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->day05、day06

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->day04、day05

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->day05

作业: 笔试题: 实现两种类型间的继承

作业: 高频鄙视题: 如何定义一个更像Java的严格的构造函数。

3. 单词列表: 小程序->单词->JSCORE->day05、day06

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！

4. （这六天）必须用画图软件亲手画图！

如何提问: （vip待遇的剩余时间15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:

\*\*\*\*面向对象\*\*\*\*

1. 自定义继承

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

1. 严格模式

2. 保护对象

3. Object.create

4. 替换this

\*\*\*\*面向对象\*\*\*\*

一. 自定义继承

对应小程序视频: 小程序->在线->JSCORE->day05 2. 自定义继承...

1. 问题: 如果整个父对象都不是想要的！

解决: 其实可以修改一个对象的父对象——换爹

2. 如何: 2种:

(1). 只更换一个对象的父对象: 其实就是修改对象的\_ \_proto\_ \_继承新的父对象

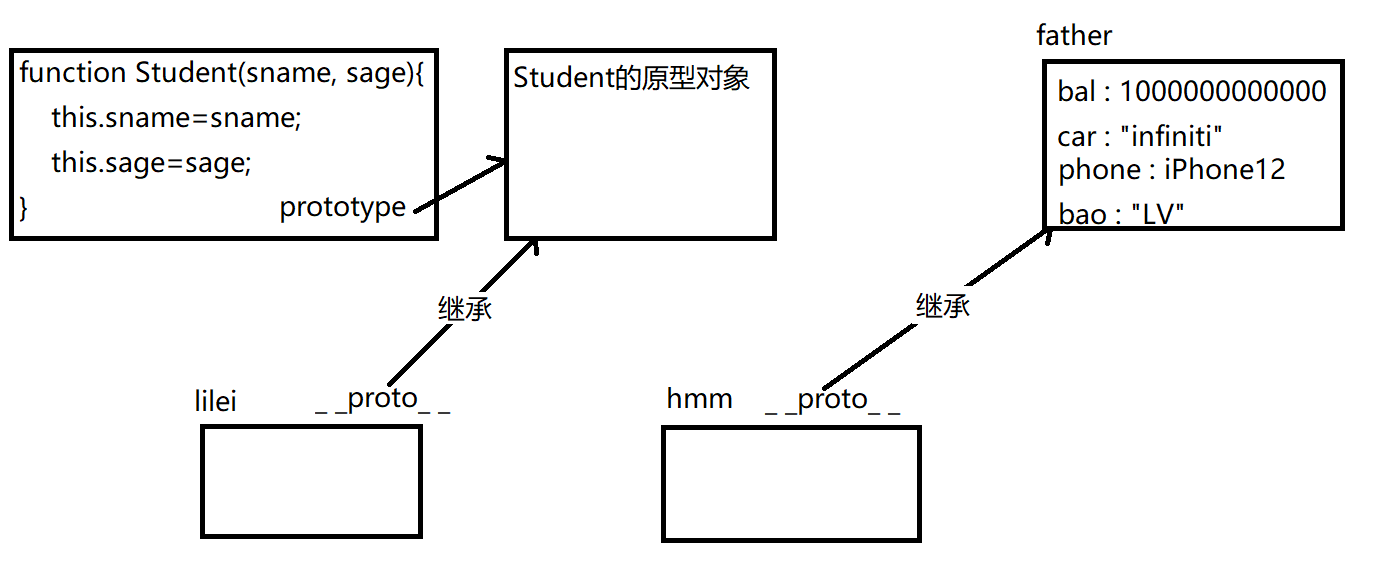
a. 子对象.\_ \_proto\_ \_=新父对象

b. 问题: 有些浏览器不允许擅自修改\_ \_proto\_ \_

解决: Object.setPrototypeOf(子对象, 新父对象) —— 今后建议

设置原型对象的子对象(为)新父对象

c. 示例: 仅更换hmm一个对象的爹

  
 1\_setPrototypeOf.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 Student(sname,sage){        this.sname=sname;        this.sage=sage;      }      var lilei=new Student("Li Lei",18);      var hmm=new Student("Han Meimei",19);      var father={        bal:100000000000,        car:"infiniti"      }      console.log(lilei);      console.log(hmm);      //hmm修改自己的\_ \_proto\_ \_属性指向father      // hmm.\_\_proto\_\_=father;      Object.setPrototypeOf(hmm, father);      //比较:      console.log(hmm.bal, hmm.car);      console.log(lilei.bal, lilei.car);    </script>  </body>  </html>  运行结果: |

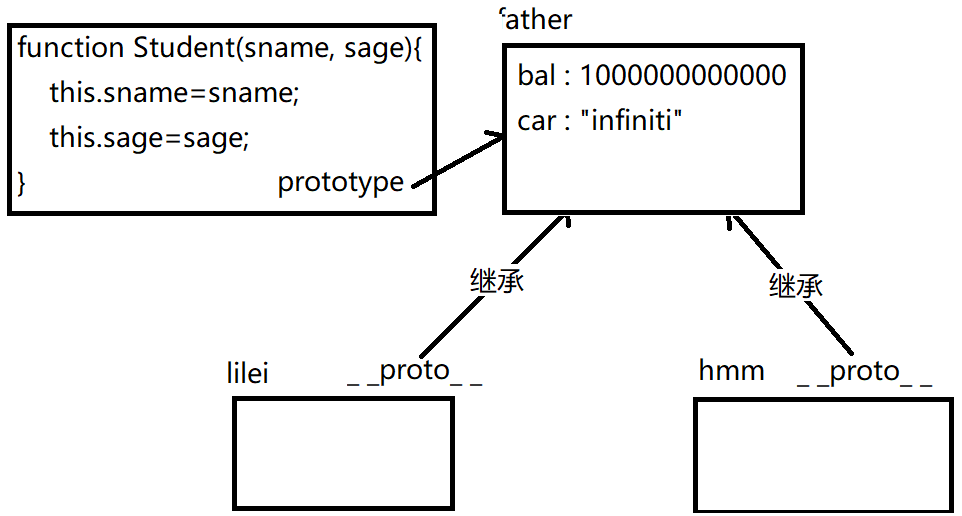
(2). 更换所有子对象的父对象:

a. 其实就是更换构造函数的prototype属性:

构造函数.prototype=新父对象

b. 时机: 必须在创建子对象之前就要更换！

c. 示例: 同时更换lilei和hmm的父对象



2\_prototype.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 Student(sname,sage){        this.sname=sname;        this.sage=sage;      }        var father={        bal:100000000000,        car:"infiniti"      }      //在生lilei和hmm之前就换原型对象:      Student.prototype=father;      var lilei=new Student("Li Lei",18);      var hmm=new Student("Han Meimei",19);      console.log(lilei);      console.log(hmm);      //结果:      console.log(lilei.bal, lilei.car);      console.log(hmm.bal, hmm.car);    </script>  </body>  </html>  运行结果: |

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

ES5: ECMAScript标准的第五个版本

一. 严格模式：

对应小程序视频: 小程序->在线->JSCORE->day05 3. ES5 严格模式...

1. 什么是严格模式: 比普通的js运行机制要求更严格的新的运行机制。

2. 为什么: js语言本身有很多广受诟病的缺陷

3. 何时使用严格模式: 今后几乎所有的js程序都必须运行在严格模式下

4. 如何启用严格模式: 在当前作用域的顶部: "use strict";

启用 严格

5. 新规定:

(1). 禁止给未声明过的变量赋值:

a. 旧js中: 给未声明的变量赋值，不报错，而是在全局自动创建该变量

b. 缺点: 极容易造成全局污染和歧义！

c. 启用严格模式: 禁止给未声明过的变量赋值——报错！

d. 好处: 大大减少了全局污染的可能！也避免了部分拼写错误！便于调试

e. 示例: 尝试给未声明的变量赋值:

3\_use\_strict.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>      "use strict";//启用严格模式      function send(){        var gf;        //想给我的女朋友发一条私信        //gf="今晚308,w84u";        //不小心错发给了前女友        qgf="今晚308,w84u";//严格模式下，会报错！避免了误会发生        console.log(gf);      }      send();      //公开！      console.log(qgf);      //console.log(gf);//报错！    </script>  </body>  </html>  运行结果:  Uncaught ReferenceError: qgf is not defined  at send (3\_use\_strict.html:17)  at 3\_use\_strict.html:20 |

(2). 静默失败升级为错误:

a. 什么是静默失败: 程序执行不成功，但也不报错！

b. 缺点: 极其不便于程序的调试

c. 启用严格模式: 几乎将所有静默失败都升级为了错误

d. 好处: 便于程序调试！

e. 示例: 演示静默失败

4\_use\_strict.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>      "use strict";      var eric={        eid:1001,        ename:"埃里克"      };      //要求员工的编号不能修改！      //设置eric对象的eid属性为只读      Object.defineProperty(eric,"eid",{        writable:false      });//先不要问，稍后重点讲      //试图修改eric的eid属性      eric.eid=-2; //报错：      //Cannot assign to read only property 'eid'      // 不能    赋值  给    只读      属性    eid      console.log(eric);    </script>  </body>  </html>  运行结果：  Uncaught TypeError: Cannot assign to read only property 'eid' of object '#<Object>'  at 4\_use\_strict.html:24 |

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

a. 旧js中: 普通函数中的this默认指window

b. 缺点: 极大的增加了全局污染的概率！

c. 启用严格模式: 普通函数调用中的this不再指向window，而是undefined。

d. 好处: this后不能随便加.xxx，加上就报错！有效的防止了因为this造成的全局污染！

e. 示例: 演示错误的使用构造函数

5\_use\_strict.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>      "use strict";      function Student(sname,sage){        this.sname=sname;//报错: Cannot set property 'sname' of undefined        //this->undefined，不能加.sname，加上就错！        this.sage=sage;        //没有return      }      var lilei=new Student("Li Lei",11);      //错误的使用了构造函数:      var hmm=Student("Han Meimei",12);              //this->window              //this.sname -> window.sname="Han Meimei"              //this.sage -> window.sage=12      console.log(lilei);      console.log(hmm);      console.log(sname);//全局      console.log(sage);//全局      console.log(window);    </script>  </body>  </html>  运行结果：  Uncaught TypeError: Cannot set property 'sname' of undefined  at Student (5\_use\_strict.html:13)  at 5\_use\_strict.html:20 |

(4). 禁用arguments.callee

a. 什么是arguments.callee: 是在一个函数内自动获得当前函数本身 的关键词

b. 何时: 递归调用时

c. 为什么: 如果在递归调用时，在函数内写死函数名，是非常不好的

因为: 紧耦合，万一外部函数名发生了变化，总要记得修改内部写死的函数名，一旦忘记修改内部写死的函数名，程序立刻出错！

d. 所以: 将来就算用递归算法，也不能在函数内写死函数名。应该用arguments.callee代替写死的函数名，自动获得当前函数本身。与函数名无关了！

e. 好处: 松耦合: 从此即使修改外部的函数名，函数内部一点都不用修改！保持不变即可！

f. 为什么禁用: 其实是不推荐使用递归调用。

g. 递归调用的问题: 重复计算量太大！

h. 解决: 绝大多数的递归都可以用循环来解决——难度极高！

（了解）小程序->在线->JSCORE->day05 3.1 严格模式 斐波那契数列

i. 总结:

1). 如果使用递归算法可以轻松解决问题，又不影响效率，完全可以首选递归算法。只不过不要用arguments.callee了！依然在函数中写死函数名。

2). 只有用递归算法非常影响效率时，才被迫寻找循环算法代替递归。

j. 示例: 使用递归实现斐波那契数列:

6\_use\_strict.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>      "use strict";      var eric={        eid:1001,        ename:"埃里克",        salary:12000      }      //系统要求: eid只读, ename禁止删除, salary禁止遍历      Object.defineProperties(eric,{        eid:{          writable:false,          configurable:false        },        ename:{          configurable:false        },        salary:{          enumerable:false,          configurable:false        }      })        //试图重新打开被关闭的writable开关！      // Object.defineProperty(eric,"eid",{      //   writable:true,      //   configurable:true //不可逆      // });//报错: Cannot redefine property: eid      // //         不能   重新定义   属性      eid      //试图修改eid      //eric.eid=-2; //报错: Cannot assign to read only property 'eid'      //试图删除ename      //delete eric.ename; //报错: Cannot delete property 'ename'      //试图遍历eric所有属性(包括salary)      for(var key in eric){        console.log(`属性名: ${key}, 属性值:${eric[key]}`)      }      console.log(eric);      //用.强行访问salary      console.log(`eric.salary=${eric.salary}`)    </script>  </body>  </html>  运行结果：  Uncaught TypeError: 'caller', 'callee', and 'arguments' properties may not be accessed on strict mode functions or the arguments objects for calls to them  at f (6\_use\_strict.html:26)  at 6\_use\_strict.html:30 |

二. 保护对象: 阻止别人的程序对我们自己的对象执行不合理的非法的修改操作。

1. 保护属性:

对应小程序视频: 小程序->在线->JSCORE->day05 4. 保护对象属性

(1). 问题: js中的对象毫无自保能力！外界的程序可以随意修改对象中的属性

(2). 解决:

a. ES5标准中，每个对象中的每个属性，都变成一个缩微的小对象

b. 每个属性小对象中，都包含四个更小的属性:

1). value: 为当前属性保存属性值

2). writable: 开关，控制是否可修改当前属性值

3). enumerable: 开关，控制着是否可用for in遍历到该属性

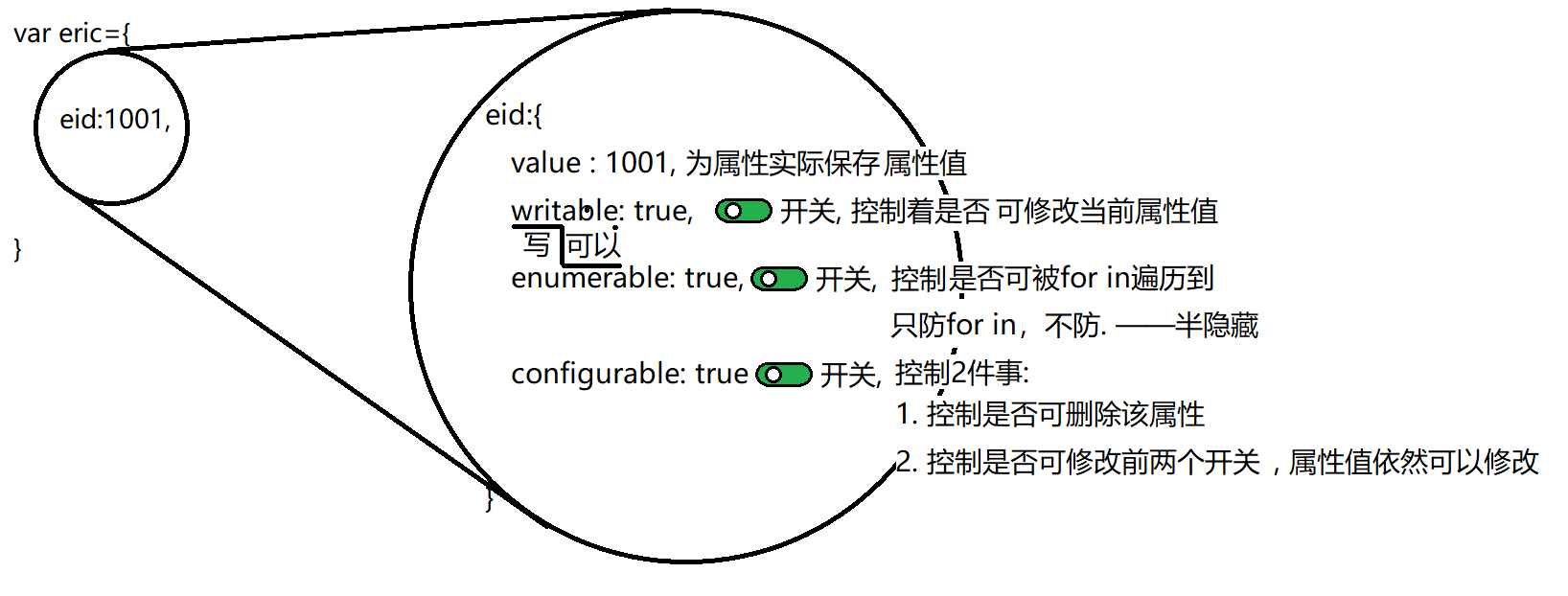
强调: 只防for in，防不住.——半隐藏，浅紫色

4). configurable: 开关，控制2件事

i. 控制着是否可删除当前属性

ii. 控制着是否可修改前两个开关

强调: configurable开关，一旦改为false！不可逆！任何其它程序，都无权重新打开开关！除非修改源代码！



(3). 如何修改开关的值来限制对属性的操作:

~~a. 错误: 细微的开关属性，不能用.直接访问！~~

b. 只能用专门的函数:

1). 只修改一个属性中的多个开关:

(重新)定义 属性

i. Object.defineProperty(对象名, "属性名",{

开关名: true或false

... : ...

});

ii. 问题: 我们关上的开关，别人可以随意再打开——相当于没有保护！

iii. 解决: 今后只要修改writable和enumerable开关时，都要同时设置configurable:false，作为双保险！禁止任何程序再打开开关！

iv. 示例: 使用defineProperty()操作属性的开关

7\_defineProperty.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>      "use strict";      var eric={        eid:1001,        ename:"埃里克",        salary:12000      }      //系统要求:      //eid只读,      Object.defineProperty(eric,"eid",{        writable:false,        configurable:false //不可逆 //双保险      });      //ename禁止删除,      Object.defineProperty(eric,"ename",{        configurable:false      })      //salary禁止遍历      Object.defineProperty(eric,"salary",{        enumerable:false,        configurable:false //双保险      })        //试图重新打开被关闭的writable开关！      // Object.defineProperty(eric,"eid",{      //   writable:true,      //   configurable:true //不可逆      // });//报错: Cannot redefine property: eid      // //         不能   重新定义   属性      eid      // //试图修改eid      // eric.eid=-2; //报错: Cannot assign to read only property 'eid'      //试图删除ename      //delete eric.ename; //报错: Cannot delete property 'ename'      //试图遍历eric所有属性(包括salary)      for(var key in eric){        console.log(`属性名: ${key}, 属性值:${eric[key]}`)      }      console.log(eric);    </script>  </body>  </html>  运行结果:  属性名: eid, 属性值:1001  属性名: ename, 属性值:埃里克  {eid: 1001, ename: "埃里克", salary: 12000} |

v. 问题: Object.defineProperty()一次只能修改一个对象中的属性。如果对象中有很多属性都需要保护，则反复写Object.defineProperty()太麻烦了！

2). 同时修改多个属性的多个开关

a. Object.defineProperties(对象名,{

属性名:{

开关: true或false,

... : ...

},

属性名:{

开关: true或false,

... : ...

},

... ...

})

b. 示例: 使用defineProperties()操作属性的开关

8\_defineProperties.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>      "use strict";      var eric={        eid:1001,        ename:"埃里克",        salary:12000      }      //系统要求: eid只读, ename禁止删除, salary禁止遍历      Object.defineProperties(eric,{        eid:{          writable:false,          configurable:false        },        ename:{          configurable:false        },        salary:{          enumerable:false,          configurable:false        }      })        //试图重新打开被关闭的writable开关！      // Object.defineProperty(eric,"eid",{      //   writable:true,      //   configurable:true //不可逆      // });//报错: Cannot redefine property: eid      // //         不能   重新定义   属性      eid      //试图修改eid      //eric.eid=-2; //报错: Cannot assign to read only property 'eid'      //试图删除ename      //delete eric.ename; //报错: Cannot delete property 'ename'      //试图遍历eric所有属性(包括salary)      for(var key in eric){        console.log(`属性名: ${key}, 属性值:${eric[key]}`)      }      console.log(eric);      //用.强行访问salary      console.log(`eric.salary=${eric.salary}`)    </script>  </body>  </html>  运行结果:  属性名: eid, 属性值:1001  属性名: ename, 属性值:埃里克  {eid: 1001, ename: "埃里克", salary: 12000}  eric.salary=12000 |

(4). 问题: 开关只能机械的保护一种情况，无法灵活的设定自定义的保护条件

比如: 员工的年龄，可以修改，但是值必须介于18~65之间！

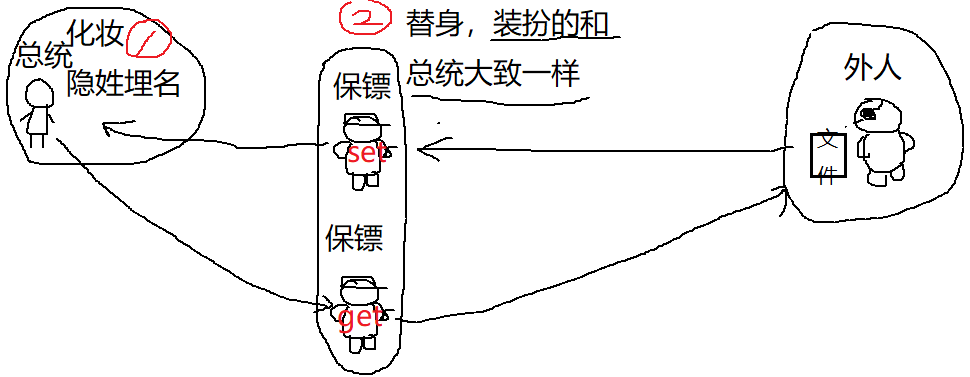
(5). 解决: 访问器属性

对应小程序视频: 小程序->在线->JSCORE->day05 5. 访问器属性

a. 什么是访问器属性: 不实际保存属性值，仅提供对其它数据属性的保护 的特殊属性——保镖！

b. 何时: 如果需要用自定义的规则限制用户对属性的操作时，都要用访问器属性

c. 如何: 2步:



1). 先定义一个隐姓埋名且半隐藏的数据属性实际保存数据:

2). 请保镖:

i. 保镖的名称，必须冒名顶替要保护的属性——迷惑外界

ii. 保镖一请就是一对儿: 名字都固定了：get和set

iii. 因为保镖可以执行验证和加工等操作，所以两个保镖应该都是函数

其中: get函数，专门负责从受保护的数据属性中读取属性值

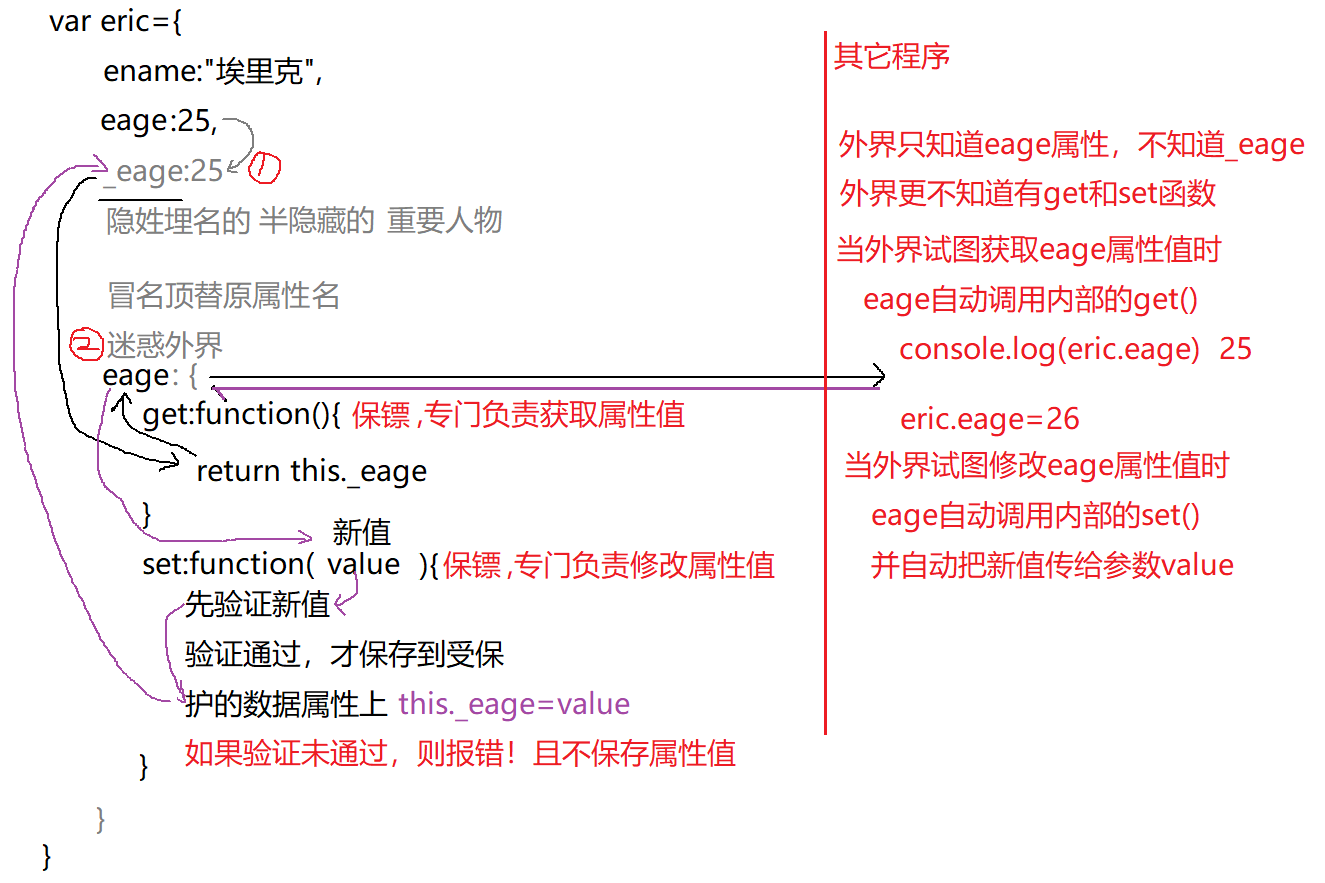
set 函数，专门负责接收新值，并保存到受保护的数据属性上——修改

d. 外界如何使用访问器属性:

1). 前提: 外界不知道有受保护的属性，也不知道自己访问的是访问器属性，更不知道还有get和set函数。外界只知道有一个属性可以获取和修改值，仅此而已。

2). 当外界试图获取属性值时，访问器属性会自动调用get函数，从受保护的属性中取出值

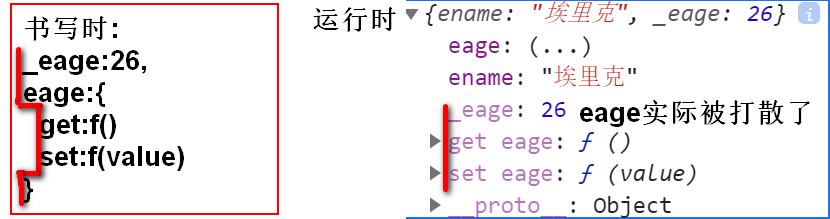
3). 当外界试图修改属性值时，访问器属性会自动调用set函数，并将要修改的新值先传给set函数的value形参，先验证。如果新值验证通过才由set函数保存到受保护的变量中。如果验证不通过，则可以报错，且不保存非法的属性值！



e. 访问器属性中的this为什么不指访问器属性本身，而指当前访问器属性所在对象？

因为访问器属性书写时，是放在一个{}中，但是，一旦进入对象，就被打散了！get和set两个函数直接隶属于所在的对象，并且与受保护的数据属性平级！

所以get和set中的this，指当前get/set函数直接隶属于的对象！



f. 示例: 使用访问器属性保护对象的年龄属性

9\_get\_set.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 eric={        ename:"埃里克",        eage:25      }      //eric的eage属性值可以改，但必须介于18~65之间      Object.defineProperties(eric,{        //1. 先添加一个半隐藏的\_eage属性，实际保存原eage属性的值        \_eage:{          //将原eric对象的eage属性值搬家到\_eage中隐藏起来          value:eric.eage,          //因为eage是可以修改:          writable:true,          //因为\_eage应该半隐藏:          enumerable:false,          //双保险:          configurable:false        },        //2. 将原eage属性替换为保镖:        eage:{//冒名顶替原属性名，迷惑外界          //保镖一请就是一对儿:          get:function(){//保镖            console.log(`自动调用了一次eage内的get(),返回${this.\_eage}`);            //专门负责从受保护的\_eage中取出现在的属性值            return this.\_eage;          },          set:function(value){//保镖//value将来会自动接住要修改的新值            console.log(`自动调用一次eage中的set(),形参value=${value}`);            //专门负责接收新值，验证后，保存到受保护的数据属性中            if(value>=18&&value<=65){              console.log(`验证通过，保存到\_eage中`)              this.\_eage=value;            }else{              console.log(`验证不通过，报错！`)              throw Error("年龄超范围！必须介于18~65之间！")            }          },          //访问器属性应该代替受保护的属性抛头露面          enumerable:true,          //访问器属性不能随意被删除，双保险          configurable:false          //访问器属性已经没有value和writable:          //因为访问器属性自己不实际保存属性值，所以没有value属性          //因为writable不好用！太单一！所以我们才被迫使用访问器属性！所以是访问器属性代替了开关writable的作用。所以访问器属性中也不再需要writable开关了！        }      })      //外界不知道有\_eage，get和set，只知道有一个eage属性保存年龄      //外界试图获取年龄时: 会发生什么事儿？自动调用get      console.log(eric.eage);      //外界试图修改年龄时: 会发生什么事儿？自动调用set      eric.eage=26;      console.log(eric);      //外界试图修改年龄为非法的-2,会发生什么事儿？      eric.eage=-2;    </script>  </body>  </html>  运行结果: |

(6). 问题: 无论用开关，还是访问器属性，都是在保护单个属性，无法防住对对象结构的破坏！比如：防不住别人恶意为对象添加新属性

2. 保护结构: 防止别人的程序擅自篡改对象的属性结构！3个级别:

对应小程序视频: 小程序->在线->JSCORE->day06 1. 保护对象结构:

(1). 防扩展: 禁止给对象添加新属性:

Object.preventExtensions(对象)

阻止 扩展

问题: preventExtensions()只防添加，不防删除，如果一个对象所有属性都禁止删除，就必须给每个属性手工添加configurable:false——太繁琐了

(2). 密封: 既禁止扩展，又禁止删除属性

a. Object.seal(对象)

密封

b. 原理:

1). 既自动调用preventExtensions()

2). 又自动设置所有属性的configurable:false!

c. 所以: 一般使用了seal()，就不用preventExtensions()了

d. 结论: 今后，绝大多数对象，保护到seal级别就够了！

e. 示例: 使用防扩展和密封保护对象结构

10\_preventExtensions\_seal.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>      "use strict";      var eric={        eid:1001,        ename:"埃里克"      }      Object.defineProperty(eric,"eid",{        writable:false      });      //防扩展:      //Object.preventExtensions(eric);      //        阻止    扩展      //密封:      Object.seal(eric);      //尝试修改eric的eid      //eric.eid=-2;      //试图为eric添加新属性      //eric.Eid=-2;//报错:      //Cannot add property Eid, object is not extensible      // 不能  添加  属性    Eid(因为)对象是 不   可以扩展的      //尝试删除ename      //delete eric.ename;//报错:      //Cannot delete property 'ename'      console.log(eric);    </script>  </body>  </html>  运行结果： |

(3). 冻结: 既禁止扩展，又禁止删除，甚至禁止修改所有属性值！

a. Object.freeze(对象);

b. 原理: 3件事:

1). 既自动调用preventExtensions()

2). 又自动设置所有属性的configurable:false

3). 又自动修改所有属性的writable:false

c. 何时: 绝大多数情况下，属性值还是应该可以修改的！所以不用冻结！但是，如果一个对象被多个模块同时使用，最好禁止修改属性值。否则，任由其中一个模块擅自修改共用对象的属性值，一定会牵一发而动全身！

d. 示例: 冻结一个对象:

11\_freeze.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>      "use strict";      var pool={        ip:"127.0.0.1",        port:3306,        db:"xz"      };      //希望pool中的所有属性值禁止修改      Object.freeze(pool);      //试图修改pool的ip属性值      pool.ip="192.168.0.100";//报错: Cannot assign to read only property 'ip'    </script>  </body>  </html>  运行结果：  Uncaught TypeError: Cannot assign to read only property 'ip' of object '#<Object>'  at 11\_freeze.html:20 |

总结:

4. 面向对象: 封装， 继承， 多态

(1). 封装: 3种:

a. 用{}只创建一个对象:

var 对象名={

属性名:属性值,

... : ... ,

方法名: function(){

... this.属性名 ...

}

}

b. 用new Object():

1). 2步:

i. var 对象名=new Object()

ii. 对象名.属性名=属性值;

对象名.方法名=function(){ ... }

2). 对象底层也是关联数组:

i. 都是名值对儿的集合

ii. 都可用[""]和.方式访问成员。但是如果属性名来自于变量就只能用[]

iii. 访问不存在的属性，不报错，返回undefined

判断是否包含某个属性: 对象.属性名!==undefined

iv. 强行给不存在的属性赋值，不报错，而是自动添加该属性

给对象添加新属性，唯一办法，强行赋值！

v. for in遍历

c. 只要反复创建多个相同结构的对象都用构造函数:

1). 2步:

i. 定义构造函数:

function 类型名(形参1,形参2, ...){

this.属性名1=形参1;

this.属性名2=形参2;

//构造函数中不要再包含方法定义定义!

}

ii. 用new 调用构造函数:

var 对象名=new 类型名(属性值1, 属性值2,...)

2). new做了4件事:

i. 创建一个新的空对象

ii. 设置新对象的\_ \_proto\_ \_继承构造函数的原型对象(prototype)

iii. 调用构造函数，传参，替换this为新对象。强行赋值为新对象添加属性

iv. 返回新对象的地址，保存到=左边的变量中。

3). 优点: 重用对象结构代码

4). 缺点: 如果构造函数中包含方法定义，则每次创建新对象都会重复创建相同方法的副本。——浪费内存！

(2). 继承:

a. 今后，只要同一类型所有子对象共用的方法和属性值，都要集中保存在构造函数的原型对象中！

构造函数.prototype.属性名/共有方法名=属性值/function(){ ... }

b. 自有属性和共有属性:

1). 获取属性值:都可用"子对象.属性名"

2). 修改属性值:

i. 自有属性: 子对象.自有属性名=新值

ii. 共有属性: 构造函数.prototype.共有属性名=新值

c. 内置类型原型对象:

1). 11种内置类型: String, Number, Boolean, Array, Date, RegExp, Math, Error, Function, Object, global

2). 类型: 构造函数+原型对象

i. 构造函数: 创建子对象

ii. 原型对象: 为所有子对象保存共有成员

3). 查看该类型共有哪些API: 类型名.prototype

4). 该类型缺少想用的方法: 类型名.prototype.共有新方法=function(){ ... }

d. 原型链: 保存着一个对象可用的所有属性和方法。控制着属性和方法的使用顺序：先自有再共有——就近原则！

(3). 多态: 重点讲重写：如果子对象觉得从父对象继承来的成员不好用，可以在子对象自己内部重写和父对象同名的成员，覆盖父对象的成员，优先使用自己的。

(4). 自定义继承:

a. 只修改一个对象的父对象:

1). 子对象.\_ \_proto\_ \_=父对象

2). Object.setPrototypeOf(子对象, 父对象)

b. 修改所有子对象的父对象

1). 构造函数.prototype=父对象

2). 必须在创建子对象之前替换

|  |
| --- |
| 面向对象总结:  1. 封装: 先创建对象: 2种:  (1). 如果只创建一个对象: {}  (2). 如果反复创建多个相同结构的对象: 构造函数  2. 继承: 所有子对象共用的方法和属性值，必须集中定义在原型对象中  3. 多态: 如果从父对象继承来的成员不好用！可以重写自己的！  4. 如果觉得整个爹都不好，可以换爹，也可以换构造函数的老公 |

5. 严格模式: "use strict";

(1). 禁止给未声明过的变量赋值

(2). 静默失败升级为错误

(3). 普通函数调用中的this不指window，而是指undefined

(4). 禁用arguments.callee

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)

|  |
| --- |
| 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(){ ... })() 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:

小程序->在线->JSCORE->day05 2. 自定义继承...

3. ES5 严格模式...

4. 保护对象属性

5. 访问器属性

小程序->在线->JSCORE->day06 1. 保护对象结构:

作业:

1. 复习今日问题清单: 小程序->首页->JSCORE->day05、day06

2. 看小程序视频预习: 小程序->首页->JSCORE-> day06

2. 替换this, call, apply, bind...

3. every原理

4. forEach

6. map原理

7. filter

8. reduce 原理 简单

完成视频中练习。