正课:

1. ES5

2. ES6

1. ES5

1. 保护对象:

为什么: JS中的对象，毫无自保能力，随时可以添加新属性，可以删除现有属性，随时可以修改属性值为任意值。

何时: 我希望限制修改对象的结构或属性值时

如何: 2个层面

1. 保护单个属性:

ES5属性进行了新的划分：2大类

命名属性: 可用.随时访问到的属性

由细分为两大类:

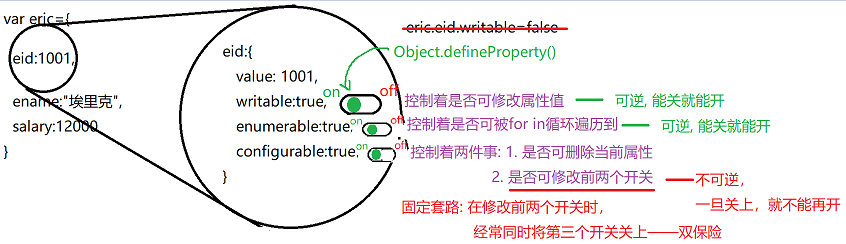
数据属性: 直接存储属性值的

访问器属性: 不直接存储属性值，仅提供对其它数据属性的保护——保镖

内部属性: 不能用.访问到的属性

如何保护数据属性:

ES5规定，对象中的每个属性其实又是一个缩微的小对象。



每个属性内部都隐藏着三个开关，分别保护着是否可修改属性值，是否可遍历属性值，是否可删除属性。

问题: 内部隐藏的开关属性，不能通过属性.开关方式操作。

解决:

Object.defineProperty(对象,"属性",{

开关: true/false,

... : ...,

})

问题: 开关，咱们可以关上，别人也可以打开！

解决: 只要修改前两个开关，都要伴随将configurable:false。而且configurable:false是不可逆的！一旦改为false，不能再改回true。

问题: Object.defineProperty()一次只能改一个属性。

解决: Object.defineProperties(对象,{

属性:{

开关: true/false,

... : ...

},

... : {

... : ...

}

})

问题: 其实enumerable:false，只是半隐藏。仅防得住for in遍历。防不住.直接访问。

解决: js语言依然存在缺陷。暂时不考虑解决。

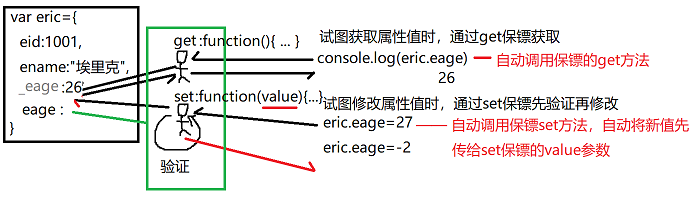
问题: 使用开关保护属性，比较单一。无法用自定义的规则随心所欲的保护属性。

解决: 如果需要更灵活更复杂的逻辑来保护属性时，就要为属性请专门的保镖——访问器属性

如何使用访问器属性保护数据属性：

什么是访问器属性: 不实际存储属性值，仅提供对另一个数据属性的保护

何时: 只要使用自定义规则保护属性时



如何: 2步:

1. 请保镖：2步:

1. 先将要保护的实际存储数据的属性隐姓埋名:

比如:

//用defineProperty() 添加实际存储数据的数据属性

Object.defineProperty(eric,"\_eage",{

value: 26,

writable:true,

enumerable:false,

configurable:false

})

2. 添加访问器属性，访问器属性使用原数据属性的名称，冒名顶替:

Object.defineProperty(eric,"eage",{

get:function(){

return this.\_eage;

},

set:function(value){

//value会自用接住这次要赋的新值

if(value>=18&&value<=65){

this.\_eage=value;

}else{

throw Error("年龄超范围！")

}

},

configurable:false,

enumerable:true

})

问题: 访问器属性内是否还需要些writable和value?

答: 不需要：

value肯定不需要，因为访问器属性自己根本不存值！其实使用get方法代替了value属性

writable也不需要。因为set是writable的升级版！就是让改！且必须符合要求！所以set也是代替了writable的。

2. 通过保镖访问数据属性：

对外界来说，访问器属性的用法和数据属性毫无差别！只不过在执行时:

试图获取属性值时，自动调用保镖的get方法，试图修改属性值时，自动调用保镖的set方法，并将新值先传给set方法的value参数验证。验证通过，才能赋值。验证不通过，则报错！

2. 保护整个对象结构: 3个级别

1. 防扩展: 禁止给对象添加新属性！

Object.preventExtensions(对象)

阻止 扩展

结果: 禁止强行给当前对象添加新属性

问题: 只能防扩展，不防删除

2. 密封: 在兼具防扩展同时，又进一步禁止删除属性。但是，属性值还是可以修改的！

Object.seal(对象)

强调:

1. 如果使用了seal，就不用使用preventExtensions()了。因为seal中涵盖了防扩展的特点。

2. 其实,seal就是自动将所有属性的configurable都自动改为了false。从此，只用用seal，所有属性上的configurable:false，都可省略！默认都是关着的！false

一般的对象保护到密封级别就够了

3. 冻结: 即不能添加新属性，又不能删除现有属性。甚至连属性值都不能修改！

Object.freeze(obj)

原理: 不但禁止添加新属性

且自动修改所有configurable：false

且自动修改所有writable:false

何时: 很多模块共同使用的对象，不能随意被一方篡改！

总结:

1. 单个属性:

//只读: writable:false

//禁止删除: configurable:false ——可省略

因为将来整个对象会启用密封，会自动禁止删除

//禁止遍历: enumerable:false 只能禁止for in，无法阻止.直接访问。

//自定义规则保护属性: 访问器属性

\_eage:{

value: 值,

writable: true,

enumerable:false,

configurable:false

}

eage:{

get:function(){ return this.\_eage; },

set:function(value){

if(value满足条件)

就this.\_eage=value

else

报错

},

enumerable:true,

configurable:false

}

2. 保护结构:

只要密封就够了: Object.seal(obj)

结果: 即禁止添加新属性，又禁止删除现有属性

2. Object.create():

什么是: 创建一个子对象，继承一个父对象

何时: 如果没有构造函数，也想创建子对象时

如何: Object.create()可以做3件事:

1. 前两件事: 创建一个子对象，并让新子对象继承指定的父对象:

var child=Object.create(father);

2. 第三件事: 可以给孩子添加自有属性，但是必须以Object.defineProperties类似的语法添加属性。

var child=Object.create(father,{

属性名:{

value: 属性值,

writable: true/false,

... : ...,

},

... :{

}

});

总结: Object.create()共做了3件事:

1. 创建一个空的子对象

2. 继承父对象

3. 添加自有属性

3. call() apply() bind()

相同: 都能替换函数执行时，函数内不想要的this！

何时: 只要函数执行时，函数内的this不是想要的，就可以用这三个函数换！

如何: 2种:

1. 在函数执行时，临时替换一次this为想要的对象

要调用的函数.call(替换this的对象, 实参值列表)

比如:

function calc(base,bonus1,bonus2){

console.log(`${this.ename}的总工资是${base+bonus1+bonus2}`);

}

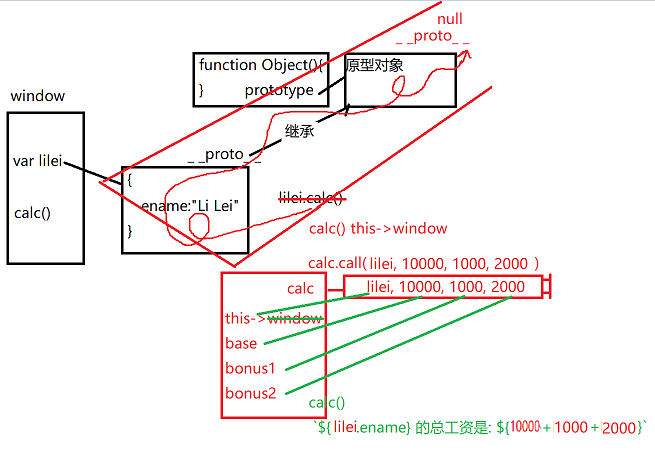
var lilei={ ename:"Li Lei"};

calc.call(lilei,10000,1000,2000);

效果: 在本次调用函数时

先替换this为call中第一个参数指定的对象。这里calc中的this被临时替换为了对象lilei。

之后的实参列表一对一赋值给原函数的形参列表。比如：这里10000给base, 1000给bonus1,2000给bonus2



如果原函数要求的参数是多个参数单独传入，而给定的实参值列表却是一个数组时，就要用apply()替换call()

要调用的函数.apply(

替换this的对象, 包含实参值的数组)

执行时的效果: 比如:

var arr=[4000,5000,6000];

calc.apply(hmm,arr);

//多一步:打散数组参数为单个值，再传入

// 4000, 5000, 6000

// ↓ ↓ ↓

// calc(base, bonues1, bonues2)

2. 基于原函数，创建一个新函数副本，但永久绑定新函数中的this为指定的对象：

vs call/apply：

call/apply只执行一次！且只临时绑定一次

.bind(): 创建一个副本，且永久绑定，即使日后反复调用函数副本，this也不会丢失或错乱！

何时: 如果要替换this的函数，不是立刻执行，也不是只执行一次！——bind()专门用于替换回调函数中的this

因为回调函数往往不是立刻执行，且不止执行一次。

如何: 2步:

1. 先基于原函数创建一个一模一样的新函数副本，但永久绑定this为指定的新对象

var fun2=原函数.bind(替换this的对象)

说明: 暂时不要传递实参值，因为不是立刻调用，只是创建函数副本而已。

比如: var calc\_l=calc.bind(lilei);

结果:

//calc\_l:function(base,bonus1,bonus2){

// this=lilei;

// console.log(`${this.ename}的总工资是${base+bonus1+bonus2}`);

//}

2. 反复调用副本函数，而不再担心this错误！——此时再传入必要的实参值列表。

fun2(实参值列表)

fun2中的this早就被绑定为了想要的对象。

比如: calc\_l(10000, 1000,2000);

虽然传入lilei，但是this->lilei

其实: bind不但可以永久绑定this，而且还可永久绑定部分实参值！

var fun2=原函数.bind(替换this的对象, 实参值,...)

比如: var calc\_l=calc.bind(lilei,10000);

效果： 不但this被永久替换为了想要的对象，且第一个参数也被提前永久绑定成了想要的实参值。

//calc\_l:function(base,bonus1,bonus2){

// this=lilei;

// base=10000;

// console.log(`${this.ename}的总工资是${base+bonus1+bonus2}`);

//}

今后，调用这个函数副本时，只要传剩余的参数即可。

比如: calc\_l(1000,2000);

其中this->lilei, base->10000

4. 数组函数:

ES5中为数组新增了很多好用的函数

1. 查找一个元素在数组中的位置:

var i=arr.indexOf(元素值, fromi)

数组中indexOf函数的用法和string中的indexOf完全一样！

原理: 在数组arr中，从开始位置fromi向后，找下一个和“元素值”相同的元素出现的下标位置！

返回值: 如果找到，就返回元素所在的位置下标i

如果找不到了，就返回-1

何时: 只要查找一个元素是否包含在数组中或想获得元素在数组中的位置时，都可用indexOf。

2. 判断: 2种:

1. 判断数组中是否所有元素都符合要求:

var bool=arr.every(function(elem, i, arr){

return 判断条件表达式

})

原理:

every会自动遍历arr中每个元素，每遍历一个元素，就自动调用一次回调函数。在调用回调函数时，会自动传入三个值: 1. 当前元素值->elem, 2. 当前所在位置i->i, 3. 当前数组对象->arr。回调函数内部根据传入的三个值，判断当前一个元素是否符合要求，将判断结果返回到回调函数外部。every会收集回调函数返回的判断结果。只有当所有的元素，经过回调函数的判断后，都返回true，整个every才返回true！只要有一个元素经过回调函数的判断，返回false，整个every都自动退出，并返回false！

2. 判断数组中是否包含符合要求的元素:

var bool=arr.some()

单词列表:

1. enumerable: 枚举/遍历 在程序中专指for in循环

2. configurable: 可配置的

3. redefine: re 重复 define 定义 redefine 重新定义

4. invoke: 调用

5. call: 调用 calc.call() 调用calc函数——执行

6. prevent: 阻止

7. extension 扩展

8. -ible 可以xxxx 同 -able

9. seal 密封

10. freeze 冻结

11. apply 应用/使用 calc.apply() 使用calc函数——执行