提前准备:

1. 今天的历届师兄师姐问过的经典的高频的问题清单都已经整理在小程序->首页->JSCORE->day06里

2. 今天的课程中所需的单词列表都已经整理在小程序->单词->JSCORE->day06里

3. 今天的重点视频、扩展高频鄙视题视频已经整理在小程序->在线->JSCORE->day06里

如果不知道问什么: 谁问的越多！谁培训费花的越值！

1. 笔记中或画图中那句话或者那个位置不明白，立刻问！

2. 案例中哪行代码或注释不明白，立刻问！

3. 课下自己做项目，功能不会做，立刻问！

4. 刷笔试题时遇到不会的笔试题，立刻问！——公众号: 前端大全 定期总结大厂笔试题

5. 看小程序问题清单时，遇到看不懂的问题，立刻问！

衡量是否学会的标准：

1. 可以用自己的话把注释的步骤写出来！才算会！（函数名或程序忘了都没关系！）

2. 可以自己画出内存变化过程图！才算会！（凡是只用眼睛看想得出揭露的！就都是错误的结论！）

3. 可以把老师上课的案例，经过修改后，应用到自己的项目里

正课:

/\*ES5\*/

1. 保护对象结构
2. Object.create()
3. call()/apply()/bind()
4. 数组新增的函数

/\*ES6\*/

1. 回顾模板字符串
2. let
3. for of
4. 保护对象结构:
5. 问题: 使用Object.defineProperty()或Object.defineProperties()只能保护对象的单个属性，无法防止添加新属性
6. 保护对象结构，主要就是防止为一个对象随意添加新属性，所以删除现有属性，破坏对象结构
7. 如何: 3个级别

(1). 防扩展: 禁止给一个对象添加新属性

a. Object.preventExtensions(对象)

阻止 扩展

b. 意为: 防止为对象扩展新属性

c. 强行添加: 报错: Cannot add property Eid, object is not extensible

不能 添加 属性 Eid (因为)对象 是 不 可扩展的

d. 示例: 阻止为eric添加新属性

|  |
| --- |
| "use strict";  var eric={    eid:1001, //只读    ename:"埃里克"  }  Object.defineProperty(eric,"eid",{    writable:false,    configurable:false  })  //阻止为eric扩展新属性  Object.preventExtensions(eric);  //尝试修改eid属性  //eric.eid=-2;  //尝试为eric添加新的不同名的eid属性  // eric.Eid=-2;  // eric.\_eid=1;  console.log(eric); |

(2). 密封:

a. 问题: 几乎所有对象的每个属性，都应该禁止删除，但是，每个属性都写configurable:false，非常繁琐！而且preventExtensions()只防扩展不防删除

b. 密封一个对象: 禁止给这个对象添加新属性以及删除现有属性

c. 何时: 几乎所有的对象，都应该保护到密封级别

d. 如何: Object.seal(obj)

密封

e. 意为: 既阻止为obj对象添加新属性，又禁止删除现有属性

f. 原理: seal()做两件事:

1). 自动调用preventExtensions()，所以使用seal，就不用preventExtensions了

2). 自动将每个属性的configurable:false，所以使用seal，就不用给每个属性重复写configurable

g. 示例: 密封一个对象:

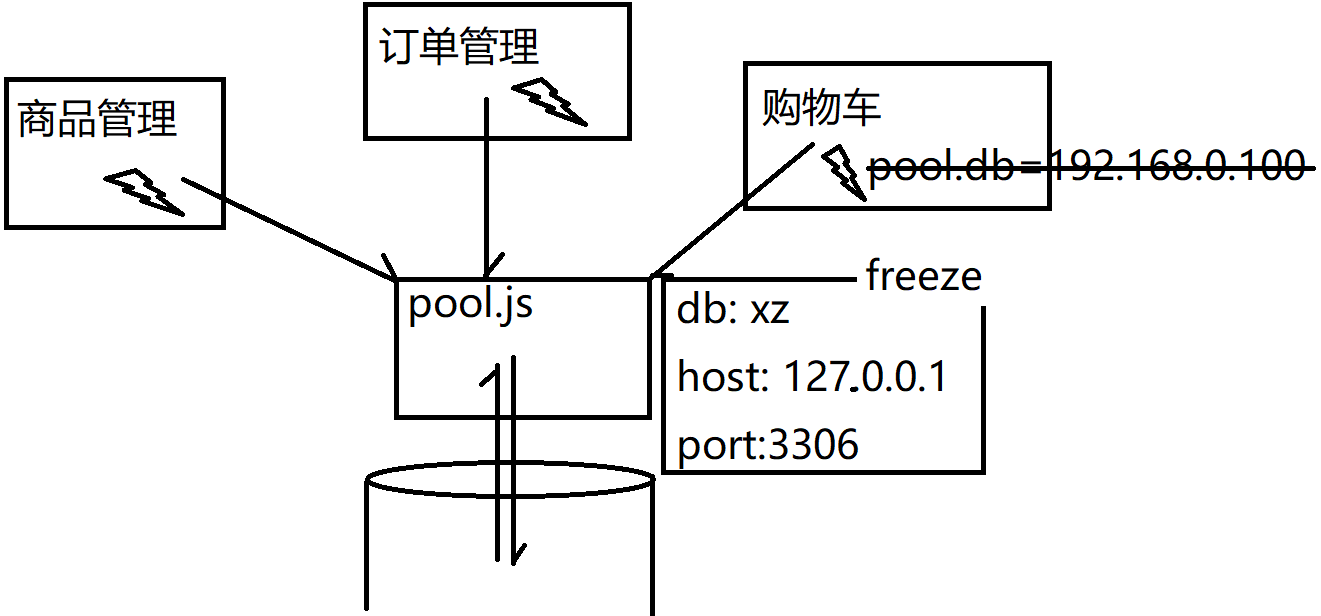
|  |
| --- |
| "use strict";  var eric={    eid:1001, //只读    ename:"埃里克" //禁止删除  }  Object.defineProperties(eric,{    eid:{      writable:false    }  })  //阻止为eric扩展新属性  //Object.preventExtensions(eric);  //密封eric对象: 既可防扩展，又可防删除  Object.seal(eric);  //尝试修改eid属性  //eric.eid=-2;  //尝试为eric添加新的不同名的eid属性  //eric.Eid=-2;  //尝试删除ename属性  //delete eric.ename;  console.log(eric); |

(3). 冻结: (过于严格，只在特殊情况下使用)

a. 什么是冻结: 既禁止添加删除属性，同时又禁止修改属性值！

b. 如何: Object.freeze(obj)

c. 何时: 今后只要一个对象，被多个模块共用，则这个对象中的成员，就不能随意被某个模块修改，否则牵一发而动全身！



d. 意为: 禁止添加删除obj中的属性，甚至连属性值都不能改了！

e. 原理:

1). 自动调用preventExtensions()

2). 自动将每个属性的configurable:false

3). 自动将每个属性的writable:false

f. 示例: 冻结pool对象

|  |
| --- |
| "use strict";  var pool={    host:"127.0.0.1", //writable:false    port:3306, //writable:false    db:"xz" //writeable:false  }  //冻结pool对象  Object.freeze(pool);  pool.host="192.168.0.100";  pool.db="test";  console.log(pool); |

1. Object.create()
2. 什么是Object.create(): 专门基于一个现有的父对象，创建一个新的子对象，继承父对象
3. 何时: 即使没有构造函数妈妈，只有一个父对象，也想创建子对象，继承父对象
4. 如何:

(1). var 新子对象=Object.create(父对象)

a. 意为: 创建一个新的子对象，自动设置新子对象的\_ \_proto\_ \_指向父对象

b. 强调: 这样添加完的对象，是没有任何自有属性的，只能用爹里的共有属性

b. 问题: 如果新的子对象需要添加自有属性

(2). 即创建子对象继承父对象，同时又为子对象添加自有属性

a. var 子对象=Object.create(父对象,{

//内容同defineProperties中的内容

自有属性1: {

value: 属性值,

三个开关: true或false,

… : …

},

… : {

…

}

})

1. 总结: Object.create()一共干了3件事？

(1). 先创建一个新的子对象

(2). 让新的子对象继承父对象

(3). 为新子对象添加自有属性

5. 示例: 使用Object.create()创建子对象，继承父对象，并添加自有属性

|  |
| --- |
| var father={    bal:10000000000,    car:"infiniti"  }  var hmm=Object.create(father,{    phone:{      value:"iPhoneX",      writable:true,      enumerable:true    },    bao:{      value:"LV",      writable:true,      enumerable:true    }  });  Object.seal(hmm);  console.log(hmm);  console.log(hmm.bal, hmm.car);  console.log(hmm.phone, hmm.bao); |

1. call()/apply()/bind()
2. 都能替换函数中不想要的this为想要的对象
3. 何时: 今后，只要一个函数中的this不是想要的，都可用这个三个函数来更换this指向想要的对象
4. 但是分为两种情况:

(1). 只在一次调用函数时，临时替换一次this为指定对象

a. 要调动的函数.call(替换this的对象, 实参1, 实参2,…)

b. call做了三件事:

1). 调用函数

2). 用call的第一个参数对象临时替换函数中的this为这个对象

3). 将call从第二个参数开始的实参值传给正在调用的函数的形参变量

c. 总结: call其实可以让一个对象，去调用任何一个原本和自己没有任何关系的函数。并让当前对象自己临时成为那个函数的主人！

d. 示例: 用call()让lilei去调用原本自己不能调用到的函数

|  |
| --- |
| //比如有一个共用的函数jisuan，可计算某个员工的工资  //window中:  function jisuan(base,bonus1,bonus2){    console.log(`${this.ename}的总工资是${base+bonus1+bonus2}`)  }  //有两个员工:  var lilei={ename:"Li Lei"};  var hmm={ename:"Han Meimei"};  //两个员工都想用jisuan()函数，计算自己的薪资  //错误1: jisuan() this->window，所以不是想要的  //错误2: lilei.jisuan() 错误！因为李雷自己没有jisuan()函数，同时李雷的爹是Object的原型对象，也没有jisuan()函数，所以lilei无权调用jisuan()函数!  console.log(lilei);  console.log(window);  //lilei.jisuan(); //报错: lilei.jisuan()不是一个函数  //正确: 因为jisuan()在全局定义，所以可直接调用  //      但是因为jisuan()中的this不是lilei，想换成lilei，就要用call()来调用jisuan()  jisuan.call(lilei,10000,1000,2000)  //jisuan()中的this被临时替换为->lilei  //其实执行的一瞬间，等效于执行lilei.jisuan()  //其实相当于让lilei调用了一个原本不可能被调用到的函数——抢  //hmm也想抢jisuan()函数计算自己的薪资  jisuan      .call( hmm,4000, 5000, 6000)  //                  |    ↓     ↓     ↓  //function jisuan(  ↓  base,bonus1,bonus2){    //console.log(`${this.ename}的总工资是${base+bonus1+bonus2}`)  //} |

e. 如果要抢的函数的形参变量有多个，但是要传给这个函数的所有实参值却是放在一个数组中给的！出现不一致！

1). 就要用apply()代替call()

2). 要抢的函数.apply(替换this的对象, 保存多个实参值的数组)

3). apply也做了三件事:

i. 也调用函数——同call()

ii. 也用apply的第一个参数对象临时替换函数中的this为这个对象——同call

iii. 将apply第二个保存多个实参值的数组，先打散！再按顺序逐个传给正在调用的函数的形参变量

4). 示例: 使用apply()代替call()打散数组

|  |
| --- |
| //比如有一个共用的函数jisuan，可计算某个员工的工资  //window中:  function jisuan(base,bonus1,bonus2){    console.log(`${this.ename}的总工资是${base+bonus1+bonus2}`)  }  //有两个员工:  var lilei={ename:"Li Lei"};  var hmm={ename:"Han Meimei"};  //但是hmm的工资是放在数组中给的  var arr=[4000,5000,6000];  jisuan     .apply( hmm,        arr        );  //                  |        打 | 散  //                  |   4000, 5000 , 6000  //                  |     ↓     ↓      ↓  //function jisuan(  ↓   base,bonus1,bonus2){    //console.log(`${this.ename}的总工资是${base+bonus1+bonus2}`)  //} |

(2). 基于原函数，创建一个新函数，并永久绑定this为指定对象

a. 问题: 如果需要反复频繁抢一个函数，每次都要用call或apply，每次都要写自己的对象名，就很繁琐，不方便了

b. 解决: 为当前对象买一个专属的函数。

c. var 新函数=原函数.bind(替换this的对象)

d. 暂时只做两件事:

1). 创建一个和原函数一模一样的副本

2). 永久替换this为指定的对象

e. 结果: 今后再反复调用新函数副本时，不需要再.call(对象名)。函数中的this也指向当前对象自己了。

f. 示例: 使用bind的基本用法仅永久替换this

|  |
| --- |
| //比如有一个共用的函数jisuan，可计算某个员工的工资  //window中:  function jisuan(base,bonus1,bonus2){    console.log(`${this.ename}的总工资是${base+bonus1+bonus2}`)  }  //有两个员工:  var lilei={ename:"Li Lei"};  var hmm={ename:"Han Meimei"};  //如果两个人需要反复频繁给自己算薪资，总是抢，就很不方便了  //李雷下决心自己买一个专属于自己的  var jisuan2=jisuan.bind(lilei);  //bind干了2件事:  //1. 基于原函数jisuan()，创建出一个一模一样的新函数副本  //2. 将bind后第一个实参对象，永久替换进了新函数副本中，永久代替了this  //jisuan2:function jisuan(base,bonus1,bonus2){    //               lilei    //                 ↓    //console.log(`${this.ename}的总工资是${base+bonus1+bonus2}`)  //}  //结果: jisuan2() 永久的专属于lilei了  // jisuan.call(lilei,10000,1000,2000)  console.log(jisuan);  console.log(jisuan2);  console.log(jisuan==jisuan2)  jisuan2(10000,1000,2000);  jisuan2(10000,500,500);  //hmm也买一个自己专属的  var jisuan3=jisuan.bind(hmm);  jisuan3(4000,5000,6000) |

g. 其实: bind不但可以永久绑定this，还可以永久绑定部分固定不变的实参值

1). var 新函数=原函数.bind(替换this的对象, 将来固定不变的实参值)

2). 其实bind能做3件事:

i. 基于原函数jisuan()，创建出一个一模一样的新函数副本

ii. 将bind后第一个实参对象，永久替换进了新函数副本中，永久代替了this

iii. 永久绑定部分形参变量的值为一个固定的值

3). 结果: 今后再反复调用新函数副本时，不需要再.call(对象名)，且固定不变的实参值，也不需要重复传入了。

h. 示例: 使用bind()永久替换this和部分固定的实参值

|  |
| --- |
| //比如有一个共用的函数jisuan，可计算某个员工的工资  //window中:  function jisuan(base,bonus1,bonus2){    console.log(`${this.ename}的总工资是${base+bonus1+bonus2}`)  }  //有两个员工:  var lilei={ename:"Li Lei"};  var hmm={ename:"Han Meimei"};  //如果两个人需要反复频繁给自己算薪资，总是抢，就很不方便了  //李雷下决心自己买一个专属于自己的  //其实bind不止能永久绑定this，还能绑定部分实参值  var jisuan2=jisuan.bind(lilei,10000);  //                       ↓     ↓  //                      this  base  //bind干了3件事:  //1. 基于原函数jisuan()，创建出一个一模一样的新函数副本  //2. 将bind后第一个实参对象，永久替换进了新函数副本中，永久代替了this  //3. 永久绑定部分形参变量的值为一个固定的值  //jisuan2:function jisuan(bonus1,bonus2){    //               lilei                10000    //                 ↓                    ↓    //console.log(`${this.ename}的总工资是${base+bonus1+bonus2}`)  //}  //结果: jisuan2() 永久的专属于lilei了,且永久不需要输入base，base也是10000了  // jisuan.call(lilei,10000,1000,2000)  console.log(jisuan);  console.log(jisuan2);  console.log(jisuan==jisuan2)  jisuan2(1000,2000);  jisuan2(500,500);  //hmm也买一个自己专属的  var jisuan3=jisuan.bind(hmm,4000);  jisuan3(5000,6000) |

4. 总结: 何时使用call()/apply()/bind()

(1). 只是临时调用一次函数，临时替换一次this: call或apply

a. 多数情况下默认用call

b. 特殊情况: 多个实参值是放在数组中给的，需要打散数组后，再逐个传参: apply()

(2). 如果反复调用函数，希望永久替换this为指定对象时: bind()

1. 数组新增函数:
2. 做判断用的:

(1). 判断数组中是否所有元素都符合要求:

a. var bool=arr.every(function(elem, i, arr){ //必须有三个形参变量

每个

return 判断条件 //必须返回一个判断条件

})

b. 原理:

1). every()中自带for循环，自动遍历arr数组中每个元素

2). 每遍历到一个元素，就自动调用一次回调函数，每次调用回调函数时，都会自动传入三个值

i. elem : 接住当前正在遍历到的数组元素的值

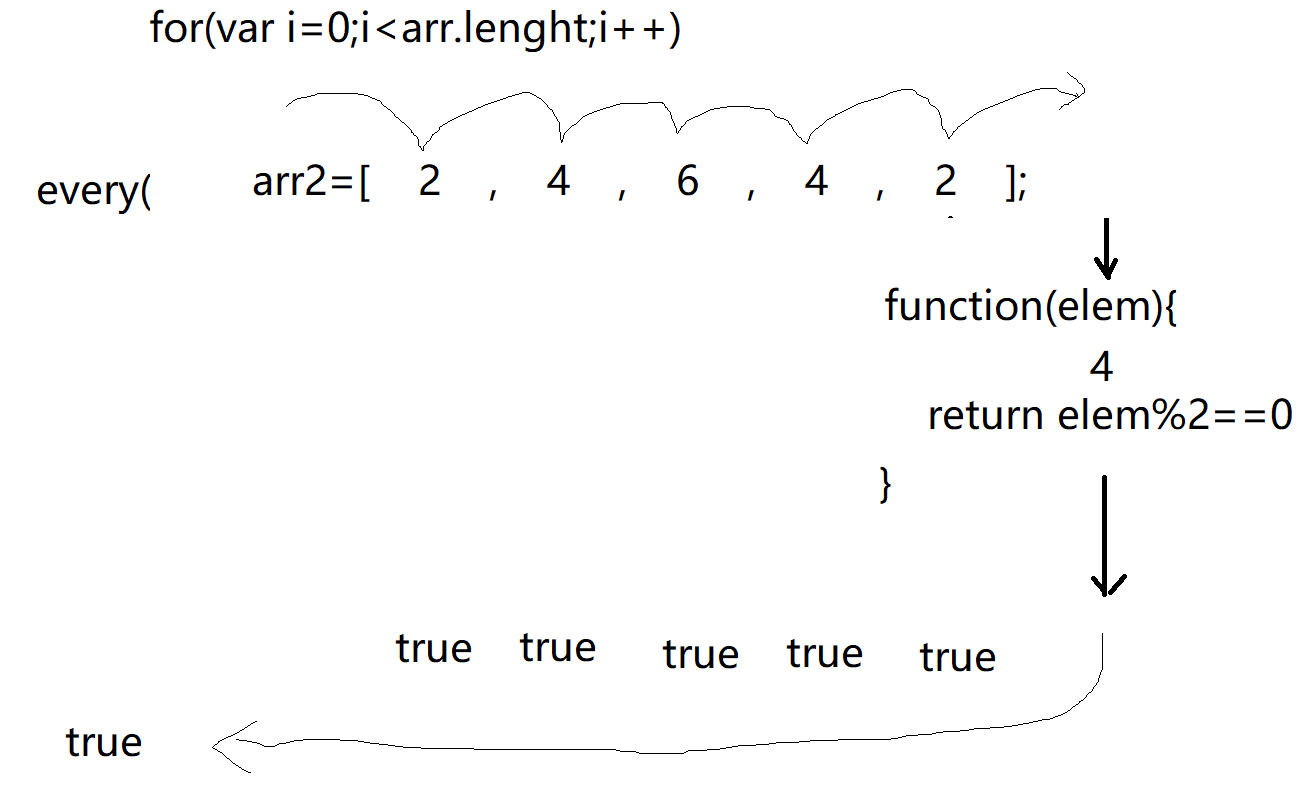
ii. i : 接住当前正在遍历到的元素的位置

iii. arr: 接住当前正在遍历的.前的数组对象本身

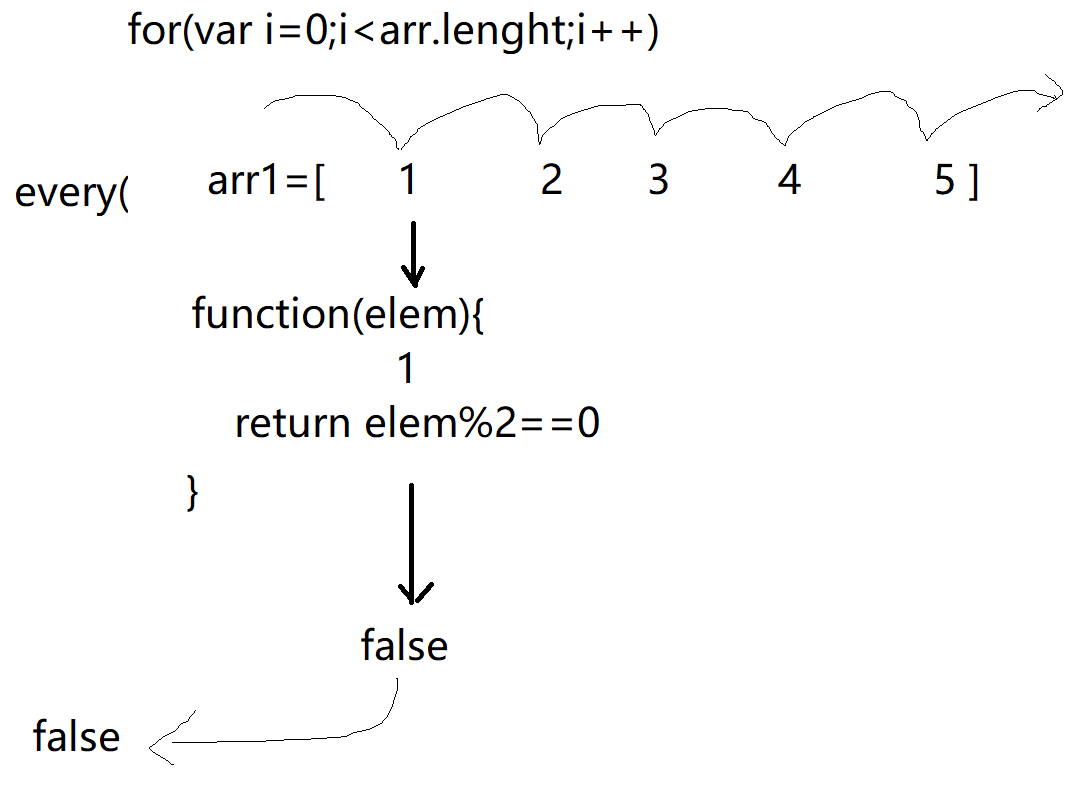
3). 每次调用回调函数时，回调函数内部都会根据提前设定好的判断条件，来判断当前元素是否符合要求

4). 如果当前元素经过条件判断，符合要求，本次回调函数就会返回true。否则回调函数返回false

5). every()只有在收到回调函数返回true时，才有必要继续遍历下一个元素，检查下一个元素是否符合要求。只有数组中所有元素经过检查，都返回true，整个判断才返回true，说明当前数组所有元素都符合要求。



6). 如果某一次回调函数检查某个元素，不符合要求，就返回false。如果every()收到回调函数返回false，则不再继续遍历后续元素，而是直接返回false，判断整个数组不是所有元素都符合要求。



c. 示例: 判断一个数组是否全由偶数组成:

|  |
| --- |
| var arr1=[1,2,3,4,5];  var arr2=[2,4,6,4,2];  //想判断哪个数组都是偶数组成  console.log(    arr1.every(function(elem){      console.log(`自动调用了一次回调函数,本次检查arr1中的${elem},返回${elem%2==0}`)      return elem%2==0;//当前元素是偶数吗？    }),    arr2.every(function(elem){      console.log(`自动调用了一次回调函数,本次检查arr2中的${elem},返回${elem%2==0}`)      return elem%2==0;    })  ) |

d. 示例: 判断哪个数组是升序排列

|  |
| --- |
| var arr1=[1,2,3,4,5];  var arr2=[2,4,6,4,2];  //想判断哪个数组是升序  console.log(    arr1.every(function(elem,i,arr){      console.log(`自动调用arr1.every()中的回调函数,用${arr}中的当前元素${elem}和下一个位置的元素${arr[i+1]}做比较，判断结果为${i<arr.length-1?elem<=arr[i+1]:true}`);      //如果当前位置i，还没到数组最后一个位置(length-1)，才正常比较。如果已经到最后一个元素位置，则直接返回true！放他过去！      return i<arr.length-1?elem<=arr[i+1]:true;//当前元素值<=下一个位置的元素值    }),    arr2.every(function(elem,i,arr){      console.log(`自动调用arr2.every()中的回调函数,用${arr}中的当前元素${elem}和下一个位置的元素${arr[i+1]}做比较，判断结果为${i<arr.length-1?elem<=arr[i+1]:true}`)      return i<arr.length-1?elem<=arr[i+1]:true;    })  ) |

(2). 判断数组中是否包含符合条件的元素

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

一些

return 判断条件

})

b.原理:

1). some()中也自带for循环，自动遍历arr数组中每个元素

2). 每遍历到一个元素，就自动调用一次回调函数，每次调用回调函数时，都会自动传入三个值

i. elem : 接住当前正在遍历到的数组元素的值

ii. i : 接住当前正在遍历到的元素的位置

iii. arr: 接住当前正在遍历的.前的数组对象本身

3). 每次调用回调函数时，回调函数内部都会根据提前设定好的判断条件，来判断当前元素是否符合要求

4). 如果当前元素经过条件判断，符合要求，本次回调函数就会返回true。否则回调函数返回false

5). some()只有在收到回调函数返回false时，才有必要继续遍历下一个元素，检查下一个元素是否符合要求。只有数组中所有元素经过检查，都返回false，整个判断才返回false，说明当前数组不包含符合要求的元素

6). 如果某一次回调函数检查某个元素，符合要求，就返回true。如果some()只要收到回调函数返回true，则不再继续遍历后续元素，而是直接返回true，说明整个数组中至少包含一个符合要求的元素.

c. 示例: 判断哪个数组包含奇数:

|  |
| --- |
| var arr1=[1,2,3,4,5];  var arr2=[2,4,6,4,2];  //想判断哪个数组包含奇数  console.log(    arr1.some(function(elem){      console.log(`自动调用了一次arr1.some()中的回调函数,本次检查arr1中的${elem},返回${elem%2!=0}`)      return elem%2!=0;//当前元素是奇数吗？    }),//true    arr2.some(function(elem){      console.log(`自动调用了一次arr1.some()中的回调函数,本次检查arr2中的${elem},返回${elem%2!=0}`)      return elem%2!=0;    }) //false  ) |

1. 遍历: 对数组中每个元素执行相同的操作

(1). 仅遍历原数组中每个元素，简写for循环：

a. 回顾: 旧的遍历数组:

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

arr[i] 可获得当前元素值

}

b. 问题: 几乎没有可以简化的空间了

c. 解决: 今后遍历数组，都可用arr.forEach()来简化for循环

d. arr.forEach(function(elem,i,arr){ //可以后三个形参

//只要读出原数组中的元素值，就算成功，所以无需返回值！

})

e. 原理:

1). forEach也自带for循环，可自动遍历数组中每个元素

2). 每遍历一个元素就自动调用一次回调函数，调用回调函数时，都自动传入三个值:

i. elem : 接收当前正在遍历的元素值

ii. i : 接收当前正在遍历的位置

iii. arr : 接收到当前正在遍历的.前的数组对象

f. 示例: 点名

|  |
| --- |
| //点名:  var names=["亮亮","然然","东东"];  //旧  // for(var i=0;i<names.length;i++){  //   alert(`${names[i]} - 到！`);  // }  //ES5  //names.forEach(function(elem){ alert(`${elem} - 到！`) })  //箭头函数进一步简写（明天讲！先别问，或者自己先看小程序day07。）  names.forEach(n=>alert(`${n} - 到！`)); |

(2). 取出原数组中的值，执行相同的操作后，放入新数组中保存:

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

//因为加工后的新元素值要放入新数组中返回，所以回调函数必须要有返回值return !

return 加工后的新元素值

})

b. 原理:

1). map第一件事儿是先创建一个新的空数组！

2). map内部也自带for循环，自动遍历原数组中每个元素

3). 每遍历一个元素，就自动调用一次回调函数，每次调用回调函数时，都自动传入三个值:

i. elem : 接住当前正在遍历到的数组元素的值

ii. i : 接住当前正在遍历到的元素的位置

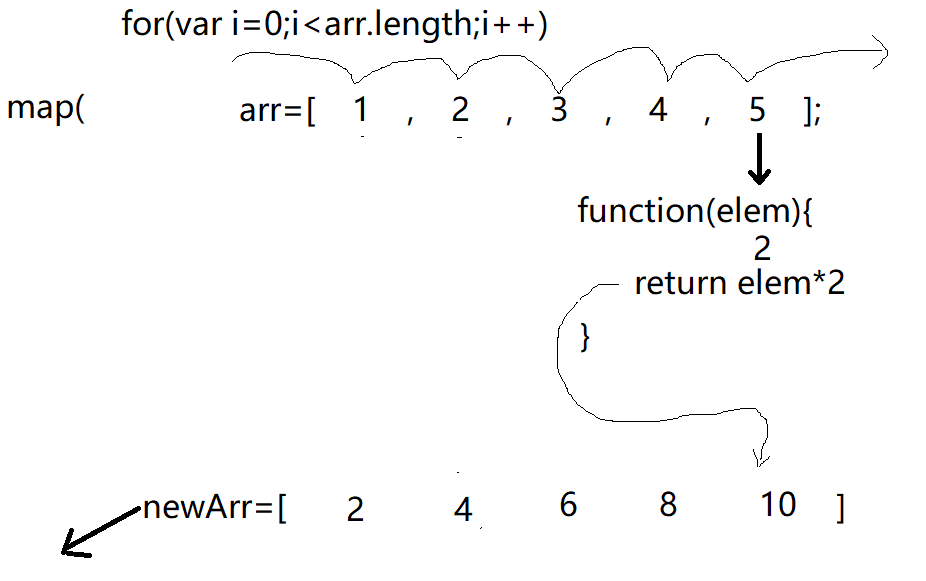
iii. arr: 接住当前正在遍历的.前的数组对象本身

4). 回调函数内部对当前元素值进行加工后，返回新元素值给map()函数

5). map()函数负责将回调函数返回的新元素值，放入新数组中相同位置

6). 当遍历结束，则map()返回新数组。原数组内容保持不变！

c. 示例: 将数组中每个元素值\*2，放入新数组中返回!



|  |
| --- |
| var arr=[1,2,3,4,5];  //希望将每个元素值\*2，但是保护原数组不变，返回\*2后的新数组  var arr2=arr.map(    //先创建一个空数组等着接新元素值    //var newArr=[];    //自动遍历原数组中每个元素值    //for(var i=0;i<arr2.lenght;i++){      //每遍历一个元素就自动调用一次回调函数，并传入当前元素值      //       arr2[i]      //         ↓      function(elem){        return elem\*2      }//(elem=arr[i])      //将回调函数返回值，放入新数组中相同位置      //newArr[i]=\*2后的新值    //}    //当遍历结束，返回接住所有新元素值的新数组    //return newArr;  );  //arr2=newArr;  console.log(arr);  console.log(arr2); |

(3)总结:

a. 如果单纯遍历原数组，不产生新数组时，就用forEach

b. 希望保护原数组，而产生新数组时，就用map()

1. 过滤和汇总:

(1). 过滤: 复制出数组中符合条件的元素，组成新的子数组返回。原数组保持不变

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

//因为要根据回调函数的判断结果来决定是否将当前元素放入新数组中返回，所以需要返回一个判断条件

return 判断条件

})

b. 原理:

1).filter第一件事儿是先创建一个新的空数组！

2). filter内部也自带for循环，自动遍历原数组中每个元素

3). 每遍历一个元素，就自动调用一次回调函数，每次调用回调函数时，都自动传入三个值:

i. elem : 接住当前正在遍历到的数组元素的值

ii. i : 接住当前正在遍历到的元素的位置

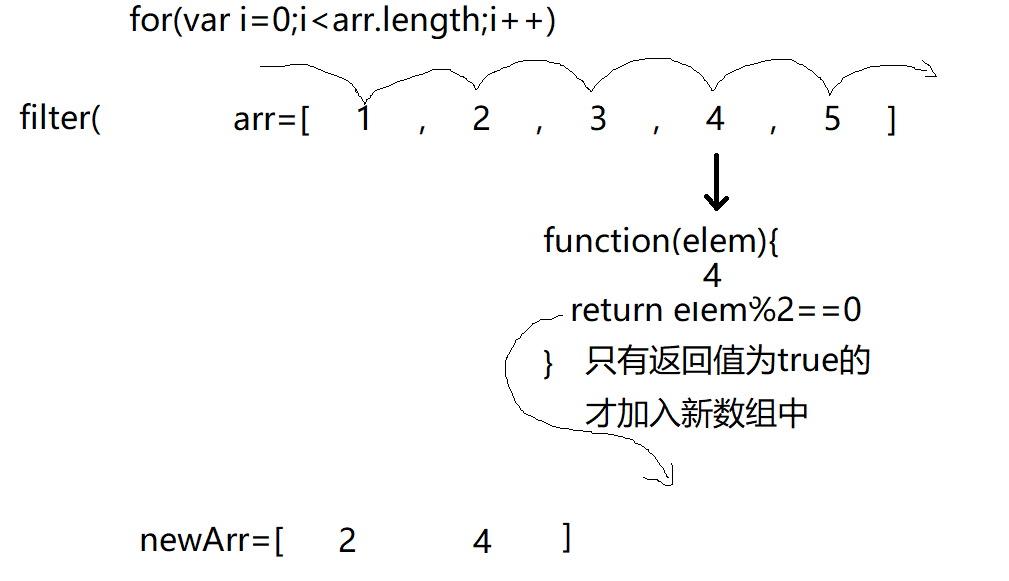
iii. arr: 接住当前正在遍历的.前的数组对象本身

4). 回调函数内部对当前元素值进行判断，返回判断结果

5). filter()函数根据回调函数返回的判断结果来决定是否将当前元素追加进新数组中。只有那些经过判断返回true的元素，才有资格被追加到新数组中

6). 当遍历结束，则filter ()返回新数组。原数组内容保持不变！

c. 示例: 复制出数组中的偶数元素，组成新数组



|  |
| --- |
| var arr=[1,2,3,4,5];  //想复制出数组中的偶数元素放入新数组中  var arr2=arr.filter(function(elem){    return elem%2==0;//只有偶数元素值才能放入新数组中  })  console.log(arr);  console.log(arr2); |

总结:面向对象: 面向对象三大特点:

1. 封装: 2种

(1). 只创建一个对象: {}

(2). 反复创建多个相同结构的对象时: 构造函数

1. 继承: 所有子对象共用的方法都应该定义在原型对象中:

(1). 如果只给原型对象添加一个新的共有方法:

构造函数.prototype.新方法=function(){ … }

(2). 如果整个原型对象都不想要了，可以整体替换原型对象

构造函数.prototype={

新方法:function(){ … },

… : …

}

1. 多态: 如果从父对象继承来的共有成员不好用，就可在当前子对象内重写与父对象中同名的成员
2. 自定义继承:

(1). 只更换一个对象的爹:

子对象.\_\_proto\_\_=新爹

或

Object.setPrototypeof(子对象, 新爹)

(2). 如果要更换所有子对象的爹:

构造函数.prototype=新爹

时机: 开始创建子对象之前就要更换

总结: this共有5种用法:

1. obj.fun() this->obj
2. new Fun() this->new正在创建的新对象
3. fun() 或 (function(){ … })() this->window
4. 构造函数.prototype.共有方法=function(){ … } this->将来调用这个函数的某个子对象
5. 访问器属性中的this: 指当前访问器属性所在的对象本身(eric对象)

Object.defineProperty(eric, “eage”, {

get:function(){ return this.\_eage},

set:function(value){ this.\_eage=value }

})

总结: 保护对象:

1. 保护单个属性

(1). 通过开关保护属性: 3个开关

a. writable: 控制着是否可修改属性值

b. enumerable: 控制着是否可被for in遍历到，防不住.

c. configurable: 控制两件事: 1. 是否可删除该属性, 2. 是否可修改前两个开关

(2). 如果需要用自定义规则保护属性: 访问器属性

a. 先将原属性隐姓埋名且半隐藏

Object.defineProperty(eric,”\_eage”,{

value: 原属性值eric.eage,

writable: true,

enumerable: false, //半隐藏, 防for in，但是不防”.”

configurable:false //禁止删除，一旦给为false不可逆

})

b. 请保镖: 冒名顶替原属性

Object.defineProperty(eric,”eage”,{

get:function(){

return this.\_eage;

},

set:function(value){

if(value>=18&&value<=65){

this.\_eage=value;

}else{

throw Error(“年龄必须介于18~65之间”)

}

}

})

1. 保护对象结构: 常用2个级别:

(1). 一般对象如果只禁止添加删除属性: Object.seal(obj)

(2). 只有个别多个模块共用的对象才需要冻结: Object.freeze(obj)

总结: ES5中的武器:

1. 如果没有构造函数，只有一个父对象，也想创建子对象时: Object.create()
2. 替换函数中不想要的this:

(1). 在调用函数时，临时替换一次this: call或apply

a. 默认情况多数用call()

b. 多个实参值是放在一个数组中给的，需要打散数组，再逐个传参: apply()

(2). 如果希望反复调用函数，并且永久绑定this和部分实参值: bind()

3. 数组新函数:

(1). 判断:

a. 判断数组中是否所有元素都符合要求: every

b. 判断数组中是否包含符合要求的元素: some

(2). 遍历:

a. 单纯遍历，不返回新数组: forEach()

b. 保护原数组不变，返回新数组: map()

作业:

1. (必须)看小程序视频学习forEach中按值传递的坑

小程序->在线->JSCORE->day06->4. forEach和5. forEach 按值传递

1. (必须)看小程序视频学习如何用map动态生成页面元素

小程序->在线->JSCORE->day06->6. map 原理…

1. (必须)看小程序视频学习数组中的indexOf函数

小程序->在线->JSCORE->day06->作业: 数组家也有indexOf函数

1. (扩展)看小程序视频学习扩展笔试题:

小程序->在线->JSCORE->day06->

作业: 1. 将类数组对象转为数组

作业: 2. 高难度笔试题 用原生call()模拟实现bind函数。

作业: 使用原生js仿写forEach和map函数