小程序: WEB问题速查

1. 问题清单: 小程序->首页->JSCORE->day06、07

当老师在回答其它同学的问题时，你可以打开问题清单浏览曾经回答过的问题！

2. 高清无码小视频: 小程序->在线->JSCORE->day06、day07

2部分: 1. 课上重点知识和案例的回放！2. 重点扩展笔试面试题讲解扩展视频

今天扩展视频:

小程序->在线->JSCORE->day06

4.1 every判断数组是否升序排列

6. forEach按值传递

7. map原理...动态生成页面元素

所有以"作业"开头的视频

小程序->在线->JSCORE->day07 所有以"作业"开头的视频

所有扩展视频绝\*\*\*不要求当天必须看完\*\*\*！而且和第二天课程也没有关系！

只要在\*\*\*两个月后\*\*\*鄙视面试前\*\*\*看完就行\*\*\*！

3. 单词列表: 小程序->单词->JSCORE->day06、day07

第三阶段上课约定:

1. 上课千万不要跟着记笔记和写代码！

2. 拼写错误！自己找！一旦让我发现是拼写错误，自己还懒得找，我可以帮你找。但是，如果我找到，你需要手抄20遍，给我拍照！

3. 一个原理或知识点，只有自己可以画出图来，才算自己学会！

如何提问:

1. 上课时，案例中，笔记中那句话哪行代码看不懂，立刻微信私信问我！

2. 小程序中问题清单里哪个问题和回答看不懂！立刻微信私信问我！

3. 个人项目中哪个功能不会做！立刻微信私信问我！

4. 刷笔试题时，遇到不会的笔试题！立刻微信私信问我！

下载源代码: 右键->链接另存为！

正课:

\*\*\*\*\*ES5\*\*\*\*\*

1. 数组函数

\*\*\*\*\*ES6\*\*\*\*\*

1. 模板字符串

2. let

3. 箭头函数  
4. for of

5. 参数增强

\*\*\*\*\*ES5\*\*\*\*\*

一. 数组函数

1. 遍历: 2个:

(1). forEach:

对应小程序视频: 小程序->在线->JSCORE->day06 5. forEach ...

a. 什么是: 专门遍历数组中每个元素，执行相同的操作！

b. 问题: for循环虽然也可以遍历数组中的元素，但是语法上已经无法进一步简化！

c. 解决: 今后直到遍历索引数组，都用forEach来代替for循环

d. 如何:

数组.forEach(function(当前元素值, 当前位置, 当前数组){

对当前元素执行相同的操作

})

e. 原理:

1). forEach内已经自带for循环，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

3). 每次调用回调函数时，都会自动传入三个值:

i. 将当前正在遍历的元素值传给回调函数第一个形参

ii. 将当前正在遍历的下标位置传给回调函数第二个形参

iii. 将当前数组对象传给回调函数的第三个形参

4). 在回调函数内部，可以用当前元素值执行操作

f. 问题: 因为forEach是数组家的函数，所以只能遍历索引数组，无法遍历字符串或类数组对象等；for循环几乎是万能的！

g. 示例: 使用forEach点名

1\_forEach.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      var arr=["亮亮","然然","东东"];      //点名:      //for      // for(var i=0;i<arr.length;i++){      //   console.log(`${arr[i]} - 到！`)      // }      //forEach      // arr.forEach(function(elem,i,arr){      //   console.log(`${elem} - 到！`)      // })      // arr.forEach(function(elem){      //   console.log(`${elem} - 到！`)      // })      arr.forEach(elem=>console.log(`${elem} - 到！`));    </script>  </body>  </html>  运行结果:  亮亮 - 到！  然然 - 到！  东东 - 到！ |

(2). map:

对应小程序视频: 小程序->在线->JSCORE->day06 7. map ...

a. 什么是: 基于原数组的内容，改造出一个新的数组

b. 如何:

var 新数组=原数组.map(function(当前元素值, 当前下标, 当前数组){

return 将当前元素值修改后得到的新值

})

c. 原理:

1). 先创建一个新的空数组等待。

2). map内已经自带for循环，自动遍历数组中每个元素

3). 每遍历一个元素，就自动调用一次回调函数

4). 每次调用回调函数时，都会自动传入三个值:

i. 将当前正在遍历的元素值传给回调函数第一个形参

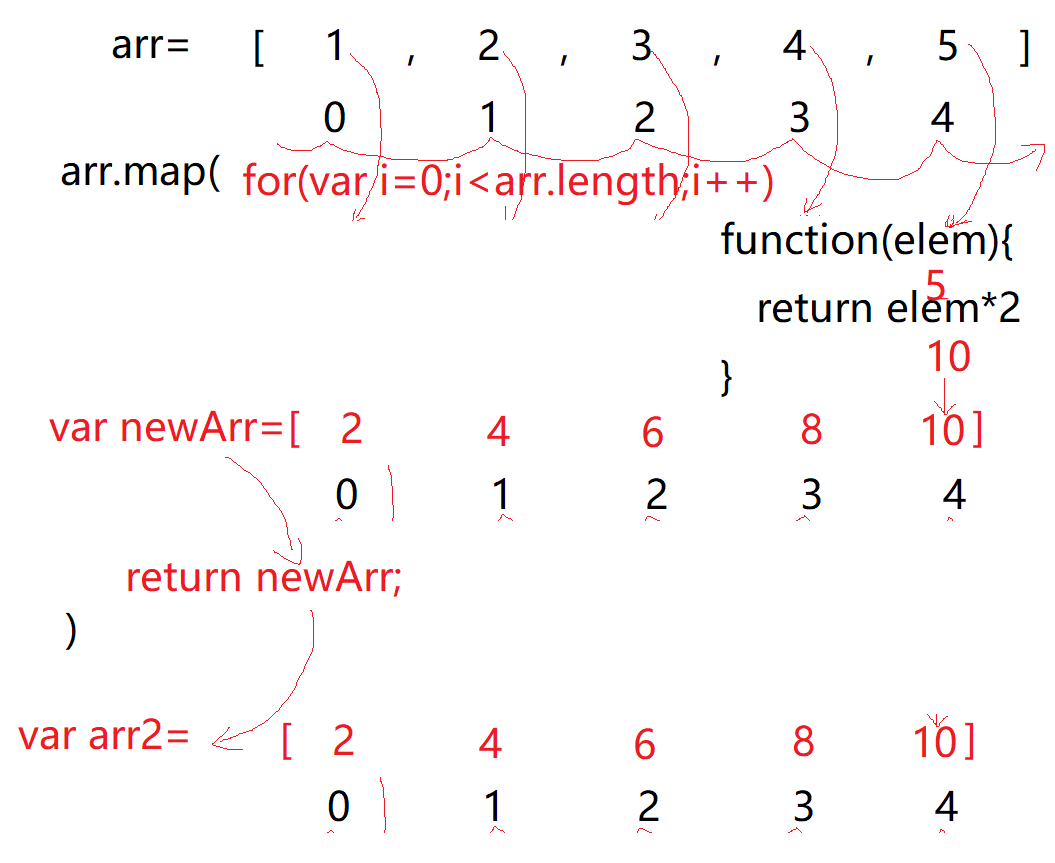
ii. 将当前正在遍历的下标位置传给回调函数第二个形参

iii. 将当前数组对象传给回调函数的第三个形参

5). 在回调函数内部，将当前元素值经过修改后，得到新的元素值，返回出来

6). map接到回调函数返回的新元素之后，自动将新元素值，放入新数组中对应的位置.

7). 遍历结束，将新数组返回出来！



d. 强调: 原数组内容保持不变！而是返回新数组！

e. 示例: 将一个数组中的元素值\*2后，放入新数组返回，原数组不变

2\_map.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      var arr=[1,2,3,4,5];      //想将原数组中每个值\*2,生成新数组，原数组保持不变      var arr2=arr.map(function(elem,i,arr){        console.log(`arr.map()自动调用一次回调函数，elem=${elem}, i=${i}, arr=${arr}, 加工后返回新元素值${elem\*2}，放到新数组中${i}位置`)        return elem\*2      })      console.log(arr2);//2,4,6,8,10      console.log(arr);//1,2,3,4,5    </script>  </body>  </html>  运行结果:  arr.map()自动调用一次回调函数，elem=1, i=0, arr=1,2,3,4,5, 加工后返回新元素值2，放到新数组中0位置  arr.map()自动调用一次回调函数，elem=2, i=1, arr=1,2,3,4,5, 加工后返回新元素值4，放到新数组中1位置  arr.map()自动调用一次回调函数，elem=3, i=2, arr=1,2,3,4,5, 加工后返回新元素值6，放到新数组中2位置  arr.map()自动调用一次回调函数，elem=4, i=3, arr=1,2,3,4,5, 加工后返回新元素值8，放到新数组中3位置  arr.map()自动调用一次回调函数，elem=5, i=4, arr=1,2,3,4,5, 加工后返回新元素值10，放到新数组中4位置  [2, 4, 6, 8, 10]  [1, 2, 3, 4, 5] |

2. 过滤：

对应小程序视频: 小程序->在线->JSCORE->day06 8. filter

(1). 什么是: 选取出原数组中符合条件的元素放入新数组中返回。

(2). 如何: 过滤

var 新数组=原数组.filter(function(当前元素值, 当前位置, 当前数组){

return 判断条件

})

(3). 原理:

a. 先创建一个新的空数组等待。

b. filter内已经自带for循环，自动遍历数组中每个元素

c. 每遍历一个元素，就自动调用一次回调函数

d. 每次调用回调函数时，都会自动传入三个值:

1). 将当前正在遍历的元素值传给回调函数第一个形参

2). 将当前正在遍历的下标位置传给回调函数第二个形参

3). 将当前数组对象传给回调函数的第三个形参

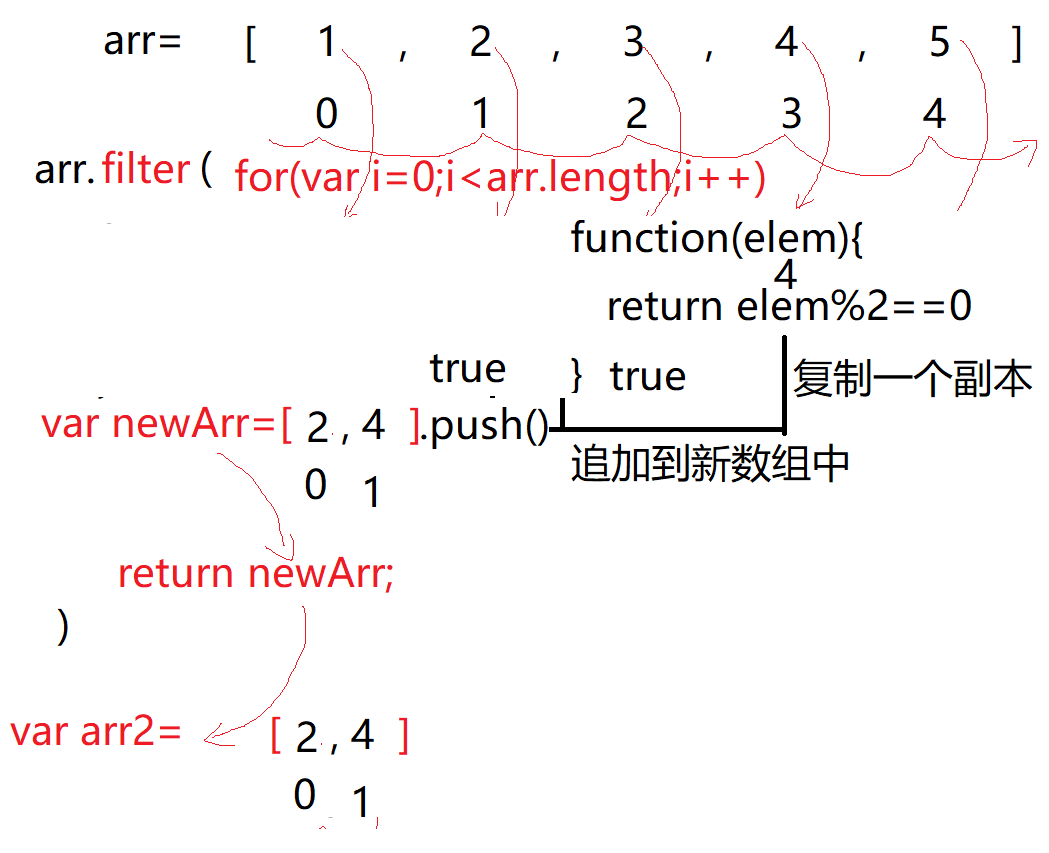
e. 在回调函数内部，根据本次传入的元素值判断当前元素是否符合条件的要求，并返回判断结果

f. filter接到回调函数返回的判断结果之后:

1). 如果当前元素判断结果为true，说明当前元素符合条件，则filter自动将当前元素追加到新数组中保存

2). 如果当前元素判断结果为false，说明当前元素不符合条线，则filter什么也不干，继续遍历下一个元素。

g. 遍历结束，将新数组返回出来！



(4). 强调: filter会把符合条件的元素复制一个副本放入新数组中。原数组始终保持不变！

(5). 示例: 过滤出数组中的偶数，放入新数组返回

3\_filter.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      var arr=[1,2,3,4,5];      //想过滤出数组中的偶数      var arr2=arr.filter(function(elem){        console.log(`arr.filter()自动调用一次回调函数,elem=${elem},返回判断结果:${elem%2==0}`)        //判断当前元素是否是偶数        return elem%2==0;      });      console.log(arr2);//2,4      console.log(arr);//1,2,3,4,5    </script>  </body>  </html>  运行结果:  arr.filter()自动调用一次回调函数,elem=1,返回判断结果:false  arr.filter()自动调用一次回调函数,elem=2,返回判断结果:true  arr.filter()自动调用一次回调函数,elem=3,返回判断结果:false  arr.filter()自动调用一次回调函数,elem=4,返回判断结果:true  arr.filter()自动调用一次回调函数,elem=5,返回判断结果:false  [2, 4]  [1, 2, 3, 4, 5] |

3. 汇总:(仅以累加求和举例)

对应小程序视频: 小程序->在线->JSCORE->day06 9. reduce ...

(1). 什么是: 对数组中所有元素经过统计后得出一个最终的结论

(2). 如何:

减少/汇总

var 结果=数组.reduce(

function(临时汇总值, 当前元素值, 当前位置, 当前数组){

return 将当前元素值汇总到临时汇总值中，计算出新的临时汇总值

},

起始值

)

(3). 原理:

a. 先创建一个新的变量准备保存临时汇总值。这个新变量的初始值为reduce的第二个实参值——起始值。

b. reduce内已经自带for循环，自动遍历数组中每个元素

c. 每遍历一个元素，就自动调用一次回调函数

d. 每次调用回调函数时，都会自动传入四个值:

1). 先将临时汇总值传给回调函数的第一个形参

2). 将当前正在遍历的元素值传给回调函数第二个形参

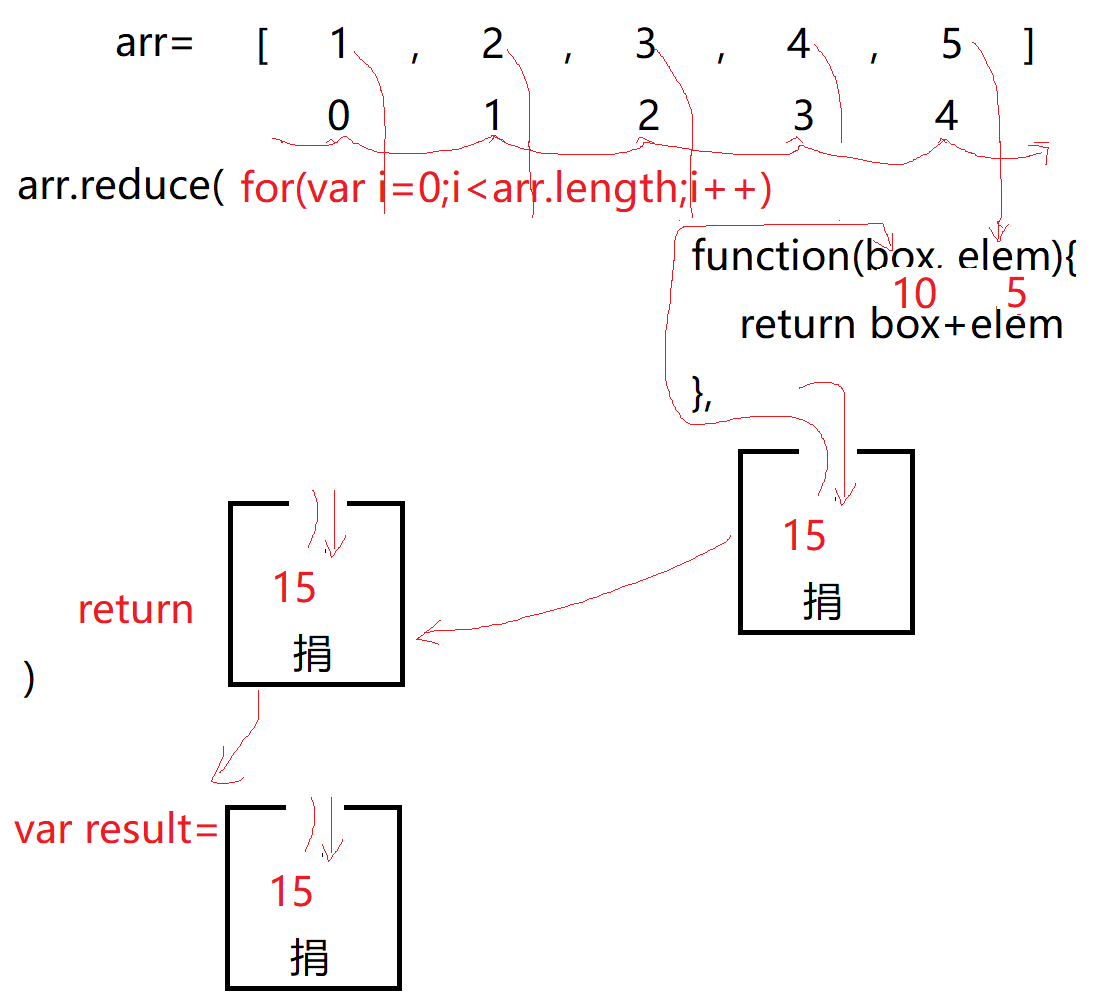
3). 将当前正在遍历的下标位置传给回调函数第三个形参

4). 将当前数组对象传给回调函数的第四个形参

e. 在回调函数内，将当前元素值和临时汇总值相加，计算出新的临时汇总值，并返回新的临时汇总值。

f. reduce接到回调函数返回的新的临时汇总值之后，自动将新的临时汇总值保存到变量中，为继续汇总下一个元素做准备。

g. 遍历结束，将变量中保存的汇总值返回，就是我们最终想要的统计结果。



(4). 示例: 使用reduce统计数组中所有元素的和

4\_reduce.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>  var arr=[1,2,3,4,5];      //想对数组内容求和      var result=arr.reduce(        function(box,elem){          console.log(`arr.reduce()调用一次回调函数，临时汇总值box=${box}, elem=${elem}, box+elem计算出新汇总值:${box+elem}`)          return box+elem;        },        0      );      console.log(result);//15      var arr2=[6,7,8,9,10];      result=arr2.reduce(        function(box,elem){          console.log(`arr2.reduce()调用一次回调函数，临时汇总值box=${box}, elem=${elem}, box+elem计算出新汇总值:${box+elem}`)          return box+elem        },        result      )      console.log(result);//55    </script>  </body>  </html>  运行结果:  arr.reduce()调用一次回调函数，临时汇总值box=0, elem=1, box+elem计算出新汇总值:1  arr.reduce()调用一次回调函数，临时汇总值box=1, elem=2, box+elem计算出新汇总值:3  arr.reduce()调用一次回调函数，临时汇总值box=3, elem=3, box+elem计算出新汇总值:6  arr.reduce()调用一次回调函数，临时汇总值box=6, elem=4, box+elem计算出新汇总值:10  arr.reduce()调用一次回调函数，临时汇总值box=10, elem=5, box+elem计算出新汇总值:15  15  arr2.reduce()调用一次回调函数，临时汇总值box=15, elem=6, box+elem计算出新汇总值:21  arr2.reduce()调用一次回调函数，临时汇总值box=21, elem=7, box+elem计算出新汇总值:28  arr2.reduce()调用一次回调函数，临时汇总值box=28, elem=8, box+elem计算出新汇总值:36  arr2.reduce()调用一次回调函数，临时汇总值box=36, elem=9, box+elem计算出新汇总值:45  arr2.reduce()调用一次回调函数，临时汇总值box=45, elem=10, box+elem计算出新汇总值:55  55 |

\*\*\*\*\*ES6\*\*\*\*\*

一. 模板字符串: 复习

对应小程序视频: 小程序->在线->JSCORE->day06 10. 模板字符串

1. 什么是模板字符串: 支持换行和动态生成内容的特殊字符串

2. 为什么: 因为+本意是算术计算中的加法计算，如果拼接字符串也用+，就会和算术计算的加法混淆或冲突

3. 何时: 今后只要一个字符串的内容包含需要根据变量动态生成的部分，都要用模板字符串代替+来动态生成字符串。

4. 如何:

(1). 整个字符串要用一对儿反引号``包裹

(2). 在``中支持', "", 换行等。

(3). 在``中凡是需要根据变量动态生成的部分内容都要用${}包裹

5. 总结: ${}中能放什么，不能放什么?

(1). ${}中能放什么: 变量、算术计算、三目、调用函数、创建对象、访问数组元素——所有有返回值的合法的js表达式都可以放在${}中

(2). ${}中不能放什么: 程序结构(分支和循环)以及没有返回值的js表达式。

6. 示例: 使用模板字符串动态拼接字符串内容

5\_templateString.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      var uname="dingding";      console.log(`Welcome ${uname}`);      var sex=1;      console.log(`性别: ${sex==1?"男":"女"}`);      var price=12.5;      var count=5;      //.toFixed(2)保留两位小数四舍五入——复习第一阶段      console.log(`      单价: ¥${price.toFixed(2)}      数量: ${count}      ====================      小计: ¥${(price\*count).toFixed(2)}      `);      //Date与日期不会的同学，复习第一阶段Date对象      var orderTime=1602309995261;      console.log(`下单时间: ${new Date(orderTime).toLocaleString()}`);      var arr=[        "日","一","二","三","四","五","六"      // 0    1    2    3    4    5   6      ]      var d=new Date().getDay();      //  6      console.log(`今天星期${arr[d]}`);      //                     六    </script>  </body>  </html>  运行结果:  Welcome dingding  性别: 男  单价: ¥12.50  数量: 5  ====================  小计: ¥62.50    下单时间: 2020/10/10 下午2:06:35  今天星期六 |

二. let

对应小程序视频: 小程序->在线->JSCORE->day06 11. let

1. 问题: 旧js中声明变量都用var，但是var有两个广受诟病的缺陷：

(1). 会被声明提前——打乱了程序正常的执行顺序

(2). 没有块级作用域:

a. 块级作用域: 除了函数以外的if,else,else if,while等分支和循环结构的{}范围。

b. 在旧js中，分支结构和循环结构的{}都不是作用域。

c. 导致{}内的变量，会影响{}外部的变量！

2. 示例: 演示var的两个缺点对程序的影响：

6\_var.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      var t=0;//在全局定义一个变量，累加每个函数的执行时间      function fun1(){        console.log(`fun1执行耗时0.3s`);        t+=0.3;      }      function fun2(){//是作用域，拦住var t;        //var t;//被提前到这里        //结果fun2中从此就有了局部变量t        console.log(`fun2执行耗时0.8s`);        t+=0.8;//0.8会优先加给局部变量t，不再加个全局的t。        //在某次代码更新过程中，无意中添加了以下代码        //不会执行的代码        if(false){//不是作用域！限制不住var t          //var t会被声明提前，t=new Date()留在原地          var t=new Date();          console.log(`今天是${t.toLocaleString()}`);        }      }      fun1();//t+=0.3      fun2();//t+=0.8      console.log(`共耗时${t}s`);//0.3+0.8=1.1    </script>  </body>  </html>  运行结果:  fun1执行耗时0.3s  fun2执行耗时0.8s  共耗时0.3s |

3. 解决: 今后所有变量的声明都要用let代替var。如果是常量的声明，就用const。

4. let的优点: 2个:

(1). 不会被声明提前！

(2). 让分支结构和循环结构的{}也变成一级作用域——块级作用域。块内的变量，不会跑到块外影响外部的程序！

5. 示例: 用let代替var

7\_let.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      var t=0;//在全局定义一个变量，累加每个函数的执行时间      function fun1(){        console.log(`fun1执行耗时0.3s`);        t+=0.3;      }      function fun2(){//是作用域，拦住var t;        console.log(`fun2执行耗时0.8s`);        t+=0.8;//程序执行到这里时，还不知道if(){}块内还有一个t变量，所以这里的t+=0.8，是加到全局变量t上的！        //在某次代码更新过程中，无意中添加了以下代码        //不会执行的代码        if(false){//let让if后的{}也成为一级作用域，就能拦住变量t，不影响外部！          //let不会被声明提前          let t=new Date();          console.log(`今天是${t.toLocaleString()}`);        }      }      fun1();//t+=0.3      fun2();//t+=0.8      console.log(`共耗时${t}s`);//0.3+0.8=1.1    </script>  </body>  </html>  运行结果：  fun1执行耗时0.3s  fun2执行耗时0.8s  共耗时1.1s |

6. 原理: 新瓶装旧酒

(1). 可能在底层自动给重名的变量改名，来避免冲突

(2). 可能在底层自动添加匿名函数自调，来限制变量的提前！

7. 其实: let并没有让分支和循环的{}真的成为块级作用域:

if(...){

let t=...; //会留在原地，不会提前，不会影响外部

var a=...;//依然会被提前，会超出if的范围，影响外部！

}

8. 示例: 通过修改变量名或匿名函数自调来实现和let相同的效果：

8\_function.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      var t=0;//在全局定义一个变量，累加每个函数的执行时间      function fun1(){        console.log(`fun1执行耗时0.3s`);        t+=0.3;      }      function fun2(){//是作用域        //var \_t;//因为变量名不同，所以对t+=0.8        console.log(`fun2执行耗时0.8s`);        t+=0.8;        //在某次代码更新过程中，无意中添加了以下代码        //不会执行的代码        if(false){//不是作用域          //两个办法:          //1. 修改变量名          //var \_t=new Date();          //2. 匿名函数自调方式，划分临时作用域          (function(){//是作用域            var t=new Date();            console.log(`今天是${t.toLocaleString()}`);          })();          var a=10;        }      }      fun1();//t+=0.3      fun2();//t+=0.8      console.log(`共耗时${t}s`);//0.3+0.8=1.1    </script>  </body>  </html>  运行结果：  fun1执行耗时0.3s  fun2执行耗时0.8s  共耗时1.1s |

8. let的小脾气:

(1). 不会被声明提前，所以不允许提前使用let声明的变量！

(2). 在同一作用域内，不能重复let两个同名的变量

(3). 即使在全局用let声明的变量，在window中也是找不到的！

9. 示例: 演示let的小脾气:

9\_let2.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      //var a;      console.log(a);//undefined      var a=10;      //尝试在let b之前使用变量b      //console.log(b);//报错:Cannot access 'b' before initialization      //不能在初始化b变量之前使用b变量      //let b=10;      console.log(window.a);//10      //(function(){      let c=10;      //尝试在let c之后重复let c      //let c=100;//报错: Identifier 'c' has already been declared      //变量c已经被声明过了！      //尝试输出window中的c变量，查看let c是否会添加到全局window对象中。      console.log(window.c);//undefined      //})()    </script>  </body>  </html>  运行结果:  undefined  10  undefined |

三. 箭头函数:

对应小程序视频: 小程序->在线->JSCORE->day07 3. 箭头函数arrow

1. 问题: 旧js中到处都是function！很繁琐

2. 解决: 今后程序中几乎都用箭头函数来简写function。——今后在项目中应该几乎看不到function才对！

3. 如何: 3句话:

(1). 去掉function，在()和{}之间加=>

(2). 如果参数列表只有一个形参，可省略()

(3). 如果函数体只有一句话，可省略{}

如果仅有的一句话函数体是return xxx，就必须省略return

4. 示例: 使用箭头函数简写常用function

10\_function.html —— 未简化时的样子

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      function add(a,b){        return a+b;      }      console.log(add(3,5));//8      var arr=[23,12,123,3,2,1];      arr.sort(function(a,b){return a-b});      console.log(arr);//1,2,3,12,23,123      var str="you can you up";      str=str.replace(/\b[a-z]/ig,function(kw){        return kw.toUpperCase();      });      console.log(str);//You Can You Up      var arr=["亮亮","然然","东东"];      arr.forEach(function(elem){        console.log(`${elem} 到！`)      })        var arr=[1,2,3,4,5];      var arr2=arr.map(function(elem){        return elem\*2;      })      console.log(arr2);//2,4,6,8,10      console.log(arr);//1,2,3,4,5      var arr=[1,2,3,4,5];      var arr2=arr.filter(function(elem){        return elem%2==0      });      console.log(arr2);//2,4      console.log(arr);//1,2,3,4,5      var arr=[1,2,3,4,5];      var result=arr.reduce(function(box,elem){        return box+elem;      },0)      console.log(result);//15      var n=5;      var timer=setInterval(function(){        n--;        if(n==0){          console.log("boom!!!");          clearInterval(timer);        }else{          console.log(n);        }      },1000);      (function(){        var end=new Date();        console.log(`页面加载完成,at:${end.toLocaleString()}`)      })();    </script>  </body>  </html>  运行结果:  8  (6) [1, 2, 3, 12, 23, 123]  You Can You Up  亮亮 到！  然然 到！  东东 到！  (5) [2, 4, 6, 8, 10]  (5) [1, 2, 3, 4, 5]  (2) [2, 4]  (5) [1, 2, 3, 4, 5]  15  页面加载完成,at:2020/10/10 下午4:34:28  4  3  2  1  boom!!! |

11\_arrow.html —— 简化后

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      // function add(a,b){      //   return a+b;      // }      // var add=function(a,b){      //   return a+b;      // }      var add=(a,b)=>a+b;      console.log(add(3,5));//8      var arr=[23,12,123,3,2,1];      arr.sort((a,b)=>a-b);      console.log(arr);//1,2,3,12,23,123      var str="you can you up";      str=str.replace(        /\b[a-z]/ig,kw=>kw.toUpperCase()      );      console.log(str);//You Can You Up      var arr=["亮亮","然然","东东"];      arr.forEach(        elem=>console.log(`${elem} 到！`)      );        var arr=[1,2,3,4,5];      var arr2=arr.map(elem=>elem\*2)      console.log(arr2);//2,4,6,8,10      console.log(arr);//1,2,3,4,5      var arr=[1,2,3,4,5];      var arr2=arr.filter(elem=>elem%2==0);      console.log(arr2);//2,4      console.log(arr);//1,2,3,4,5      var arr=[1,2,3,4,5];      var result=        arr.reduce((box,elem)=>box+elem,0)      console.log(result);//15      var n=5;      var timer=setInterval(()=>{        n--;        if(n==0){          console.log("boom!!!");          clearInterval(timer);        }else{          console.log(n);        }      },1000);      (()=>{        var end=new Date();        console.log(`页面加载完成,at:${end.toLocaleString()}`)      })();    </script>  </body>  </html>  运行结果:  同上 |

5. 箭头函数特点: 箭头函数内外的this可以保持一致！

6. 示例: 使用箭头函数简写回调函数的this，让内外this相同

12\_arrow\_this.html —— 都用function，出错

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      var lilei={        sname:"Li Lei",        friends:["亮亮","然然","东东"],        intr:function(){          //Li Lei认识亮亮          //Li Lei认识然然          //Li Lei认识东东          //遍历friends数组中每个人名          //因为这个this属于intr()，而将来intr()是被lilei.intr()调用的。所以这个this->lilei，是正确的！          this.friends.forEach(            function(elem){              //因为这个this属于回调函数内部，而回调函数定义时前边即没有点，又没有new，且还不是我们自己调用，是forEach()函数自动调用，所以，回调函数中的this默认->window，本例中就是错误的！              console.log(`${this.sname} 认识 ${elem}`)            }          )        }      }      lilei.intr();    </script>  </body>  </html>  运行结果:  undefined 认识 亮亮  undefined 认识 然然  undefined 认识 东东 |

13\_arrow\_this2.html —— 只修改内部的function为箭头函数，结果正确

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      var lilei={        sname:"Li Lei",        friends:["亮亮","然然","东东"],        intr:function(){//作用域          //Li Lei认识亮亮          //Li Lei认识然然          //Li Lei认识东东          //遍历friends数组中每个人名          //因为这个this属于intr()，而将来intr()是被lilei.intr()调用的。所以这个this->lilei，是正确的！          this.friends.forEach(            //箭头函数的特异功能: 让箭头函数内部的this与外部的this保持一致！            //本例中: 内部this->外部的this->lilei            //都正确了！            elem=>console.log(`${this.sname} 认识 ${elem}`)            //function(elem){//作用域              //...this.sname...            //}          )        }      }      lilei.intr();    </script>  </body>  </html>  运行结果:  Li Lei 认识 亮亮  Li Lei 认识 然然  Li Lei 认识 东东 |

7. 何时不能使用箭头函数: 今后如果刚好不希望函数中的this与外部的this相通时，就不能用箭头函数简写！

比如: 对象中的方法就不能用箭头函数简化

8. 示例: 将对象方法也改成箭头函数——出错  
14\_arrow\_this3.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      var lilei={        sname:"Li Lei",        friends:["亮亮","然然","东东"],        intr:()=>{//把墙拆了          //Li Lei认识亮亮          //Li Lei认识然然          //Li Lei认识东东          //遍历friends数组中每个人名          //因为intr()的墙也被拆了，这里的this直接冲出intr()范围。          //又因为var lilei对象的{}，本来就不是作用域，不是墙，所以拦不住this          //所以，外部的this最终直接指向全局window          //导致intr()内两个this都指向window！全错了！          this.friends.forEach(            //因为改成箭头函数，把墙拆了，所以内部的this指向外部的this            elem=>console.log(`${this.sname} 认识 ${elem}`)          )        }      }      lilei.intr();    </script>  </body>  </html>  运行结果:  Uncaught TypeError: Cannot read property 'forEach' of undefined |

9. 问题: 对象的方法如何简写？

10. 解决: ES6中，提供了专门简写对象方法的新语法:

(1). var 对象={

... : ... ,

~~方法名:function(){ ... },~~

方法名(){ ... }

}

(2). 好处: 即不改变原有this的指向，又能省略function。

11. 示例: 使用对象方法简写，即省略function，又不影响this  
15\_arrow\_this\_final.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      var lilei={        sname:"Li Lei",        friends:["亮亮","然然","东东"],        intr(){//即简化了function，又没有改变作用域，也不影响内部的this          //Li Lei认识亮亮          //Li Lei认识然然          //Li Lei认识东东          //遍历friends数组中每个人名          //因为将来intr()被lilei.调用，所以这里的this->lilei          this.friends.forEach(            //因为改成箭头函数，把墙拆了，所以内部的this指向外部的this，都指向lilei了！            //都对了！            elem=>console.log(`${this.sname} 认识 ${elem}`)          )        }      }      lilei.intr();    </script>  </body>  </html>  运行结果:  Li Lei 认识 亮亮  Li Lei 认识 然然  Li Lei 认识 东东 |

四. for of

1. 问题:

(1). 传统for循环:

a. 优点: 可以遍历多种数组、类数组对象、字符串

b. 缺点: 语法是固化的，没有简化的空间

(2). arr.forEach:

a. 优点: 可以进一步简化

b. 缺点: 只能索引数组使用！其它类型不能使用。

2. 解决: 今后只要遍历数字下标的一切，都首选for of

3. 如何:

for(var 变量 of 索引数组/类数组对象/字符串){

//of会依次取出后边的数组或类数组对象中每个元素值

//自动将本次取出的属性值保存到of前的变量里

}

4. 问题: vs 旧for

(1). 旧for: 灵活: 即可控制遍历的方向，又可以控制遍历的步调

(2). for of: 不灵活: 永远只能从头到尾，挨个遍历！

5. vs for in :

(1). in每次取出的是下标，不是属性值

(2). of每次直接取出属性值，直接可用！

6. 示例: 使用for循环, forEach, for of分别遍历数组和类数组对象

16\_for\_of.html

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0">    <title>Document</title>  </head>  <body>    <script>      var arr=["亮亮","然然","东东"];      //for      // for(var i=0;i<arr.length;i++){      //   console.log(`${arr[i]} - 到！`)      // }      //forEach      // arr.forEach(function(elem){      //   console.log(`${elem} - 到！`)      // })      // arr.forEach(elem=>console.log(`${elem} - 到！`));      //for of      for(var sname of arr){        console.log(`${sname} - 到!`)      }        //需求: 定义函数add，可计算任意多个数字的和      function add(           ){      // arguments[           ].length      //            0  1  2 ...        var total=0;        //for        // for(var i=0;i<arguments.length;i++){        //   total+=arguments[i];        // }        //forEach —— 错        // arguments.forEach(function(elem){        //   total+=elem        // })//报错: arguments.forEach is not a function        //for of        for(var num of arguments){          total+=num        }        return total;      }      console.log(add(1,2,3));//6      console.log(add(1,2,3,4,5));//15    </script>  </body>  </html>  运行结果:  亮亮 - 到!  然然 - 到!  东东 - 到!  6  15 |

总结:

6. 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)

7. 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,

... :...

},

... : { ... }

})

8. 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, ...)

9. 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)

10. 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

(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. 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东西，都用for of

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | for(var i=0; i<arr.length; i++) | 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 | × | × | × | √ |

总结: this 4种:

1. obj.fun() this->点前的obj对象 比如， lilei.intr() this->lilei

2. new Fun() this->new正在创建的新对象

3. 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

4. fun() 和回调函数 和匿名函数自调 this->默认指window

今日对应小程序视频列表:

小程序->在线->JSCORE->day06

5. forEach ...

7. map ...

8. filter

9. reduce ...

10. 模板字符串

11. let

小程序->在线->JSCORE->day07

3. 箭头函数arrow

4. for of

作业：

1. 复习今日问题清单：小程序->首页->JSCORE->day06,07

2. (必须)看小程序视频学习ES6中关于对象属性和方法的简写  
 小程序->在线->JSCORE->day07 作业: ES6中关于对象中属性和方法的简写

3. (必须)看小程序视频学习如何使用回调函数让多个异步函数顺序执行

小程序->在线->JSCORE->day07 作业: 如何使用回调函数让多个异步函数顺序执行

4. 预习小程序->在线->JSCORE->day07,day08内容