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->day08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->day08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->day08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:

1. 参数解构:

2. class

3. \*\*\*\*\*Promise

一. 参数解构:

1. 问题: 如果一个函数，多个形参变量都不确定将来有没有值时，但是又要求传入的实参值和形参变量必须一一对应！

2. 错误的解决:

(1). 剩余参数或arguments: 实参值必须传给指定的形参变量，不能传错了！虽然剩余参数和arguments可以接受不确定个数的实参值。但是没有顺序和位置的限制。

(2). 默认参数: 虽然可以给所有的形参都加默认参数，但是用户调用函数时，无法跳过前边的个别形参，仅给后边的形参变量传值！——没有这种语法！

3. 正确的方法: 参数解构

4. 什么是参数解构: 将所有的形参和实参都包裹在对象结构中，再传参！

5. 何时: 今后只要一个函数，多个形参变量都不确定将来有没有值时，但是又要求传入的实参值和形参变量必须一一对应！就要用参数解构

6. 如何: 2步:

(1). 定义函数时: 将函数的所有形参变量都装扮成一个对象结构

function 函数名({

属性名1: 形参1,

属性名2: 形参2,

... : ...

}){

... ...

}

(2). 调用函数: 将所有实参值，也要装扮成和定义函数时形参对象一模一样的结构

函数名({

属性名1: 实参值1,

属性名2: 实参值2,

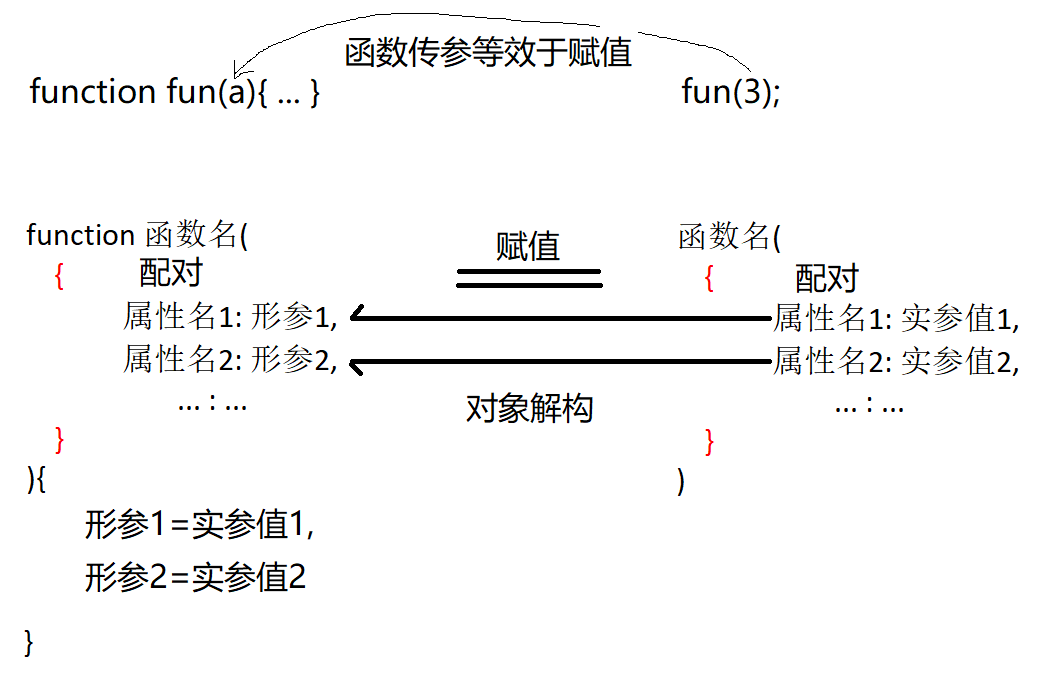
... : ...

})

强调: 一旦定义函数时，采用了对象解构的形式，则调用函数时，必须传入一个对象结构，至少是一个{}

7. 结果: 形参1=实参值1, 形参2=实参值2, ... ...

8. 原理: 其实参数解构就是对象解构在函数传参时的应用



9. 简写:

(1). 通常用于配对的属性名和形参变量名都是相同！所以，形参对象结构中，每个形参变量只写一个名字即可！但是一个名字两用: 既配对，又当形参变量名

强调: 虽然定义函数时，属性名和形参变量可以合为一个名字，但是调用时，属性名和实参值必须都要写完整！

(2). 如果属性名和形参变量合为一个之后，如果给形参提供默认值，就可用ES6的参数默认值简写！

10. 示例: 用对象解构实现订餐函数

|  |
| --- |
| //想定义一个点套餐的函数  //套餐有默认值  //用户可以随意选择更换  // function order({//装扮  // //  配对   形参变量  //    zhushi:zhushi,  //   xiaochi:xiaochi,  //   yinliao:yinliao  // }){  function order({zhushi="香辣鸡腿堡",xiaochi="薯条",yinliao="可乐"}){    // zhushi===undefined&&(zhushi="香辣鸡腿堡");    // xiaochi===undefined&&(xiaochi="薯条");    // yinliao===undefined&&(yinliao="可乐");    console.log(`      您点的套餐是:        主食:${zhushi},        小吃:${xiaochi},        饮料:${yinliao}      请当面查验清点    `)  }  //套餐中一个都不换！  order({});    //套餐中全都换！  order({//装扮  //  配对    实参值     zhushi:"奥尔良烤腿堡",    xiaochi:"红豆派",    yinliao:"奶茶"  });    //只想换主食，另外两个保持默认  order({//装扮  // 配对   实参值    zhushi:"巨无霸"  });    //只想换小吃?  order({//装扮  //  配对   实参值    xiaochi:"薯泥"  });    //只想换饮料?  order({//装扮  //  配对   实参值    yinliao:"红茶"  })    //想换小吃和饮料  order({//装扮  // 配对    实参值    yinliao:"牛奶",    xiaochi:"辣翅"  }) |

二. class:

1. 什么是class: 集中存储一个类型的构造函数和原型对象的程序结构

2. 为什么: 因为旧的js中，明明属于一个类型的构造函数和原型对象方法，是分开写的。不符合"封装"的要求。

3. 何时: 今后只要定义一种新类型(包括构造函数和原型对象方法时)，都用class来包裹原来的构造函数和原型对象方法

4. 如何:

(1). 用class{}包裹之前的构造函数和原型对象方法

(2). 构造函数名提升为整个class的类型名，不仅仅只属于构造函数

今后所有构造函数必须更名为constructor

构造函数中的内容和旧js中写法保持不变！

(3). 从此class中的原型对象方法，无需再加类型名.prototype前缀以及"=function"也可省略！——今后，凡是直接放在class中的方法，默认都是放在原型对象里的！

说明: class内的构造函数和多个原型对象方法之间不用加,或;分割

5. 使用class: class的使用和旧js中构造函数和原型对象方法的使用完全一样！存储结构完全一样的！——class其实就是"新瓶装旧酒"！

6. 示例: 使用class创建学生类型:

|  |
| --- |
| //定义一个学生类型，要求:  //将来每个学生都要有学生姓名和学生年龄两个属性  //将来每个学生都应该会自我介绍  //1. 用class{}包括构造函数和原型对象方法  //类型  class Student{    constructor(sname,sage){      this.sname=sname;      this.sage=sage;    }    intr(){      console.log(`I'm ${this.sname}, I'm ${this.sage}`)    }  }  var lilei=new Student("Li Lei",11);  console.log(lilei)  lilei.intr(); |

7. 共有属性问题: 在class中强烈不推荐共有属性:

(1). 直接放在class中的属性，绝不是共有属性，而会成为将来每个子对象的自有属性

(2). 写死在构造函数里的属性，更不是共有属性。所有构造函数里的属性，将来一律都会成为每个孩子的自有属性。

(3). 其实可以通过特殊的方式添加共有属性，但是强烈不推荐:

a. 在构造函数中，向原型对象中添加共有属性

b. 强调: 一点要先判断原型对象中如果没有这个共有属性，才强行添加。如果已经有这个共有属性了，则不要再重复添加！

(4). 示例: 实现共有属性:

|  |
| --- |
| //定义一个学生类型，要求:  //将来每个学生都要有学生姓名和学生年龄两个属性  //将来每个学生都应该会自我介绍  //1. 用class{}包括构造函数和原型对象方法  //类型  class Student{    //className="初一2班"    constructor(sname,sage){      //this.className="初一2班";      this.sname=sname;      this.sage=sage;      //只有刚创建第一个子对象时，原型对象中没有这个共有属性，才自动添加      if(Student.prototype.className===undefined){        Student.prototype.className="初一2班";      }      //否则如果原型对象中已经有这个共有属性了，就不再重复添加    }    intr(){      console.log(`I'm ${this.sname}, I'm ${this.sage}`)    }  }  var lilei=new Student("Li Lei",11);//首次调用构造函数，因为原型对象中暂时没有className共有属性，所以自动向原型对象中添加一个共有属性  var hmm=new Student("Han Meimei",12);//再次调用构造函数，因为原型对象中已经有了className共有属性，所以不再重复向原型对象中添加className属性  console.log(lilei);  console.log(hmm);  //过了一年大家同时升一级  Student.prototype.className="初二2班";  console.log(lilei.className, hmm.className);  //班里招了一位新同学  var lily=new Student("Lily",10);//再次调用构造函数，因为原型对象中已经有了className共有属性，所以不再重复向原型对象中添加className属性  console.log(lily);  console.log(lilei.className, hmm.className, lily.className);  //     希望        初二2班        初二2班          初二2班 |

8. 访问器属性:

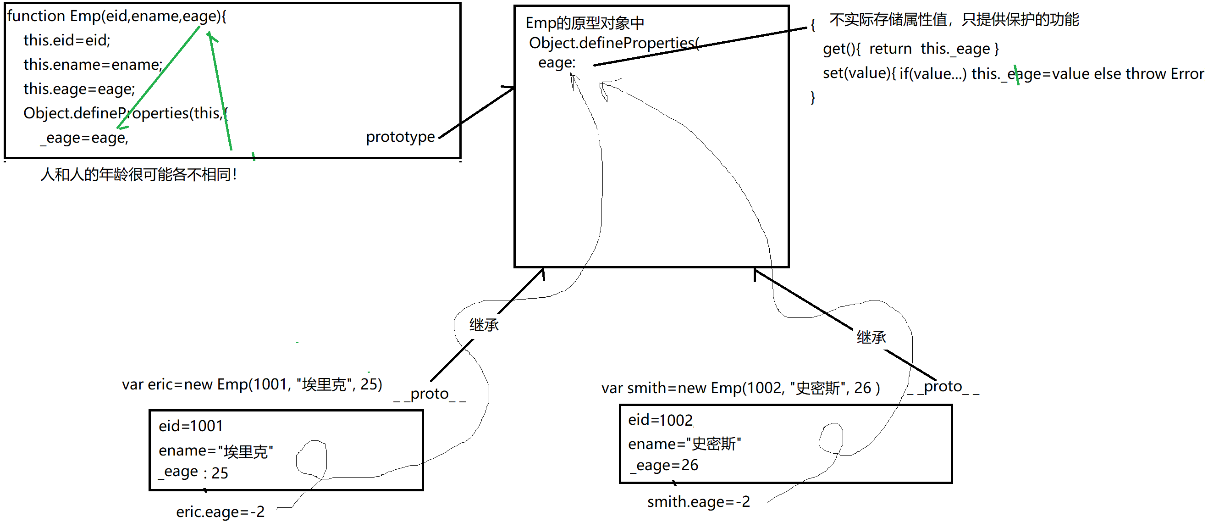
(1). 旧的构造函数，如何保护将来创建的所有子对象

a. 在构造函数妈妈肚子里就要保护将来要出生的子对象。

其实就是保护this，因为构造函数中this->将来要创建的新子对象

b. 但是，因为访问器属性只提供保护功能，不实际存属性值，且所有孩子的访问器属性的保护逻辑规则都是统一的。所以，访问器属性应该只在原型对象中定义一份即可

但是，隐姓埋名的半隐藏的实际存储属性值的\_eage，必须放在构造函数妈妈肚子里。因为将来每个孩子的年龄很可能各不相同！



(2). 示例: 保护一个构造函数创建的多个子对象

|  |
| --- |
| "use strict";  function Emp(eid, ename,eage){    this.eid=eid;    this.ename=ename;    this.eage=eage;    //要保护的是将来要创建的子对象    Object.defineProperties(this,{      eid:{ writable:false },      //1. 定义一个隐姓埋名的半隐藏的数据属性      \_eage:{        value:eage,        writable:true,        enumerable:false      }    })    //密封将来要出生的子对象    Object.seal(this);  }  //所有孩子共用的保镖功能应该只放在所有孩子父元素中一份即可  Object.defineProperties(Emp.prototype,{    //2. 请保镖，冒名顶替:    eage:{//周三讲的ES6简写      get(){        return this.\_eage;      },      set(value){        if(value>=18&&value<=65){          this.\_eage=value        }else{          throw Error("年龄必须介于18~65之间")        }      }    }  })  var eric=new Emp(1001, "埃里克", 25);  //试图修改eid属性  //eric.eid=0;  //试图删除ename属性  //delete eric.ename  //试图修改eage为-2  //eric.eage=-2;  console.log(eric);  var smith=new Emp(1002,"史密斯",26);  //试图修改eid属性  //smith.eid=0;  //试图删除ename属性  //delete smith.ename  //试图修改eage为-2  //smith.eage=-2;  console.log(smith); |

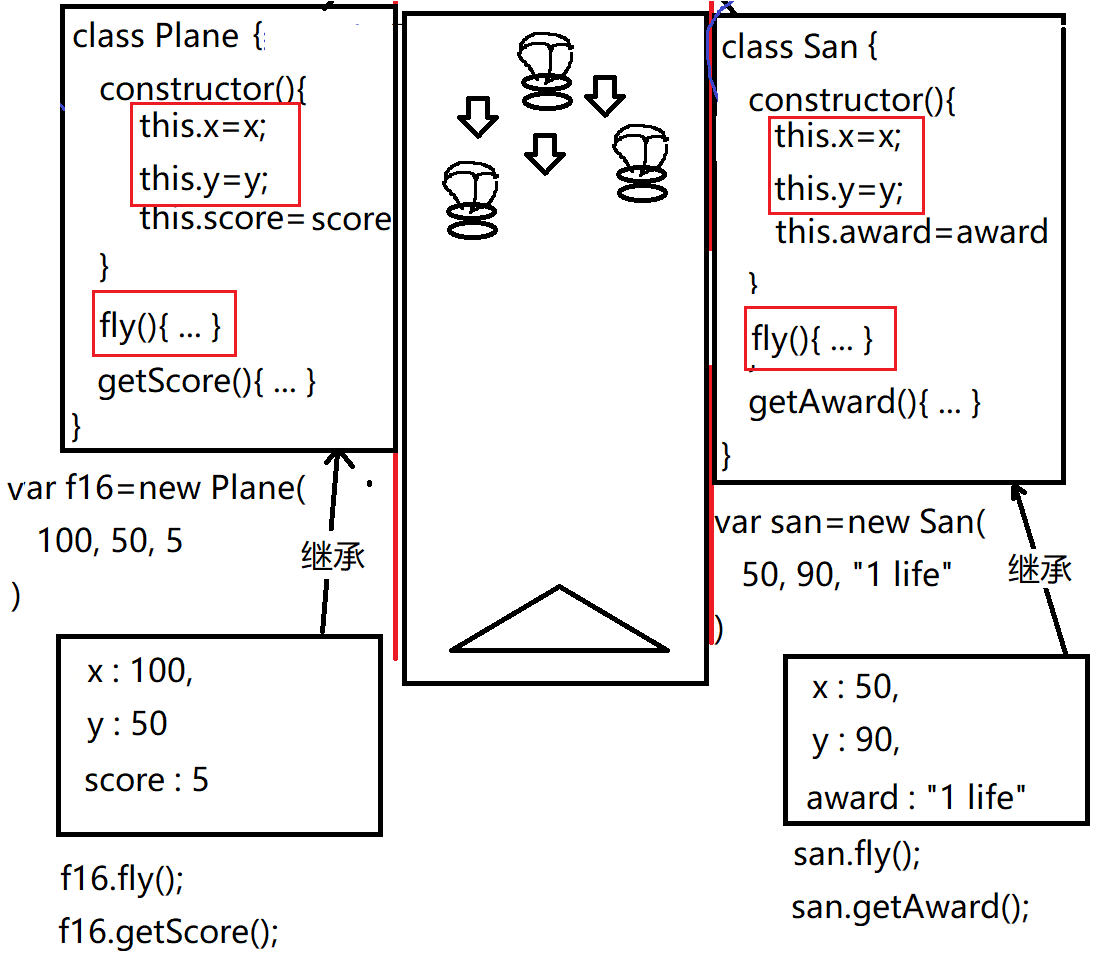
(3). 在class中如何保护将来的子对象？同普通构造函数时的保护方式。只不过访问器属性有简写！

(4). 示例: 为class添加访问器属性：

|  |
| --- |
| "use strict";  class Emp{    constructor(eid, ename,eage){      this.eid=eid;      this.ename=ename;      this.eage=eage;      //要保护的是将来要创建的子对象      Object.defineProperties(this,{        eid:{ writable:false },        //1. 定义一个隐姓埋名的半隐藏的数据属性        \_eage:{          value:eage,          writable:true,          enumerable:false        }      })      //密封将来要出生的子对象      Object.seal(this);    }    //向该类型的原型对象中添加一个访问器属性eage，包含两个保镖函数get和set    get eage(){      return this.\_eage;    }    set eage(value){      if(value>=18&&value<=65){        this.\_eage=value      }else{        throw Error("年龄必须介于18~65之间")      }    }  }  var eric=new Emp(1001, "埃里克", 25);  //试图修改eid属性  //eric.eid=0;  //试图删除ename属性  //delete eric.ename  //试图修改eage为-2  //eric.eage=-2;  console.log(eric);  var smith=new Emp(1002,"史密斯",26);  //试图修改eid属性  //smith.eid=0;  //试图删除ename属性  //delete smith.ename  //试图修改eage为-2  //smith.eage=-2;  console.log(smith); |

9. 两种类型间的继承:

(1). 问题: 两种类型之间包含部分重复的属性定义和方法定义



(2). 解决: 多定义一个父类型class，集中保存两种子类型相同部分的属性定义和方法定义

(3). 如何:

a. 定义一个父类型:

1). 父类型构造函数中，保存多个子类型相同部分的属性结构定义

2). 父类型class中，保存多个子类型相同部分的方法定义

b. 让子类型继承父类型:

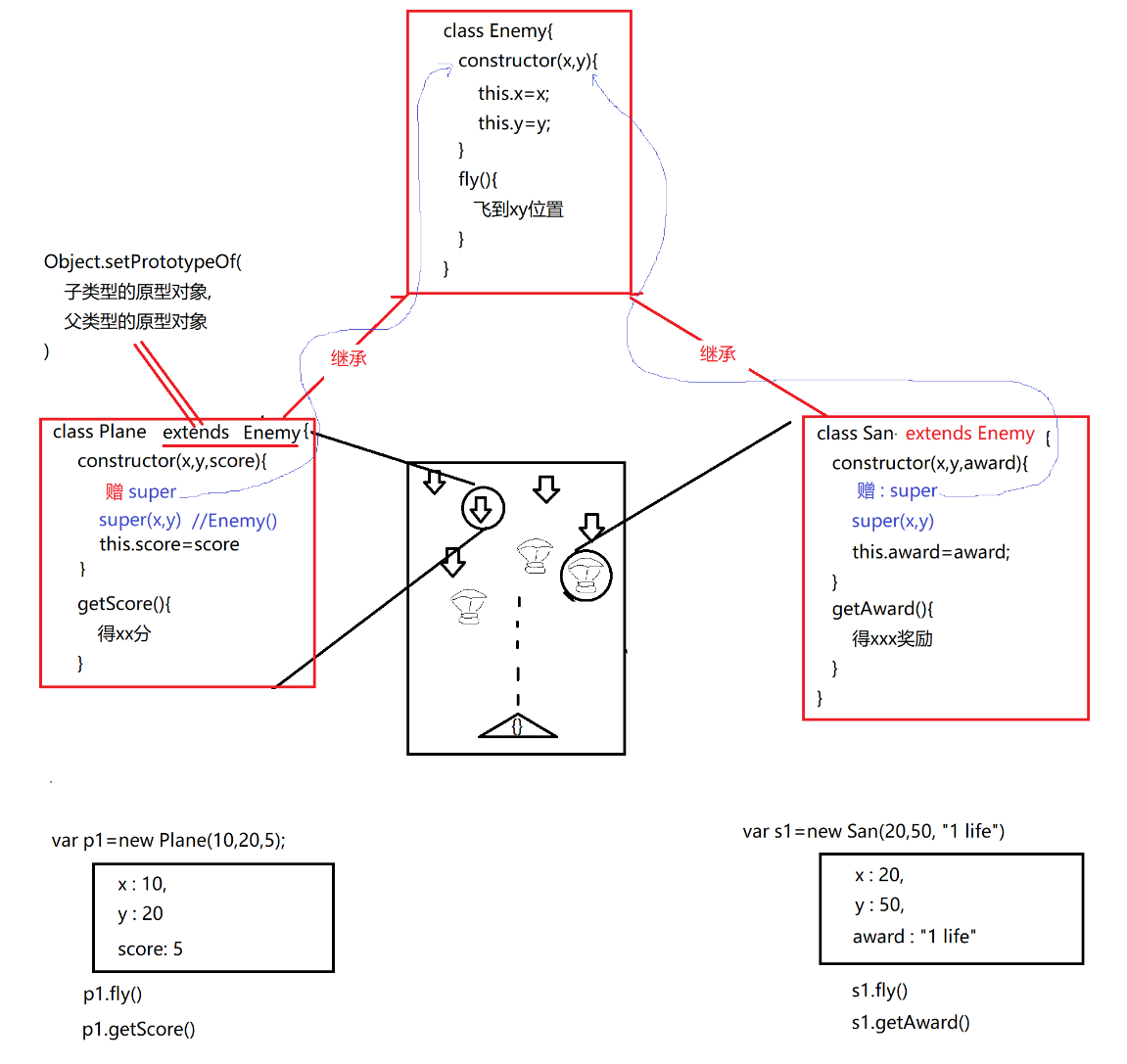
1). 子类型class用"extends 父类型"，取代Object.setPrototypeOf(xxx, xxx)，让子类型继承父类型

2). 在子类型构造函数中调用父类型构造函数，两个构造函数共同创造出将来的子对象。——将来的子对象中同时包含父类型和子类型中构造函数的属性定义。

如何在子类型构造函数中调用父类型构造函数？

extends附赠了一个关键词super，super指向父类型的构造函数。调用super，就等效于调用父类型的构造函数了。

强调: super()必须先于子类型的属性调用！



(4). 示例: 使用class和继承实现飞机大战数据结构

|  |
| --- |
| 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;    }    getScore(){      console.log(`击落一架敌机得${this.score}分`)    }  }  class San extends Enemy{    constructor(x,y,award){      super(x,y);      this.award=award;    }    getAward(){      console.log(`击落一个降落伞得${this.award}奖励`)    }  }  var p1=new Plane(10,20,5);  var s1=new San(20,50,"1 life");  console.log(p1);  console.log(s1);  p1.fly();  p1.getScore();  s1.fly();  s1.getAward(); |

三. Promise

1. 问题: 实际开发中，有时需要多个异步任务必须按顺序执行！

2. 错误: 单纯顺序调用两个异步函数！无法保证顺序执行！异步函数谁也不等谁！

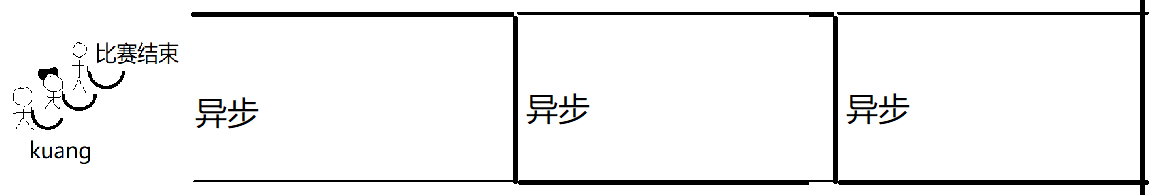
3. 不好的解决方案: 回调函数方式解决:

(1). 前一个异步函数:

a. 定义一个形参变量，准备暂时保存下一个异步函数对象，只保存，暂不执行

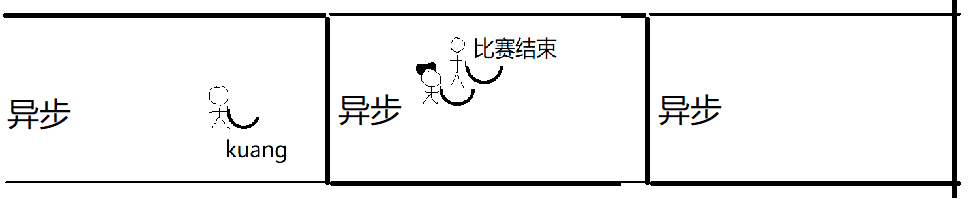
就好像，前一个函数把后一个函数抱在了怀里！

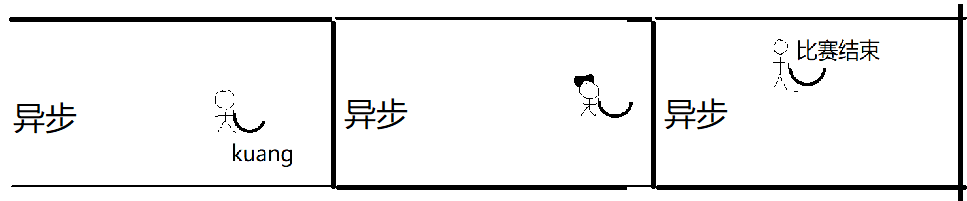
b. 当前一个函数执行完自己的最后一句话之后，再执行形参变量里提前保存的下一项任务。

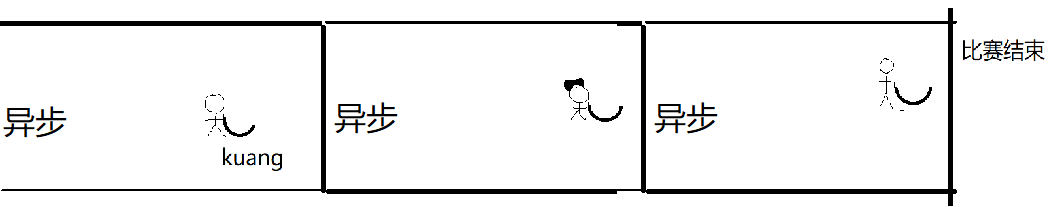


(2). 调用前一个函数时，暂时将后一个函数传入前一个函数的形参变量中暂时保存，暂不执行。

(3). 当前一个函数调用后，前一个函数内部会自动执行提前保存的下一项任务——承诺！







4. 示例: 使用回调函数解决多个异步函数必须顺序执行的问题：

|  |
| --- |
| //亮定义一个筐，准备暂时保存下一项任务  function liang(kuang){    console.log(`亮起跑...`);    setTimeout(function(){      console.log(`亮到达终点!`);      //当亮自己的任务结束后      kuang();//才取出筐里的东西开始执行    },6000)  }  //然也定义一个筐，准备暂时保存下一项任务  function ran(kuang){    console.log(`然起跑...`);    setTimeout(function(){      console.log(`然到达终点`);      //当然自己的任务结束后      kuang();//才取出筐里的东西开始执行    },4000)  }  //东也定义一个筐，准备暂时保存下一项任务  function dong(kuang){    console.log(`东起跑...`);    setTimeout(function(){      console.log(`东到达终点`)      //当东自己的任务结束后      kuang();//才取出筐里的东西开始执行    },2000)  }  console.log(`比赛开始`);  //错误:  // //先调用亮  // liang();//异步  // //再调用然  // ran();//异步  // //谁也不等谁！  //不好的做法:回调函数  liang(    function(){//kuang      ran(        function(){//kuang          dong(            function(){//kuang              console.log(`比赛结束！`)            }          )        }      )    }  );  //kuang里装的ran，但是暂不执行！  //当6s后，liang内部会自动调用kuang里装的ran()。 |

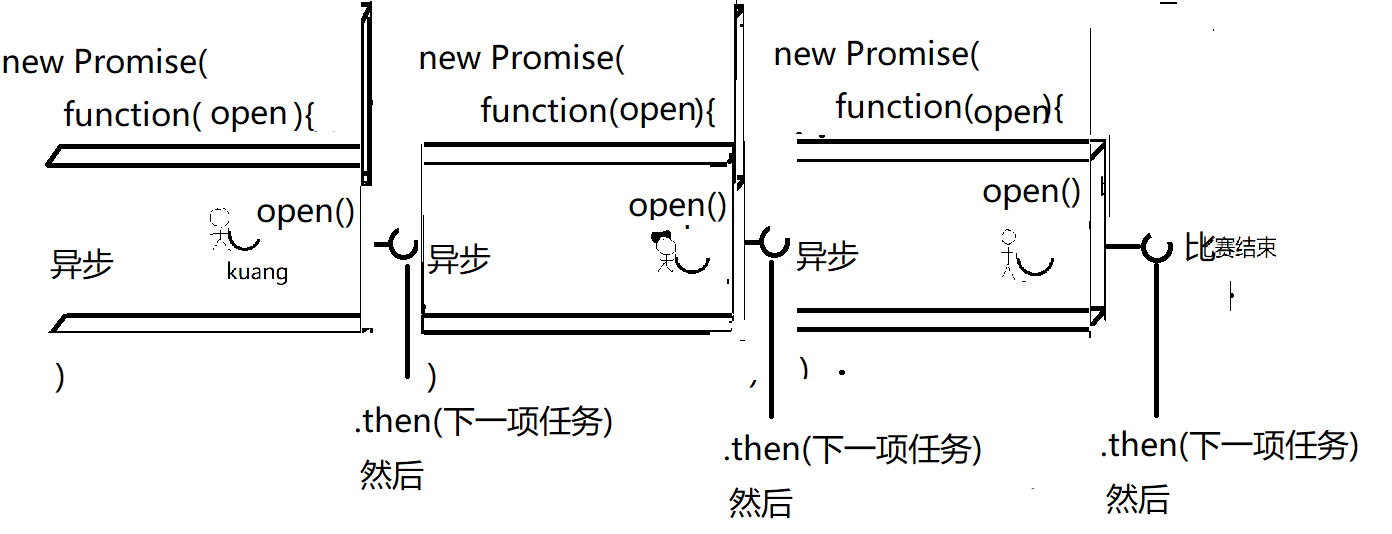
5. 问题: 如果多个异步函数，需要顺序执行时，就会形成很深的嵌套结构！——回调地狱！

6. 好的解决: Promise

(1). 将异步函数改造成一个new Promise格子间，并返回格子间对象，与下一项任务相连。

(2). 在new Promise()格子间内的异步任务内部，前一项任务需要决定何时调用开门的函数。只要前一项任务内部调用了开门函数，就会通知下一项任务开始执行。

(3). 调用前一项任务，然后调用.then()连接下一项任务。



7. 结果: 当前一项任务结束后，自动调用开门函数，下一项任务就自动开始执行了。

8. 示例: 使用promise实现多个异步任务必须顺序执行

|  |
| --- |
| function liang(){    //只有return出来的格子间对象，才能和下一项任务串联    return new Promise(//创建一个格子间对象      function(open){//只有function才能保存多条语句的任务        //赠:开门的函数        console.log(`亮起跑...`);        setTimeout(function(){          console.log(`亮到达终点!`);          open();//亮执行完开门！通知下一项任务可以继续！        },6000)      }    )  }  function ran(){    return new Promise(      function(open){        console.log(`然起跑...`);        setTimeout(function(){          console.log(`然到达终点`);          open();        },4000)      }    )  }  function dong(){    return new Promise(      function(open){        console.log(`东起跑...`);        setTimeout(function(){          console.log(`东到达终点`)          open();        },2000)      }    )  }  console.log(`比赛开始`);  //先让亮串联然  liang()//return new Promise()  .then(ran)//因为ran也return了new Promise()  .then(dong)//因为dong也return了new Promise()  .then(()=>console.log(`比赛结束`)) |

9. 优点: 无论多少异步函数必须顺序执行，都不会发生嵌套。

总结: ES5

1. 只有父对象，没有构造函数也想创建子对象: Object.create()

2. 替换函数中不想要的this: call apply bind

(1). 只在本地调用函数时，临时替换一次this: call

如果多个实参值放在一个数组中给的，需要先打散再传参: apply

(2). 希望反复调用函数副本，并永久绑定this或部分参数时: bind

3. 数组函数:

(1). 判断一个数组中是否所有元素都符合要求: every

(2). 判断数组中是否包含符合要求的元素: some

(3). 单纯遍历原数组中每个元素: forEach

(4). 保护原数组，生成一个新数组: map

(5). 复制出符合条件的元素组成新临时子数组: filter

(6). 对数组中的元素求和或汇总: reduce

总结: this

1. obj.fun() this->obj

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

3. new Fun() this->new正在创建的子对象

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

5. 访问器属性: this->访问器属性所在的对象本身

Object.defineProperty(eric,"eage",{

get:function(){

return this.\_eage

},

set:function(value){

this.\_eage=value;

}

})

总结: ES6

1. 拼接字符串: `...${js表达式}...`

2. 声明变量: let

3. 简化function: 箭头函数 =>

4. 简化普通for循环: for of

5. 希望调用函数时没有传实参值，形参变量也有默认值使用时: function fun(形参=默认值){}

6. 代替arguments: function fun(...数组名){ ... }

7. 调用函数时打散数组，再传参: fun(...数组)

8. 复制一个对象或数组: var arr2=[...arr] var obj2={...obj}

9. 合并多个对象或数组: var arr3=[...arr1, ...arr2] var obj3={...obj1, ...obj2}

10. 希望仅提取出数组中个别元素单独使用: var [变量1, 变量2, ...]=数组

11. 希望仅提取出对象中个别属性或方法单独使用: var {属性名1, 属性名2,...}=对象

作业:

1. (必须)看小程序视频复习使用回调函数和Promise技术实现多个异步函数顺序执行

小程序->在线->JSCORE->day08 3. 回调函数 callback hell 和promise 解决回调地狱...

2. (必须)看小程序视频学习promise前后两个任务之间传参:

小程序->在线->JSCORE->day08 5. promise 传参

3. (必须)看小程序视频学习封装发送ajax请求的函数：

小程序->在线->JSCORE->day-8

作业: 封装发送ajax请求的函数

4. (扩展)看小程序视频学习Promise高级技巧:

小程序->在线->JSCORE->day08

4. promise 状态 错误处理 reject catch

6. Promise.all 等待多个异步任务同时执行完再继续执行后续任务 ES7...

5. (扩展)看小程序视频学习Promise高频笔试题:

小程序->在线->JSCORE->day08

作业: 高频笔试题 封装支持Promise的ajax函数 ...

6. (礼物)飞机大战游戏源代码:

小程序->在线->JSCORE->day08 赠送: 飞机大战游戏源代码