logo.png

面向对象

H5 web前端

[www.bufanui.com](http://www.bufanui.com)

# call()和apply()

call()的官方解释，“调用一个对象的一个方法，以另一个对象替换当前对象。”

javaScript中有一个call和apply方法，其作用基本相同，但也有略微的区别。其实就是更改对象的内部指针，即改变对象的this指向的内容。这在面向对象的js编程过程中有时是很有用的，作用：

* 传递this
* 扩充对象能力

## ****call方法:****

语法：call([thisObj[,arg1[, arg2[,   [,.argN]]]]])

定义：调用一个对象的一个方法，以另一个对象替换当前对象。

说明：

call 方法可以用来代替另一个对象调用一个方法。call 方法可将一个函数的对象上下文从初始的上下文改变为由 thisObj 指定的新对象。

如果没有提供 thisObj 参数，那么 Global 对象被用作 thisObj。

## ****apply方法：****

语法：apply([thisObj[,argArray]])

定义：应用某一对象的一个方法，用另一个对象替换当前对象。

说明：

或者不是 arguments 对象，那么将导致一个 TypeError。

如果没有提供 argArray 和 thisObj 任何一个参数，那么 Global 对象将被用作 thisObj， 并且无法被传递任何参数。

**简单的例子：**

|  |
| --- |
| var x = "我是全局变量";　　　　//定义全局变量x  function a(){　　　　　　　　　//定义函数类结构a  this.x = "我是在函数类结构a中声明的";  }  //定义普通函数，弹出当前指针所包含的变量x的值  function f(){  alert (this.x);  }  //返回值为“我是在函数类结构a中声明的哦”  f.call(new a());  // f(); |

|  |
| --- |
| function Person(name, age, job) {  this.name = name;  this.age = age;  this.job = job;  this.sayName = function () {  alert(this.name);  }  }  // Person.apply(window,['wang',27,'php']);  Person.call(window,'张',20,'h5');  window.sayName(); |

# 二、执行环境、作用域链

执行环境（execution content）是javascript中最为重要的一个概念。执行环境定义了变量或函数有权访问的其他数据，决定了它们各自的行为。每一个执行环境都有一个与之关联的变量对象，环境中定义的所有变量和函数都保存在这个对象中。虽然我们的代码无法访问这个对象，但是解析器在处理数据时会在后台执行它。

全局执行环境是最外围的一个执行环境。根据ECMA实现所在的宿主环境不同，表示执行环境的对象也不同。

虽然目前我们遇到的最外层执行环境都是window对象，但是不代表所有的JavaScript代码执行环境都是window，比如node.js就没有window的执行环境。

每一个函数都有自己的执行环境。当执行流进入一个函数时，函数的环境就会被推入一个环境栈中。而在函数执行后，栈将其环境弹出，把控制权返还给之前的执行环境。当代码在一个环境中执行时，会创建变量对象的一个作用域链（scope chian）。作用域的用途，是保证对执行环境有权访问的所有变量和函数的有序访问。

作用域链案例:

1. 下级可以访问上级

2. 一级一级的查找

3. 如果找到，不再找 （例子：重新赋值）

4. 如果一直没找到 未定义

|  |
| --- |
| var color1 = "red";  function aa() {  var color2 = "green";  color2 = "black";  function bb() {  var color3 = "yellow";  color3 = color2;  color2 = color1;  color1 = 'pink';  console.log('color3='+color3);  console.log('color2='+color2);  }  bb();  }  aa();  console.log(color1); |

# 三、垃圾回收、块级作用域

对于其他语言来说，如C,C++,需要开发者手动的来跟踪内存，而JS的垃圾回收机制使得JS开发人员无需再关系内存的情况，所有的内存分配以及回收都会由垃圾回收器自动完成，执行环境会对执行过程中占有的内存负责。其原理就是找出那些不在被使用的变量，然后释放其所占有的内存。回收器一般是按照固定的时间间隔或者预设的时间进行处理的。

javaScript是一门具有自动垃圾收集机制的编程语言。开发人员不必关系内存分配和回收的问题。

离开作用域的值将会被自动回收

**标记法 标记不被占用的变量，定时回收**

**计数法 给初始变量一个num，如果被执行完毕，则--，如果为负值，则表示已经执行完毕，可回收。**

* 块级作用域

Js中没有块级作用域

For(var I = 0 ;I < 10; I ++){

}

让for循环变成一个块级作用域

模拟块状作用域 函数的自执行

# 四、Closur闭包

概念1: 闭包和函数有着紧密的关系，它是函数代码在运行过程中的一个动态的环境，是一个运行期的、动态的概念。

概念2: 闭包就是能够读取其他函数内部变量的函数。

......

问题：什么时候可以读取到其他函数内部变量？

函数嵌套函数、内部函数引用外部函数的参数和变量、参数和变量不会被垃圾回收机制所收回，所以，在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

* 闭包有什么好处？

1.希望一个变量长期驻扎在内存当中。

2.避免全局变量的污染

3.模拟私有成员变量

|  |
| --- |
| function f1(){  　　　　var n=999;  　　　　nAdd=function(){n+=1}  　　　　function f2(){  　　　　　　alert(n);  　　　　}  　　　　return f2;  　　}  　　var result=f1();  　　result();  　　nAdd();  　　result(); |

**案例：**

|  |
| --- |
| var name = "The Window";  　　var object = {  　　　　name : "My Object",  　　　　getNameFunc : function(){  　　　　　　return function(){  　　　　　　　　return this.name;  　　　　　　};  　　　　}  　　};  　　alert(object.getNameFunc()()); |

|  |
| --- |
| var name = "The Window";  　　var object = {  　　　　name : "My Object",  　　　　getNameFunc : function(){  　　　　　　var that = this;  　　　　　　return function(){  　　　　　　　　return that.name;  　　　　　　};  　　　　}  　　};  　　alert(object.getNameFunc()()); |

# 五、继承

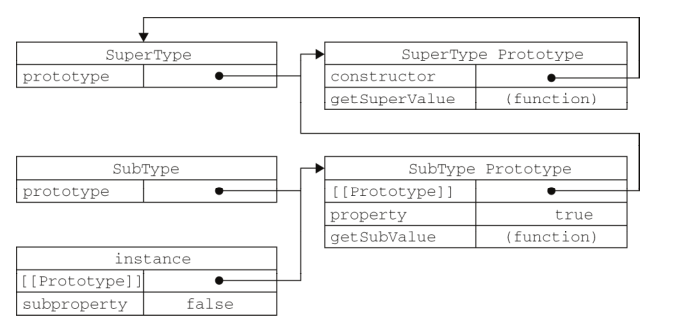
许多 OO 语言都支持两种继承方式：接口继承和实现继承。js无法实现接口(interface)继承，只支持实现继承，而且其实现继承主要是依靠原型链来实现的。

## 5.1 原型链

将原型链作为实现继承的主要方法，基本思想是利用原型让一个引用类型继承另一个引用类型的属性和方法。

让原型对象等于另一个类型的实例，就构成了实例与原型的链条，这就是所谓原型链的基本概念，也就实现了继承。

|  |
| --- |
| function SuperType(){  this.property = true;  }  SuperType.prototype.getSuperValue = function(){  return this.property;  };  function SubType(){  this.subproperty = false;  }  //继承了 SuperType  SubType.prototype = new SuperType();  SubType.prototype.getSubValue = function (){  return this.subproperty;  };  var instance = new SubType();  alert(instance.getSuperValue()); //true |



### 5.1.1 谨慎地定义方法

子类型有时候需要重写超类型中的某个方法，或者需要添加超类型中不存在的某个方法。但不管怎样，给原型添加方法的代码一定要放在替换原型的语句之后。

|  |
| --- |
| function SuperType(){  this.property = true;  }  SuperType.prototype.getSuperValue = function(){  return this.property;  };  function SubType(){  this.subproperty = false;  }  //继承了 SuperType  SubType.prototype = new SuperType();  // 添加新方法  SubType.prototype.getSubValue = function (){  return this.subproperty;  };  // 重写超类型中的方法  SubType.prototype.getSuperValue = function (){  return false;  };  var instance = new SubType();  alert(instance.getSuperValue()); //false |

不能使用对象字面量创建原型方法，因为这样做就会重写原型链。

|  |
| --- |
| function SuperType(){  this.property = true;  }  SuperType.prototype.getSuperValue = function(){  return this.property;  };  function SubType(){  this.subproperty = false;  }  //继承了 SuperType  SubType.prototype = new SuperType();  // 使用字面量添加新方法，会导致上一行代码无效  SubType.prototype = {  getSubValue : function (){  return this.subproperty;  },  someOtherMethod : function (){  return false;  }  };  var instance = new SubType();  alert(instance.getSuperValue()); //error! |

### 5.1.2 原型链的问题

原型链虽然很强大，可以用它来实现继承，但它也存在一些问题。其中，最主要的问题来自包含引用类型值的原型。如果存在引用类型属性，会导致共享

|  |
| --- |
| function SuperType(){  this.colors = ["red", "blue", "green"];  }  function SubType(){  }  //继承了 SuperType  SubType.prototype = new SuperType();  var instance1 = new SubType();  instance1.colors.push("black");  alert(instance1.colors); //"red,blue,green,black"  var instance2 = new SubType();  alert(instance2.colors); //"red,blue,green,black" |

原型链的第二个问题是：在创建子类型的实例时，不能向超类型的构造函数中传递参数。

## 5.2 借用构造函数

在解决原型中包含引用类型值所带来问题的过程中，开发人员开始使用一种叫做借用构造函数（constructor stealing）的技术（有时候也叫做伪造对象或经典继承）。这种技术的基本思想相当简单，即在子类型构造函数的内部调用超类型构造函数。

|  |
| --- |
| function SuperType(){  this.colors = ["red", "blue", "green"];  }  function SubType(){  // 继承了 SuperType  SuperType.call(this);  }  var instance1 = new SubType();  instance1.colors.push("black");  alert(instance1.colors); //"red,blue,green,black"  var instance2 = new SubType();  alert(instance2.colors); //"red,blue,green" |

### 5.2.1 传递参数

|  |
| --- |
| function SuperType(name){  this.name = name;  }  function SubType(){  //继承了 SuperType，同时还传递了参数  SuperType.call(this, "Nicholas");  //实例属性  this.age = 29;  }  var instance = new SubType();  alert(instance.name); //"Nicholas";  alert(instance.age); //29 |

### 5.2.2 借用构造函数的问题

如果仅仅是借用构造函数，那么也将无法避免构造函数模式存在的问题——方法都在构造函数中定义，因此函数复用就无从谈起了。

## 5.3 组合继承

组合继承（combination inheritance），有时候也叫做伪经典继承，指的是将原型链和借用构造函数的技术组合到一块，从而发挥二者之长的一种继承模式。其背后的思路是使用原型链实现对原型属性和方法的继承，而通过借用构造函数来实现对实例属性的继承。既通过在原型上定义方法实现了函数复用，又能够保证每个实例都有它自己的属性。

|  |
| --- |
| function SuperType(name){  this.name = name;  this.colors = ["red", "blue", "green"];  }  SuperType.prototype.sayName = function(){  alert(this.name);  };  function SubType(name, age){  //继承属性  SuperType.call(this, name);  this.age = age;  }  //继承方法  SubType.prototype = new SuperType();  SubType.prototype.constructor = SubType;  SubType.prototype.sayAge = function(){  alert(this.age);  };  var instance1 = new SubType("Nicholas", 29);  instance1.colors.push("black");  alert(instance1.colors); //"red,blue,green,black"  instance1.sayName(); //"Nicholas";  instance1.sayAge(); //29  var instance2 = new SubType("Greg", 27);  alert(instance2.colors); //"red,blue,green"  instance2.sayName(); //"Greg";  instance2.sayAge(); //27 |

组合继承避免了原型链和借用构造函数的缺陷，融合了它们的优点，成为 JavaScript 中最常用的继承模式。而且， instanceof 和 isPrototypeOf() 也能够用于识别基于组合继承创建的对象。

## 5.4 第三方框架实现继承

~~base2.js simple.js //~~国外牛人 封装的简单继承方式 按照规则 能够方便的实现继承。

|  |
| --- |
| var Person = Class. extend ( {  init: function (isDancing ) {  this. dancing = isDancing;  }  } );  var Ninja = Person.extend({  init: function(){  this.\_super( false );  }  });  var p = new Person(true);  p.dancing; // => true  var n = new Ninja();  n.dancing; // => false |

有几点需要留意：

* 构造函数须简单（通过init函数来实现），
* 新定义的类比须继承于已有的类，
* 所有的‘类’都继承于始祖类：Class，因此如果要创建一个全新的类，该类必须为Class的子类，
* 最具挑战的一点：父类的被覆写方法必须能访问到（通过配置上下文环境）。  
  在上面的示例中，你能发现通过this.\_super()来调用Person父类的init()和dance()方法。

对结果相当满意：使类的定义结构化，保持单一继承，并且能够调用超类方法。

案例：图形周长计算