准备: 微信小程序: WEB问题速查 小恐龙图标

1. 历届师兄师姐在今天课程中曾经问过的高频的经典的问题清单和解答: 小程序->首页->JSCORE->day05

自己可能没有发现问题，但是想知道别的同学都问过哪些问题

2. 今日课程对应的重点案例的视频以及鄙视面试所需的扩展知识视频：小程序->在线->JSCORE->day05

如果因为某些原因错过直播讲解，不用着急！直接看小程序中视频的回看即可。

如果tmooc不能用了，立刻看小程序视频，小程序视频比tmooc要全，甚至有扩展的新知识。

如果学有余力，将来有能力拿更高薪资，小程序中所有视频必须都要看。但是不必现在都看完。只要在笔试面试前都看完即可！

问题: 个人版小程序有功能限制：禁止在线播放视频

解决: 长按视频连接地址，完整复制视频连接地址和验证码到浏览器打开。通过百度网盘下载后观看。

3. 今日课程中包含的英语单词的翻译：小程序->单词->JSCORE->day05

英文不太好的同学，可以提前看一下第二天需要用到哪些英文单词。

约定: 2件事:

1. 第三阶段禁止上课跟着敲笔记和敲代码！谁抄笔记抄的勤，谁抄代码抄的勤，谁学的越烂！

2. 每个完整功能的案例，至少写三遍:

(1). 连代码+注释抄一遍 —— 明白这个例子是要做什么事儿

(2). 保留注释，自己试着看着注释，将代码填回来 —— 复习函数和对象

(3). 删除所有代码和注释，自己试着用自己的话，把注释写回来，再翻译为代码！—— 只有自己可以写出人话的注释，这个例子才算自己会！

如果没有问题的同学: （享受vip一对一答疑的时间还剩17天）

1. 上课时那句话没听懂，笔记中那句话看不懂，案例中那句话看不懂，立刻问！

2. 小程序问题清单中哪个问题或解答看不懂，立刻问！

3. 小程序视频中哪个视频看不懂，立刻问

4. 个人项目中哪个功能不会做，立刻问

5. 平时刷笔试题时，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. 继承

2. 多态

3. 自定义继承

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

1. 严格模式

\*\*\*\*面向对象\*\*\*\*

一. 继承

对应小程序视频: 小程序->在线->JSCORE->day04 7.8.9

1. 自有属性和共有属性

(1). 什么是:

a. 自有属性: 保存在当前子对象内，只归当前子对象自己独有的属性

b. 共有属性: 保存在原型对象中，归该类型下所有子对象共有的属性

(2). 何时：

a. 如果希望每个子对象都不相同的属性值，应该定义为自有属性

b. 如果希望多个子对象共用相同的属性值时，应该定义为共有属性

(3). 如何:

a. 所有构造函数中的属性或通过.强行添加给对象的属性都是自有属性

b. 构造函数.prototype.共有属性=属性值 —— 共有属性

(4). 坑:

a. 想获取属性值则自有属性和共有属性都可用“对象.属性名”

b. 修改属性值:

1). 修改自有属性，必须用子对象.属性名=新值

2). 修改共有属性，必须用原型对象.属性名=新值

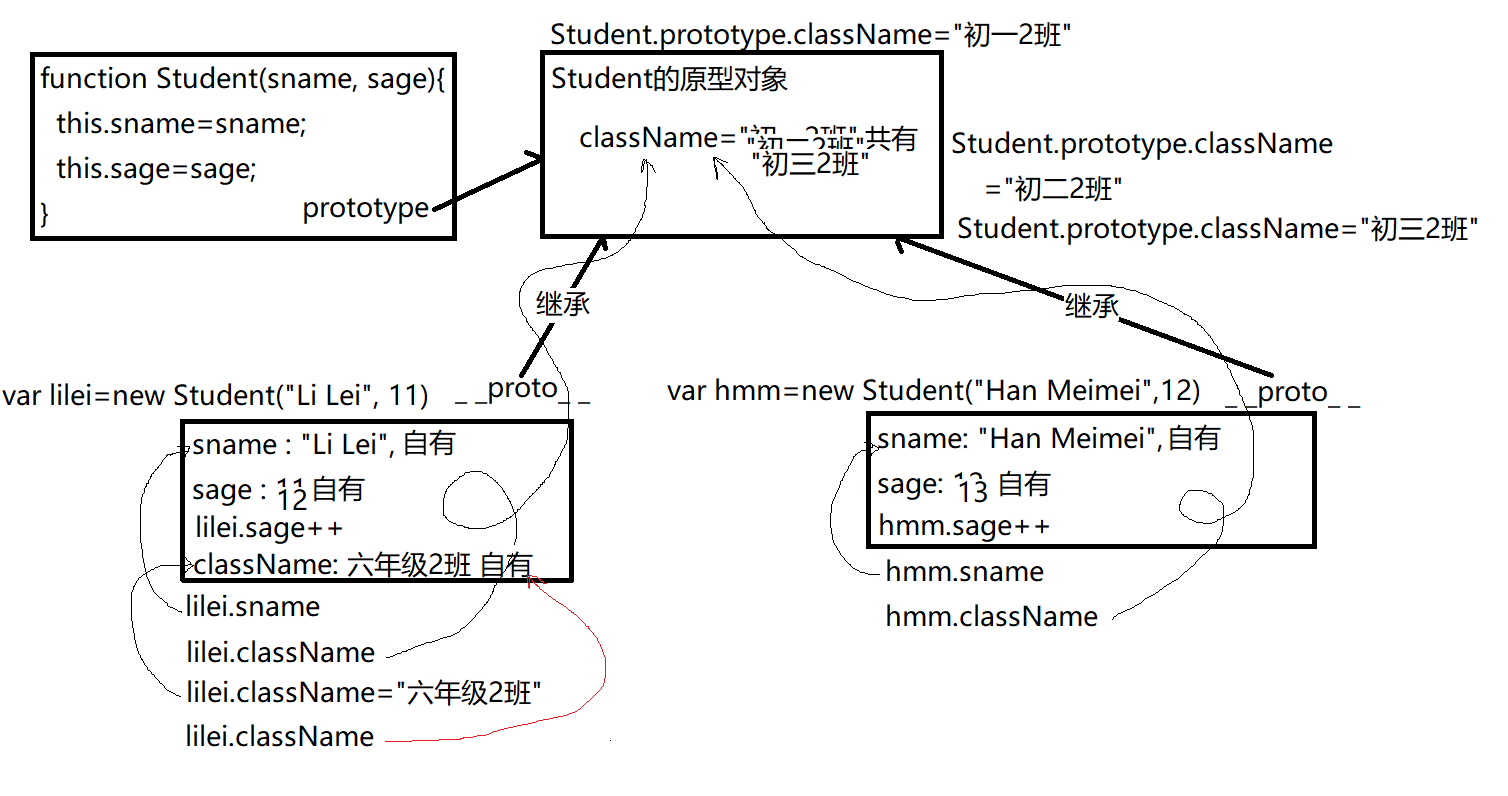
i. 不能用子对象修改共有属性 —— 不要尝试这样做！no zuo no die!

ii. 如果用子对象强行修改共有属性: 不会报错

①js绝不允许子对象直接修改共有属性，所以共有属性保持不变

②js会自动给当前子对象添加一个和要修改的共有属性同名的自有属性

结果: 当前子对象在这个共有属性的使用上和其它子对象分道扬镳，再无统一之日。



(5). 示例: 测试获取和修改自有属性和共有属性

|  |
| --- |
| <script>  //定义一个构造函数描述所有学生的统一属性结构  function Student(sname,sage){  this.sname=sname;  this.sage=sage;  }  //希望将来的所有学生都是同一个班级的学生  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);  //想获得lilei和hmm的sage //自有  console.log(lilei.sage, hmm.sage);  //想获取lilei和hmm的className //共有  console.log(lilei.className, hmm.className);  //修改lilei和hmm的年龄+1 //自有  lilei.sage++;  hmm.sage++;  //再次获得lilei和hmm的sage //自有  console.log(lilei.sage, hmm.sage);  //lilei和hmm都升了一级  //正确:  Student.prototype.className="初二2班";  //再次获取lilei和hmm的className //共有  console.log(lilei.className, hmm.className);  //错误: 李磊想拖着全班一起留级！  lilei.className="六年级2班";  //再次获取lilei和hmm的className //共有  console.log(lilei.className, hmm.className);  // 六年级2班 初二2班  console.log(lilei);  console.log(hmm);  //又过了一年，大家一起升到初三2班  Student.prototype.className="初三2班";  //再次获取lilei和hmm的className //共有  console.log(lilei.className, hmm.className);  // 六年级2班 初三2班  </script>  运行效果: |

2. 内置类型的原型对象

(1). 什么是内置类型: ES标准中规定的浏览器自带的类型

(2). 包括: 11种:

String Number Boolean

Array Date Math RegExp

Error

Function Object

global(在浏览器中被window代替)

(3). 什么是类型: 由一个构造函数和一个原型对象组成的一个整体，称为类型，用于描述一类对象的统一结构和方法定义。

(4). 每种类型: 都包含两部分:

a. 构造函数:

1). 描述该类型下将来所有子对象的统一结构

2). 反复创建该类型的子对象

b. 原型对象: 替该类型下所有子对象保存共有的方法

(5). 比如: Array就是一个中类型，包括2部分:

a. 构造函数: function Array(){ ... ... }

用于反复创建数组类型的子对象: var arr=new Array(1,2,3) 可简写为 var arr=[1,2,3]

b. 原型对象: Array.prototype保存了所有数组家孩子共用的方法:

.sort() .push() .reverse() .slice() .indexOf() ... ...

将来随着ES标准不断更新，Array.prototype一定会继续加入很多更好用的共有方法。

(6). 问题: 如果我们经常对数组做一种操作，但是数组的原型对象中没有提供现成的共用方法，怎么办？——可以自定义一个共有方法，强行添加到数组原型对象中。

a. 强行添加自定义共有方法:

Array.prototype.自定义共有方法=function(形参列表){

... 想获得将来调用这个方法的.前的某个具体的数组，应该用: this

}

b. 调用自定义共有方法: 和调用内置共有方法完全一样！

数组.自定义共有方法(实参值列表)

(7). 示例: 为数组家添加数组元素求和的sum函数，并调用

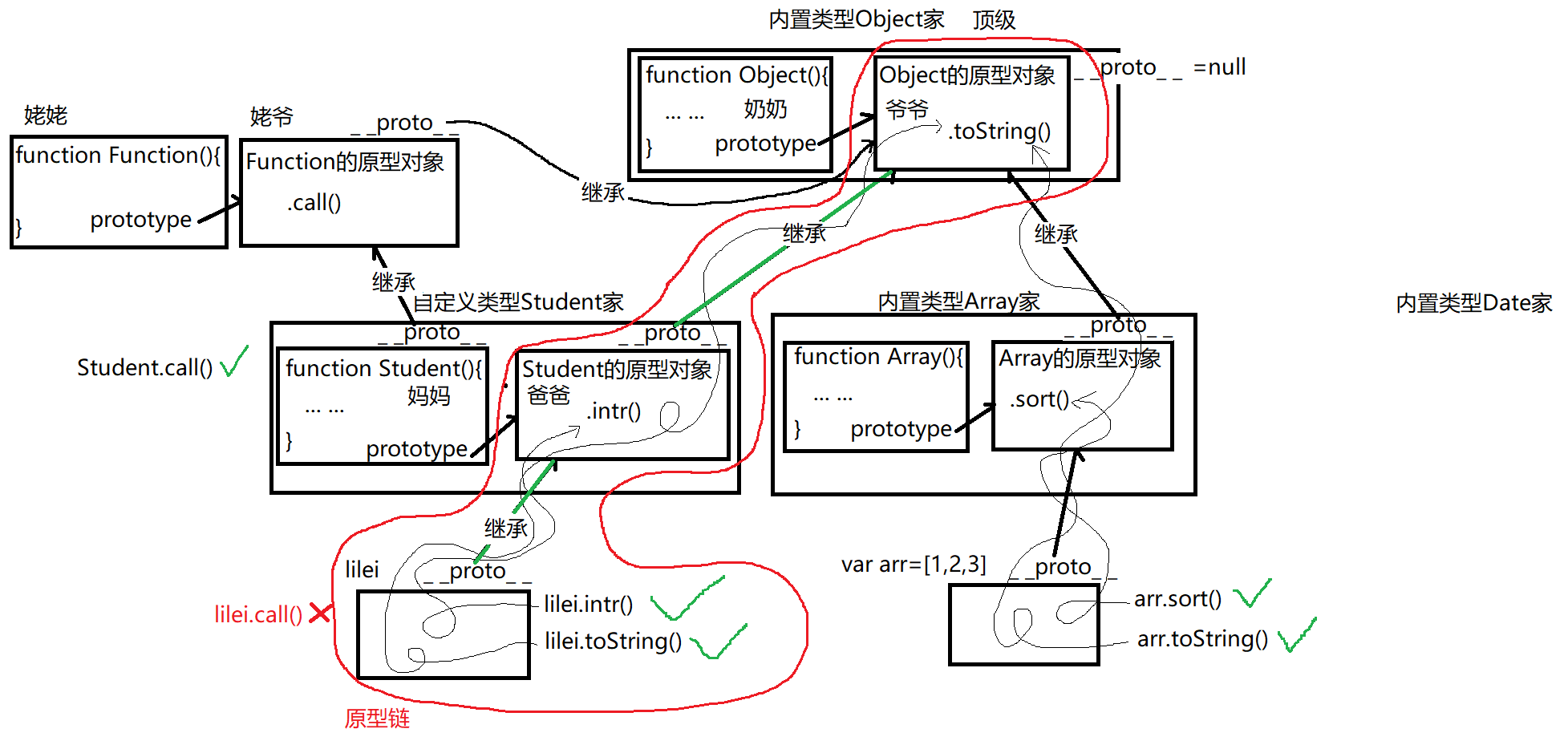
|  |
| --- |
| <script>  //如果我的项目中经常需要对数组中的元素值求和  //但是假设数组的原型对象中并没有提供求和的函数sum  //我们就可以想数组原型对象中强行添加一个自定义的sum函数  Array.prototype.sum=function(){  console.log(`调用了一次Array.prototype中我们自定义的sum()函数`)  var total=0;  //遍历将来要求和的数组中每个元素  //this->将来调用.sum()函数的.前的某一个数组家孩子对象  //比如: arr1.sum() this->arr1  //比如: arr2.sum() this->arr2  for(var i=0;i<this.length;i++){  total+=this[i];  }  return total;  }  console.log(Array.prototype);  //测试:  //两个数组家孩子  var arr1=[1,2,3]  var arr2=[1,2,3,4,5];  //可共用数组原型对象中我们自定义的sum()函数，谁调用，this就指谁  console.log(arr1.sum());//6  console.log(arr2.sum());//15  </script>  运行效果:  [sum: ƒ, constructor: ƒ, concat: ƒ, copyWithin: ƒ, fill: ƒ, …]  调用了一次Array.prototype中我们自定义的sum()函数  6  调用了一次Array.prototype中我们自定义的sum()函数  15 |

3. 原型链:

(1). 什么是: 由多级父对象逐级继承形成的链式结构

(2). 保存着: 子对象可以使用的所有共有的属性和方法

(3). 控制着: 属性和方法的使用顺序: 先自有，再共有 —— 就近



二. 多态：

对应小程序视频: 小程序->在线->JSCORE->day05 1. 多态 override toString()

1. 什么是: 一个函数在不同的情况下表现出不同的状态

2. 包括: 2种:

(1). 重载(overload): 一个函数根据传入的实参值不同，动态选择执行不同的逻辑

(2). 重写(override): ?

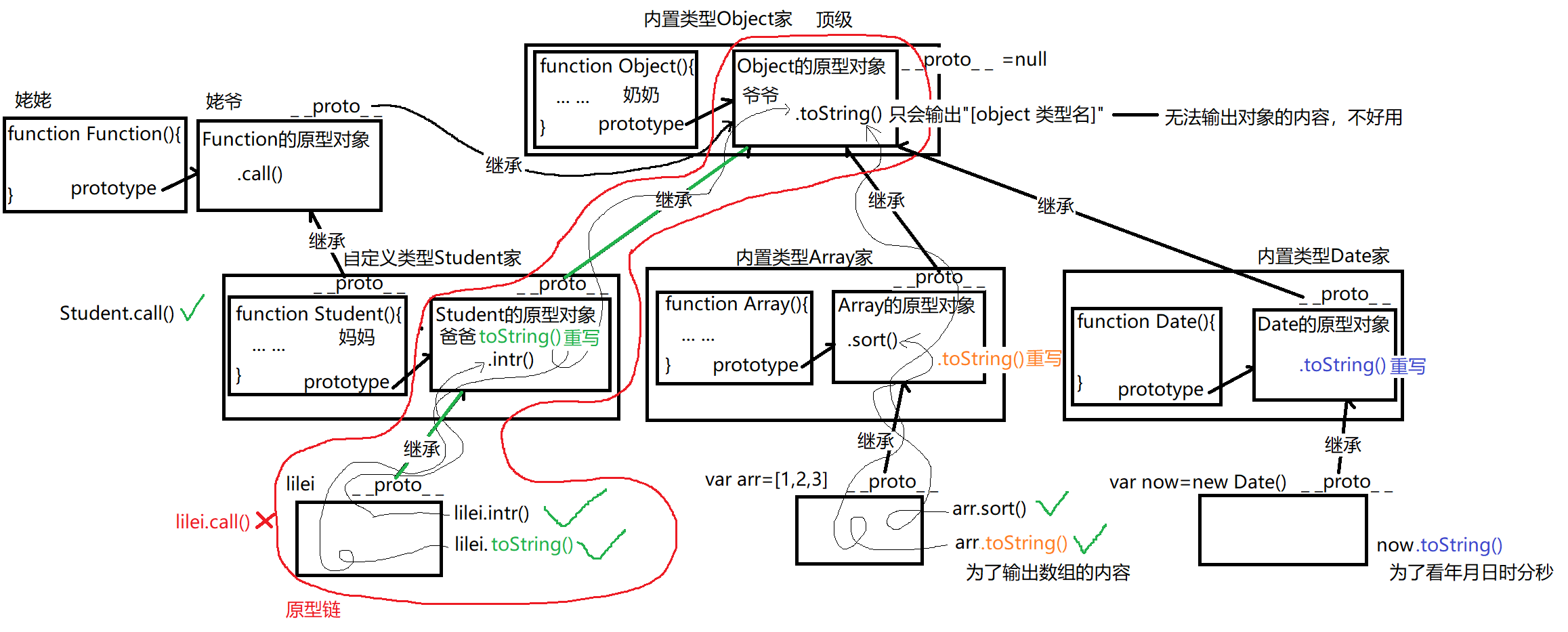
推翻

3. 重写:

(1). 什么是: 在子对象中定义一个和父对象中的成员同名的自有成员

(2). 为什么: 从父对象继承来的东西不一定总是好用的！

(3). 何时: 只要子对象觉得从父对象继承来的东西不好用！都可以在子对象内定义同名成员，重写父对象中的同名成员。

(4). 如何: 在子对象内定义和父对象中的成员同名的自有成员  


(5). 示例: 为自定义类型或自定义对象重写好用的toString()方法

|  |
| --- |
| <script>  // var lilei={//new Object()  // sname: "Li Lei",  // sage:11,  // //只给李磊重写toString()方法  // toString:function(){  // return `{ sname: ${this.sname}, sage: ${this.sage} }`  // }  // }  function Student(sname, sage){  this.sname=sname;  this.sage=sage;  }  //如果希望将来所有学生都有好用的toString(),则toString应该放在Student类型的原型对象中  Student.prototype.toString=function(){  return `{ sname: ${this.sname}, sage: ${this.sage} }`;  //格式可以自己随便改！  }  var lilei=new Student("Li Lei", 11);  var hmm=new Student("Han Meimei",12);  console.log(lilei.toString());  console.log(hmm.toString());  var arr=[1,2,3]; //new Array()  var now=new Date(); //new Date()  //对三个对象分别调用toString()——本意为了输出对象的内容  console.log(lilei.toString());  console.log(arr.toString());  console.log(now.toString())  </script>  运行效果：  { sname: Li Lei, sage: 11 }  { sname: Han Meimei, sage: 12 }  { sname: Li Lei, sage: 11 }  1,2,3  Tue Jun 02 2020 14:55:20 GMT+0800 (中国标准时间) |

三. 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 Object.setPrototypeOf()

1. 何时: 如果整个父对象都不想要！都想换掉！就可以自定义继承

2. 如何: 2种:

(1). 只更换一个子对象的父对象:

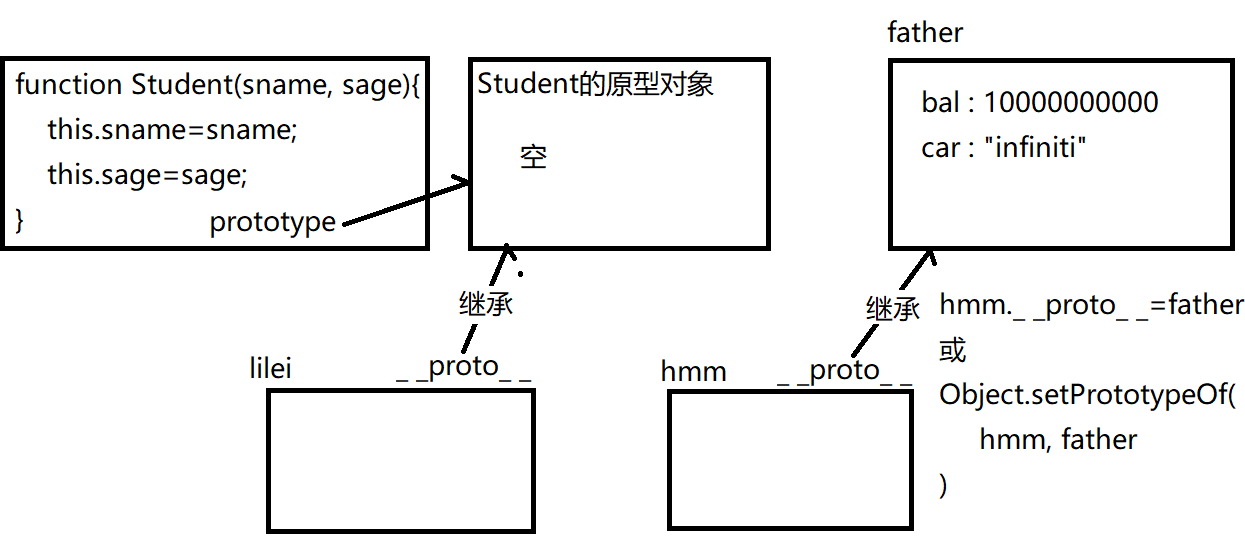
a. 子对象.\_ \_proto\_ \_=新父对象

b. 问题: 有些浏览器可能不允许直接使用\_ \_proto\_ \_

c. Object.setPrototypeOf(子对象, 新父对象)

设置原型对象的子对象 为新父对象

d. 示例: 只更换一个对象的爹:



|  |
| --- |
| <script>  function Student(sname, sage){  this.sname=sname;  this.sage=sage;  }//prototype=>{ } //旧爹 空的  var lilei=new Student("Li Lei",18);  var hmm=new Student("Han Meimei",19);  //输出李雷和韩梅梅的爹  console.log(lilei.\_\_proto\_\_);  console.log(hmm.\_\_proto\_\_);  //新爹 诱惑  var father={  bal:100000000000000,  car:"infiniti"  }  //hmm想认father当爹  //hmm.\_\_proto\_\_=father;//有时有的浏览器不允许直接使用\_\_proto\_\_  Object.setPrototypeOf(hmm, father);  //输出李雷和韩梅梅的爹  console.log(lilei.\_\_proto\_\_);  console.log(hmm.\_\_proto\_\_);  //输出李雷和韩梅梅的银行存款和car  console.log(hmm.bal, hmm.car);  console.log(lilei.bal, lilei.car);  </script>  运行结果:  {constructor: ƒ}  {constructor: ƒ}  {constructor: ƒ}  {bal: 100000000000000, car: "infiniti"}  100000000000000 "infiniti"  undefined undefined |

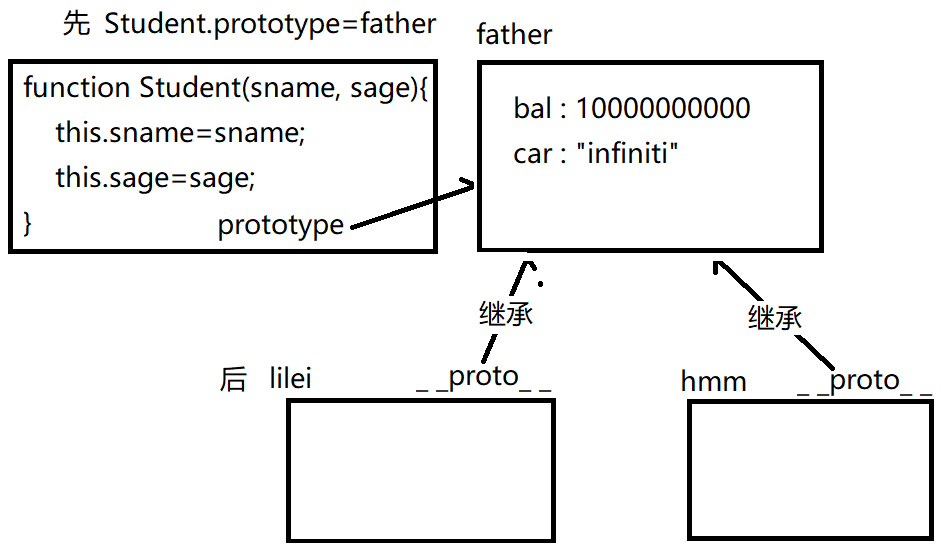
(2).更换多个子对象的爹:

a. 修改构造函数的原型对象: 构造函数.prototype=新对象

b. 问题: 如果在已经创建子对象之后，再换构造函数的原型对象，已经创建的子对象的原型对象是不会自动跟着改变的——无法达到更换所有子对象的原型对象的效果

c. 时机: 如果想修改多个子对象的原型对象，必须在创建子对象之前就要更换！

d. 示例: 同时更换lilei和hmm的爹



|  |
| --- |
| <script>  function Student(sname, sage){  this.sname=sname;  this.sage=sage;  }//prototype=>{ } //旧爹 空的  //输出构造函数旧的原型对象  console.log(Student.prototype);  //新爹 诱惑  var father={  bal:100000000000000,  car:"infiniti"  }  //修改构造函数的原型对象为father  Student.prototype=father;  //输出构造函数新的原型对象  console.log(Student.prototype);  //创建子对象  var lilei=new Student("Li Lei",18);  var hmm=new Student("Han Meimei",19);  //输出李雷和韩梅梅的爹  console.log(lilei.\_\_proto\_\_);  console.log(hmm.\_\_proto\_\_);  //输出李雷和韩梅梅的银行存款和car  console.log(hmm.bal, hmm.car);  console.log(lilei.bal, lilei.car);  </script>  运行结果:  {constructor: ƒ}  {bal: 100000000000000, car: "infiniti"}  {bal: 100000000000000, car: "infiniti"}  {bal: 100000000000000, car: "infiniti"}  100000000000000 "infiniti"  100000000000000 "infiniti" |

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

ECMAScript 第五版 对旧js打补丁 —— 各个击破！没有联系

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 use strict...

1. 什么是: 比旧的js运行机制要求更严格的模式

2. 为什么: 因为旧的js有很多广受诟病的缺陷

3. 何时: 今后所有js程序都要运行在严格模式下！

4. 如何启用严格模式: 在当前<script>标签内顶部或js文件的顶部添加: "use strict"

启用 严格模式

5. 新规定:

(1). 禁止给未声明的变量赋值:

a. 旧js中：强行给未声明过的变量赋值，不报错，自动在全局创建该变量

b. 缺点: 极易造成全局污染！

c. 严格模式：强行给未声明过的变量赋值，报错！xxx变量未定义！

d. 好处: 极大的减少了全局污染的可能！也避免了程序调试中的歧义。

e. 示例: 使用严格模式禁止给未声明过的变量赋值

|  |
| --- |
| <script>  "use strict";//启用严格模式  //想定义一个函数可以给女朋友发送一条消息  function send(){  var gf; //女朋友  //我想告诉她: 今晚308,w84u  //但是不小心变量名拼错了  //js没有变量检查机制，即使写错变量名，变量不存在，也会在全局创建该变量  pyq="今晚308,w84u";//严格模式: 报错！阻止了一次误操作的发生  console.log(`我女朋友收到:${gf}`);  }  send();  //在全局输出变量pyq的内容  console.log(`我的朋友圈:${pyq}`);  </script>  运行结果:  Uncaught ReferenceError: pyq is not defined  at send (6\_use\_strict.html:18)  at 6\_use\_strict.html:21 |

(2). 静默失败升级为错误：

a. 什么是静默失败: 执行不成功也不报错

b. 缺点: 极其不便于程序的调试

c. 严格模式: 所有静默失败都被升级为错误！

d. 优点: 极其便于程序的调试，避免歧义

e. 示例: 使用严格模式让静默失败升级为错误

|  |
| --- |
| <script>  "use strict";  //一会儿要讲的新知识，先不要问为什么  //一个员工eric  var eric={  eid:1001,  ename:"埃里克"  }  //公司规定员工编号不能随意修改——只读  // 定义 属性 eric的eid属性为不可修改  Object.defineProperty(eric,"eid",{writable:false});  //试图修改eric的eid属性值  eric.eid=1002;//严格模式: 报错:  //Cannot assign to read only property 'eid' of object  //不能 赋值 给 只读 属性 eid  //不能给只读属性eid赋值——23行  console.log(eric);//没改成  </script>  运行结果:  Uncaught TypeError: Cannot assign to read only property 'eid' of object '#<Object>'  at 7\_use\_strict.html:23 |

(3). 普通函数和匿名函数自调中的this不再指向window，而是undefined

a. 旧js中: fun() 或 (function(){ ... })() this->window

b. 缺点: 极易造成全局污染

c. 严格模式: fun() 或 (function(){ ... })() this=undefined

d. 优点: 杜绝了因为错误的使用this而造成的全局污染！

e. 示例: 使用严格模式杜绝因为错误使用构造函数而导致的全局污染

|  |
| --- |
| <script>  "use strict";  function Student(sname,sage){  this.sname=sname;  this.sage=sage;  //没有return，所以没有返回值，相当于undefined  }  //本来构造函数应该用new调用  var lilei=new Student("Li Lei",11);  console.log(lilei);  //但是不小心忘记写new了  var hmm=Student("Han Meimei",12);  //undefined.sname=sname; //报错:  //Cannot set property 'sname' of undefined  //不能 设置 属性 sname 的 undefined  //不能设置undefined的sname属性  //undefined.sage=sage; //报错  console.log(hmm);//undefined  //console.log(sname, sage);//在全局直接输出本该保存到对象内的属性，却可以直接使用！——全局污染！  </script>  运行结果:  Student {sname: "Li Lei", sage: 11}  Uncaught TypeError: Cannot set property 'sname' of undefined  at Student (8\_use\_strict.html:13)  at 8\_use\_strict.html:21 |

(4). 不推荐使用arguments.callee了

a. 什么是arguments.callee: 在函数内部自动获得当前函数本身 的关键词

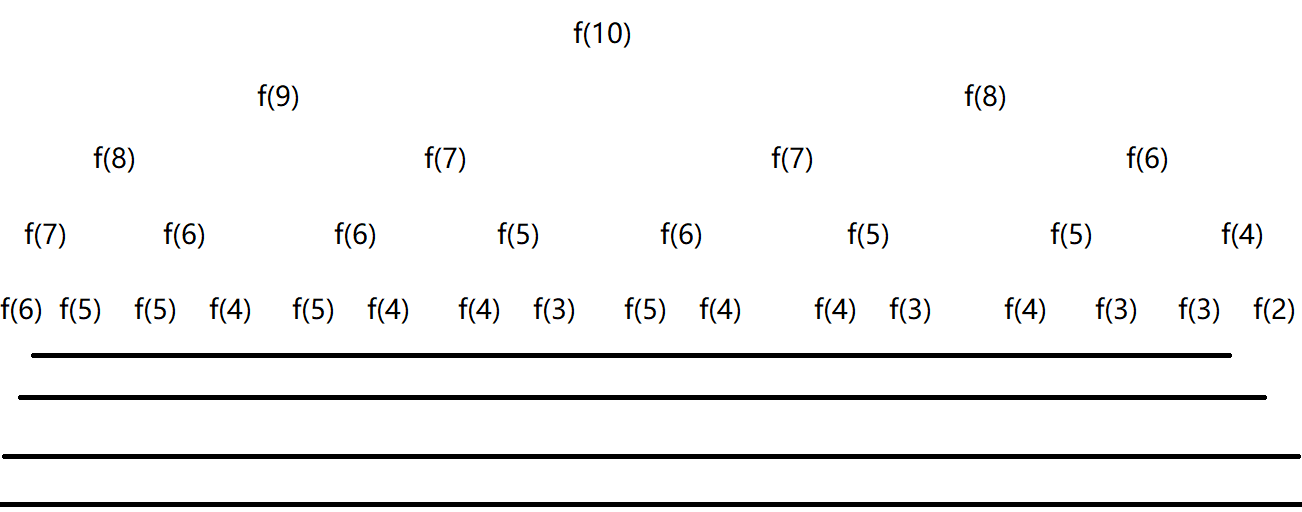
b. 何时: 专门用于递归调用时

c. 问题: 如果在递归中写死函数名，则一旦外部函数名发生变化，程序极容易出错——紧耦合

d. 解决: 在递归调用的函数内部用arguments.callee代替写死的函数名

e. 优点: arguments.callee在运行时，可自动获得当前正在执行的函数对象本身——即使外部函数改名了，也能照样获得当前函数本身——松耦合！

f. 问题: 递归算法本身效率极低——重复计算量太大！



g. 严格模式: 禁用arguments.callee就暗示着，不推荐使用递归算法

h. 解决: 绝大多数递归算法都可用循环方式代替——难度极高！

（扩展）小程序->在线->JSCORE->day05 3.ES5严格模式 ... 视频中有循环实现的菲波那契数列算法的详细讲解。——忘了菲波那契数列的同学，晚上可以看我讲的视频。

i. 总结: 不要走极端！

1). 如果采用递归算法并没有严重影响效率，依然可以采用递归算法，只不过只能在递归函数中写死函数名——退而求其次

2). 除非采用的递归算法严重影响了效率，才有必要费劲将递归转为循环。

j. 示例: 使用arguments.callee和递归实现菲波那契数列

|  |
| --- |
| <script>  "use strict";  //小程序->在线->JSCORE->day05 3.ES5严格模式 ...  //复习第一阶段讲过的斐波那契数列  //1 1 2 3 5 8 13 21 34 55 ...  //1 2 3 4 5 6 7 8 9 10  //总结:  //第一项是1,  //第二项还是1,  //从第三项开始，当前项的值等于相邻的前两项值的和  //定义一个函数求出斐波那契数列中第n项时的值是多少  function f(n){  if(n<3){  return 1;  }else{  // 递归 递归  return arguments.callee(n-1)+arguments.callee(n-2);//报错  //自动获得当前函数名 f f  }  }    console.log(f(10));//55  </script>  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 (9\_argument.callee.html:26)  at 9\_argument.callee.html:31 |

总结

1. Function

(1). 创建函数: 3种:

a. 用声明方式: 会被声明提前

function 函数名(形参变量列表){

函数体;

return 返回值

}

b. 用赋值方式: 不会被声明提前

var函数名=function (形参变量列表){

函数体;

return 返回值

}

c. 用new: 几乎不用

var 函数名=new Function("形参变量1","形参变量2",...,"函数体")

揭示了js中函数的本质:

function 其实是new Function()的简写

函数其实也是一个对象

函数名只是一个普通的变量而已

(2). 重载:

a. 何时: 一件事可能根据传入实参值的不同选择执行不同的逻辑时就要用重载

b. 如何:

1). 只定义一个函数，不要定义任何形参变量

2). 调用函数时，在函数内arguments对象会自动接住所有传入函数的实参值列表（类数组对象:有下标，有length，可遍历，但是不是数组家孩子）

3). 在函数内通过判断arguments的length属性不同或arguments中实参值不同来选择不同的逻辑执行

4). 如果想获得本次传入的实参值: arguments[下标]

(3). 匿名函数:

a. 何时: 如果一个函数只用一次时，不会重复使用，就要用匿名函数

b. 优点: 节约内存

c. 如何: 2种情况:

1). 回调函数: 主函数(function(形参变量){ ... })

2). 匿名函数自调: (function(){ ... })()

何时: 今后所有js代码都应该放在一个匿名函数自调中——避免使用全局变量

(4). 作用域和作用域链:

a. 作用域: js中包含两种作用域:

1). 全局作用域: window 始终在内存中，保存所有全局变量。全局变量特点：

i. 优点: 可重用

ii. 缺点: 随处可用——极易被污染

2). 函数作用域: 只在函数调用时才临时创建，函数调用后，函数作用域就释放。保存着函数内所有局部变量(2种: 形参和函数内var)。局部变量特点:

i. 优点: 仅函数内可用，不会被污染

ii. 缺点: 不可重用

b. 作用域链: 其实每个函数对象在创建时，都有一个属于自己的作用域链:

1). 保存了一个函数可用的所有作用域和变量

2). 控制了变量的使用顺序: 先函数内后全局——就近

(5). 闭包:

a. 何时: 想重用一个变量，但是又想保护变量不被篡改——其实就是给一个函数保护一个专属的变量，仅函数自己可反复用，别人无法使用。

b. 如何: 3步:

1). 用外层函数包裹内层函数和要保护的变量

2). 外层函数将内层函数return到外层函数外部

3). 外部的使用者需要调用外层函数，获得返回的内层函数对象，并将内层函数对象保存在变量中，反复使用。

c. 闭包形成的原因: 外层函数调用后，外层函数的作用域对象被内层函数的作用域链引用着，无法释放，成为内层函数的私有财产，就形成了闭包。

2. 面向对象: 三大特点: 封装 继承 多态

(1). 封装: 3种:

a. 用{}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()：用的不多，2步 揭示: js对象底层都是关联数组

1). var 对象名=new Object()

2). 对象名.新属性=新值;

对象名.方法名=function(){ ... this.属性名 ...}

c. 用构造函数反复创建多个相同结构的对象: 2步

1). function 类型名(形参变量列表){

this.属性名=形参;

... = ... ;

~~this.方法名=function(){ ... this.属性名 ... }~~

}

2). var 对象名=new 类型名(实参值列表)

new做了4件事:

a. 创建一个新的空对象

b. 让新的子对象自动继承构造函数的原型对象——认爹/继承

c. 调用构造函数，替换this为新对象，通过强行赋值方式，将构造函数中规定的属性强行添加到新对象中

d. 将新对象地址保存到=左边的变量中

(2). 继承:

a. 何时：只要同一类型多个子对象都要使用的共有方法，应该集中定义在构造函数的原型对象中:

b. 如何: 构造函数.prototype.共有方法=function(){ ... ... }

原型对象 强行赋值

c. 结果: 凡是被这个构造函数new出来的所有子对象，都自动继承当前构造函数的原型对象。所有子对象无需重复创建方法，就可直接使用原型对象中的方法。

(3). 多态: （重写）

a. 何时: 如果子对象觉得从父对象继承来的东西不好用！就可以重写！

b. 如何: 只要在子对象中定义和父对象中成员同名的自有属性。

c. 结果: 子对象再调用这个成员时，只使用自有的成员，不会再舍近求远使用父对象中不好用的成员了。

(4). 自定义继承: 2种:

a. 只更换一个子对象的爹: 2种:

1). 子对象.\_ \_proto\_ \_=新爹

2). Object.setPrototypeOf(子对象, 新爹)

b. 同时更换多个子对象的爹: 构造函数.prototype=新对象 (在创建子对象之前换)

总结: this 4种

判断this指谁，一定不要看this定义在哪儿！一定只看调用时.前是谁或有没有new.

1. obj.fun() this->.前的obj

2. new Fun() this->new本次正在创建的一个新对象

3. 原型对象.prototype.共有方法=function(){ ... } this->将来调用这个共有方法的.前的某个子对象

4. fun()或(function(){ ... })() this->默认指window ——记住，规定，没有为什么

对应小程序视频:

小程序->在线->JSCORE->day04 7.8.9

小程序->在线->JSCORE->day05

1. 多态 override toString()

2. 自定义继承 Object.setPrototypeOf()

3. ES5 严格模式 use strict arguments.callee 斐波那契 循环

作业:

1. 复习今日问题清单(小程序->首页->JSCORE->day04、day05)，预习明日问题清单(小程序->首页->JSCORE->day06)

2. 看小程序视频预习保护对象属性:

小程序->在线->JSCORE->day05 4. 保护对象属性 开关 writable ... ...

3. 看小程序视频预习访问器属性:

小程序->在线->JSCORE->day05 5. 访问器属性 get set ...

4. （学有余力!!!）看小程序视频学习闭包和面向对象鄙视题: 画图，判断输出结果

小程序->在线->JSCORE->day04 所有高频鄙视题开头的视频