return 条件  })  filter先创建一个新的空数组  会拿回调函数去每个元素上执行一次  每次执行时，都获得当前元素的判断结果  只有返回结果为true的元素，才被放入新数组中返回 |
| 汇总: 对数组中的元素进行各种统计，得出最终结论（结果可能是数值或对象）  var result=arr.reduce(function(prev,elem,i,arr){  //prev: 截至到当前元素之前的临时汇总值  return prev+elem;  },start) //开始的初始值 |

**八．ES6**

**不改变原理基础上尽量简化代码**

|  |
| --- |
| 1.变量声明:  let/const: 代替var/const来声明变量和常量的  ES5缺点：  a.因为传统的var和const会被声明提前，ES6  b.没有块级作用域: for if while都不是作用域，其中的变量会被提前到外部，影响外部的代码。  ES6优点:  a.不会被提前，不允许提前使用未声明的变量  b.让for if while等程序块，也变为作用域  *原理: let自动添加匿名函数自调来划分临时作用域，并将变量名前自动加\_，与其他变量区分。* |
| 2.参数增强:  ①默认参数(default): 即使用户不传入参数，参数也有备用的默认值代替  步骤： 定义函数时: function fun(形参,  …,形参=默认值)  *强调: 有默认值得参数，必须在形参列表的末尾*  ②剩余参数(rest): 代替arguments接收所有不确定个数的参数  arguments: 2个问题:  a.不是数组类型，不能使用数组API  b.只能获得全部参数，无法有选择  的获得部分  语法: function fun(形参, ...数组名)(定义时)  优点:数组名获得的是纯数组，且有选择的获得部分，数组可收集除前几个确定参数之外的剩余参数   |  | | --- | | function calc(ename,...bonus){  document.write(`${ename}的总工资是${bonus.reduce((prev,elem)=>prev+elem)}</br>`)}  calc("lilei",1200,1300,10000)  // lilei的总工资是12500 |  1. 散播(spread): 代替apply专门用于打散数组为单个元素，再传入函数。   apply的问题: 主要功能是替换this，顺便可以打散数组为单个值。（不是主业）  步骤：调用函数时: fun(...数组) |
| 3.箭头函数: 对回调函数或匿名函数自调的简写  **步骤：**回调函数: function (形参,…){ … }  可简化为: (形参,…)=>{ … }   |  | | --- | | 如果只有一个形参，可省略()  如果函数体中只有一句话，可省略{}  如果这仅有的一句话还是return xxx，则必须省略return |   *特点: 箭头函数内外的this是同一个/共通的*  总结: 如果希望内外this相同时，应该简化，如果反而希望内外this不同时，不能简化！ |
| 4.解构: 从一个大的对象中抽取想要的部分成员，单独使用。释放到全局，方便调用。  三种:   1. 数组解构: 从数组中抽取想要的元素出来，单独使用   **步骤：** 下标对下标:  var arr=[1,2,3];  ↓  var [x,y,z]=arr  console.log( x,y,z)  结果: x=1, y=2, z=3  arr[0]//麻烦，且没有意义   1. 对象解构: 从对象中抽取想要的成员出来，单独使用   **步骤：** 属性名对属性名  var obj={x:1, y:2, z:3}  ↓ ↓ ↓  var {x:a, y:b, z:c}=obj;  obj.x obj.y obj.z //麻烦  结果: a=1, b=2, c=3  简写:属性名和变量名一致时，只写一个 var {x:x, y:y, z:z}=obj;  可简写为var {x,y,z}=obj;   1. 参数解构: 其实是对象解构在函数传   **何时:** 多个参数都可选时  **步骤：**2步:  a.定义时:将参数列表定义为对象语法:  //function fun({属性1:形参1,属性2:形参2,...})  function fun({形参1,形参2,...}){ ... }  b.调用时: 将传入的参数放在一个对象中整体传入  fun({  属性1: 值1, ...  })  执行: fun将整个实参对象传给形参对象，形参对象通过解构，从实参对象中抽取对应的参数值。  如果找不到对应的，则形参值默认为undefined |
| 5.for of: 最简化的遍历索引数组或类数组对象的方式：  总结: 遍历索引数组:  ①for(var i=0;i<arr.length;i++){  var elem=arr[i];  }  最灵活，万能  ②arr.forEach((elem,i,arr)=>{ elem})  无法控制遍历顺序和步调  ③for(var elem of arr){ … }  of会依次取出arr中每个元素值，保存在of前的变量中  无法获得位置i  无法修改原数组中的值   |  | | --- | | for of 和 for in比较  for of 专门遍历下标为数字的索引数组/类数组对象----数字下标  of取得是元素值  for in 专门遍历下标为自定义名称的关联数组/对象----自定义下标  in取得是属性名 | |
| 6.class:  ES6对整个面向对象语法的简化   1. 对象直接量的简化   对象名和变量名相同的保留一个  方法去掉 “：function”     1. 对创建一种类型的简化:   **定义：** 集中描述一类对象统一属性结构和行为的程序结构，今后只要创建一种类型，必须用class  **步骤：**  a.用class{}包裹原来的构造函数和原型对象方法  b.构造函数名提升为class名，构造函数要更名为constructor  c.直接定义在class内的方法，默认保存在原型对象中，且不用加Xxx.prototype前缀和”=function”③继承: 2步  a.class child extends father{}  不再需Object.setPrototypeOf(…)  b.子类型构造函数中: super(参数值)  不需要传入this，super自动指向  extends后的父类型，super()是调用  父类型构造函数的意思  *super必须放在构造函数第一句*  ④访问器属性: 依然要在构造函数内定义受保护的隐藏的数据属性  在class内:  get 访问器属性名(){ return this. …}  set 访问器属性名(value){ 验证value并给this.xxx赋值}  ⑤静态方法:  class 类名{  constructor(){}  方法(){}  static 静态方法 (){ }  } |
| ⑥Promise: 代替回调函数，实现多个异步调用，顺序执行(前一件事办完，再办)  错误做法：仅顺序编写  test1();  test2();  传统的做法: 利用回调函数:    问题: 使用回调函数方式实现多个异步调用顺序执行会多级函数嵌套，导致回调地狱(callback hell)  根源: 所有回调函数规定，在调用函数前，就要提前传入到函数中  解决: 让回调函数在函数后传入  步骤：  前提: 不要在参数列表里传递回调函数了！  （1）定义函数支持Promise  在原函数内，用new Promise()包裹所有原代码  再在源代码外层套一层function(){}  function()中必须接收Promise附赠的open开关  在当前函数异步任务调用后，自动打开开关open();   |  | | --- | | function fun(){  return new Promise(function(open){  异步任务  异步任务执行完:open()  })  } |   （2）将多个任务串联起来:   |  | | --- | | 第一个函数().then(第二个函数).then(…) |   强调: 中间的then中的函数，不要加()，因为不是立刻执行，且中间的函数必须支持Promise   |  | | --- | | 错误处理: 2步:   1. new Promise(function(open,err){   //如果出错:  err(“错误消息”)  //通向最后的.catch()  })  ②在函数1().then().then()…..catch(function(errMsg){ … })  无论中间哪个then出错，都会执行最后的catch，并将then中err(“错误消息”)传给errMsg。 | |

**DOM**

**一．什么是DOM: Document Object Model**

|  |
| --- |
| DOM:专门操作网页内容的API标准--W3C  优点：统一所有浏览器操作网页内容的API  几乎所有浏览器100%兼容DOM API  包括: 增删改查+事件绑定  查找网页上的元素  修改元素  添加新元素  删除现有元素  事件绑定  使用：只要js操作网页内容，只能用DOM API |

**二．DOM Tree**

|  |
| --- |
| ①定义：内存中保存一个网页中所有内容的树形结构  优点：因为网页中的内容，都是有明显的上下级包含关系的。  使用：只有理解了网页的树型结构，才能顺利找到要操作的DOM元素  步骤：  a.当浏览器读到一个HTML文件时，会自动在内存中创建一个唯一的树根节点：document  b.浏览器解析HTML中的内容，每读到一项内容，就创建一个节点对象，然后将节点对象添加DOM树上    ②节点对象:  网页中每一项内容(元素，属性，文本)都是一个节点对象  节点对象的三个公共属性：(了解)  .nodeType: 判断该节点对象的类型，值为一个数字  包括: document 9 根节点  element 1 元素节点  attribute 2 属性节点  text 3 文本节点  问题: 只能判断类型，不能进一步判断元素名  .nodeName: 判断节点的名称:  包括: document #document  Element 全大写的标签名  Attribute 获得属性名  Text #Text  .nodeValue: 获得节点的值  包括: document null  Element null  Attribute 属性值  Text 文本内容 |

**三．查找: 四种**

|  |
| --- |
| 1. 不需要查找，就可直接获得的元素:  <html> document.documentElement  <head> document.head  <body> document.body  <form> document.forms[i/id] |
| 2. 按节点间关系查找:  ①**节点树**: 包含所有网页内容的最完整的树结构  两大类关系:  a.父子关系:  node.parentNode 获得node节点  的父节点  node.childNodes 获得node下所有  直接子节点的集合  node.firstChild 获得node下第  一个直接子节点  node.lastChild 获得node下最后  一个直接子节点  b.兄弟关系:  node.previousSibling 获得node节  点的前一个兄弟  node.nextSibling 获得node节点  的后一个兄弟  问题: 节点树会受到看不见的回车，换行，空字符的影响  程序员通常只关心元素节点  解决:元素树,仅包含元素节点的树结构**②元素树**  好处:不包含看不见的空字符  强调: 元素树不是一棵新树，只是节点树的子集。  两大类关系:  a.父子关系:  node.parentElement 获得node节  点的父元素  node.children 获得node下所有直  接子元素的集合  node.firstElementChild 获得node下  第一个直接子元素  node.lastElementChild 获得node  下最后一个直接子元素  b.兄弟关系:  node.previousElementSibling 获得  node节点的前一个兄弟元素  node.nextElementSibling 获得node  节点的后一个兄弟元素  D:\0myself\正式课\11DOM\DAY01\day01\iterator.png  **③递归遍历**: 指定父元素下所有后代元素，2步:  a. 先定义函数，仅遍历指定父元素的直接子元素  b.对每个直接子元素，调用和父元素完全相同的操作    问题: 必须先获得一个元素，才能按节点间关系查找  解决: 用HTML特征查找 |
| 3.按HTML特征查找: 4种  id tag name class  何时使用：在没有获得任何元素的情况下，可作为首次查找之用   1. 按id查找:   var elem=document.getElementById("id");  返回值: 返回一个元素对象，找不到，返回null  强调: 只能用document调用   1. 按标签名查找:   var elems=parent.getElementsByTagName  (“tag”)  返回值: 返回所有符合条件的元素对象的集合  如果找不到，返回空集合: [].length=0  强调: a.可在任意父元素上调用  b.**不仅查找直接子元素**，且在所  有后代中查找，返回多个元素  组成的类数组对象  **③** 按name属性查找:  var elems=document.getElementsByName  (“name”)  使用： 专门查找表单中收集数据的表单元素  返回值: 多个表单元素的集合，如果找不到返回[]  如果明知道只找到一个元素: elems[0]  强调: 只能用document调用  返回多元素组成的类数组对象  **④** 按class属性查找:  var elems=任意父元素.getElementsByClass  Name(“class”)  返回值: 多个元素的集合，如果找不到返回[]  强调: a.可在任意父元素上调用，控  制查找范围  b.返回多元素组成的类数组对象  c.**不仅查找直接子元素**，且在所  有后代中查找  d.如果一个元素有多个class饰，  则只需要其中一个class就  可找到该元素。  问题: 一次只能用一个条件查找，如  果查找条件复杂，代码会很繁琐！ |
| 4.用选择器查找元素: 2个:  使用:当查找条件复杂时,首选按选择器查找   1. 仅查找一个元素:（只返回第一个）   var elem=parent.querySelector("选择器")   1. 查找多个元素:   var elems=parent.querySelectorAll("选择器")   |  | | --- | | 按HTML查找 vs 按选择器查找:  （1）返回值:  按HTML查找返回动态集合: 不实际存储属性值，每次访问集合都重新查找DOM树  按选择器查找返回非动态集合: 实际存储完整属性值，每次访问集合不需要重复查找DOM树  （2）首次查找效率:  按HTML查找，效率高！  按选择器查找，效率低  （3）易用性:  按HTML查找功能单一  按选择器查找，功能强大，代码简洁  总结: 查找优化  a.如果用一个条件就可找到元素时，首选按HTML查找  b.如果查找条件复杂时，首选按选择器查找 | |
| DOM操作：5步   1. 查找触发事件的元素 2. 绑定事件处理函数 3. 查找要修改的元素 4. 修改元素   在处理函数中应该用this关键字获得将来触发事件的当前元素本身 |

练习：实现伸缩二级菜单

|  |
| --- |
|  |

|  |
| --- |
|  |

**四．修改（3种）内容/属性/样式**

|  |
| --- |
| **1.内容**：修改和获取元素内容   * 1. 原始HTML片段:（首选）   elem.innerHTML 可获取或设置元素开始标签到结束标签之间的HTML片段   * 1. 纯文本内容:（存在兼容性问题）   elem.textContent 可获取或设置元素开始标签到结束标签之间的纯文本内容  比innerHTML多做两件事:  a.去掉内嵌的标签  b.特殊符号翻译为正文   * 1. 表单元素的值: elem.value   (单标签不能用) |
| **2.属性**: 3种:   1. HTML标准属性: HTML标准中规定的属性 2种方式:   **核心DOM API:** DOM最初指定的可操作一切结构化文档(HTML、XML)的API: 可操作任何出现在元素开始标签中的属性  4个API:   |  | | --- | | elem.getAttribute("属性名")  elem.setAttribute("属性名","值")  elem.removeAttribute("属性名","值")  elem.hasAttribute("属性名") |   **HTML DOM API**: 专门针对HTML中固定的属性，提供的简化版API  HTML DOM提前将HTML规定的标准属性都定义在了元素对象上，可用 .直接访问 elem.标准属性名  冲突: class属性: elem.className  因为class是ES中的保留字   1. 状态属性: 4个:  |  | | --- | | enabled disabled selected checked |   值都是bool类型  不能用核心DOM 4个API  只能用HTML DOM打.访问       1. 自定义扩展属性:   定义： HTML标准中没有规定的，程序员自己添加的属性  作用： 2种:  a.在客户端临时缓存业务数据，减少向服务器端发送请求的次数  b.代替其他选择器(id,class,元素)，为元素绑定行为  id 只能选一个元素  元素 限制太死板  class 是样式专用，不稳定  解决: 今后只要为元素添加行为，都应该用自定义扩展属性  步骤：bootstrap:  a.为想要添加行为的元素，指定自定义扩展属性  <a data-toggle="dropdown"  b.用属性选择器找到有自定义扩展属性的元素绑定行为:  var elems=document.querySelector  All("[data-toggle=dropdown]")  for(var elem of elems){  elem.on事件名=function(){ ... }}  如何:  1. 定义时: <ANY 自定义属性名="值"  2. 获取或设置自定义属性的值:   |  | | --- | | 不能用.修改，因为内存中的元素对象中不包含自定义扩展属性  用核心DOM API可以访问自定义扩展属性:  elem.getAttribute("属性名")  elem.setAttribute("属性名","新值")  elem.hasAttribute("属性名")  elem.removeAttribute("属性名")  HTML5:  1. 如果在定义自定义扩展属性时加了data-前缀  2. 在程序中: elem.dataset.属性名  其中: dataset可自动收集data-\*开头的属性 | |
| **3.样式**:  ①修改一个css样式:  elem.style.css属性名="新值"  其中: style代表元素开始标签中的内联样式style属性  强调: 所有css属性名都要去-，变驼峰命名  ②获取一个css样式:  错误: elem.style 只能获得内联样式  解决: 今后只要获取css属性值，都用计算后的样式  什么是计算后的样式: 最终应用到元素上的所有css属性的集合。并将相对单位的属性值计算为绝对单位(px)  如何: 2步:  a．先获得计算后的完整样式对象  var style=getComputedStyle(elem)  b. 从style中获得想要的css属性值  style.css属性名  强调: 计算后的所有样式，只读。因为计算后的样式来源不确定，不知道多少地方正在同时使用。  总结: 今后只要修改css属性，都用elem.style.css属性=值  今后只要获取css属性，都用getComputedStyle  问题: elem.style，一次只能修改一个css属性，如果批量修改多个css属性时，代码繁琐  解决: 今后只要批量应用样式，都用class |

**五．添加/删除元素:**

|  |  |
| --- | --- |
| ①添加元素: 3步:  （1）创建新的空元素:  var a=document.createElement(“a”)  <a></a>  （2）设置关键属性:  a.innerHTML=”to tmooc”  a.href=http://tmooc.cn  <a href=”http://tmooc.cn”>to tmooc </a>  （3）将新元素挂到DOM树上指定父元素下: 3种:  父元素.appendChild(a) 将a追加到parent的所有直接子元素之后  父元素.insertBefore(a,child) 将a插入到parent的子元素child之前  父元素.replace(a,child)用a代替parent的子元素child的位置  /\*\**动态生成表格\*\*/*     |  | | --- | | 优化：尽量减少操作DOM树的次数，每修改一次DOM树，都会导致重排重绘，降低页面加载效率  HTML加载过程:  html -> DOM Tree  ↓  RenderTree -> layout -> paint  ↑  css -> CSS Model  解决: 2种:  a.如果同时添加父元素和多个子元素，应该先在内存中，将所有子元素添加到父元素后，再最后一次性将父元素添加到DOM树  b.如果父元素已经在页面上了，要添加多个平级子元素时。应该使用文档片段：  文档片段: 内存中，临时保存多个平级子元素的虚拟父元素  使用：如果父元素已经在页面上了，要添加多个平级子元素  步骤： 3步:  a.创建文档片段对象:  var frag=  document.createDocumentFragment();  b.将多个子元素临时加入到文档片段中:  frag.appendChild(child)  c.将frag整体一次性加入到DOM树的父元素下  parent.appendChild(frag) | |

练习：级联下拉列表

|  |
| --- |
|  |

|  |
| --- |
| ②删除: parent.removeChild(child)  child.parentNode.removeChild(child)  让父元素，删除其下的一个子元素child |

**六．HTML DOM常用对象:**

|  |
| --- |
| 1.Image : 代表页面上一个<img>元素  简化: 创建: var img=new Image();  强调: 特例！其它元素，不能new! |
| 2.Select: 代表页面上一个<select>元素:  属性: value 可获得当前选中项的值  如果选中的option，没有value属性，则用内容代替  selectedIndex 可获得选中项在select中的下标位置  options 可获得select下所有option对象  options.length 可获得select下所有option的个数  length = options.length  .add(option) 代替: .appendChild()  问题: .add一次只能添加一个option，不支持frag  .remove(i) 移除i位置的option  Option: 代表页面上一个<option>元素  创建: var opt=new Option(text,value)  代替: .createElement(“option”)  .innerHTML=text;  .value=value; |
| 3. table  （1）Table:代表着页面上一个<table>元素，直接管着行分组:  **①**添加行分组:  var thead=table.createTHead();  相当于:  //.createElement(“thead”);  //.appendChild(thead);  var tbody=table.createTBody();  var tfoot=table.createTFoot();  **②**删除行分组: table.deleteTHead()  table.deleteTFoot()  获取行分组: table.tHead  table.tFoot  table.tBodies[i]  (table可以有多个tbody，所以table下是数组，无法控制删除tbody)  （2）行分组对象: 代表页面上table下一个行分组元素管着行:  ①添加新行: var tr=行分组.insertRow(i)  固定套路:  行分组.insertRow() 末尾追加！  行分组.insertRow(0) 开头插入!  ②删除行: 行分组.deleteRow(i)  强调: i 是行在行分组内相对下标位置  矛盾:  行分组.deleteRow(i) i 要求是行分组内的相对下标位置  每个行上都有一个rowIndex属性， 记录着行在整个表中的下标位置  所以: 行分组.deleteRow(tr.rowIndex)  结果会删除tr的下一行  解决: 今后删除行，一定不要用行分  组作为.前的主语  应该用table.deleteRow(tr.rowIndex)  **③**获取行: 行分组.rows[i]  （3）tr管着格:  添加格: var td=tr.insertCell(i)  通常都是末尾追加: tr.insertCell()  局限: 只能添加td，不能添加th  删除格: tr.deleteCell(i)  获取格: tr.cells[i] |
| 4. Form: 代表页面上一个<form>元素  获得表单对象: var form=document.forms[i]  ①属性: .elements 获得表单中所有表单元素的集合(input/button/textarea/select)  .elements.length 获得表单中表单元素的个数  .length => .elements.length  方法: form.submit() 手动提交表单  自定义表单提交:   <input type="button"  .onclick=function(){  先验证所有表单元素的值是否符合要求  只有所有表单都符合要求  才手动调用form.submit() }  ②表单元素对象:  获取:var elem=form.elements[i/name/id]  更简化: 如果表单元素有name属性，则可直接写:  form.name |

|  |
| --- |
| 方法: elem.focus() 自动获得焦点  elem.blur() 自动失去焦点 |

**BOM**

|  |
| --- |
| 1什么是BOM: Browser Object Model  专门操作浏览器软件的API  问题: 没有标准！很多API已经废弃  包括:  window: 是包含所有ES,DOM,BOM  原生API的全局对象  history: 掌管历史记录  location: 掌管地址栏  document: 掌管所有网页内容: DOM  navigator: 掌管浏览器的配置信息  screen: 掌管显示设备的信息  event: 掌管着事件相关操作 |
| 2.Window:  ①2个角色:  a.代替ES中的global充当全局作用域  对象  b.封装所有ES,DOM和BOM的原生API  的大的对象  ②属性: 获得浏览器窗口大小: 2组:  a.完整窗口大小:  window.outerWidth/outerHeight  用途: outerHeight<指定值, 可判断窗口是否最小化了  b.文档显示区大小:  window.innerWidth/innerHeight  文档显示区：浏览器窗口中，专门用于显示网页内容的区域 |
| 3.打开和关闭窗口:  API: window.open()  window.close()  打开新链接: 4种:   * 1. 在当前窗口打开，可后退   Html: <a href=”url” target=”\_self”>  Js: open(“url”,”\_self”)   * 1. 在当前窗口打开，禁止后退   Js: location.replace("新url")  原理: 用新url代替history中现有的url   * 1. 在新窗口打开，可打开多个   Html: <a href=”url” target=”\_blank”>  Js: open(”url”, ”\_blank”)   * 1. 在新窗口打开，只能打开一个   Html: <a href="url" target="自定义窗口名">  Js: open("url","自定义窗口名")   |  | | --- | | 其实, <a>的target属性是新窗口的名字(name)  name是在内存中唯一标示一个窗口的名称  浏览器规定，相同名称的窗口，只能打开一个  新打开的同名窗口，会覆盖旧窗口  预定义窗口名:  \_self: 自动获得当前窗口自己的名字给新窗口  \_blank: 不指定窗口名，让浏览器随机生成窗口名 | |
| 4.history:  **定义：** 保存当前窗口打开后，成功访问过的url的历史记录栈(数组)  控制着: 前进后退  **步骤：** history.go(n)  history.go(1) 前进一步  history.go(-1) 后退一步  history.go(-2) 后退2步  history.go(0) 刷新 |
| 5. location:  **定义：**专门保存当前窗口正在打开的url的对象，还提供了API  **使用：**a.分段获得url中部分内容时  b.页面跳转/刷新时  **属性:** .href 获得完整的url信息  .protocol 协议 http: /https:/ftp:  .host 主机名+端口号  .hostname 主机名  .port 端口号  .pathname 相对路径  .hash #锚点地址  .search ?查询字符串  **方法:**  a.在当前窗口打开，可后退:  location.href="新url"  location.assign("新url ")  可简写为: location="新url"  b.在当前窗口打开，禁止后退:  location.replace("新url")  c.刷新页面:  普通刷新: 优先从浏览器本地缓存中获取资源，本地缓存中没有或过期，才被迫重新从服务器下载。  history.go(0)  location.reload()  F5  强制刷新: location.reload(true)  强迫: 始终从服务器下载新资源 |
| 6. navigator:  **定义：**专门保存浏览器配置信息的对象  包括:  ①.cookieEnabled: 判断是否启用cookie  **什么是cookie:** 客户端本次持久保存用  户私密信息的小文件  **优点:**程序内存中的数据都是临时的。  **使用:**只要希望持久保存的数据，都应  该放在cookie中  典型: 记住密码  设置cookie：设置-高级-隐私xxx内容  设置-cookie-启用/禁用  ②.plugins: 包含所有插件的集合  **使用:**判断是否安装某个插件:  **步骤**：if(navigator.plugins[“插件名”]===  undefined)  说明没装  else  说明装了  ③.userAgent: 保存浏览器名称和版本号的字符串  **使用：**专门用于判断浏览器的名称和版本号 |
| 7. 定时器:  2种:   1. 周期性定时器: setInterval()   **定义：** 让程序每隔一段时间间隔，反复执行一项任务  **步骤：**  启动: var n=setInterval(function(){  任务  }, 间隔ms)  停止: clearInterval(n)  //n是内存中定时器的唯一序号   1. 一次性定时器: setTimeout()   **定义：** 让程序延迟执行一项任务，执行后自动停止定时器。  **步骤：**  启动: var n=setTimeout(function(){  任务  }, 延迟ms)  停止: clearTimeout (n)  练习：倒计时牌    倒计时自动关闭/跳转页面 |
|  |

**事件:**

**1事件:**

|  |  |  |
| --- | --- | --- |
| **定义**：浏览器自动触发的，或者用户手动触发的页面内容/状态的改变  **什么是事件处理函数**: 在事件发生时，自动指定的回调函数  **使用**： 如果希望在发生事件时，执行一项任务  **步骤**： 事件处理函数都要先绑定到元素上：3种:  a.HTML中: <ANY on事件名="js语句"  问题: 不符合内容与行为分离的原则，不便于维护  b.在js中,用赋值方式:  elem.on事件名=function(){  this->当前触发事件的.前的元素  }  当事件发生时: elem.onclick()  问题: 一个事件只能绑定一个处理函数  c.在js中，添加事件监听对象:  elem.addEventListener("事件名",处理函数)  elem.removeEventListener(“事件名”,原处理函数)   |  | | --- | | 原理：在elem元素的监听队列中添加一个监听"事件名"的对象，其中包含一个处理函数。当elem元素上发生"事件名"时，会遍历监听队列，找到符合条件的处理函数，自动执行处理函数。 | | 强调: 1. 如果一个处理函数可能被移除，绑定时就不能用匿名函数，应该用具名函数  2. 移除时，必须通过函数名找到原函数对象 | |

**2.** **事件模型**

|  |
| --- |
| 事件模型:事件发生时，浏览器的执行过程，DOM规定: 3个阶段  ①捕获: 由外向内，依次记录各级父元素上绑定的处理函数  ②目标触发: 优先触发目标元素上的处理函数  目标元素: 最初实际触发事件的那一个元素  ③冒泡: 由内向外，按捕获的记录，反向执行各级父元素上绑定的处理函数  所有浏览器的开发者一致认为，点在内层元素上，等效于也点在外层元素上 |

**3.事件对象:**

|  |
| --- |
| 定义：事件发生时，自动创建的保存事件信息并提供操作事件的API的对象  **使用：**1. 想获得事件信息时  2. 想改变事件的默认行为时  **步骤：**  获取事件对象: 事件对象总是作为处理函数的第一个参数自动传入处理函数  elem.onclick = function(e) { e->事件对象}  ↑  当事件发生: elem.onclick(新建event对象) |

API:

|  |
| --- |
| 1. 取消冒泡: e.stopPropagation()   //停止 蔓延 |
| 2. 利用冒泡:  优化: 尽量减少事件监听对象的个数！  因为: 浏览器触发事件处理函数是采用遍历的方式查找监听对象。如果监听对象多，遍历慢  解决: 当多个子元素，需要绑定相同的事件时，其实只在父元素绑定一次即可！所有子元素通过冒泡共用！  2个难题:  ①获得目标元素，而不是父元素  错误: this->父元素  正确: e.target  ②判断目标元素是否想要的  在处理函数中先用if判断e.target是否想要的  只有是想要的时，才执行操作 |
| 3. 阻止默认行为:  **使用**：只要元素自带的默认行为不是想要的，就可阻止  e.preventDefault();  **常用**:  ①用a当按钮时: 阻止a自动添加#xxx到地址栏  ②其实submit按钮也能阻止提交！  input type=submit 触发  form.onsubmit=function(e){  如果验证不通过  e.preventDefault() }  ③HTML5的拖拽API |
| 4. 鼠标位置: 3组:  ①相对于屏幕左上角的坐标:  e.screenX e.screenY  ②相对于文档显示区左上角的坐标:  e.clientX e.clientY  ③相对于当前元素左上角的坐标:  e.offsetX e.offsetY  **滚动事件:**  window.onscroll=function(){  var  scrollTop=document.body.scrollTop ||document.documentElement.scrollTop  }    练习：拖动 |

**JQERY**

**一．什么是jQuery**

|  |  |
| --- | --- |
| **jQuery是**: 第三方开发的，执行DOM操作的极简化的函数库   |  | | --- | | 第三方: 先下载才能用  执行DOM操作: 学JQuery还是在学DOM  极简化: jQuery对DOM的每个操作/API都进行了终极的简化  函数库: jQuery中使用函数来解决一切问题 | |
| **优点** 2个原因:   1. 是DOM操作API的终极简化 2. 屏蔽了大部分浏览器兼容性问题:凡jQuery让用的，都解决了兼容性问题   **使用：**今后所有PC端的项目/框架底层都是jQuery |

**二．如何使用:**

|  |
| --- |
| 下载: jquery.com  版本: 1.x 支持IE8  jquery.js: 未压缩版本(uncompress development)   1. a.有完备的注释 2. b.有清晰的代码结构 3. c.有见名知义的变量名   **使用：**学习/开发阶段，可读性好  **缺点:** 体积大，不便于快速加载  jquery.min.js: 压缩版本(compress product)  a.删除注释   1. b.清楚代码结构 2. c.极简化了变量名   **使用：** 生产环境，体积小，便于快速加载  **缺点:** 几乎不可读  2.x 不再支持IE8  3.x 不再支持IE8  新特性: 1. 运行在Strict Mode下  2. 引进for...of循环，代替each  3. 新动画API: requestAnimation  Frame()  4. 兼容JS Promises |

**三．jQuery的原理:**

|  |
| --- |
| **引入<script src="jquery.js"**  其实是向全局添加一种新类型: jQuery，包含2部分  a.构造函数: function jQuery(){}  b.原型对象: jQuery.prototype={所有简化版API }  限制: 只有jQuery创建的子对象，才能使用jQuery简化版API  所有，如果想用jQuery简化版API操作DOM元素，都要先创建一个jQuery对象，封装要操作的DOM元素  **如何创建: 2种:**  ①先查找，再创建  本来: var $btn1=new jQuery("选择器")  但是: 因为new jQuery使用非常频繁，所以，jQuery构造函数进行了改造:  function jQuery(){  return new jQuery(); }  $=jQuery()  所以，创建jQuery对象，只要:  **var $btn1=$("选择器")**  执行: 先创建jQuery对象  再用选择器找到要操作的DOM元素对象，保存进jQuery对象中  调用API时: 对jQuery对象调用API，等效于对其内部封装的DOM元素调用对等的API.  ②直接封装已经获得的DOM元素：  **var $btn=$(this)**  this默认返回DOM元素，不能使用jQuery家的API  $(this)是将this返回的DOM元素封装成一个jQuery对象  创建jQuery对象的结果是:  *jQuery对象其实是一个封装多个DOM元素的类数组对象* |
| 几乎所有jQuery API都有三大特性:   1. 自带遍历效果:   $btns.click(function(){ … }) 可给找到的所有btn都绑 定事件处理函数  $lis.css(“css属性”,”值”) 可给找到的所有li都设置css属性   1. 一个函数两用:   如果给了新值，就修改；没给新值，就获取  **$().html() 获取内容**  **$().html("新内容") 修改内容**  **$().css("css属性") 获取css属性值**  **$().css("css属性","新值") 修改css属**  **性值**  **③**几乎所有API都返回正在使用的jQuery对象，可以用链式操作简化代码 |

**四．查找:**

|  |  |
| --- | --- |
| 1.用选择器查找:  var $elem=$("选择器")  如果选择器找到多个元素，$elem其实是个类数组对象，完全可以保存多个元素  包括: jQuery支持所有CSS选择器，并扩展了少量新选择器  ①基本选择器: 同CSS  #id elem .class \* 群组选择器  ②层级选择器: 同CSS  父 后代 父>直接子级  兄+弟 兄~弟  ③子元素过滤: 同CSS  :first-child :last-child  :nth-child(n)(1开始) :only-child  ④基本过滤: 位置过滤: jQuery新增的  先将符合条件的元素集中放入一个集合中，统一从0开始编号，再取指定为的元素。  :first/last :even/odd :eq/gt/lt(i)  *（底层用js模拟，不是css选择器）*   |  | | --- | |  | |
| 用链式操作简化代码 |
| 2.属性选择器:同CSS  [属性名]  [属性名=值]  [属性名^=开头]  [属性名$=结尾]  [属性名\*=部分]  [属性名!=值]（新增，非css）  *CSS中多个选择器连用默认就表示”而且”的意思，“,”表示或者的意思*  否定一个条件: :not(seletor) |
| 3.可见性过滤: :visible :hidden  :hidden 只能匹配display:none和  inputtype=hidden |
| 4.内容过滤:用元素的内容作为判断条件:  ①用元素的内容文本作为条件:  **:contains(文本)** 匹配元素内容中包含指定"文本"的元素  ②用子元素特征来判断或选中父元素:  **:has(selector)** 匹配包含符合selector要求的子元素的父元素    **:empty** 选择内容为空的元素  **:parent** 选择内容不为空的元素 |
| 状态过滤:  ①css中  :enabled :disabled :checked :selected  ②表单元素过滤:  :input 选择所有表单元素:  input button select textarea  *input是元素选择器，只能选择input元素*  :[type] 每种type都对应一种专门的选择器:  :text :password :checkbox :radio :submit :reset…  扩展:  $().is(selector) 判断当前元素是否符合selector条件的要求  返回值: bool |

**五．修改:**

|  |
| --- |
| 1.内容:  原始HTML片段: $().html([新HTML])  代替 .innerHTML  纯文本内容: $().text([新内容])  代替 .textContent  表单元素的值: $().val([新值])  代替 .value |
| 2.属性  ①HTML标准属性:  $().attr("属性名"[,"值"]) 代替:  .getAttribute() .setAttribute()  其实可修改一切出现在开始标签中的值为字符串的属性  ②状态属性: enabled disabled checked selected  $().prop(“属性名”[,值]) 代替: elem.状态  其实可修改一切能用.访问的内存中对象上的属性  ③自定义扩展属性:  $().attr(“属性名”[,”值”])  $().prop()不能访问自定义扩展属性   |  | | --- | | 同时修改多个属性:  $().attr/prop/css({  属性:值,  … : …  }) |   //单击图片，切换下一张 |
| 3.样式:  ①修改: elem.style.css属性  获取: getComputedStyle(elem)  $().css("css属性"[,值])  a.单个数值的属性值，不必加单位px  b.如果给了新属性值，就自动执行elem.style修改内联样式；如果没给新属性值，就自动执行getComputedStyle()获得计算后的属性值。  $().css({  Css属性:值, //长度不用加单位  … : … })  ②操作class:  $().addClass(“class1 class2 …”)  添加新class  $().removeClass(“class1…”)  移除现有的class  $().hasClass(“class1”)  判断是否包含指定class  $().toggleClass(“class”)  在有和没有”class”之间来回切换   |  | | --- | | 等效于: if(!$btn.hasClass("down"))  $btn.addClass("down");//添加down  else//否则，就移除down  $btn.removeClass("down"); | |

六．按节点间关系查找:

|  |
| --- |
| 1. 父子:  $().parent()  代替.parentNode  $().children(["selector"])  代替.children仅查找直接子元素  可筛选: 仅获得符合selector要求  的直接子元素  $().find(“selector”)  可在所有后代中查找符合selector  条件的元素  $().children(":first-child")  代替 .firstElementChild  $().children(":last-child")  代替.lastElementChild |
| 2. 兄弟  $().prev() 代替.previousElementSibling  $().prevAll() 之前所有  $().next() 代替.nextElementSibling  $().nextAll() 之后所有  $().siblings() 除自己之外所有兄弟 |

七．添加/删除/替换/克隆

|  |
| --- |
| 1.添加: 2步:  用HTML片段批量创建元素:  var $elem=$(“HTML片段”)  将新元素添加到DOM树:  $parent.append($elem)  代替了.appendChild  $elem.appendTo($parent)  return $elem .后续操作  $parent.prepend($elem)  将$elem插入到$parent子元素开头  $elem.prependTo($parent)  return $elem .后续操作  $child.before($elem)  代替parent.insertBefore(elem,child)  $elem.insertBefore$child  return $elem .后续操作  $child.after($elem)  将$elem插入到$child之后  $elem.insertAfter$child  return $elem .后续操作 |
| 2.删除: $elem.remove(); |
| 3.替换:  $child.replaceWith($elem)  代替parent.replaceChild(elem, child)  $elem.replaceAll($child)  return $elem .后续操作   |  | | --- | |  | |
| 4.克隆: var $elem2=$elem.clone();  //替换战机图片   |  | | --- | |  | |

八．事件绑定:

|  |
| --- |
| 1.普通绑定  ① **$().on("事件名",处理函数)**  代替 .addEventListener()  比如: $().on(“click”,function(){ … })  对于常用的21种事件, 可更简化:  **$().事件名(处理函数)**  比如: $().click(function(){ … })  ② **$().off(“事件名”, 处理函数)**  代替 .removeEventListener()  只触发一次，自动解绑  **$().one("事件名",function(){ … })** |
| 2.利用冒泡  $(parent)**.on**(“事件名”,”选择器”,function(){  //2个简化:  //1. this->e.target  //2. 不用写if，而是将判断条件选择器直接交给on作为参数，由on自动判断  })  //利用冒泡简化删除子元素盒子   |  | | --- | |  | |

**专题：页面加载后执行**:

|  |
| --- |
| 1. DOM内容加载后就提前触发:  DOMContentLoaded  仅需要等待DOM内容(HTML和js)加载完成就可提前执行  *link和href都是异步执行*    今后，只要和CSS、图片无关的js初始化操作(比如:事件绑定/添加/删除/修改/遍历元素内容)都应该在DOM内容加载后就提前执行  jQuery重新重新封装:  $(document).ready(function(){ … })  ->$().ready(function(){ … })  ->$(function(){ … })  *所以，几乎所有的jQuery代码都会包含在$(function(){…})* |
| 2. 整个页面加载后才触发:  等HTML,CSS,JS,图片加载完  今后，依赖于CSS和图片的代码，必须放在window.onload中  $(window).load(function(){//addEventListener}) |

**鄙视: jQuery中$的原理:**

|  |
| --- |
| $是jQuery类型的别名  使用$可以做4件事:  ①$(“选择器”) 查找并封装DOM元素为jQuery对象  优化: 如果查找条件简单时，默认用HTML查找  如果查找条件复杂时，首选用选择器查找  ②$(DOM元素) 直接将DOM元素封装为jQuery对象  ③$(“HTML片段”) 用HTML片段创建DOM元素  ④$(function(){…}) 绑定DOM内容加载后触发的事件处理函数 |

**鼠标事件:**

|  |  |  |
| --- | --- | --- |
| ①鼠标移入移出  mouseover mouseout  问题: 反复进出子元素，也会反复触发父元素上的处理函数  解决: 都用mouseenter和mouseleave代替，即使反复进出子元素也不会反复触发父元素上的处理函数  如果同时绑定了mouseenter和mouseleave，其实只需要绑定一个hover就可以:  $().hover(//mouseenter+mouseleave  function(){ ... },//给mouseenter  function(){ ... } //给mouseleave)  如果两个处理函数刚巧可以合并为一个处理函数，其实只写一个处理函数也行   |  | | --- | |  | |  |   ②模拟触发:  即使没有点击按钮，也能执行按钮的处理函数:  2种:  $elem.trigger(“事件名”)  可简写为: $elem.事件名() |

3. 动画:

|  |
| --- |
| 1. 简单动画: 三种写死的动画效果  1. 显示隐藏: $().show() $().hide() $().toggle()（推崇）  默认:不带参数，用display:block/none 实现的瞬间显示隐藏  想要动画效果，必须添加一个ms参数，表示持续时间  2. 上滑下滑: $().slideUp() $().slideDown() $().slideToggle()  3. 淡入淡出: $().fadeIn() $().fadeOut() $().fadeToggle()  问题: 1. 写死的动画效果，无法维护  2. 用js定时器程序实现的动画，效率远不如css高！ |
| 2. 万能动画: 对任意css属性都可应用动画效果  $().animate({  Css属性: 目标值,  … : …  //只支持单个数值的属性  //不支持颜色动画  （名:驼峰;值:不加单位）  },持续ms)  强调: 程序中的css属性名不能用-，都要去-变驼峰 |
| 3．排队和并发:  1. 排队: 多个css属性先后变化  对一个元素，先后调用多个动画API，多个动画API之间是排队执行  2. 并发: 多个css属性同时变化  放在一个animate中的多个css属性默认就是并发变化 |
| 4．停止动画: $().stop()  问题: 仅停止当前正在播放的动画，队列中后续动画依然会执行  解决: $().stop(true) 停止当前动画且清空动画队列  选择器: :animated 匹配或者判断正在播放动画的元素  动画播放后自动执行:  每个动画api都有最后一个回调函数参数，会在动画播放后自动执行 |
| 总结: 首选CSS: 效率高，可维护  和交互有关的动画，只能用JS定时器/JQUERY  和动态添加/删除DOM元素有关的动画，也必须用JS定时器/jQuery |

4. 类数组对象操作:

|  |  |  |
| --- | --- | --- |
| 2个API:  1. 遍历:  //实例方法  $()->jQuery()->new jQuery()  **$().each(function(i,elem){})**  //forEach(function(elem,i,arr))  //i 获得当前位置  //elem 获得当前元素对象  //静态方法  //$->jQuery  **$.each(类数组/数组,function(i,elem){ })**   |  | | --- | |  |   *三种遍历求和*   |  | | --- | |  | |
| 2. 查找:  $("所有元素").index("要找的元素")  在"所有元素"的jQuery对象中，查找"要找的元素"的位置  如果在同一个父元素下找子元素的位置:  $(要找的子元素).index() |

5. 添加自定义API:

|  |
| --- |
| 其实就是在jQuery的原型对象中添加自定义函数:  jQuery.fn=jQuery.prototype  定义新函数:  jQuery.fn.新函数=function(){ }  调用新函数:  $(...).新函数() |

**JQERYday04**

正课:

1. 插件:

2. ajax:

3. 跨域:

1. 插件: 也称组件

什么是: 拥有专属的HTML，CSS和js的独立页面区域

为什么: 重用！

何时: 只要一个功能/区域可能被反复使用时

如何:

3个来源:

1. 官方插件：jquery ui

依赖于jQuery, 必须先引入jQuery，再引入jQuery UI

如何:

1. 引入jQuery UI的css

2. 按组件的要求，自行编写HTML结构和内容

3. 引入jQuery.js, 再引入jQuery UI.js

4. 查找插件的父元素，调用jQuery UI的插件函数

$(插件的父元素).插件函数()

侵入性: 插件根据自己的需要，自动添加样式类或自定义扩展属性

2. 第三方插件：

3. 自定义插件：

定义:

前提: 已经用HTML，CSS，JS实现了插件的效果和功能

1. 将插件的CSS提取出来，保存在一个独立的css文件

要求: 为了避免和其他插件存在相同的样式类发生冲突，必须保证每个样式类都要以统一的插件类名作为前缀！

2. 定义插件的js: 向jQuery的原型对象中添加自定义插件函数

jQuery.fn.插件函数=function(){

//this->$(父元素)

//2件事:

//1. 侵入class:

//2. 绑定事件处理函数:

}//调用时: $(父元素).插件函数();

使用插件: 同使用jQueryUI的做法

1. 引入插件的css

2. 按照插件要求编写html

3. 引入jquery.js和插件.js

4. 查找插件父元素，调用插件函数

2. ajax:

$.ajax({

url:"服务端接口地址",

type:"get/post",

data:{ 参数1:值1, ... },

dataType:"json", //可自动将服务器返回的json字符串转为对象

success:function(res){//onreadystatechange 返回响应，且响应成功时自动触发

//res会自动获得服务端返回的数据

//用res执行DOM操作

}

})//jquery 3.x 支持Promise

.then(function(res){

})

3. 跨域:

什么是: 一个域名下网页，向另一个域名下发送请求，请求另一个域名下的资源

比如: 现在在http://localhost/index.html下

<script src="http://www.jquery.com/jquery.js"

<img src="http://tmooc.cn/stylesheet/img/logo.png"

<link rel="stylesheet"

href="http://v4.bootcss.com/bootstrap.css"

问题: ajax的xhr对象，禁止发送跨域请求

包括:

1. 一级域名不同: www.a.com -> www.b.com

2. 二级域名不同: oa.tedu.com -> hr.tedu.com

3. 端口不同: localhost:5500 -> localhost:3000

||

4. 协议不同: http:localhost -> https:localhost

80 443

5. 即使同一台机器: 域名 -> IP

localhost 127.0.0.1

如何发送异步跨域请求:

1. JSONP: JSON with Padding 填充式json

问题1: ajax不能发送跨域请求

解决1: 请<script>元素帮助发送请求

问题2: <script>发送请求，必须返回一条可执行的js语句

解决2: 修改服务端res.write()，其中，将要返回的数据，填充进一条可执行的js语句中，一起返回。

问题3: 服务端返回的js语句是写死的，众口难调

解决3: 在客户端定义一个处理函数

function show(res){

//对res执行任何想要的操作

}

服务端返回一条函数调用语句，函数名必须和客户端定义的函数名保持一致:

res.write(`show('${weather}')`)

返回: show('晴 -10~-2 from dong')

在客户端执行时:

调用show函数:

参数res自动得到了'晴 -10~-2 from dong'

实现了: 客户端操作与服务端的分离

问题4: 服务端将函数名规定死，也是众口难调

解决4: 客户端发送:

<script src="url?callback=客户端函数名"

服务端:

先获得req中callback变量中的函数名

再将函数名拼接到res.write()中，代替写死的函数名

实现了: 处理逻辑和函数名与服务端无关

问题5: <script>在客户端只写死一次，仅能在首次加载页面时执行一次，无法反复发送请求

解决5: 动态添加<script>元素

在单击事件中:

$('<script src="http://localhost:3000?callback=doit">').appendTo("body");

问题6: 新增的<script>无法自动删除，造成积压

解决6: 在自定义的回调函数结尾，查找最后一个script，删除。

其实: $.ajax可自动实现jsonp效果:

$.ajax({

url:"url",

... : ... ,

dataType:"jsonp", //使用jsonp方式请求服务端

success:function(res){ ... }

，

原理: 同以上6步:

1. 动态创建script元素发送请求

2. 自动为success匿名函数定义随机函数名拼接到url?callback=随机函数

3. success函数执行后，自动删除script元素

强调:jsonp单靠客户端无法实现，必须服务端负责拼接函数名和要返回的数据。所以必须客户端服务端同时修改才可支持。

4.服务端跨域：（不需要客户端做任何改变）

CORS：Cross

**VUEday01**

正课:

1. 原生 vs 函数库 vs 框架
2. 什么是Vue
3. Vue的原理
4. 绑定语法
5. 指令
6. 原生 vs 函数库 vs 框架

原生: DOM/BOM

浏览器自带的

繁琐

函数库: jQuery

第三方的

简单

仅是对每个步骤的API进行了简化

没有对流程进行简化，依然存在大量重复操作

框架: Vue/AngularJS/React

框架是半成品的项目

优点: 从流程上根本的简化了开发，消除了重复操作

1. 什么是Vue:

Vue是渐进式的基于MVVM的纯前端的js框架

渐进式: 可根据需要，选择部分组件逐渐应用

**使用：** 凡是以数据为主的项目都可用vue开发

**步骤：** 2种:

1. 下载vue.js到客户端，在网页中引入:

官网: cn.vuejs.org

最新的版本: 2.5

选择: 开发版: 包含完备的调试信息

生产版: 删除了调试信息

步骤:

1. <script src=”js/vue.js”>

2. 编写静态页面:

<div id=”app”>

… 在需要显示数据的位置用{{变量}}占位…

</div>

3. 在自定义脚本中:

<script>

var vm=new Vue({

el:”#app”,

data:{

页面需要的变量:值,

… : …

}

})

结果: data中的变量值，自动替换{{}}的位置

修改内存中data下的变量值，页面自动更新

1. 脚手架

3. Vue的原理:

旧的划分:

HTML: 定义网页内容——静态

CSS: 添加网页样式——静态

JS: 添加交互行为——动态的，过于强大，重复代码太多

MVVM模式: 对前端内容进行重新划分:

View视图/界面: HTML+CSS

Model模型数据: 在内存中定义的或Ajax请求回来的，要加载到页面的所有数据的统称

模型数据中的某一个值，称为模型变量！

ViewModel 框架: 自动将页面(View)和模型数据(Model)同步

1. 虚拟DOM树:

是基于原生DOM树生成的，仅包含可能变化的元素的简化版DOM树

**优点：** 仅包含可能变化的内容，极其便于快速遍历。

还封装了原生DOM操作，避免重复编码

仅修改发生变化的元素，效率高！

1. 响应系统:

当模型变量发生变化时，可自动通知虚拟DOM树，修改对应的真实DOM元素节点

如何通知: 自动给每个模型变量添加get()和set()访问器属性。每次修改模型变量时，都会触发set()，在set()中发送通知给DOM树。

4. 绑定语法:

{{模型变量}} 学名: 插值语法: Interpolation

外号: 大胡子

{{表达式}}

{{实例化对象}}

{{全局函数}}

{{对象的属性}}

{{数组的元素}}

{{三目}}

不能写程序结构：if else while for do while switch case

问题: 只能绑定内容，不能绑定属性和事件

5. 指令(directive)

**定义：** 增强HTML功能的特殊属性

**优点：** HTML本身是静态的，没有程序必须的元素: 变量、分支、循环

**使用：** 只要需要加强HTML的功能时

**步骤：** <ANY 指令名=”值”>

包括:

1. v-bind: 绑定属性:

**步骤：**<ANY v-bind:属性名=”模型变量/表达式”

可简写为: <ANY :属性名=”模型变量/表达式”

1. v-on: 绑定事件处理函数

**步骤：** <ANY v-on:事件名=”处理函数(参数值,…,$event)”>

可简写为:

<ANY @事件名=” 处理函数(参数值,…,$event)”

处理函数定义在:

new Vue({

el:”#app”,

data:{ … },

methods:{

处理函数(形参,…){

… …

}

}

})

获得事件对象e: 2步:

1. <ANY @事件名=”处理函数($event)”
2. methods:{

处理函数(e){

e就是DOM的事件对象

}

}

事件修饰符: 代替事件对象的API

1. 阻止默认行为: e.preventDefault()

<ANY @click.prevent=”处理函数()”

1. 取消冒泡: e.stopPropagation()

<ANY @click.stop=”处理函数()”

3. v-if: 根据条件控制元素的可见不可见

单独使用: 控制一个元素的可见不可见

**步骤：** <ANY v-if=”条件表达式”>

联合使用: v-if v-else-if v-else控制多个元素选其一显示

4. v-show: 根据条件控制元素的可见不可见

v-show是通过display:none隐藏——效率高

vs v-if 是通过修改DOM树(是否加载DOM节点)实现隐藏——效率低

但是，如果控制多个元素选其一显示时，首选v-if v-else-if v-else。。。

5. v-for: 反复生成多个相同结构的HTML片段

**步骤：** <parent>  
 <child v-for=”(value,i) in/of 数组”>

<sub>value和i可用于更子级元素的绑定

6. v-text 和 v-html:

代替{{}}绑定元素的内容:

如果绑定的是HTML片段: 用v-html

如果绑定的是纯文本内容: 用v-text

7. v-cloak: 在Vue对象加载完之前，临时隐藏受监视的元素

**步骤：** 2步:

1. 自定义选择器属性[v-cloak]{ display:none }
2. 在要隐藏的元素上，添加v-cloak属性:

<ANY v-cloak >

原理: 当Vue对象加载完，自动查找v-cloak属性移出

8. v-once: 仅在首次加载时，绑定一次，之后模型变量即使再变化，也不自动更新

原理: 一旦首次绑定结束，就从虚拟DOM中移除该元素

**步骤：** <ANY v-once>绑定语法

9. v-pre: 保护元素内容中的{{}}不被vue编译

**VUEday02**

正课:

1. 双向绑定:
2. 绑定class和style属性
3. 计算属性
4. 自定义指令——了解
5. 过滤器
6. Axios
7. 双向绑定:

问题: :value的方式只能将内存中模型变量的值绑定到页面，不能将页面的修改自动同步到内存中的模型变量上——单向绑定

解决: 双向绑定: 既能将内存中模型变量的值绑定到页面，又能将页面的修改自动同步到内存中的模型变量上。

**使用：** 只要绑定可修改的表单元素，都用v-model

**步骤：** v-model:value=”模型变量”

简写: v-model=”模型变量”

原理: 所有带v-model绑定的元素，都被加入一个监视队列(watch)，由一个死循环不断监视队列中元素的内容变化。只要发生变化，就直接修改绑定的模型变量

监视函数: 只要页面变化，就自动执行的函数

1. 绑定class和style
2. 可将class和style作为字符串绑定

问题: 拼接字符串很麻烦！

1. 用对象绑定class和style

<ANY :style=”模型变量” 模型变量是一个对象

new Vue({

el:

data:{

模型变量:{

Css属性:值, //不能省略单位

… : …

}

}

})

<ANY class=”写死的class” :class=”模型变量” 也是对象

结果: class和:class最终会合并为一个class=””

new Vue({

el:

data:{

模型变量:{

“class名”:true, 表示启用的class

“class名”:false, 表示不启用的class

}

}

})

3. 计算属性:

**定义：** 不实际存储属性值，要依靠其他属性，动态计算获得

优点: 只计算一次，结果可被vue缓存

vs methods: methods中的方法，每绑定一次，就重复执行一次，不会被缓存

**步骤：** 2步:

1. new Vue({

computed:{

计算属性名(){

return 计算后的值

}

}

})

1. 绑定与普通属性完全一样！

4. 自定义指令(directive):

**步骤：**

创建自定义指令:

Vue.directive(“指令名”,{ //不要加v-

inserted(el){//当当前元素被插入到DOM树上后立刻执行！

//el是纯DOM元素对象

//可用原生DOM API对el做任何事情

}

})

使用自定义指令:

<ANY v-指令名

5. 过滤器(filter):

**定义：** 对原始数据进行再加工之后，再绑定

**使用：** 如果原始数据不能直接使用

比如: 日期: ms ->yyyy/mm/dd

性别: 1/0 –> 男/女

状态: 数字: 10、20、30、40->未付款、已付款、已发货、已签收

**步骤：**

定义过滤器:

Vue.filter(‘过滤器名’,function(val,自定义参数){

return 过滤后的新值

})

使用过滤器:

:属性=”模型变量 | 过滤器” -> 实际绑定的结果是过滤器过滤之后的值

{{模型变量 | 过滤器(参数值) | 更多过滤器 … }}

6. axios:

**定义：** 支持Promise的专门发送http请求的函数库

**使用：** 只要在Vue中发送ajax请求，都用axios代替$.ajax

**优点：** 因为今后可能Vue不再包含jQuery

**步骤：**

先引入axios.min.js

Get: axios.get(“url”,{

params:{ 参数:值, 参数:值, … }

}).then(res=>{

//res.data中才是服务端返回的结果

})

Post: axios.post(“url”,”变量=值&变量=值…”).then(res=>{ … })

**VUEday03**

正课:

1. 组件:
2. 组件生命周期:
3. 组件化开发
4. 组件间传参
5. 路由
6. 组件:

**定义：** 拥有专属的HTML，CSS和数据的页面独立区域

**优点：** 1. 便于重用

2. 松散耦合

3. 分工协作

**使用：** 今后，只要页面上一个独立的区域，都必须封装为组件，今后每个页面都是由多个组件组成的。

**步骤：**

1. 创建Vue组件: 2步:
   1. 先在<template></template>中定义组件的HTML片段

要求<template>内只能有一个唯一的父元素包裹

* 1. 创建Vue组件

Vue.component(“组件名”,{

template:”选择器”, //找到页面上一个<template>元素，并监控其中的内容——组件的HTML片段

data(){ //每使用一次组件，就调用一次data函数，为当前组件生成一个新data对象

return {

模型变量

}

},

…同new Vue()

})

1. 使用组件: 一个组件，其实就是一个可重用的标签！组件名就是标签名

运行时: 用<template>中的HTML片段代替<组件名>标签的位置

2. 组件生命周期:

**定义：** 一个组件的加载过程

包括: 4个阶段:

1. Create创建：创建组件实例对象，并创建data对象
2. Mount挂载：将组件模板中的元素，绑定后，挂载到DOM树
3. Update：模型变量被更新
4. Destory：要销毁一个组件

生命周期钩子函数:

**定义：** 每个生命周期阶段抛出的事件处理函数

**使用：** 只要希望在某个生命周期阶段执行一项任务时，都要写在该阶段对应的构造函数中。

包括:

beforeCreate: 创建组件实例之前触发

created: 创建组件实例之后，但挂载组件到DOM树之前触发

new Vue($data:模型数据 $el:undefined)

已经可以发送ajax请求

暂时不能执行DOM操作！

beforeMount: 挂载组件到DOM树之前触发

mounted: 挂载组件到DOM树之后触发

new Vue($data:模型数据 $el:虚拟DOM树)

也可以发送ajax请求

可执行DOM操作！

beforeUpdate: 模型数据被更新前触发

updated: 模型数据更新后触发

beforeDestroy: 销毁组件对象之前触发

destroyed: 销毁组件后触发

3. 组件化开发:

组件: 页面中一个独立的，可重用的区域

如何进行组件化开发:

拿到页面后，先划分区域，每个区域都是拥有专有HTML，CSS，数据的独立组件

分为:

1. 根组件: new Vue()

通常一个页面，只有一个根组件

1. 全局组件: Vue.component()

**定义：** 可在页面中任何位置使用的组件

1. 局部组件: 也称子组件:

**定义：** 只能用于某个特定父组件内的组件

**步骤：** 2步:

1. 将作为子组件的全局组件，降级为普通的对象

强调: 组件对象名应该是将来的组件标签名改为驼峰命名

1. 在父组件中添加components属性:{

子组件对象, …

}

强调: Vue会将驼峰命名的子组件，转化为-命名的标签名.

4. 组件间传值:

父子:

1. 父->子:

问题: 父组件的模型变量，子组件无权直接使用

解决: “属性下行”

**步骤：** 2步:

1. 子: 定义属性:

子组件对象{

…

props:[“模型变量”,…]

}

说明: props中的模型变量:

对外: 可用作元素标签中的属性，让父组件可以通过绑定赋值。

对内: 等效于data{ … }

1. 父: 绑定属性，赋值:

<template id=”tpl父”>

<子组件 :模型变量=”父的模型变量”

2. 子->父: 事件上行

问题: 子组件无权直接删除父组件中的模型变量的内容

解决: 事件上行

**步骤：** 2步:

1. 父: <template>  
    <子组件 @自定义事件=”父的处理函数”

父组件对象={

…

methods:{

处理函数(形参){ 实际修改父组件的模型变量}

}

}

1. 子: 子组件对象={

…

methods:{

事件处理函数(){

This.$emit(“自定义事件”,要给父的值)

}

}

}

**VUEday04**

正课:

1. SPA
2. Vue-router
3. 脚手架
4. SPA: 单页面应用

**定义：** 整个应用程序只有一个完整的HTML页面

每个“页面”其实仅是一段HTML片段/组件

所谓的”页面跳转”，只是将HTML中特定区域替换为新组件的HTML片段

**优点：** 多页面 vs 单页面

|  |  |
| --- | --- |
| 多页面 | 单页面 |
| 项目中包含多个独立的完整HTML页面  缺: 每次跳转都向服务器重新请求页面，增加了请求的次数 | 整个项目只有一个完整的HTML页面，其它页面只是一段HTML片段/组件  优: 每次跳转不需要向服务器发送请求，减少了请求次数  缺: 不利于搜索引擎优化 |
| 页面跳转: 删除整棵DOM树，重新下载HTML，并重建整棵DOM树  缺: 效率低 | 页面跳转: 仅替换当前DOM树中的局部内容为新HTML片段，无需重建DOM树和等待下载  优: 效率高！ |
| 如果两个页面都要用到同一个资源，必须反复请求下载 | 如果两个“页面”都要用到同一个资源 |

1. Vue-router:

**定义：** vue中专门实现单页面应用的路由器组件

**使用：** 只要在vue中实现单页面应用，都用vue-router

**步骤：**

SPA应用的原理:

1. 先定义一个完整的HTML，其中包含多个<template>
2. 将<template>封装为子组件对象
3. 定义路由字典:

路由: 一个相对路径+一个组件

路由字典: 包含多对儿路径与组件对应关系的集合

**步骤：** var routes=[

{path:”/”,component:home}, //默认首页

{path:”/home”,component:home},

{path:”/login”,component:login},

{path:”/\*”,component:{template:”<h1>404:xxx</h1>”}}

];

1. 将路由字典装入新创建的路由器对象中:

var router=new VueRouter({ routes })

1. 将路由器对象加入new Vue中:

new Vue({ el:”#app”, router })

1. 在<div id=”app”>中添加<router-view>占位标签

运行时: 地址栏中: index.html#/相对路径

router可监控#/相对路径的变化，并自动替换<router-view>为指定的组件片段.

嵌套路由:

**定义：** 在一级路径下，又细分未多个二级路径

**使用：** 当多个”页面”/组件中包含部分相同的页面结构时

**步骤：**

1. 定义一个父级组件，包含相同部分的结构和一个<router-view>
2. 在路由列表中:

var routes=[

{path:”/一级路径”,component:父组件,children:[

{path:”/一级路径/二级路径”, component:子组件},

… …

]}

]

运行时:

当访问路径以一级路径开头时: 就先用父组件代替div#app中的<router-view>

当一级路径之后，跟上二级路径: 再用子组件代替父组件中的<router-view>

路由跳转: 2种:

1. 在HTML中写死:

<router-link to=”/login”>文本</router-link>

在运行时，会自动翻译为<a href=”#/login”>文本

1. 在js中用程序跳转:

this.$router.push(“/新地址”)

//Vue模拟history的方式进行跳转

路由传参: 2步:

1. 在路由列表中定义参数:

{path:”/路径/:参数名”,component:组件}

强调: 一旦定义参数，则必须带参才能访问当前路径

1. 在跳转时，传参:

在HTML中: <router-link to=”/路径/参数值”>

强调: 跳转时，不要加:

在程序中:

1. 在下一个组件的程序中获得参数值:

this.$route.params.参数名

问题: 在带参数的“页面”，仅修改地址栏中的参数值，不会重新绑定页面内容

解决: 添加watch监听地址栏$route:

watch:{//监控

"$route":{//当地址栏中的内容发生变化时

handler(to){//自动调用handler

//to: 新的$route对象

this.lid=to.params.lid;

},

immediate:true//立刻

}

}

3. 脚手架:

**定义：** 半成品项目，已经封装了大部分重复性劳动，只需要添加个性化功能即可。

**优点：** 1. 节约代码量; 2. 统一项目结构

**步骤：**

1. 安装脚手架命令行工具: vue-cli 用于在本地生成脚手架代码

npm i –g @vue/cli

1. 用vue-cli命令行工具在本地硬盘生成脚手架项目源代码

打开命令行cmd，进入想要创建项目的父级文件夹下:

执行: vue create 项目名/项目文件夹名

第一步: 提示: 使用默认npm下载会很慢，是否使用taobao镜像: Y或n都行

第二步: 提示: 使用默认配置，还是手动设置配置:   
 选: Manually select features

第三步: 选择组件: 按空格键选中或取消选中

选: Babel: 翻译工具: 将es6甚至更高版本的js翻译为es5的代码

Router: vue-router

Vuex: 下个老师讲

第四步: Use history mode for router:

默认模式: index.html#/相对路径

History模式: index.html/相对路径——需要服务器端配置重定向

选: n 不使用history

第五步: 将配置分别保存在多个配置文件中，还是集中保存在一个package.json中

选: In package.json

第六步: 是否将本次配置保存为今后项目的默认配置: 选 n

1. 安装完成后:

cd 项目目录

运行: npm run serve

提示: 编译成功

浏览: <http://localhost:8080>看到示例页面

脚手架的项目结构:

public: 保存不需要编译，就可直接浏览的静态资源:

比如：图片，再比如，公用的css，公用的js

同时还保存着整个项目唯一的完整的HTML页面

index.html

src: 保存着项目的所有源代码

assets: 专门用于保存，需要被打包的静态资源: css,图片或js

App.vue: 保存着根组件的<template>和根组件的css

main.js: 保存着new Vue()，监视着App.vue中的<div id=”app”>,以及对Vue的配置

运行时: App.vue和main.js 共同替换index.html中的<div id=”app”>

views: 专门保存“页面组件”的文件夹，将来有几个“页面”，views中，就有几个.vue文件

components: 专门保存各个页面中的子组件的文件夹

router.js: 专门定义路由器和路由列表的模块

每个.vue文件: 都是一个独立的组件，拥有专属的HTML，CSS，JS和数据

每个.vue文件中包含三部分:

<template>

<div>

组件的HTML片段  
 </div>

</template>

<script>

//引入子组件模块: @=”src/”

//相当于: var 组件对象名=require(“xxx.vue”)

Import 组件对象名 from “@/components/子组件.vue”

//将当前组件抛出为模块:

export default {

data(){

return { 模型变量 }

},

//同Vue中的Vue.component或new Vue()

//定义子组件

componets:{

子组件对象名

}

}

</script>

<style scoped>/\*仅限于当前组件范围内可用的样式\*/

当前组件的<template>专用的Css样式

.fade{ … }

</style>

安装axios: npm i –s axios

配置axios:

main.js中: import axios from 'axios'

Vue.prototype.axios=axios;

Xxx.vue中: 使用axios:

this.axios.get/post(…)