■JS的解析与执行过程

|  |  |  |
| --- | --- | --- |
| JS的解析与执行过程 | 全局 | 预处理阶段 |
| 执行阶段 |
| 函数 | 预处理阶段 |
| 执行阶段 |

●全局代码的解析与执行

◎全局预处理阶段，扫描全局代码的以下内容：

1. 用声明的方式创建的函数
2. 用var定义的变量

◎讲扫描到的函数和方法保存到全局作用域的变量对象中：

1. 用声明的方式创建的函数的值是对该函数的引用
2. 用var定义的变量的值是undefined

例如：

全局代码中有如下内容：

var a = 2;

function b (){alert(1);}

那么预处理扫描后，会把以下内容加到全局变量对象中：

window{

a:undefined,

b:对b函数的引用

}

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

alert(g===undefined); //弹出true，因为g是表达式方式声明的函数，在预处理阶段只被认为是var声明的变量，其值被保存为undefined

f(); //会输出1，因为f是用声明的方式创建的函数，被预处理保存在全局变量对象window中了

g(); //会报错，"Uncaught TypeError: undefined is not a function"。因为g是用表达式的方式声明的函数，只是被认为是var声明的变量，所以值被保存为undefined

function f (){

console.log(1);

}

var g = function(){

console.log(2);

}

</script>

</body>

</html>

◎如果标识符有冲突的处理：

全局预处理会先扫描函数声明，然后再扫描var变量。

处理函数声明有冲突，会覆盖

处理变量有冲突，会忽略（因为值都是undefined）

处理函数和变量的冲突，会以函数为重

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

alert(a); //会打印出第二个函数a的代码，因为函数声明冲突会采用覆盖

alert(b); //会打印undefined，因为var变量冲突会忽略，因为值都是undefined

alert(c); //会打印函数c的代码，因为函数声明和变量冲突时，会以函数为重

function a (){

console.log(1);

}

function a (){

console.log(2);

}

var b=3;

var b=5;

function c (){

console.log(3);

}

var c=7;

</script>

</body>

</html>

◎预处理后的执行

预处理后，开始一行一行执行代码，代码的执行过程中，不断更新全局变量对象中的内容

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

// 经过预处理,全局变量对象是这样的: window{a:undefined, f:引用, g:undefined}

alert(a); //打印undefined

// alert(b); //报错，因为window中没有b

alert(f); //打印函数f的代码

alert(g); //打印undefined

var a =5;

b = 6;

// 预处理完成后,开始一行一行执行代码,执行到这里时,用执行后的结果更新了window{a:5, f:引用, b:6，g：undefined}

alert(b); //打印6

function f(){

console.log(1);

}

var g =function(){

console.log(2);

}

// 代码执行到这里后,window中又更新了g的值window{a:5, f:引用, b:6，g：引用}

alert(g); //打印函数g的引用

</script>

</body>

</html>

●函数的预处理与执行

◎函数预处理阶段

函数每调用一次，就产生一个活动对象，函数执行完毕后活动对象被销毁（闭包除外）

活动对象中包括：

1. 函数的参数（值是undefined，执行时如果传入具体值，则用具体值来更新undefined）
2. 函数内部声明式函数
3. 函数内部var变量
4. arguments

处理冲突与全局处理方式一样

◎执行函数时，如果函数中不用var定义变量，例如b=2;则执行该行代码后，b被加到全局变量对象中。（没执行时预处理不会添加）

■作用域链及闭包

●每个执行环境（execution context）都有一个与之关联的变量对象（variable object），该执行环境中定义的所有变量和函数都保存在这个变量对象中。

●执行环境有全局执行环境和局部执行环境：

◎全局执行环境被认为是window对象，因此全局变量和函数都是作为window的属性和方法创建的。全局环境直到应用程序退出时才被销毁。

◎局部执行环境中的所有代码执行完毕后，该环境被销毁，里面的所有变量和函数也随之被销毁。

●函数：

◎每个函数都有自己的执行环境。

◎当创建一个函数时，其作用域链被保存在内部属性[[Scope]]中。（Scope中保存的是创建该函数时的变量对象。比如在全局作用域下创建一个函数a，Scope保存的就是全局变量对象，即window对象。然后在a里又创建一个函数b，那么b的Scope保存的就是a的Scope加上a的活动对象）

◎当调用这个函数时，首先会创建一个当前函数的执行环境及活动对象（activation object）（我理解活动对象就是当前函数的变量对象）。然后从[[Scope]]中读取作用域链，并将新创建的活动对象加到作用域链的最前端。组成目前执行环境的完整的作用域链。

◎当这个函数执行完毕后，局部活动对象被销毁，内存中只保留原本它的[[Scope]]

例子：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

var q=1;

function a (num) {

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

}

return function b (nn){

return nn + num +i; //在闭包b中可以访问其父级a函数的活动对象,如num和i,注意,访问的是最终结果,比如for循环的i是最终循环完的值10

};

}

var n1=a(3); //a执行完毕后，其活动对象不会被销毁，因为b的作用于链仍然再引用这个活动对象。

var n2=n1(2);

alert(n2); //返回15 nn(2)+num(3)+i(10)

n1=null; //解除对b的引用，已释放内存，销毁a的活动对象。

</script>

</body>

</html>

●例子详解：

●作为包裹闭包函数b的a函数，发生了以下事情：

◎当创建a函数时，全局执行环境的全局变量对象被保存在a函数的内部属性[[Scope]]中。

◎调用a时，发生了以下事情：

1. 创建a的执行环境及活动对象（活动对象理解成新创建的执行环境的变量对象）。
2. 复制[[Scope]]中的对象，然后加上新创建的活动对象，构成当前函数的作用域链。

◎a执行完毕后，发生了以下事情：

本应该销毁a的活动对象，内存中仅保留全局作用域[[Scope]]，但是，由于a里面的b函数的作用域链仍然在引用a的这个活动对象，所以a执行完毕后它的活动对象也不会被销毁。

●作为闭包的b函数，发生了以下事情：

◎当创建b函数时，全局执行环境的全局变量对象加上a的活动对象被保存在b函数的内部属性[[Scope]]中。

◎调用b时，发生了以下事情：

1. 创建b的执行环境及活动对象。
2. 复制[[Scope]]中的对象（即，全局变量对象，加上执行a后保留下来的a的活动对象）再加上新创建的b的活动对象，组成完整的b的作用域链。

b执行完毕后，b的活动对象会被销毁。（因为b里面没有引用b的闭包函数了）。

●最后，a,b都执行完了，可是内存里还保留着a的活动对象，所以用n1=null来清除

后记：比如在b里要找一个变量q 那么首先会在b的活动对象中找，如果找不到就顺着作用域链找其父级的词法环境，也就时b的scope中保存的对象。如果b的scope中也没有，那么就报错了。

全局执行环境

全局变量对象

a的执行环境

a的活动对象

b的活动对象

b的执行环境

◎全局变量对象包括：

变量q

函数a

◎a的活动对象包括：

变量q

函数a

参数num

变量i

函数b

◎b的活动对象包括：

变量q

函数a

参数num

变量i

函数b

参数nn

●注意：[[scope]]保存的是引用而不是复制

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

function f(){

var num=1;

// 创建g时会将全局变量对象加f的活动对象保存到g的[[scope]]中,但保存的是引用,而不是复制

function g(){

alert(num);

}

num++;

g(); //执行g时，因为f的活动对象num的值发生了变化，而g得scope中是保存的对f活动对象的引用，因此也会相应变化

}

f(); //输出2

</script>

</body>

</html>

■创建对象

创建对象的三种模式：工厂模式；构造函数模式；原型模式

●工厂模式

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

function createPerson(name, age, job){

var o=new Object();

o.name=name;

o.age=age;

o.job=job;

o.sayName=function(){

alert(this.name);

};

return o;

}

var person1=createPerson("Tom", 29, "worker");

var person2=createPerson("Jerry", 22, "teacher");

console.log(person1.name);

person2.sayName();

</script>

</body>

</html>

●构造函数模式

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

function Person(name, age, job){

this.name=name;

this.age=age;

this.job=job;

this.sayName=function(){

alert(this.name);

};

}

var person1=new Person("Tom", 29, "worker");

var person2=new Person("Jerry", 22, "teacher");

Person("Alice", 25, "stundent"); //不使用new直接调用函数，那么this指的上一级对象，这里添加到了window

console.log(person1.age);

person2.sayName();

window.sayName(); //会弹出Alice

</script>

</body>

</html>

◎函数的调用方法有两种：

一种是直接调用方法，即，函数名加括号

另一种是构造函数调用方法，即，new加空格加函数名加括号

◎当在全局作用域中调用一个函数时，this对象总是指向Global对象（在浏览器中时window对象）。因此，上面例子中Person("Alice", 25, "stundent")将属性和方法添加给了window对象。之后可以通过window对象来调用其属性或方法。例如window.sayName

◎使用new操作符调用函数的方法叫构造函数方法。以这种方法调用的构造函数会经历以下4个步骤：

1. 创建一个新对象。
2. 将构造函数的作用域赋给新对象（因此this就指向了这个新对象）。
3. 执行构造函数中的代码（为这个新对象添加属性和方法）。
4. 返回新对象。

◎操作习惯：当创建一个函数时，如果希望创建构造函数，则函数名的首字母建议大写，这样以后便于区分哪些是普通函数，哪些是构造函数。

●原型模式概述

◎我们创建的每个函数都有一个prototype属性，这个属性是一个指针，指向一个对象。如果使用new操作符来把这个函数当作构造函数调用，那么prototype属性指向的对象就叫做这个构造函数创造出来的对象实例的原型对象。

◎Prototype属性指向的对象最初只有一个constructor属性，该属性也是一个指针，指向prototype属性所在的函数。

◎而被构造函数创建出来的对象实例的内在属性[[prototype]]指向其原型对象。

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

// 创建一个构造函数Person

function Person(){

}

// 定义通过Person构造函数来创建的对象实例的原型对象的属性和方法

Person.prototype.name="Tom";

Person.prototype.age=29;

Person.prototype.job="worker";

Person.prototype.sayName=function(){

alert(this.name);

};

var person1=new Person();

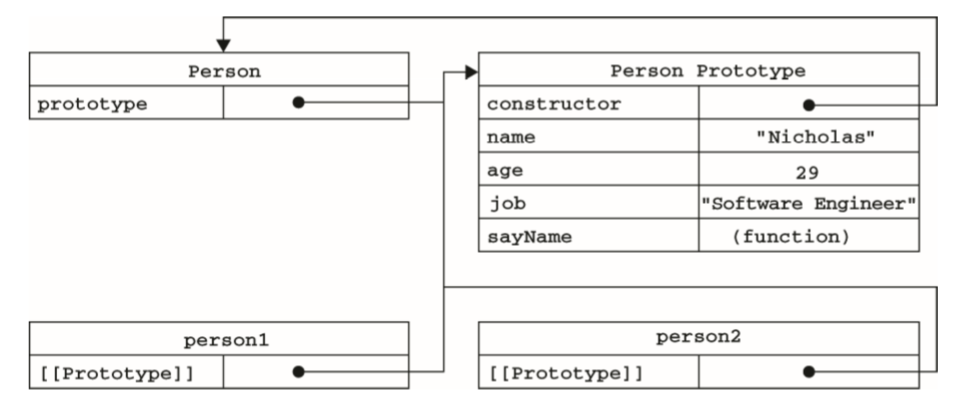
var person2=new Person();

console.log(person1.job == person2.job); //会输出true，因为person1和person2引用同一个原型对象。

</script>

</body>

</html>



总结：

原型和实例的关系：每个构造函数都有一个原型对象（构造函数的prototype属性指向这个原型对象），原型对象都包含一个指向构造函数的指针（constructor），而实例都包含一个指向原型对象的内部指针（[[prototype]]）。

◎通过简单方式来设置原型对象属性和方法，如下例

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

// 创建一个构造函数Person

function Person(){

}

var friend=new Person();

// 简单方式定义通过Person构造函数来创建的对象实例的原型对象的属性和方法

Person.prototype={

name:"Tom",

age:29,

job:"worker",

sayName:function(){

alert(this.name);

}

}

var person1=new Person();

var person2=new Person();

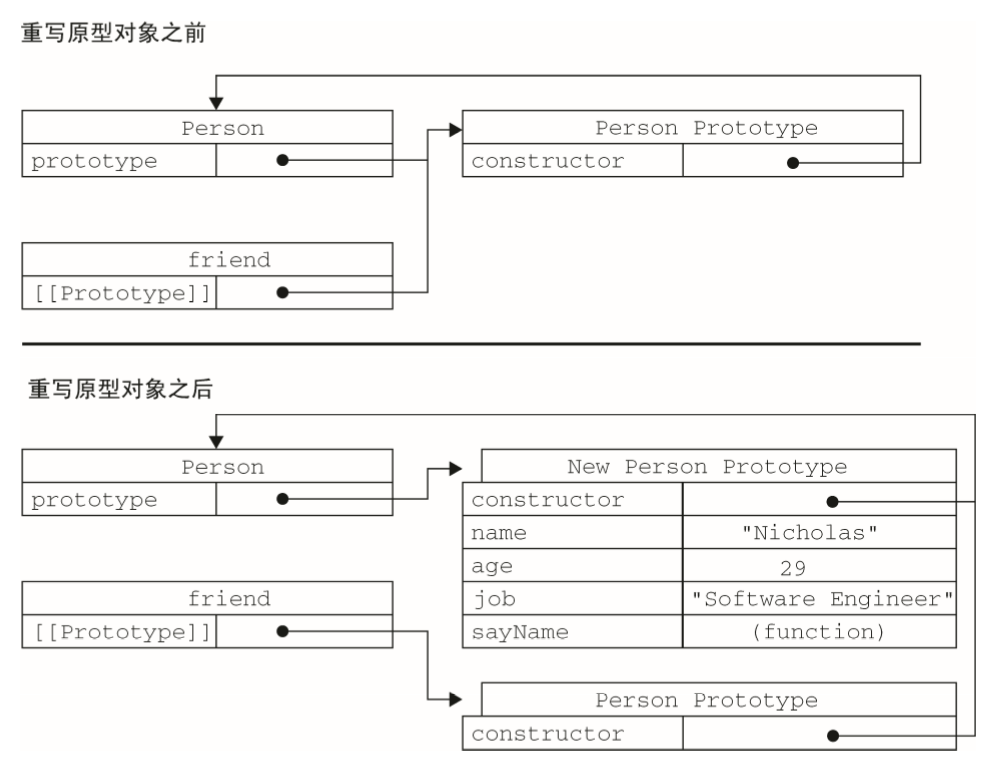
console.log(person1.job == person2.job); //会输出true，因为person1和person2引用同一个原型对象。

console.log(friend.name); //会输出undefined，因为使用简单方式重写构造函数Person的prototype后，实际是建立了一个新对象。和之前friend的原型对象不是一个对象了。

</script>

</body>

</html>



注意：这种方法重写的原型对象的constructor属性是可枚举的，而默认情况下，原生的constructor属性是不可枚举的。

◎原型对象的问题

因为基本类型的值是复制,引用类型的值是引用,所以在构造函数实例中修改基本类型的值不会影响原型,而修改引用类型的值就会影响原型。

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

// 创建一个构造函数Person

function Person(){

}

// 简单方式定义通过Person构造函数来创建的对象实例的原型对象的属性和方法

Person.prototype={

name:"Tom",

age:29,

job:"worker",

sayName:function(){

alert(this.name);

},

friends:["Jerry", "Alice"]

}

var person1=new Person();

var person2=new Person();

// 因为基本类型的值是复制,引用类型的值是引用,所以person1中修改基本类型的值不会影响原型,而修改引用类型的值就会影响原型.

person1.friends.push("Jon");

person1.age=22;

console.log(person1.friends); //输出"Jerry,Alice,Jon"

console.log(person2.friends); //输出"Jerry,Alice,Jon"

console.log(person1.age); //输出22

console.log(person2.age); //输出29

</script>

</body>

</html>

组合使用构造函数模式和原型模式来解决以上问题

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

function Person(name, age, job){

this.name=name;

this.age=age;

this.job=job;

this.friends=["Jo1", "Jo2"];

}

Person.prototype={

constructor:Person,

sayName:function(){

alert(this.name);

}

}

var person1=new Person("Tom", 29, "worker");

var person2=new Person("Jerry", 22, "teacher");

person1.friends.push("Jo3");

console.log(person1.friends); //输出"Jo1,Jo2,Jo3"

console.log(person2.friends); //输出"Jo1,Jo2"

</script>

</body>

</html>

●继承和原型链概念初步

如之前总结过的，原型和实例的关系：每个构造函数都有一个原型对象（构造函数的prototype属性指向这个原型对象），原型对象都包含一个指向构造函数的指针（constructor），而实例都包含一个指向原型对象的内部指针（[[prototype]]）。

而通过让原型对象等于另一个类型的实例，来实现原型链和继承。

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

// 创建一个构造函数SuperType

function SuperType(){

this.property=true;

}

SuperType.prototype.getSuperValue=function(){

return this.property;

};

// 创建另一个构造函数subType

function SubType(){

this.subproperty=false;

}

// 通过让SubType的原型对象等于SuperType,而使SubType继承了SuperType

SubType.prototype=new SuperType();

SubType.prototype.getSubvalue=function(){

return this.subproperty;

}

// 创建SubType的实例,这个实例可访问上面构建好的原型链

var instance=new SubType();

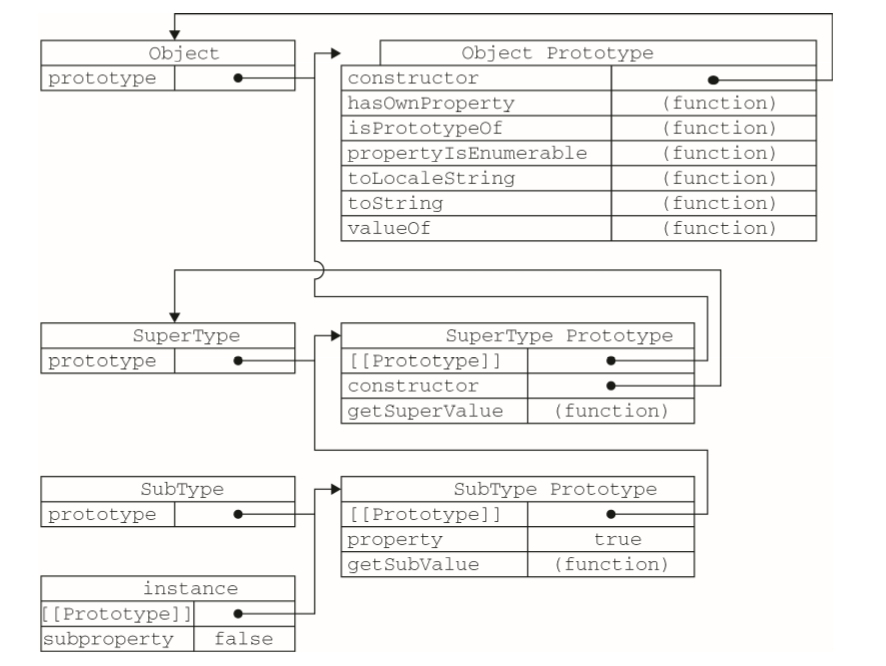
console.log(instance.getSubvalue());

console.log(instance.getSuperValue());

</script>

</body>

</html>



说明：因为说有引用类型都继承了Object，所以，所有函数的默认原型都是Object的实例。因此默认原型都会包含一个内部指针[[prototype]]，指向Object.prototype

■闭包的小例子

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

(function(){

// 在这里定义变量可以防止外部冲突，只有内部才可以访问

var a=5;

var b=6;

function f(){

alert(a+b);

}

window.f=f; //使外部可以调用函数f

})();

f(); //可正确弹出结果11

</script>

</body>

</html>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

function f1(){

var a=10;

var b=5;

// 创建一个闭包函数f2,它能访问其父级函数f1的作用域链

function f2(){

console.log(a+b);

}

f2();

}

f1();

</script>

</body>

</html>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

function f1(){

var a=10;

var b=5;

// 使f1返回一个闭包函数f2

return function f2(){

console.log(a+b);

}

}

var re=f1(); //执行f1函数，其返回一个函数f2，并将返回的函数保存在变量re中

re(); //此时re相当于f2函数

</script>

</body>

</html>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

var a=0;

function add (){

a++;

alert(a);

}

add(); //弹出1

add(); //弹出2

// 为了减少全局变量,可采用以下闭包的方法来完成上面例子的任务

function f(){

var a=0;

return function g(){

a++;

alert(a);

}

}

var re=f();

re(); //弹出1

re(); //弹出2

</script>

</body>

</html>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

function calFactory(base){

return function(max){

var total=0;

for(var i=1; i<=max; i++){

total +=i;

}

return total + base;

}

}

var adder=calFactory(2);

alert(adder(3)); //输出8 （base2 + max3 1+2+3）

</script>

</body>

</html>

封装的例子如下：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

<script type="text/javascript">

// 以下实现用闭包进行封装

(function(){

var m=0;

function getM(){

return m;

}

function setM(val){

m=val+2;

}

// 将两个函数发布出去,使得在外部可以调用

window.g=getM;

window.s=setM;

})();

s(7);

alert(g()); //弹出9（7+2）

</script>

</body>

</html>