**JavaScript Function 函数深入解析**

整理了JavaScript中函数Function的各种，感觉函数就是一大对象啊，各种知识点都能牵扯进来，不单单是 Function 这个本身原生的引用类型的各种用法，还包含执行环境，作用域，[闭包](http://www.codeceo.com/article/javascript-bibao.html" \o "闭包" \t "_blank)，上下文，私有变量等知识点的深入理解。

**函数中的return**

1. return 语句可以不带有任何返回值，在这种情况下（ return; 或函数中不含 return 语句时），函数在停止执行后将返回 undefiend 值。这种用法一般在需要提前停止函数执行而又不需要返回值的情况下。
2. return false 可以阻止元素的默认事件。
3. return 返回的是其所在函数的返回值
4. function n(){
5. (function(){
6. return 5;
7. })();
8. }
9. n();// undefined
10. //立即执行匿名函数中的return语句其实是返回给它所在的匿名函数的。
11. function n(){
12. var num= (function(){
13. return 5;
14. })();
15. console.log(num);
16. }

**Function类型**

函数实际上是对象，每个函数实际上都是 Function 类型的实例。而且与其他引用类型一样具有属性和方法。函数名实际上是一个指向内存堆中某个函数对象的指针。

定义函数的方式

1. 函数声明
2. function sum(num1,num2){
3. return num1+num2;
4. }
5. 函数表达式
6. var sum=function(num1,num2){
7. return num1+num2;
8. };

定义了一个变量 sum 并将其初始化为一个函数，注意到 function 关键字后面并没有函数名，这是因为在使用函数表达式定义函数，没必要使用函数名，通过变量 sum 即可引用函数。还要注意函数末尾有个分号，就像声明其他变量一样。

1. new 构造函数，虽然这种用法也是函数表达式，但该用法不推荐。因为这种语法会导致解析两次代码（第一次是解析常规的ECMAScript代码，第二次是解析传入构造函数中的字符串），影响性能。  
   使用 Function 构造函数，构造函数可以接受任意数量的参数，但最后一个参数始终都被看成是函数体，前面的参数则枚举出了新函数的参数。
2. var sum=new Function('num1','num2','return num1+num2;');
3. sum;//
4. function anonymous(num1,num2
5. /\*\*/) {
6. return num1+num2;
7. }

当使用不带圆括号的函数名是访问函数指针，而非调用函数。

**理解参数**

ECMAScript中所有参数传递的都是值（即使是引用也是传递的地址值，不是引用传递参数（**可参考**[**JavaScript传递参数是按值传递还是按引用传递**](https://www.zhihu.com/question/27114726)））。ECMAScript函数不介意传递进来多少个参数，也不在乎传进来的参数是什么数据类型。之所以这样，是因为ECMAScript中的参数在内部是用一个数组表示的。函数接收到的始终都是这个数组，而不关心数组中包含哪些参数。在函数体内，可以通过 arguments 对象来访问这个数组。从而获取传递给函数的每个参数。

1. function func(){
2. console.log(Object.prototype.toString.call(arguments));
3. }
4. func();// [object Arguments]
5. 关于 arguments 的行为，它的值永远与对应命名参数的值保持同步。因为 arguments **对象中的值会自动反映到对应的命名参数**。所以修改 arguments[1] ，也就修改了 num2 。不过这并不是说读取这两个值会访问相同的内存空间，它们的内存空间是独立的，但他们值会同步（WHY？？），要是JavaScript能直接访问内存就好了验证一下。
6. 但如果只传入了一个参数，那么 arguments[1] 设置的值不会反映到命名参数中，这是因为 arguments 对象的长度是由传入参数个数决定的，不是由定义函数时的命名参数个数决定的，没有传递值的命名参数将自动被赋予 undefiend 值，这就跟定义了变量但没初始化一样。
7. function doAdd(num1,num2){
8. console.log(arguments.length);
9. console.log(num2)
10. arguments[1]=10;
11. console.log(num2);
12. }
13. doAdd(5,0);//2 0 10
14. doAdd(5);//1 undefiend undefined

**没有重载**

ECMAScript函数不能像传统意义上那样实现重载，而在其他语言中（Java），可以为一个函数编写两个定义，只要这两个定义的签名（接收参数的类型和数量）不同即可。

不能实现重载的原因：

1. ECMAScript函数没有签名，因为其参数是由包含零个或多个值的数组来表示的。没有函数签名，真正的重载是不可能做到的。在ECMAScript中定义两个名字相同的的函数，则该名字只属于后定义的函数。如何实现类似于Java中的重载呢，其实可以通过判断传入函数的参数类型和个数来做出不同响应。
2. function reload(){
3. if(arguments.length==0){
4. console.log('没传参');
5. }else if(arguments.legth==1){
6. console.log('传了一个参数');
7. }
8. }
9. 深入理解：将函数名想象为指针，也有助于理解为什么ECMAScript中没有函数重载的概念。
10. function add(){
11. return 100;
12. }
13. function add(num){
14. return num+200;
15. }
16. //实际上和下面代码没什么区别
17. function add(){
18. return 100;
19. }
20. add=function(num){
21. return num+200;
22. }

**函数声明和函数表达式**

实际上解析器在向执行环境中加载数据时，对函数声明和函数表达式并非一视同仁。

[JavaScript运行机制浅探](http://www.codeceo.com/article/javascript-run-learn.html) 中了解到对于解释型语言来说，编译步骤为：

1. 词法分析（将字符流转换为记号流，是一对一的硬性翻译得到的是一堆难理解的记号流）
2. 语法分析（这里进行所谓的变量提升操作，其实我觉得是把这些提升的变量保存在语法树中。要构造语法树，若发现无法构造就会报语法错误，并结束整个代码块的解析）
3. 之后可能有语义检查，代码优化等。得到语法树后就开始解释执行了。解释性语言没有编译成二进制代码而是从语法树开始执行。

解析器会先读取函数声明，并使其在执行任何代码之前可用。至于函数表达式，则必须等到执行阶段才会被真正赋值。什么意思呢？虽然两者都进行了变量提升，待真正执行时构造活动对象从语法树种取声明添加到执行环境中，但一个是函数提升，一个是变量提升。

1. //函数声明
2. console.log(func);//function func(){}
3. function func(){
4. }
5. //函数表达式
6. console.log(func1);// undefined
7. var func1=function(){};
8. console.log(func1);// function(){}

**作为值的函数**

因为ECMAScript中的函数名本身就是变量，所以函数也可以作为值来使用。不仅可以像传递参数一样把一个函数传递给另一个函数，而且可以将一个函数作为另一个函数的结果返回。

1. function callSomeFunction(someFunction,someArgument){
2. return someFunction(someArgument);
3. }
4. function concated(str){
5. return "Hi "+str;
6. }
7. callSomeFunction(concated,'xx');// 'Hi xx'

从一个函数中返回另一个函数的应用：假设有一个对象数组，想要根据某个对象属性对数组进行排序，但传给 sort() 方法的比较函数要接收两个参数，即要比较的**值**。我们需要一种方式来指明**按照哪个属性来排序**。我们可以定义一个函数它接收一个属性名，然后根据这个属性名来创建一个比较函数。默认情况下， sort 函数会调用每个对象的 toString() 方法以确定它们的次序。

1. function createCompare(property){
2. return function(obj1,obj2){
3. var value1=obj1[property],
4. value2=obj2[property];
5. if(value1<value2) return -1;
6. else if(value1>value2) return 1;
7. else return 0;
8. }
9. }
10. var data=[{name:'aa',age:20},{name:'bb',age:12},{name:'cc',age:30}];
11. data.sort(createCompare("age"));// [{name:'bb',age:12},{name:'aa',age:20},{name:'bb',age:30}]

**函数的内部属性**

arguments ：类数组对象，包含传入函数中所有参数。是每个函数自身的属性，之所以可以直接访问 arguments ，是因为命名空间？？以下变化是为了加强JavaScript语言的安全性，这样第三方代码就不能在相同的环境下窥视其他代码了。

* callee 属性：是一个指针，指向拥有 arguments 对象的函数。严格模式访问会导致错误。

1. //一般阶乘函数
2. function factorial(num){
3. if(num<=1){ return 1;}
4. else {
5. return num\*factorial(num-1);
6. }
7. }

定义阶乘函数用到递归算法，这样定义是没问题。  
缺点：这个函数的执行与函数名 factorial 紧紧耦合在一起。万一出现改变函数指向的这种情况就不太好了，

1. factorial=function(){}
2. factorial(3);// undefiend

为了消除这种现象。

1. function factorial(num){
2. if(num<=1){ return 1; }
3. else{
4. return num\*arguments.callee(num-1);
5. }
6. }

* 这样无论引用函数使用的是什么名字都可以保证完成递归。
* caller 属性：不过在非严格模式下这个属性始终是 undefiend 。即使在严格模式下访问也会出错。增加这个属性是为了分清 arguments.caller 和函数对象上的 caller 属性。

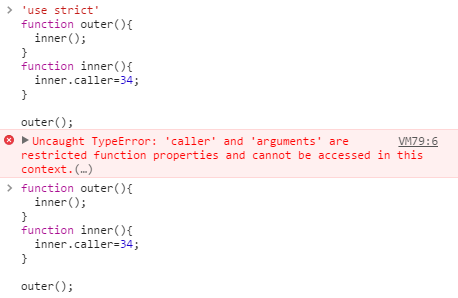
1. function a(){
2. return Object.getOwnPropertyNames(arguments);
3. }
4. a();// ["length", "callee"]

this ：行为与Java/C#中的 this 大致类似。 this 引用的是函数据以执行**环境对象**（当在网页的全局作用域中调用函数时， this 对象引用的就是 window ）。

caller ：不止是ECMAScript5中新增函数对象上的属性，还是 arguments 上的属性。保存着调用当前函数的函数的引用。如果是在全局作用域中调用当前函数，它的值为 null 。  
Object.getOwnPropertyNames(Function);// ["length", "name", "arguments", "caller", "prototype"]

1. function outer(){
2. inner();
3. }
4. function inner(){
5. console.log(inner.caller); //为了实现更松散的耦合，arguments.callee.caller
6. }
7. outer();// function outer(){ inner()}

严格模式下不能为函数的 caller 属性赋值，否则会导致出错。



**函数的属性和方法**

* length：表示函数希望接收的命名参数的个数（也就是定义的形参的个数）。

1. function sayName(name){
2. //
3. }
4. function sum(num1,num2){
5. //
6. }
7. function sayHi(){
8. //
9. }
10. sayName.length;// 1
11. sum.length;// 2
12. sayHi.length;// 0

* prototype：对于ECMAScript中的引用类型而言，prototype是保存它们所有实例方法的真正所在。诸如toString和valueOf等方法实际上都保存在Object.prototype名下（原生构造函数比如Function,Array等 在自己原型上重写了toString）。在ECMAScript5中，prototype属性是不可枚举的，因此使用for-in无法发现。 Object.getOwnPropertyDescriptor(Function,’prototype’);//Object {writable: false, enumerable: false, configurable: false}
* 每个函数上有两个可用的方法：apply和call。这两个方法实际上是在Function.prototype上， Object.getOwnPropertyNames(Function.prototype);// ["length", "name", "arguments", "caller", "apply", "bind", "call", "toString", "constructor"] 它是在JavaScript引擎内部实现的。因为是属于Function.prototype，所以每个Function的实例都可以用（自定义的函数也是Function的实例）。都是在特定的作用域或自定义的上下文中调用执行函数，实际上等于设置函数体内 this 对象的值。

1. apply ：参数一为在其中运行函数的作用域，参数二为参数数组（可以是数组，也可以是 arguments 对象）。
2. function sum(num1,num2){
3. return num1+num2;
4. }
5. function callSum1(num1,num2){
6. return sum.apply(this,arguments);//sum.apply(this,[num1,num2])
7. }
8. callSum1(10,30);// 40

严格模式下，未指定环境对象而调用函数， this 值不会转型为 window 。除非明确把函数添加到某个对象或者调用 apply 或 call ，否则 this 值将是 undefined

1. call ：参数一没有变化，变化的是其余参数都是直接传递给函数，参数必须都列出来。
2. function callSum1(num1,num2){
3. retrun sum.call(this,num1,num2);
4. }
5. callSum1(10,30);// 40

call 和 apply 真正强大的地方是能够扩充函数赖以运行的作用域，改变函数的执行环境。

1. bind ：ECMAScript5定义的方法，也是 Function.prototype 上的方法。用于控制函数的执行上下文，返回一个新函数，这个函数的 this 值会被绑定到传给 bind() 函数中的值。
2. window.color="red";
3. var o={color:'blue'};
4. function sayColor(){
5. console.log(this.color);
6. }
7. var newobj=sayColor.bind(o);
8. newobj;// function sayColor(){
9. console.log(this.color);
10. }
11. newobj==sayColor;// false
12. newobj();// blue

深入理解：可以将函数绑定到指定环境的函数。接收一个函数和一个环境，返回在给定环境中调用给定函数的函数。

1. function bind(func,context){
2. return function(){
3. func.apply(context,arguments);//这里创建了一个闭包，arguments使用的返回的函数的，而不是bind的
4. }
5. }

当调用返回的函数时，它会在给定环境中执行被传入的函数并给出所有参数。

1. function bind(func,context,args){
2. return function(){
3. func.call(context,args);
4. };
5. }
6. toString，toLocaleString ：返回函数代码的字符串形式，返回格式因浏览器而异，有的返回源码，有的返回函数代码的内部表示，由于存在差异，用这个也实现不了什么功能。
7. valueOf ：返回函数的自身引用。

**变量，作用域，内存问题**

JavaScript接近词法作用域，变量的作用域是在定义时决定而不是在执行时决定，也就是说词法作用域取决于源码。

JavaScript引擎在执行每个函数实例时，都会为其创建一个执行环境，执行环境中包含一个AO变量对象，用来保存内部变量表，内嵌函数表，父级引用列表等语法分析结构（变量提升在语法分析阶段就已经得到了，并保存在语法树中，函数实例执行时会将这些信息复制到AO上）。

ECMA-262定义，JavaScript松散类型的本质决定了它只在特定时间用于保存特定值的一个名字而已，由于不存在定义某个变量必须要保存何种数据类型值得规则，变量的值及其数据类型可在脚本的生命周期内改变。

* 基本类型和引用类型的值：ECMAScript变量可能包含两种不同数据类型的值：基本类型值，引用类型值。

1. 基本类型值：简单的数据段。
2. 引用类型值：那些可能由多个值构成的对象。是保存在内存中的对象，JavaScript不允许直接访问内存中的位置，也就说不能直接操作对象的内存空间。在操作对象时实际上是在操作对象的引用而不是实际的对象。为此，引用类型值是按引用访问的。（这种说法不严密，当复制保存着对象的某个变量时，操作的是对象的引用。但在为对象添加属性时，操作的是实际的对象）  
   在将一个值赋给变量时，解析器必须确定这个值是基本类型值还是引用类型值。5种基本数据类型： Undefined，Null，Boolean，Number，String （很多语言中字符串以对象形式来表示因此被认为是引用类型，但ECMAScript放弃这一传统）。这5种基本类型是按值访问的，因此可以操作保存在变量中的实际的值。

* 动态的属性
* 复制变量的值：在从一个变量向另一个变量复制基本类型值和引用类型值时，也存在不同。  
  如果从一个变量向另一个变量复制基本类型的值，会在变量对象上创建一个新值，然后把该值复制到为新变量分配的位置上。  
  当从一个变量向另一个变量赋值引用类型值值时，同样也会将存储在变量对象中的值复制一份放到为新变量分配的空间中，不同的是，这个值的副本实际上是个指针（可以理解为复制了地址值），而这个指针指向存储在堆中一个对象。复制操作结束后两个变量实际上将引用同一个对象。
* 传递参数：ECMAScript中所有函数的参数都是按值传递的，把函数外部的值复制给函数内部的参数，就和把值从一个变量复制到另一个变量一样。基本类型值得传递如同基本类型变量的复制一样，引用类型值的传递如同引用类型变量的复制一样。很多人错误认为：在局部作用域中修改的对象会在全局作用域中反映出来这就说明是按引用传递的。为了证明对象是按值传递的，

1. function setName(obj){
2. obj.name="xx";
3. obj=new Object();
4. obj.name="bb";
5. }
6. var p=new Object();
7. setName(p);
8. p.name;// "xx"

如果是按引用传递的，即传递的不是地址值而是堆内存中整个p对象，在 setName 中为其添加了一个新名字叫 obj ，又给其添加 name 属性后，将这个 obj 内容重新填充为新对象，那么之前的那个对象就不存在了更别说有 ”xx” 的名字属性，但是 p.name 仍然访问到了。这说明即使在函数内部修改了参数值，但原始的引用仍然保持未变。实际上，当在函数内部重写 obj 时，这个变量引用的就是一个局部对象了，而这个局部对象会在函数执行完毕后被立即销毁。

* 类型检测：检测一个变量是不是基本数据类型用 typeof 是最佳工具，但如果变量的值是除了函数的对象或 null  typeof [];// ”object” typeof null;// ”object” ，变量值为函数时 typeoffunction(){};// ”function” （ECMA-262规定任何在内部实现 [[call]] 方法的对象都应该在应用 typeof 操作符返回 ”function” ）。但在检测引用类型值时，这个操作符用处不大，因为我们并不是想知道它是个对象，而是想知道它是某种类型对象。如果变量是给定引用类型的实例， instanceof 操作符会返回 true 。所有引用类型值都是 Object 的实例。如果使用 instanceof 操作符检测基本类型的值，则该操作符始终会返回 false ，因为基本类型不是对象。

**执行环境及作用域**

* 执行环境（execution context）：也称为作用域，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。全局执行环境是最外围的一个执行环境，跟据ECMAScript实现所在的宿主环境不同，表示执行环境的对象也不一样，web浏览器中全局执行环境是 window 对象。某个执行环境中所有代码执行完毕后该环境被销毁，保存在其中的所有变量和函数定义也随之销毁（全局执行环境直到应用程序退出例如关闭网页或浏览器时才被销毁）。每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，在函数执行后，栈将其环境弹出，将控制权返回给之前的执行环境。ECMAScript程序中的执行流正是由这个机制控制着。函数的每次调用都会创建一个新的执行环境。执行环境分为创建和执行两个阶段，

1. 创建：解析器初始化变量对象或者活动对象，它由定义在执行环境中的变量，函数声明，参数组成。在这个阶段，作用域链会被初始化，this的值也最终会被确定。
2. 执行：代码被解释执行

* 变量对象（variable object）：环境中定义的所有变量和函数都保存在这个对象中。虽然用代码无法访问它，但解析器在处理数据时会在后台使用它。如果这个环境是函数，则将活动对象（activation object）作变量对象
* 作用域（scope）和上下文（context）：函数的每次调用都有与之紧密相关的作用域和上下文。作用域是基于函数的，上下文是基于对象的。作用域涉及到被调函数中变量的访问，上下文始终是 this 关键字的值，它是拥有当前所执行代码的对象的引用。上下文通常取决于函数是如何被调用的。
* 作用域链（scope chain）：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。它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。作用域链的前端始终都是当前执行的代码所在环境的变量对象。活动对象在最开始时只包含一个变量即 arguments 对象（这个对象在全局环境中不存在），作用域链的下一个变量对象来自包含（外部）环境，再下一个变量对象则来自下一个包含环境，这样一直延续到全局执行环境。

1. var color = "blue";
2. function changeColor(){
3. if(color=="blue"){
4. color="red";
5. }else{
6. color="blue";
7. }
8. }
9. changeColor();
10. console.log(color);// red

标识符解析是沿着作用域链一级一级地搜索标识符的过程，函数 changeColor 作用域链包含两个对象：它自己的变量对象（其中定义着 arguments 对象）和全局环境的变量对象。可以在函数内部访问到变量 color 就是因为可以在这个作用域链中找到它。内部环境可以通过作用域链访问所有外部环境，但外部环境不能访问内部环境的任何变量和函数。函数参数也被当作变量来对待，因此其访问规则与执行环境中的其他变量相同。

* 延长作用域链：有些语句可以在作用域的前端临时添加一个变量对象，该变量对象会在代码执行后被移除。当执行流进入下列语句时，作用域链就会加长。

1. try-catch 语句的 catch 块：对 catch 语句来说，会创建一个新的变量对象，其中包含的是被抛出的错误对象的声明。<=IE8版本中，在 catch 语句中捕获的错误对象会被添加到执行环境的变量对象而不是 catch 语句的变量对象，换句话说，即使是在 catch 块的外部也可以访问到错误对象。
2. with 语句：会将指定的对象添加到作用域链中。
3. function buildUrl(){
4. var qs="?debug=true";
5. with(location){
6. var url=href+qs;
7. }
8. return url;
9. }
10. buildUrl();// "http://i.cnblogs.com/EditPosts.aspx?postid=5280805?debug=true"

with 语句接收的是一个 location 对象，因此其变量对象中就含有 location 对象的所有属性和方法，且这个变量对象被添加到了作用域链的最前端。当在 with 语句中引用变量 href （实际引用的是 location.href ）可以在当前的执行环境中找到，当引用变量 qs 时，引用的则是在下一级执行环境中的变量。由于JavaScript中没有块级作用域，所以在函数内部可以访问 url 才能 return 成功，说明 url 并不是添加到 location 所在的变量对象中。  
这两个语句都会在作用域的前端添加一个变量对象。

* 没有块级作用域：在其他类C的语言中，由花括号封闭的代码块都有自己的作用域（如果用ECMAScript的话来讲，就是他们自己的执行环境），因而支持根据条件来定义变量。如果是在C/C++/Java中， color 会在 if 语句执行完后被销毁，但在JavaScript中， if 语句中的变量声明会将变量添加到当前的执行环境中。

1. if(true){
2. var color="red";
3. }
4. console.log(color);// red
5. 声明变量：使用 var 声明的变量会自动被添加到最接近的环境中。在函数内部，最接近的环境就是函数的局部环境；在 with 语句中，最接近的环境是函数环境。如果初始化变量时没有使用 var 声明，该变量会自动被添加到全局环境。
6. 查询标识符：当在某个环境中为了读取或写入而引用一个标识符时，必须通过搜索来确定该标识符代表什么。搜索过程从作用域链的前端开始，向上逐级查询与给定名字匹配的标识符。如果在局部环境中找到了该标识符，搜索过程停止，变量就绪。如果在局部环境中未找到该变量名，则继续沿作用域链向上搜索。搜索过程将一直追溯到全局环境。如果局部环境存在同名标识符，就不会使用位于父环境中的标识符。

**函数表达式**

1. if(condition){
2. function sayHi(){
3. console.log("Hi");
4. }
5. }else{
6. function sayHi(){
7. console.log("Yo");
8. }
9. }

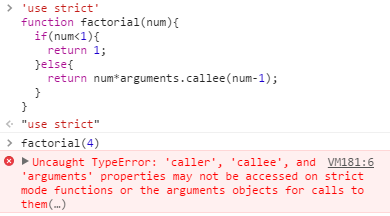
以上代码会在 condition 为 true 时使用 sayHi() 的定义，否则就使用另一个定义。实际上这在ECMAScript中属于无效语法，JavaScript引擎会尝试修正错误，将其转换为合理的状态。但问题是浏览器尝试修正的做法不一样。大多数浏览器会返回第二个声明。此种方式很危险，不应该出现你的代码中。在chrome中：

1. if(true){
2. function sayHi(){
3. console.log("Hi");
4. }
5. }else{
6. function sayHi(){
7. console.log("Yo");
8. }
9. }//function sayHi(){ 没有函数声明的变量提升？？
10. console.log("Hi");
11. }
12. if(false){
13. function say(){
14. console.log("Hi");
15. }
16. }else{
17. function say(){
18. console.log("Yo");
19. }
20. }//function say(){
21. console.log("Yo");
22. }
23. console.log(sa);//undefined 能输出undefiend说明函数声明并没有提升而是进行的变量提升
24. if(false){
25. function sa(){
26. console.log("Hi");
27. }
28. }else{
29. function sa(){
30. console.log("Yo");
31. }
32. }//function sa(){
33. console.log("Yo");
34. }

修正：使用函数表达式，那就没什么问题了。

1. var sayHi;
2. if(condition){
3. sayHi=function(){
4. console.log("Hi");
5. }
6. }else{
7. sayHi=function(){
8. console.log("Yo");
9. }
10. }

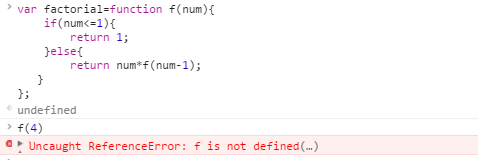
**递归**



在严格模式下，不能通过脚本访问 arguments.callee 。不过可以使用命名函数表达式来完成相同结果。

1. var factorial=(function f(num){
2. if(num<=1){
3. return 1;
4. }else{
5. return num\*f(num-1);
6. }
7. });

注意是用命名函数表达式，单单把命名函数赋值给 factorial 也可以，但是并不能通过f的名字访问



**闭包**

闭包是指有权访问另一个函数作用域中的变量的函数。创建闭包的常见方式，就是在一个函数内部创建另一个函数。之所以能够访问外部作用域的变量，是因为内部函数的作用域链中包含外部作用域。当一个函数被调用的时候，

1. 创建一个执行环境（execution context）及相应的作用域链
2. 使用 arguments 和其他命名参数的值来初始化活动对象（activation object），但在作用域链中，外部函数的活动对象始终始终处于第二位…直至作为作用域链终点的全局执行环境。

function compare(value1,value2){

if(value1<value2){

return -1;

}else if(value1>value2){

return 1;

}else{

return 0;

}

}

var result=compare(5,10);

当调用 compare() 时，会创建一个包含 arguments ， value1 ， value2 的活动对象，全局执行环境的变量对象（包含 result 和 compare ）在 compare() 执行环境的作用域链中处于第二位。

后台的每个执行环境都有一个表示变量的对象（变量对象），全局环境的变量对象始终存在，而像 compare() 函数这样的局部环境的变量对象，则只在函数执行过程中存在。在**创建 compare()**函数时，会创建一个预先包含全局对象的作用域链，这个作用域链被保存在 compare 内部的 [[Scope]] 属性中。当**调用 compare()**函数时，会为函数创建一个执行环境，然后通过复制函数的 [[Scope]] 属性中的对象构建起执行环境的作用域链。此后又有一个活动对象被创建并被推入执行环境作用域链的最前端。对于这个例子中， compare 函数的执行环境而言，其作用链包含两个变量对象：本地活动对象和全局变量对象。显然，**作用域链的本质上是一个指向变量对象的指针列表，它只引用但不包含实际的变量对象。**

无论什么时候在函数中访问一个变量，就会从作用域链中搜索具有相应名字的变量，一般来讲当函数执行完后，局部活动对象会被销毁，内存中仅保留着全局作用域（全局执行环境的变量对象）。但是闭包的情况又有所不同。在另一个函数内部定义的函数会将包含函数（外部函数）的活动对象添加到它的作用域链里，当外部函数执行完后其活动对象不会被销毁，因为匿名函数的作用域链仍然在引用这个活动对象。换句话说只是外部函数它自己的作用域链被销毁，但活动对象还存在内存中。直到内部函数被销毁后（例如在外部解除了对闭包即内部函数的引用： func=null; ，解除相当于是闭包仅是执行完后），外部函数的活动对象才会被销毁。

由于闭包会携带包含它的函数的作用域，因此会比其他函数占用更多的内存。过多使用闭包可能会导致内存占用过多，建议只在绝对必要再考虑使用。但有的优化后的JavaScript引擎如V8会尝试回收被闭包占用的内存。

闭包缺点：作用域链的这种配置机制引出了一个副作用即闭包只能取得包含函数中任何变量的最后一个值。因为闭包保存的是整个变量对象，而不是某个特殊的变量。

function createFunctions(){

var result=new Array();

for(var i=0;i<3;i++){

result[i]=function(){

return i;

};

}

return result;

}

createFunctions()[0]();// 3

createFunctions()[1]();// 3

createFunctions()[2]();// 3

当执行 createFunctions 时，它的活动对象里有 arguments=[] ， result=undefiend ， i=undefiend ，执行完 createFunctions 后， result=[function(){return i},function(){return i},function(){return i}]，i=3 ；当此时执行 result 数组时，访问到的i的值总是为3，因为沿着 function(){return i;} 的作用域链查找变量，在外层函数的活动对象上找到i总是为3。数组中每个函数的作用域链中都保存着 createFunctions 的活动对象，所以这些函数们引用的都是同一个活动对象，同一个变量i。

解决方案：要的就是当时执行时的变量i，那么当时把这个i临时保存一下就可以了，但是保存在哪呢？将i保存在 function(){return i;} 的活动对象中，怎么保存呢？传给 arguments 就好了，只传进来还不行

function createFunctions(){

var result=new Array();

for(var i=0;i<3;i++){

result[i]=function(i){

return i;

};

}

return result;

}

createFunctions()[0]();// undefiend

因为访问i的时候先从自己所在函数的执行环境的活动对象搜索起，找到i发现 i=undefiend 有值就停止向上搜索了。问题就出在上一步中将i保存在活动对象中， result[i]=function(i){return i;} 这句的执行并没有给匿名函数传参，这只是表达式的赋值操作，又不是执行匿名函数。所以现在需要的就是通过某种方式去执行函数的操作把i的值当实参传进去，简单！在匿名函数外部加一层立即执行的匿名函数（这也增加了一层作用域了）。

function createFunctions(){

var result=new Array();

for(var i=0;i<3;i++){

result[i]=(function(i){

return function(){

return i;

}

})(i);

}

return result;

}

createFunctins()[0]();// 0

**this对象**

this 对象是在运行时基于函数的执行环境绑定的：

1. 全局函数中， this 等于 window
2. 函数被作为某个对象的方法调用时， this 等于那个对象
3. 匿名函数的执行环境具有全局性， this 指向 window
4. 通过 call() 或 apply() 改变函数执行环境的情况下， this 就会指向其他对象。

由于闭包编写的方式不同， this 的表现：

var name="the window";

var obj={

name:"the obj",

getNameFunc:function(){

//console.log(this==obj);

return function(){

console.log(this.name);

}

}

}

obj.getNameFunc()();// the window

obj.getNameFunc() 返回了一个新函数，然后在再全局环境中执行该函数。为什么匿名函数没有取得其包含作用域（外部作用域）的 this 对象呢？每个函数在被调用时，都会自动获得两个特殊的变量： this （创建作用域时获得）和 arguments （创建活动对象获得），内部函数在搜索这两个变量时，只会搜索到自己的活动对象为止，因此永远不可能直接访问外部函数的这两个变量。不过把外部函数作用域的 this 保存在一个闭包能够访问到的变量里就可以让闭包访问该对象了。

下面几种情况特殊的 this ：

var name="the window";

var obj={

name:"the obj",

getName:function(){

return this.name;

}

};

obj.getName();// "the obj"

(obj.getName)();// "the obj"

(obj.getName=obj.getName)();// "the window"

第一个是直接调用，第二个是调用后立即执行的表达式，第三个是执行了一条赋值语句，然后再调用返回的结果，赋值语句的返回了一个函数，然后全局环境下调用这个函数，见下图



**模仿块级作用域**

function outputNumber(count){

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

console.log(i);

}

var i;// 只变量提升，到后面执行代码步骤时候略过此

console.log(i);

}

outputNumber(3);// 0 1 2 3

JavaScript不会告诉你是否多次声明了同一个变量，遇到这种情况，它只会对后续的声明视而不见（不过它会执行后续声明中的变量初始化）。

匿名函数可以用来模仿块级作用域（私有作用域），语法如下：

(function(){

//这里是块级作用域

})();

以上代码定义并立即调用了一个匿名函数，将函数声明包含在一对圆括号中，表示它实际上是一个函数表达式。对于这种语法的理解：

var count=5;

outputNumbers(count);

这里初始化了变量 count 将其值设为5。但是这里的变量是没有必要的，因为可以把值直接传给函数  outputNumbers(5);  这样做之所以可行，是因为变量不过是值的另一种表现形式，因此用实际的值替换变量没有问题。

var someFunc=function(){

//这里是块级作用域

};

someFunc();

既然可以使用实际的值来取代变量 count ，那这里也用实际的值替换函数名。

function(){

//这里是块级作用域

}();

然而会报错，是因为JavaScript将 function 关键字当作一个函数声明的开始，而函数声明后面不能跟圆括号。但是函数表达式后面可以圆括号，这也就是为什么这样可以执行

var someFunc=function(){

//这里是块级作用域

}();

要将函数声明转化为函数表达式，

(function(){

//这里是块级作用域

})();

function outputNumber(count){

(function(){

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

console.log(i);

}

})();

console.log(i);// 报错

}

outputNumber(3);// 0 1 2

在 for 循环外边加了一个私有作用域，在匿名函数中定义的任何变量都会在执行结束时被销毁。在私有作用域中访问变量 count ，是因为这个匿名函数是一个闭包，它能访问包含作用域的所有变量。这种技术经常在全局作用域中被用在函数外部从而限制向全局作用域中添加过多的变量和函数。这种做法还可以减少闭包占用内存问题，因为没有指向匿名函数的引用，只要函数执行完毕，就可以立即销毁其作用域链了。

**私有变量**

严格来讲，JavaScript中没有私有成员的概念，所有对象属性都是公有的。不过有私有变量的概念，任何在函数中定义的变量，都可认为是私有变量，因为不能在函数外部访问这些变量。私有变量包括函数的参数，局部变量，在函数内定义的其他函数。如果在函数内部创建一个闭包，那么闭包通过自己的作用域也可以访问这些变量。利用这一点创建用于访问私有变量的公有方法。

把有权访问私有变量和私有函数的方法叫特权方法（privileged method）。

* 两种在自定义对象上创建特权方法的方式

1. 在构造函数中定义特权方法

function MyObject(){

//私有变量和私有函数

var privateVariable=10;

function privateFunction(){

return false;

}

// 特权方法

this.publicMethod=function(){

privateVariable++;

return privateFunction();

};

}

new MyObject();

这个模式在构造函数内部定义了所有私有变量和函数，又继续创建了能够访问这些私有成员的特权方法。能在构造函数中定义特权方法是因为特权方法作为闭包有权访问在构造函数中定义的所有变量和函数。对这个例子而言，变量 privateVariable 和方法 privateFunction 只能通过特权方法 publicMethod 访问。在创建 MyObject 实例后除了使用 publicMethod() 这一途径外没任何办法可以直接访问私有变量和函数。  
利用私有和特权成员，可以隐藏那些不应该被直接修改的数据

function Person(name){

this.getName=function(){

return name;

};

this.setName=function(value){

name=value;

};

}

var p1=new Person("aa");

p1.getName();// "aa"

var p2=new Person("bb");

p2.getName();// "bb"

p1.getName();// "aa"

以上方法定义两个特权方法，在 Person 构造函数外部没有任何办法直接访问 name ，由于这两个方法是在构造函数内部定义的，它们作为闭包能够通过作用域链访问 name 。私有变量 name 在每个 Person 实例都不相同，这么说吧，每次调用构造函数都会重新创建这两个方法， p1.getName 和 p2.getName 是不同的函数，虽然调用的是内存中同一个 Person 函数。但 new 构造新实例的步骤是：先创建新实例对象；再在该实例上调用 Person 函数初始化作用域及作用域链 this 等；再添加属性等。不管换成是

var o1={},o2={};

Person.call(o1,'aa');

Person.call(o2,'bb');

o1.getName();// "aa"

还是换成

function Person(obj,name){

obj.getName=function(){

return name;

};

obj.setName=function(value){

name=value;

};

}

var o1={},o2={};

Person(o1,"aa");

Person(o2,"bb");

o1.getName();// "aa"

都调用了两次 Person ，因为每次调用 Person 就会初始化 Person 的作用域，所以 p1.getName 和 p2.getName 所处的外围作用域是不一样的（之前还认为因为是调用了内存中同一个 Person ，以为 p1.getName 和 p2.getName 有同一个外围作用域，没考虑到每次调用函数实例都会重新初始化作用域）。  
缺点：在构造函数中定义特权方法要求你必须使用构造函数模式来达到这个目的。构造函数模式的缺点是针对每个实例都会创建同一组新方法，使用静态私有变量来实现特权方法就可以避免这个问题。

1. 静态私有变量通过在私有作用域中定义私有变量和函数，也可以创建特权方法。基本模式如下：

(function(){

//私有变量和私有函数

var privateVariable=10;

function privateFunction(){

return false;

}

//构造函数

MyObject=function(){};

//公有/特权方法

MyObject.prototype.publicMethod=function(){

privateVariable++;

return privateFunction();

};

})();

这个模式创建了个私有作用域，并在其中封装了一个构造函数和相应方法。公有方法在原型上定义，这一点体现典型原型模式。注意到这个模式在定义构造函数时并没使用函数声明，而是使用函数表达式，因为函数声明只能创建局部函数，我们也没有在声明 MyObject 时使用 var 关键字，就是想让它成为一个全局变量，能够在私有作用域之外被访问。但严格模式下未经声明的变量赋值会导致出错。可以修改为

'use strict';

var MyObject;

(function(){

//私有变量和私有函数

var privateVariable=10;

function privateFunction(){

return false;

}

//构造函数

MyObject=function(){};

//公有/特权方法

MyObject.prototype.publicMethod=function(){

privateVariable++;

return privateFunction();

};

})();

其实我觉得不用立即执行的匿名函数也可以实现这种在私有作用域中定义私有变量函数的模式，只要把这些放在一个函数中就可以了，然后再执行这个函数。

function staticFunction(){

//私有变量和私有函数

var privateVariable=10;

function privateFunction(){

return false;

}

//构造函数

MyObject=function(){};

//公有/特权方法

MyObject.prototype.publicMethod=function(){

privateVariable++;

return privateFunction();

};

}

staticFunction();

—–分割线—-  
这种模式在与构造函数中定义特权方法的主要区别就在于私有变量和函数是由实例共享的因为只调用了即只初始化了一次父环境（意思就是 p1.getName 和 p2.getName 所在的父环境都是同一个，不像构造函数模式中那样拥有各自父环境）。由于特权方法是在原型上定义的，因此所有实例都使用同一个函数。而这个特权方法作为一个闭包总是保存着对包含作用域的引用。

(function(){

var name="";

Person=function(value){

name=value;

};

Person.prototype.getName=function(){

return name;

};

Person.prototype.setName=function(value){

name=value;

};

})();

var p1=new Person("aa");

p1.getName();// "aa"

var p2=new Person("bb");

p2.getName();// "bb"

p1.getName();// "bb"

Person 构造函数和 getName 和 setName 都有权访问私有变量 name 。在这种模式下，变量 name 就成了一个静态的，由所有实例共享的属性。在一个实例上调用 setName() 会影响所有实例。  
以这种模式创建静态私有变量会因为使用原型而增进代码复用，但每个实例都没有自己的私有变量。  
多查找作用域链中的一个层次，就会在一定程度上影响查找速度，这正是使用闭包和私有变量的一个不足之处。

* 模块模式：前面的模式用于为自定义类型创建私有变量和特权方法。道格拉斯所说的模块模式则是为单例创建私有变量和特权方法，所谓单例（singleton）指的就是只有一个实例的对象，JavaScript是以对象字面量的方式来创建单例对象的。

var singleton={

name:value,

method:function(){

//这里是方法的代码

}

};

模块模式通过为单例添加私有变量和特权方法能够使其得到增强。语法如下：

var singleton=function(){

//私有变量和私有函数

var privateVariable=10;

function privateFunction(){

return false;

}

//特权/公有方法和属性

return {

publicProperty:true,

publicMethod:function(){

privateVariable++;

return privateFunction();

}

}

}();

这个模式使用了一个返回对象的匿名函数，将一个对象字面量作为函数返回。本质上这个对象字面量定义的是一个单例的公共接口。这种模式在需要对单例进行某些初始化同时又需要维护其私有变量时是非常有用的。

var application=function(){

//私有变量和函数

var components=new Array();

//初始化

components.push(new BaseComponent());

//公共

return {

getComponentCount:function(){

return components.length;

},

registerComponent:function(component){

if(typeof component=="object"){

components.push(component);

}

}

}

}();

在web应用程序中，经常需要使用一个单例来管理应用程序级的信息。如果必须创建一个对象并以某些数据对其进行初始化，同时还要公开一些能够访问这些私有数据的方法，那就可以使用模块模式。这种模式创建的每个单例都是 Object 的实例。

* 增强的模块模式：如果想让单例是某种类型的实例，改进了模块模式，在返回对象之前加入对其增强的代码。同时还必须添加某些属性和方法对其加以增强。

var singleton=function(){

//私有变量和私有函数

var privateVariable=10;

function privateFunction(){

return false;

}

//创建对象

var obj=new CustomType();

//添加特权/公有属性和方法

obj.publicProperty=true;

obj.publicMethod=function(){

privateVariable++;

return privateFunction();

}

return obj;

}();

var application=function(){

//私有变量和函数

var components=new Array();

//初始化

components.push(new BaseComponent());

//创建application的一个局部版本

var app=new BaseComponent();

app.getComponentCount=function(){

return components.length;

};

app.registerComponent=function(component){

if(typeof component=="object"){

components.push(component);

}

};

return app;

}();

**闭包的作用总结：**

1. 使用闭包可以在JavaScript中模仿块级作用域  
   创建并立即调用一个函数，这样既可以执行其中代码，又不会在内存中留下对该函数的引用。结果就是函数内部的所有变量都会被立即销毁除非将某些变量赋值给了包含作用域中的变量
2. 闭包还可用于在对象中创建私有变量  
   通过闭包来实现公有方法，通过公有方法可以访问在包含作用域中定义的变量。有权访问私有变量的公有方法叫特权方法。可以使用构造函数模式，原型模式来实现自定义类型的特权方法，使用模块模式，增强的模块模式实现单例的特权方法。