小程序: WEB问题速查

1. 问题清单: 历届师兄师姐在今天课程中集中问过的高频的重点的问题！

那些自己发现不了问题，又迫切想知道其它同学都问了什么问题，的同学

小程序->首页->JSCORE->day02、day03

当老师在回答其它同学的问题时，你不要闲着！你可以打开问题清单浏览别人曾经问过的经典问题！

2. 高清无码小视频: 小程序->在线->JSCORE->day02、day03

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

万一家里或中心网络断了！暂时看不了直播了！

因为某些事儿请假了，错过了当天的直播！

课下复习时，觉得TMOOC视频太大，浏览不方便

——打开小程序中"在线"，就可下载观看当前所讲内容和当天所讲全部内容的回放。

基础比较好的，确实学有余力的同学，想拿高薪的同学，在我答疑时，不要闲着:

要看今天扩展视频: 小程序->在线->JSCORE->day03

3. 声明提前完整笔试题

作业: 作用域 笔试题

作业: 复习 高频笔试题 按值传递

作业: 闭包 高频笔试题 funs

作业: 闭包 高频笔试题 nAd getN

3. 单词列表: 小程序->单词->JSCORE->day02、day03

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. 凡是拼写错误，自己懒得找！我可以帮你找，但是找到后，错误的单词你要手抄20遍，并拍照！

3. 每个带注释的案例至少写三遍:

(1). 注释+代码抄一遍

(2). 保留注释，删除代码，自己试着把代码翻译回来！

(3). 删除注释和代码，自己试着用自己的话把注释写回来！

一个例子只有自己可以把人话的注释写回来！才算自己会！

问题: 一听就会一做就废！原因: 你自己无法用人话精确描述自己想干什么！

如何提问:

1. 上课时，案例中，笔记中那句话哪行代码看不懂，立刻问！

2. 小程序中问题清单里哪个问题和回答看不懂！立刻问！

3. 个人项目中哪个功能不会做！立刻问！

4. 刷笔试题时，遇到不会的笔试题！立刻问！

正课: 从今天开始未来6天是js最核心的6天内功心法——自己画图！！！

1. RegExp对象

\*\*\*\*Function\*\*\*\*

1. 回顾函数

2. 创建函数三种方式

3. 重载

4. 匿名函数

5. 作用域和作用域链  
6. \*\*\*闭包\*\*\*

一. RegExp对象: (Regular Expression 规则表达式)

1. 什么是正则表达式对象: 内存中专门保存一条正则表达式，并提供用正则表达式执行查找和验证的函数 的对象

2. 为什么要有正则表达式对象: 因为正则表达式的语法不是js语法。js不认识正则表达式。所以，需要一种专门的对象让js认出正则表达式来。

3. 何时需要正则表达式对象: 今后只要在js中使用正则表达式都需要先创建正则表达式对象

4. 如何创建正则表达式对象: 2种

对应小程序视频: 小程序->在线->JSCORE->day02 5. new RegExp() 动态生成正则表达式

(1). 标准，用new创建:

a. var reg=new RegExp("正则","ig")

b. 强调: 一定要用""，~~而不是//~~

(2). 简写，用//创建

a. var reg=/正则/ig

(3). 问题: 正则表达式不是固定不变的，可能需要根据其他内容动态生成！

(4). 错误解决: 在//中直接写生成正则表达式的js程序语句

原因: //中是正则表达式的地盘，不认识js程序，所以写了也没用。

(5). 正确解决: 今后

a. 如果正则表达式是固定的，首选//

b. 如果正则表达式不是固定的，需要动态生成， 首选new RegExp(js表达式, "ig")

(6). 示例: 根据服务器端返回的敏感词数组，防守敏感词

1\_regexp.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>      //假设从服务器端用ajax请求过来一个敏感词数组      //要求: 将来数组里有什么词，我们的程序就要防住什么词！      var arr=["青天","紫烟"];      //问题: 我们不可能提前预知从服务器返回的数组中包含哪些词！      //解决: 将arr数组中每个关键词用|相连拼接成一个字符串      console.log(arr.join("|"));      //          青天|紫烟        //我们的前端js程序      //请用户输入一条消息内容      var msg=prompt("请输入消息内容");      //先定义正则表达式对象      //问题: 自然也就不可能提前在前端程序里写死正则！      //错误的解决: 将动态生成正则表达式的js语句，直接放入//中      //原因: //中是正则表达式的地盘！//中不认识js语句！      //var reg=/arr.join("|")/;      //正确的解决: 被迫使用new RegExp("正则","ig")      //原因: new RegExp()的参数刚好需要一个字符串！而我们有无数种方法可以拼出我们想要的任意字符串！      var reg=new RegExp(arr.join("|"));      //                     ||      //      new RegExp("青天|紫烟");      //       ||      //var reg=/青天|紫烟/;      console.log(reg);      //用正则表达式对象查找消息内容中是否包含敏感词      var i=msg.search(reg);      //如果包含敏感词      if(i!=-1){        //就提示红字: 包含敏感词禁止发送        document.write(`<h1 style="color:red">包含敏感词，禁止发送</h1>`)      }else{//否则        //就显示: 然哥说: 消息内容        document.write(`<h1 style="color:green">然哥说:${msg}</h1>`)      }    </script>  </body>  </html>  运行结果： |

5. 验证字符串格式:

(1). var bool=reg.test(str)

(2). 意为: 用正则表达式reg，验证字符串str是否符合格式要求

(3). 返回值:

a. 如果字符串str符合正则表达式reg的格式要求，就返回true

b. 如果字符串str不符合正则表达式reg的格式要求，就返回false

(4). 坑: test()，默认不要求从头到尾完全匹配。只要在字符串中可以找到部分内容与正则匹配，就返回true！

比如: /\d{6}/.test("abc123456") true

(5). 解决: 今后只要验证，都要在正则前加^后加$，表示从头到尾必须完整匹配

^ $

| \d{6} |

比如: /^\d{6}$/.test("abc123456") false

(6). 示例: 验证密码格式是否符合要求

2\_test.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>      //请用户输入一个密码，要求必须是6位数字      var pwd=prompt("请输入密码");      //定义正则表达式，匹配6位数字      var reg=/^\d{6}$/;      //用     正则 验证 密码的格式是否符合要求      //        ↓    ↓   ↓      var bool=reg.test(pwd);      if(bool==true){//如果验证通过        //就提示密码格式正确        document.write(`<h1 style="color:green">密码格式正确</h1>`)      }else{//否则如果验证不通过        //就提示密码格式不正确        document.write(`<h1 style="color:red">密码格式不正确</h1>`)      }    </script>  </body>  </html>  运行结果: |

6. 查找每个敏感词的内容和位置:

对应小程序视频: 小程序->在线->JSCORE->day02 6. 查找敏感词 既查找每个敏感词的内容...

(1). 问题: match()不加g，即可找内容，又可找位置，但是不能找所有；match()加g，虽然可以找所有，但是只能获得内容，无法获得位置——鱼和熊掌不可兼得！

(2). 解决: var arr=reg.exec(str)

意为查找str中第一个敏感词的内容和位置，保存在一个数组中返回

(3). 只使用一次: 和match不加g的情况是一样的！

a. 也只找第一个敏感词

b. 也返回一个数组: [ 0:"敏感词内容", index:敏感词位置 ]

(4). 如果希望用reg.exec()找所有敏感词的内容和位置:

a. 正则必须加g！

b. 反复调用reg.exec(str)，直到reg.exec()返回null为止——用循环

reg.exec()聪明: 自动知道要找下一个位置的敏感词！

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);

(5). 示例: 使用reg.exec()查找一句话中每个敏感词的内容和位置:

3\_exec.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 str="老师:请用小红 我的 朋友造句。小亮:小红是我的朋友。小然:朋友！小红是我的！";      //只找第一个敏感词: 不要加g      // var reg=/小[\u4e00-\u9fa5]/;      // //先用match找:      // var arr=str.match(reg);      // console.log(arr);      // //再用reg.exec找:      // var arr=reg.exec(str);      // console.log(arr);      //想找所有: 加g      var reg=/小[\u4e00-\u9fa5]/g;      //先用match找:      var arr=str.match(reg);      console.log(arr);      //再reg.exec()      //反复执行，只要本次找到了敏感词(arr!=null)，就有必要继续查找下一个      //do while: 因为即使不确定字符串中有没有敏感词，也必须至少查找一次时，首选do while。      do{        var arr=reg.exec(str);//返回值=match()不加g时的返回值        //arr:[ 0:"敏感词内容", index:敏感词位置]        if(arr!=null){          //console.log(arr);          console.log(`在位置${arr.index}，发现敏感词 ${arr[0]}`)        }      }while(arr!=null);    </script>  </body>  </html>  运行结果:  (5) ["小红", "小亮", "小红", "小然", "小红"]  在位置5，发现敏感词 小红  在位置16，发现敏感词 小亮  在位置19，发现敏感词 小红  在位置27，发现敏感词 小然  在位置33，发现敏感词 小红 |

\*\*\*\*Function\*\*\*\*

对应小程序视频: 小程序->在线->JSCORE->day03 1. 回顾 定义函数 调用函数 传参...

一. 回顾函数:

1. 什么是函数: 程序中专门保存一段可重用的代码片段的程序结构，再起一个名字。

2. 为什么: 重用代码

3. 何时: 只要一段代码可能被反复使用，都要先将这段代码保存在函数中。然后再反复使用函数

4. 创建函数:

function 函数名(形参变量列表){

函数体;

return 返回值;

}

5. 调用函数: var 变量=函数名(实参值列表)

6. 形参变量:

(1). 什么是: 专门接收函数执行时必需的数据的变量(虽然没有var)

(2). 为什么: 有些函数执行时，必须传入必要的数据，才能正常执行！

(3). 何时: 今后，只要一个函数执行时，必须外界传入某些数据才能正常执行时，都要定义形参变量

7. 返回值:

(1). 什么是: 一个函数的执行结果，被返回到函数外部

(2). 何时: 今后，只要函数的调用者需要获得函数的执行结果时，都要定义函数的返回值。

8. 示例: 定义一个会做饭的函数，可以接收各种食材，做饭

4\_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>      function cook(菜,蛋,面){        console.log(`炒${蛋}`)        console.log(`炒${菜}`)        console.log(`${菜}和${蛋}一起炒`)        console.log(`煮${面}`)        return `香喷喷的${菜}${蛋}${面}`      }      //周一      var 碗=cook("西红柿","鸡蛋","面");      console.log(碗);      //周二: 西红柿没了，只有黄瓜；鸡蛋没了，只有辣条；面没了，只有粉丝      var 碗=cook("黄瓜","辣条","粉丝");      console.log(碗);    </script>  </body>  </html>  运行结果:  炒鸡蛋  炒西红柿  西红柿和鸡蛋一起炒  煮面  香喷喷的西红柿鸡蛋面  炒辣条  炒黄瓜  黄瓜和辣条一起炒  煮粉丝  香喷喷的黄瓜辣条粉丝 |

对应小程序视频：小程序->在线->JSCORE->day03 2. 创建函数三种方式 声明提前

二. 创建函数: 3种方法：

1. 用声明方式创建:

(1). function 函数名(形参变量列表){

函数体

return 返回值

}

(2). 问题: 会被声明提前:

a. 什么是声明提前:

1). 在程序开始执行前

2). 程序会先将var声明的变量和function声明的函数

3). 提前到当前作用域的顶部集中创建

4). 赋值留在原地！

b. 缺点: 打乱程序正常的执行顺序！——JS语言广受诟病的缺陷！

2. 用赋值方式创建:

(1). var 函数名变量=function(形参变量列表){ 函数体; return 返回值 }

(2). 优点: 函数不会被声明提前，保证了程序的正常执行顺序——很多大师级人物都用赋值方式创建函数。

(3). 揭示:

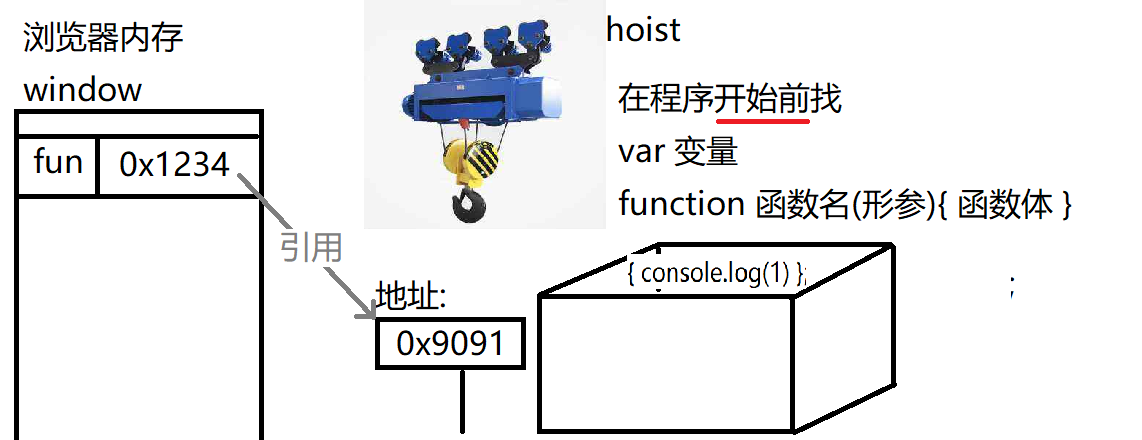
a. js中函数其实也是一个普通的对象，只不过对象中保存的是代码段不是数据而已。

b. 函数名其实就是一个很普通的变量

c. 函数名变量通过保存函数对象的唯一地址值，"引用"着函数对象

d. 当调用函数时，程序先找到函数名变量->变量中的地址->保存在地址位置的函数对象->执行函数对象中保存代码段！

c. 函数在内存中到底如何存储的



(4). 示例: 鄙视题: 判断声明提前后，程序的输出结果:

5\_hoist.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>      //鄙视题:      // function fun(){ console.log(1) };      // fun(); //2      // function fun(){ console.log(2) };      // fun(); //2      var fun=function(){ console.log(1) };      fun(); //1      var fun=function(){ console.log(2) };      fun(); //2    </script>  </body>  </html>  运行结果：  1  2 |

3. 用new 创建:

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

(2). 很少用！

(3). 但是揭示了: function 函数名(){ ... } //被声明提前

↓

相当于var 函数名=function(){ ... }

↓

相当于var 函数名=new Function(...)

(4). 结论: function 底层等效于 new Function()

function的本质是创建一个函数对象

对应小程序视频: 小程序->在线->JSCORE->day03 4. 重载 overload pay

三. 重载: overload

1. 问题: 一件事，根据传入实参值的不同，动态执行不同的逻辑。

2. 不好的做法: 为同一件事，每种不同的逻辑，单独定义一个函数，起一个专门的函数名

原因: 增加了函数名和单词的数量，不好记忆，也不便于使用！

3. 好的解决方法——重载: 相同函数名，不同参数列表的多个函数，在调用时，可根据传入的实参值不同，自动选择匹配的一个函数执行！

4. 何时: 今后只要根据传入的参数值不同，自动执行不同的逻辑时，都要用重载

5. 优点: 减少了函数名和单词的个数，便于记忆和使用！

6. 如何:

(1). 问题: js语言不支持传统的重载方式——多个同名函数

因为: js中多个同名函数，只有最后一个函数会覆盖之前所有同名函数而存活下来

(2). 解决: 借助于arguments对象来变向实现重载效果

a. 什么是arguments:

1). 每个函数中自带的——不用自己创建，就可直接使用

2). 在调用函数时，能自动接收所有传入函数的实参值的——内容

3). 类数组对象——存储结果

i. 什么是类数组对象: 长得像数组的对象

ii. vs 数组:

相同点: 1. 下标, 2. .length, 3. for循环遍历

不同点: 类型不同！类数组对象不是数组家孩子！而是Object家的孩子。所以，类数组对象无权使用数组家的函数！——不是一家人不进一家门！

b. 如何用arguments实现重载:

1). 只定义一个函数，只起一个统一的名字，不包含任何形参变量，函数内包含所有可能的逻辑

2). 根据arguments在调用时自动接住的实参值不同，可以在函数内动态判断决定本次该执行哪种逻辑.

7. 问题: 既然js中一个函数不定义形参，照样可以传实参值，那么今后是不是都不用定义形参了？

答: 不是! 自定义形参变量和arguments[i]

自定义形参变量: 简单, 见名知义，可以告诉调用者如何正确使用函数正确传参

——今后依然首选自定义形参变量

只有参数个数不确定时，才被迫使用arguments代替！

8. 示例: 使用重载实现一个pay函数，支持三种支付方式

6\_overload.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>      //想定义一组付款函数，支持三种付款方式: 手机支付, 现金支付，刷卡支付      //手机支付: 不需要给收款员什么东西，而是自己扫二维码      //现金支付: 需要给收款员一笔现金      //刷卡支付: 需要给收款员银行卡和密码两个数据      //都是  "支付"      function pay(                     ){      // arguments{                     }.length=2      //              0            1        //如果没有传入实参就手机支付        if(arguments.length==0){          console.log(`手机支付...`)        }else if(arguments.length==1){          //否则如果传入了1个实参就现金支付          console.log(`现金支付...收您${arguments[0]}元`)        }else{//否则（暂时只考虑三种情况）          console.log(`刷卡支付...从您卡号${arguments[0]}扣款成功！`)        }      }      //调用时:      //其它:      pay(); //0个实参      pay(100); //1个实参      pay("6553 1234","123456"); //2个实参    </script>  </body>  </html>  运行结果:  手机支付...  现金支付...收您100元  刷卡支付...从您卡号6553 1234扣款成功！ |

对应小程序视频: 小程序->在线->JSCORE->day03 4. 重载overload pay

四. 匿名函数:

1. 什么是: 定义函数时，不指定函数名的函数

2. 为什么: 节约内存，划分临时作用域

3. 何时: 2种情况:

(1). 如果一个函数只使用一次时，都首选匿名函数。

(2). 如果希望划分临时作用域，避免不同功能之间变量互相污染时

4. 如何: 2种情况:

(1). 绝大多数回调函数，都使用匿名函数——节约内存

a. 因为: 匿名函数用完后，自动释放！

b. 反例: 如果不用匿名函数，回调函数用完无法释放，浪费内存！

(2). 匿名函数自调:

a. 什么是匿名函数自调: 定义一个匿名函数后，立刻调用自己！

b. 何时: 今后几乎所有js代码都要放在匿名函数自调中！

c. 为什么: 避免使用全局变量，造成全局污染！——今后在公司中，禁止使用全局变量！

d. 如何:

(function(){

要执行的任务

})() //创建匿名函数后，立刻执行！

结果: 匿名函数内用过的局部变量不会遗留在全局，而是随匿名函数一起释放了！

强调: 匿名函数包与不包，对功能是没有影响的！但是，对内存影响很大！

e. 示例: 使用匿名函数自调禁止产生全局变量

7\_anony.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>      //不希望start变量在完事后，依然留在内存中！      (function(){        //获得当前系统时间        var start=new Date(); //局部变量        //提示: 开始加载页面内容        alert(`开始加载页面内容, at:${start.toLocaleString()}`)      })();    </script>    <script>      //不希望end变量在使用后依然遗留在内存      (function(){        //获得当前系统时间        var end=new Date(); //局部变量        //提示:页面内容加载完成        alert(`页面内容加载完成, at:${end.toLocaleString()}`);      })();      //变量start和end，在弹出提示后，就不会再使用了！      //但是，因为start和end是全局变量，所以永久的留在window中占用内存空间，全局污染！      console.log(window);    </script>  </body>  </html>  运行结果:      f12 console中，展开window，找不到start变量和end变量，说明释放了 |

总结: 以下内容都不用背！

1. 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",从哪个位置开始找)

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

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)

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

d. 查找每个敏感词的内容和位置: reg.exec()

(2). 替换敏感词: 2种

a. 将所有敏感词都替换为统一的新值: 变量=str.replace(/正则/ig, "新值")

b. 根据每次找到的敏感词不同动态替换成不同的新值:

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. 切割符是固定不变的: var arr=str.split("切割符")

b. 切割符不是固定不变的: var arr=str.split(/正则/i)

c. 打散字符串为字符数组: var arr=str.split("")

|  |
| --- |
| 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，如果找不到，都返回-1  2. 如果原函数返回的是一个数组或一个对象，如果找不到，都返回null  3. （未完...待续...） |

|  |
| --- |
| 数组本质: js中所有数组以及对象底层都是关联数组！本质上是没有索引数组！  (1). 数组和对象底层，其实都是名值对儿的集合  (2). 每个数组的下标和每对象属性的属性名，都是字符串类型  (3). 无论访问数组的元素，还是访问对象的属性，标准写法都是["下标名称"]  (4). 简写:  a. 如果下标名称不是数字，可简写为".下标名称"  b. 如果下标是数字，只能简写为"[下标]" |

2. 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)

(3). 既查找每个关键词的内容又查找每个关键词的位置:

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);

3. 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(){ ... }，底层等效于 var 函数名=new Function(...) |

(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。

今日对应小程序视频:

小程序->在线->JSCORE->day02 5. new RegExp() 动态生成正则表达式

6. 查找敏感词 既查找每个敏感词的内容，又查找...

小程序->在线->JSCORE->day03 1. 回顾 定义函数 调用函数 传参...

2. 创建函数三种方式 声明提前

4. 重载 overload pay

作业:

1. 复习今日小程序问题清单（小程序->在线->JSCORE->day02、day03）

2. 定义函数add，可计算任意多个数的和

function add(){

//在这里补齐缺少的代码

}

//测试代码:

console.log(add(1,2,3));//6

console.log(add(1,2,3,4,5));//15

答案: 小程序->首页->搜: add

结果中: 作业: 重载 定义一个函数add...

3. 看小程序视频复习按值传递

小程序->在线->JSCORE->day03 6. 简单 作用域 按值传递

4. 看小程序视频预习明日难点内容：

小程序->在线->JSCORE->day03

7. 原理 作用域和作用域链

8. 闭包 closure 压岁钱

9. 关于闭包的比喻(三个小故事)