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->day08

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->day07、day08

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，浏览不方便或TMOOC不能看了！

——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->day08

4. promise 状态 错误处理

5. promise 传参

6. Promise.all

作业: 高频鄙视题 封装支持Promise的ajax函数，向服务器发送请求，按类型检索商品数量...

是否高频: 看我的朋友圈！历届拿到高薪的师兄师姐，反馈回来的鄙视面试重点内容！

有些有心的同学，已经把朋友圈里反复出现的高频词总结出来！

3. 单词列表: 小程序->单词->JSCORE->day07、day08

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！

出错: 先开f12，console，看错误提示！

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: （vip待遇的剩余时间12天）

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！

谁问的问题越多！谁的培训费交的越值！

正课: ES6

1. 解构

2. class

3. Promise

一. 解构(destruct): 从一个大的数组或对象中仅提取出个别想要的值，单独使用！3种:

小程序->在线->JSCORE->day07 6. 解构(上) destruct 数组解构 对象解构

1. 数组解构: 从一个数组中，仅提取出想要的个别元素单独使用

(1). [变量1, 变量2 ,...]=数组

0 1 ...

(2). 说明=左边不是真正的数组，仅仅是装扮成和等号右边相同的数组的样子。=一旦发现左右两边的结构是一样的，就会将右边数组中相同位置的元素值自动复制给=左边相同位置的变量

(3). 结果: 变量1=数组[0]——数组0位置的元素

变量2=数组[1]——数组1位置的元素

(4). 简写:

var 变量1, 变量2;

[变量1, 变量2]=arr

可简写为: var [变量1, 变量2]=arr;

(5). 解构不连续位置上的元素值:

[变量1, , 变量2, ...]=arr

0 1 2

结果: 变量1=arr[0]; 变量2=arr[1];

(6). 按值传递: 解构也是给变量赋值，同样遵守按值传递

a. 如果数组中保存的是原始类型的值，则解构时，变量得到的是值的副本，将来修改变量值，不会影响原数组中原始值。

b. 如果数组中保存的是引用类型的值，则解构时，变量得到的是元素对象的地址值，将来通过变量修改对象内容时，同样影响原数组中元素对象。

(7). 示例: 使用数组解构，解构出数组中对应位置的元素值，单独使用

1\_destruct\_arr.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 products=[        {pname:"小米",price:3488},        {pname:"华为",price:5488},        {pname:"苹果",price:8488},        {pname:"三星",price:6488},        {pname:"VIVO",price:2488},        {pname:"OPPO",price:1488}      ];      //只想用前三个商品对象，暂时不用后两个商品对象:    //var p1,p2,p3;      var [p1,p2,p3]=products;      console.log(p1); //地址      p1.price-=1000; //修改变量引用的对象      console.log(products[0]); //跟着改变      console.log(p2);      console.log(p3);      var arr=[1,2,3,4,5,6];    //var a,b,c;      var [a,b,c]=arr;      console.log(a);//1 //副本      a-=1; //修改变量      console.log(arr[0]);//保持不变      //如果向要第1个，第3个，第6个商品    //var p1,p3,p6;      //错误:    //[p1,p3,p6]=products;    // 0  1  2   0  1  2      //正确:    var [p1, ,p3, , ,p6]=products;    //   0  1 2  3 4 5   0    2   5    //2件事:    // 1. 先声明三个变量    // 2. 再从数组解构      console.log(p1);      console.log(p3);      console.log(p6);    </script>  </body>  </html>  运行结果:  {pname: "小米", price: 3488}  {pname: "小米", price: 2488}  {pname: "华为", price: 5488}  {pname: "苹果", price: 8488}  1  1  {pname: "小米", price: 2488}  {pname: "苹果", price: 8488}  {pname: "OPPO", price: 1488} |

2. 对象解构：从一个大的对象中仅提取出个别属性值和方法，单独使用！

(1). 如何:

var 变量1, 变量2;

({属性名1:变量1, 属性名2: 变量2, ...}=对象)

(2). 强调: 如果变量已经存在，只需要解构，必须在外围用一个()包裹起来！

(3). 说明: =左边不是一个真正的对象，只是装扮成和=右边对象相同结构。以此来骗取=右边对象的信任，从而顺利拿到=右边对象中同名属性的属性值，保存在=左边的变量里。

(4). 结果: 变量1=对象.属性1

变量2=对象.属性2

(5). 简写: 也可以同时声明变量并解构

var {属性名1:变量1, 属性名2:变量2}=对象

(6). 问题: 因为通常原对象中的属性和方法起名已经很好用了，解构时使用的变量希望沿用原属性名和方法名，就会出现写两遍属性名和方法名的情况：

var {uname:uname, logout:logout }=对象

(7). 简写: 如果希望沿用原属性名和方法名作为解构后的变量名，不改名，则可以只写一个名字:

var {uname, logout}=对象

一个名字两用：既当做配对的属性名，又当做将来单独使用的变量名

(8). 强调: 内部用到this的函数，不能解构。

原理: 解构后，方法名单独使用，调用时，前边什么都没有。方法中的this->window了！不再指向方法所在的对象。this.属性名立刻返回undefined，出错。

(9). 示例: 使用对象解构提取出对象中个别属性和方法:

2\_destruct\_obj.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 user={        uname:"dingding",        nicheng:"东东",        upwd:"123456",        login(){ //昨天作业第二个: login:function(){          console.log(`注册...`)        },        logout(){          console.log(`注销...`)          //通常带this的函数不能解构          //console.log(`注销${this.nicheng}成功...`)        }      }      //只想用uname属性值和logout方法      //    属性名:变量,属性名:变量      //var {uname:un, logout:lo}=user;      // console.log(un);      // lo();      //一般原对象中属性和方法名命名已经很好了，没必要改名      //   属性名:变量 , 属性名:变量      //var {uname:uname, logout:logout}=user;      //简写: 当就打算使用原属性名和方法名作为变量名——不改名，其实只写一个名字即可      // var uname,logout;      // ({ uname, logout }=user);      var {uname,logout}=user;      //一个名字两用: 既当做配对的属性名，又当做将来单独使用的变量名！      console.log(uname);      logout();    </script>  </body>  </html>  运行结果:  dingding  注销... |

3. 参数解构

小程序->在线->JSCORE->day08 1. 解构(下) destruct 参数解构

(1). 问题: 今后如果一个函数多个参数不确定有没有值，但是又要求实参值必须传给指定的形参变量！

(2). 错误的解决: 单靠参数默认值，无法解决如此复杂的问题！

(3). 正确的解决: 参数解构

(4). 如何: 2步:

a. 定义函数时，就要把形参列表装扮为对象的结构

function 函数名({

// 配对 : 接收实参值

属性名1: 形参1="默认值1",

属性名2: 形参2="默认值2",

... : ...

}){

函数体

}

b. 调用函数时，实参值列表，也应该放在一个相同结构的对象中传入

函数名({

属性名: 实参值,

... : ...

})

说明1: 实参值对象结构，不一定包含形参对象结构中所有属性，可以任选任意一个属性传值！

说明2: 实参值对象结构中的:前的属性名不能随意修改，因为必须和定义函数时形参列表对象中的:前的属性名配对，才能成功传入实参值。

(5). 结果: 调用时：

a. 相同属性名对应的实参值，会自动传给定义函数时相同属性名对应的形参变量！

b. 没有提供对应属性名的实参值，则相同属性名的形参变量就收到undefined，从而启用参数默认值。

(6). 简写: 定义函数时，通常形参属性名和形参变量名都是相同相同的。所以其实，定义函数时，可以只写一个形参名。一个名字两用：既当做配对的属性名，又当做形参变量名.

function 函数名({

属性名也是形参名="默认值1",

... =...

}){

函数体

}

(7). 示例: 使用参数解构实现点套餐的函数，可以满足各种点餐换餐需求  
 3\_destruct\_params.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 order({      //属性名 : 形参      // 配对    接实参值      // zhushi:zhushi="香辣鸡腿堡",      // xiaochi:xiaochi="薯条",      // yinliao:yinliao="可乐"      // }){      //一个名字两用: 既当做配对的属性名，又当做形参变量名！      function order({zhushi="香辣鸡腿堡",xiaochi="薯条",yinliao="可乐"}){        console.log(`您点的套餐是:          主食:${zhushi},          小吃:${xiaochi},          饮料:${yinliao}        `)      }      //第一个人着急赶火车，希望不用更换任何菜品，直接点套餐默认菜品      order({});        //第二个人不着急，每个菜品都想换:      order({        zhushi:"奥尔良烤腿堡",        xiaochi:"菠萝派",        yinliao:"咖啡"      });        //第三个人只想换饮料，其余两个保持不变      order({        yinliao:"豆浆"      })        //第四个人只想换小吃，其余两个保持不变      order({        xiaochi:"土豆泥"      })      //第五个人想换主食和饮料      order({        zhushi:"巨无霸",        yinliao:"奶茶"      })      </script>  </body>  </html> |

二. class:

对应小程序视频: 小程序->在线->JSCORE->day08 2. class extends

1. 问题: 旧js中，定义一种类型，需要定义构造函数和原型对象。但是，构造函数和原型对象是分开写的！不像一家人！不符合封装的要求！

2. 解决: 今后只要定义一种类型都要用class{}包裹构造函数和原型对象方法

3. 如何: 3句话:

(1). 用class{}包裹构造函数和原型对象方法

(2). 将构造函数名放在class之后，作为整个类型的名字。今后所有构造函数名统一改名为"constructor"

(3). 所有原型对象方法，无需再加"类型.prototype"前缀和"=function"。今后只要直接放在class内的函数，默认就是原型对象中的共有方法！

说明: class内构造函数和方法之间，不用逗号和分号分隔！

4. 结果: 除了写代码时，写法简化之外，底层的原理与旧js完全相同！——新瓶装旧酒

5. 示例: 使用class，简化创建学生类型

4\_class.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>      class Student{        constructor(sname,sage){          this.sname=sname;          this.sage=sage;        }        //默认就是放在原型对象中的        intr(){          console.log(`I'm ${this.sname},I'm ${this.sage}`)        }      }      var lilei=new Student("Li Lei",11);      console.log(lilei);      lilei.intr();    </script>  </body>  </html>  运行结果:  Student {sname: "Li Lei", sage: 11}  sage: 11  sname: "Li Lei"  \_\_proto\_\_:  constructor: class Student  intr: ƒ intr()  \_\_proto\_\_: Object  I'm Li Lei,I'm 11 |

6. 共有属性值:

(1). 错误的做法1: 在构造函数中写死一个属性值

原因: 凡是在构造函数中用this.xxx方式添加的属性，都会成为子对象的自有属性

(2). 错误的做法2: 在class中直接写死一个属性值

原因: 直接放在class中写死的属性值，不会保存在原型对象中成为共有属性，依然会添加到子对象中成为子对象的自有属性

但是，直接写在class中的方法，却是保存在原型对象中的共有方法。

(3). 正确的方法: 依然使用旧js中: 类型名.prototype.共有属性=值

(4). 示例: 测试哪种凡是设置共有属性值正确:

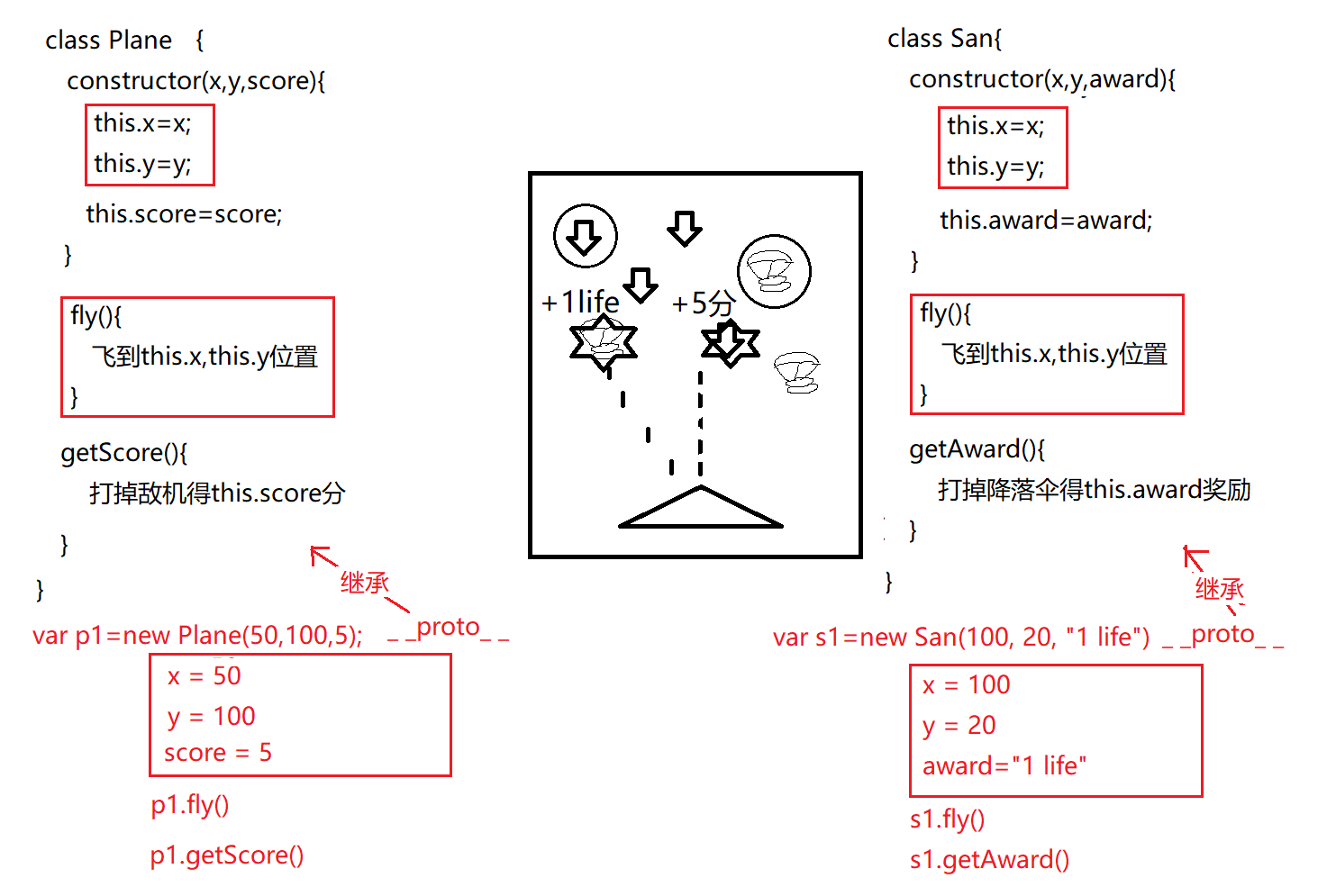
5\_property.html

|  |
| --- |
| <!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>      class Student{        //错误2:        //className="初一2班"        //直接放在class内的属性值，默认不会放在原型对象中，而是成为子对象的自有属性        //今后框架也不能这么写！        constructor(sname,sage){          //错误1:          //this.className="初一2班";//自有属性          //凡是在构造函数中用this.xxx方式添加的属性，都会成为子对象的自有属性          this.sname=sname;          this.sage=sage;        }        //默认就是放在原型对象中的        intr(){          console.log(`I'm ${this.sname},I'm ${this.sage}`)        }      }      //旧js中:      Student.prototype.className="初一2班"        var lilei=new Student("Li Lei",11);      var hmm=new Student("Han Meimei",12);      console.log(lilei);      console.log(hmm);    </script>  </body>  </html>  运行结果:  Student {sname: "Li Lei", sage: 11}  sage: 11  sname: "Li Lei"  \_\_proto\_\_:  className: "初一2班"  constructor: class Student  intr: ƒ intr()  \_\_proto\_\_: Object  Student {sname: "Han Meimei", sage: 12}  sage: 12  sname: "Han Meimei"  \_\_proto\_\_:  className: "初一2班"  constructor: class Student  intr: ƒ intr()  \_\_proto\_\_: Object |

7. 两种类型间的继承:

对应小程序视频: 小程序->在线->JSCORE->day08 2. class extends

(1). 问题: 两种class之间可能包含部分相同的属性结构和方法定义。如果在两个class中重复定义相同的属性结构和方法，非常不便于今后的维护。



(2). 示例: 使用两种class，描述飞机大战游戏中的敌机和降落伞类型

6\_without\_extends.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>      class Plane{        constructor(x,y,score){          this.x=x;          this.y=y;          this.score=score;        }        fly(){          console.log(`飞到x:${this.x},y:${this.y}位置`)        }        getScore(){          console.log(`击落一架敌机得${this.score}分`)        }      }      var p1=new Plane(50,100,5);      console.log(p1);      p1.fly();      p1.getScore();      class San{        constructor(x,y,award){          this.x=x;          this.y=y;          this.award=award;        }        fly(){          console.log(`飞到x:${this.x},y:${this.y}位置`)        }        getAward(){          console.log(`打掉一个降落伞得${this.award}奖励`)        }      }      var s1=new San(100,20,"1 life");      console.log(s1);      s1.fly();      s1.getAward();    </script>  </body>  </html>  运行结果:  Plane {x: 50, y: 100, score: 5}  score: 5  x: 50  y: 100  \_\_proto\_\_:  constructor: class Plane  fly: ƒ fly()  getScore: ƒ getScore()  \_\_proto\_\_: Object  飞到x:50,y:100位置  击落一架敌机得5分  San {x: 100, y: 20, award: "1 life"}  award: "1 life"  x: 100  y: 20  \_\_proto\_\_:  constructor: class San  fly: ƒ fly()  getAward: ƒ getAward()  \_\_proto\_\_: Object  飞到x:100,y:20位置  打掉一个降落伞得1 life奖励 |

(3). 解决: 再额外定义一个父类型class，集中存储相同部分的属性结构和方法定义。再让两种类型继承父类型class。

(4). 如何: 2步:

a. 先定义父类型class:

1). 父类型class的构造函数中集中定义相同部分的属性结构

2). 父类型原型对象中集中定义相同的方法定义。

b. 让子类型class，继承父类型class: 2步:

1). 让子类型class用extends关键字继承父类型class:

class 子类型 extends 父类型{

2). 在子类型构造函数中用super关键字借用父类型构造函数与子类型构造函数联合，共同创造出子对象中所有属性。

constructor( ... ... ){

super(...) //super自动调整this指向正在创建的新对象，无需自己手动call()

//super必须是子类型构造函数中第一句话！否则报错！

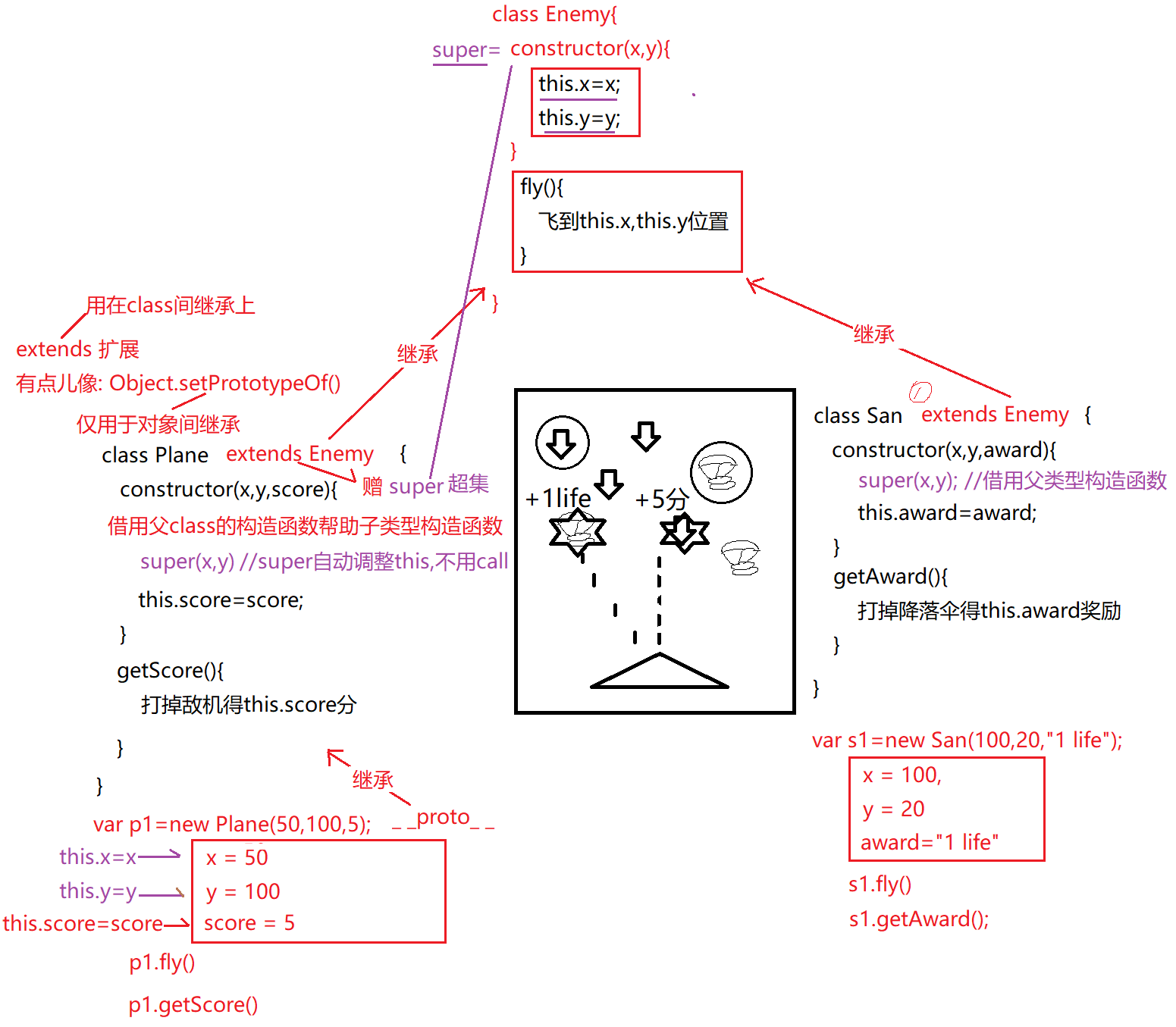
//因为万一出现重复的属性，程序中永远只允许子类型的属性覆盖父类型的属性。绝不允许父类型的属性覆盖子类型！

this.其它属性=xxx;

}

}

(5). 结果: 将来用new 子类型()创建出的子对象，也可以拥有父类型class规定的自有属性，也可以使用父类型class中的共有方法！



(6). 示例: 使用extends优化以上飞机大战程序的class结构

7\_extends.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>      class Enemy{        constructor(x,y){          this.x=x;          this.y=y;        }        fly(){          console.log(`飞到x:${this.x},y:${this.y}位置`)        }      }      class Plane extends Enemy{        constructor(x,y,score){          super(x,y);          this.score=score;          //super(x,y)//报错！        }        getScore(){          console.log(`击落一架敌机得${this.score}分`)        }      }      var p1=new Plane(50,100,5);      console.log(p1);      p1.fly();      p1.getScore();      class San extends Enemy{        constructor(x,y,award){          super(x,y);          this.award=award;          //super(x,y)//报错！        }        getAward(){          console.log(`打掉一个降落伞得${this.award}奖励`)        }      }      var s1=new San(100,20,"1 life");      console.log(s1);      s1.fly();      s1.getAward();    </script>  </body>  </html>  运行结果:  Plane {x: 50, y: 100, score: 5}  score: 5  x: 50  y: 100  \_\_proto\_\_:  constructor: class Plane  getScore: ƒ getScore()  \_\_proto\_\_:  constructor: class Enemy  fly: ƒ fly()  \_\_proto\_\_: Object  飞到x:50,y:100位置  击落一架敌机得5分  San {x: 100, y: 20, award: "1 life"}  award: "1 life"  x: 100  y: 20  \_\_proto\_\_:  constructor: class San  getAward: ƒ getAward()  \_\_proto\_\_:  constructor: class Enemy  fly: ƒ fly()  \_\_proto\_\_: Object  飞到x:100,y:20位置  打掉一个降落伞得1 life奖励 |

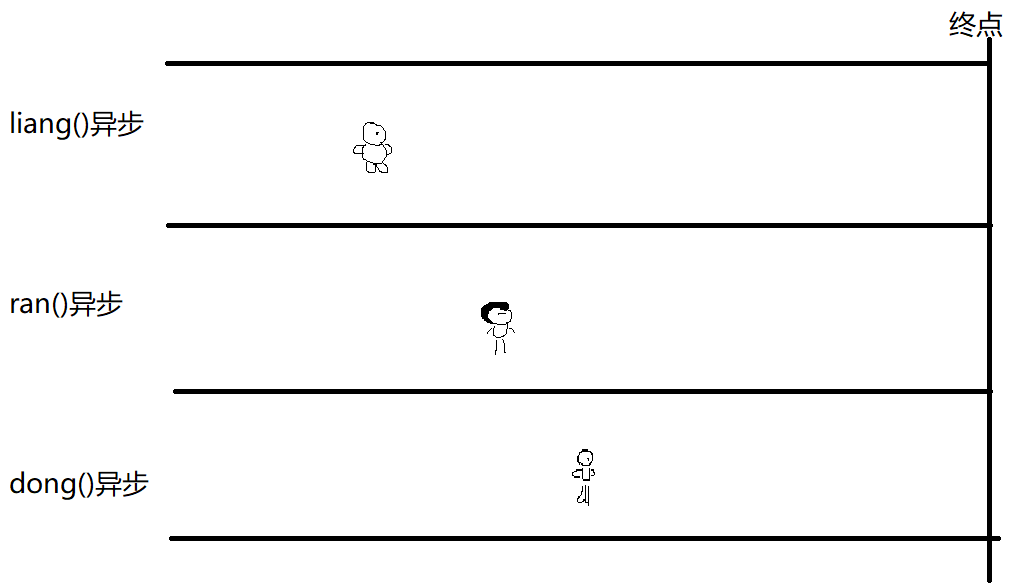
三. Promise

对应小程序视频: 小程序->在线->JSCORE->day08 3. 回调地狱 callback hell和promise

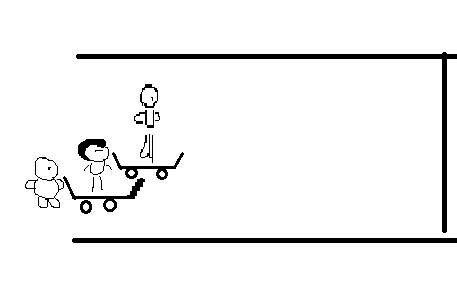
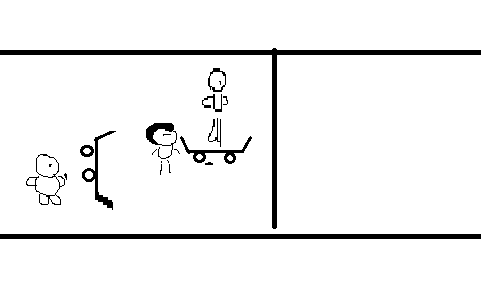
1. 问题: 希望多个异步任务/函数，也能顺序执行！

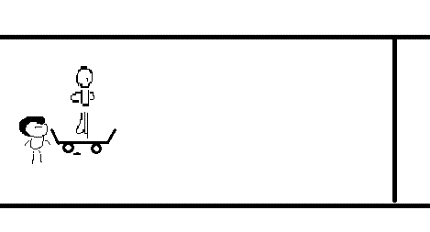
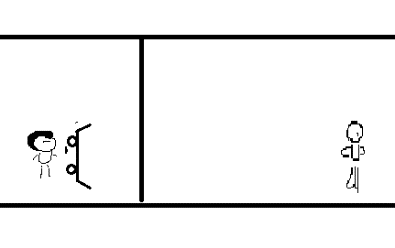
2. 错误的解决: 单纯的顺序调用多个异步函数，无法实现多个异步函数必须顺序执行的

因为: 每个异步函数都在主程序之外，单独执行，谁也不等谁！



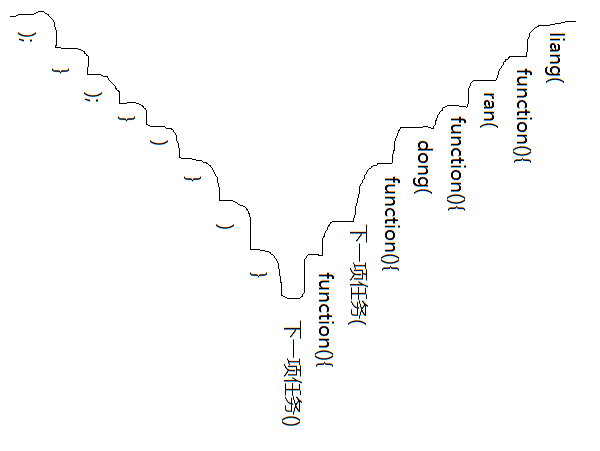
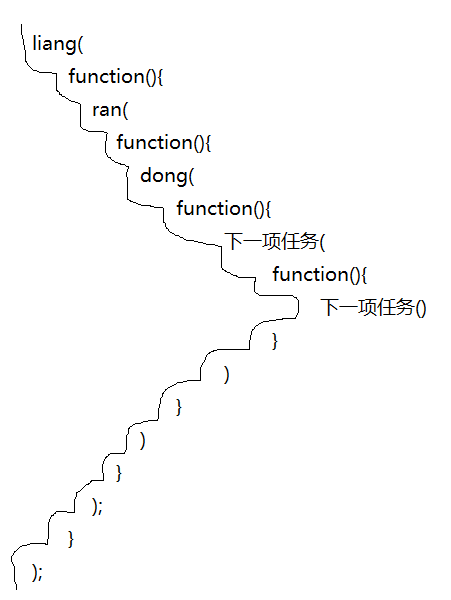
3. 不太好的解决: 通过回调函数方式: 前一个函数，提前将后一个函数抱在怀中，前一个函数不松手，后一个函数无法执行！

4. 问题: 回调地狱:

(1). 什么是: 由于回调函数嵌套层级太深，形成的代码结构，就称为回调地狱



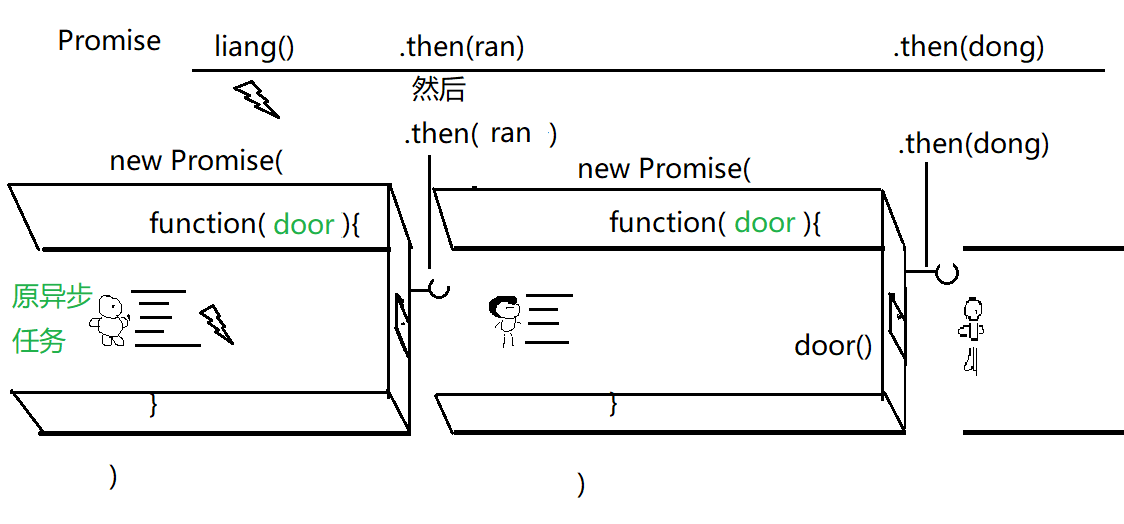
(2). 缺点: 极其不便于代码的阅读和维护！

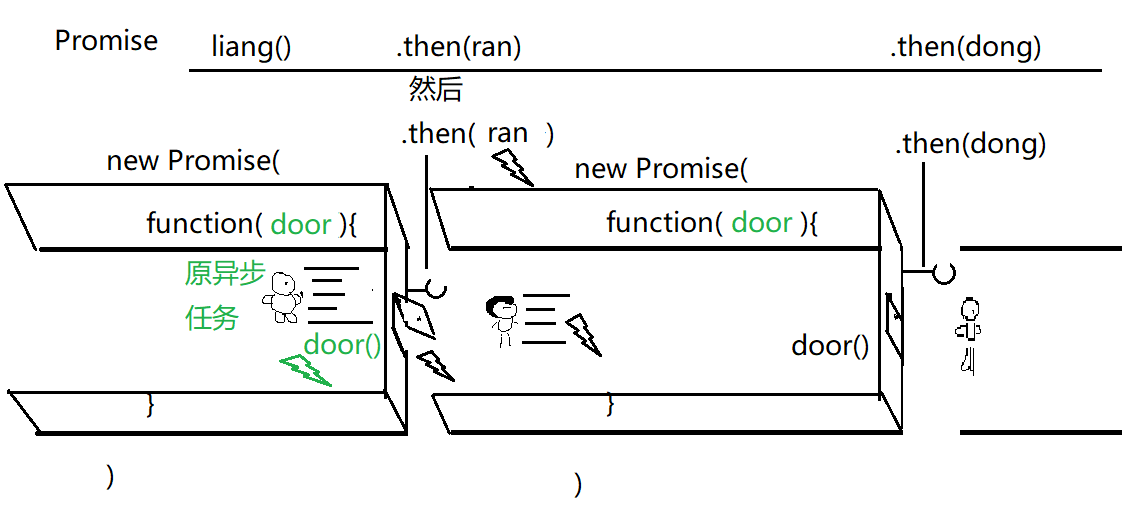
5. 示例: 使用回调函数实现多个异步任务必须顺序执行:

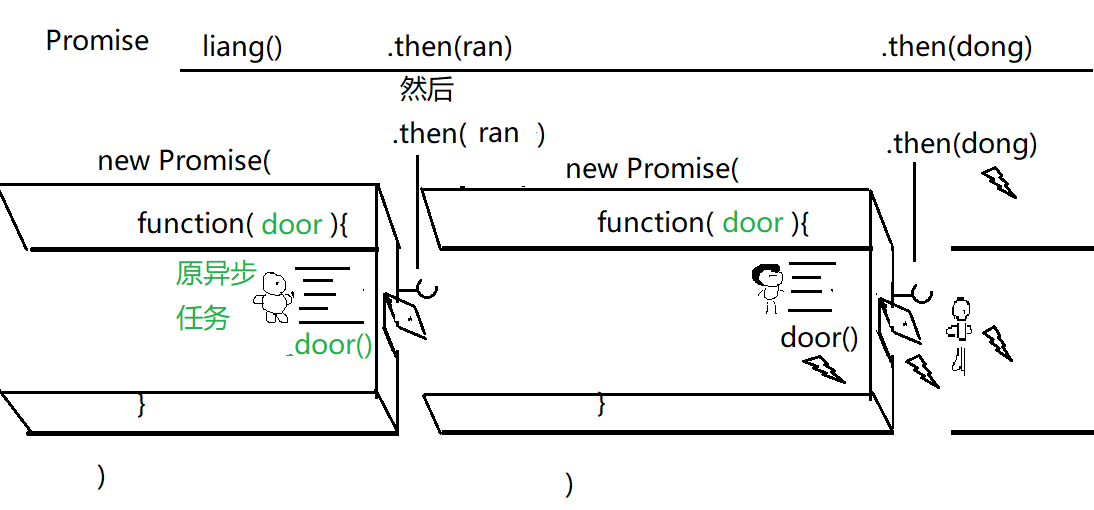
8\_callback.html

|  |
| --- |
| <!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>      //达内举办运动会，亮亮，然然，东东参加百米赛跑项目      function liang(che){        console.log(`亮起跑...`);        setTimeout(function(){//异步          console.log(`亮到达终点！`);          //在亮到达终点之后，卸车          che();          //相当于function(){ ran() }(); 于是然才开始执行        },6000)      }      function ran(che){        console.log(`然起跑...`);        setTimeout(function(){//异步          console.log(`然到达终点！`)          //在然到达终点之后，卸车          che();          //相当于function(){ dong() }() //东才开始执行        },4000)      }      function dong(){        console.log(`东起跑...`);        setTimeout(function(){//异步          console.log(`东到达终点！`)        },2000)      }      //希望三个人接力跑！一个跑完，下一个才能起跑      //错误的解决: 仅顺序调用      //因为异步函数，在主程序之外独立执行，谁也不等谁！      // liang();      // ran();      // dong();      //不好的但是正确的解决方法:      liang(        //亮的che=        function(){          ran(            //然的车=            function(){              dong()            }          );        }      );    </script>  </body>  </html>  运行结果:  亮起跑...  亮到达终点！  然起跑...  然到达终点！  东起跑...  东到达终点！ |

6. 好的解决方法: Promise:







(1). 定义前一个任务函数时，将原异步任务的代码，用new Promise(function(door){...})包裹，并返回到函数外部

function 前一项任务(){ //不要写回调函数的形参变量了！

return new Promise(

function(door){

原异步任务代码

原异步任务最后一句话执行完:

door()开门！->自动执行后续.then()中的下一项任务

}

)

}

(2). 顺序执行多个异步函数时:

前一项任务().then(后一项任务)

(3). 只要后一项任务也返回new Promise()对象，则可以继续.then:

前一项任务().then(后一项任务).then(下一项任务)...

7. 示例: 使用Promise实现多个异步函数顺序执行

9\_promise.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>      //达内举办运动会，亮亮，然然，东东参加百米赛跑项目      function liang(){        //在函数内造了一个带门的格子间！        //应该返回到外部，才能让外部知道这里是一个格子间        return new Promise(                // 赠门          function(door){            console.log(`亮起跑...`);            setTimeout(function(){//异步              console.log(`亮到达终点！`);              //在亮到达终点之后，开门              door();//->自动执行.then(下一项任务)            },6000)          }        )      }      function ran(){        return new Promise(          function(door){            console.log(`然起跑...`);            setTimeout(function(){//异步              console.log(`然到达终点！`)              //在然到达终点之后，开门              door(); //->自动执行.then(下一项任务)            },4000)          }        )      }      function dong(){        console.log(`东起跑...`);        setTimeout(function(){//异步          console.log(`东到达终点！`)        },2000)      }      //如何将格子间和下一项任务链接起来！      // liang()      // //return new Promise()格子间      // .then(ran) //会在格子间中调用door()开门后，才自动开始执行！      // //只要自动执行然，又自动返回new Promise()格子间      // .then(dong)      liang().then(ran).then(dong);//串联    </script>  </body>  </html>  运行结果:  亮起跑...  亮到达终点！  然起跑...  然到达终点！  东起跑...  东到达终点！ |

总结: 10. ES6:

1. 今后，只要拼接字符串，都用模板字符串:

(1). 整个字符串包裹在一对儿反引号`...`中

(2). 字符串中动态生成的内容必须放在${}里

(3). ${}里:

a. 可以放一切有返回值的合法的变量或js表达式。

b. 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. 今后，声明变量都用let代替var

(1). let的好处: 阻止声明提前，让代码块也变成块级作用域

(2). let的原理: 匿名函数自调+自动修改变量名

(3). let的小脾气: 禁止重复声明; 禁止提前使用; 在全局声明也不保存在window中

3. 今后，几乎所有的function都可用箭头函数简写:

(1). 如何: 3句话:

a. 去掉function，在()和{}之间加=>

b. 如果只有一个形参，可省略()

c. 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

(2). 今后:

a. 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！

b. 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  (内部其实是for) | for of | for in |
| 数字下标 | 索引数组 | √ | √ | √ | × |
| 类数组对象 | √ | × | √ | × |
| 字符串 | √ | × | √ | × |
| 自定义名称下标 | 关联数组 | × | × | × | √ |
| 对象 | × | × | × | √ |

5. 参数增强:

(1). 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){

... ...

}

(2). 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

}

(3). 打散数组: 今后调用函数时，只要单纯打散数组再传参时

a. 如何: 函数(...数组)

b. ...口诀: 定义函数时...表示收集, 调用函数时...表示打散

c. ...语法糖:

1). 复制一个数组: var arr2=[...arr1];

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

3). 复制一个对象: var obj2={ ... obj1 }

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

6. 解构:

(1). 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;

(2). 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:

({属性名也是变量名, 属性名也是变量名}=对象)

如果不加()，就必须前加var

(3). 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

(1). 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

(2). 共有属性:

a. 直接放在class内的"属性名=属性值"，依然会成为子对象的自有属性

b. 想设置共有属性，只能在class后用旧方式设置:

类型名.prototype.共有属性=属性值

(3). 继承:

class 子类型 extends 父类型{

constructor(...){

super(...);

this.属性名=形参;

}

子类型共有方法(){

...

}

}

两个关键字: extends是继承的意思，super父类型构造函数的代名词

9. Promise: 当多个异步任务必须顺序执行时，就可用promise

(1). 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()

}

)

}

(2). 调用前一项任务时:

前一项任务().then(下一项任务)

(3). 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(...)...

|  |
| --- |
| 总结: this 4种: 判断this，~~一定不要看定义在哪儿！~~只看调用时！  1. obj.fun() this->obj  2. fun() 或 (function(){ ... })() 或 多数回调函数 this->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->将来谁调用指谁，同第一种情况 |

今日对应小程序视频列表:

小程序->在线->JSCORE->day07 6. 解构(上) destruct 数组解构 对象解构

小程序->在线->JSCORE->day08 1. 解构(下) destruct 参数解构

2. class extends

3. 回调地狱 callback hell和promise

作业:

1. 复习今日问题清单: 小程序->首页->JSCORE->day08

2. 看小程序视频，学习promise高级用法:

小程序->在线->JSCORE->day08 4. promise 状态 错误处理

5. promise传参

3. 看小程序视频封装支持promise的ajax函数

4. 预习DOM: 小程序->在线->DOM->day01

0 DOM Tree树 元素 节点 标签

1 按节点间关系查找

2 按HTML特征查找

3. 购物车 shoppingcart

案例中素材，在code.tarena.com.cn/webcode/wfd2004/11\_DOM中 day01.zip

code.tarena.com.cn这个网站下载资源没有任何限制，在哪里都可以所以访问