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->day05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->day05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->day05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！）

正课:

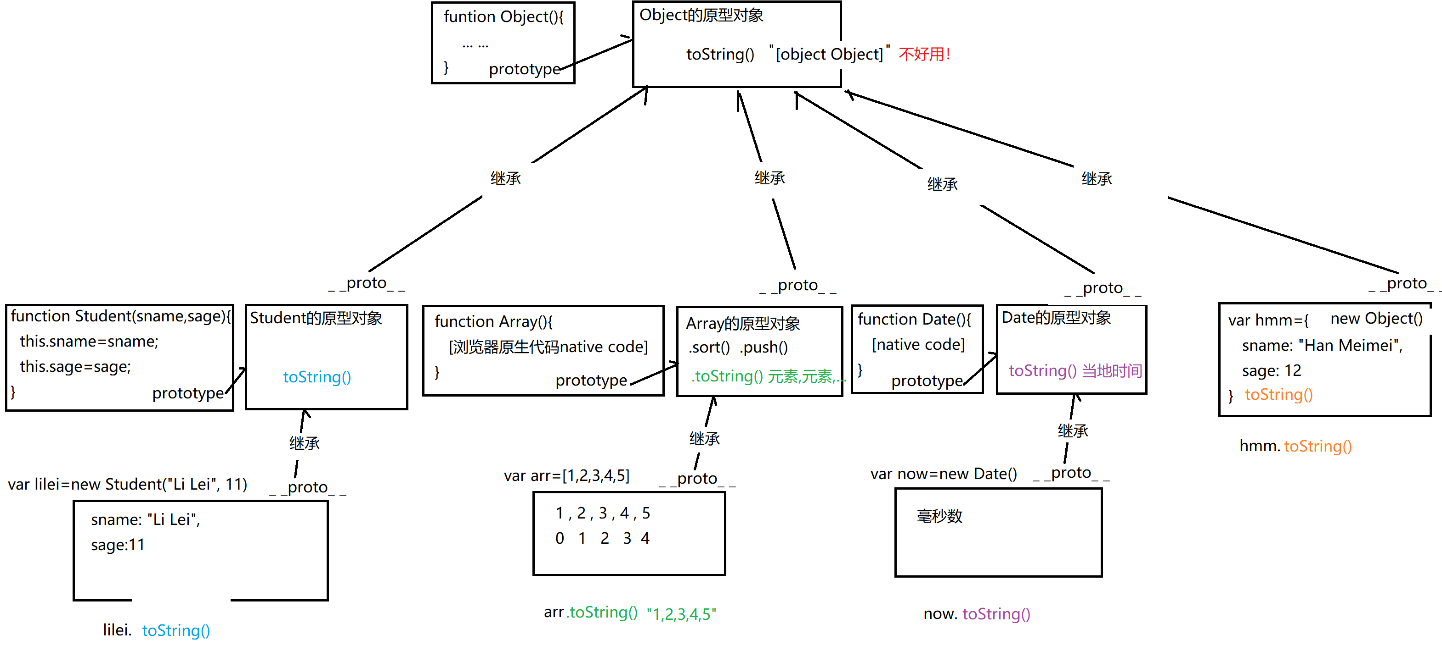
1. 多态
2. 自定义继承
3. 什么是ES5
4. 严格模式
5. 保护对象
6. 多态:
7. 什么是: 同一个函数在不同情况下表现出不同的状态
8. 其实: 多态包好: 重载(overload)和重写(override)
9. 重点讲重写: override

(1). 什么是: 在子对象中定义一个和父对象中同名的成员，以此来覆盖父对象中同名成员

(2). 为什么: 从父对象继承来的东西，不都是好用的

(3). 何时: 只要从父对象继承来的东西不好用！就可以自己在对象内部定义一个同名的成员，覆盖父对象中的成员.

(4). 如何: 在当前子对象内部定义一个同名的成员，就可覆盖父对象中的成员.



(5). 示例: 为类型和对象重写toString()方法

|  |
| --- |
| function Student(sname,sage){    this.sname=sname;    this.sage=sage;  }  //为学生家所有孩子重写toString()方法,覆盖父类型Object中不好用的toString()  Student.prototype.toString=function(){    console.log(`调用父对象Student的原型对象中重写的toString()方法`)    return `{ sname: "${this.sname}", sage:${this.sage} }`  }  var lilei=new Student("Li Lei",11);  var arr=[1,2,3,4,5];  var hmm={    sname: "Han Meimei",    sage: 12,    //为hmm自己重写toString()方法,覆盖父类型Object中不好用的toString()    toString:function(){      console.log(`调用当前对象自己内部重写的toString()方法`)      return `{ sname: "${this.sname}", sage:${this.sage} }`    }  };  var now=new Date();  console.log(lilei.toString());//"[object Object]"  console.log(arr.toString());//"1,2,3,4,5"  console.log(hmm.toString());//"[object Object]"  console.log(now.toString());//Mon Mar 16 2020 10:07:08 GMT+0800 |

1. 自定义继承:
2. 什么是: 其实如果觉得一个对象的整个父对象都不好用，就可以更换这个子对象的父对象
3. 何时: 程序中，只要觉得当前的原型对象里没有自己想要的成员，想用别的类型的原型对象里的成员，就可以更换当前对象的\_\_proto\_\_指向新的父对象。
4. 如何:

(1). 只修改一个对象的父对象

a. 不推荐做法: 对象.\_ \_proto\_ \_=新父对象

问题: \_ \_proto\_ \_有的浏览器不让用！——兼容性问题

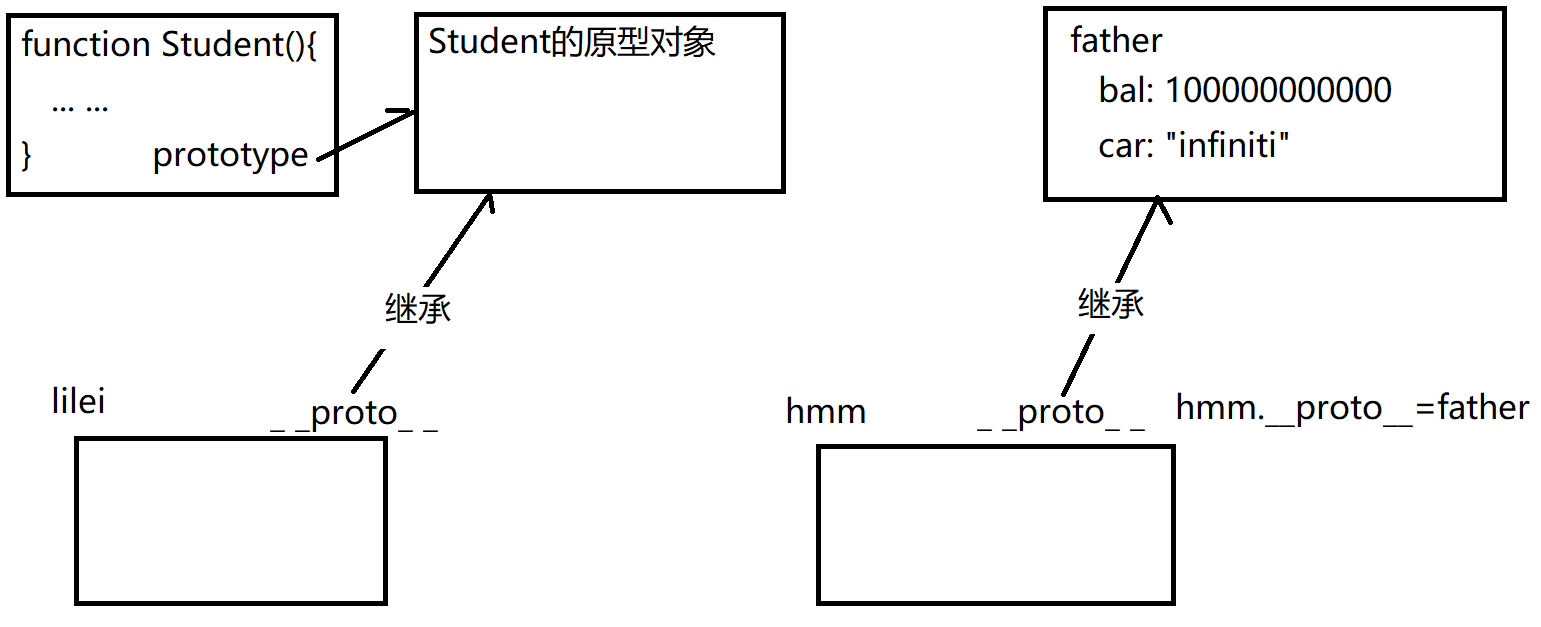
b. 用专门的函数来换爹:

1) Object.setPrototypeOf(子对象, 新的父对象)

设置 原型 的 子对象 (为)新的父对象

2). 以为设置子对象的爹(\_\_proto\_\_属性)指向新的父对象，功能等效于对象.\_ \_proto\_ \_=新父对象

c. 示例: 为hmm换爹

s

|  |
| --- |
| function Student(sname, sage){    this.sname=sname;    this.sage=sage;  }  var lilei=new Student("Li Lei",18);  var hmm=new Student("Han Meimei",19);  console.log(lilei);  console.log(hmm);  //新的父对象  var father={    bal:10000000000000,    car:"infiniti"  }  //hmm想继承新的父对象，从而使用新父对象中自己想要的成员  //hmm.\_\_proto\_\_=father;  Object.setPrototypeOf(hmm,father)  console.log(lilei.bal, lilei.car);  console.log(hmm.bal, hmm.car); |

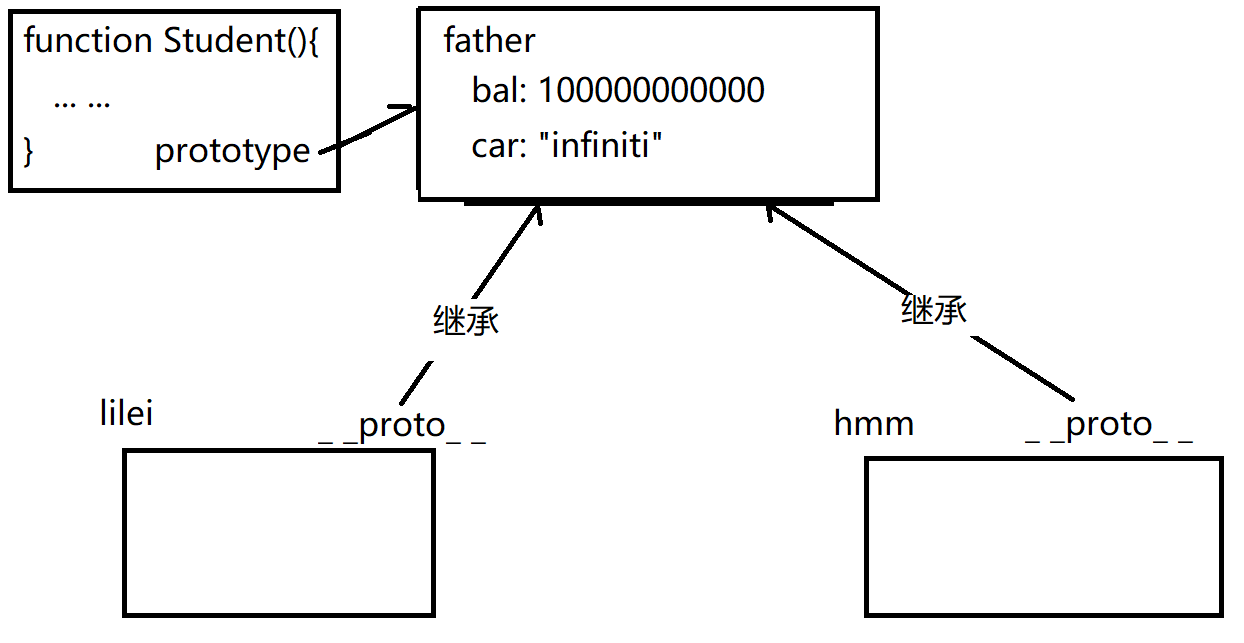
(2). 同时更换多个子对象的爹：

a. 其实就是修改构造函数的prototype属性指向新的对象

构造函数.prototype=新原型对象

b. 一定要讲究时机: 一定要在创建第一个子对象之前就要更换！

c. 示例: 更改构造函数的原型对象为新对象



|  |
| --- |
| //新的父对象  var father={    bal:10000000000000,    car:"infiniti"  }  function Student(sname, sage){    this.sname=sname;    this.sage=sage;  }  //修改Student构造函数的原型对象为father  Student.prototype=father;  var lilei=new Student("Li Lei",18);  var hmm=new Student("Han Meimei",19);  console.log(lilei);  console.log(hmm);  console.log(lilei.bal, lilei.car);  console.log(hmm.bal, hmm.car); |

1. 什么是ES5
2. 什么是ES5: 是ECMAScript标准发展到第五个版本
3. 为什么:

(1). 解决js语言中很多广受诟病的缺陷

(2). 学习其他语言，为js添加新的功能

1. 严格模式:
2. 什么是严格模式: 比普通的js运行机制要求更严格的模式:
3. 为什么: 因为js语言本身有很多广受诟病的缺陷
4. 何时: 今后所有项目都必须在严格模式下运行！
5. 如何启用严格模式: 在当前代码段的顶部添加:　“use strict”

启用 严格(模式)

1. 新规定:

(1). 禁止给未声明的变量赋值: ——优点: 减少全局污染和内存泄漏的几率

a. 旧js中: 随意给未声明过的变量赋值，不会报错！而是在”全局”自动创建该变量

b. 问题: 会造成全局污染和内存泄漏

c. 示例: 尝试给未声明过的变量赋值

|  |
| --- |
| "use strict";  //想写也给函数可以给自己的女朋友发一条私密的消息  function send(){    //var gf;    //qgf="今晚308,w84u";//假如拼写错误，已经不让使用变量了    gf="今晚308,w84u";    console.log(gf);  }  send();  console.log(qgf);  //外人应该不知道！  console.log(gf); |

(2). 静默失败升级为错误: 便于调试，减少了歧义

a. 什么是静默失败: 执行不成功，但是又不报错！

b. 问题: 极其不便于调试！

c. 示例: 尝试修改对象的只读属性

|  |
| --- |
| "use strict";  var eric={    eid:1001, //要求员工编号只读    ename:"埃里克"  }  //一会儿要讲的办法:(先不要问为什么)  //可以让eric的eid属性只读  Object.defineProperty(eric,"eid",{    writable:false  // 可修改   不  })  eric.eid=-2;  //报错:Cannot assign to read only property 'eid' of object '#<Object>'  //     不能    赋值  给    只读    属性      eid  的   对象  console.log(eric); |

(3). 匿名函数自调或普通函数调用中的this，不再指向window，而是值为undefined

a. 旧js中: 匿名函数自调或普通函数调用中的this->window

b. 问题: 极容易造成内存泄漏和全局污染

c. 示例: 使用构造函数时，不写new

|  |
| --- |
| "use strict";  function Student(sname,sage){    this.sname=sname;    this.sage=sage;  }  var lilei=new Student("Li Lei",11); //this->new  var hmm=Student("Han Meimei",12); //this->undefined  console.log(lilei);  console.log(hmm);  console.log(window); |

(4). 禁用了arguments.callee

a. 什么是arguments.callee：

1). 每个函数内自带的

2). 专门获得当前函数自己的关键词

b. 何时: 通常使用递归算法时，都用arguments.callee代替函数内写死的函数名

c. 为什么: 旧的递归有一个问题: 函数内再次调用函数本身时，函数名是写死的

不好: 紧耦合，如果函数名变了，就被迫连函数体内的名称也要跟着变！如果函数名变了，忘记修改函数内的写死的函数名，立刻就出错了！

d. 解决: 松耦合:

1). 在函数内，用arguments.callee代替写死的函数名

2). 运行时，arguments.callee可自动获得当前函数自己

vs this: 自动获得当前函数之前.前的对象

3). 将来即使外边函数名变化,arguments.callee也能自动获得当前函数的新名

e. 示例: 使用递归算法和arguments.callee实现斐波那契数列

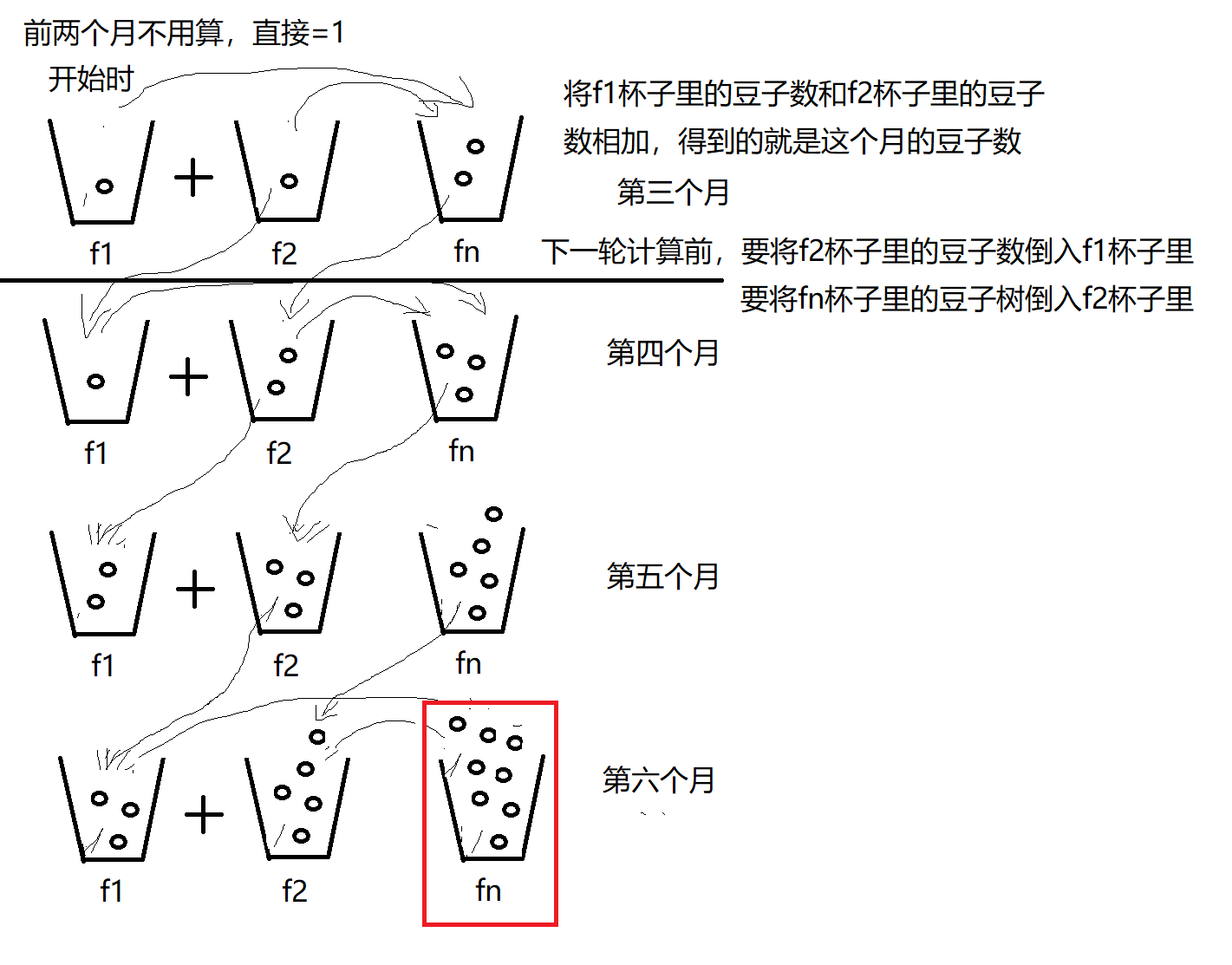
|  |
| --- |
| //斐波那契数列:  //养兔子: 第一个月只有一对兔子，兔子三个月成熟可下一对儿小兔子  //兔子的对儿数:  //月份: 1  2  3  4  5  6  7  8  9  10 ...  //对数: 1  1  2  3  5  8 13 21 34  55 ...  //结论: 每个月的兔子对数，等于相邻的前两个月兔子对儿数之和  //数学公式: f(n)=f(n-1)+f(n-2)  //n=1和n=2时，f(n)都等于1  function fib(n){    if(n<3){      return 1;    }else{      //    递归    递归      //return f(n-1)+f(n-2); //紧耦合      //argument.callee->fib函数      return arguments.callee(n-1)+arguments.callee(n-2)    }  }  console.log(fib(10));//55 |

f. 为什么禁用？不推荐使用递归算法了，但是也没有完全禁止

1). 递归算法: 效率极低，因为重复计算量太大！

g. 解决: 今后大部分递归算法，都可用循环代替！(难度非常高)

h. 示例: 使用循环代替递归实现斐波那契算法



|  |
| --- |
| function fib(n){    if(n<3){      return 1;    }else{      //准备三个杯子      var f1=1, f2=1, fn;      //反复执行相同的操作(不是从第一个月开始循环，而是从第三月开始计算)      for(var i=3;i<=n;i++){        //每次先将f1的值和f2的值相加,放入fn中        fn=f1+f2;        //为下一轮循环做准备:        f1=f2;//等号赋值是从右向左倒水        //将f2的值给f1，再将fn的值给f2        f2=fn;      }      //循环结束，fn变量中的值，就是对应月份的值      return fn;    }  }  console.log(fib(1000));//55 |

1. 总结: 严格模式四个新的规定

(1). 禁止给未声明的变量赋值

(2). 静默失败升级为错误

(3). 匿名函数或普通函数调用中的this不再指向window，而是undefined

(4). 禁用了arguments.callee

1. 保护对象:
2. 什么是保护对象: 禁止对对象的属性值或对象结构执行一些非法的操作
3. 为什么保护对象: 因为旧js中，对象毫无自保能力。可以对对象的属性进行任何操作，可以随时添加删除属性。
4. 何时保护对象: 今后只要不希望用户对对象的内容和结构执行非法的操作时，都要对对象提供保护
5. ES5中，从保护属性和保护结构两个方面为保护对象提供了工具
6. 保护单个属性:

(1). 对象中的属性不再只是一个值而已，每个属性都变成了一个缩微的小对象

(2). 每个缩微的属性小对象中都包含四个特性:

a. value: 实际为属性保存属性值

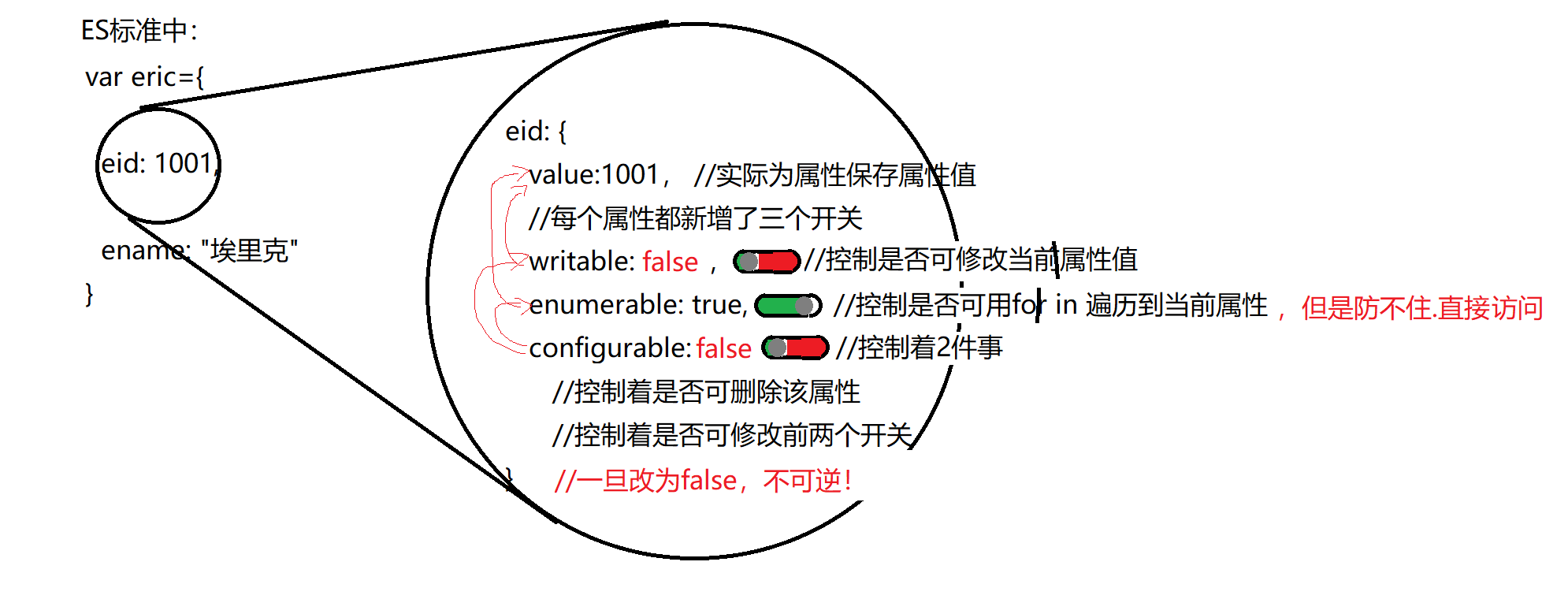
b. writable: true, 控制是否可修改当前属性值

c. enumerable: true, 控制是否可用for in遍历到这个属性

d. configurable: true, 控制2件事

1). 是否可删除该属性

2). 是否可修改前两个开关特性



(3). 如何输出一个属性的小对象，查看值或三个开关的状态: (了解)

a. var 属性小对象=Object.getOwnPropertyDescriptor(对象,”属性名”);

b. 意为: 获得指定对象中一个指定的属性的小对象:

(4). 如何修改一个属性的小对象中的开关呢？

a. 错误做法: ~~不能用.直接访问，比如: eric.eid.writable=false~~

b. 正确的做法: 必须用专门的函数:

1). 只修改一个属性的开关:

i. Object.defineProperty(对象, “属性名”, {

开关: true或false,

… : …

})

ii. 修改指定对象的指定属性小对象中的某个开关值为true或false

iii. 其实: 修改writable和enumerable开关时，都会同时设置configurable:false，作为双保险。因为configurable:false，既保护了前两个开关，又不可逆，防止重新打开！

iv. 问题: Object.defineProperty()一次只能修改一个属性的特性，如果一个对象中，多个属性的开关需要修改，则代码会很繁琐

v. 示例: 使用defineProperty修改多个属性的开关

|  |
| --- |
| "use strict";  var eric={ //公司要求    eid:1001, //eid只读    ename:"埃里克", //禁止删除ename属性    salary: 12000 //禁止用for in随便遍历  }  //尝试输出eric对象的eid属性的小对象  var obj=Object.getOwnPropertyDescriptor(eric,"eid");  //           获得 自己的属性的描述信息  console.log(obj);  //想让eid只读:  Object.defineProperty(eric,"eid",{      //  定义  属性    writable:false,    //可修改    configurable:false,//双保险    //配置  })  //想禁止删除ename属性  Object.defineProperty(eric,"ename",{    configurable:false  })  //想禁止for in循环遍历到salary属性  Object.defineProperty(eric,"salary",{    enumerable:false,    //可遍历    configurable:false, //双保险  })  //尝试把已经关闭的开关再打开  // Object.defineProperty(eric,"eid",{  //   writable:true,  //   configurable:true //不可逆  // })  //故意修改eid  // eric.eid=-2;  //故意删除ename属性  //delete eric.ename;  //就想尝试看eric的所有隐私  for(var key in eric){    console.log(`${key} : ${eric[key]}`)  }  //enumerable只能防住for in，防不住.直接访问  console.log(eric.salary); //js语言不严格，所以不要和其他严格的语言做对比。  console.log(eric); |

2). 同时修改一个对象的多个属性的开关:

i. Object.defineProperties(对象,{

属性名1:{

开关: true或false,

… : …

},

属性名2:{

… : …

}

})

ii. 意为: 同时修改指定对象中多个属性的开关

iii. 示例: 使用defineProperties同时修改一个对象的多个属性中的开关

|  |
| --- |
| "use strict";  var eric={ //公司要求    eid:1001, //eid只读    ename:"埃里克", //禁止删除ename属性    salary: 12000 //禁止用for in随便遍历  }  //想让eid只读:  //想禁止删除ename属性  //想禁止for in循环遍历到salary属性  Object.defineProperties(eric,{    eid:{      writable:false,      configurable:false,    },    ename:{      configurable:false    },    salary:{      enumerable:false,      configurable:false    }  })  //尝试把已经关闭的开关再打开  // Object.defineProperty(eric,"eid",{  //   writable:true,  //   configurable:true //不可逆  // })  //故意修改eid  // eric.eid=-2;  //故意删除ename属性  // delete eric.ename;  //就想尝试看eric的所有隐私  for(var key in eric){    console.log(`${key} : ${eric[key]}`)  }  //enumerable只能防住for in，防不住.直接访问  console.log(eric.salary); //js语言不严格，所以不要和其他严格的语言做对比。  console.log(eric); |

c. 问题: 仅靠开关，无法用自定义规则保护属性值

比如: 年龄必须介于18~65之前，三个开关都不能保护这个规则

//value不负责保护，仅负责保存属性值

//writable: 禁止修改，但是年龄可以修改，只不过值的范围人为规定而已

//enumerable: 跟值无关，是用来防止for in遍历的

//configurable: 跟值无关，是用来防止删除属性

(5). 访问器属性:

a. 什么是访问器属性: 自己不保存属性值，仅提供对另一个数据属性的保护。

b. 何时: 今后只要想用自定义的复杂规则保护属性值时，都用访问器属性

c. 如何定义访问器属性: 2步:

1). 将实际存储属性值的受保护的属性隐姓埋名并且半隐藏

Object.defineProperty(对象,”\_原属性名”,{

value: 原属性值,

enumerable:false,

configurable:false,

})

2). 请保镖，保镖要冒名顶替受保护属性的原属性名

Object.defineProperty(对象,”原属性名”,{

//因为保镖不实际存储属性值，所以保镖没有~~value属性~~

//又因为writable无法保护属性，才采用的保镖方式，所以保镖也没有~~writable属性~~

//保镖一请就是一对儿

get:function(){//专门负责从受保护的原属性中获取属性值，交给外人

return this.\_原属性名

},

set:function(value){ //专门负责接受外部的新值，判断是否能修改进原属性中保存

//value会自动接住外人给的新值

if(value符合条件){

才将新值value保存到原属性中

this.\_原属性名=value

}else{//否则如果新值不符合要求，则报错，且不修改原属性值

throw Error(“错误提示”)

}

},

//但是: 保镖依然后enumerable和configurable两个开关  
 enumerable: true, //保镖要代替原属性抛头露面

configurable: false, //保镖不能轻易被删除

})

d. 如何使用访问器属性:

1). 每当外人试图获取属性值时，会自动调用保镖的get()函数

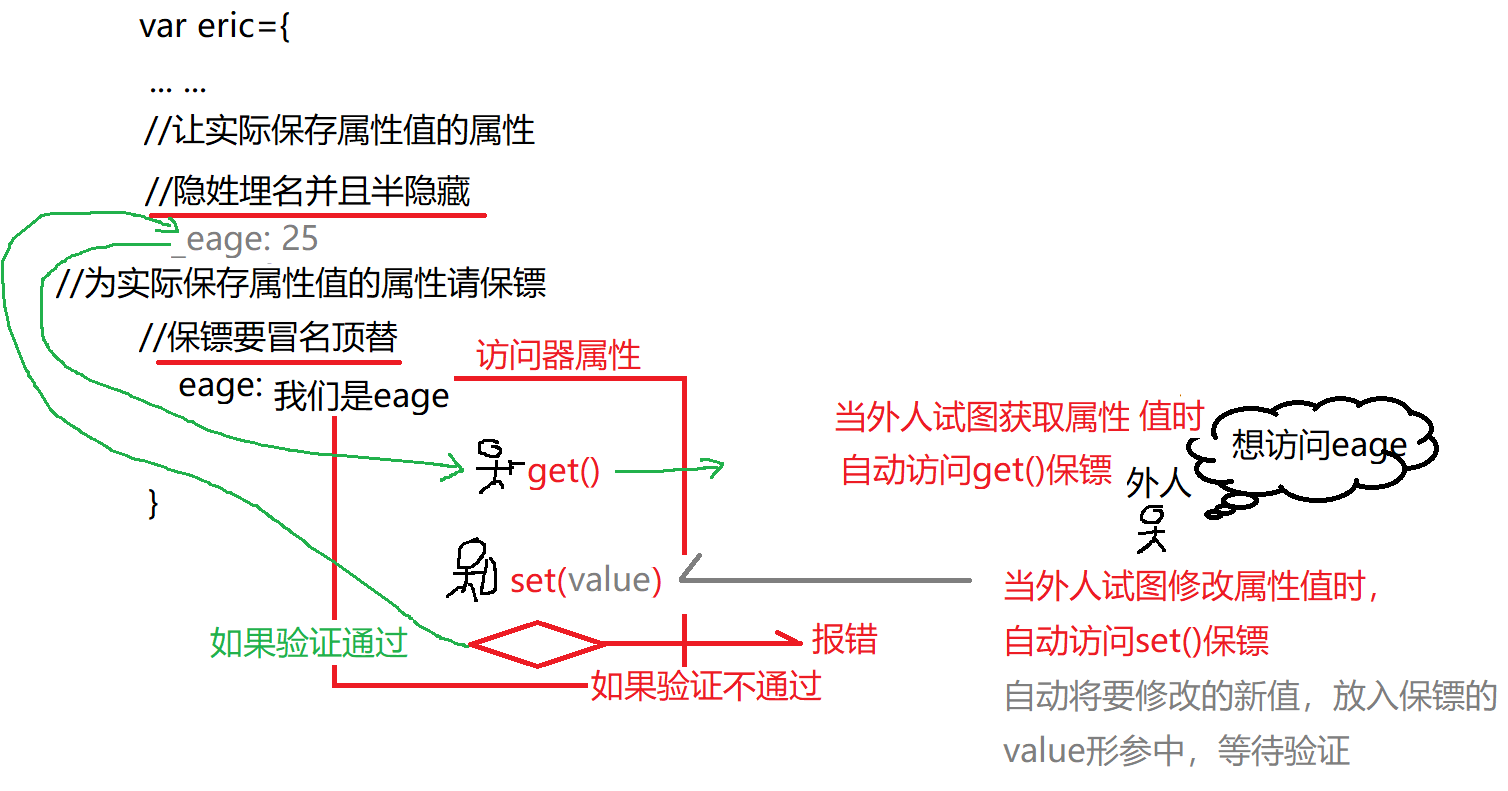
2). 每当外人试图修改属性值时，会自动调用保镖的set()函数

且会将要修改的新值，先交给set()的value参数

只有验证value通过，才将新值保存回受保护的属性中

如果验证value不通过，则报错，且不保存新值

e. 示例: 使用访问器属性保护员工年龄属性



|  |
| --- |
| var eric={    eid:1001,    ename: "埃里克",    eage: 25 //年龄虽然可以修改，但是必须介于18~65岁直接  }  //value不负责保护，仅负责保存属性值  //writable: 禁止修改，但是年龄可以修改，只不过值的范围人为规定而已  //enumerable: 跟值无关，是用来防止for in遍历的  //configurable: 跟值无关，是用来防止删除属性  //使用访问器属性:  //新增一个隐姓埋名的半隐藏的实际保存属性值的属性  Object.defineProperty(eric,"\_eage",{    value:eric.eage,    writable:true,//收保护的属性值可以修改，只不过要经过验证才能修改    enumerable:false,    configurable:false  })  //为受保护的属性请保镖，保镖要冒名顶替受保护的属性  Object.defineProperty(eric,"eage",{    //没有value，writable    get:function(){      console.log(`自动调用eage的get(),返回${this.\_eage}`);      return this.\_eage;    },    set:function(value){      console.log(`自动调用eage的set(),并收到value=${value}进行验证`)      if(value>=18&&value<=65){        this.\_eage=value;      }else{        throw Error("年龄必须介于18~65之间")      }    },    enumerable:true,    configurable:false  })  //外人试图获取eage的属性值  console.log(eric.eage); //自动调用eage的get()  //外人试图修改eage的属性值  eric.eage=26;  console.log(eric);  eric.eage=15; |

(5). 因为js是不严格的语言，所以，一旦外人知道受保护的半隐藏的属性名，依然可用.直接访问，绕过保镖！

总结: 面向对象: 三大特点: 封装 继承 多态

1. 封装:

(1). 只创建一个对象的时候: {}

(2). 反复创建多个相同解构的对象时: 构造函数

a. 定义构造函数:

function 类型名(形参1, 形参2, …){

this.属性名=形参1;

this.属性名=形参2;

/\*构造函数中决不能再放方法定义\*/

}

b. 调用构造函数:

var 新子对象=new 构造函数(属性值1, 属性值2, …)

1. 继承:

多个子对象公用的方法，都要定义在该类型的原型对象上

构造函数.prototype.共有方法=function(){

… this.属性名 …

}

总结: new做了几件事

1. 创建新的空对象
2. 让新的空对象继承构造函数的原型对象
3. 调用构造函数，并临时将构造函数中的this都替换为当前正在创建的新对象

用强行赋值的方式，为新对象添加新属性

1. 返回新对象地址保存到等号左边的变量中

总结: this 4种情况

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. 类型名.prototype.共有方法=function(){ … } this->将来调用这个共有方法的某一个子对象

作业:

1. 看小程序视频完成访问器属性练习:

小程序->在线->JSCORE->day05: 5. 访问器属性

1. (扩展)看小程序视频实现两种类型间的继承:

小程序->在线->JSCORE->day05: 作业: 笔试题: 实现两种类型间的继承 animal people

1. (扩展)看小程序视频实现一个最像java的严格的构造函数

小程序->在线->JSCORE->day05: 作业: 高频笔试题: 如何定义一个像java的严格的构造函数