JavaScript 介绍（一）

JavaScript 语言

Jason

目录

JavaScript

和Java没关系

ECMAScript

European Computer Manufacturers Association

欧洲计算机制造商协会

标准 ECMA-262

JavaScript

处于安全方面的原因

客户端JavaScript不允许对文件进行读写操作

目录

变量声明

```
var i;
var sum;
var i, sum;
```

如果用**var**声明变量时，没有指定初始值，
那么它的初始值就是**undefined**

如果给一个未用**var**声明的变量赋值，
Javascript会隐式声明该变量，
隐式声明的变量总是被创建为全局变量

变量作用域

```
1 var scope = "global";
2 function checkscope() {
3     var scope = "local";
4     document.write(scope);
5 }
6 checkscope();
```

1. 声明一个全局变量
2.
3. 声明一个同名的局部变量
4. 使用的是局部变量
5.
6. 输出“local”

全局变量与局部变量

变量作用域

```
1  scope = "global";
2  function checkscope() {
3      scope = "local";
4      document.write(scope);
5      myscope = "local";
6      document.write(myscope);
7  }
8  checkscope();
9  document.write(scope);
10 document.write(myscope);
```

1. 隐式声明一个全局变量
2.
3. 改变了全局变量
4. 使用的是全局变量
5. 隐式声明一个新的全局变量
6. 使用的是新的全局变量
7.
8. 输出“locallocal”
9. 输出“local”
10. 输出“local”

变量的隐式声明

变量作用域

```
1  var scope = "global scope";
2  function checkscope() {
3      var scope = "local scope";
4      function nested() {
5          var scope = "nested scope";
6          document.write(scope);
7      }
8      nested();
9  }
10 checkscope();
```

1. 全局变量
2.
3. 局部变量
4. 嵌套函数
5. 局部变量的嵌套作用域
6. 输出“nested scope”

变量的嵌套作用域

变量作用域

```
1  function test(o);
2      var i = 0;
3      if (typeof(o) == "object") {
4          var j = 0;
5          for (var k = 0; k < 10; k++) {
6              document.write(k);
7          }
8          document.write(k);
9      }
10     document.write(j);
11 }
```

1.
2.
3.
4.
5.
6.
7.
8. k 仍旧有定义
9.
10. j 仍旧有定义

JavaScript没有块级作用域

变量作用域

```
1 var scope = "global";
2 function checkscope() {
3     document.write(scope);
4     var scope = "local";
5     document.write(scope);
6 }
6 checkscope();
```

1.
2.
3. 输出"undefined"，而不是"global"
4.
5. 输出"local"

局部变量在整个函数体内部都有定义

但是在执行var语句之前，是不会被初始化的

变量类型

基本类型	数值 布尔值 null 未定义的值
引用类型	对象 数组 函数

```
1 var a = 3.14;
2 var b = a;
3 a = 4;
4 alert(b);
```

1. 声明并初始化一个变量
2. 把值复制到一个新变量
3. 修改原始变量的值
4. 显示3.14，副本没有改变

```
1 var a = [1, 2, 3];
2 var b = a;
3 a[0] = 99;
4 alert(b);
```

1. 初始化一个引用数组变量
2. 把引用复制到一个新变量
3. 用原始引用修改数组
4. 显示改变改变后的数组[99, 2, 3]

目录

函数定义

```
function foo(x, y)
{
    return x * y;
}
```

由关键字function构成，后面跟随的是：

- 函数名
- 一个用括号括起来的参数列表，这个列表是可选的，其中的参数用逗号分隔开
- 构成函数主体的JavaScript语句，包含在大括号中

函数定义

```
var foo = new Function("x", "y", "return x * y;");
```

Function()构造函数可以接受任意多个字符串参数。它的最后一个参数是函数的主体，其中可以包含任何Javascript语句，语句之间用逗号分隔。其他的参数都是用来说明函数要定义的形式参数名的字符串。

优点：Function()构造函数允许我们动态的建立和编译一个函数

缺点：每次调用一个函数时，Function()构造函数都要对它进行编译

函数定义

```
var foo = function(x, y)
{
    return x * y;
}
```

函数直接量是一个表达式，它可以定义匿名函数

```
var foo = function fact(x)
{
    if (x <= 1) {
        return 1;
    } else {
        return x * fact(x - 1);
    }
}
```

命名的函数直接量只是允许函数体用这个名字来引用自身

函数直接量只被解析和编译一次

函数参数

JavaScript是一种无类型语言，所以不能给函数的参数指定一个数据类型，而且JavaScript也不会检测传递的数据是不是函数所要求的类型。

JavaScript也不会检测传递的参数个数是否正确。如果传递的参数比函数所需要的个数多，那么多余的值会被忽略掉。如果传递的参数比函数所需的个数少，那么多余的参数就会被赋予undefined。

函数参数

如何检测调用函数是否传入了正确数目的参数？

如何实现能够接受任意数目的参数的函数？

函数参数

```
function foo(x, y, z)
{
    if (arguments.length == 3) {
        ...
    }
}
```

arguments是一个特殊的对象 Arguments 对象。数组arguments[]允许完全的存取参数值，即使参数没有被命名。

函数参数

```
function max()
{
    var m = Number.NEGATIVE_INFINITY;
    for (var i = 0; i < arguments.length; i++) {
        if (arguments[i] > m) {
            m = arguments[i];
        }
    }
    return m;
}
```

函数参数

```
function foo(x)
{
    alert(x);
    arguments[0] = null;
    alert(x);
}
```

arguments[]和命名了的参数是引用同一变量的不同方法。用参数名改变一个参数的值同时会改变通过arguments[]获得的值。通过arguments[]改变参数的值同样会改变用参数名获取的参数值。

函数参数

```
function (x)
{
    if (x <= 1) {
        return 1;
    } else {
        return x * arguments.callee(x - 1);
    }
}
```

arguments的callee属性用来引用当前正在执行的函数。

目录

对象创建

```
var obj = new Object();
var now = new Date();
```

```
var circle = {x: 0, y: 0, radius: 2};
var homer = {
    name: "Homer",
    age: 34,
    married: true
};
```

对象属性

```
var book = new Object();
book.title = "JavaScript";
```

可以通过把一个值赋给对象的一个新属性来创建它

提示：

如果要读取一个不存在的属性的值，那么得到undefined

对象构造函数

```
function Rectangle(w, h)
{
    this.width = w;
    this.height = h;
}

var rect = new Rectangle(2, 4);
```

构造函数是具有两个特性的JavaScript函数：

- 它由new运算符调用
- 传递给它的是一个对新创建的空对象的引用，将该引用作为关键字this的值，而且它还要对新创建的对象进行适当的初始化。

对象方法

```
function Rectangle_area()
{
    return this.width * this.height;
}

function Rectangle(w, h)
{
    this.width = w;
    this.height = h;
    this.area = Rectangle_area;
}

var rect = new Rectangle(2, 4);
var area = rect.area();
```

所谓方法，其实就是通过对象调用的JavaScript函数。

对象方法

```
function Rectangle(w, h)
{
    this.width = w;
    this.height = h;
    this.area = Rectangle_area;
}
```

问题：每次构造对象时都要给area方法赋值，并且该方法占用内存空间。

原型对象

JavaScript中的所有函数都有prototype属性，它引用了一个对象。该对象由Object()创建。在其中定义的任何属性都会被该构造函数创建的所有对象继承。

在读对象属性时，JavaScript首先检查对象本身是否具有该属性；如果没有，再检查原型对象是否具有该属性。

属性的继承只发生在读属性值时，写属性值时不会发生。

原型对象适用于方法定义和具有常量值的属性定义。

原型对象

```
function Rectangle(w, h)
{
    this.width = w;
    this.height = h;
}

Rectangle.prototype.area = function() {
    return this.width * this.height;
}

var rect = new Rectangle(2, 4);
var area = rect.area();
```

对象构造函数

每个对象都具有constructor属性，它引用的是用来初始化该对象的构造函数。这个属性是从原型对象继承来的。

```
Rectangle.prototype.constructor == Rectangle
Rectangle.constructor == Rectangle
```

确定一个未知对象的类型

```
if ((typeof obj == "object") && (obj.constructor == Rectangle))
```

对象继承

```
function Rectangle(w, h)
{
    this.width = w;
    this.height = h;
}

Rectangle.prototype.area = function() {
    return this.width * this.height;
}
```

```
function Square(size)
{
    this.width = size;
    this.height = size;
}

Square.prototype = new Rectangle(0, 0);
Square.prototype.constructor = Square;
```

对象继承

```
Square.prototype = new Rectangle(0, 0);
```

用Rectangle替换了原来的Object，Square也就继承了Rectangle的原型对象的属性。

对象继承

```
Square.prototype.constructor = Square;
```

用Rectangle替换了原来的Object后，Square的prototype.construtor也就被改为Rectangle的constructor，因此需要把Square的constructor重新改回Square。

推荐大家看的JavaScript书