# 变量

定义变量要用var操作符，后跟变量名，如下：

var color;

像这样未经初始化的变量会被赋值为undefined，因此可以在定义变量的同时设置变量的值，如下：

var color = "red";

但是要注意var操作符定义的变量将成为定义该变量的作用域中的局部变量,例如这个例子：function doSomething (){

var color = "red";

}

doSomething();

alert(color);

结果就会返回，color没有被定义，原因是var使得color成为了doSomething这个函数的局部变量，color这个变量在函数退出后就会被销毁。

如果想返回正确结果，把var去掉就可以了。

可以用一条语句定义多个变量，只要像下面这样把每个变量（初始化或者不初始化均可）用逗号分开就可以了：

var color = "red",

age = 12,

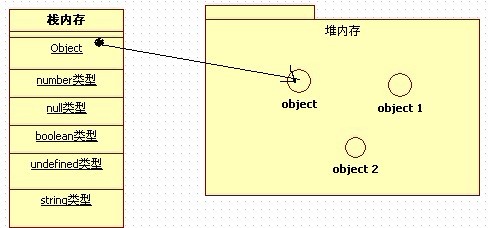
found = true;

变量又可分为两种不同的数据类型：基本类型和引用类型。

基本类型：5种Undefined,Null,Boolean,Number和String。

这五种基本数据类型的值在内存中分别占固定大小的空间，可以保存在栈内存中。对于保存基本类型的变量，他们是按值访问的。

引用类型：指那些保存在堆内存中的对象，大概意思是变量中保存的实际上只是一个指针，指向内存中的另外一个位置，该位置保存对象。当查找引用类型的变量时，可以先从栈内存中读取内存地址，然后在找到保存在堆中得值，这种方式叫做按引用访问。（注：下面会提到按值访问和按引用访问的区别。）

[](http://photo.blog.sina.com.cn/showpic.html#blogid=68cb8c790100yrxx&url=http://s4.sinaimg.cn/orignal/68cb8c79gb7b776baac33)

## 保存 基本类型和引用类型的共同点和区别：

共同点：定义它们的方式是类似的，创建一个变量并为该变量赋值。

区别：看下面的例子

var person = new Object();

person.name = "Tony";

alert (person.name); //Tony

你会看到Tony这个结果,这写代码先创建了一个对象，并保存在person中，然后给这个对象添加了一个name属性并赋值为"Tony"，然后我们alert访问这个新属性，自然得到了答案。我们并不能同样的为基本类型添加属性：

var name = "Tony";

name.age = 22;

alert (name.age); //undefined

这个例子中我们会看到我们并没有为字符串name添加上age属性。

总结：我们只能给引用类型动态的添加属性，以便未来使用。

(这里面不是没有为字符串name添上age属性，这里存在一个自动封装的问题，字符串是基本类型，而当我们访问或操作字符串的属性、方法时，js会自动将基本类型的字符串封装成对象（类似new String(name)），访问和操作之后再转换为基本类型（类似String的valueOf()），所以name.age = 22这条语句在执行过程中是对封装后的name字符串对象添加属性age，语句执行完之后封装的对象执行域退出，我们将无法再次访问此对象，alert (name.age)这条语句访问的age为再一次封装之后的字符串的属性，已经不是上次的那个对象了，自然是undefined了)

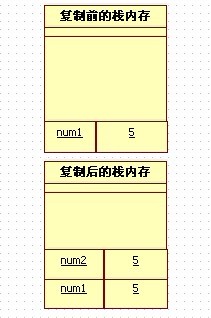
## 复制 变量的值

基本类型的复制：会在栈中创建一个新值，然后把该值复制到新变量分配的位置上。

var num1 = 5;

var num2 = num1;

储存过程如下图

[](http://photo.blog.sina.com.cn/showpic.html#blogid=68cb8c790100yrxx&url=http://s4.sinaimg.cn/orignal/68cb8c79gb7b7576c2853)

如图,num1中的5和num2中的5是完全独立的！这两个变量可以参与任何操作而不受任何影响。

引用类型的复制：同样会将储存在栈内存中的值复制一份，不同的是这个值的副本实际上是一个指针，他指向储存在堆中的一个对象。这两个对象实际上引用同一个对象。因此改变其中一个的值，就会影响另外一个。如下例：

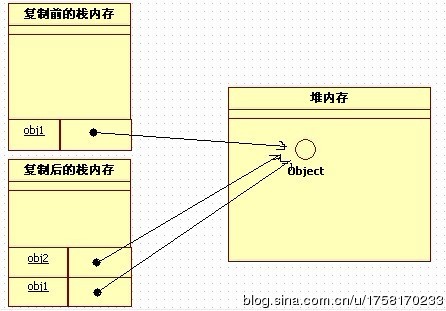
var obj1 = new Object();

var obj2 = obj1;

obj1.name = "Tony";

alert(obj2.name); //Tony

存储过程如下图：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=68cb8c790100yrxx&url=http://s12.sinaimg.cn/orignal/68cb8c79gb7b750e9705b)

如图所示，这两个变量引用的都是同一个对象。

## 传递参数：（参数只能按值传递）

1.向参数传递基本类型

function addTen (num){

num +=10;

return num;

}

var count =20;

var result = addTen(count);

alert(count);

alert(result);

这里的函数addTen有一个参数num，这个参数num实际上是函数的局部变量，在调用这个函数时，count作为参数被传给函数，这个变量的值是20，于是20就被复制给num，以便在函数addTen中使用。在函数内部，num的值被加上了10，就像上文中提到，它并不影响外面count变量的值，num和count仅仅是有相同的值。

2.向参数传递引用类型

function setName(obj){

obj.name = "Tony";

}

var person =new Object();

setName(person);

alert(person.name);

以上代码创建了一个对象，并保存在person中，然后这个对象被传送到setName函数中之后被复制给了obj,在这个函数内部，obj和person引用的是同一个对象，于是内部的obj添加了name属性后，外部的person也会有所反应，同样的被添加了name属性。

到此为止，肯定很多人认为向参数传递引用类型是按引用访问的，其实并不是，看下面的代码：

function setName(obj){

obj.name = "Tony";

obj = new Object();

obj.name = "Tom";

}

var person =new Object();

setName(person);

alert(person.name);

如果是按引用传递的话，结果会显示Tom，可是结果却是Tony，这就很好的说明向参数传递引用类型的时候仍然是按值传递的。

## 检测变量类型

### 检测基本类型

用typeof操作符 例：

var s = "Tony";

var b = true;

var i = 22;

var u =

var n = null;

var o = new Object;

alert (typeof s); //string

alert (typeof b); //boolean

alert (typeof i); //number

alert (typeof u); //undefined

alert (typeof n); //object

alert (typeof o); //object

typeof操作符可以很好的检测出基本类型。但检测引用类型时并不好用，我们不想知道他是不是对象，而是想知道他是什么类型的对象。

### 引用类型的检测

用instanceof操作符 例：

alert(person instanceof Object); //变量person是Object吗？

alert(color instanceof Array);

alert(color instanceof RegExp);

# js 闭包和匿名函数

## 匿名函数

匿名函数就是没有名字的函数。他有两种声明方式：

1.典型的函数声明：

function functionName(arg0,arg1,arg2){

//函数体

}

2.函数表达式:

var functionName = function(arg0,arg1,arg2){

//函数体

}

虽然这两种方式在逻辑上市等价的，但是他们还是存在区别的。

区别1：前者会在代码执行以前被加载到作用域中，而后者则是在代码执行到那一行的时候才会有意义。

区别2：前者会给函数指定一个名字，而后者则是创建一个匿名函数，然后将这个匿名函数赋给一个变量。

换句话说上面第二个例子：创建了一个带有3个参数的匿名函数，然后把这个匿名函数赋给了变量functionName，并没有给匿名函数指定名字。

## 闭包

书上定义是这么说的：指有权访问另一个函数作用域中的变量的函数

可是这种说法令新手难以理解。其实，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。

1.那我先来说说为什么要有闭包这么个概念吧，它产生的意义是什么呢？

（1）一个概念：函数内部可以直接读取全局变量。

那么看代码：

var n=999;

function f1(){

alert(n);

}

f1(); // 999

（2）另外一个概念：在函数外部自然无法读取函数内的局部变量

那么再看代码：

function f1(){

var n=999;

}

alert(n); //输出错误

（3）这里有一个地方需要注意，函数内部声明变量的时候，一定要使用var命令。如果不用的话，你实际上声明了一个全局变量！（我们以前也提到过的！）

function f1(){

n=999;

}

f1();

alert(n); // 999

下面关键的来了！:那就是如何从外部读取局部变量呢？

那就是在函数的内部，再定义一个函数。(也就是闭包！！)

function f1(){

var n=999;

function f2(){

alert(n); // 999

}

}

在上面的代码中，函数f2就被包括在函数f1内部，这时f1内部的所有局部变量，对f2都是可见的。但是反过来就不行，f2内部的局部变量，对f1就是不可见的。既然f2可以读取f1中的局部变量，那么只要把f2作为返回值，我们不就可以在f1外部读取它的内部变量了吗！来看代码：

function f1(){

var n=999;

function f2(){

alert(n);

}

return f2;

}

var result = f1();

result(); //999

这段代码与上面的不同点就是把f2函数作为了一个返回值，然后在调用它。这时你肯定在想最后两行什么意思啊？其实开始我也没闹明白，经过高人指点，其实这最后两行的意思就是要调用f2这个函数的返回值。这两行如果我改写一下是不是更容易明白了呢？

var result = f1();

result();

合并成为：f1()();其实结果是一样的

也可以更好的说明f2这个闭包的作用是：通过把它作为返回值（因为它能访问函数f1内的局部变量），然后从全局环境中调用这个返回值，这样自然就达到了我们的目的---从全局作用域中读取局部函数内的变量！

2.既然知道了闭包的意义，下面就来了解下闭包的用途吧！

（1）闭包的第一个用途，其实上面已经提到了，就是产生它意义：可以读取函数内部的变量

（2）闭包的第二个用途，那就是：可以让这些变量的值始终保持在内存中

第二个用途怎么理解呢？来看代码：

function f1(){

var n=999;

nAdd=function(){

n+=1

}

function f2(){

alert(n);

}

return f2;

}

var result=f1();

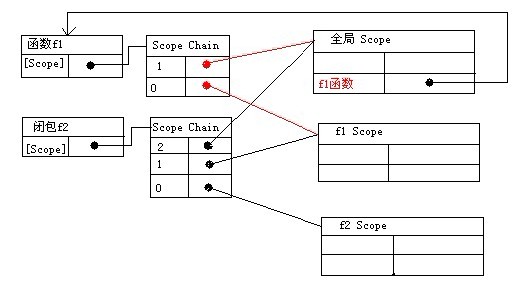
result(); // 999 输出这个f2的返回值

nAdd(); //调用nAdd函数

result(); // 1000

这里就是闭包的第二个用途：f2这个闭包会让变量n的值始终保存在内存中

光靠代码来理解第二种用途，好像没有什么说服力，下面就用画图的方法来让大家更深刻的理解！！

[](http://photo.blog.sina.com.cn/showpic.html#blogid=68cb8c790100yu61&url=http://s2.sinaimg.cn/orignal/68cb8c79gb7de7cda55b1)

第二种用途其实就和作用域链产生联系了,我来解释下：

闭包f2从f1函数中被返回后，它的作用域链被初始化为包含f1函数的活动对象和全局变量对象（黑线部分）。这样f2就可以访问在f1()函数中定义的所有变量。更为重要的是就算f1()被执行完毕后，它的活动对象也不会被销毁，因为如图f2这个闭包还在引用f1函数的活动对象，这也就是为什么上述第二种用途的原因：闭包会让变量始终保存在内存中，直到闭包被摧毁。

# 作用域和作用域链

作用域是javascript最关键的部分之一，想在javascript上更上一层，这块知识必不可少，必须弄懂。

作用域:

下面就通俗的说说什么是作用域：作用域就是变量与函数的可访问范围。在javascript中分为全局作用域和局部作用域。

全局作用域：在代码中任何地方都能访问到的对象拥有全局作用域。在Web浏览器中，他指的就是window对象，该环境直到应用程序退出才被摧毁。

局部作用局：和全局作用域相反，局部作用域一般只在固定的代码片段内可访问到（最常见的例如函数内部，所以在一些地方也会看到有人把这种作用域称为函数作用域），该环境中的代码被执行完毕后，该环境被摧毁。

## （一）全局作用域有3种情况：

1.**最外层函数和在最外层函数外面定义的变量拥有全局作用域**

（举一个我想了很久都没明白的例子，不过现在明白了）

var nameOne = "1"; //它就是最外层函数外面定义的变量

function doSomething (){ //它就是最外层函数

var nameTwo = "2";

function innerThing (){

alert(nameTwo);

}

innerThing();

}

alert(nameOne); //1

alert(nameTwo); //出现错误 因为nameTwo是在doSomething()里定义的局部变量，这里不能使用

doSomething(); //2

innerThing(); //出现错误 因为innerThing()是在doSomething()里定义的局部函数，这里不能使用

看上面这一段很简单的代码，nameOne就是最外层函数外面定义的变量，doSomething()是最外层函数，他们就是上面标题中所说的，他们拥有全局作用域，在任何地方都可以访问到他们！！

再看后面我给出了解释，也可以很明确的就知道为什么会出现错误了。

2.**所有未定义直接赋值的变量自动声明为拥有全局作用域**

看我举的一个简单的例子：

function doSomething (){

var testOne = "1";

testTwo = "2";

}

doSomething();

alert(testOne);//输出错误 如图：

alert(testTwo);//输出"2"

这段代码先声明doSomething函数，里面包含了两个变量：testOne和testTwo，然后调用doSomething函数，输出testOne和testTwo，alert(testOne)之所以会出现错误是因为var操作符使他成为了局部变量。alert(testTwo)而这里输出了正确的结果"2",是因为：所有末定义直接赋值的变量自动声明为拥有全局作用域。

3.**所有window对象的属性拥有全局作用域**

一般情况下，window对象的内置属性都都拥有全局作用域，例如window.name、window.location、window.top等等。

## （二）局部作用域

有了上面的概念和例子应该已经可以很好的理解局部作用域了，如果还想更清楚的了解下，我把上面的例子拆分出来，如下：

function doSomething (){

var testOne = "1";

function anotherThing(){

alert(testOne);

}

anotherThing();

}

doSomething();//输出"1"，因为doSomething是全局函数

alert(testOne);//错误，因为testOne是被定义到doSomething函数中的局部变量，外部不可被访问

anotherThing();//错误，anotherThing函数是doSomething函数中的局部函数，外部不可被访问

## 作用域链:

有了作用域的基础，学作用域链就会游刃有余。

函数对象有一个内部属性是[[Scope]]，该内部属性包含了函数被创建的作用域中对象的集合，这个集合被称为函数的作用域链（Scope chain）。

当一个函数创建后，它的作用域链会被创建此函数的作用域中可访问的数据对象所填充。例如：

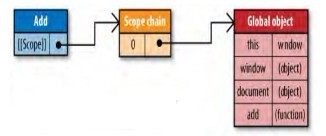
function add (num1,num2){

var sum = num1 + num2;

return sum;

}

当函数add()创建后，它的作用域链中填入了一个单独的对象，这个对象表示所有范围定义的变量。该全局对象包括诸如window,navigator和document等等。如图说明了他们的关系：

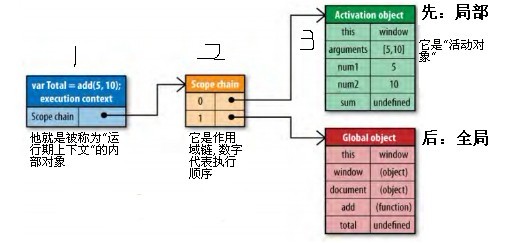
[](http://photo.blog.sina.com.cn/showpic.html#blogid=68cb8c790100ysmj&url=http://s7.sinaimg.cn/orignal/68cb8c79gb7c99a697f16)

函数add的作用域会在被执行时用到：

var total = add (5,10);

执行这个函数时会创建一个称为"运行期上下文(execution context)"的内部对象。这个对象相当重要，下面会反复提到他。一个运行期上下文定义了一个函数值执行时的环境，他是独一无二的，当函数运行完毕，它就会被摧毁。

如果对运行期上下文这个对象还是不明白看下面这个图：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=68cb8c790100ysmj&url=http://s15.sinaimg.cn/orignal/68cb8c79g78c76533f64e)

下面说下图中运行的过程：（对应图中的1，2，3）

1.当运行期被创建时，它的作用域链初始化为当前运行函数的[[Scope]]属性中所包含的对象。

2.这些值按照他们出现在函数中的顺序，被复制到作用域链中。

3.上面2过程一旦完成，一个被称为"活动对象"的新对象就会被创建，他被推入到作用域链的前端。

4. 当1（运行期上下文）被摧毁，3（活动对象）也随之被摧毁。

为了让自己更加明白作用域链，我又从网上找了几个例子，和大家一起分享，学习：

第一个例子：

var num = 1;

function total(num){

alert(num);

var num = 2;

}

total(10);

这个例子很简单，可能不用了解作用域链都可以很清楚的知道它的结果是10，因为函数total有个形参num,调用他的时候传给他一个实参10，在输出的时候结果就是10。

那么再来看第二个例子：（先别看答案，猜猜结果）

var num = 1;

function total(){

alert(num);

var num = 2;

}

var num = 10;

total();

我就做错了，我想这题肯定是迷惑我让我以为是10，我想的答案是2，我是这么想的：现在没有参数了，直接调用total函数，它会在它当前作用域中搜索num这个变量的值，结果自然为2，可是我忽略了var num = 2;是在alert(num);之后才声明的，所以最终结果是undefined。

这个例子等价于：

var num = 1;

function total(){

var num;//默认为undefined

alert(num);

num = 2;

}

var num = 10;

total();

改变作用域链：可以通过with语句和try-catch语句中得catch子句，但并不建议使用他们，因为改变了作用域链，代码的效率会大大降低，我会在另外一遍笔记中写关于高性能javascript(数据存储)的效率问题。

# this,call,apply

对于我这种初学者来说this指的到底是哪里，实在头疼,再有了前面关于变量和作用域的概念后，今天我觉定学习关于this的用法，把这个烦人的家伙解决掉！

(一)关于this

首先关于this我想说一句话，这句话记住了this的用法你也就差不多都能明白了：this指的是当前函数的对象。这句话可能比较绕，我会举出很多例子和这句话呼应的！（看下文）

1.首先看下面这段代码, 定义一个函数，弹下this,

function doSomething(){

alert(this);

}

doSomething();

我是在firefox里面调试的，所以返回的结果是[Object Window]。

那么这个 [Object Window]， 到底是什么呢？看看下面的代码

function doSomething(){

alert(this===window);

}

doSomething();

看到弹出了true， 也就是说在这种正常情况下,this其实就是window

所以,我们定义了一个全局的变量的时候，可以

var test="Tony";

也可以

window["test"]="Tony";

下面这个例子，进一步说明，正常情况下,函数里的this就是window

var test="Tony";

function doSomething(){

alert(this.test); //弹出 "Tony";

alert(window.test); //弹出 "Tony";

}

doSomething();

第一次呼应：通过以上几个例子，我想可以说这几个例子中得this指的就是他当前函数doSomething()的Window对象。那顾名思义this.test自然就等于window.test了。事实也是如此。

2.我再举一个闭包的例子，但是在这里我不会解释什么是闭包，只是讲this的用法，明天我会学习闭包，今天就先当他是一个未知数，来学习this的一个工具。

看这个例子：

var name ="The Window"; //创建了一个全局变量name

var object = { //又创建了一个对象（创建对象的方法有很多种）

name:"My Object", //创建了一个name属性（属性就是引用非函数）

doSomething:function(){ //创建了一个doSomething方法（方法就是引用了函数）

return function(){ //这个doSomething方法返回一个匿名函数

return this.name; //这个匿名函数又返回this.name

};

}

};

//由于这个doSomething方法返回的一个函数，所以这样就可以调用这个函数了

alert(object.doSomething()());

这段代码的主要解释我都注在了后面，主要想说的就是：这段代码返回的结果是"The Window",而不是我所预期的"My Object",前面不是说了吗？ this指的是当前函数的对象, 可是在这里为什么就不是呢？其实就是因为闭包的特性，由于这个匿名函数构成了一个闭包，所以他所保存的就是整个变量对象也就是Window对象。这里先不说为什么，明天学闭包的时候我还会用这个例子的，这里就先接受下来。

那么如何做才能让结果返回"My Object"呢？还是那句话this指的是当前函数的对象，那么问题就简单了，也就是让this放在doSomething方法里，而不是那个匿名函数里不就可以了吗？修改代码如下：

var name ="The Window";

var object = {

name:"My Object",

doSomething:function(){

var abc = this ;

return function(){

return abc.name;

};

}

};

alert(object.doSomething()());

第二次呼应：现在再看代码红色部分做的修改，this是不是指的是当前doSomething（）函数的object对象了？那再输出结果自然就是"My Object"了。

通过这两次呼应，是不是感觉已经对this有了一定的了解呢，其实this用到的地方还很多，以后还会慢慢讲到，比如在jQuery中this依然指的是当前函数的对象。

## （二）关于call

1.如果我们调用函数的时候，想让函数里的 this 代表别的对象，而不是当前函数的对象。有什么办法呢？那就只好使用call和apply了（对于我这种初学者，这两个东西太恶心了）

通常我们调用函数，都是直接采用以下形式调用

[调用] 函数名(); //doSomething();

其实我们还可以采用其它方法, 比如 call 这个单词，就是调用的意思，我们这里要用到它。

还是以doSomething函数为例:

中文:函数名.调用();

英文:doSomething.call();

代码如下：

var test="Tony";

function doSomething(){

alert(this.test); //弹出 "Tony";

}

doSomething(); // [调用] doSomething();

doSomething.call(); // 函数.调用()

现在我们只是做了函数的调用，还没有改变this那么怎么改变 this 呢？这个时候call就要发挥威力了。

本来doSomething函数里的this表示的是 window ,我们执行下面这句。

doSomething.call(某个对象);

这个时候doSomething里的this就变成了"某个对象",看下面的例子：

var test="Tony";

var myobj={

test : "Tom"

};

function doSomething(){

alert(this.test);

}

doSomething.call(); // 弹出 window.test，即 "Tony"

doSomething.call(myobj); // 这个时候，doSomething函数里的this指向 myobj，

//所以弹出的是 myobj.test 即 "Tom"

2.那如果里面有参数呢？看代码

var test="Tony";

var myobj={

test : "Tom"

};

function doSomething(name, age){

alert(this.test + ":" + name + age);

}

doSomething("Tony",23);//普通调用 结果Tony:Tony23

doSomething.call(myobj,"Tony",23);//call调用结果Tom:Tony23

上面代码，我们可以看出，相比普通调用, call 只是多了一个用来指定this的参数,call 干什么用的？就是干这个用的!!现在有体会了吧？

## （三）关于apply

都知道call的用法了,apply其实是一个意思。只不过是调用参数的形式不一样罢了。

call是一个一个调用参数，而apply是调用一个数组。还是用上面相同的例子：

var test="Tony";

var myobj={

test : "Tom"

};

function doSomething(name, age){

alert(this.test + ":" + name + age);

}

doSomething("Tony",23);//普通调用 结果Tony:Tony23

doSomething.call(myobj,"Tony",23);//call调用结果Tom:Tony23

doSomething.apply(object,["Tony",23]);//apply调用 结果与call相同

# 创建对象

## 传统的方法

创建对象最简单的方法，就是创建一个Object对象，然后为它添加属性和方法。

如果一个对象成员引用了一个函数，该成员就被称为一个"方法"，如果一个对象成员引用了一个非函数，该成员就被称为"属性"。

看下面这个例子：

var person = new Object();

person.name = "Tony";

person.sayName = function(){ //创建了书一个方法（因为引用了函数）

//this前面讲过，指的是当前函数的对象，在这里指的就是person对象

alert(this.name);

};

person.sayName(); //输出结果显而易见 Tony

缺点：使用同一个接口创建很多相似对象，会产生大量的重复代码。（为了解决这个问题---工厂模式出现了）

## 工厂模式

用函数来封装用特定接口创建对象的细节

比如下面这个例子：

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("Tony",23,"worker");

var person2 = createPerson("Tom",18,"student");

person1.sayName(); //Tony

person2.sayName(); //Tom

函数createPerson()能够根据接受的三个参数来创建一个包含所有信息的Person对象。

优点：解决了传统方法里创建多个相似对象的问题。

缺点：没有解决对象识别的问题。（可以靠构造函数模式来解决）

## 构造函数模式

可以创造特定类型的对象

function Person(name,age,job){

this.name = name;

this.age = age;

this.job = job;

this.sayName = function(){

alert(this.name);

};

}

var person1 =new Person("Tony",23,"worker");

var person2 =new Person("Tom",18,"student");

person1.sayName(); //Tony

person2.sayName(); //Tom

与工厂模式的不同之处：

（1）没有显示地创建构造函数

（2）直接将属性和方法赋给了this对象

（3）没有return语句

优点：动态的传递参数

缺点：每创建一个对象就会创建相同的方法函数对象，占用大量内存。（可以用原型模式来解决）

alert(person1.sayName == person2.sayName) //false

由上代码可以看出来，这两个Function实例并不相等，然而创建两个完成同样任务的Function实例没有必要。

## 原型模式

将对象属性及方法定义都写在对象的prototype里面

function Person(){

} //初始对象体内可以不定义任何东西

Person.prototype.name = "Tony";

Person.prototype.age = 22;

Person.prototype.job = "worker";

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

person1.sayName();//Tony

var person2 = new Person();

person2.sayName();//Tony 这里就是原型模式的缺点：无法动态传递构造参数

//正好与上文中的构造函数相反，这里是相同的Function对象

alert(person1.sayName == person2.sayName); //true

优点：每次创建对象时都是使用相同prototype中的方法，不会占用重复的内存

缺点：无法动态传递构造参数（构造函数模式可以解决）

## 组合构造函数模式和原型模式

function Person(name,age,job){ //构造函数模式

this.name = name;

this.age = age;

this.job =job;

this.friends = ["S","C"];

}

Person.prototype = { //原型模式

**constructor : Person**,

sayName:function(){

alert(this.name);

}

}

var person1 = new Person("Tony",22,"worker");

var person2 = new Person("Tom",18,"student");

//person1里面添加了V,并不影响person2,因为他们引用的是不同数组

person1.friends.push("V");

alert(person1.friends); //s,c,v

alert(person2.friends); //s,c

alert(person1.friends === person2.friends); //false

alert(person1.sayName === person2.sayName); //true

优点：综合了constructor及prototype方式的优点，可以动态传递构造参数，并且方法函数对象也只创建一个。

缺点：函数写在对象外面，不太符合面向对象的思想（可以用动态原型的方式来解决）

## 动态原型模式

加一个判断属性，来判断该对象是否已经被创建过，如果被创建过，则方法就不在构建

function Person(name,age,job){

//属性

this.name = name;

this.age = age;

this.job = job;

//方法

//只有在sayName方法不存时，才会添加到原型中

if(typeof this.sayName !="function"){

Person.prototype.sayName = function(){

alert(this.name);

};

}

}

var person = new Person("Tony",22,"worker");

person.sayName();

优点：将原型函数写在对象定义里面

缺点：不支持继承

# 原型,原型链

## 原型

JavaScript中的对象是基于原型的，一旦你创建一个内置对象（比如Object和Array）的实例，它们就会**自动拥有一个Object实例作为原型**。因此对象可以有两种成员类型：实例成员和原型成员。

实例成员：存在于对象实例中。

原型成员：由对象原型继承而来。

下面来举两个例子说明什么是实例成员和原型成员。

var book = {

title : "Hi",

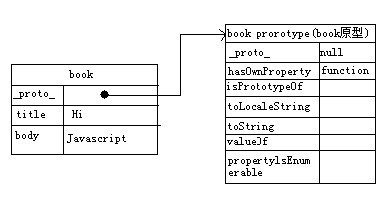
body :"Javascript"

};

alert (book.toString());// "[Object Object]"

这段代码中,对象book有两个实例成员（"Own成员"）：title和body。注意其中并没有toString()方法，但这里却返回了[Object Object]，这个toString方法是由对象book继承而来的原型成员。

下面通过图来解释下，让你有更深刻的印象：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=68cb8c790100yxan&url=http://s14.sinaimg.cn/orignal/68cb8c79gb7f76f56d42d)

当对book.toString调用时，会从对象实例开始，搜索名为"toString"的成员，一旦book没有名为toString的成员，那么会继续搜索其原型对象，直到toString（）方法被找到并执行。

可以用hasOwnProperty（）方法来判断对象是否包含特定的实例成员。

要确定对象是否包含特定属性，可以使用in操作符。还使用上面的例子，例如：

var book = {

title : "Hi",

body :"Javascript"

};

//true book中包含title这个实例成员

alert (book.hasOwnProperty("title"));

//false book中不包含toString，因为它是原型成员。

alert (book.hasOwnProperty("toString"));

//in即搜索实例，也搜索原型

alert ("title" in book); //true

alert ("toString" in book); //true

## 原型链

可是定义并用函数来创建另一种类型的原型：

//函数Book用来创建一个新的Book实例。

function Book (title,body){

//给Book添加title属性

this.title = title;

//给Book添加body属性

this.body = body;

}

//实例Book的原型添加sayTitle方法,Book.prototype的原型是Object

Book.prototype.sayTitle = function(){

alert (this.title);

};

//实例book1的原型是Book.prototype

var book1 = new Book("A","AA");

//实例book2的原型是Book.prototype

var book2 = new Book("B","BB");

alert(book1 instanceof Book);

alert(book1 instanceof Object);

alert(Object.prototype.isPrototypeOf(book1));

alert(Book.prototype.isPrototypeOf(book1));

book1.sayTitle(); //A

//首先查找book1中有没有toString()方法，

//然后向上查找Book.prototype,

//最后在Object中找到了toString()方法。

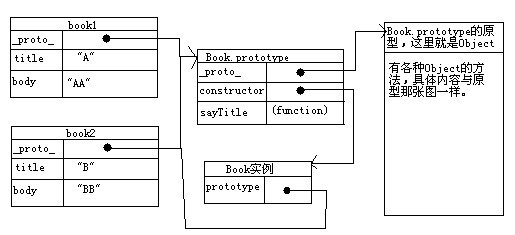
alert(book1.toString()); //[Object,Object]

我们可以用过2种方法确定原型和实例之间的关系。

1. 只要用instance来测试实例与原型链中出现过的构 造函数，都会返回true。
2. 用isPrototypeOf（）方法，只要是在原型链中出现过的原型，都可以说是该原型链所派生的实例 的原型，都会返回true。

在这个例子中，book1和book2的原型是Book.prototype,而prototype的原型是Object。这个过程就创建了一个原型链。book1和book2继承了他们成员。这也就是初级的继承的概念。

下面来看图，深刻的理解下原型链，并且对照代码，在巩固一遍：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=68cb8c790100yxan&url=http://s16.sinaimg.cn/orignal/68cb8c79gb805b706d56f)

上图中2个Book的实例book1和book2共享一个原型链。但他们自己都有各自的title和body属性，但其他部分都继承自原型。

## 谨慎的定义方法

子类有时候需要重写超类中的某个方法，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。如果不明看，来看个例子：

function ChaoLei(){

this.ageone = 1;

}

ChaoLei.prototype.getChaoleiValue = function(){

return this.ageone;

};

//创建一个子类函数

function ZiLei(){

this.agetwo = 2;

}

//子类继承了超类

ZiLei.prototype = new ChaoLei();

//为子类添加新方法getZiLeiValue方法

ZiLei.prototype.getZileiValue = function(){

return this.agetwo;

}

//重写超类中的方法

ZiLei.prototype.getChaoleiValue = function(){

return 3;

};

var instance =new ZiLei();

alert(instance.getZileiValue()); //2

alert(instance.getChaoleiValue()); //3

第一个方法getZileiValue（）被添加到了ZiLei中。

第二个方法getChaoleiValue（） 是原型链中已经存在的一个方法，这里从写，将会覆盖原来那个方法！

**必须在ChaoLei的实例替换ZiLei原型之后，再定义ZiLei的方法。**

## 重写原型链

通过原型链实现继承时，不能使用字面量创建原型方法，因为这样会重写原型链，我们还用上面的例子：

//继承了超类

ZiLei.prototype = new ChaoLei();

//添加新方法

ZiLei.prototype={

getZileiValue:function(){

return this.agetwo;

},

//重写超类中的方法

someOther:function(){

return 3;

}

}

var instance =new ZiLei();

//错误 因为原型链已经被改变！！

alert(instance.getChaoleiValue());

上面的添加新方法实际是给prototype重新赋值了，{…}会生成一个字典对象，原型被改变了。添加新方法和属性不能对prototype赋值，只能用Xxx. prototype.propName=Prop;来添加。

如：

ZiLei.prototype.someOther = function(){

return 3;

}

就只能用这个办法来重写超类中的方法了。

## 原型链的连接方式

function ClassA(){

this.a='string in classA';

}

function ClassB(){}

ClassB.prototype=new ClassA();

var b1=new ClassB();

alert(b1.a)//string in classA

ClassB.prototype={};

alert(b1.a)//string in classA

var b2=new ClassB();

alert(b2.a)//undefined

## 总结

原型就是一个对象实例，这个实例中的属性可以被拥有这个原型的对象共享。

函数对象可以通过prototype属性指定原型。

# 继承

## 原型链

原型链很强大，可以用它来实现继承，可是又存在一些问题，其中最主要的来自包含引用类型的原型。

包含引用类型值的属性会被所有实例共享；而这也正是为什么要在构造函数中，而不是原型对象中定义属性的原因。

通过原型来实现继承时，原型实际上会变成另一个类型的实例。于是，原先的实例属性也就顺理成章地变成了现在的原型属性了。来看下面的代码：

function ChaoLei(){

this.colors = ["red","black","green"];

}

function ZiLei(){

}

//继承ChaoLei

ZiLei.prototype = new ChaoLei();

var instance1 =new ZiLei();

instance1.colors.push("yellow");//red black green yellow

alert(instance1.colors);

var instance2 = new ZiLei();

alert(instance2.colors);//red black green yellow

这段代码ChaoLei构造函数（之前提过，构造函数用大写字母开头）定义了一个colors属性，该属性包含一个数组（引用类型的值）。ChaoLei的每个实例都会各自包含自己数组的colors属性。当ZiLei通过原型继承了ChaoLei之后，Zilei.prototye就成为了ChaoLei的一个实例，因此他有了自己的colors属性。结果是：ZiLei的所有实例都共享colors属性（例如：instance1和instance2），所以：对instance1进行修改，从instance2也可以反映出来。

那么如何解决这个问题呢？那就只能用下面这种办法：借用构造函数。

## 借用构造函数

在子类型构造函数的内部调用超类型构造函数。

这句话不好理解，换句话说要做到这句话的方法是：通过使用call()和apply()方法，在（将来）新创建的对象上执行构造函数，具体做法如下面这个例子：

function ChaoLei(){

this.colors = ["red","black","green"];

}

function ZiLei(){

//继承ChaoLei

ChaoLei.call(this);

}

var instance1 = new ZiLei();

instance1.colors.push("yellow");

alert(instance1.colors);

var instance2 = new ZiLei();

alert(instance2.colors);

问题：

方法都在都早函数中定义，因此函数复用就无从谈起了。

## 组合继承

用原型链实现对原型属性和方法的继承，通过借用构造函数来实现对实例属性的继承。来看例子：

function ChaoLei(name){

this.name = name;

this.colors = ["red","blue","black"];

}

ChaoLei.prototype.sayName = function(){

alert(this.name);

};

function ZiLei(name,age){

//继承属性

ChaoLei.call(this,name);

this.age = age;

}

//继承方法

ZiLei.prototype = new ChaoLei();

ZiLei.prototype.sayAge = function(){

alert(this.age);

};

var instance1 = new ZiLei("Tony",22);

instance1.colors.push("green");

alert(instance1.colors); // "red","blue","black","green"

instance1.sayName();//Tony

instance1.sayAge();//22

var instance2 = new ZiLei("Tom",30);

alert(instance2.colors);// "red","blue","black"

instance2.sayName();//Tom

instance2.sayAge();//30

这个例子有点长，我们一步一步看：

1. ChaoLei构造函数中定义了两个属性：name和colors
2. ChaoLei的原型中定义了一个方法：sayName（）
3. ZiLei构造函数中调用ChaoLei构造函数，同时传入了name参数，同时自己定义了一个age属性
4. ZiLei的原型继承ChaoLei的实例，然后又在该新原型上定义了新方法sayAge()
5. 这个ZiLei的两个不同实例(instance1和instance2)都用他们自己的属性，也可以使用相同的方法了。

# 关键字和方法

## hasOwnProperty

hasOwnProperty方法是来判定对象是否包含指定名称的属性，不会向原型链搜索。

## in

要想搜索原型链可以用in关键字（'constructor' in obj）。

# 对象

对象是属性的无序集合，每个属性存放一个原始值、对象或者函数。

## 基本类型和对象

javascript中有三种基本数据类型，字符串(string)，数值(number)，布尔值(boolean),其余全是对象。

alert(typeof 'asdf');//string

alert(typeof 22);//number

alert(typeof true);//boolean

alert(typeof new function(){});//object

基本数据类型在需要时会自动包装成对象类型，完成后再拆装为基本类型。

包装过程：

alert(typeof new String('asdf'));//object

alert(typeof new Number(22));//object

alert(typeof new Boolean(true));//object

将对象转换为基本类型则是通过这样的方式：通过调用对象的valueOf()方法来取得对象的值，如果和上下文的类型匹配，则使用该值。如果valueOf取不到值的话，则需要调用对象的toString()方法，而如果上下文为数值型，则又需要将此字符串转换为数值。

## 对象的原型\_\_proto\_\_属性

每个对象都有一个\_\_proto\_\_属性，这个属性指向另一个对象。\_\_proto\_\_属性指向的对象为拥有\_\_proto\_\_属性对象的原型。由于原型也是对象，所以它也有原型，这样就成了一个链结构叫原型链。原型链最终一个原型的\_\_proto\_\_会指向null。

var obj=new function(){};

alert(obj.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_);//null

alert(obj.\_\_proto\_\_);//[object Object]

alert(obj.\_\_proto\_\_.\_\_proto\_\_);//[object Object]

函数对象的\_\_proto\_\_属性

function func(){var a=1;return 'adf';};

alert(typeof func);//function

alert(func instanceof Function);//true

alert(func.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_);//null

alert(func.\_\_proto\_\_);//function(){}

alert(typeof func.\_\_proto\_\_);//function

alert(func.\_\_proto\_\_ instanceof Function);//false

alert(func.\_\_proto\_\_.\_\_proto\_\_);//[object Object]

函数的\_\_proto\_\_属性很奇怪，是function类型但不是Function实例。

调用对象上的一个方法，由于方法在JavaScript对象中是对另一个函数对象的引用，因此解释器会在对象中查找该属性，如果没有找到，则在其内部对象\_\_proto\_\_属性上搜索，由于\_\_proto\_\_属性与对象本身的结构是一样的，因此这个过程会一直回溯到发现该属性，则调用该属性，否则，报告一个错误。

## 对象的原型是怎么来的

\_\_proto\_\_属性是在new操作中产生的。

var p = new ClassA ();

以上创建对象过程可以用以下过程模拟：

var p = {};

ClassA.apply(p);

p.\_\_proto\_\_= ClassA.prototype;

也就是说对象的原型是其构造函数的prototype属性。这样就很好理解用原型实现继承的过程了。

构造函数的prototype是用其创建的对象的原型。

## 深入constructor属性

### constructor属性

一个对象如何知道是哪个函数构造的它呢？通过constructor属性可以知道。

// 等价于 var foo = new Array(1, 56, 34, 12);

var arr = [1, 56, 34, 12];

alert(arr.constructor === Array); // true

// 等价于 var foo = new Function();

var Foo = function() { };

alert (Foo.constructor === Function); // true

// 由构造函数实例化一个obj对象

var obj = new Foo();

alert (obj.constructor === Foo); // true

// 将上面两段代码合起来，就得到下面的结论

alert (obj.constructor.constructor === Function); // true

### constructor属性的由来

那constructor属性是怎么来的呢？要想弄清这个问题就要知道constructor属性的具体位置，是在对象里还是在原型里。

var p=new function(){};

alert(p.hasOwnProperty('constructor'));//false

alert(p.\_\_proto\_\_.hasOwnProperty('constructor'));//true

**hasOwnProperty方法是来判定对象是否包含指定名称的属性，不会向原型链搜索。要想搜索原型链可以用in关键字（'constructor' in obj）。**

由上面代码可知constructor属性是对象原型中的属性。我们再来做一个测试就更清楚了。

function Person(name) {};

alert(Person.prototype.constructor===Person);//true

函数对象的prototype最终会成为对象的原型，其constructor自然也就是对象原型的一部分，这样obj. constructor自然也是成立的。可以参见new关键字的作用。

### 不可靠的constructor

为什么说constructor不可靠，因为prototype是可改变的，在用原型模拟继承时这是经常的事。

function ClassA(){}

function ClassB(){}

ClassB.prototype=new ClassA();

var obj=new ClassB();

alert(obj.constructor===ClassB);//false

alert(obj.constructor===ClassA);//true

alert(obj.\_\_proto\_\_ instanceof ClassA);//true

alert(obj.\_\_proto\_\_.hasOwnProperty('constructor'));//false

alert(obj.\_\_proto\_\_.\_\_proto\_\_.hasOwnProperty('constructor'));//true

为什么obj的constructor跑到了原型的原型中？

obj的原型是new ClassA()，是没有constructor属性的，ClassA的prototype没有改变是有constructor属性的，所以new ClassA()的对象的原型是有constructor属性的，所以constructor属性就到了原型的原型中了。

所以要想用constructor属性来判定对象类型，就要手工设置其constructor属性。

function ClassA(){}

function ClassB(){}

ClassB.prototype=new ClassA();

ClassB.prototype.constructor=ClassB;

var obj=new ClassB();

这样生成的对象就正常了。

### instanceof关键字也不可靠

假设有一条这样的语句：

obj instanceof ClassA;

在上面的语句执行过程中，虚拟机会把ClassA.prototype和obj的\_proto\_链上的节点逐个进行比较，如果找到相等的节点，则返回true，否则返回false。

function ClassA(){}

function ClassB(){}

function ClassC(){}

ClassB.prototype=new ClassA();

var obj=new ClassB();

alert(obj instanceof ClassB);//true

alert(obj instanceof ClassA);//true

ClassB.prototype={};

alert(obj instanceof ClassB);//false

由于ClassB改了prototype属性，所以instanceof也不好使了。

## typeof

typeof用以获取一个变量的类型，typeof一般只能返回如下几个结果：number,boolean,string,function,object,undefined。

## new关键字的作用

创建具备特定类型的对象可以通过使用new关键字用构造函数模式来完成。

function ClassA(){

this.name=" ClassA ";

}

var p = new ClassA ();

在这个过程中，相当于以ClassA ()为“模版”创建了一个新对象p，它具备ClassA构造器中的成员变量和构造器原型。并且ClassA将this设置为p。

过程模拟:

var p = {};

ClassA.apply(p);

p.\_\_proto\_\_= ClassA.prototype;

在调用构造函数的过程中会经历以下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指向这个新对象)
3. 执行构造函数中的代码(为这个新对象添加属性)
4. 返回新对象

接下来说说构造函数里包含return的情况

构造函数在没有指定返回值的情况下，默认会返回新对象实例。

如果指定的返回值是基本数据类型的话，仍然会返回新对象实例。

function A(){

this.x=3;

return "OK";

}

var a = new A();

alert(a instanceof A )// true;

alert("x" in a )// true

如果指定返回对象了的话，被返回的对象就成了指定的对象值。在这种情况下，this值所引用的对象就被丢弃了。

function B(){

this.x=3;

return Object("OK");

}

var b = new B();

alert("x" in b )//false

alert(b instanceof B )//false

alert(b instanceof String )//true

## 深入this关键字

函数中的this默认代表调用函数的对象，为什么说默认呢，因为可以通过call和apply设置this。

function func(){

alert(this.toString());

}

func();//[object Window]

var obj=new function(){};

obj.f=func;

obj.f();//[object Object]

输出的是什么意思？object说明this是一个对象，Window指出this的类型是什么。

在浏览器中Window类型有一个可用实例window，window对象是浏览器javascript引擎的顶层对象。任何函数调用都是类似于obj.func()形式的，只不过当obj为顶层对象时可以不用写。所以所有全局的函数和变量都是这个对象的属性。

测试：

var testVar=’ testVar通过window调用’;

function testWindow(){

alert(‘testWindow通过window调用’);

}

window. testWindow ();//testWindow通过window调用

alert(window. testVar);// testVar通过window调用

调用是成功的。

所以,我们定义了一个全局的变量或函数的时候，以下三种方式是相同的。

var test="Tony";

function f1(){}

//--------------------

window.test="Tony"

window.f1= function (){};

//------------------------------

window["test"]="Tony";

window["f1"]= function (){};

因此，func()调用者是window，输出[object Window]；obj.f()调用者是obj，输出[object Object]。正是应了”函数中的this默认代表调用函数的对象”。

为什么不用var声明的变量为全局变量，因为有个隐藏前缀window对象。

obj={

func:function(){

v1='in func no var';

var v2='in func with var';

}

}

obj.func();

alert(obj.v1);//undefined

alert(window.v1);//in func no var

alert(obj.v2);//undefined

在对象obj的func函数中声明的变量v1因为没有添加var变成了全局变量了。

为了更好的理解”函数中的this默认代表调用函数的对象”，我们看一个闭包的例子：

var name = "The Window";

var object = {

name : "My Object",

doSomething : function() {

return function() {

return this.name;

};

}

};

alert(object.doSomething()());//The Window

为什么会是The Window？object.doSomething()()可以等同于

var func= object.doSomething();

func();

可以看到返回的函数会被widow对象调用，自然其中的this也是window对象了。this.name也就成了window.name了。

## 关于call和apply

如果我们调用函数的时候，想让函数里的 this 代表别的对象，而不是当前函数的对象。有什么办法呢？那就只好使用call和apply了。

call和apply是一个意思，只不过是调用参数的形式不一样罢了。call是一个一个调用参数，而apply是调用一个数组。

call测试

var test="Tony";

var myobj={

test : "Tom"

};

function testCall(name, age){

alert(this.test + ":" + name + age);

}

testCall ("Tony",23);//普通调用 结果Tony:Tony23

testCall.call(myobj,"Tony",23);//call调用结果Tom:Tony23

testCall.apply (myobj,["Tony",23]);// apply调用结果Tom:Tony23

## 封闭

在构造函数内用闭包实现。

缺点，闭包函数不可共用，每个对象都有。

# Function

## 类型的类型

**类型是Function的实例。**

alert( String instanceof Function);//true

alert( Number instanceof Function);//true

alert( Boolean instanceof Function);//true

alert( Object instanceof Function);//true

alert( Function instanceof Function);//true

alert(typeof String);//function

alert(typeof Number);//function

alert(typeof Boolean);//function

alert(typeof Object);//function

alert(typeof Function);//function

## 类型的内容

alert( String);//function String() {[native code]}

alert( Number);//function Number() {[native code]}

alert( Boolean);//function Boolean() {[native code]}

alert( Object);//function Object() {[native code]}

alert( Function);//function Function() {[native code]}

## 类型的\_\_proto\_\_属性

alert( String.\_\_proto\_\_);//function(){}

alert( Number.\_\_proto\_\_);//function(){}

alert( Boolean.\_\_proto\_\_);//function(){}

alert( Object.\_\_proto\_\_);//function(){}

alert( Function.\_\_proto\_\_);//function(){}

alert( String.\_\_proto\_\_===Function.prototype);//true

alert( Number.\_\_proto\_\_===Function.prototype);//true

alert( Boolean.\_\_proto\_\_===Function.prototype);//true

alert( Object.\_\_proto\_\_===Function.prototype);//true

alert( Function.\_\_proto\_\_===Function.prototype);//true

类型都是由一个构造函数创建的，这个构造函数的prototype为function(){}。

alert( Function.prototype);//function(){}

再次说明类型是Function的实例。

## 类型的prototype

alert( String.prototype);//''

alert( Number.prototype);//0

alert( Boolean.prototype);//false

alert( Object.prototype);//[object Object]

alert( Function.prototype);//function(){}

类型的prototype决定了其创建对象的\_\_proto\_\_属性，可以看new关键字的作用。

类型的区别从这里开始。

alert(typeof String.prototype);//object

alert(typeof Number.prototype);//object

alert(typeof Boolean.prototype);//object

alert(typeof Object.prototype);//object

alert(typeof Function.prototype);//function

## 类型的constructor

alert(String.hasOwnProperty('constructor'));//false

alert(String.\_\_proto\_\_.hasOwnProperty('constructor'));//true

alert(String.constructor);//function Function() {[native code]}

alert(Number.constructor);//function Function() {[native code]}

alert(Boolean.constructor);//function Function() {[native code]}

alert(Object.constructor);//function Function() {[native code]}

alert(Function.constructor);//function Function() {[native code]}

类型都是用function Function() {[native code]}创建的。function Function() {[native code]}是什么呢？

alert(Function); //function Function() {[native code]}

alert(String.constructor===Function);//true

alert(Number.constructor===Function);//true

alert(Boolean.constructor===Function);//true

alert(Object.constructor===Function);//true

alert(Function.constructor===Function);//true

再次说明了类型是Function的实例。

## Function

//一个环结构

alert(Function.prototype);//function(){}

alert(Function.\_\_proto\_\_===Function.prototype)//true

alert(Function.prototype.constructor===Function)//true

//一个奇怪的对象，是函数类型，但不是由Function创建。

alert(Function.prototype instanceof Object);//true

alert(Function.prototype instanceof Function);//false

alert(typeof Function.prototype);//function

// Function？？？

alert(Function.prototype.\_\_proto\_\_)//[object Object]

alert(Function.prototype.\_\_proto\_\_===Object.prototype)//true

alert(Function.prototype.\_\_proto\_\_===Number.prototype)//false

alert(Function.prototype.\_\_proto\_\_.\_\_proto\_\_);//null

## 总结

有一个神秘对象function(){}，它没有prototype属性，所以不能创建对象；它是function类型但不是Function实例，它的\_\_proto\_\_为object对象和Object.prototype是一个对象。

Function的\_\_proto\_\_为function(){}，function(){}的constructor为Function，这是函数对象的顶层结构，因为函数对象也是由函数对象创建的，上层一定是一个环。

所有类型的\_\_proto\_\_为function(){}，而function(){}的constructor为Function，说明类型是由Function创建的。

constructor

prototype

constructor

prototype

prototype

\_\_proto\_\_

\_\_proto\_\_

prototype

\_\_proto\_\_

constructor

\_\_proto\_\_

function(){}

Function

String

Number

Boolean

Array

…

Object

object对象

Function

function(){}

constructor

prototype

Array

空

constructor

prototype

String

空

一个函数func

空

每个函数（可以new）的prototype指向不同的对象，其constructor属性又指向本身，从而对象的constructor属性指向创建它的函数。类型是Function的实例，在创建时赋给prototype不同的值。

alert(String.prototype);//空

alert(String.prototype===Object.prototype);//false

# 作用域链和闭包

## 作用域

作用域就是变量与函数的可访问范围。在javascript中分为全局作用域和局部作用域。

全局作用域：在代码中任何地方都能访问到的对象拥有全局作用域。在Web浏览器中，他指的就是window对象，该环境直到应用程序退出才被摧毁。

局部作用局：和全局作用域相反，局部作用域一般只在固定的代码片段内可访问到（最常见的例如函数内部），该环境中的代码被执行完毕后，该环境被摧毁。

## 作用域链

函数对象有一个内部属性是[[Scope]]，该内部属性包含了函数被创建的作用域中对象的集合，这个集合被称为函数的作用域链（Scope chain）。

当一个函数创建后，它的作用域链会被创建此函数的作用域中可访问的数据对象所填充。例如：

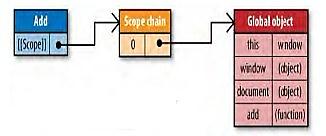
function add (num1,num2){

var sum = num1 + num2;

return sum;

}

当函数add()创建后，它的作用域链中填入了一个单独的对象，这个对象表示所有范围定义的变量。该全局对象包括诸如window,navigator和document等等。如图说明了他们的关系：

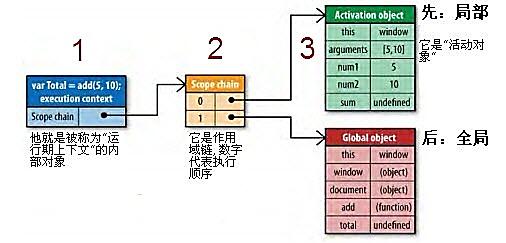
[](http://photo.blog.sina.com.cn/showpic.html#blogid=68cb8c790100ysmj&url=http://s7.sinaimg.cn/orignal/68cb8c79gb7c99a697f16)

函数add的作用域会在被执行时用到：

var total = add (5,10);

执行这个函数时会创建一个称为"运行期上下文(execution context)"的内部对象。一个运行期上下文定义了一个函数值执行时的环境，他是独一无二的，当函数运行完毕，它就会被摧毁。

如果对运行期上下文这个对象还是不明白看下面这个图：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=68cb8c790100ysmj&url=http://s15.sinaimg.cn/orignal/68cb8c79g78c76533f64e)

下面说下图中运行的过程：（对应图中的1，2，3）

1.当运行期被创建时，它的作用域链初始化为当前运行函数的[[Scope]]属性中所包含的对象。

2.这些值按照他们出现在函数中的顺序，被复制到作用域链中。

3.上面2过程一旦完成，一个被称为"活动对象"的新对象就会被创建，他被推入到作用域链的前端。

4. 当1（运行期上下文）被摧毁，3（活动对象）也随之被摧毁。

## 作用域链例子

var num = 1;

function total(){

alert(num);

var num = 2;

}

var num = 10;

total();

最终结果是undefined，因为局部作用域中有num变量，但在调用后定义，所以调用时就undefined了。

## 闭包

用简单的语句来描述JavaScript中的闭包的概念：由于JavaScript中，函数是对象，对象是属性的集合，而属性的值又可以是对象，则在函数内定义函数成为理所当然，如果在函数func内部声明函数inner，然后在函数外部调用inner，这个过程即产生了一个闭包。

闭包的效果可以简单描述为“使函数外部可以读取函数内部的局部变量”。

闭包的一切效果都基于外部调用inner函数会导致func的上下文不会被摧毁。

function f1(){

var n=999;

nAdd=function(){

n+=1

}

function f2(){

alert(n);

}

return f2;

}

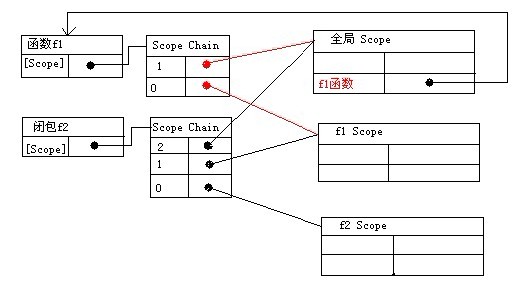
var result=f1();

result(); // 999 输出这个f2的返回值

nAdd(); //调用nAdd函数

result(); // 1000

因为f2的上下文依赖于f1的上下文所以在f1运行完后其上下文不会被摧毁，其中的值一直都存在着。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=68cb8c790100yu61&url=http://s2.sinaimg.cn/orignal/68cb8c79gb7de7cda55b1)

## 闭包的用途

### 对象的封闭

封装的效果为未经授权的客户代码无法访问到我们不公开的数据。看这个例子：

function Person(name) {

// private variable

var address = "The Earth";

// public method

this.getAddress = function() {

return address;

}

// public variable

this.name = name;

}

// public

Person.prototype.getName = function() {

return this.name;

}

// public

Person.prototype.setName = function(name) {

this.name = name;

}

每一次new Person()时都会创建一个上下文，其中局部变量address就在其中。新创建的对象中会有一个方法getAddress()可以调用上下文中的address，这就形成闭包，使上下文的生命周期和创建的对象一样。每一个Person都会关联一个上下文，这样就可以实现属性的封闭了。

### 缓存

设想我们有一个处理过程很耗时的函数对象，每次调用都会花费很长时间，那么我们就需要将计算出来的值存储起来，当调用这个函数的时候，首先在缓存中查找，如果找不到，则进行计算，然后更新缓存并返回值，如果找到了，直接返回查找到的值即可。闭包正是可以做到这一点，因为它不会释放外部的引用，从而函数内部的值可以得以保留。

var CachedSearchBox = (function() {

var cache = {}, count = [];

return {

attachSearchBox : function(dsid) {

if (dsid in cache) {// 如果结果在缓存中

return cache[dsid];// 直接返回缓存中的对象

}

var fsb = new uikit.webctrl.SearchBox(dsid);// 新建

cache[dsid] = fsb;// 更新缓存

if (count.length > 100) {// 保正缓存的大小<=100

delete cache[count.shift()];

}

return fsb;

},

clearSearchBox : function(dsid) {

if (dsid in cache) {

cache[dsid].clearSelection();

}

}

};

})();

CachedSearchBox.attachSearchBox("input1");

这样，当我们第二次调用CachedSearchBox.attachSerachBox(“input1”)的时候，我们就可以从缓存中取道该对象，而不用再去创建一个新的searchbox对象。

### 柯里化

柯里化就是预先将函数的某些参数传入，得到一个简单的函数，但是预先传入的参数被保存在闭包中，因此会有一些奇特的特性。

var adder = function(num){

return function(y){

return num + y;

}

}

var inc = adder(1);

var dec = adder(-1);

这里的inc/dec两个变量事实上是两个新的函数，可以通过括号来调用，比如下例中的用法：

//inc, dec现在是两个新的函数，作用是将传入的参数值(+/-)1

alert(inc(99));//100

alert (dec(101));//100

alert (adder(100)(2));//102

alert (adder(2)(100));//102

柯里化可以创建有状态的函数，函数的状态保存在其关联的上下文中。再看一个例子：

//update会返回一个函数，这个函数可以设置id属性为item的web元素的内容

function update(item) {

return function(text) {

$("div#" + item).html(text);

};

}

// Ajax请求，当成功是调用参数callback

function refresh(url, callback) {

var params = {

type : "echo",

data : ""

};

$.ajax({

type : "post",

url : url,

cache : false,

async : true,

dataType : "json",

data : params,

// 当异步请求成功时调用

success : function(data, status) {

callback(data);

},

// 当请求出现错误时调用

error : function(err) {

alert("error : " + err);

}

});

}

refresh("action.do?target=news", update("newsPanel"));

refresh("action.do?target=articles", update("articlePanel"));

refresh("action.do?target=pictures", update("picturePanel"));