[**一篇相当不错的function入门文章**](http://www.cnblogs.com/jenry/archive/2008/02/15/1069519.html)

**词语翻译列表:**

function:函数（Function未翻译）  
declare:定义  
assign:指派，分配  
functionbody:函数体（就是函数的内容）  
object:对象  
property:属性  
unnamed:匿名（在这里没翻译成未命名）  
object oriented programming:面相对相编程  
class:类（比如后面的class data type我翻译成类数据类型）  
pointer:指针  
reassign:重新分配  
nest:嵌套  
feature:功能，特性  
local/global:局部/全局  
blueprint:蓝图（？）  
user defined:用户自定义  
instance:实例  
prototype:原型（除了标题都不翻译）  
internal:内部  
constructor:构造器  
duplication:

**函数：定义**

有以下这些方法可以定义一个函数。所有这些都是有效的，但是它们在后台如何实现的则有一些差别。

***常用的写法***

一般大家都用这个写法来定义一个函数：

**CODE:**functionName([parameters]){functionBody};

**Example D1：**

**CODE:**  
function add(a, b)   
{                       
  return a+b;  
}                       
alert(add(1,2));        // 结果 3

当我们这么定义函数的时候，函数内容会被编译（但不会立即执行，除非我们去调用它）。而且，也许你不知道，当这个函数创建的时候有一个同名的对象也被创建。就我们的例子来说，我们现在有一个对象叫做“add”（要更深入了解，看底下函数：对象节。）

***匿名函数***

我们也可以通过指派一个变量名给匿名函数的方式来定义它。

**Example D2**

**CODE:**  
var add=function(a, b)   
{                       
  return a+b;  
}                       
alert(add(1,2));        // 结果 3

这个代码和前一个例子做了同样的事情。也许语法看起来比较奇怪，但它应该更能让你感觉到函数是一个对象，而且我们只是为这个对指派了一个名称。可以把它看做和 var myVar=[1,2,3]一样的语句。以这种方式声明的函数内容也一样会被编译。

当我们指派一个这样的函数的时候，我们并不一定要求必须是匿名函数。在这里，我作了和ExampleD2一样的事情，但我加了函数名“theAdd”，而且我可以通过调用函数名或者是那个变量来引用函数。

**Example D2A**

**CODE:**  
var add=function theAdd(a, b)   
{                       
  return a+b;  
}                       
alert(add(1,2));           // 结果 3  
alert(theAdd(1,2));        // 结果也是 3

使用这种方式来定义函数在面向对象编程中是很有用的，因为我们能像底下这样使一个函数成为一个对象的属性。

**CODE:**  
var myObject=new Object();  
myObject.add=function(a,b){return a+b};    
// myObject 现在有一个叫做“add”的属性（或方法）  
// 而且我能够象下面这样使用它  
myObject.add(1, 2);

我们也能够通过使用运算符new来定义一个函数。这是一个最少见的定义函数的方式并且并不推荐使用这种方式除非有特殊的理由（可能的理由见下）。语法如下：

CODE:  
varName=new Function([param1Name, param2Name,...paramNName], functionBody);

**Example D3:**

**CODE:**  
var add=new Function("a", "b", "return a+b;");  
alert(add(3,4));        // 结果 7

我在这里有两个参数叫做a和b，而函数体返回a和b的和。请注意new Function(...)使用了大写F，而不是小写f。 这就告诉javascript，我们将要创建一个类型是Function的对象。 还要注意到，参数名和函数体都是作为字符串而被传递。我们可以随心所欲的增加参数，javascript知道函数体会是右括号前的最后一个字符串（如果没有参数，你能够只写函数体）。你没必要将所有东西都写在一行里（使用\或者使用字符串连接符+来分隔长代码）。\标记告诉JavaScript在下一行查找字符串的其余部分。例子如下：

**Example D4**

**CODE:**  
var add=new Function("a", "b",   
  "alert" +                      // 注意 "+"  
  "('adding '+a+' and ' +b);\    // 和 "\"的不同用法  
   return a+b;");  
alert(add(3,4));        // 结果 7

采用这种方式定义函数会导致函数并没被编译，而且它有可能会比用其它方式定义的函数要慢。至于为什么，看一下这个代码：

**Example D5**

**CODE:**

function createMyFunction(myOperator)  
{  
  return new Function("a", "b", "return a" + myOperator + "b;");  
}

var add=createMyFunction("+");                // 创建函数 "add"   
var subtract=createMyFunction("-");           // 创建函数 "subtract"   
var multiply=createMyFunction("\*");           // 创建函数 "multiply"   
// test the functions  
alert("加的结果="+add(10,2));                  // 结果是 12  
alert("减的结果="+subtract(10,2));             // 结果是 8  
alert("乘的结果="+multiply(10,2));             // 结果是 20  
alert(add);

这个有趣的例子创建了三个不同的function，通过实时传递不同的参数来创建一个新Function。因为编译器没法知道最终代码会是什么样子的，所以new Function(...)的内容不会被编译。那这有什么好处呢？嗯，举个例子，如果你需要用户能够创建他们自己的函数的时候这个功能也许很有用，比如在游戏里。我们也许需要允许用户添加“行为”给一个“player”。但是，再说一次，一般情况下，我们应该避免使用这种形式，除非有一个特殊的目的。

**函数：对象**

函数是javascript中的一种特殊形式的对象。它是第一个[b〕类数据类型（class data type）。这意味着我们能够给它增加属性。这里有一些需要注意的有趣观点：

***对象的创建***

就像刚才提及的，当我们定义一个函数时，javascript实际上在后台为你创建了一个对象。这个对象的名称就是函数名本身。这个对象的类型是function。在下面的例子，我们也许不会意识到这一点，但我们实际上已经创建了一个对象：它叫做Ball。

**Example 1**

**CODE:**  
function Ball()       // 也许看起来有点奇怪，但是这个声明  
{                     // 创建了一个叫做Ball的对象  
  i=1;  
}                       
alert(typeof Ball);     // 结果 "function"

我们甚至能将这个对象的内容打印出来而且它会输出这个函数的实际代码，Example2: 点击 alert(Ball);来看看Ball的内容。

***属性的添加***

我们能够添加给Object添加属性，包括对象function。因为定义一个函数的实质是创建一个对象。我们能够“暗地里”给函数添加属性。比如，我们这里定义了函数Ball，并添加属性callsign。

**CODE:**  
function Ball()       // 也许看起来有点奇怪，但是这个声明  
{                     // 创建了一个叫做Ball的对象，而且你能够  
}                     // 引用它或者象下面那样给它增加属性  
Ball.callsign="The Ball"; // 给Ball增加属性  
alert(Ball.callsign); // 输出 "The Ball"

***指针***

因为function是一个对象，我们能够为一个function分配一个指针。如下例，变量ptr指向了对象myFunction。

**CODE:**  
function myFunction(message)   
{   
  alert(message);  
}  
var ptr=myFunction;  // ptr指向了myFunction  
ptr("hello");         // 这句会执行myFunction：输出"hello"

我们能够运行这个函数，就好像这个函数名已经被指针名代替了一样。所以在上面，这行ptr("hello"); 和myFunction("hello");的意义是一样的。

指向函数的指针在面向对象编程中相当有用。例如：当我们有多个对象指向同一个函数的时候（如下）：

**Example 4A**

**CODE:**  
function sayName(name)   
{   
  alert(name);  
}  
var object1=new Object();      // 创建三个对象  
var object2=new Object();  
var object3=new Object();  
object1.sayMyName=sayName;       // 将这个函数指派给所有对象  
object2.sayMyName=sayName;  
object3.sayMyName=sayName;  
    
object1.sayMyName("object1");    // 输出 "object1"  
object2.sayMyName("object2");    // 输出 "object2"  
object3.sayMyName("object3");    // 输出 "object3"

因为只有指针被保存（而不是函数本身），当我们改变函数对象自身的时候，所有指向那个函数的指针都会发生变化。我们能够在底下看到：

**Example 5:**

**CODE:**

function myFunction()   
{   
  alert(myFunction.message);  
}  
myFunction.message="old";  
var ptr1=myFunction;                 // ptr1 指向 myFunction  
var ptr2=myFunction;                 // ptr2 也指向 myFunction

ptr1();                     // 输出 "old"  
ptr2();                              // 输出 "old"

myFunction.message="new";

ptr1();                     // 输出 "new"  
ptr2();                              // 输出 "new"

***指针的指向***

我们能够在一个函数创建之后重新分配它，但是我们需要指向函数对象本身，而不是指向它的指针。在下例中，我将改变myfunction()的内容。

**Example 6:**

**CODE:**  
function myFunction()   
{   
  alert("Old");  
}  
myFunction(); // 输出 "Old"  
myFunction=function()  
{  
  alert("New");  
};  
myFunction(); // 输出 "New"

旧函数哪里去了？？被抛弃了。

如果我们需要保留它，我们可以在改变它之前给它分配一个指针。

**Example 6A:**

**CODE:**  
function myFunction()   
{   
  alert("Old");  
}  
var savedFuncion=myFunction;  
myFunction=function()  
{  
  alert("New");  
};  
myFunction();    // 输出 "New"  
savedFuncion();  // 输出 "Old"

不过要小心，象下面这样的例子并不会有作用，因为是创建了另一个叫做myFunctionPtr的函数而不是修改它。

**Example 6B:**

**CODE:**  
function myFunction()   
{   
  alert("Old");  
}  
var savedFunc=myFunction;  
savedFunc=function()  
{  
  alert("New");  
};  
myFunction();            // 输出 "Old"  
savedFunc();             // 输出 "New"

***内嵌函数***

我们还能够在一个函数中嵌套一个函数。下例，我有一个叫做getHalfOf的函数,而在它里面，我有另一个叫做calculate的函数。

**Example 7**

**CODE:**

function getHalfOf(num1, num2, num3)       
{   
  function calculate(number)  
  {  
    return number/2;  
  }

  var result="";  
  result+=calculate(num1)+" ";  
  result+=calculate(num2)+" ";  
  result+=calculate(num3);  
}           
var resultString=getHalfOf(10,20,30);  
alert(resultString);         // 输出 "5 10 15"

你只能在内部调用嵌套的函数。就是说，你不能这么调用：getHalfOf.calculate(10)，因为calculate只有当外部函数(getHalfOf())在运行的时候才会存在。这和我们前面的讨论一致（函数会被编译，但只有当你去调用它的时候才会执行）。

***调用哪个函数？***

你也许正在想命名冲突的问题。比如，下面哪一个叫做calculate的函数会被调用？

**Example 8**

**CODE:**

function calculate(number)  
{  
  return number/3;  
}

function getHalfOf(num1, num2, num3)       
{   
  function calculate(number)  
  {  
    return number/2;  
  }

  var result="";  
  result+=calculate(num1)+" ";  
  result+=calculate(num2)+" ";  
  result+=calculate(num3);  
}           
var resultString=getHalfOf(10,20,30);  
alert(resultString);         // 输出 "5 10 15"

在这个例子中，编译器会首先搜索局部内存地址，所以它会使用内嵌的calculate函数。如果我们删除了这个内嵌（局部）的calculate函数,这个代码会使用全局的calculate函数。

**函数：数据类型及构造函数**

让我们来看看函数的另一个特殊功能－－这让它和其它对象类型截然不同。一个函数能够用来作为一个数据类型的蓝图。这个特性通常被用在面向对象编程中来模拟用户自定义数据类型(user defined data type)。使用用户自定义数据类型创建的对象通常被成为用户自定义对象(user defined object)。

***数据类型***

在定义了一个函数之后，我们也同时创建了一个新的数据类型。这个数据类型能够用来创建一个新对象。下例，我创建了一个叫做Ball的新数据类型。

**Example DT1**

**CODE:**

function Ball()  
{  
}  
var ball0=new Ball(); // ball0 现在指向一个新对象

alert(ball0);         // 输出 "Object"，因为 ball0 现在是一个对象

这样看来，ball0=new Ball()作了什么？new关键字创建了一个类型是Object的新对象（叫做ball0）。然后它会执行Ball()，并将这个引用传给ball0（用于调用对象）。下面，你会看到这条消息：“creating new Ball”，如果Ball()实际上被运行的话。

**Example DT2**

**CODE:**  
function Ball(message)  
{  
  alert(message);  
}  
var ball0=new Ball("creating new Ball");  // 创建对象并输出消息  
ball0.name="ball-0";                      // ball0现在有一个属性：name  
alert(ball0.name);                        // 输出 "ball-0"

我们可以把上面这段代码的第6行看做是底下的代码6-8行的一个简写：

**CODE:**function Ball(message)  
{  
  alert(message);  
}  
var ball0=new Object();  
ball0.construct=Ball;  
ball0.construct("creating new ball");  // 执行 ball0.Ball("creating..");  
ball0.name="ball-0";                        
alert(ball0.name);

这行代码ball0.construct=Ball和Example 4中的ptr=myFunction语法一致。

如果你还是不明白这行的含义那就回过头再复习一下Example 4。注意：你也许考虑直接运行ball0.Ball("...")，但是它不会起作用的，因为ball0并没有一个叫做Ball("...")的属性，并且它也不知道你究竟想作些什么。

***添加属性***

当我们象上面那样使用关键字new创建一个对象的时候，一个新的Object被创建了。我们可以在创建之后给这个对象添加属性（就好像我在上面那样添加属性name。而接下来的问题就是如果我们创建了这个对象的另外一个实例，我们得象下面那样再次给这个新对象添加这个属性。）

**Example DT3 (creates 3 ball objects)**

**CODE:**

function Ball()  
{  
}  
var ball0=new Ball(); // ball0 现在指向了类型Ball的一个新实例  
ball0.name="ball-0";  // ball0 现在有一个属性"name"

var ball1=new Ball();  
ball1.name="ball-1";

var ball2=new Ball();

alert(ball0.name);    // 输出 "ball-0"  
alert(ball1.name);    // 输出 "ball-1"  
alert(ball2.name);    // 哦，我忘记给ball2添加“name”了！

我忘记给ball2添加属性name了，如果在正式的程序中这也许会引发问题。有什么好办法可以自动增加属性呢？嗯，有一个：使用this关键字。this这个词在function中有特别的意义。它指向了调用函数的那个对象。让我们看看下面的另一个示例，这时候我们在构造函数中添加上这些属性：

**Example DT4**

CODE:  
function Ball(message, specifiedName)  
{  
  alert(message);  
  this.name=specifiedName;                  
}  
var ball0=new Ball("creating new Ball", "Soccer Ball");    
alert(ball0.name);                   // prints "Soccer Ball"

请记住：是new关键字最终使得构造函数被执行。在这个例子中，它将会运行Ball("creating new Ball", "Soccer Ball");而关键字this将指向ball0。  
因此，这行：this.name=specifiedName变成了ball0.name="Soccer Ball"。  
它主要是说：给ball0添加属性name，属性值是Soccer Ball。   
我们现在只是添加了一个name属性给ball0，看起来和上一个例子中所做的很象，但却是一个更好更具扩展性的方法。现在，我们可以随心所欲的创建许多带有属性的ball而无需我们手动添加它们。而且，人们也希望创建的Ball对象能够清晰的看懂它的构造函数并且能够轻松找出Ball的所有属性。让我们添加更多属性到Ball里。

**Example DT5**

**CODE:**

function Ball(color, specifiedName, owner, weight)  
{  
  this.name=specifiedName;                  
  this.color=color;  
  this.owner=owner;  
  this.weight=weigth;  
}  
var ball0=new Ball("black/white", "Soccer Ball", "John", 20);    
var ball1=new Ball("gray", "Bowling Ball", "John", 30);    
var ball2=new Ball("yellow", "Golf Ball", "John", 55);    
var balloon=new Ball("red", "Balloon", "Pete", 10);

alert(ball0.name);                        // 输出 "Soccer Ball"  
alert(balloon.name);                      // 输出 "Balloon"  
alert(ball2.weight);                      // 输出 "55"

嘿！使用面向对象术语，你能够说Ball是一个拥有如下属性的对象类型：name, color, owner, weight。

***将对象赋给属性***

我们并没被限制只能添加形如字符串或者数字之类的简单数据类型作为属性。我们也能够将对象赋给属性。下面，supervisor是Employee的一个属性.

**Example DT6**

**CODE:**

function Employee(name, salary, mySupervisor)  
{  
  this.name=name;                  
  this.salary=salary;  
  this.supervisor=mySupervisor;  
}  
var boss=new Employee("John", 200);

var manager=new Employee("Joan", 50, boss);    
var teamLeader=new Employee("Rose", 50, boss);

alert(manager.supervisor.name+" is the supervisor of "+manager.name);  
alert(manager.name+"\'s supervisor is "+manager.supervisor.name);

会输出什么呢？

就像你在上面这个例子中看到的那样，manager和teamLeader都有一个supervisor属性，而这个属性是类型Employee的一个对象。

***将函数作为属性***

任何类型的对象都可以作为一个属性，回忆一下前面的Example 4(不是Example DT4)，函数也是一个对象。所以你可以让一个函数作为一个对象的一个属性。下面，我将添加两个函数getSalary和addSalary。

**Example DT7**

**CODE:**

function Employee(name, salary)  
{  
  this.name=name;                  
  this.salary=salary;

  this.addSalary=addSalaryFunction;

  this.getSalary=function()  
                 {  
                   return this.salary;  
                 };  
}  
function addSalaryFunction(addition)  
{  
  this.salary=this.salary+addition;  
}

var boss=new Employee("John", 200000);  
boss.addSalary(10000);                    // boss 长了 10K 工资……为什么老板工资可以长这么多:'(  
alert(boss.getSalary());                  // 输出 210K……为什么默认工资也那么高……:'(

addSalary和getSalary演示了几种将函数赋给属性的不同方法。如果你记得我们最开始的讨论；我讨论了三种声明函数的不同方式。所有那些在这里都是适用的，但是上面展示的两个最常用。

让我们看看有什么不同。下面，注意一下9-12行的代码。当这部分代码执行的时候，函数getSalary被声明。如前面数次提到的，一个函数声明的结果是一个对象被创建。所以这时候boss被创建（接下来的第19行），而boss里有一个getSalary属性。

**CODE:**

function Employee(name, salary)  
{  
  this.name=name;                  
  this.salary=salary;

  this.addSalary=addSalaryFunction;

  this.getSalary=function()  
                 {  
                   return this.salary;  
                 };  
}  
function addSalaryFunction(addition)  
{  
  this.salary=this.salary+addition;  
}

var boss=new Employee("John", 200000);  
var boss2=new Employee("Joan", 200000);  
var boss3=new Employee("Kim", 200000);

当你创建这个对象的更多实例时(boss2和boss3),每一个实例都有一份getSalary代码的单独拷贝；而与此相反，addSalary则指向了同一个地方(即addSalaryFunction)。

看看下面的代码来理解一下上面所描述的内容。

**Example DT8**

**CODE:**

function Employee(name, salary)  
{  
  this.name=name;                  
  this.salary=salary;

  this.addSalary=addSalaryFunction;  
  this.getSalary=function()  
                 {  
                   return this.salary;  
                 };  
}  
function addSalaryFunction(addition)  
{  
  this.salary=this.salary+addition;  
}

var boss1=new Employee("John", 200000);  
var boss2=new Employee("Joan", 200000);

// 给getSalary函数对象添加属性  
boss1.getSalary.owner="boss1";  
boss2.getSalary.owner="boss2";  
alert(boss1.getSalary.owner);   // 输出 "boss1"  
alert(boss2.getSalary.owner);   // 输出 "boss2"  
// 如果两个对象指向同一个函数对象，那么  
// 上面两个输出都应该是“boss2”。

// 给addSalary函数对象添加属性  
boss1.addSalary.owner="boss1";  
boss1.addSalary.owner="boss2";  
alert(boss1.addSalary.owner);   // 输出 "boss2"  
alert(boss2.addSalary.owner);   // 输出 "boss2"  
// 因为两个对象都指向同一个函数，（子乌注：原文写are not pointing to the same function，疑为笔误）  
// 当修改其中一个的时候，会影响所有的实例（所以两个都输出“boss2”）.

也许不是重要的事情，但这里有一些关于运行类似上面的getSalary的内嵌函数的结论： 1) 需要更多的存储空间来存储对象（因为每一个对象实例都会有它自己的getSalary代码拷贝）；2) javascript需要更多时间来构造这个对象。

让我们重新写这个示例来让它更有效率些。

**Example DT9**

**CODE:**

function Employee(name, salary)  
{  
  this.name=name;                  
  this.salary=salary;

  this.addSalary=addSalaryFunction;  
  this.getSalary=getSalaryFunction;  
}  
function getSalaryFunction()  
{  
  return this.salary;  
}

function addSalaryFunction(addition)  
{  
  this.salary=this.salary+addition;  
}

看这儿，两个函数都指向同一个地方，这将会节约空间和缩短构造时间（特别是当你有一大堆内嵌函数在一个构造函数的时候）。这里有另外一个函数的功能能够来提升这个设计，它叫做prototype，而我们将在下一节讨论它。

**函数：原型**

每一个构造函数都有一个属性叫做原型(prototype,下面都不再翻译，使用其原文)。这个属性非常有用：为一个特定类声明通用的变量或者函数。

***prototype的定义***

你不需要显式地声明一个prototype属性，因为在每一个构造函数中都有它的存在。你可以看看下面的例子：

**Example PT1**

**CODE:**  
function Test()  
{  
}  
alert(Test.prototype);  // 输出 "Object"

***给prototype添加属性***

就如你在上面所看到的，prototype是一个对象，因此，你能够给它添加属性。你添加给prototype的属性将会成为使用这个构造函数创建的对象的通用属性。

例如，我下面有一个数据类型Fish，我想让所有的鱼都有这些属性：livesIn="water"和price=20；为了实现这个，我可以给构造函数Fish的prototype添加那些属性。

**Example PT2**

**CODE:**  
function Fish(name, color)  
{  
  this.name=name;  
  this.color=color;  
}  
Fish.prototype.livesIn="water";  
Fish.prototype.price=20;

接下来让我们作几条鱼：

**CODE:**  
var fish1=new Fish("mackarel", "gray");  
var fish2=new Fish("goldfish", "orange");  
var fish3=new Fish("salmon", "white");

再来看看鱼都有哪些属性：

**CODE:**  
for (int i=1; i<=3; i++)  
{  
  var fish=eval("fish"+i);   // 我只是取得指向这条鱼的指针  
  alert(fish.name+","+fish.color+","+fish.livesIn+","+fish.price);  
}

输出应该是：

**CODE:**  
"mackarel, gray, water, 20"  
"goldfish, orange, water, 20"  
"salmon, white  water, 20"

你看到所有的鱼都有属性livesIn和price,我们甚至都没有为每一条不同的鱼特别声明这些属性。这时因为当一个对象被创建时，这个构造函数将会把它的属性prototype赋给新对象的内部属性\_\_proto\_\_。这个\_\_proto\_\_被这个对象用来查找它的属性。

你也可以通过prototype来给所有对象添加共用的函数。这有一个好处：你不需要每次在构造一个对象的时候创建并初始化这个函数。为了解释这一点，让我们重新来看Example DT9并使用prototype来重写它：

***用prototype给对象添加函数***

**Example PT3**

**CODE:**

function Employee(name, salary)  
{  
  this.name=name;                  
  this.salary=salary;  
}  
Employee.prototype.getSalary=function getSalaryFunction()  
{  
  return this.salary;  
}

Employee.prototype.addSalary=function addSalaryFunction(addition)  
{  
  this.salary=this.salary+addition;  
}

我们可以象通常那样创建对象：

**CODE:**  
var boss1=new Employee("Joan", 200000);  
var boss2=new Employee("Kim", 100000);  
var boss3=new Employee("Sam", 150000);

并验证它：

**CODE:**  
alert(boss1.getSalary());   // 输出 200000  
alert(boss2.getSalary());   // 输出 100000  
alert(boss3.getSalary());   // 输出 150000

这里有一个图示来说明prototype是如何工作的。这个对象的每一个实例(boss1, boss2, boss3)都有一个内部属性叫做\_\_proto\_\_，这个属性指向了它的构造器(Employee)的属性prototype。当你执行getSalary或者addSalary的时候，这个对象会在它的\_\_proto\_\_找到并执行这个代码。注意这点：这里并没有代码的复制(和Example DT8的图表作一下对比)。