第三阶段内容预告:

1. JS高级（8天）: JS语言高级的功能和核心的原理——基本功和内功心法——字、词、成语

(强调：不要想着做功能！)

2. DOM（4天）: 专门实现网页交互功能的技术——才可以考虑页面上的功能如何实现

问题: 步骤繁琐

优点: 几乎是万能的！

3. jQuery（4天）: 简化版的DOM——根据你所在城市的需求决定jquery学到什么程度

问题: 仅PC端——用的越来越少了——但是，很多城市很多公司，还在使用jquery。

4. VUE（5天）: 前端开发的终极简化

优点: 极其简单

缺点: 你看不到它的执行过程！

1. 微信:

第一个字: 微 wei w (微|w(ei)?)

中间: 有或没有空格 也可能多个空格 \s\*

第二个字: 信 xin x (信|x(in)?)

2. 电子邮件:

zhangdong@tedu.cn

@前: 一个以上的字符数字或\_都行 \w+

@

@后 第一个.前: 一个以上的字母数字或\_都行，也可以有- [\w-]+

.

.后: 一个以上的字母数字或\_都行 \w+

.cn: 可有可无，最多一次

\w+@[\w-]+\.\w+(\.cn)?

总结: 正则

1. 最简单的正则: 敏感词的原文

2. 如果规则中某一位字符上有多种备选字时: [备选字的列表] —— 字符集

3. 如果[]中部分字符是连续的: [第一个字-第二个字]

一位汉字: [\u4e00-\u9fa5]  
4. 四种最简化的预定义字符集:

一位数字: \d

一位字母、数字或\_都行: \w

一位空字符: \s

一位任意字符都行: .

5. 定义字符集出现的次数: 量词

(1). 有明确数量边界的:

a. 必须n个，不能多，不能少: {n}

b. 至少n个，最多m个: {n,m}

c. 至少n个，多了不限: {n,}

(2). 没有明确数量边界的:

a. 可有可无，多了不限: \*

b. 可有可无，最多一个: ?

c. 至少一个，多了不限: +

6. 两个子规则中选其一匹配即可: 规则1 | 规则2

7. 希望部分子规则临时组成一个整体先匹配，再和其他规则联合匹配: （） 分组

总结: String类型提供的正则相关函数: 3件事

1. 查找敏感词: 4种情况:

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

(2). 用正则模糊查找多个敏感词的位置: var i=str.search(/正则/i)

(3). 获取敏感词的内容:

a. 只获取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:敏感词的位置 ]

b. 查找所有敏感词的内容，但是无法获得位置:

var arr=str.match(/正则/ig);

arr: [ "敏感词1", "敏感词2", ... ]

(4). 既获取每个敏感词的内容，又获取每个敏感词的位置

reg.exec()

|  |
| --- |
| 总结: 返回值:  1. 如果原函数返回下标i，则找不到返回: -1  2. 如果原函数返回数组，则找不到返回: null |

2. 替换敏感词: 2种:

(1). 所有敏感词都替换为统一的新值:

str=str.replace(/正则/ig, "新值")

(2). 高级替换:

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

return 根据keyword不同，动态选择不同的新值返回

})

(3). 删除敏感词: 将敏感词替换为空字符窜

str=str.replace(/正则/g, "")

3. 切割字符串: 2种:

(1). 简单切割: 切割符是固定的:

var arr=str.split("切割符")

固定套路: 打散字符串为字符的数组: var arr=str.split("")

(2). 复杂切割: 切割符不是固定的，有多种

var arr=str.split(/正则/) 不加g也能找所有

2. RegExp对象:

(1). 创建: 2种:

a. 如果正则表达式固定不变: var reg=/正则/ig

b. 如果正则表达式需要动态拼接生成: var reg=new RegExp("正则", "ig")

(2). 2个函数:

a. 验证一个字符串的格式是否正确: var bool=reg.test(str)

//reg中必须同时前加^后加$

b. 既查找每个敏感词的内容，又查找每个敏感词的位置

do{

var arr=reg.exec(str);

if(arr!=null){

... ...

}

}while(arr!=null);

3. Function:

(1). 定义函数时: 3种方式:

a. 用声明方式:

function 函数名(形参变量列表){

函数体;

return 返回值;

}

问题: 会被声明提前

b. 用赋值方式:

var函数名=function (形参变量列表){

函数体;

return 返回值;

}

优点: 函数定义部分不会声明提前

c. 用new来创建:

var 函数名=new Function(... ...)

其实: 抛开声明提前不谈，三种方式创建的函数，其实创建出的函数，以及调用方式，完全相同的！没有任何差别！

本质: function 函数名(){ ... } (除了会被声明提前之外)

等效于 var 函数名=function(){ ... }

等效于 var 函数名=new Function(){ ... }

(2)重载:

a. 何时: 当一件事需要根据传入不同参数，执行不同的逻辑

b. 如何: 3步

1). 只定义一个函数，不要定义形参变量列表

2). 在函数内部，用arguments类数组对象接住将来所有传入的实参值列表

3). 在函数内部，判断arguments中的实参值个数(length)，或判断arguments中的内容，根据不同的实参值情况，动态选择执行不同的逻辑

3. 匿名函数:

(1). 何时: 回调函数 以及 匿名函数自调

(2). 匿名函数自调:

a. 今后几乎所有js代码都要放在匿名函数自调中。避免使用全局变量

b. 如何:

1). 标准写法:

var 返回值=(function(形参列表){

函数体;

return 返回值;

})(实参值列表);

2). 杀马特写法:

+function(){ ... }()

!function(){ ... }{}

... ...

4. 作用域和作用域链:

(1). Js中只有两种作用域:

a. 全局作用域对象window:

1). 自动保存所有全局变量

2). 在程序的任何位置都可访问

b. 函数作用域对象:

1). 只在调用函数时创建，调用后自动释放

2). 函数作用域对象中专门保存函数的局部变量: 2种:

i. 形参

ii. 函数内var出的变量

c. js中和其它语言的差别: 没有块级作用域:

1). 什么是块作用域: 分支结构(if else if else)或循环结构(for while do while switch)的{}，在其它编程语言中称为块级作用域

2). 在其它编程语言中，分支结构或循环结构{}内声明的变量，出了{}，不能使用！

3). Js中，因为没有块级作用域，所以，在分支结构或循环结构{}中声明的变量，出了{}，依然可以使用

4). 示例: 有没有块作用域情况下，程序不同的写法

|  |
| --- |
| <script>      //js中:      //如果随机生成一个数是偶数      if(parseInt(Math.random()\*10)%2==0){        var result="偶数"      }else{        var result="奇数"      }      console.log(result);      //其他有块级作用域的语言中:      var result="";      if(parseInt(Math.random()\*10)%2==0){        result="偶数"      }else{        result="奇数"      }      console.log(result);      //js中:      //求1~100的所有数字的和      for(var i=1,sum=0;i<=100;i++){        sum+=i;      }      console.log(sum);      //在其他有块作用域的语言中:      var sum=0;      for(var i=1;i<=100;i++){        sum+=i;      }      console.log(sum);    </script> |

(2). 作用域链:

a. 每个函数对象上其实都保存着一个作用域链列表

b. 普通函数作用域链中都有两个格子:

1). 离自己近的格子，不调用函数时，暂时为空，当调用函数时，会引用临时创建的函数作用域对象

2). 离自己远的格子始终保存window

c. 作用域链保存着当前函数所有可用的变量

d. 作用域链控制着变量的使用顺序: 先局部，后全局

5. 闭包:

(1). 什么是闭包: 外层函数调用后，外层函数的作用域对象，被内层函数引用着无法释放，形成了闭包对象.

(2). 为什么: 全局变量和局部变量都有不可兼得的优缺点:

a. 全局变量: 优点: 可重用; 缺点: 极易被污染——今后极力避免使用全局变量

b. 局部变量: 优点: 不会被外部篡改; 缺点: 不可重用

(3). 何时: 希望既可重用一个变量，又包含该变量不会被随意篡改时

其实是想给一个函数或一组函数保存一个专属的变量

(4). 如何: 3步:

a. 用外层函数包裹要保护的变量和要使用变量的内层函数

(了解: 如果内层函数中没有用到外层函数的局部变量，则不会形成闭包

比如: function outer(){

var i=0;

function fun(){

console.log(a)

}

}

)

b. 外层函数将内层函数对象返回到外层函数外部

c. 想使用内层函数的人应该调用外层函数获得内层函数对象，并保存在变量中，可反复使用

(5). 结果: 外层函数的局部变量，只归内层函数专属，其他人无法修改受保护的变量。

(6). 一句话概括闭包如何形成: 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。

(7). 闭包缺点: 比一般的函数，多占用一块内存——外层函数的作用域对象

(8). 解决: 将保存内层函数的变量=null

结果: 释放了内层函数，导致内层函数引用的外层函数作用域对象被释放

(9). 鄙视时: 画简图:

a. 找外层函数和内层函数之间的局部变量。——要保护的变量

b. 找外层函数妈妈共向外生出了几个内层函数孩子: 在外层函数内3种途径

1). return function(){ ... }

2). 全局变量=function(){ ... }

3). return arr或obj，但是arr或obj中，可能包含函数成员

c. 结果:

1). 妈妈一次生出的多个孩子，共享同一个闭包变量，一个函数修改了闭包变量，则另一个函数也受影响

2). 妈妈分两次生的不同的孩子和闭包变量之间，互不影响，毫无关系。

2. 面向对象:

(1). 封装:

a. 当只想创建一个新对象，且已经提前知道新对象中所有成员时，可用{}

var 对象名={

属性名: 属性值,

... : ... ,

方法名: function(){

... this.属性名

}

}

b. 如果只想创建一个新对象，但是暂时不知道新对象的成员，后续知道了成员，再补充时，用2步:

1). 先创建空对象: var 对象名={} //其实就是new Object()的简写

2). 在强行为新对象添加新属性:

i. 对象名.属性名=属性值

ii. 对象名.方法名=function(){

... this.属性名 ...

}

c. 如果希望反复创建多个相同结构的对象时，用构造函数: 2步

1). 先定义构造函数:

function 类型名(形参变量列表){

this.属性名=属性值;

... = ... ;

//构造函数中不要再包含方法定义

this.方法名=function(){

... this.属性名 ...

}

//所有方法都集中添加到原型对象中

}

2). 用new 调用构造函数:

var 对象名=new 类型名(实参值列表)

(2). 继承: 所有子对象共用的方法，集中定义在构造函数的原型对象中:

构造函数.prototype.共有方法=function(){

... ...

}

|  |
| --- |
| 今天的this: 4种: 判断this，不要看定义在哪儿！！！只看被谁调用！！  1. obj.fun() fun中的this->.前的obj对象  2. new Fun() Fun()中的this->new正在创建的新对象  3. 构造函数.prototype.方法=function(){ ... this ... } 原型对象方法中的this->将来调用这个方法的.前的某个子对象  4. 匿名函数自调或普通函数调用fun() 在没有.没有new时，this->默认指window  比如:  <script>      (function(){ console.log(this) })();      function fun(){        console.log(this)      }      fun();    </script>  输出结果:  window  window |

总结: 面向对象: 三大特点: 封装 继承 多态

1. 封装: 创建对象 3种方法:

(1). 只创建一个对象，且已经知道对象的成员是什么:

var 对象名={

属性名: 属性值,

... : ... ,

方法名: function(){

... this.属性名 ...

}

}

(2). 只创建一个对象，但是暂时不知道对象的成员: 2步

a. 先创建一个空对象等着: var 对象名={} //new Object()的简写

b. 等知道对象的成员之后:

对象.属性名=属性值

对象.方法=function(){ ... this.属性名 ...}

(3). 想反复创建多个相同结构的对象时: 2步:

a. 先定义构造函数:

//因为构造函数是为了描述同一类型的多个对象统一的属性结构，所以，构造函数名通常是一种类型的名称，比如学生，商品，订单，用户...

function 类型名(形参变量列表){

this.属性名=形参变量;

... = ...

//今后构造函数中不要包含方法定义

}

b. 用new调用构造函数创建对象

var 对象名=new 类型名(属性值列表);

|  |
| --- |
| new 做了4件事:  1. 创建一个新的空对象  2. 让子对象继承构造函数的原型对象（自动设置子对象的\_ \_proto\_ \_指向构造函数的prototype对象）  3. 调用构造函数，同时将构造函数中的this临时替换为new正在创建的这个新对象  结果构造函数中的每一句话，都会强行给新对象添加构造函数早就规定好的统一的新属性  4. 返回新对象地址，保存到等号左边的变量中 |

2. 继承:

(1). (自动)每定义一个构造函数，都会自动附赠一个公共的原型对象(prototype)

(2). (自动)当用new创建子对象时，第二步new会自动设置子对象的\_ \_proto\_ \_指向构造函数的原型对象(prototype)。结果，子对象继承父对象

(3). (自动)从此，父对象中的成员，子对象无需重复创建，就可直接使用

(4). (需要我们做的) 向原型对象中为所有子对象添加共有的方法

构造函数.prototype.共有方法=function(){

... this.属性名 ...

}

3. 多态:

重写: 如果从父对象继承来的个别成员不好用！就可在子对象中定义同名成员。结果，子对象再使用同名成员时，优先使用自己自有的同名成员。不再使用父对象不好用的同名成员。

4. 自定义继承: 如果子对象觉得整个父对象都不好用，可认别的父对象当爹

(1). 只修改一个子对象的父对象:

a. 不推荐: 子对象.\_\_proto\_\_=新父对象

b. 推荐: Object.setPrototypeOf(子对象, 新父对象)

(2). 如果更换该类型下所有子对象的父对象

a. 构造函数.prototype=新父对象

b. 时机: 在创建子对象之前，就要替换！

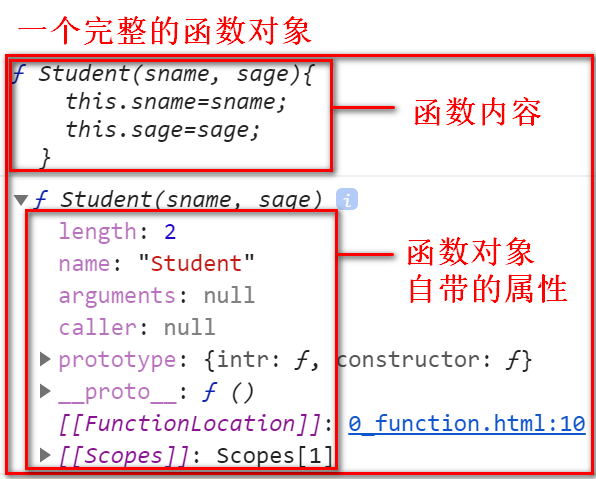
|  |
| --- |
| this共有: 4种: 判断this一定不要看定义在哪儿！！只看调用时.前是谁。  1. obj.fun() fun中的this->.前的obj对象  2. new Fun() Fun中的this-> new正在创建的新对象  3. 构造函数.prototype.fun=function(){ ... } fun中的this->将来调用这个fun()函数的当前类型的某一个子对象  4. (function(){ ... })() 或 普通函数调用fun() this默认->window |

补: 函数也是一个对象，既保存函数体，同时也有自己的属性

console.log(Student) 仅输出函数内容

console.dir(Student) 仅输出函数对象在内存中的属性

以上两个输出的内容合起来，才是一个完整的函数对象。



|  |
| --- |
| 补: 不能用for in来遍历索引数组  因为in不仅遍历当前对象的所有自有成员，而且会延\_ \_proto\_ \_继续遍历原型对象中深紫色的共有成员（忽略浅紫色的成员）。 |

ES5:

1. 严格模式: 4个新要求:

(1). 禁止给未声明的变量赋值

(2). 静默失败升级为错误

(3). 匿名函数自调和普通函数调用中的this不再指window，而是undefined

(4). 禁用了arguments.callee，不推荐使用递归。

2. 保护对象:

(1). 保护对象的属性:

a. 使用开关对对象的属性提供简单的保护:

1). 每个属性包括三个开关:

i. writable: 控制是否可修改属性值

写 可以 ——可修改

ii. enumerable: 控制是否可被for in遍历到。但是只是半隐藏，只防for in，不防.

遍历 可以 ——可以被遍历

iii. configurable: 控制2件事:

配置 可以 ——可以配置

1. 是否可删除该属性

2. 是否可修改前两个特性：因为一旦改为false不可逆，所以凡是修改前两个特性时，都要带上configurable:false，作为双保险。

2). 如果只修改一个属性的开关:

Object.defineProperty(对象, "属性名", {

开关: true 或 false,

... : ...

})

3). 如果想同时修改一个对象的多个属性的开关:

Object.defineProperties(对象,{

属性名: {

开关: true 或 false,

... : ...

},

属性名: {

... : ...,

}

... : { ... }

})

b. 用访问器属性，保护对象的属性:

1). 定义访问器属性: 2步:

i. 将被保护的数据属性隐姓埋名，并且半隐藏:

var eric={

\_eage:25

}

Object.defineProperties(eric,{

\_eage:{

enumerable:false,

configurable:false

},

ii. 为受保护的数据属性请保镖:

eage:{//保镖应该冒名顶替原数据属性的值

//保镖一请就是一对儿

get:function(){ //专门负责从受保护的数据属性中获取值

return this.\_eage

},

set:function(value){//专门负责将新值经过验证后，再保存回受保护的数据属性中

if(value符合要求){

this.\_eage=value

}else{

throw Error(`错误提示`)

}

},

enumerable:true, //访问器属性顶替受保护的属性抛头露面

configurable:false, //访问器属性不能被轻易删除

}

})

2). 如何使用访问器属性: 访问器属性的用法和普通属性完全一样

i. 获取属性值时: 对象.访问器属性 自动调用访问器属性的get

ii. 修改属性值时: 对象.访问器属性=新值 自动调用访问属性的set，同时自动将=右侧的新值传给set的value形参变量。经过验证value后，再赋值。

(2). 保护对象的结构: 3个级别:

a. 防扩展: Object.preventExtensions(对象)

只禁止给对象添加新属性

b. 密封: Object.seal(对象)

既禁止给对象添加新属性，又禁止删除现有属性（自动设置configurable:false）

c. 冻结: Object.freeze(对象)

既禁止给对象添加新属性

又禁止删除现有属性（自动设置configurable:false）

同时还禁止修改属性值(自动设置writable:false)

3. Object.create()

(1). 何时: 当没有构造函数时，也想为一个父对象创建子对象

(2). 如何:

var 子对象=Object.create(父对象,{

自有属性名:{

value:属性值,

writable:true,

enumerable:true,

configurable:false

},

... : { ... }

})

(3). 做了三件事:

a. 创建一个新的空对象

b. 让新对象继承父对象

c. 为新对象添加自有属性

4. 替换this:

(1). 调用一次函数，同时临时替换一次this:

a. 默认call: 要调用的函数.call(替换this的对象, 实参值列表)

b. 如果实参值列表是放在一个数组中给的，和要调用的函数发生不一致，应该用apply打散数组，再传参

要调用的函数.apply(替换this的对象, 数组)

(2). 反复调用函数，永久替换this，应该用bind:2步

a. 先用bind创建函数副本:

var 新函数=原函数.bind(替换this的对象, 固定不变的实参值)

b. 再反复调用新函数: 不用再传入替换this的对象和部分已经固定的实参值

新函数(除已经固定的实参值之外其他实参值列表)

5. 数组新增函数:

(1). 判断:

a. 判断数组中是否包含符合条件的元素

var bool=arr.some(function(val){

return val是否符合条件

})

b. 判断数组中是否所有元素都符合要求

var bool=arr.every(function(val){

return val是否符合条件

})

(2). 遍历:

a. 如果仅遍历原数组，就用:

arr.forEach(funtion(val){

... 不需要return ...

})

b. 如果遍历原数组，修改后，返回新数组，保持原数组不变，就用:

var 新数组=arr.map(function(val){.

.. return 修改val获得新值;  
 })

(3). 过滤和汇总:

a. 过滤: 复制出数组中符合条件的元素，放入新数组返回

var 新数组=arr.filter(function(val){

return 判断val是否符合要求的条件

})

b. 汇总: （仅以对数组中所有元素求和来举例）

var 统计结果=arr.reduce(

function(prev,val){

prev+=val;

return prev

},

0 //汇总开始时的起始值——基数。

)

总结: this 5种

1. obj.fun() this->obj

2. new Fun() this->新对象

3. 构造函数.prototype.fun=function(){} this->将来调用fun函数的某一个子对象

4. fun() 或 (function(){ })() this->window

5. 回调函数中的this->window

总结: ES6

1. 模板字符串: 今后所有字符串拼接都要用模板字符串

`... ${ 一切有返回值的js表达式 } ...`

2. let: 今后凡是变量的声明都用let代替var

凡是常量的声明都用const

3. 箭头函数: 今后程序中，几乎看不到任何function

(1). 去掉function，在()和{}之间加=>

(2). 如果只有一个形参，则可省略()

(3). 如果函数体只有一句话，可省略{}，以及函数体结尾的;

但是如果仅有的一句话还是return，则必须省略return

强调: 不希望内外this相同时，不能用箭头函数简写

4. for of: 专门简写标准的普通for循环(从头到尾依次遍历)的语法

for(var val of 一切数字下标的东西){

//val会依次获得of后的数组中每个元素值，无法获得元素位置

}

5. 参数增强:

(1). 参数默认值: 专门解决仅最后一个形参变量不确定有没有的情况

function 函数名(形参1, 形参2, ..., 最后一个形参=默认值){

... ...

}

(2). 剩余参数: 代替arguments，来接住不确定个数的实参值

定义函数时

function 函数名(形参1, 形参2, ...数组){

//...自动收集除之前形参变量之外的剩余实参值列表，放入数组中

}

(3). 打散数组: 调用函数时，将一个数组先打散为多个实参值，再分别传给函数的形参变量

函数名(实参值1, 实参值2, ...数组)

6. 解构:

(1). 数组解构: var [变量1, 变量2, ... ]=数组

(2). 对象解构: var {属性名1, 属性名2, ...}=对象

一个属性名两用：既当做属性名去对象中配对要解构的属性，又当做将来接受属性值单独使用的变量名

(3). 参数解构: 如果多个形参不确定有没有，且又要求实参值必须传给对应的形参变量时。2步

function 函数名({ 形参1=默认值, 形参2=默认值, ... }){

}

函数名({

形参1: 实参值1,

形参2: 实参值2,

... : ...

})

7. class

(1). 今后只要想用构造函数反复创建多个相同结构的子对象时，都要改为用class来创建

class 类型名{

constructor(形参列表){

this.属性名=形参;

... = ... ;

}

原型对象方法(){

... this.属性名...

}

}

var 子对象=new 类型名(属性值列表)

强调: 虽然改写成class，但是底层原理和构造函数+原型对象是完全一样的

(2). 两种class间的继承: 如果多个class拥有部分相同的属性结构和方法定义，则可以定义一个父级class来集中保存所有子级class相同部分的属性和方法。然后，再让子类型class继承父类型class。——结果子类型class创建出的子对象，子类型和父类型所有的属性和方法都能使用

class 父类型{

constructor(形参列表){

this.相同部分的属性名=形参

... = ...

}

相同部分的方法名(){

... ...

}

}

class 子类型 extends 父类型{

constructor(形参列表){

super(形参变量列表); //借用父类型构造函数帮子类型构造函数共同创建子对象中的自有属性

this.子类型独有属性=形参

}

}

var obj=new 子类型(属性值列表)

总结: this 5种: 判断this一定不要看定义在哪儿！只看调用时前边有没有.或new！

1. obj.fun() this->.前的obj

2. fun() 或 (function(){})() this->window

3. new Fun() this->new正在创建的新对象

4. 构造函数.prototype.fun=function(){ ... } this->将来调用这个函数的.前的某个子对象

5. 回调函数中的this->window