总结:

1. 只要验证字符串格式或查找、屏蔽敏感词时都要用正则

(1). 最简单的正则: 一个敏感词的原文

(2). 某一位字符上可能有多种备选字时用: [备选字列表]

(3). 如果[]中部分字符是连续的，可用: [x-x]

a. 一位小写字母: [a-z]

b. 一位大写字母: [A-Z]

c. 一位字母(大小写都行): [A-Za-z]

d. 一位字母或数字都行: [0-9A-Za-z]

e. 一位汉字: [\u4e00-\u9fa5]

(4). 预定义字符集:

a. \d 一位数字

b. \w 一位数字、字母或\_

c. \s 空格、tab、换行等空字符

d. . 任意字符

(5). 如果规定一个字符集或子规则反复出现的次数时就用量词:

a. 有明确数量边界的量词:

1). {n} =n 必须n个，不能多也不能少

2). {n,m} n个<= <=m个

3). {n,} n个<= 多了不限

b. 没有明确数量边界的量词:

1). \* 0个<= 可有可无，多了不限

2). ? 0个或1个 可有可无，最多一个

3). + 1个<= 至少一个，多个不限

(6). 两个规则中选其一匹配即可: 规则1|规则2

(7).希望将多个子规则分为一组先联合匹配，再和分组外的其他规则联合匹配:

(多个子规则)

(8). 匹配特殊位置: 3个

a. 字符串的开头位置: ^

b. 字符串的结尾位置: $

c. 英文句子中的单词的左右边界: \b

2. String家提供的正则相关函数: 3件事

(1). 查找敏感词: 4种情况

a. 查找一个固定的敏感词出现的位置:   
 var i=str.indexOf("敏感词")

//如果找不到，返回-1

b. 用正则查找多种敏感词出现的位置:   
 var i=str.search(/正则/i)

//如果找不到，返回-1

c. 查找敏感词的内容:

1). 查找第一个敏感词的内容和位置:   
 var arr=str.match(/正则/i)

//arr: [ 0:"敏感词内容", index:敏感词位置 ]

//如果找不到返回null

2). 查找所有敏感词的内容，不关心位置:   
 var arr=str.match(/正则/ig)

//arr: [ 敏感词1, 敏感词2, ... ]

//如果找不到返回null

d. 查找每个敏感词的内容和位置: reg.exec

|  |
| --- |
| 补: js中所有数组底层本质都是关联数组(下标都为字符串)  1. 访问数组中元素值的标注写法:  arr["下标"]  2. 简写:  a. 如果下标为自定义字符串名称，可简写为:  arr.自定义名称的下标  b. 如果下标为数字内容的字符串，可简写为:  arr[数字下标] |

|  |
| --- |
| 总结: 查找方法的返回值规律  1. 如果原函数返回的是下标位置i，如果找不到，都返回-1  2. 如果原函数返回的是一个数组arr或一个对象obj，如果找不到，都返回null  3. （未完...待续...） |

(2). 替换敏感词: 2种

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

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

return 根据本次敏感词动态生成一个新值

})

c. 删除敏感词:   
 变量=str.replace(/正则/ig, "")

(3). 切割字符串:

a. 简单切割:   
 var arr=str.split("切割符")

b. 复杂切割:   
 var arr=str.split(/正则/i)

c. 打散字符串为字符数组:   
 var arr=str.split("")

3. RegExp对象:

(1). 创建正则表达式对象:

a. 如果正则是固定的:   
 var reg=/正则/ig

b. 如果正则需要动态生成:   
 var reg=new RegExp("正则",ig)

(2). 验证字符串格式:   
 var bool=reg.test(str)   
 reg必须同时前加^后加$

(3). 既查找每个关键词的内容又查找每个关键词的位置: (待续)

do{

var arr=reg.exec(str);

if(arr!=null){

获得本次找到的敏感词的内容(arr[0])和位置(arr.index)

}

}while(arr!=null);

4. 函数:

(1). 创建函数三种方式:

a. function 函数名(形参列表){ 函数体; return 返回值 } //会被声明提前，不好

b. var 函数名=function(形参列表){ 函数体; return 返回值 }//不会被声明提前，首选

c. var 函数名=new Function("形参1", "形参2", ... , "函数体; return 返回值")

|  |
| --- |
| 函数本质:  1). 函数也是一个对象，对象中保存着函数的函数体代码  2). 函数名只是一个普通的变量，函数名通过函数对象地址，引用着函数对象  3). function在底层等效于new Function()  function 函数名(){ ... }和var 函数名=function(){}在底层都会被翻译为  var 函数名=new Function(...)  只不过function 函数名(){}是先提前，再翻译  而var 函数名=function(){}是不提前，原地翻译 |

(2). 重载: 今后，一件事，根据传入不同的参数值，动态执行不同的逻辑时，都用重载

function 一个函数名(不写形参变量){

//arguments对象自动接住所有实参值

if(arguments.length==0){

执行一种逻辑

}else if(arguments.length==1){

执行另一种逻辑

}else{

执行其它逻辑

}

}

其中arguments是类数组对象: 和数组相比:

a. 相同点: 也有下标，length属性，也可for循环遍历

b. 不同点: 不是数组类型，无法使用数组家的函数

(3). 匿名函数:

a. 所有回调函数优先使用匿名函数——用完释放，节约内存

b. 所有js代码都应该保存在匿名函数自调中，禁止使用全局变量，避免全局污染！

(function(){

要执行的js代码

})();

结果: 匿名函数内的都是局部变量，不会产生全局变量。

局部变量随匿名函数一起释放。不会污染全局。

(4). 作用域和作用域链: (跟着视频亲自画图！！！)

a. 作用域:

1). 全局作用域:window，保存全局变量

优: 可重用，缺: 随处可用, 极易被污染

2). 函数作用域: 保存局部变量

局部变量包括2中: 函数中var出的变量和形参变量

优: 仅函数内可用，不会被污染，缺: 不可重用

3). 函数作用域对象原理:

i. 每个函数定义时都自带好友列表，好友列表里2个格子，一个是空，一个引用window

ii. 调用函数时临时创建函数作用域对象保存函数局部变量。并将函数作用域对象的地址保存到函数好友列表中离自己近的格子里。

iii. 函数执行过程中按就近原则先在自己的函数作用域对象中找局部变量使用。如果找不到，才被迫去全局window中找变量使用.

iv. 函数调用后，好友列表中离自己近的格子清空，导致函数作用域对象以及内部的局部变量被释放！——所以局部变量不可重用！

b. 作用域链: 保存一个函数所有可用的作用域对象的链式结构(好友列表)学名就叫作用域链。  
 1). 作用域链保存着一个函数可用的所有变量

2). 作用域链控制着变量的使用顺序。先局部后全局。

(5). 闭包: （跟着视频亲自画图!!!）  
 a. 只要希望给一个函数保护一个可反复使用的专属变量，又防止这个变量被外界篡改时，都用闭包。

b. 闭包三步:

1). 用外层函数妈妈包裹要保护的变量和内层函数

2). 外层函数妈妈用return把内层函数孩子返回到外部

3). 外部想使用内层函数的人，必须调用外层函数，才能获得return出来的内层函数对象。并将内层函数保存在一个变量中反复使用。

c. 闭包形成的原因: 外层函数调用后，外层函数的作用域对象被内层函数引用着无法释放，形成了闭包对象

d. 闭包的缺点: 闭包比一般的函数占用多一块内存——外层函数的函数作用域对象。  
 所以，用完闭包后，应该尽快释放:   
 保存内层函数的变量=null

5. 面向对象: 封装 继承 多态

自己跟着视频，一步一步画图，自己标顺序，知识才能变成自己的

(1). 封装: 3种:

a. 用{}创建一个对象:

var 对象名={

属性名:属性值,

... : ... ,

方法名: function(){

... this.属性名 ...

}

}

b. 用new Object():

1). 2步:

i. var 对象名=new Object()

ii. 对象名.属性名=属性值;

对象名.方法名=function(){ ... }

2). 对象底层也是关联数组:

i. 都是名值对儿的集合

ii. 都可用[""]和.方式访问成员。

如果属性名来自于变量，就只能用[]，不要加""

iii. 访问不存在的属性，都不报错，返回undefined

判断是否包含某个属性:

对象.属性名!==undefined

iv. 强行给不存在的属性赋值，都不报错，而是自动添加该属性

给对象添加新属性，唯一办法，强行赋值:

对象名.新属性名=新值

v. 都可用for in遍历

c. 只要反复创建多个相同结构的对象都用构造函数:

1). 2步:

i. 定义构造函数:

function 类型名(形参1,形参2, ...){

this.属性名1=形参1;

this.属性名2=形参2;

//构造函数中不要再包含方法定义定义!

}

ii. 用new 调用构造函数:

var 对象名=new 类型名(属性值1, 属性值2,...)

2). new做了4件事:

i. 创建一个新的空对象

ii. 让新对象继承(\_ \_proto\_ \_)构造函数的原型对象

iii. 调用构造函数，传入实参，并自动替换构造函数中的this为new正在创建的新对象。构造函数中，通过强行赋值的方式为新对象添加规定的属性，并保存属性值。

iv. 返回新对象的地址，保存到=左边的变量中。

3). 优点: 重用对象结构代码

4). 缺点: 如果构造函数中包含方法定义，则每次创建新对象都会重复创建相同方法的副本。——浪费内存！

(2). 继承:

a. 今后，只要同一类型所有子对象共用的方法和属性值，都要集中保存在构造函数的原型对象中！

构造函数.prototype.属性名/共有方法名=属性值/function(){ ... }

b. 自有属性和共有属性:

1). 获取属性值:都可用"子对象.属性名"

2). 修改属性值:

i. 自有属性: 子对象.自有属性名=新值

ii. 共有属性: 构造函数.prototype.共有属性名=新值

c. 内置类型原型对象:

1). 11种内置类型/对象: String, Number, Boolean, Array, Date, RegExp, Math(对象), Error, Function, Object, global(对象)

2). 一种类型=构造函数+原型对象

i. 构造函数: 创建子对象

ii. 原型对象: 为所有子对象保存共有成员

3). 查看该类型共有哪些API: 类型名.prototype

4). 该类型缺少想用的方法: 类型名.prototype.共有新方法=function(){ ... }

d. 原型链: 保存着一个对象可用的所有属性和方法。控制着属性和方法的使用顺序：先自有再共有——就近原则！

(3). 多态: 重点讲重写：如果子对象觉得从父对象继承来的成员不好用，可以在子对象自己内部重写和父对象同名的成员，覆盖父对象的成员，优先使用自己的。

\*\*\*\*\*面向对象终极总结: 封装，继承，多态

①封装: 创建对象，2种:

如果只创建一个对象: {}

如果反复创建多个相同结构的对象: 构造函数

②继承: 所有子对象共用的属性值和方法，都要放在构造函数的原型对象中

③多态: 重写: 只要觉得从父对象继承来的成员不要用，都在子对象中重写同名成员

④如果觉得这个父对象对象都不好用，可以自定义继承: 2种:

1). 只换一个子对象的父对象: 2种:

i. 子对象.\_\_proto\_\_=新父对象

ii. Object.setPrototypeOf(子对象, 新父对象)

2). 更换多个子对象的原型对象: 构造函数.prototype=新对象

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

6. 严格模式: "use strict";

(1). 禁止给未声明过的变量赋值

(2). 静默失败升级为错误

(3). 普通函数调用中的this不指window，而是指undefined

(4). 禁用arguments.callee

|  |
| --- |
| 总结: this 判断this时，一定不要看他定义在哪儿。必须看它在哪里以何种方式调用 5种:  1. obj.fun() this->点前的obj对象  2. fun() this->默认指window  3. new Fun() this->new正在创建的新对象  4. 类型名.prototype.共有方法=function(){ ... }  this->将来谁调用这个函数，就指谁  将来调用这个函数的.前的某个子对象 5. 访问器属性中的this，指访问器属性所在的当前对象 |

7. 保护对象:

(1). 保护属性:

a. 每个属性包含三个开关:

1). writable: 控制是否可修改属性值

2). enumerable: 控制着是否可被for in遍历到，但是只防for in不防.

3). configurable: 控制

i. 是否可删除当前属性

ii. 是否可修改writable和enumerable两个开关

强调: configurable一旦改为 false，不可逆！

b. 只修改一个属性的多个开关:

Object.defineProperty(对象名, "属性名",{开关: true/false})

c. 修改多个属性的多个开关:

Object.defineProperties(对象名,{

属性名:{ 开关:true/false, ... },

... : ...

})

d. 如果用自定义的规则保护属性时，只能用访问器属性: 2步:

Object.defineProperties(对象,{

//1). 先定义一个隐姓埋名且半隐藏的数据属性:

\_属性名:{

value: 属性的初始值,

writable:true,

enumerable:false,

configurable:false

},

//2). 再定义访问器属性保镖冒名顶替要保护的属性

属性名:{

get:function(){

return this.\_属性名

},

set:function(value){ //value ← 要修改的新属性值

先验证value

如果验证通过，this.\_属性名=value

否则如果验证未通过，不但不保存新属性值，还会报错

},

enumerable:true,

configurable:false

}

})

外界使用访问器属性时和使用普通属性一样:

对象.属性名

外界试图获取访问器属性值时，自动调用get()

外界试图修改访问器属性值时，自动调用set()

(2). 保护结构: 3个级别

a. 防扩展: Object.preventExtensions(对象)

b. 密封: Object.seal(对象)

c. 冻结: Object.freeze(对象)

8. 如果没有构造函数，也想创建子对象，继承父对象:

var 新子对象=Object.create(父对象,{

自有属性:{

value:属性值,

开关:true或false,

... :...

},

... : { ... }

})

9. 替换this: 3种:

(1). 在一次调用函数时，临时替换this，首选:   
 函数.call(对象, 实参值,...)

(2). 临时替换一次this，但是需要打散数组再传参时，被迫改为:

函数.apply(对象, 数组)

(3). 创建一个一模一样的新函数并永久绑定this和部分实参值:

var 新函数名=原函数.bind(对象, 固定实参值, ...)

10. 数组函数:

(1). 判断:

a. 判断数组中是否所有元素都符合要求:

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

return 判断条件

})

b. 判断数组中是否包含符合要求的元素:

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

return 判断条件

})

(2). 遍历:

a. 单纯简化for循环变量原数组中每个元素:

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

对当前元素执行操作

})

b. 保护原数组不变，返回遍历加工后的新数组

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

return 加工后的一个新元素值

})

(3). 过滤: 复制出数组中符合要求的元素放入新数组返回

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

return 判断条件

})

(4). 汇总: 遍历数组中每个元素，经过求和或其他汇总方式，统计出一个最终结论

var 结果=arr.reduce(function(box,value,i,arr){

return box和value计算出的新临时汇总值

}, 起始值)

11. ES6:

(1). 模板字符串: 今后，只要拼接字符串，都用模板字符串代替+:

a. 整个字符串包裹在一对儿反引号`...`中

b. 反引号``中支持换行、""、''均可使用

c. 反引号中需要动态生成的内容必须放在${}里

d. ${}里:

1). 可以放一切有返回值的合法的变量或js表达式。

2). 不能放程序结构(分支和循环)以及没有返回值的js表达式

(2). let: 今后，声明变量都用let代替var

a. let的好处:

1). 阻止声明提前

2). 让代码块(分支和循环的{})也变成块级作用域，{}块内的变量出了{}无法使用，不会影响外部

b. let的小脾气:

1). 在同一作用域内禁止重复声明;

2). 禁止提前使用;

3). 在全局声明也不保存在window中

c. let底层相当于匿名函数自调

(3). 箭头函数: 今后，几乎所有的function都可用箭头函数简写:

a. 如何: 3句话:

1). 去掉function，在()和{}之间加=>

2). 如果只有一个形参，可省略()

3). 如果函数体只有一句话，可省略{}

如果仅有的一句话还是return，必须省略return

b. 特点: 箭头函数内部this与外部this保持一致。

c. 今后:

1). 如果函数中没有this或者恰好希望函数内this与函数外this保持一致时，可用箭头函数简写！

2). 如果不希望内外this相同时不能使用箭头函数简写。

(4). for of: 今后只要遍历数字下标的索引数组、类数组对象和字符串，都用for of

for in: 今后只要遍历自定义下标名称的对象和关联数组，都用for in

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

(5). 参数增强:

a. 参数默认值: 定义函数时最后一个形参不确定有没有实参时

function 函数名(形参1, ..., 最后形参=默认值){

... ...

}

b. 剩余参数: 定义函数时: 只要多个形参不确定，都用剩余参数

function 函数名(其它形参, ...数组名){

//...收集除其它形参外多余的实参值保存到指定数组名的数组中

}

c. 打散数组: 今后调用函数时，只要单纯打散数组再传参时

1). 如何: 函数(...数组)

2). ...口诀: 定义函数时...表示收集, 调用函数时...表示打散

3). ...语法糖:

i. 复制一个数组: var arr2=[...arr1];

ii. 合并多个数组和元素值: var arr3=[...arr1,值,...arr2,值]

iii. 复制一个对象: var obj2={ ... obj1 }

iv. 合并多个对象和属性: var obj3={ ...obj1, 属性:值, ...obj2, 属性:值 }

(6). 解构:

a. 只要想提取出数组中个别元素值，单独使用时，就数组解构:

[变量1, 变量2]=arr;

b. 只要想提取出对象中个别属性值或方法，单独使用时，就对象解构:

var {属性名也是变量名, 属性名也是变量名}=对象

c. 只要多个参数不确定有没有，又要求实参值必须传给指定的形参时，就参数结构:

定义函数时:

function 函数名({

属性名也是形参名="默认值",

属性名也是形参名="默认值",

... = ...

}){

函数体

}

调用函数时:

函数名({

要修改的形参名: 实参值，

... : ... ,

})

(7). class extends

a. 定义class:

class 类型名{

constructor(形参列表){

this.属性名=形参;

... = ...;

}

共有方法(){

... this.属性 ...

}

}

class的底层原理和用法与旧js构造函数用法完全相同！——新瓶装旧酒

b. 继承: 只要两种类型间包含部分相同的属性结构和方法定义，都要额外定义一个父类型集中保存两种类型相同的属性结构和方法定义。然后再让子类型继承父类型

class 子类型 extends 父类型{

constructor(...){

super(...);

this.属性名=形参;

}

子类型共有方法(){

...

}

}

两步,两个关键字: extends是继承的意思，super()是调用父类型构造函数，请父类型构造函数帮忙添加相同部分的属性的意思。

(8). Promise: 当多个异步任务必须顺序执行时，就可用promise

a. 前一项任务:

function 前一项任务(){

return new Promise(

function(door){

原异步任务;

原异步任务最后一句话之后door()

}

)

}

b. 调用前一项任务时:

前一项任务().then(下一项任务)

c. 如果下一项任务也返回new Promise对象，则可以继续用.then()相连

前一项任务().the(下一项任务).then(之后任务).then(...)...

d. 前后两个任务间传值:

1). 前一项任务内:

door(一个变量);

//只能传一个变量

//如果传多个值，可放在数组或对象中

2). 后一项任务声明时:

function 后一项任务(形参){ ... }

//前一项任务:door()中传什么

//后一项任务:形参就接住什么