# JS面向对象

## 1 小知识点

### 1.1 null与undefined

null： 不再需要使用某个对象，即需要解除对象的引用，才会使用null；

只有手动给变量赋值为null时，变量的值才是null，类型为object。

undefined：已经定义了对象，却没有赋值，类型为undefined。

### 1.2 关键字in

in关键字主要使用在 for in 循环中，可以用来判断：属性是否存在对象中。

|  |
| --- |
| **var** obj = {  name : **"123"**,  age : 30 }; **for**(**var** i **in** obj){  console.log(i); *//输出属性名 name age* } console.log((**"name" in** obj));  *//结果为true。注意：字符串类型的键，这里不加 "" 包裹，结果为 false* |

in在操作数组时，判断索引是否存在，而不是判断值。

|  |
| --- |
| **var** arr = [1,2,3]; console.log(0 **in** arr); *//true* console.log(**"0" in** arr); *//true* |

判断值是否存在：使用for或者 arr.indexOf()---会返回索引，没找到返回-1。

|  |
| --- |
| **var** arr = [1,2,3]; console.log(arr.indexOf(3)); *//输出2* console.log(arr.indexOf(5)); *//输出-1* |

### 1.3 关键字delete

delete 可以用来删除对象中的成员。

|  |
| --- |
| **var** obj = {  name : **"lisi"**,  age : 30 }; **delete** obj.name; *//obj.name 为undefined* |

delete 还可以删除未使用过的已声明变量。

|  |
| --- |
| num = 100; **delete** num; *//如果以var开头，无法删除* console.log(num); *//报错！！且会说明错误原因是 num undefined。*  *//删除未声明*  **result** = delete num1; **console**.log(**window**.num1,**result**); *//输出 undefined true* |

### 1.6 变量作用域

变量起作用的范围就是变量的作用域。在JS中，只有函数能产生作用域。

1. 块级作用域：JS中没有块级作用域

演示：

for(int i = 0;i < 10;i++){

int j = i;

}

print(i);

print(j);

如果有块级作用域，显示结果均以为undefined。

1. 词法作用域

在代码写好的那一刻，变量的作用域已经确定了。

var a =123;

function f1(){

console.log(a);

}

function f2(){

var a = 456;

f1();

}

f2();

输出123

词法作用域规则：

函数允许访问函数外的数据；

整个代码结构中只有函数才能产生作用域；

作用域首先使用提升规则分析；

如果当前作用域有该变量，就不考虑外面的变量。

1. 动态作用域

代码写好的时候，作用域并不是就在写好变量时就能确定的，JS中没有。

在动态作用域下会输出：456。即执行f1()时，f1内部没有a，会去其所在作用域中找a。

JS代码执行的步骤：预解析再执行

在预解析阶段，JS会对以var声明的变量名、以function开头的语句块，进行提升操作。

alert(a);

var a = 3;

提升为：

var a;

alert(a);

a = 3;

弹出undefined，如果没有var a = 3，直接平白无故alert(a) 会报错。

同名函数提升规则：提升最后一个，因为在提升时，后者会将前者覆盖。

fn();

function fn(){

console.log("first");

}

function fn(){

console.log("last");

}

//输出last

//变量和函数同名

//在提升的时候，如果有变量和函数同名，会忽略掉变量，只提升函数

alert(foo); //undefined 函数体

function foo(){}

var foo = 2;

alert(foo); //2

//预解析 提升后的代码

function foo(){};

alert(foo);

foo=2;

alert(foo);

变量的提升也是看作用域的：

var num = 456;

function test(){

console.log(num);

var num = 10;

}

test();

//输出undefined

var num = 456;

function f1() {

console.log(num);

num = 10;

}

f1();

console.log(num);

//打印 456 10

//函数表达式不会被提升

func();

var func = function(){

alert("你猜我会不会被调用");

}

//提升后的代码

var func;

func();

func = function(){

alert("你猜我会不会被调用");

};

//条件式函数声明是否会被提升，取决浏览器

//条件式函数声明不推荐去写

foo(); //这里会报错，因为未被提升

if(true){

function foo(){

console.log("123");

}

}

// foo();

### 1.7 作用域链

//只要是函数就可以创造作用域

//函数中又可以再创建函数

//函数内部的作用域可以访问函数外部的作用域

//如果有多个函数嵌套，那么就会构成一个链式访问结构，这就是作用域链

//f1--->全局

function f1(){

//f2--->f1--->全局

function f2(){

//f3---->f2--->f1--->全局

function f3(){

}

//f4--->f2--->f1---->全局

function f4(){

}

}

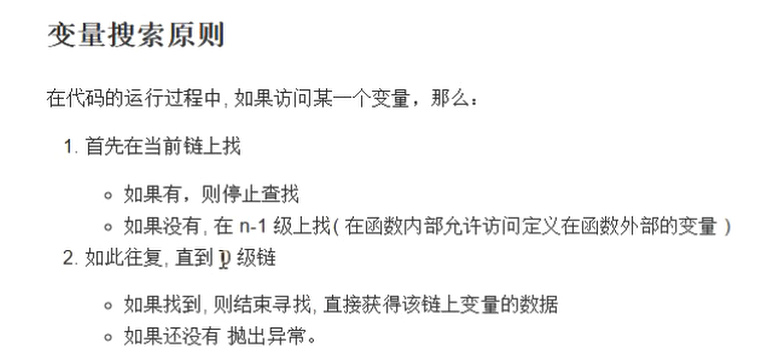
//f5--->f1---->全局

function f5(){

}

}

### 1.8 变量搜索原则



### 1.9 闭包

对外不公开的空间，在JS中，函数其实就是闭包。闭包的原理就是作用域访问原则：上级作用域无法直接访问下级作用域中的变量。

闭包要解决的问题：让外界能够间接访问函数内部的成员。

function foo(){

var num = 123;

function inner(a){

num = a;

console.log(num);

}

return inner;

}

var fn = foo();

fn(456);

/\*

function inner(a){} 其实写法是：

var inner = function(a){}

\*/

//上述写法简化：

function foo(){

var num = 123;

return function(a){

num = a;

console.log(num);

};

}

var fn = foo();

fn(789);

但是我们往往需要在不传参，fn()时候是为了获取函数内部的数据值，优化写法：

function foo(){

var num = 123;

return function(a){

if(a){

num = a;

console.log(num);

}else{

console.log(num);

}

};

}

var fn = foo();

fn(); //输出123

fn(789); //输出789

fn(); //输出789

获取多个数据：

// function foo() {

// var name = "张国荣";

// var age = 18;

//

// return [

// function(){

// return name;

// },

// function(){

// return age;

// }

// ]

// }

//

// var getName = foo();

// console.log(getName[0]());

// console.log(getName[1]());

function foo() {

var name = "张国荣";

var age = 18;

return {

getName:function () {

return name;

},

getAge:function () {

return age;

}

}

}

var obj = foo();

console.log(obj.getName());

console.log(obj.getAge());

使用对象返回：

function foo() {

var name = "高金彪";

var gender = "female";

return {

getName:function () {

return name;

},

setName:function(value){

name = value;

return name;

},

setGender:function(value){

gender = value;

// return gender;

},

getGender:function(){

return gender;

}

};

}

var obj = foo();

console.log(obj.getGender());

console.log(obj.setGender("雄"));

// console.log(obj.getGender());

闭包的作用

function foo(){

var name = "潘明";

var badLevel = -1000000000000000000000000000000000;

return {

getName: function () {

return name;

},

setName: function (value) {

name = value;

return name;

},

getBadLevel:function(){

return badLevel;

},

setBadLevel:function (value) {

//在函数外部想要修改数据

//只能通过函数内部的方法

//我们可以在函数内部定义的这个方法里

//设置安全措施，校验之类的操作

//可以保证系统的安全性和稳定性

if(value > 0 ){

throw "你敢说我坏！！！";

}

badLevel = value;

return badLevel;

}

}

}

//

var obj = foo();

// obj.setName("高金彪");

obj.setBadLevel(obj.getBadLevel() \* -1 \* Math.pow(10,10000000));

console.log(obj.getBadLevel());

//闭包的作用

//最基本的作用：可以通过闭包返回的函数或者方法，来修改函数内部的数据

//创建一个私有的空间，保护数据

//外部想要访问数据，只能通过函数提供的方法

//在提供的方法中，我们可以设置一些校验逻辑，让数据变得更加安全

var name = "";

var age = 19;

//张二狗

name = "二狗";

//隔壁老王

name = "隔壁老王到此一游";

### 1.10 作用域分段

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

<script>

foo()

function foo(){

console.log("第一个script标签内的函数")

};

</script>

<script>

foo()

function foo(){

console.log("第2个script标签内的函数")

}

</script>

<!--

第一个script标签内的函数

19-作用域分段.html:16 第2个script标签内的函数 -->

</head>

<body>

</body>

</html>

## 2 对象的创建

### 2.1 方式一：字面量

|  |
| --- |
| var obj = {  name : "张学友",  sayHello : function () {  console.log("Hello World");  }  } |

缺点：不能把JSON对象当做模板来进行new操作，只能使用点语法一个一个赋值。

### 2.2 方式二：Object创建

var p = new Object();

p.age = 13;

缺点同字面量创建。

### 2.3 方式三：工厂函数

|  |
| --- |
| **function** createSong() {  **var** obj = **new** Object();  obj.name = **"一千个理由"**;  obj.singer = **"张学友"**;  obj.sing = **function** () {  console.log(**"doadoadoa"**)  }  **return** obj; } **var** obj1 = createSong(); |

### 2.4 方式四：构造函数

|  |
| --- |
| **function** Person() { *//首字母大写用来强调自己是构造函数* **this**.name = **"尼古拉斯凯奇"**; //不添加this类似于window.name 会污染全局   **this**.age = 50;  **this**.sayHello = **function** () {  console.log(**"Hello World"**);  } } **var** p = **new** Person(); |

注意：

1-构造函数是用来初始化对象的，new才是用来创建对象的；

2-构造函数如果没有参数，new调用构造函数时，可以省略括号；

3-构造函数如果没有return，会把构造函数内的this对象返回给对象。

4-new的执行过程是：

a创建一个空对象

b把this指向这个空对象

c把空对象的内部原型 指向 构造函数的原型

d构造函数如果没有return，把空对象返回

如果return了基本数据类型，则仍然返回这个空对象

如果return了复杂数据类型，则替换掉空对象

代码演示---非可运行实际代码：

|  |
| --- |
| // function Cat(){  //  // this = {}; //1  // this.\_\_proto\_\_ = Cat.prototype; //2  //  // this.age = 18; //3  // this.show = function(){console.log(123)};  //  // return this; //4  // } |

### 2.5 方式四演进：原型创建

如果使用多个构造函数创建对象，多个构造函数的内部的方法一样，这时候会造成资源浪费，我们可以在外部将这个通用函数书写出来，但是又在成了全局污染：

|  |
| --- |
| **function** studyMethod() {  console.log(**this**.name + **"刻苦学习"**); } **function** Student(stuName) {  **this**.name = stuName;  **this**.study = studyMethod; } **var** stu = **new** Student(**"lis"**); stu.study(); studyMethod(); *//没有调用者，会输出 undefined刻苦学习* |

解决办法：使用原型创建。

在构造函数被创建出来的时候，系统会默认帮构造函数创建并关联一个空对象，即——原型对象，获取原型的方式：

构造函数.prototype

原型中的成员都可以被相应构造函数创建的对象所使用。利用这个特点，我们可以将构造函数中的的函数，放到原型中存储，就解决了全局变量污染的问题。

|  |
| --- |
| *//构造函数：构造函数中的方法将覆盖原型中的同名方法* **function** Cat(name){  **this**.name = name;  **this**.eat = **function** () {  console.log(**"构造函数eat..."**);  } }  *//原型添加成员的两种方式* Cat.prototype.eat = **function** () {  console.log(**"原型eat...."**); } Cat.prototype[**"dance"**] = **function** () {  console.log(**"原型dance..."**); } **var** c = **new** Cat(**"Green"**); c.eat(); c.dance();  *//也可以直接替换原型对象* Cat.prototype = {  msg : **"温柔的小猫"** } console.log(c.msg); *//这时候输出是undefined* **var** c1 = **new** Cat(**"Green1"**); console.log(c1.msg); *//可以正确输出* |

注意：

1-点语法赋值时，会直接给对象设置属性，对象没有该属性，将直接添加。不会去原型中查找、修改。

2-原型中的属性是引用类型时，所有对象共有该属性，且一个对象对其进行修改，其他对象都会受到影响。

3-一般情况下，不会将属性放在原型中，原型中只存放共享方法。

### 1.4 对象添加成员

JS中的键值对组合其实就是对象。

对象中值类型存储的是对应类型的值，引用类型存储的是引用地址，其数据类型为object，会在内存中单独存储。

我们可以直接通过 点语法 或者 [] 来给对象添加成员。

|  |
| --- |
| **var** obj = {  name : **"lisi"**,  age : 30 }; obj[0] = **function** () {  console.log(**"添加成功"**); } obj[{}] = **function** () { *//类似添加了 obj["object Object"]* console.log(**"添加成功"**); } console.log(obj);  *//{ '0': [Function],name: 'lisi',age: 30,'[object Object]': [Function] }* |

### 2.6 rwr=3 方法调用顺序

调用方法、属性的顺序：

1-在对象自己内部查找

2-在原型中查找

3-以上属性没找到即undefined，方法没找到就会报错 not a function

### 2.7 如何访问原型

构造函数访问原型：构造函数.prototype

实例对象访问原型：实例对象.\_\_proto\_\_

由于部分浏览器不支持实例对象访问原型的方法，所以不推荐这样使用，仅在调试时使用。

### 2.8 原型对象属性

原型对象在创建出来的时候，这个空对象包含2个属性：

1-默认有一个 constructor 属性，指向构造函数；

2-部分浏览器有 \_\_proto\_\_ 属性，指向自己；

3-原型中还有实例对象需要的方法；

4-原型上如果有属性，只有原型自己能修改这个属性；

如果实例对象也创建了一个和原型一样的属性（属性名一样），那么不会去修改原型的值，而是给实例对象自己添加了属性。读取属性先找实例对象自己的，如果没有再去找原型。

注意：如果使用替换的方式替换原型，如果新原型没有constructor属性，会影响三角关系，为保证 构造函数--原型---对象 三者的合理，应在替换时候手动加入constructor属性。

Person.prototype = {

constructor : Person

};

所以我们创建对象还可以这样创建：

|  |
| --- |
| **function** Person(name) {  **this**.name = name; } **var** p1 = **new** Person(**"zs"**); console.log(Person.prototype.constructor); console.log(p1.constructor); *//直接用对象访问原型中的属性* **var** p2 = **new** p1.constructor(**"lisi"**); *//相当与使用了构造函数创建* console.log(p2.name); |

### 2.8 标准创建对象

|  |
| --- |
| **function** Person(option){  **this**.\_init(option); }  Person.prototype = {  \_init:**function**(option){ *//（构造函数内部方法需要加\_,不加也可以，但不推荐）* **this**.name = option.name;  **this**.age = option.age;  },  show:**function**(){  console.log(1111);  }  };  **var** p = **new** Person({  name:**'lisi'**,  age:30 }); p.show(); |

这样写的好处，如果按照2.4的方式写，构造函数传参，那么以后如果需要新增参数，就会去很多地方进行修改，这种标准书写方式就可以避免。

## 3 三大特性-继承

面向对象三大特性：封装、继承、多态

多态：大多用于强类型语言。

### 3.1 混入式继承

|  |
| --- |
| **var** singer1= {  name : **"张学友"**,  sayHello : **function** () {  console.log(**"Hello World"**);  } } **var** singer2 = {  } **for**(**var** k **in** singer1){ *//k只能获得键，singer[k]获得值* singer2[k] = singer1[k]; } console.log(singer2); |

### 3.2 原型继承

因为原型中的成员可以被相关的不同对象共享。

|  |
| --- |
| **var** father = {  sayHello : **function** () {  console.log(**"实现了继承"**);  } }  **function** Son(name,age) {  **this**.name = name;  **this**.age = age; }  *//我们想让son对象继承 father对象的sayHello函数* Son.prototype = father; **var** son = **new** Son(**"lisi"**,30); son.sayHello(); |

注意：

直接替换Son原型会有隐患。原型中的成员可能会丢失，我们也可以对原型进行混入式继承：

for(var k in father){

Person.prototype[k] = father[k];

}

知识点：我们不推荐扩展内置对象，但是可以使用下列方法更安全的扩展。

|  |
| --- |
| **function** MyArray() {} MyArray.prototype = **new** Array(); *//MyArray即成为了具备和Array一样的功能* |

### 3.3 经典继承

《JavaScript语言精粹》作者提出了继承方案：

|  |
| --- |
| **function** jicheng(obj) {  **var** o = {};  o.\_\_proto\_\_ = obj;  **return** o; } **var** o = jicheng({name : **"zs"**}); console.log(o.name); |

ECMA根据上述方式在ECMA5中制定了标准继承语法：

Object.create(obj) 返回值为一个继承了obj的对象。

|  |
| --- |
| *//制作一个实现继承的工具* **function** create(obj) {  **if**(Object.create){  **return** Object.create(obj);  }**else** { *//解决不兼容问题ECMA5* **function** F() {  }  F.prototype = obj;  **return new** F();  } } |

## 4 原型链

### 4.1 原型链介绍

每个构造函数都有原型对象；

每个对象都有构造函数；

每个构造函数的原型都是一个对象，那么这个原型对象也有构造函数，这个构造函数也有原型对象；

这样形成了一个链式结构，成为原型链；

最终的对象为Object。

### 4.2 属性搜索原则

1、先在对象自身查找

2、再去当前对象的原型中查找

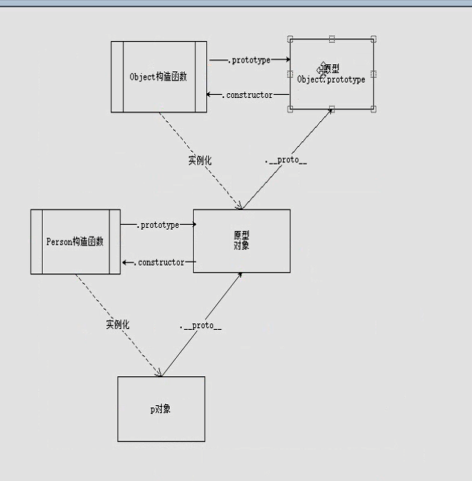
3、再找原型的原型对象...依次向上查找

4、最后查找Objct.prototype

5、找不到报错

|  |
| --- |
| *//原型链展示：动物-人-老师-数学老师* **function** Animal() {  **this**.gender = **"male"**; } Human.prototype = **new** Animal(); Human.prototype.constructor = Human; **function** Human() {  **this**.actionWay = **"run"**; }  Teacher.prototype = **new** Human(); Teacher.prototype.constructor = Teacher; **function** Teacher() {  **this**.skill = **"teach"**; }  MathTeacher.prototype = **new** Teacher(); MathTeacher.prototype.constructor = MathTeacher; **function** MathTeacher() {  **this**.name = **"lisi"**; } **var** t = **new** MathTeacher(); console.log(t); |

### 4.3 原型链关系图



## 5 函数调用模式

|  |
| --- |
| *// 1-函数执行模式* **function** fn1(a,b){  *// console.log(this); //this 即是window* **return** a + b; } console.log(fn1(2,3));  *//2-对象方法调用模式 所有的事件响应方法都是对象方法调用模式 //3-构造器调用模式 this指向构造出来的对象。 //4-call和applye调用模式* **function** fn2(a,b) {  **this**.result = a + b; } **var** s = {}; fn2.call(s,3,4); *//这个方法调用时，this指向s* console.log(s.result); *//apply 和 call 是一样的用法，只不过apply第二个参数用数组进行传递* |