JavaScript高级编程

目录

[JavaScript高级编程 1](#_Toc501479707)

[一 、JS基础概念 7](#_Toc501479708)

[1.JavaScript简介 7](#_Toc501479709)

[2.JavaScript标签 8](#_Toc501479710)

[3.JS数据类型 9](#_Toc501479711)

[4.注释 9](#_Toc501479712)

[5.变量 9](#_Toc501479713)

[5.1 typeof 11](#_Toc501479714)

[5.2 undefined 11](#_Toc501479715)

[5.3 Null 11](#_Toc501479716)

[5.4 Boolean 11](#_Toc501479717)

[5.5 Number 类型 12](#_Toc501479718)

[5.6 String类型 12](#_Toc501479719)

[5.7 Object 类型 12](#_Toc501479720)

[6.操作符 12](#_Toc501479721)

[6.1.一元操作符 12](#_Toc501479722)

[6.2. 一元加和减操作符 13](#_Toc501479723)

[6.3 .位操作符 \*\*\*\*\* 13](#_Toc501479724)

[6.4. 布尔操作符 13](#_Toc501479725)

[6.5.算数运算符 14](#_Toc501479726)

[6.6.关系操作符 14](#_Toc501479727)

[6.7.条件运算符 14](#_Toc501479728)

[6.8.赋值操作符 14](#_Toc501479729)

[6.9.逗号操作符 14](#_Toc501479730)

[7 语句 14](#_Toc501479731)

[7.1 if语句 15](#_Toc501479732)

[7.2 do-while语句 15](#_Toc501479733)

[7.3 while 语句 15](#_Toc501479734)

[7.4 for 循环 15](#_Toc501479735)

[7.5 for -in 循环 15](#_Toc501479736)

[7.6 label标签 16](#_Toc501479737)

[7.7 break和continue语句 16](#_Toc501479738)

[7.8 with 语句 16](#_Toc501479739)

[7.9 switch 语句 17](#_Toc501479740)

[8 函数 17](#_Toc501479741)

[8.1 函数的参数 18](#_Toc501479742)

[8.2 没有重载 19](#_Toc501479743)

[9小结 19](#_Toc501479744)

[二、变量、作用域和内存问题 19](#_Toc501479745)

[1.基本类型和引用类型的值 19](#_Toc501479746)

[1.1 动态的属性 20](#_Toc501479747)

[1.2 复制变量 20](#_Toc501479748)

[1.3 传递参数 20](#_Toc501479749)

[2 .执行环境即作用域 21](#_Toc501479750)

[2.1 延长作用域链 21](#_Toc501479751)

[2.2 没有块级作用域 22](#_Toc501479752)

[3．垃圾收集 22](#_Toc501479753)

[3.1 标记清除 22](#_Toc501479754)

[3.2 引用计数 22](#_Toc501479755)

[3.3 性能问题 22](#_Toc501479756)

[3.4 管理内存 22](#_Toc501479757)

[4 .小结 23](#_Toc501479758)

[三、引用类型 24](#_Toc501479759)

[3.1 Object 类型 24](#_Toc501479760)

[3.2 Array 数组 25](#_Toc501479761)

[3.2.1 检测数组 isArray() 25](#_Toc501479762)

[3.2.2 转换方法 25](#_Toc501479763)

[3.2.3 栈方法 26](#_Toc501479764)

[3.2.4 队列方法 26](#_Toc501479765)

[3.2.5 重排序方法 27](#_Toc501479766)

[3.2.6 操作方法 27](#_Toc501479767)

[3.2.7 位置方法 28](#_Toc501479768)

[3.2.8 迭代方法 28](#_Toc501479769)

[3.2.9 归并方法 29](#_Toc501479770)

[3.3 Date类型 29](#_Toc501479771)

[3.4 RegExp类型 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*！！！！！ 30](#_Toc501479772)

[3.4.1 RegExp 实例属性 30](#_Toc501479773)

[3.4.2 实例方法 31](#_Toc501479774)

[3.4.3 RegExp构造函数属性 31](#_Toc501479775)

[3.4.4 模式的局限性 32](#_Toc501479776)

[3.5 Function类型 32](#_Toc501479777)

[3.5.1 没有重载 33](#_Toc501479778)

[3.5.2 函数声明与函数表达式 33](#_Toc501479779)

[3.5.3 作为值的函数 34](#_Toc501479780)

[3.5.4 函数内部属性 34](#_Toc501479781)

[3.5.5 函数属性和方法 35](#_Toc501479782)

[3.5.6 自执行函数 35](#_Toc501479783)

[3.6 基本包装类型 36](#_Toc501479784)

[3.6.1 Boolean类型 36](#_Toc501479785)

[3.6.2 Number类型 37](#_Toc501479786)

[3.6.3 String类型 37](#_Toc501479787)

[3.7 单体内置方法 40](#_Toc501479788)

[3.7.1 Global对象 40](#_Toc501479789)

[3.7.2 Math对象 42](#_Toc501479790)

[3.8 小结 \*\*\*\*\* 43](#_Toc501479791)

[四、面向对象的程序设计 44](#_Toc501479792)

[4.1理解对象 44](#_Toc501479793)

[4.1.1、属性类型 ------待完善 44](#_Toc501479794)

[4.1.2、读取属性的特性（待续） 47](#_Toc501479795)

[4.1.3、对象属性的获取 47](#_Toc501479796)

[4.1.4、删除对象属性 47](#_Toc501479797)

[4.1.5、检测对象属性 47](#_Toc501479798)

[4.2 创建对象 47](#_Toc501479799)

[4.2.1 工厂模式 47](#_Toc501479800)

[4.2.2 构造函数模式 48](#_Toc501479801)

[4.2.3 原型模式 48](#_Toc501479802)

[4.2.4 组合使用构造函数模式和原型模式. 52](#_Toc501479803)

[4.3.5 动态原型模式 52](#_Toc501479804)

[4.3.6 寄生构造函数模式 53](#_Toc501479805)

[4.3.6 稳妥构造函数模式 53](#_Toc501479806)

[4.3 继承 54](#_Toc501479807)

[4.3.1 原型链 54](#_Toc501479808)

[4.3.2 借用构造函数 57](#_Toc501479809)

[4.3.3 组合继承 57](#_Toc501479810)

[4.3.4 原型式继承 58](#_Toc501479811)

[4.3.5寄生式继承 59](#_Toc501479812)

[4.3.6 寄生组合式继承 60](#_Toc501479813)

[4.4 小结 61](#_Toc501479814)

[五、函数表达式 62](#_Toc501479815)

[5.1 递归 62](#_Toc501479816)

[5.2 闭包 63](#_Toc501479817)

[5.2.1 闭包与变量 63](#_Toc501479818)

[5.2.2 关于this对象 64](#_Toc501479819)

[5.2.3 内存泄漏 65](#_Toc501479820)

[5.3 模仿块级作用域 65](#_Toc501479821)

[5.4 私有变量 66](#_Toc501479822)

[5.4.1 静态私有变量 66](#_Toc501479823)

[5.4.2 模块模式 67](#_Toc501479824)

[5.4.3 增强的模块模式 68](#_Toc501479825)

[5.5 小结 69](#_Toc501479826)

[六、 BOM 70](#_Toc501479827)

[6.1 window对象 70](#_Toc501479828)

[6.1.1 全局作用域 70](#_Toc501479829)

[6.1.2 窗口关系及框架 71](#_Toc501479830)

[七、客户端检测 71](#_Toc501479831)

[八、DOM 71](#_Toc501479832)

[九、DOM拓展 71](#_Toc501479833)

[十、DOM2和DOM3 71](#_Toc501479834)

[十一、事件 71](#_Toc501479835)

[十二、表单脚本 71](#_Toc501479836)

[十三、使用Canvas绘图 71](#_Toc501479837)

[十四、HTML5脚本编程 71](#_Toc501479838)

[十五、错误处理与调试 71](#_Toc501479839)

[十六、JavaScript与xml 72](#_Toc501479840)

[十七、E4X 72](#_Toc501479841)

[十八、JSON 73](#_Toc501479842)

[18.1语法 73](#_Toc501479843)

[18.1.1简单值 73](#_Toc501479844)

[18.1.2 对象 74](#_Toc501479845)

[18.1.3 数组 74](#_Toc501479846)

[18.2解析与序列化 74](#_Toc501479847)

[18.2.1 JOSN对象 74](#_Toc501479848)

[18.2.2 序列化选项 75](#_Toc501479849)

[18.2.3 解析选项 77](#_Toc501479850)

[18.3 小结 77](#_Toc501479851)

[十九、Ajax与Comet 78](#_Toc501479852)

[19.1 XMLhttpRequest对象(XHR) 78](#_Toc501479853)

[19.1.1 XHR 用法 79](#_Toc501479854)

[19.1.2 HTTP头部信息 81](#_Toc501479855)

[19.1.3 Get请求 82](#_Toc501479856)

[19.1.4 POST请求 82](#_Toc501479857)

[19.2 XMLHttpRequest 2级 83](#_Toc501479858)

[19.2.1 FormData 83](#_Toc501479859)

[19.2.2 超时设定 83](#_Toc501479860)

[19.2.3 overrideMineType() 方法 84](#_Toc501479861)

[19.3 进度事件 (待完善。。。) 84](#_Toc501479862)

[19.4 跨资源共享（CORS）待完善 84](#_Toc501479863)

[19.5 其他跨域技术(待完善) 84](#_Toc501479864)

[19.5.1 图像Ping 85](#_Toc501479865)

[二十、高级技巧 85](#_Toc501479866)

[20.1 高级函数 85](#_Toc501479867)

[20.1.1 安全的类型检查 85](#_Toc501479868)

[20.1.2 作用域安全的构造函数 85](#_Toc501479869)

[20.1.3 惰性载入函数（适用于执行一次便可确定最终结构，无须重复执行相同代码） 87](#_Toc501479870)

[20.1.4 函数绑定（不太熟悉需要深入看下…） 89](#_Toc501479871)

[20.1.5 函数柯里化(622页) 91](#_Toc501479872)

[二十一、离线应用与客户端存储 91](#_Toc501479873)

[二十二、最佳实践 91](#_Toc501479874)

[22.1 可维护性 91](#_Toc501479875)

[22.1.1 什么是可维护性代码 91](#_Toc501479876)

[22.1.2 代码约定 92](#_Toc501479877)

[22.1.3 松散耦合 92](#_Toc501479878)

[22.1.4、编程实践(680页) 93](#_Toc501479879)

[22.2性能 95](#_Toc501479880)

[22.2.1、注意作用域 95](#_Toc501479881)

[22.2.2、选择正确的方法 96](#_Toc501479882)

[22.2.3、最小化语句数 99](#_Toc501479883)

[22.2.4、优化DOM交互 100](#_Toc501479884)

[22.3、部署 102](#_Toc501479885)

[22.3.1、构建过程 102](#_Toc501479886)

[22.3.2、验证 103](#_Toc501479887)

[22.3.3、压缩 104](#_Toc501479888)

[22.4、小结 104](#_Toc501479889)

[二十三、新兴的API. 105](#_Toc501479890)

[二十四、ECMAScript Harmony（待续…） 105](#_Toc501479891)

# 一 、JS基础概念

## 1.JavaScript简介

JavaScript专为与网页进行交互的脚本语言，由三部分组成：

1）ECMAScript提供核心功能

2）DOM 文档对象模型，提供访问与操作网页内容的方法与接口

3）BOM浏览器对象模型，提供与浏览器交互的接口与方法

在HTML中引入JavaScript文件的方式是使用<script>标签，该元素具有几个属性：

1. async
2. charset
3. defer
4. src
5. type一般情况下是text/javascript为默认属性

在<script>元素中引入JS文件的方式有两种：

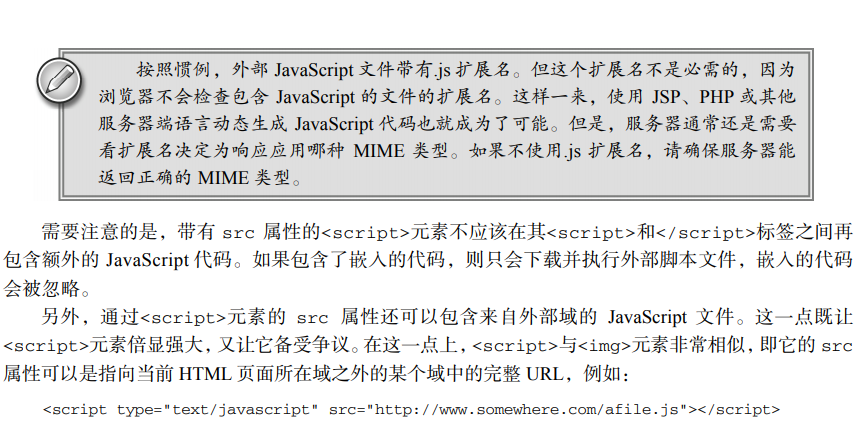
嵌入式

外联式

包含在<script>中的JS代码将从上之下执行；

在<script>中嵌入JavaScript代码时不能使用任何的”</script>”字符串，因为会被误解析为结束<script>的标志。但通过转义字符则可解决此问题“<\/script>”

当使用外联式引入JS文件时，src属性则是必需的；



Defer属性与async属性可以改变各<script>的解析顺序，正常情况下

## 2.JavaScript标签

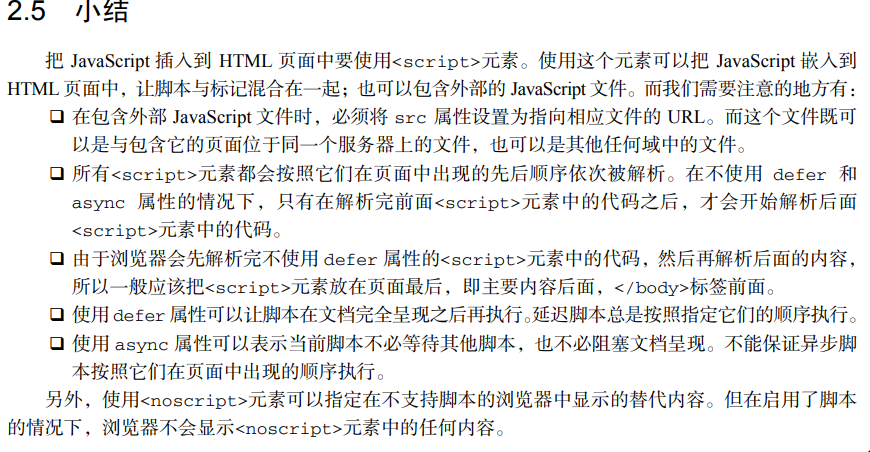
在HTML中，把JavaScript文件放在head标签，会导致延迟加载，只有JavaScript文件全部代码都被下载、解析和执行完成以后，才能呈现页面，

一般把JavaScript文件引入到body标签之后。

\*\*\*\*延迟脚本

defer属性是<script>的属性，只适用于外部脚本文件

在<script>标签使用defer属性表示立即下载，但延迟执行



## 3.JS数据类型

JS是区分大小写的，JS标识符是由数字、字母、下划线或美元符号组成，第一个字符不能使数字，（标识符也可以包括ASCII或Unicode，但不推荐），标识符命名规则一般遵循小驼峰命名法，即首字母小写

标识符不能与保留字、关键字相同。

## 4.注释

// 单行注释

/\*\*

\*/ 多行注释

## 5.变量

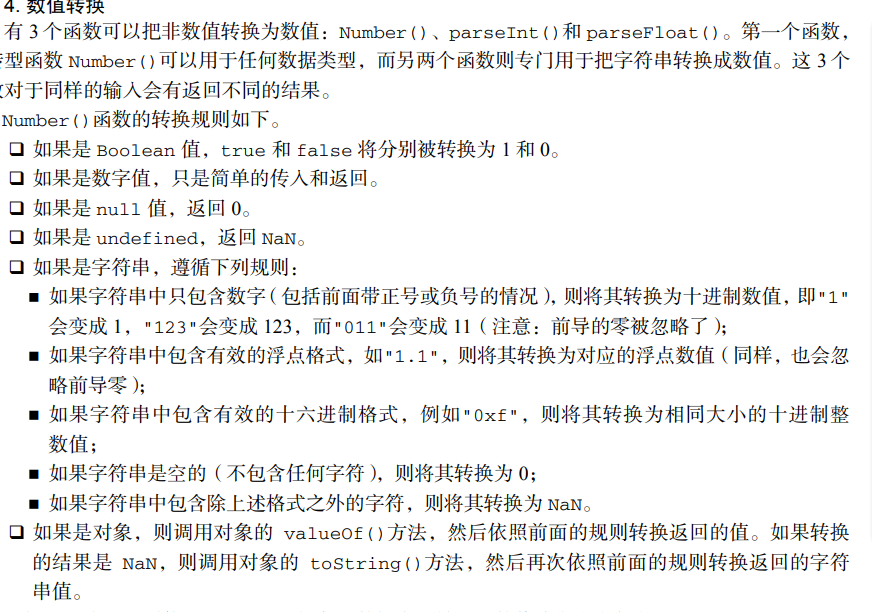
ECMAScript的变量是松散类型的，即可以用来保存任何数据类型；

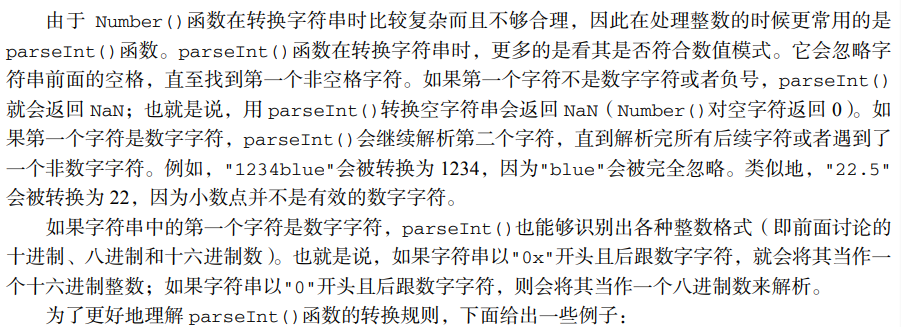
使用var创建变量，假如变量存在函数中，则创建了一个局部变量量，当不使用var创建变量时 则相当于创建一个全局变量，但一般建议使用var创建变量，因为不使用var可能造成全局命名空间污染。

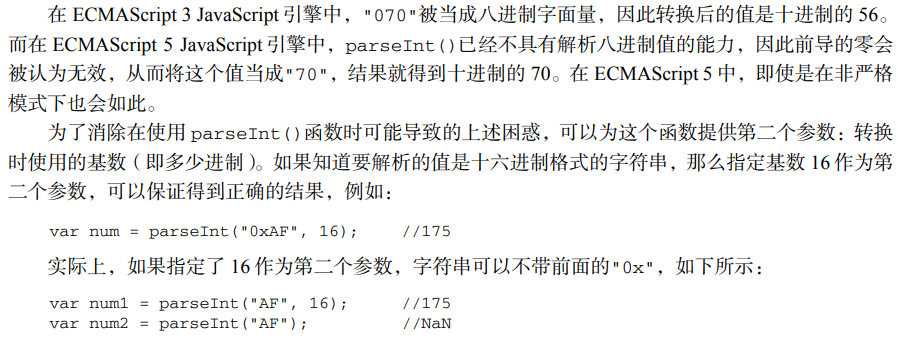
JS有5中简单的数据类型：undefined、null、String、Number、Boolean

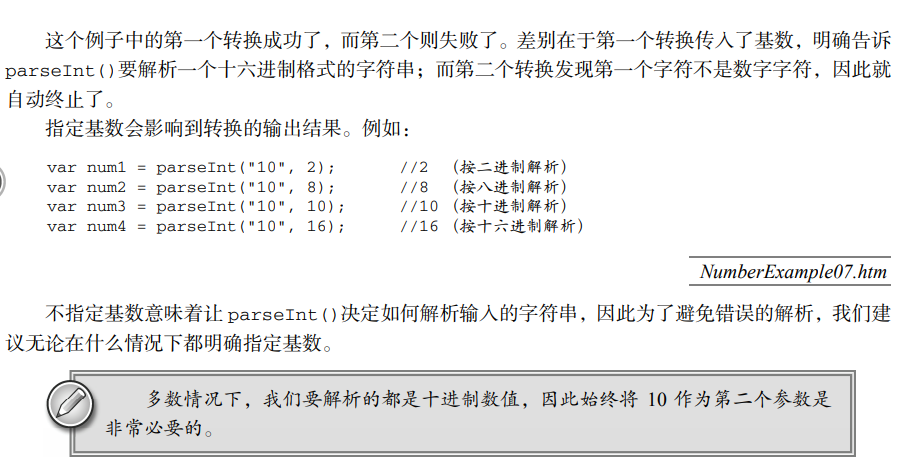
一种复杂类型：Object

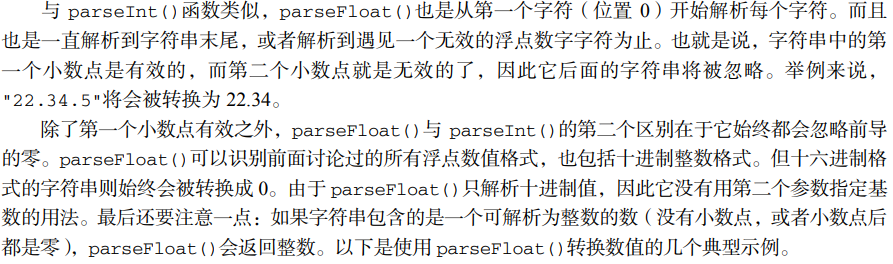
ECMAScript不支持任何自定义类型











### 5.1 typeof

用来判断数据项的类型；

注意，null会返回“Object”；undefined 会返回“Undefined”。

typeof来判断数据类型其实并不准确，

### **5.2 undefined**

Undefined类型只有一个值，即undefined；使用var声明变量但未进行赋值初始化，该变量值就是undefined

### 5.3 null

Null类型也是只有一个值的数据类型，即null；null可以用来表示一个空对象

注意：null和undefined使用“==”比较时返回true；但使用“===”则返回false。

### 5.4 Boolean

Boolean类型只有两个值：true和false；

可以使用Boolean(item) 将一个数据项转换为布尔值

### 5.5 Number 类型

1、浮点数值

该数值中必须包括一个小数点，并且该小数点后面必须包括一位数字；虽然该数据类型可以不包括整数，但不推荐。

2、数值范围

ECMAScript能保存的最大值保存在Number.MAX\_VALUE常量中

3、NaN 即非数值

在ECMAScript中任何数值除以0都会返回NaN；

isNaN( item );对于接受的数据项会自动转换为对应的数值，当无法转换时则会返回true

数值转换

### 5.6 String类型

字符串是不可能修改的

### 5.7 Object 类型

对象可以通过new操作符来创建var obj = new Object()；一般情况下如果不给对象构造函数传递参数，后面的括号可以省略，但不推荐使用。

·· Object类型所具有的任何属性和方法同样存在与更具体的对象中。Object类型是所有它的实例的基础。

Object的每个实例都具有一下属性和方法：

constructor:保存着用于保存当前对象的函数；

hasOwnProperty（propertyName）：用于检查给定的属性和方法在当前对象实例中是否存在；

isPrototypeOf(object)：用于检查传入的对象是否为传入对象的原型；

valueOf ()：返回对象的字符串、数值或布尔值表示

toString ()：返回对象的字符串表示。

## 6.操作符

算术操作符、位操作符、关系操作符、相等操作符

当对象与数值进行操作时，会首先调用toString()或者valueOf()方法将对象转换成相应的类型

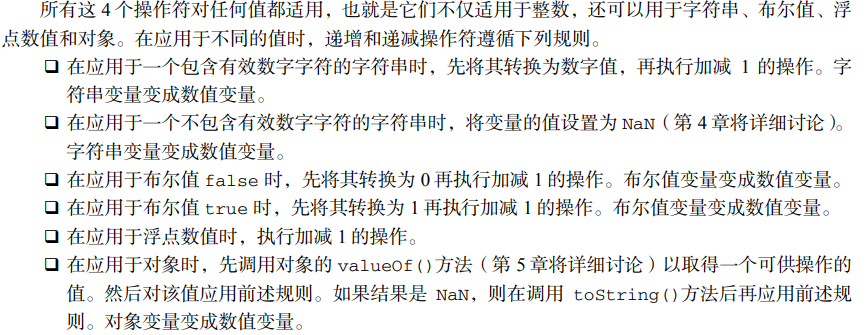
### 6.1.一元操作符

只能操作一个值的操作符 ++ 、 -- ！

++ 、-- 在前：先运算后输出；在后：先输出在运算

！ ：取反操作

对自增、自减操作符：



### 6.2. 一元加和减操作符

一元加操作符 以 （+）表示，放在数值前面对数值不会产生任何影响；但将在非数值前就会想Number()函数将数据项进行转类型。注：不会转换ASCII码

一元减操作符 以(-) 表示，主要用于表示负数；一元减操作符作用于数值上会将数值转换为负数，而作用于非数值上时，会遵循一元减操作符规则进行转换成数值，然后将得到的数值转换成负数。

### 6.3 .位操作符 \*\*\*\*\*

位操作符是根据内存中数值的位数来操作数值的，

### 6.4. 布尔操作符

布尔操作符一共有三个：与（and）、或（or）、非（not）

* 与 操作使用 &&表示；
* 或 操作使用 || 表示；
* 非 操作使用 ！ 表示。

对于（与 &&）和（或 ||）操作符会形成“短路”，对于&& 操作符，当第一个布尔表达式为假false时，将不会执行后面的表达式；当 || 操作时，当第一个表达式为真true时，后面的表达式将不会执行。

一般我们可以通过 || 操作符对变量进行赋值，如：

var name = true;

var host；

var obj = name || host；

当name为真时，会将true赋值给obj，当name为false时会将host 赋值给obj,但host未定义时会报错

### 6.5.算数运算符

+加、-减、\*乘 、/除 、%取模

如果取值运算结果超出ECMAScript的数值表示范围会返回一个 Infinity或 -Infinity

NaN 属性是代表非数字值的特殊值

### 6.6.关系操作符

大于 >、 大于或等于 >=、 小于<、 小于或等于<=、 等于==、 不等于!=

全等于 === 、全部等于！==

* 全等于和全部等于会首
* 先比较类型。
* 等于和不等于进行比较时会首先进行相应的转换

### 6.7.条件运算符

条件运算符 布尔表达式**？**表达式1 **：**表达式2

var name = BooleanExperssion? value1 : value2

当布尔表达式为true时，取value的值；当为假时，取value2的值

### 6.8.赋值操作符

+=、-=、\*=、/= 、%=

<<= >>= >>>=

### 6.9.逗号操作符

**，逗号操作符，逗号操作符可于声明多个变量，也可用于赋值**

## 7 语句

### 7.1 if语句

形式为：

if (expression) {

代码块；

}

if (expression) {

代码块

} else {

代码块；

}

其他 ：if…else if if…else if…else

### 7.2 do-while语句

语法：do {

代码块；

} while(expression)

循环体至少执行一次

### 7.3 while 语句

语法：while (expression) {

代码块；

}

循环体可能一次也不执行

### 7.4 for 循环

语法：

for (initialization; expression; post-loop-expression) { statement；}

在循环初始化表达式中声明变量可以使用var，也可以不使用var；变量的初始化也可以在外部。

由于ECMAScript中不存在块级作用域，因此在在循环内部定义的变量也可以在外部访问到。

for 循环的三个表达式都可以省略，当全部省略就会创建一个无限循环

### 7.5 for -in 循环

for-in 循环是一种精准迭代语句，可以用来枚举对象属性。语法：

for (property in expression) statement

为了保证使用局部变量,一般使用**var**操作符ECMAScript 对象的属性没有顺序。因此，通过 for-in 循环输出的属性名的顺序是不可预测的。 具体来讲，所有属性都会被返回一次，但返回的先后次序可能会因浏览器而异。 但是，如果表示要迭代的对象的变量值为 null 或 undefined，for-in 语句会抛出错误。 ECMAScript 5 更正了这一行为；对这种情况不再抛出错误，而只是不执行循环体。为了保证最大限度的兼容性，建议在使用 for-in 循环之前，先检测确认该对象的值不是 null 或 undefined。

for-in循环适用于遍历对象属性或原型链的，对于遍历以数字索引的应当使用for循环常规写法。

在JS中使用for-in记性循环遍历时，尽量使用**hasPwnProperty()**方法获取是否为当前对象的属相，除非查找原型链，应当注释

例如： for （pro in Object）{

if(Object.hasPwnProperty(pro)){

语句块；

}

}

### 7.6 label标签

语法：label: statement

加标签的语句一般都 要与 for 语句等循环语句配合使用.

### 7.7 break和continue语句

break 和 continue 语句用于在循环中精确地控制代码的执行。其中，break 语句会立即退出循环， 强制继续执行循环后面的语句。而 continue 语句虽然也是立即退出循环，但退出循环后会从循环的顶部继续执行。

break 和 continue 语句都可以与 label 语句联合使用，从而返回代码中特定的位置。这种联合 使用的情况多发生在循环嵌套的情况下。

### 7.8 with 语句

with 语句的作用是将代码的作用域设置到一个特定的对象中。

with 语句的语法如下：

with (expression) statement;

定义 with 语句的目的主要是为了简化多次编写同一个对象的工作，

如下面的例子所示：

var qs = location.search.substring(1);

var hostName = location.hostname;

var url = location.href;

上面几行代码都包含 location 对象。如果使用 with 语句，可以把上面的代码改写成如下所示：

with (location) {

var qs = search.substring(1);

var hostName = hostname;

var url = href;

}

严格模式下不允许使用 with 语句，否则将视为语法错误.

### 7.9 switch 语句

switch (expression) {

case value: statement

break;

case value: statement

break;

case value: statement

break;

case value: statement

break;

default: statement

}

switch 语句中的每一种情形（case）的含义是：“如果表达式等于这个值（value），则执行后面的 语句（statement）”。而 break 关键字会导致代码执行流跳出 switch 语句。如果省略 break 关键字， 就会导致执行完当前 case 后，继续执行下一个 case。最后的 default 关键字则用于在表达式不匹配前面任何一种情形的时候，执行机动代码（因此，也相当于一个 else 语句）。

通过为每个 case 后面都添加一个 break 语句，就可以避免同时执行多个 case 代码的情况。假如确实需要混合几种情形，不要忘了在代码中添加注释，说明你是有意省略了 break 关键字。

首先，在JavaScript中可以在 switch 语句中使用任何数据类型（在很多其他语言中只能使用数值），无论是字符串，还是对象都没有 问题。其次，每个 case 的值不一定是常量，可以是变量，甚至是表达式。

## 8 函数

函数可以封装任意多条语句，可以在任何时候任何地方调用执行。ECMAScript中函数使用function关键字声明，基本语法为：

function functionName (args\_list) {

代码块；

}

也可以通过自执行函数进行入参,例如：

-----------------------------------------------------------------------

JavaScript的自执行函数 使用()表示自执行，（）只是表示自执行作用

(function(){})(); 是 把函数当作表达式解析，然后执行解析后的函数，与

相当于 var a = function(){}; a(); a得到的是函数

(function(){}()); 是把函数表达式和执行当作语句直接执行，

与!function(){}() 和+function(){}()相同作用

相当于 var a = function(){}(); a得到的是结果

自执行函数避免用函数名污染全局命名空间

-------------------------------------------------------------------------------

ECMAScript 中的函数在定义时不必指定是否返回值。实际上，任何函数在任何时候都可以通过 return 语句后跟要返回的值来实现返回值。特别注意的是，return之后的语句将不再执行。

一个函数中也可以包含多个 return 语句；

return 语句也可以不带有任何返回值。在这种情况下，函数在停止执行后将返回 undefined 值。这种用法一般用在需要提前停止函数执行而又不需要返回值的情况下

对于 return ；语句 一般返回undefined值

### 8.1 函数的参数

ECMAScript 中的参数在内部是用一个数组来表示的；实际上，在函数体内可以通过 arguments 对象来 访问这个参数数组，从而获取传递给函数的每一个参数

在JavaScript文件内部可以使用arguments来接受全部参数，接收的参数形式为数组形式。

ECMAScript 函数的一个重要特点：命名的参数只提供便利，但不是必需的。

通过访问 arguments 对象的 length 属性可以获知有多少个参数传递给了函数。

是 arguments 对象可以与命名参数一起使用，即在函数中获取参数既可以通过实参参数名也可以通过arguments来获取，例如：

function doAdd(num1, num2) {

if (arguments. length == 1) {

alert(num1 + 10);

} else if (arguments.length == 2)

{alert (arguments [0] + num2);

}

}

关于 arguments 的行为，还有一点比较有意思。那就是它的值永远与对应命名参数的值保持同步；如上例中arguments[0] == num1, 而当arguments[0] = num1；进行赋值时，相应的会把arguments[0]对应的参数名的值进行重新赋值，即修改某个参数传入值。

但arguments[0] == num1只是说明两个值是相同的，但它们的内存空间是独立的，只是值会同步，

arguments对象的长度是由传入的参数个数决定的，不是由定义函数时的命名参数的个数决定的。 （在JavaScript中形参和实参的数量并不严格相等，实参数量可以少于形参）

关于参数还要记住最后一点：

没有传递值的命名参数将自动被赋予 undefined 值。这就跟定义了 变量但又没有初始化一样。例如，如果只给 doAdd()函数传递了一个参数，则 num2 中就会保存 undefined 值。

严格模式对如何使用 arguments 对象做出了一些限制。首先，赋值会变得无效。也就是说，即使把 arguments[1]设置为 10，num2 的值仍然还是 undefined。其次，重写 arguments 的值会导致语法错误（代码将不会执行）。

ECMAScript 中的所有参数传递的都是值，不可能通过引用传递参数。

### 8.2 没有重载

ECMAScript 函数不能像传统意义上那样实现重载。

当存在两个及以上的相同函数名，后定义的函数会覆盖先定义的函数

### 8.3 函数的调用

分为方法调用、构造函数调用、间接调用

### 8.4 函数的实参与形参

JavaScript中函数定义并未指定函数参数类型的类型，函数调用也未对传入的实参做任何类型检查，甚至不用检查传入形参的个数。

#### 8.4.1 可选形参

当函数调用传入的实参个数比函数声明时指定的形参个数少，剩下的形参的值都设为undefined；在函数调用时形参是否可选或是否可以省略应当保持较好的适应性。因此应当给省略的参数一个合理的默认值，即当可选参数为undefined时，给予默认值，使用“ **||** ”。例如var a = a || [];当a为undefined时，默认为空数组。

默认情况下也可以使用null作为一个无意义的占位符。

#### 8.4.2 可变长的实参列表

当函数调用传入的实参个数大于函数定义的形参个数时，没办法直接获取未命名值的引用。为解决传入实参个数多于形参个数，使用arguments内置对象arguments指向实参的引用，是一个类数组，可以使用索引下标来访问实参。

* 因此，当传入实参个数与形参个数不一致的情况下：实参个数小于形参个数，未传入实参的形参会默认为undefined；当实参个数大于形参个数，对于未命名的形参可以通过arguments类数组使用索引获取，例如第一项参数可以用arguments[0]表示，第二项参数可以使用arguments[1]表示。同理，也可以通过arguments与索引来改变参数值。
* 通常，对于实参和形参不相同情况：省略的实参都将是undefined，多出的参数会自动省略。

## 9小结

下简要总结了 ECMAScript 中基本的要素。

* ECMAScript 中的基本数据类型包括 Undefined、Null、Boolean、Number 和 String。
* 与其他语言不同，ECMAScript 没有为整数和浮点数值分别定义不同的数据类型，Number 类型可 用于表示所有数值。
* ECMAScript 中也有一种复杂的数据类型，即 Object 类型，该类型是这门语言中所

有对象的基础类型。

* 严格模式为这门语言中容易出错的地方施加了限制。
* ECMAScript 提供了很多与 C 及其他类 C 语言中相同的基本操作符，包括算术操作符、布尔操作 符、关系操作符、相等操作符及赋值操作符等。
* ECMAScript 从其他语言中借鉴了很多流控制语句，例如 if 语句、for 语句和 switch 语句等。 ECMAScript 中的函数与其他语言中的函数有诸多不同之处。
* 无须指定函数的返回值，因为任何 ECMAScript 函数都可以在任何时候返回任何值。
* 实际上，未指定返回值的函数返回的是一个特殊的 undefined 值。
* ECMAScript 中也没有函数签名的概念，因为其函数参数是以一个包含零或多个值的数组的形式 传递的。
* 可以向 ECMAScript 函数传递任意数量的参数，并且可以通过 arguments 对象来访问这些参数。
* 由于不存在函数签名的特性，ECMAScript 函数不能重载。

# 二、变量、作用域和内存问题

## 1.基本类型和引用类型的值

ECMAScript可能包含两种不同的数据类型的值：基本类型值和引用类型值。

基本类型是按值访问的，可以直接操纵保存在变量中实际的值；

引用类型的值保存在内存中的对象，JS不允许直接访问内存中的值即不能直接操作对象的内存空间。引用类型的值是按引用访问的。

### 动态的属性

定义基本类型值和引用类型值的方式是类似的：创建一个变量并为该变量赋值，对于引用类型的值，我们可以为其添加属 性和方法，也可以改变和删除其属性和方法。

只能给引用类型值动态地添加属性，为基本类型添加属性，饮用可能会造成错误。

### 复制变量

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，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复制引用类型的值时，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一个指针，而这个指针指向存储在堆中的一个对象。复制操作结束后，两个变量实际上将引用同一个对象。因此，改变其中一个变量，就会影响另一个变量。相当于两个引用类型变量指向同一个空间地址。

### 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。基本类型值的传递如同基本类型变量的复制一样，而 引用类型值的传递，则如同引用类型变量的复制一样。

访问变量有按值和按引用两种方式，而参数只能按值传递。

在向参数传递基本类型的值时，被传递的值会被复制给一个局部变量（即命名参数，或者用 ECMAScript 的概念来说，就是 arguments 对象中的一个元素）。在向参数传递引用类型的值时，会把这个值在内存中的地址复制给一个局部变量，因此这个局部变量的变化会反映在函数的外部。

以基本类型进行参数传递并不影响变量的值

通过引用类型进行传参，即使在函数内部修改了参数的值，但原始的引用仍然保持未变。

在函数中传参时，可以将对象属性作为实参传递。参数赋值时，可以使用 或“ || ”；函数也可以作为值进行处理；

* 1. **检测类型**

typeof 通常用于检测基本类型

instanceof 可以用于检测引用类型

根据规定，所有引用类型的值都是 Object 的实例。因此，在检测一个引用类型值和 Object 构造 函数时，instanceof 操作符始终会返回 true。当然，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。

## 2 .执行环境即作用域

执行环境定义了变量或函数有权访问的其他数据，决定了它们的行为。

全局执行环境是最外围的执行环境。全局执行环境被认为是window对象，所有的全局变量和函数都是作为window对象的属性和方法创建的。环境被销毁，保存在环境中的属性高和方法也会随之被销毁，，而全局执行环境知道应用程序退出才会被销毁。

每个函数都有自己的执行环境。

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 chain）。作用域链的用途，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。作用域链的前端，始终都是当前执行的代码所在环境的变量对象。如果这个环境是函数，则将其活动对象（activation object）作为变量对象。活动对象在最开始时只包含一个变量，即 arguments 对象（arguments 对象在全局环境中是不存在的）。作用域链中的下一个变量对象来自包含（外部）环境，而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；全局执行环境的变量对象始终都是作用域链中的最后一个对象。

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。

搜索过程始终从作用域链的前端开始， 然后逐级地向后回溯，直至找到标识符为止（如果找不到标识符，通常会导致错误发生）。

* 在局部作用域中定义的变量可以在局部环境中与全局变量互换使用
* 内部环境可以通过作用域链 访问所有的外部环境，但外部环境不能访问内部环境中的任何变量和函数。这些环境之间的联系是线性、有次序的。每个环境都可以向上搜索作用域链，以查询变量和函数名；但任何环境都不能通过向下搜索作用域链而进入另一个执行环境。

### 2.1 延长作用域链

执行环境的类型总共只有两种——全局和局部（函数），但还是有其他办法来延长作用域链，在两种情况下会发生这种现象。具体来说，就是当执行流进入下列任何一个语句时，作用域链就会 得到加长：

* + try-catch 语句的 catch 块；
  + with 语句。

这两个语句都会在作用域链的前端添加一个变量对象。对 with 语句来说，会将指定的对象添加到作用域链中。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。

待续。。。

### 2.2 没有块级作用域

#### 2.2.1 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就是函数的环境。如果初始化变量时没有使用var声明，变量就会自动被添加到全局环境。

#### 2.2.2 查询标识符

搜索过程从作用域的前端开始，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，就停止搜索；如果在局部环境中没有找到就逐级向上查询，直到追溯到全局环境的变量对象。如果在全局环境中也没有找到，则说明该变量尚未声明。

## 3．垃圾收集

JS具有自动垃圾收集机制，即执行环境会管理代码执行过程中使用的内存。

垃圾收集机制的原理：释放那些不再使用的变量所占用的内存。

### 3.1 标记清除

标记清除是JavaScript最常用的垃圾收集方式。

### 3.2 引用计数

不太常见的垃圾收集策略叫做引用计数（reference counting）。引用计数的含义是跟踪记录每个值被引用的次数。

### 3.3 性能问题

JavaScript 引擎的垃圾收集例程改变了工作方式：触发垃圾收集的变量分配、 字面量和（或）数组元素的临界值被调整为动态修正。

IE7 中的各项临界值在初始时与 IE6 相等。如果垃圾收集例程回收的内存分配量低于 15%，则变量、字面量和（或）数组元素的临界值就会加倍。如果例程回收了85%的内存分配量，则将各种临界值重置回默认值。

### 3.4 管理内存

在JavaScript中是分配给 Web 浏览器的可用内存数量通常要比分配给桌面应用程序的少，目的是防止运行 JavaScript 的网页耗尽全部系统内存而导致系统崩溃。

优化内存占用的最佳方式，就是为执行中的代码只保存必要的数据。一旦数据不再有用，最好通过将其值设置为 null 来释放其引用——这个做法叫做解除引用（dereferencing）。这一做法适用于大多数全局变量和全局对象的属性。局部变量会在它们离开执行环境时自动被解除引用。

解除一个值的引用并不意味着自动回收该值所占用的内存。解除引用的真正作用是让值脱离执行环境，以便垃圾收集器下次运行时将其回收。

## 4 .小结

JavaScript 变量可以用来保存两种类型的值：基本类型值和引用类型值。

基本类型的值源自以下5种基本数据类型：Undefined、Null、Boolean、Number 和 String。

基本类型值和引用类型值具有以下特点：

* 1. 基本类型值在内存中占据固定大小的空间，因此被保存在栈内存中;
  2. 从一个变量向另一个变量复制基本类型的值，会创建这个值的一个副本；
  3. 引用类型的值是对象，保存在堆内存中；
  4. 包含引用类型值的变量实际上包含的并不是对象本身，而是一个指向该对象的指针； 从一个变量向另一个变量复制引用类型的值，复制的其实是指针，因此两个变量最终都指向同一个对象；
  5. 确定一个值是哪种基本类型可以使用 typeof 操作符，而确定一个值是哪种引用类型可以使用 instanceof 操作符。 所有变量（包括基本类型和引用类型）都存在于一个执行环境（也称为作用域）当中，这个执行环境决定了变量的生命周期，以及哪一部分代码可以访问其中的变量。

以下是关于执行环境的几 点总结：

1. 执行环境有全局执行环境（也称为全局环境）和函数执行环境之分；
2. 每次进入一个新执行环境，都会创建一个用于搜索变量和函数的作用域链；
3. 函数的局部环境不仅有权访问函数作用域中的变量，而且有权访问其包含（父）环境，乃至全局环境；
4. 全局环境只能访问在全局环境中定义的变量和函数，而不能直接访问局部环境中的任何数据；
5. 变量的执行环境有助于确定应该何时释放内存。 JavaScript是一门具有自动垃圾收集机制的编程语言，开发人员不必关心内存分配和回收问题。

可以对 JavaScript 的垃圾收集例程作如下总结。

* + 离开作用域的值将被自动标记为可以回收，因此将在垃圾收集期间被删除。
  + “标记清除”是目前主流的垃圾收集算法，这种算法的思想是给当前不使用的值加上标记，然 后再回收其内存。
  + 另一种垃圾收集算法是“引用计数”，这种算法的思想是跟踪记录所有值被引用的次数，JavaScript 引擎目前都不再使用这种算法；但在 IE中访问非原生 JavaScript 对象（例如DOM 元素）时，这种 算法仍然可能会导致问题。
  + 当代码中存在循环引用现象时，“引用计数”算法就会导致问题。

解除变量的引用不仅有助于消除循环引用现象，而且对垃圾收集也有好处。为了确保有效地回 收内存，应该及时解除不再使用的全局对象、全局对象属性以及循环引用变量的引用。

# 三、引用类型

引用类型的值（对象）是引用类型的一个实例。在 ECMAScript 中，引用类型是一种数据结构，用于将数据和功能组织在一起。它也常被称为类，但这种称呼并不妥当。尽管 ECMAScript 从技术上讲是一门面向对象的语言，但它不具备传统的面向对象语言所支持的类和接口等基本结构。引用类型有时候也被称为对象定义，因为它们描述的是一类对象所具有的属性和方法。

对象是某个特定引用类型的实例，新对象是使用new操作符后跟一个构造函数创建的，构造函数本身就是一个函数，只不过该函数是出于创建新对象而创建的。

## 3.1 Object 类型

大多说引用类型都是Object类型的实例。

创建Object实例的方式有两种：

一种是使用new操作符跟Object的构造函数； var person = new Object ();

另外一种是使用对象字面量的表示法。 var person= {};在使用对象字面量语法时，属性名也可以使用字符串，使用对象字面量语法时，如果留空其花括号，则可以定义只包含默认属性和方法的对象.

在JavaScript语言核心中，原始类型都包含内置构造函数。

所有通过对象字面量创建的对象都具有同一个原型对象，并可以通过JavaScript代码Object.prototype获取对原型对象的引用。而通过关键字new和构造函数的调用创建的对象的原型就是构造函数的prototype属性的值。

Object.create（）方法用于创建对象，第一个参数表示要创建的对象，第二个参数是可选参数，表示对创建对象的增强，形式：

*var* myObj = Object.create(obj,{country:{value:"中国"}});。

通过传入参数null创建一个没有原型对象的新对象，通过这种方式创建的对象，不会继承任何属性和方法，甚至不包括基础方法。语法：**var o = Object.create(null);**

如果想创建一个空对象，可以传入参数Object.prototype：

**var o = Object.create(**Object.prototype**);**

可以通过任意原型创建新对象即可以使任意对象可继承；

对象字面量也是向函数传递大量可选参数的首选方式。

传参最好的做法 是对那些必需值使用命名参数，而使用对象字面量来封装多个可选参数。

一般来说，访问对象属性时使用的都是点表示法，JavaScript 也可以使用方括号表示法来访问对象的属性。在使用方括号语法时，应该将要访问的属性以字符串的形式放在方括号中。方括号语法的主要优点是可以通过变量 来访问属性。

如果属性名中包含会导致语法错误的字符，或者属性名使用的是关键字或保留字，也可以使用方括号表示法。

属性名中是可以包含非字母非数字的，这时候就可以使用方括号表示法来访问它们。 通常，除非必须使用变量来访问属性，否则我们建议使用点表示法。

## 3.2 Array 数组

ECMAScript 数组的大小是可以动态调整的，即可以随着数据的添加自动增长以容纳新增数据。

创建数组的基本方式有两种。第一种是使用 Array 构造函数，如果预先知道数组要保存的项目数量，也可以给构造函数传递该数量，而该数量会自动变成 length 属性的值；也可以使用数组字面量创建数组，数组字面量的语法允许有可选的结尾的逗号。故**[ , , ]**只有两个元素且长度为2.

在使用 Array 构造函数时也可以省略 new 操作符。

在读取和设置数组的值时，要使用方括号并提供相应值的基于 0 的数字索引。

方括号中的索引表示要访问的值。如果索引小于数组中的项数，则返回对应项的值；。如果设置某个值的索引超过了数组现有项数，数组就会自动增加到该索引 值加 1 的长度（就这个例子而言，索引是 3，因此数组长度就是 4）；

数组的项数保存在其 length 属性中，这个属性始终会返回 0 或更大的值；

数组的 Array.length 属性并不是只读的。因此，通过设置这个属性，可以从数组的末尾移除项或向数组中添加新项。如

var colors = ["red", "blue", "green"]; // 创建一个包含3个字符串的数组

colors.length = 2; -------修改了colors的长度

alert (colors [2]); //undefined

如果将其 length 属性设置为大于数组项数的值，则新增的每一项都会取得 undefined 值.

* 就是说在JavaScript中数组的length属性是可以修改的，当修改后的数值大于数组长度时，会默认给新增的每一个赋予一个undefined值。同样也可以给数组的末尾添加数据项。当向超出数组Array.length-1的位置赋值时，会自动增加数组长度，未赋值的位置的值为undefined。

### 3.2.1 检测数组 isArray()

### 3.2.2 转换方法

所有的对象都具有toString()、toLocaleString()、valueOf()方法

输出数组时会默认调用toString ()方法，

* 当数组含有toString()、toLocaleString()、valueOf() 方法时 的对象时，数组调用toString()、toLocaleString()、valueOf() 的方法时，会调用数组每一项的toString()、toLocaleString()、valueOf()方法。

数组继承的 toLocaleString()、toString()和 valueOf()方法，在默认情况下都会以逗号（，）分隔的字符串的形式返回数组项。可以通过 join()方法，则可以使用不同的分隔符来构建这个字符串。join()方法只接收一个参数，即用作分隔符的字符串，然后返回包含所有数组项的字符串。

如果不给 join()方法传入任何值，或者给它传入 undefined，则使用 作为分隔符。IE7 及更早版本会错误的使用字符串"undefined"作为分隔符。

如果数组中的某一项的值是 null 或者 undefined，那么该值在 join()、 toLocaleString()、toString()和 valueOf()方法返回的结果中以空字符串表示；

* 即数组中的null或undefined值，则调用join()、toLocaleString()、toString()和 valueOf()方法以空字符串形式展现。

### 3.2.3 栈方法

ECMAScript提供了可以让数组的行为类似栈结构的方法。栈是一种先进后出，后进先出的数据结构，即最晚添加的元素最先移除。在栈数据结构中，项的插入叫做推入，项的移除叫做弹出，并且只发生在栈的顶部。

ECMAScript为数组提供栈行为了pop()和push()方法。

* push() 方法可以传入任意数量的参数，将他们依次插入数组的末尾，并返回修改后数组的长度；
* pop()方法移除数组的最后一个数据项，减少数组的长度，并返回弹出的数据项；

### 3.2.4 队列方法

栈数据结构的访问规则是先进后出；而队列数据结构的访问规则是先进先出。队列在列表的末端添加项，在队列的前端移除项。

push()方法向数组的末端添加数据项，shift()方法移除数组的第一项并返回该项，同时将数组长度减1，

unshift() 方法与shift()方法作用相反，向数组的前端添加任意项。

**数组 与 栈和队列**

* pop()方法可以将数组末端的数据项删除并返回删除的数据项，数组长度减1；
* push() 方法可以向数组的末端插入任意数量的数据项，并返回修改后的数组长度；
* shift() 方法可以移除数组前端的第一项并返回该删除项，数组长度减1；
* unshift() 方法可以向数组的前端添加任意数量的数据项，并返回修改后的数组长度；

数组通过pop()与push()方法可以模拟栈数据结构行为，即在数组的末端进行添加删除 栈行为 -----先进后出；

数组通过shift()与push()方法可以模拟队列的行为，即在数组的前端进行删除，在数组的末端进行添加 队列行为 -----先进先出

### 3.2.5 重排序方法

数组的重排序方法：reverse() 和sort()方法

reverse()方法将数组进行反转排序；

sort() 方法在默认情况下，将数组数据项按升序进行排列；sort()方法为了进行排序，会调用数组每一个数据项的toString()方法，比较的得到的字符串，即sort方法进行比较排序的最终为字符串。为解决这个问题sort()方法可以接受一个比较函数作为参数，比较函数可以接受两个参数，如果第一个参数位于第二个参数之前，则返回一个负数；如果两个参数相等则返回0,；如果第一个参数位于第二个参数之后则返回一个正数。该比较函数适用于大多数数据类型，只需要将其作为参数传递给sort()方法即可。例如

function compare (value1, value2) {

if (value1 < value2) {

return -1;

} else if (value1 > value2) {

return 1;

} else {

return 0; var values = [0, 1, 25, 10, 15];

} values.sort (compare);

} alert(values); // 0,1,10,15,25

通过比较函数来确定排序原则,

通过比较函数返回值的交换，可以改变数组数据项的排序方式。

如果比较函数的返回值小于0,则不交换原数组中元素的位置，否则交换原数组中元素的位置。

对于数值类型或者其valueOf()方法会返回数值类型的对象类型，可以使用一个更简单的比较函数。这个函数只要用第二个值减第一个值即可。

function compare (value1, value2) {

return value2 - value1;

}

由于比较函数通过返回一个小于零、等于零或大于零的值来影响排序结果，因此减法操作就可以适 当地处理所有这些情况。

### 3.2.6 操作方法

concat ()方法 基于当前数组中的所有项创建一个新数组。这个方法会先创建当前数组一个副本，然后将接收到的参数 添加到这个副本的末尾，最后返回新构建的数组。在没有给concat()方法传递参数的情况下，它只是复制当前数组并返回副本。如果传递给 concat()方法的是一或多个数组，则该方法会将这些数组中的每一项都添加到结果数组中。如果传递的值不是数组，这些值就会被简单地添加到结果数组的末尾。

slice () 基于当前数组中的一个或多个项创建一个新数组；slice()可以接收一个或两个参数，即返回项的起始位置和结束为止。当只有一个参数时，该方法返回从起始位置到数组结束为止的所有数据项，当有两个参数时，该方法返回从起始位置到结束位置的数据项但不包括结束位置的数据项。 \*\*\*\*slice()方法不会影响原数组。

如果 slice()方法的参数中有一个负数，则用数组长度加上该数来确定相应的位置。例如，在一个包含 5 项的数组上调用 slice(-2,-1)与调用 slice(3,4)得到的结果相同。如果结束位置小于起始位置，则返回空数组。

* **splice()** 方法 主要用途是向数组的中部插入项，但使用这种方法的方式则有如下 3 种:
  + **删除**：可以删除任意数量的项，只需指定2个参数：要删除的第一项的位置和 要删除的项数。例如，splice(0,2)会删除数组中的前两项。
  + **插入**：可以向指定位置插入任意数量的项，只需提供3个参数：起始位置、0（要删除的项数）和 要插入的项。如果要插入多个项，可以再传入第四、第五，以至任意多个项。例如，splice(2,0,"red","green")会从当前数组的位置2开始插入字符串"red"和"green"。
  + **替换**：可以向指定位置插入任意数量的项，且同时删除任意数量的项，只需指定3个参数：起始位置、要删除的项数和 要插入的任意数量的项。插入的项数不必与删除的项数相等。例如， splice (2,1,"red","green")会删除当前数组位置 2的项，然后再从位置2开始插入字符串 "red"和"green"。

splice()方法始终都会返回一个数组，该数组中包含从原始数组中删除的项（如果没有删除任何项，则返回一个空数组），即splice()返回一个由删除的数据项组成的数组，当删除项为0时，返回一个空数组。

### 3.2.7 位置方法

ECMAScript为数组定义了两个位置方法：indexOf() 与lastIndexOf()方法；这两个方法都接受参数：要查找的项和查找的起始位置（可选）。其中indexOf()从起始位置开始查找；lastIndexOf()方法从末尾开始查找。这两个方法都返回要查找的数据项的位置，或者在没找到的情况下返回-1，要求查找的项必须严格相等（就像使用 === 一样）。

### 3.2.8 迭代方法

ECMAScript一共为数组定义了5个迭代方法，每个方法都接收两个参数：要在每一项上运行的函数（匿名函数）和（可选的）要运行该函数的作用域对象---影响this的值。而传入这些方法的函数会接收三个参数：数组项的值，该项在数组的位置和数组对象本身。

every() 方法对数组中的每一项运行给定函数，如果该函数对数组中的每一项都返回true，则返回true；

filter() 方法对数组中的每一项运行给定函数，返回该函数会返回true的数据项组成的数组；

forEach() 方法对数组的每一项运行该函数，该函数没有返回值；

map() 方法对数组的每一项运行该函数，返回每次函数调用的结果组成的数组；

some() 方法对数组的每一项运行该函数，如果该函数对任一项返回true，则返回true；

以上方法都不会修改数组中的包含的值。

其中，every()和some()方法类似，every()方法对数组中的每一项进行判断，都返回true，才会返回true；some()只需要其中一项返回true，就会返回true。

filter()方法会返回函数运行结果为true的数据项；

forEach()方法没有返回值；

map()方法的返回值是匿名函数运行返回的结果值组成的数组。

**foreach 方法**

forEach和map都支持2个参数：一个是回调函数（item,index,list）和上下文；

数组中的当前项item,当前项的索引index,原始数组input；

forEach方法中的this是ary,匿名回调函数中的this默认是window；

理论上这个方法是没有返回值的，仅仅是遍历数组中的每一项，不对原来数组进行修改；

但是我们可以自己通过数组的索引来修改原来的数组；

兼容写法：/\*\*

\* forEach遍历数组

\* @param callback [function] 回调函数；

\* @param context [object] 上下文；

\*/

Array.prototype.myForEach = function myForEach(callback,context){

context = context || window;

if('forEach' in Array.prototye) {

this.forEach(callback,context);

return;

}

//IE6-8下自己编写回调函数执行的逻辑

or(var i = 0,len = this.length; i < len;i++) {

callback && callback.call(context,this[i],i,this);

}

}

**map方法**：

和forEach非常相似，都是用来遍历数组中的每一项值的，用来遍历数组中的每一项；

……区别：map的回调函数中支持return返回值；return的是啥，相当于把数组中的这一项变为啥（并不影响原来的数组，只是相当于把原数组克隆一份，把克隆的这一份的数组中的对应项改变了）；

……不管是forEach还是map 都支持第二个参数值，第二个参数的意思是把匿名回调函数中的this进行修改。

兼容写法：/\*\*

\* map遍历数组

\* @param callback [function] 回调函数；

\* @param context [object] 上下文；

\*/

Array.prototype.myMap = function myMap(callback,context){

context = context || window;

if('map' in Array.prototye) {

return this.map(callback,context);

}

//IE6-8下自己编写回调函数执行的逻辑

var newAry = [];

for(var i = 0,len = this.length; i < len;i++) {

if(typeof callback === 'function') {

var val = callback.call(context,this[i],i,this);

newAry[newAry.length] = val;

}

}

return newAry;

}

### 3.2.9 归并方法

数组的归并方法：reduce()和reduceRight()方法。这两个归并方法都会迭代数组所有项，并构建一个最终返回的值。不同的是，reduce()方法是从数组的第一项开始逐个遍历到最后；而reduceRight()方法是从数组的末端向前遍历。

这两个方法都有两个参数，一个是在每一项上调用的函数，一个是作为归并基础的初始值。传给这两个方法的函数接收四个参数：前一个值、当前值、项的索引及数组对象。这个函数返回的任何值都会作为第一个参数自动传给下一项迭代。

* 第一次迭代发生在数组的第二项上，因此第一个参数是数组的第一项，第二个参数是数组的第二项。

使用 reduce()还是 reduceRight()，主要取决于要从哪头开始遍历数组。除此之外，它们完全相同。

**可以理解，reduce()归并方法的入参匿名函数发生第一次迭代是发生在数组的第二项上，第一次迭代的第一个参数是数组的第一项，第二个参数是数组的第二项，然后迭代的每一次返回值都作为下一次迭代的匿名函数的第一个参数。**reduceRight()类似，只不过从反方向开始遍历。例如：

|  |
| --- |
| var values = [1,2,3,4,5];  var sum = values.reduce(function(prev, cur, index, array){  return prev + cur;  });  alert(sum); //15 |

### 3.2.10 类数组对象

## 3.3 Date类型

Date 类型使用自 UTC（Coordinated Universal Time，国际协调时间）1970 年 1 月 1 日午夜（零时）开始经过的毫秒数来保存日期。在使用这种数据存储格式的条件下，Date 类型保存的日期能够精确到 1970 年 1 月 1 日之前或之后的285616年。

创建日期使用new操作符和Date构造函数。

在调用 Date 构造函数而不传递参数的情况下，新创建的对象自动获得当前日期和时间。如果想根据特定的日期和时间创建日期对象，必须传入表示该日期的毫秒数（即从UTC 时间 1970年 1 月 1 日午夜起至该日期止经过的毫秒数）。为了简化这一计算过程，ECMAScript 提供了两个方法：Date.parse() 和 Date.UTC()。 其中，Date.parse()方法接收一个表示日期的字符串参数，然后尝试根据这个字符串返回相应日期的毫秒数。

Data.now()方法，返回表示调用这个方法时的日期和时间的毫秒数

## 3.4 RegExp类型 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*！！！！！

正则表达式创建语法,

以字面量形式来定义的正则表达式：var expression = /pattern/ flags。

其中模式（pattern）部分可以是任何简单或复杂的表达式，可以包含字符类、限定符、分组、向前查找或向后索引。每个正则表达式都可以带一个或多个标志（flags），用以表明正则表达式的行为。

正则表达式的匹配模式支持3个标志。

**g**:表示全局模式（global），即模式将被应用于所有字符串，而非在发现第一个匹配项时立即停止;

**i**:表示不区分大小写（case-insensitive）模式 ，即在正确匹配项时忽略模式与字符串的大小写;

**m**:表示多行（multiline）模式，即在到达一行文本末尾时还会继续查找下一行中是否存在与模式相匹配的项。模式中使用的所有元字符都必须转义

正则表达式中的元字符包括：

( [ { \ ^ $ | ) ? \* + . ] }

元字符在正则表达式中都有一或多种特殊用途，因此如果想要匹配字符串中包含的这些字符，就必须对它们进行转义.

使用RegExp构造函数可以创建正则表达式，它接收两个参数：一个是要匹配的字符串模式；一个是可选的标识字符串。

可以使用字面量定义的任何表达式，都可以使用构造函数来定义

传递给 RegExp 构造函数的两个参数都是字符串（不能把正则表达式字面量传递给 RegExp构造函数）。由于RegExp构造参数的模式是字符串，因此在某些情况下需要对字符进行转义。所有元字符都必须进行双重转义那些转义过的字符也是如此。

在ECMAScript 3 中**正则表达式字面量始终会共享同一个RegExp实例，而RegExp构造函数创建一个新RegExp实例都是一个新实例**。

ECMAScript 5 明确规定，使用正则表达式字面量必须像直接调用 RegExp 构造函数一样，每次都创建新的 RegExp实例。

### 3.4.1 RegExp 实例属性

RegExp的实例属性不会重置。

RegExp每个实例都具有属性

* + global：布尔值，表示是否设置了 g 标志。
  + ignoreCase：布尔值，表示是否设置了i 标志。
  + lastIndex：整数，表示开始搜索下一个匹配项的字符位置，从 0 算起。
  + multiline：布尔值，表示是否设置了 m 标志。
  + source：正则表达式的字符串表示，按照字面量形式而非传入构造函数中的字符串模式返回。

source属性保存的是规范形式的字符串，即字面量形式所用的字符串。

通过这些属性可以获知一个正则表达式的各方面信息，但却没有多大用处，因为这些信息全都包含在模式声明中。

var pattern1 = /\[bc\]at/i;

pattern1.global

### 3.4.2 实例方法

RegExp对象的主要方法是**exec()**，该方法是专门为捕获组而设计的。exec()接收一个参数，即要应用模式的字符串，返回包含第一个匹配项信息的数组，或者在没有匹配项的情况下返回null。该方法返回的虽然是Array数组的实例但包含有额外的属性：index和input。其中，index表示匹配项在字符串中的位置，而input表示应用正则表达式的字符串。在数组中，第一项是与整个模式匹配的字符串，其他项是与模式中的捕获组匹配的字符串（如果模式中没有捕获组，则该数组只包含一项）。

对于 exec()方法而言，即使在模式中设置了全局标志（g），它每次也只会返回一个匹配项。在不设置全局标志的情况下，在同一个字符串上多次调用 exec()将始终返回第一个匹配项的信息。而在设 置全局标志的情况下，每次调用 exec()则都会在字符串中继续查找新匹配项

全局模式下每次调用 exec()都会返回字符串中的下一个匹配项，直至搜索到字符串末尾为止。此外，还应该注意模式的lastIndex属性的变化情况。在全局匹配模式下，lastIndex 的值在每次调用exec()后都会增加，而在非全局模式下则始终保持不变

RegExp对象的第二个方法是test()方法，它接受一个字符串参数。在模式与该参数匹配的情况下返回 true；否则，返回 false。在只想知道目标字符串与某个模式是否匹配，但不需要知道其文本内容的情况下，使用这个方法非常方便

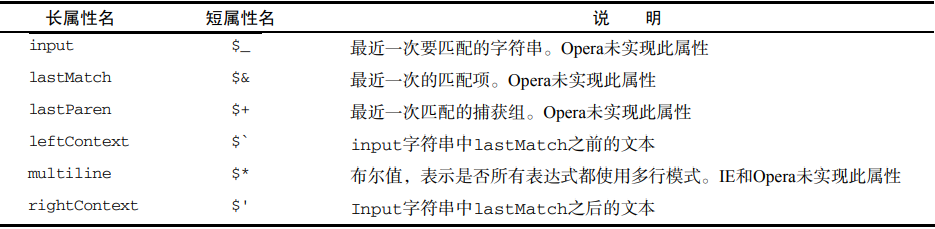
RegExp 实例继承的 toLocaleString()和 toString()方法都会返回正则表达式的字面量，与创 建正则表达式的方式无关。

正则表达式的 valueOf()方法返回正则表达式本身。

### 3.4.3 RegExp构造函数属性

RegExp 构造函数包含一些属性（这些属性在其他语言中被看成是静态属性）。这些属性适用于作用域中的所有正则表达式，并且基于所执行的最近一次正则表达式操作而变化。这些属性分别有一个长属性名和一个短属性名（Opera 是例外，它不支持短属性名）。

RegExp构造函数的属性



使用这些属性可以从 exec()或 test()执行的操作中提取出更具体的信息。

RegExp 构造函数的各个属性返回了下列值：

* + - input 属性返回了原始字符串；
    - leftContext 属性返回了单词 short 之前的字符串，而 rightContext 属性则返回了 short 之后的字符串；
    - lastMatch 属性返回最近一次与整个正则表达式匹配的字符串，即 short；
    - lastParen 属性返回最近一次匹配的捕获组，即例子中的 s。

如前所述，例子使用的长属性名都可以用相应的短属性名来代替。只不过，由于这些短属性名大都不是有效的 ECMAScript标识符，因此必须通过方括号语法来访问它们

还有多达 9 个用于存储捕获组的构造函数属性。访问这些属性的语 法是 RegExp.$1、RegExp.$2…RegExp.$9，分别用于存储第一、第二……第九个匹配的捕获组。在 调用 exec()或 test()方法时，这些属性会被自动填充。

### 3.4.4 模式的局限性

下面列出了 ECMAScript 正则表达式不支持的特性：

* + 匹配字符串开始和结尾的\A 和\Z 锚 ***注***：但支持以插入符号（^）和美元符号（$）

来匹配字符串的开始和结尾

* + 向后查找（lookbehind） ***注***：但完全支持向前查找（lookahead）。
  + 并集和交集类
  + 原子组（atomic grouping）
  + Unicode 支持（单个字符除外，如\uFFFF）
  + 命名的捕获组 ***注***：但支持编号的捕获组。
  + s（single，单行）和 x（free-spacing，无间隔）匹配模式
  + 条件匹配
  + 正则表达式注释

## 3.5 Function类型

Function 类可以表示开发者定义的任何函数。

用 Function 类直接创建函数的语法如下：

var function\_name = new function(arg1, arg2, ..., argN, function\_body)

每个arg都是一个参数，最后一个参数是函数主体（要执行的代码）。这些参数必须是字符串。

函数实际上是对象。每个函数都是Function类型的实例,而且都与其他引用类型一样具有属性和方法。由于函数都是对象，因此函数名实际上也是一个指向函数对象的指针，不会与某个函数绑定。创建函数的方法

1、函数通常是使用函数声明语法定义的。

function functionName(){}

2、可以通过函数表达式，即函数字面量表示

var functionName = function (){};

在使用函数表达式定义函数时，没有必要使用函数名,并且此种方式，注意函数末尾有一个分号，就像声明其他变量时一样

3. 使用 Function 构造函数。

Function 构造函数可以接收任意数量的参数，但最后一个参数始终都被看成是函数体，而前面的参数则枚举出了新函数的参数。

var sum = new Function("num1", "num2", "return num1 + num2"); // 不推荐

Function的构造函数可以传入任意数量的字符串实参，最后一个实参所表示的文本就是函数体,它可以包含任意的JavaScript语句，并以逗号分隔。如果构造函数不包含任何参数，只需要给构造函数传入一个字符串—即函数体即可。创建后可直接通过变量名调用

var s = sum(1,2)

由于函数名仅仅是指向函数的指针，因此函数名与包含对象指针的其他变量没有什么不同。换句话说，一个函数可能会有多个名字

**需要注意的是**：

Function()构造函数并不需要通过传入实参以指定函数名，Function()构造函数创建的像函数字面量一样的匿名函数。

Function构造函数允许JavaScript在运行时动态的创建并编译函数；

每次调用Function构造函数都会解析执行函数体，并创建新的函数对象；但这种方式创建会严重影响程序的性能；

Function构造函数创建的函数并不是使用词法作用域，函数体代码的编译总是在顶层函数执行。

**注意，使用不带圆括号的函数名是访问函数指针，而非调用函数。**

### 3.5.1 没有重载

在ECMAScript中当声明的函数中存在两个及以上相同的同名函数，后定义的同名函数会覆盖先定义的同名函数。

### 3.5.2 函数声明与函数表达式

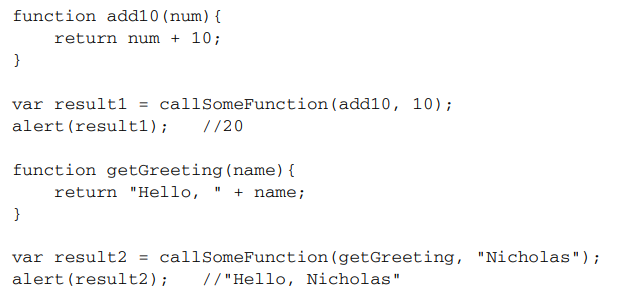
在ECMAScript中解析器会优先读取函数声明，并使其在任何代码执行之前可以被访问；函数表达式则必须执行到它所在的代码行才会被真正解析执行。

在代码开始执行之前，解析器就已经通过一个名为函数声明提升

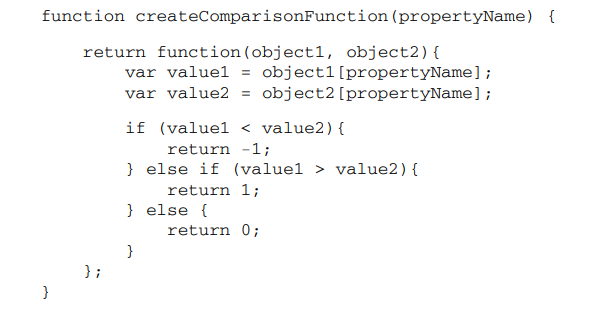
（function declaration hoisting）的过程，读取并将函数声明添加到执行环境中。对代码求值时，JavaScript引擎在第一遍会声明函数并将它们放到源代码树的顶部。所以，即使声明函数的代码在调用它的代码后面，JavaScript引擎也能把函数声明提升到顶部。而对于函数表达式，在未执行函数表达式之前不能调用函数表达式，否则会报错。

### 3.5.3 作为值的函数

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：



从一个函数中返回另一个函数：



### 3.5.4 函数内部属性

在函数内部有两个特殊的对象：arguments与this。

arguments对象是一个类数组对象，包含所有传入函数的参数，并且拥有一个callee的属性,该属性是一个指针，指向拥有这个arguments对象的函数，可以通过arguments.callee()获取函数名。通过arguments.callee方式能够解耦合。arguments对象的callee属性指向拥有这个arguments对象的函数。

this是函数内部的另一个特殊对象；this引用的是函数据以执行的环境对象——或者也可以说是this值（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）。

另一个函数对象的属性：caller。这个属性中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。

当函数在严格模式下运行时，访问arguments.callee会导致错误。ECMAScript 5 还定义了 arguments.caller属性，但在严格模式下访问它也会导致错误，而在非严格模式下这个属性始终是 undefined。定义这个属性是为了分清 arguments.caller和函数的 caller 属性。以上变化都是为了加强这门语言的安全性，这样第三方代码就不能在相同的环境里窥视其他代码了。 严格模式还有一个限制：不能为函数的 caller 属性赋值，否则会导致错误。

arguments 在严格模式与非严格模式下是不同的：非严格模式下，函数里的arguments仅仅是一个标识符；在严格模式下，arguments变成了保留字，函数无法使用arguments作为形参名或局部变量名，也不能给arguments赋值。

### 3.5.5 函数属性和方法

每个函数都包含两个属性：length 和 prototype。其中，length属性表示函数希望接收的命名参数的个数。

对于ECMAScript中的引用类型而言，prototype保存它们所有实例方法。在 ECMAScript 5中，prototype 属性是不可枚举的，因此使用 for-in 无法发现。

每个函数都包含两个非继承而来的方法：apply()和call()。这两个方法都是在特定的作用域中调用函数，实际上等于设置函数体内的this对象的值。call()方法与apply()方法的作用相同，它们的区别仅在于接收参数的方式不同。

apply() 接收两个参数：一个是在其中运行函数的作用域；一个是参数数组，该参数既可以是Array的实例也可以是arguments对象。

call() 方法，第一个是运行函数的作用域；其余参数直接传递给函数，即在使用call()方法时需要把传给函数的参数逐个列出来。

在严格模式下，未指定环境对象而调用函数，则 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。

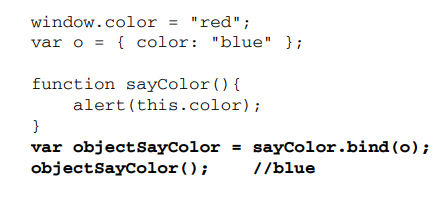
这两个方法能够扩充函数的作用域。使用call()（或apply()）来扩充作用域的最大好处，就是对象不需要与方法有任何耦合关系。

bind()方法 该方法会创建一个函数的实例，其this值会绑定到传给bind()函数的值。

对于apply()和call()方法，两者的作用相同，区别在于接收参数的方式不同：两者第一个参数都接收运行函数的作用域，apply()方法接收第二个参数为数组，为运行函数的形参参数组成的数组，可以是Array的实例，也可以是arguments对象；但call()方法接收的其余参数需列举出来。

bind()函数会创建一个函数的实例，并且this值会绑定到传给bind()函数的值；可以理解将方法绑定到某个对象并创建一个函数实例，并可以向方法传参，需要列举参数。

例如

将sayColor()绑定到对象o上并创建函数实例，也可以以列举方式向函数传参

### 3.5.6 自执行函数（作为全局命名空间的函数）

JavaScript的自执行函数 使用()表示自执行，（）只是表示自执行作用

**(function(){})()**; 是 把函数当作表达式解析，然后执行解析后的函数，与

相当于 var a = function(){}; a(); a得到的是函数

**(function(){}())**; 是把函数表达式和执行当作语句直接执行，与**!function(){}()** 和**+function(){}()**相同作用

相当于 var a = function(){}(); a得到的是结果

自执行函数避免用函数名污染全局命名空间

在多个文件中使用严格模式而不必在写很多行“use strict”，可以使用立即执行函数(尽可能少使用严格模式)

例如:

(function(){

“use strict”

function doSomething(){};

function doElseSomething(){};

}())

## 3.6 基本包装类型

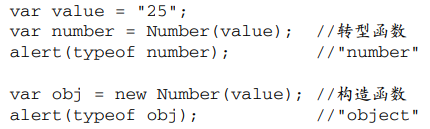
3个特殊的引用类型：Boolean, Number, String

每当读取一个基本类型值的时候，后台机会创建一个对应的基本包装类型的对象。基本类型是不具有属性和方法的，当我们调用基本类型时，实际是在后台创建了一个相应的基本包装类型的对象。

引用类型与基本包装类型的区别就在于对象的生存期。使用new操作符创建的引用类型的实例，在执行流离开当前作用域之前都一直保存在内存中。而自动创建的基本包装类型的对象，则只存在于一 行代码的执行瞬间，然后立即被销毁。这意味着我们不能在运行时为基本类型值添加属性和方法。

对基本包装类型的实例调用typeof会返回"object"，而且所有基本包装类型的对象都会被转换为布尔值 true。

要注意的是，使用new调用基本包装类型的构造函数，与直接调用同名的转型函数是不一样的，如：



### 3.6.1 Boolean类型

Boolean类型是与布尔值相对应的引用类型。要创建Boolean对象，可以使用new操作符，并传入true或false。

Boolean 类型的实例重写了valueOf()方法，返回基本类型值true 或false；重写了toString() 方法，返回字符串"true"和"false"。

最常见的问题就是在布尔表达式中使用 Boolean 对象，

基本类型与引用类型的布尔值有两个区别：一是typeof操作符对基本类型返回“boolean”,而对引用类型返回“Object”；二是Boolean对象是Boolean类型的实例所以使用instanceof操作符测试Boolean对象会返回true，而测试基本类型的布尔值则返回“false”。

### 3.6.2 Number类型

Number类型始于数字值对应的引用类型。要创建Number对象可以调用Number类型的构造方法并传入相应的数据项。如 var name = new Number(20) ;

Number类型也相应的继承了Object类型的一些方法，同时Number类型还提供了一些格式化数值为字符串的方法：

toFixed() 该方法会按照指定的小数位返回数值的字符串表示。如：var num = 10;

Number.toFixed（2）；--------10.00

如果数值中包含的小数位比指定的小数位多，则接近指定最大小数位的值就会被舍入。

toExponential() 该方法返回以指数形式表示的数值的字符串形式。该方法接收一个参数，用以表示指定输出数字的小数位。

toPrecision() 该方法可以返回固定大小格式，也可以返回指数形式；接收一个参数，表示数值的所有位数（不包括指数部分）。toPrecision()方法可以表现1到21位小数

实际上，toPrecision()会根据要处理的数值决定到底是调用toFixed()还是调用 toExponential()。 而这三个方法都可以通过向上或向下舍入，做到以最准确的形式来表示带有正确小数位的值。

### 3.6.3 String类型

String类型是字符串的基本包装类型。同样可以使用new操作符创建String对象。所有的String对象的方法也可以在所有字符串值中访问到。

String类型的每个实例都有一个length属性，表示字符串中包含的字符个数。即使字符串中包含双字节字符（不是占一个字节的ASCII 字符），每个字符也仍然算一个字符。

String类型

#### 1、字符方法：

charAt()与chartCodeAt()；这两个方法都接收一个参数，即基于0的字符位置。

其中，charAt()以单字符串形式（ECMAScript 中没有字符类型）返回指定位置上的字符

chartCodeAt()返回的是指定位置上字符的字符编码。

在一些浏览器中可以使用方括号加数字索引来访问字符串中的字符。

#### 2、字符串操作方法

concat() 方法拼接一个或多个字符串，并返回拼接后的字符串。可接收任意数量的参数。一般情况下，我们较多使用“+”连接符拼接字符串

slice()、substr()和 substring()。 这三个方法都会返回被操作字符串的一个子字符串，而且也都接受一或两个参数。第一个参数指定子字符串的开始位置，第二个参数（在指定的情况下）表示子字符串到哪里结束。具体来说，slice()和substring()的第二个参数指定的是子字符串最后一个字符后面的位置，即截取的子字符串不包括结束位置的字符。而 substr()的第二个参数指定的则是返回的字符个数。如果没有给这些方法传递第二个参数，则将字符串的长度作为结束位置。与 concat()方法一样，slice()、substr()和 substring()也不会修改字符串本身的值——它们只是返回一个基本类型的字符串值，对原始字符串没有任何影响。

在传递给这些方法的参数是负值的情况下，slice()方法会将传入的负值与字符串的长度相加，substr()方法将负的第一个参数加上字符串的长度，而将负的第二个参数转换为 0；最后，substring()方法会把所有负值参数都转换为 0。

对于substring()方法会将较小的数值作为起始位置，而将较大的位置作为结束位置。

#### 3、字符串位置方法

从字符串中查找子字符串的方法：indexOf()和 lastIndexOf()。

这两个方法都是从一个字符串中搜索给定的子字符串，然后返子字符串第一次出现的位置（如果没有找到该子字符串，则返回-1）。这两个方法的区别在于：indexOf()方法从字符串的开头向后搜索子字符串，而lastIndexOf()方法是从字符串的末尾向前搜索子字符串。

这两个方法都可以接收可选的第二个参数，表示从字符串中的哪个位置开始搜索。换句话说，indexOf()会从该参数指定的位置向后搜索，忽略该位置之前的所有字符；而 lastIndexOf()则会从指定的位置向前搜索，忽略该位置之后的所有字符。在使用第二个参数的情况下，可以通过循环调用 indexOf()或 lastIndexOf()来找到所有匹配的子字符串，即将每次搜索的字符位置索引传给下一次搜索的开始位置。

#### 4、trim()方法

该方法会返回一个字符串的副本，删除字符串前后的空格，返回结果。

#### 5、字符串大小写转换方法

涉及字符串大小写转换的方法有4个：toLowerCase()、toLocaleLowerCase()、toUpperCase()和 toLocaleUpperCase()。

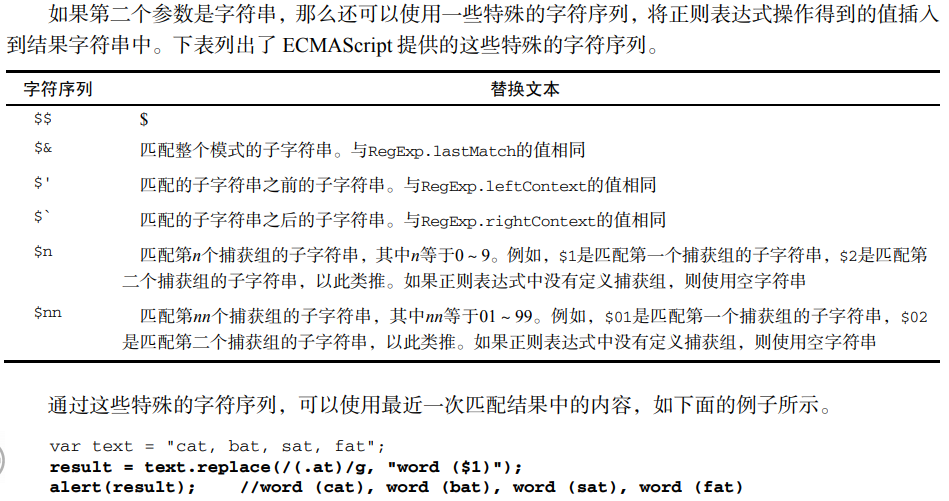
其中toLocaleUpperCase()与toLocaleLowerCase()是针对特定地区实现的。

一般来说，在不知道自己的代码将在哪种语言环境中运行的情况下，还是使用针对地区的方法更稳妥一些。

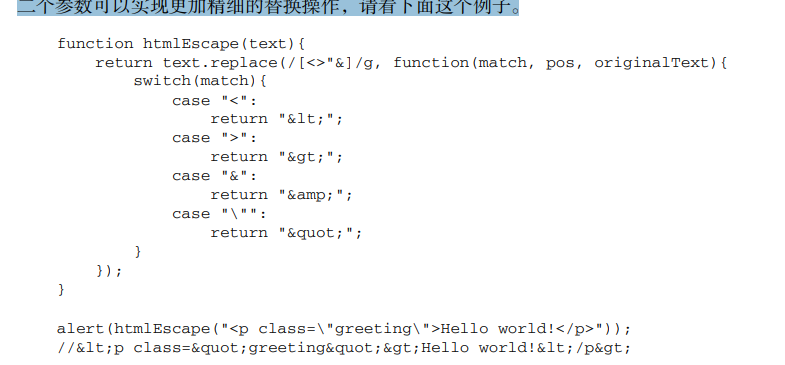
#### 6、字符串的模式匹配方法

match()方法，本质上与RegExp的exec()方法相同；该方法接收一个参数，可以是正则表达式字面量值，也可以是RegExp对象。match()方法返回了一个数组（该数组是由匹配项的值组成的），该数组与执行RegExp的exec()方法得到的数组一样：数组的第一项是与整个模式匹配的字符串，之后的每一项（如果有）保存着与正则表达式中的捕获组匹配的字符串。

search()方法，该方法的唯一参数与match()方法的参数相同：由字符串或者RegExp对象指定的一个正则表达式。search()方法返回字符串中第一个匹配项的索引；如果没有找到匹配项则返回-1.该方法始终从字符串开头向后查找。

replace()方法用于替换字符串。该方法可以接受两个参数：第一个参数可以是RegExp对象或者是字符串（该字符串不会被转换成正则表达式），第二个参数是一个字符串或者函数。如果该方法的第一个参数是字符串，那么就只会匹配替换第一个字符串；如果要替换所有的匹配项字符串，就需要提供一个正则表达式而且需要制定全局（g）标志。

*如果第二个参数是函数,*在只有一个匹配项（即与模式匹配的字符串）的情况下，会向这个函数传递三个参数：模式匹配项、模式匹配项在字符串中的位置和原始字符串。在正则表达式中定义了多个捕获组的情况下，传递给函数的参数依次是模式的匹配项、第一个捕获组的匹配项、第二个捕获组的匹配项……，但最后两个参数仍然分别是模式的匹配项在字符串中的位置和原始字符串。这个函数应该返回一个字符串，表示应该被替换的匹配项使用函数作为 replace()方法的第二个参数可以实现更加精细的替换操作，请看下面这个例子。



第二个参数中可以使用特殊字符 $ , 通过$1，$2,$n的形式匹配正则表达式的子表达式，对匹配的字符串进行定向匹配。

split()方法，可以基于给定的分隔符将字符串分割为多个子字符串，以数组的形式。分隔符可以是字符串，也可以是一个 RegExp 对象（这个方法不会将字符串看成正则表达式）。split()方法可以接受可选的第二个参数，用于指定数组的大小，以便确保返回的数组不会超过既定大小。

#### 7、localeCompare()方法

该方法比较两个字符串，并返回下列值中一个：

♥ 如果字符串在字母表中应该排在字符串参数之前，则返回一个负数（大多数情况下是-1，具体 的值要视实现而定）；

♥ 如果字符串等于字符串参数，则返回 0；

♥ 如果字符串在字母表中应该排在字符串参数之后，则返回一个正数（大多数情况下是1，具体的值同样要视实现而定）。

localeCompare()返回的数值取决于实现。

#### 8、fromCharCode()方法

该方法是String构造函数的静态方法。接收一个或多个字符编码，转换成字符串。从

本质上来看，这个方法与实例方法charCodeAt() 执行的是相反的操作。来看一个例子： alert(String.fromCharCode(104, 101, 108, 108, 111)); //"hello"

9、HTML方法

## 3.7 单体内置方法

对内置对象的定义是：“由ECMAScript实现提供的、不依赖于宿主环境的对象，这些对象在ECMAScript程序执行之前就已经存在了。”即不必显式地实例化内置对象。如：Object、Array、String、Global、Math。

### 3.7.1 Global对象

Global对象，即全局对象。ECMAScript 中的 Global 对象在某种意义上是作为一个终极的“兜底儿对象”来定义的。换句话说，不属于任何其他对象的属性和方法，最终都是它的属性和方法。事实上，没有全局变量和全局属性，所有在全局作用域定义的属性和方法都是Global对象的属性和方法。

#### 1.URI编码方法

Global对象的encodeURI()和encodeURIComponent()方法可以对URII（Uniform Resource Identifiers，通用资源标识符）进行编码。URI编码方法可以对URI进行编码用特殊的UTF-8编码替换所有的无效标志符。

encodeURI()方法主要对整个URI进行编码，而encodeURIComponent()主要对URI的某一段进行编码。两者区别在于： encodeURI()方法不会对本身属于URI的特殊字符如冒号、正斜杠、问号和井字号进行编码；而encodeURIComponent()会对所有发现的非标准字符进行编码。

encodeURIComponent()方法会使用对应的编码替换所有非字母数字字符

一般来说，我们使用encodeURIComponent()方法的时候要比使用encodeURI()更多，因为在实践中更常见的是对查询字符串参数而不是对基础URI进行编码。

与 encodeURI()和 encodeURIComponent()方法对应的两个方法分别是 decodeURI()和 decodeURIComponent()。其中，decodeURI()只能对使用encodeURI()替换的字符进行解码。decodeURIComponent()能够解码使用encodeURIComponent()编码的所有字符，即它可以解码任何特殊字符的编码。

URI 方法 encodeURI()、encodeURIComponent()、decodeURI()和 decodeURIComponent()用于替代已经被ECMA-262第3版废弃的escape()和unescape() 方法。URI方法能够编码所有 Unicode字符，而原来的方法只能正确地编码 ASCII 字符。 因此在开发实践中，特别是在产品级的代码中，一定要使用 URI方法，不要使用 escape() 和 unescape()方法。

#### 2. eval()方法

eval()方法就像是一个完整的ECMAScript解析器，它只接受一个参数，即要执行的 ECMAScript（或JavaScript）字符串,即eval()方法只接收一个参数，要执行的ECMAScript或JavaScript代码。

JS解析器会将eval()方法解析出来的代码放回到原来的位置，被执行的代码具有与该执行执行环境相同的作用域链。。这意味着通过 eval()执行的代码可以引用在包含环境中定义的变量。在eval()中创建任何变量或函数都不会被提升，因为它们包含在一个字符串中，他们只能在eval()方法执行的时候创建。

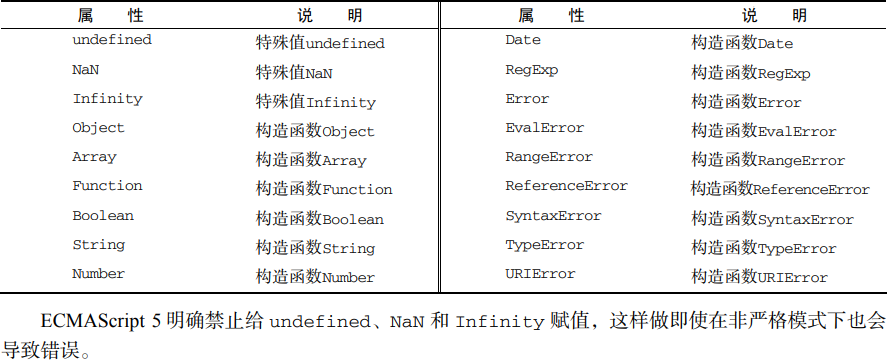
在严格模式下，外部访问不到eval()中创建的任何变量或函数。

同时该方法由于能够执行代码字符串，可能会引起代码注入

#### 3. Global对象的属性

如undefined，NaN、inifinty都是Global对象的属性。所有原生引用类型的构造函数，如Function，Object，都是Global对象的属性。

Global对象的全部属性：



#### 4.window对象

在全局作用域中声明的所有变量和函数，都是window对象的属性。

在没有给函数明确指定this值的情况下（无论是通过将函数添加为对象的方法，还是通过调用call()或apply()），this 值等于 Global 对象。而像这样通过简单地返回this 来取得Global对象，在任何执行环境下都是可行的。

JavaScript中window对象与Global对象的区别：

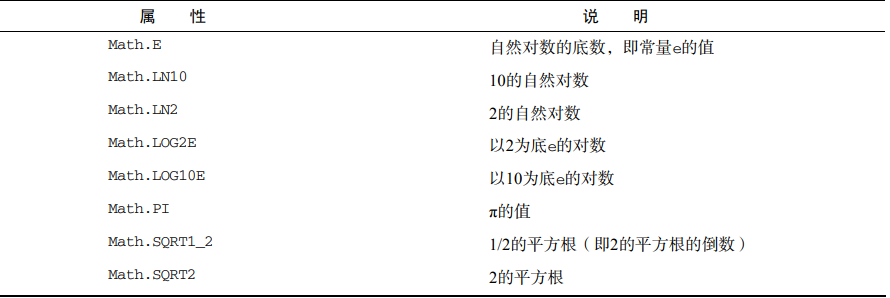
1. global是javascript运行时所在宿主环境提供的全局对象
2. window对象是浏览器的一个web api,可以说是global在浏览器中的具体表现
3. global对象是单体内置对象，即不依赖宿主环境的对象，而window对象依赖浏览器。

### Math对象

该对象保存了数学公式和信息。

#### 1.Math对象的属性

Math对象常用的属性：



#### 2. min()和 max()方法

其中，min()和 max()方法用于确定一组数值中的最小值和最大值。这两个方法都可以接收任意多 个数值参数，

这两个方法经常用于避免多 余的循环和在 if 语句中确定一组数的最大值。 要找到数组中的最大或最小值，可以像下面这样使用 apply()方法

var values = [1, 2, 3, 4, 5, 6, 7, 8];

var max = Math.max.apply(Math, values); 这个技巧的关键是把 Math 对象作为 apply()的第一个参数，从而正确地设置 this 值。然后，可 以将任何数组作为第二个参数。

#### 3．舍入方法

Math.ceil() 执行向上舍入，即总是将数值向上取最接近的整数;

Math.floor() 执行向下舍入，即总是将数值向下去接近的整数;

Math.round()执行标准舍入，四舍五入

#### 4.random()方法

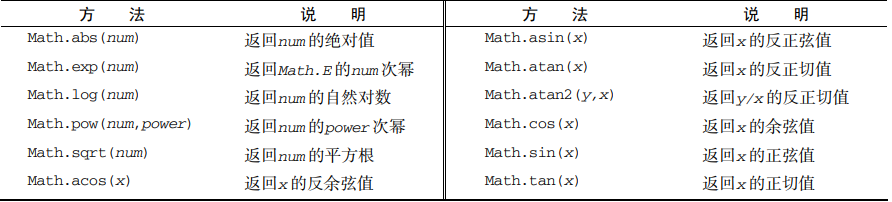
Math.random()方法返回一个大于0小于1的随机数。

利用 Math.random()从某个整数范围内随机选择一个值。

值 = Math.floor(Math.random() \* 可能值的总数 + 第一个可能的值)

即：MinValue + Math.random()\*(MaxValue-MinValue + 1)

#### 5.其他方法

Math对象的其他数学方法

## 3.8 小结 \*\*\*\*\*

对象在JavaScript中被称为引用类型的值，并且有一些内置的引用类型可以用来创建特定的对象。

1. 引用类型与传统面向对象程序设计中的类相似，但实现不同；
2. Object是基础类型，其他所有类型都从Object继承了基本行为；
3. Array类型是一组值的有序列表，同时还提供了操作和转换值的功能；
4. Date类型提供了有关日期和时间的信息，包括当前时间和日期及相关计算功能；
5. RegExp类型是ECMAScript中支持正则表达式的一个接口，提供了最基本的和一些高级的正则表达式功能；

函数实际上是Function类型的实例，so函数也是对象，因此函数也具有一些方法，来增强其行为。

因为有了基本包装类型，所以JavaScript中基本类型值可以被当作对象来访问。三种基本包装类型：Boolean，String，Number,他们有共同的特征：

* 每个基本包装类型都映射带同名的基本类型
* 在读取模式下访问基本类型值时，就会创建相对应的基本包装类型的一个对象，方便数据操作；
* 操作基本类型值的语句一旦执行完毕，就会立即销毁创建的包装对象

在所有代码执行之前，作用域中就已经存在两个内置对象：Global和Math。在大多ECMAScript中都不能直接访问Global对象；不过web浏览器实现了承担该角色的window对象。全局变量和函数都是Global对象的属性，

# 四、面向对象的程序设计

面向对象（Object-Oriented）的语言有一个共同的标志，他们都有 类 的概念。通过类可以创建任意多个具有相同属性和方法从的对象。ECMAScript中没有 类 的概念。

ECMA-262中把对象定义为“无序属性的集合，其属性可以包含基本值、对象、函数”。即对象是一组没有特定顺序的值对象的每个属性或方法都有名字，而每个名字都映射到一个值。，我们可以把 ECMAScript 的对象想象成散列表：无 非就是一组名值对，其中值可以是数据或函数。

每个对象都是基于引用类型创建的，引用类型可以是原生类型也可以是自定义引用类型。

## 4.1理解对象

创建自定义对象最简单的方式就是创建一个Object的实例，然后再为它添加属性和方法。创建对象可参考（[三、引用类型 3.1Object类型）](#_3.1_Object_类型)。

也可以通过 对象字面量 创建对象。

### 4.1.1、属性类型 ------待完善

特性是实现JavaScript引擎用的，在JavaScript中不能直接访问它们。为表示特性是内部值，将它们放置在两对方括号中，如[[Enumerable]]。

ECMAScript有两种属性：数据属性和访问器属性。

#### 1、数据属性

数据属性包含一个数据值的位置。在这个位置可以读取和写入值。数据属性有4 个可以

描述其行为的特性：

* + [[Configurable]] 表示是否能够通过delete删除属性从而重新定义属性；能否修改属性的特性，或者能否把属性修改为访问器属性。能够直接在对象上定义属性，它们的这个特性默认为true。
  + [[Enumerable]] 表示能否通过for-in循环返回属性。该属性默认为true；
  + [[Writable]] 表示能否修改属性的值。默认为true；
  + [[Value]] 包含这个属性的数据值，读取属性值时，从这个位置读；写入属性值时把新值保存在这个位置；该特性值默认为undefined；

直接在对象上定义的属性，它们的[[Configurable]]、[[Enumerable]] 和[[Writable]]特性都被设置为 true，而[[Value]]特性被设置为指定的值

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

要更改属性默认的特性，必须使用ECMAScript 5中的Object.defineProperty()方法。该方法接收三个参数：属性所在的对象，属性的名字和一个描述符对象。其中，描述符（descriptor）对象的属性必须是：configurable、enumerable、writable 和 value。设置其中的一或多个值，可以修改对应的特性值。例如：

var person = {};

Object.defineProperty(person, "name", { writable: false, value: "Nicholas" }); alert(person.name); //"Nicholas"

person.name = "Greg";

alert(person.name); //"Nicholas"

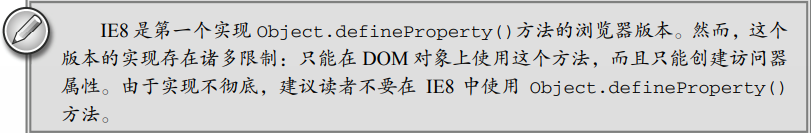
上例创建了一个名为 name 的属性，它的值"Nicholas"是只读的。这个属性的值是不可修改 的，如果尝试为它指定新值，则在非严格模式下，赋值操作将被忽略；在严格模式下，赋值操作将会导 致抛出错误。

表示当属性是只读的情况下，属性的值是不可修改的。

configurable 设置为 false，表示不能从对象中删除属性。如果对这个属性调用 delete，则 在非严格模式下什么也不会发生，而在严格模式下会导致错误。而且，一旦把属性定义为不可配置的， 就不能再把它变回可配置了。

可以多次调用 Object.defineProperty()方法修改同一个属性，但在把 configurable 特性设置为 false 之后就会有限制了。

在调用 Object.defineProperty()方法时，如果不指定，configurable、enumerable 和 writable 特性的默认值都是 false。多数情况下，可能都没有必要利用 Object.defineProperty() 方法提供的这些高级功能

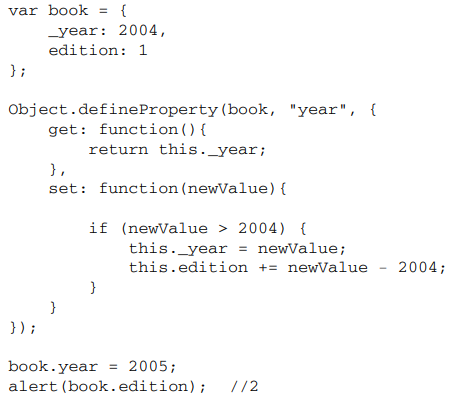


#### 2、访问器属性

访问器属性不包含数据值。它们包含一对setter和getter函数。在读取访问器属性时，会调用getter函数，负责返回有效的值；在写入访问器属性时会调用setter函数并传入新值，此函数负责如何处理数据。访问器属性有如下4个特性：

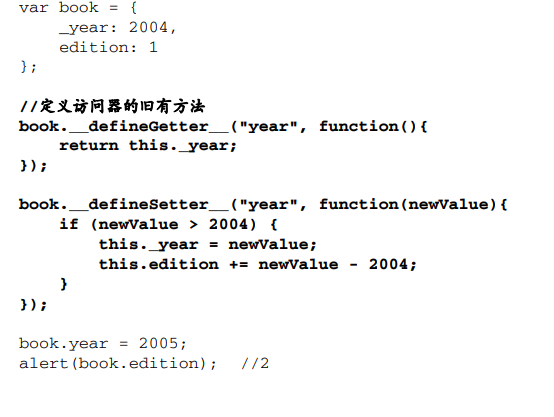
* [[Configurable]]：表示能否通过 delete 删除属性从而重新定义属性，能否修改属性的特 性，或者能否把属性修改为数据属性。对于直接在对象上定义的属性，这个特性的默认值为 true。
* [[Enumerable]]：表示能否通过 for-in 循环返回属性。对于直接在对象上定义的属性，这 个特性的默认值为 true。
* [[Get]]：在读取属性时调用的函数。默认值为 undefined。
* [[Set]]：在写入属性时调用的函数。默认值为 undefined。

访问器属性不能直接定义，必须使用 Object.defineProperty()来定义。



以上代码创建了一个 book 对象，并给它定义两个默认的属性：\_year 和 edition。\_year 前面 的下划线是一种常用的记号，用于表示只能通过对象方法访问的属性。而访问器属性 year 则包含一个 getter 函数和一个 setter 函数。getter 函数返回\_year 的值，setter 函数通过计算来确定正确的版本。因此， 把 year 属性修改为 2005 会导致\_year 变成 2005，而 edition 变为 2。这是使用访问器属性的常见方 式，即设置一个属性的值会导致其他属性发生变化。

不一定非要同时指定 getter 和 setter。只指定 getter 意味着属性是不能写，尝试写入属性会被忽略。 在严格模式下，尝试写入只指定了 getter 函数的属性会抛出错误。类似地，只指定 setter 函数的属性也 不能读，否则在非严格模式下会返回 undefined，而在严格模式下会抛出错误。 支持 ECMAScript 5 的这个方法的浏览器有 IE9+（IE8 只是部分实现）、Firefox 4+、Safari 5+、Opera 12+ 和 Chrome 。在这个方法之前，要创建访问器属性，一般都使用两个非标准的方法： \_\_defineGetter\_\_()和\_\_defineSetter\_\_()。这两个方法最初是由 Firefox 引入的，后来 Safari 3、 Chrome 1 和 Opera 9.5 也给出了相同的实现。使用这两个遗留的方法，可以像下面这样重写前面的例子。



在不支持 Object.defineProperty() 方法的浏览器中不能修改 [[Configurable]] 和 [[Enumerable]]。

### 4.1.2、读取属性的特性（待续）

使用 ECMAScript 5 的 Object.getOwnPropertyDescriptor()方法，可以取得给定属性的描述 符。这个方法接收两个参数：属性所在的对象和要读取其描述符的属性名称。返回值是一个对象，如果 是访问器属性，这个对象的属性有 configurable、enumerable、get 和 set；如果是数据属性，这 个对象的属性有 configurable、enumerable、writable 和 value。

### 4.1.3、对象属性的获取

可以通过点（.）或方括号（[ ]）运算符来获取对象的属性。运算符左侧应当是一个表达式，返回一个对象。对于点运算符来说，右侧必须是一个以属性名称命名的简单标识符，该标识符必须直接出现在JavaScript程序中，它们不是数据类型程序无法修改它们；对于方括号运算符来说，方括号内必须是一个计算结果为字符串的表达式，这个字符串就是属性的名称，程序运行时依旧可以修改该字符串表达式。

因此，当我们需要在程序运行时，修改对象属性时，可以使用方括号来访问对象的属性。**字符串值是动态的，可在运行时修改；标识符是静态的，必须写死在程序中**。

当我们获取对象属性时，首先应该判断该属性是否存在对象中，假如不存在，程序可能会报错。

### 4.1.4、删除对象属性

delete运算符可以删除对象属性。Delete运算符只是断开对象与属性的宿主关系，而不会操作属性的。注意的是：delete删除的只能是对象的自有属性，而不能删除继承属性。

同时delete运算符也不能删除那些可配置属性为false的属性。

### 4.1.5、检测对象属性

可以通过in运算符、hasOwnProperty()、propertyIsEnumerable()方法来检测属性是否存在。

有一种情景只能使用in运算符，即in可以区分不存在的属性和存在但值为undefined的属性。使用“!==”可以区分undefined与null

hasOwnProperty()用于检测是否是对象的自有属性，对于继承属性将返回false；

propertyIsEnumerable()方法是hasOwnProperty()方法的增强版，只有当检测到自有属性并且该属性的可枚举属性为true才返回true。

### 4.1.6、对象的类属性

对象的类属性是一个字符串，用以表示对象的类型信息。默认的toString()方法（继承自Object.prototype）返回的类属性形式：[object class]。要想获取对象的类，可以调用对象的toString方法，然后提取返回字符串的第8个到倒数第二个位置之间的字符。但一般可以通过Object.prototype.toString.call();方法来获取[object class],进而获取类class属性。

### 4.1.7、对象的可拓展性

对象的可拓展性表示是否可以给对象添加属性。所有的内置对象和自定义对象都是显式可拓展的。宿主对象的可拓展性是由JavaScript引擎定义的。

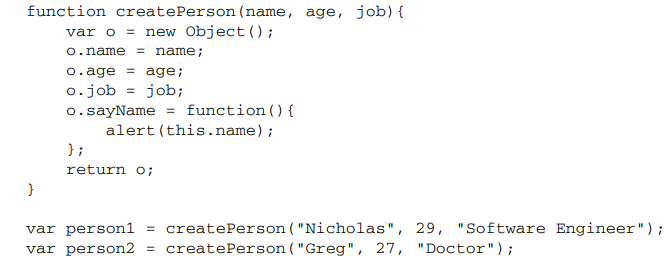
对对象的修改，可[参见第22章第一节第四小节尊重对象所有权](#_1._尊重对象所有权)

## 4.2 创建对象

创建对象的方式：详见第三章第一节

### 4.2.1 工厂模式

工厂模式抽象了创建具体对象的过程。用函数封装用特定接口创建对象的细节。例如：



### 4.2.2 构造函数模式

构造函数可以用来创建特定类型的函数。除了ECMAScript原生构造函数，还可以自定义构造函数。

按照惯例构造函数始终都应该以一个大写字母开头；而非构造函数则应该以一个小写字母开头。通过构造函数创建对象必须使用new操作符，。以这种方式调用构造函数实际上会经历以下 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 就指向了这个新对象）； (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。。

1.将构造函数当做函数

构造函数和其他函数的区别在于调用他们方式不同。任何函数，只要通过new操作符来调用，都可以被看做构造函数。

当在全局作用域中调用一个函数时，this 对象总是指向 Global 对象（在 浏览器中就是 window 对象）。

2.构造函数问题

使用构造函数的主要问题，就是每个方法都要在实例上重新创建一遍。因为函数也是对象，因此每定义一个函数也就实例化一个对象，可以将对象的函数转移到构造函数外部

### 4.2.3 原型模式

每一个函数都有一个prototype属性，该属性是个指针，指向一个对象，而这个对象的用途是包含可以由特定类型的所有实例共享的属性和方法。prototype就是通过调用构造函数而创建的那个对象实例的原型对象。原型可以让所有对象实例共享包含它的属性和方法。即，可以不必在构造函数中定义对象实例的信息，而是将这些对象信息添加到原型中。

构造函数模式是将对象信息在构造函数中添加；而原型模式将对象实例的属性方法绑定到原型上。

要理解原型模式的工作原理，则必须先了解原型对象的性质：

#### 1.原型对象

每一个构造函数创建对象时，都会自动创建一个prototype属性，该属性指向函数的原型对象。默认情况下，所有原型对象都会自动获得一个constructor（构造函数）属性，这个属性包含一个指向prototype属性所在函数的指针。通过这个构造函数，还可以继续为原型对象添加属性和方法。

创建了自定义的构造函数后，函数的原型对象会默认只会取得constructor属性，而其他方法都是从Object对象继承来的。当调用构造函数创建对象实例后，该实例的内部将包含一个指针（内部属性），指向构造函数的原型对象。原型对象在ECMA-262 第 5 版中被称为[[Prototype]][[1]](#footnote-0)。但在脚本中没有访问标准方法访问[[Prototype]]，但在Firefox、Safari 和 Chrome浏览器中对象支持一个属性--\_\_proto\_\_;但在其他实现中，这个属性对脚本则完全不可见。这个连接存在于实例与构造函数的原型对象之间，而不是存在于实例与构造函数之间。

虽然所有实现都无法访问到[[Prototype]],但可以通过isPrototypeOf()方法来确定对象之间是否存在关系。从本质上讲，如果[[Prototype]]指向调用 isPrototypeOf()方法的对象 （Person.prototype），那么这个方法就返回 true。

isPrototype()方法是原型对象的方法。

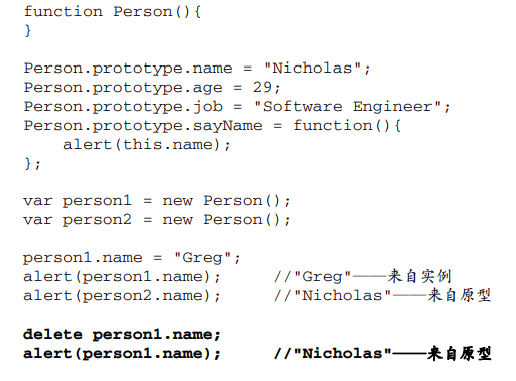
ECMAScript 5 中添加了一个新方法：Object.getPrototypeOf();在所有支持的实现中，这个方法返回[[Prototype]]的值。该方法的参数是实例对象。

使用 Object.getPrototypeOf() 可以方便地取得一个对象的原型，而这在利用原型实现继承（本章稍后会讨论）的情况下是非常重要的。 支持这个方法的浏览器有 IE9+、Firefox 3.5+、Safari 5+、Opera 12+和 Chrome。

每当代码读取某个对象的某个属性时，都会执行一次搜索，目标是具有给定名字的属性。搜索首先 从对象实例本身开始。如果在实例中找到了具有给定名字的属性，则返回该属性的值；如果没有找到， 则继续搜索指针指向的原型对象，在原型对象中查找具有给定名字的属性。如果在原型对象中找到了这个属性，则返回该属性的值。也就是说，在我们调用person1.sayName()的时候，会先后执行两次搜索。首先，解析器会问：“实例person1有sayName属性吗？”答：“没有。”然后，它继续搜索，再问：“person1的原型有sayName属性吗？”答：“有。”于是，它就读取那个保存在原型对象中的函数。当我们调用person2.sayName()时，将会重现相同的搜索过程，得到相同的结果。而这正是多个对象实例共享原型所保存的属性和方法的基本原理。

原型最初只包含constructor属性，而该属性也是共享的，因此可以通过对象实例访问。

* 可以通过原型对象访问保存在实例对象中值，但不能通过原型对象重写原型中的值。
* 如果在实例中添加一个属性，而新增的属性名与原型中的一个属性名相同，那么就在该实例中创建属性，将屏蔽原型中的同名属性。

当为对象实例添加一个属性时，该属性将会屏蔽原型中的同名属性；也就是说添加的属性会阻止访问原型中的同名属性，但不会修改哪个同名属性。即使将这个属性设置为null，也只会在实例中设置这个属性，而不会恢复其指向原型的连接。不过，使用delete操作符则可以完全删除实例属性，从而让我们能够重新访问原型中的属性。

使用hasOwnProperty()方法可以检测一个属性是存在于实例中，还是存在于原型中。这个方法（不要忘了它是从Object继承来的）只在给定属性存在于对象实例中时，才会返回true。

ECMAScript5的Object.getOwnPropertyDescriptor()方法只能用于实例属性，要取得原型属性的描述符，必须直接在原型对象上调用Object.getOwnPropertyDescriptor()方法。

#### 2.原型与in操作符

有两种方式使用in操作符：单独使用和在for-in循环中使用。

单独使用时，in操作符会在通过对象能够访问给定属性时返回true，无论该属性存在于实例还是原型中。

* 同时使用hasOwnProperty()方法和in操作符，就可以确定该属性到底是存在于对象中，还是存在于原型中

由于in操作符只要能够通过对象访问到属性时就返回true，hasOwnProperty()只在属性存在于实例中才会返回true；只要in操作符返回true而hasOwnProperty()方法返回false就能够确定属性存在于原型中。

hasPrototypeProperty()方法判断属性是否存在与原型中，是返回true，反之返回false。我们应该知道，实例对象中的属性会屏蔽原型中的同名属性，但不会修改属性。

在使用for-in循环时，返回的是所有能够通过对象访问的、可枚举的属性，既包括实例中的属性有包括原型中的属性。屏蔽了原型中的不可枚举属性（即将[[Enumerable]]标记为false的属性）的实例属性也会在for-in循环中返回。为根据规定，所有开发人员定义的属性都是可枚举的——只有在IE8及更早版本中例外。

IE早期版本的实现中存在一个bug，即屏蔽不可枚举属性的实例属性不会出现在for-in循环中。

* 要取得对象上所有可枚举的实例属性，可以使用ECMAScript5Object.keys()方法。这个方法接收一个对象作为参数，返回一个包含所有可枚举属性的字符串数组。
* 如果你想要得到所有实例属性,无论它是否可枚举,都可以使用**Object.getOwnPropertyNames()方法**。不可枚举的constructor属性也会被获取。

#### 3.更简单的原型语法

用一个包含所有属性和方法的对象字面量来重写整个原型对象，如：

functionPerson(){}；

Person.prototype= {

name : "Nicholas",

age : 29,

job: "Software Engineer",

sayName : function () {

alert(this.name);

}

};

将Person.prototype设置为等于一个以对象字面量形式创建的新对象，最终结果相同。但有一个例外：constructor属性不再指向Person了。前面曾经介绍过，每创建一个函数，就会同时创建它的prototype对象，这个对象也会自动获得constructor属性。

而我们在这里使用的语法，本质上完全重写了默认的prototype对象，因此constructor属性也就变成了新对象的constructor属性（指向Object构造函数），不再指向Person函数。此时，尽管instanceof操作符还能返回正确的结果，但通过constructor已经无法确定对象的类型了

用instanceof操作符测试Object和Person仍然返回true，但constructor属性则等于Object而不等于Person了。如果constructor的值真的很重要，可以像下面这样特意将它设置回适当的值。

function Person(){ }

Person.prototype = {

constructor : Person,

name : "Nicholas",

age : 29,

job: "Software Engineer",

sayName : function () {

alert(this.name);

}

};

注意，以这种方式重设constructor属性会导致它的[[Enumerable]]特性被设置为true。默认情况下，原生的constructor属性是不可枚举的，因此如果你使用兼容ECMAScript5的JavaScript引擎，可以试一试Object.defineProperty()。

//重设构造函数，只适用于ECMAScript5兼容的浏览器Object.defineProperty(Person.prototype, "constructor", {

enumerable: false,

value: Person

});

#### 原型的动态性

由于在原型中查找值是一次搜索。我们对原型对象上所做的任何修改都可以立即从实例

上反映出来。即使是先创建了实例后修改原型也一样。

实例与原型之间的松散连接关系，实例与原型之间的连接只不过是一个指针，而非一个副本

尽管可以随时为原型添加属性和方法，并且修改能够立即在所有对象实例中反映出来，但如果是重写整个原型对象，那么情况就不一样了。我们知道，调用构造函数时会为实例添加一个指向最初原型的[[Prototype]]指针，而把原型修改为另外一个对象就等于切断了构造函数与最初原型之间的联系。

请记住：实例中的指针仅指向原型，而不指向构造函数。

重写原型对象切断了现有原型与任何之前已经存在的对象实例之间的联系；它们引用的仍然是最初的原型。

#### 5.原生对象的原型

所有的原生引用类型都在其构造函数的原型上定义了方法。

通过原生对象的原型，不仅可以取得所有默认方法的引用，而且可以定义新方法。可以像修改自定义对象的原型一样修改原生对象的原型，随时添加方法。

#### 6.原生对象的问题

原型模式省略了为构造函数初始化参数，所有属性在相同情况下都取得相同的属性值。原型模式的最大问题是由其共享的本性所导致的。

原型中的属性被很多实例共享，一般不会单独使用原型模式。

### 4.2.4 组合使用构造函数模式和原型模式.

创建自定义类型的最常见方式就是组合使用构造函数模式和原型模式。构造函数模式用于定义实例属性，原型模式用于定义方法和共享的属性。

实例属性通过构造函数创建，方法和共享的属性由原型模式创建。

### 4.3.5 动态原型模式

动态原型模式，把所有信息都封装在了构造函数中，而通过在构造函数中初始化原型（仅在必要的情况下），可以通过检查某个应该存在的方法是否有效，来决定是否应该初始化原型。例如：

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

Person.prototype.sayName = function(){

alert(this.name);

};

}

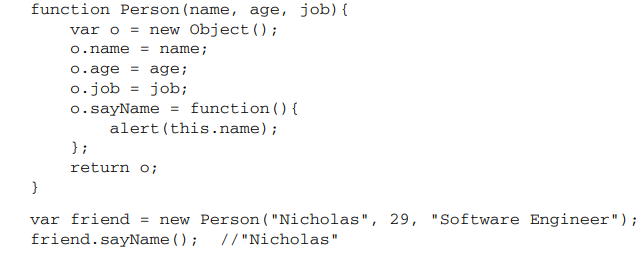
其中，if 语句检查的可以是初始化之后应该存在的任何属性或方法——不必用一大堆

if 语句检查每个属性和每个方法；只要检查其中一个即可。对于采用这种模式创建的对象，还可以使用 instanceof 操作符确定它的类型。

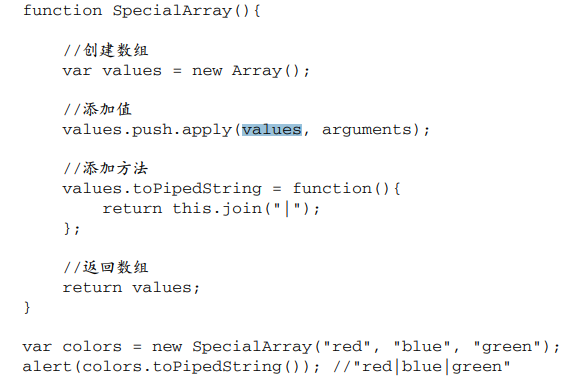
### 4.3.6 寄生构造函数模式

在前几种模式不适用的情况下可以使用寄生构造函数模式。

该模式的基本思想是，创建一个函数，该函数仅仅用于封装创建对象的代码，再返回新创建的对象。例如：



构造函数在不返回值的情况下，默认会返回新创建的对象实例，而通过在构造函数的末尾添加一个return语句，可以重写调用构造函数时返回的值。

这个模式可以在特殊的情况下用来为对象创建构造函数。假设我们想创建一个具有额外方法的特殊数组。由于不能直接修改Array构造函数，因此可以使用这个模式。

对于寄生构造函数模式，首先，返回的对象与构造函数或与构造函数的原型的属性无关；即构造函数返回的对象与在构造函数外部创建的对象一样，因此不能依赖于instanceof操作符来确定对象类型。由于存在上述问题，我们建议在可以使用其他模式的情况下，不要使用这种模式。

### 4.3.6 稳妥构造函数模式

道格拉斯·克罗克福德（Douglas Crockford）发明了JavaScript中的稳妥对象（durable objects）这个概念。稳妥对象，指的是没有公共属性，而且其方法也不引用this的对象。稳妥对象最适合在一些安全的环境中（这些环境会禁止使用this和new），或者在防止数据被其他应用程序改动时使用。稳妥构造函数遵循与寄生构造函数类似的模式，区别在于，一是创建对象的实例方法不能使用this；二是不使用new操作符来调用构造函数。

按照稳妥构造函数的要求，可以将前面的 Person 构造函数重写如下。

function Person(name, age, job){

//创建要返回的对象

var o = new Object();

//可以在这里定义私有变量和函数

//添加方法

o.sayName = function(){

alert(name);

};

//返回对象

return o;

}

注意，在以这种模式创建的对象中，除了使用 sayName()方法之外，没有其他办法访问name 的值。可以像下面使用稳妥的 Person 构造函数。

var friend = Person("Nicholas", 29, "Software Engineer");

friend.sayName(); //"Nicholas"

这样，变量 friend 中保存的是一个稳妥对象，而除了调用 sayName()方法外，没有别的方式可 以访问其数据成员。即使有其他代码会给这个对象添加方法或数据成员，但也不可能有别的办法访问传 入到构造函数中的原始数据。稳妥构造函数模式提供的这种安全性，使得它非常适合在某些安全执行环境——例如，ADsafe（www.adsafe.org）和 Caja（http://code.google.com/p/google-caja/）提供的环境—— 下使用。

与寄生构造函数模式类似，使用稳妥构造函数模式创建的对象与构造函数之间也 没有什么关系，因此 instanceof 操作符对这种对象也没有意义。

## 4.3 继承

一般OO语言都支持两种继承方式：接口继承和实现继承。其中，接口继承只继承方法签名；实现继承则继承实际的方法。由于ECMAScript中函数没有签名，所以无法实现接口继承；ECMAScript中只支持实现继承，且实现继承的主要是依靠原型链来实现的。

### 4.3.1 原型链

ECMAScript中只支持实现继承，而实现继承主要以靠原型链实现。

基本思想：利用原型让一个引用类型继承另外一个引用类型的属性和方法。

构造函数、原型和实例的关系：

每个构造函数都有一个prototype属性，该属性是一个指针，指向原型对象，原型对象包含一个constructor属性，也是一个指针，指向构造函数的指针，而实例都包含一个指向原型对象的内部指针。一个原型对象等于另外一个类型的实例，构成原型链。

原型属性是在实例对象创建之初就设置好的。通过对象字面量创建的对象的原型使用Object.prototype作为它们的原型。通过new操作符和构造函数创建的对象使用构造函数的prototype属性作为它们的原型。通过Object.create()创建的对象使用第一个参数（也就是null）作为它们的原型。

原型链的本质实际上是重写原型对象，代之以一个新类型的实例。可以理解，原型链是将一个构造函数的原型对象指向另一个新类型的实例，继承新类型的属性和方法；同样新类型的原型对象可以由其他类型的实例替换，继而形成原型链。通过实例对象可以获取被继承对象的属性和方法。当实现继承后，继承者将重写原型对象的constructor属性，指向被继承者。实际上是将继承者的原型指向被继承者的原型

通过实现原型链，实际上扩大了原型搜索机制。即，当以读取模式访问实例属性时，首先会在实例对象中搜索该属性，假如实例中不存在该属性则会在原型对象中搜索该属性。而当实现原型链后，搜索会沿着原型链继续向上搜索，直至到原型链的末端

*function* SuperPeople(){

*this*.property = true;

}

SuperPeople.prototype.getSuperValue = *function*(){

return *this*.property;

}

*function* SubPeople(){

*this*.property = false;

}

SubPeople.prototype = new SuperPeople();

SubPeople.prototype.getSubValue = *function*(){

return *this*.property;}}

}

*var* instance = new SubPeople();

如上例，继承者会获得被继承着的属性和方法，并覆盖同名属性和方法，并重写继承者SubPeople的constructor属性使其指向继承者的构造函数，

1.别忘记默认的原型

所有的引用类型都继承了Object类型，这种继承也是通过原型链实现的。所有函数的默认原型都是Object类型的实例，因此所有默认原型都包含一个内部指针-----指向Object.prototype原型对象。SubType 继承了 SuperType，而 SuperType 继承了 Object。当调用 instance.toString()
时，实际上调用的是保存在 Object.prototype 中的那个方法。

2.确定原型和实例的关系

确定原型与实例之间的关系？

一是使用instanceof操作符，只要测试实例与原型链中出现过的实例就会返回true；

alert(instance instanceof Object); //true

alert(instance instanceof SuperPeople); //true

alert(instance instanceof SubPeople); //true

二是 使用isPrototypeOf()方法，同样只要是测试在原型链上出现过的实例都会返回true；

alert(Object.prototype.isPrototypeOf(instance)); //true alert(SuperPeople.prototype.isPrototypeOf(instance)); //true alert(SubPeople.prototype.isPrototypeOf(instance)); //true

3.谨慎的定义方法

子类有时候需要重写父类的有关方法，或者添加父类不存在的某个方法，但给原型添加方法的代码一定要放在替换原型的语句之后

在通过原型链实现继承时，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

//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;

}

};

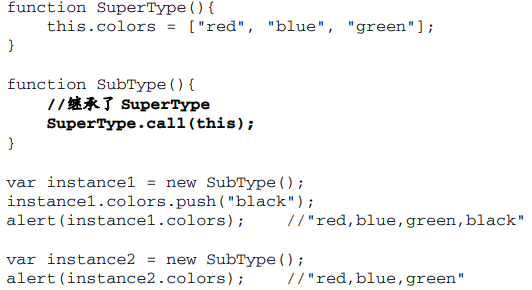
1. 原型链问题

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。想必大家还记得，我们前面介绍过包含引用类型值的原型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，而不是在原型对象中定义属性的原因。在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。

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。实际上， 应该说是没有办法在不影响所有对象实例的情况下，给超类型的构造函数传递参数。有鉴于此，再加上 前面刚刚讨论过的由于原型中包含引用类型值所带来的问题，实践中很少会单独使用原型链。

### 4.3.2 借用构造函数

借用构造函数的技术，又称为伪造对象或经典继承。基本思想是，在子类型构造函数内部调用父类的构造函数。

通过使用 apply()和 call()方法也可以在（将来）新创建的对象上执行构造函数，实现继承。

通过使用 call()方法（或 apply()方法 也可以），我们实际上是在（未来将要）新创建的 SubType 实例的环境下调用了 SuperType 构造函数。在新 SubType 对象上执行 SuperType()函数中定义的所有对象初始化代码。SubType 的每个实例就都会具有自己的 colors 属性的副本了。

#### 1. 传递参数

相对于原型链而言，借用构造函数有个优势，即可以在子类构造函数中向超类构造函数传参-----使用apply()或call()方法。

#### 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。而且，在超类型的原型中定义的方法，对子类型而言也是不可见的，结 果所有类型都只能使用构造函数模式。考虑到这些问题，借用构造函数的技术也是很少单独使用的。

### 4.3.3 组合继承

组合继承，也成为伪经典继承，是指将原型链和借用构造函数组合使用*实现继承*。

基本思想：**使用原型链实现对原型属性和方法的继承，使用借用构造函数实现对实例属性的继承**

例：

*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

### 4.3.4 原型式继承

道格拉斯·克罗克福德：借助原型可以基于已有的对象创建新对象

function object(o){

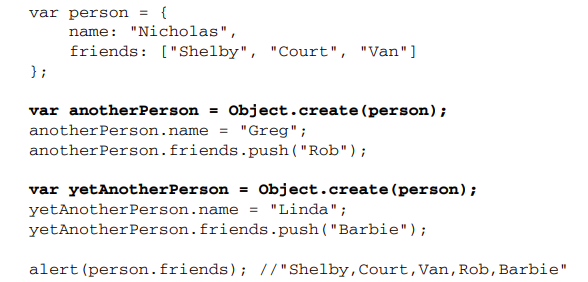
function F(){}；

F.prototype = o; //重构原型对象

return new F();

}

ECMAScript 5 新增了Object.create()方法进而规范原型式继承，该方法接收两个参数：一个用作新对象原型的对象，一个用作为新对象增加额外属性的对象（第二个可选）。



Object.create()方法的第二个参数与Object.defineProperties()方法的第二个参数格式相同：每个属性都是通过自己的描述符定义的。以这种方式指定的任何属性都会覆盖原型对象上的同名属 性。

var anotherPerson = Object.create(person,

{ name: {

value: "Greg"

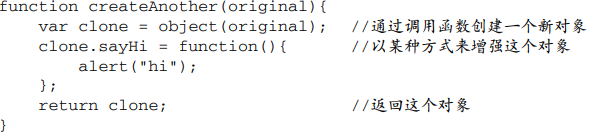
}

});

包含引用类型值的属性始终都会共享相应的值，就像使用原型模式一样。

### 4.3.5寄生式继承

寄生式继承同样由克罗克福德推广。寄生式继承与计生构造函数和工厂模式类似，即创建一个仅用于封装继承过程的函数，该函数在内部以某种方式增强函数，最后返回对象。如：



createAnother()函数接收了一个参数，也就是将要作为新对象基础的对象

考虑对象而不是自定义类型和构造函数的情况下，寄生式继承也是一种有用的模式。前面示 范继承模式时使用的 object()函数不是必需的；任何能够返回新对象的函数都适用于此模式。

### 4.3.6 寄生组合式继承

组合继承的问题在于无论在什么情况下都会调用两次超类型的构造函数：**一次在创建子类原型时，一次在子类构造函数内部**。子类型最终会包含超类型对象的全部实例属性但不得不在调用子类构造函数时重写属性。

为解决此问题，

寄生组合式继承 通过借用构造函数来继承属性，通过原型链的混合形式来继承方法。

其基本思路是：不必为了指定子类的原型而调用超类型的构造方法，只需要超类的一个副本。

本质上，通过寄生式继承来继承超类型的原型，再讲结果指定给子类型的原型。

寄生组合式继承的基本模式如下所示。

function inheritPrototype(subType, superType){

var prototype = object(superType.prototype); //创建对象，超类原型的一个副本 prototype.constructor = subType; //增强对象

subType.prototype = prototype; //指定对象

}

可以看做，将超类型原型的副本指定给子类型的原型

inheritPrototype()函数实现了寄生组合式继承的最简单形式。

这个函数接收两个参数：子类型构造函数和超类型构造函数。在函数内部，第一步是创建超类型原型的一个副本。第二步是为创建的副本添加 constructor 属性，从而弥补因重写原型而失去的默认的 constructor 属性。 最后一步，将新创建的对象（即副本）赋值给子类型的原型。

这个例子的高效率体现在它只调用了一次 SuperType 构造函数，并且因此避免了在 SubType. prototype 上面创建不必要的、多余的属性。与此同时，原型链还能保持不变；因此，还能够正常使用 instanceof 和 isPrototypeOf()。开发人员普遍认为寄生组合式继承是引用类型最理想的继承范式。

## 4.4 小结

ECMAScript支持OO编程，但不支持类或接口，对象可以在代码执行过程中创建并增强，因此具有动态性而非严格定义的实体。在没有类的情况下，采用下列模式创建对象：

* + 工厂模式，使用简单函数创建对象，为对象添加属性和方法，然后返回对象。后被构造函数模式所代替。
  + 构造函数模式，可以创建自定义引用类型，可以像创建内置对象实例一样使用new操作符但它的每个成员都无法得到复用，包括函数。函数与对象具有松散耦合特点
  + 原型模式，使用构造函数的prototype属性来指定那些应该共享的属性或方法。组合使用原型模式和构造函数模式时，使用构造函数定义实例属性，使用原型定义共享属性和方法。

JavaScript主要通过原型链来实现继承，原型链的构建是通过将一个实例对象赋值给另一个类型的原型来实现的。因此，子类就可以访问父类型的属性和方法。原型链的问题在于对象实例共享所有继承的属性和方法，不适宜单独使用。解决这个问题就是借用构造函数，即在子类的构造函数内部调用父类的构造函数，这样每个实例都具有自己的属性，同时还能保证只使用构造函数模式来定义类型。使用最多的继承就是组合继承，即通过原型链定义共享的属性和方法，通过构造函数定义实例属性。

其他继承模式：

* 原型式继承：可以在不必预先定义构造函数的情况下实现继承，本质是执行对给定对象的浅复制。该副本可得到增强。
* 寄生式继承：与原型式继承相似，基于某个对象或信息创建对象，增强对象并返回对象。效率较低可与组合继承一起使用。
* 寄生组合式继承：，集寄生式继承和组合继承的优点与一身，是实现基于类型继承的最有效方式。

# 五、函数表达式

函数的定义有两种方式：1.使用函数表达式

2.使用函数声明

函数声明的语法为：function FunctionName(args){//函数体}

Firefox、Safari、Chrome 和 Opera 都给函数定义了一个非标准的 name 属性，通过这个属性可以获取给定的函数名。这个属性值永远等候跟在function关键字后面的标识符。

函数声明的一个重要特性便是函数声明提升，即在代码执行之前会首先读取函数声明，所以无论函数声明的位置在什么位置，都会首先读取函数声明。

函数表达式：常用语法

Var functionName = function(args){

//函数体

}

这种形式可以看做创建一个函数并赋值给一个变量，创建的函数称为匿名函数，匿名函数的name属性是空字符串

函数表达式和其他表达式一样必须先赋再在使用。

能够创建函数再赋值给变量，也就能够把函数作为其他函数的值返回。

## 5.1 递归

递归函数是在一个函数通过名字调用自身的情况下创建的。例如：

arguments的callee属性是个指针，指向拥有arguments属性的函数，可以使用arguments.callee来代替函数名。确保调用的安全性。

function doMath(args){

if (args <= 1) {

return 1;

} else {

return args\*arguments.callee(args-1);

}

}

在严格模式下，不能通过脚本访问arguments.callee，访问这个属性会导致错误。不过，可以使用命名函数表达式来达成相同的结果。

创建了一个名为f()的命名函数表达式，然后将它赋值给变量 factorial。即便把函数赋值给了另一个变量，函数的名字 f 仍然有效，所以递归调用照样能正确完成。这种方式在严格模式和 非严格模式下都行得通。

*var* factorial = (*function* f(*num*){

if (num <= 1){

return 1;

} else {

return num \* f(num-1);

}

});

## 5.2 闭包

闭包，是有权访问另一个作用域中变量的函数。

创建闭包常用的方式是，在一个函数内部创建另一个函数。

当某个函数被调用时，会首先创建一个执行环境及相应的作用域链。然后，使用 arguments 和其他命名参数的值来初始化函数的活动对象（activation object）。但在作用域链中，外部函数的活动对象始终处于第二位，外部函数的外部函数的活动对象处于第三位，……直至作为作用域链终点的全局执行环境。

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；局部环境的变量对象只有在函数执行的过程中存在。

作用域链本质上是一个指向变量指针的指针列表，只引用但不实际包含变量对象。

无论在什么时候在函数中访问一个变量，都会从作用域链搜索相对应名的变量。一般来说，当函数执行完毕后，局部活动对象就会销毁，内存中仅保存全局作用域（全局执行环境的变量对象）。

在另一个函数内部定义的函数会将包含外部函数的活动对象添加到它的作用域链中。

一般，我们将一个对象设为null，来解除引用进行垃圾回收。

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内存。过度使用闭包会导致占用内存过多。

### 5.2.1 闭包与变量

在作用域链中，闭包只能获取包含函数中的任一变量的最后一个值。闭包保存的是整个变量，而不是某个特殊变量。

*function* createFunctions() {

*var* result = new Array();

for(*var* i = 0; i < 10; i++) {

result[i] = *function*() {

return i;

};

}

return result;

}

这个函数会返回一个函数数组。表面上看，似乎每个函数都应该返自己的索引值，即位置 0 的函数返回 0，位置 1 的函数返回 1，以此类推。但实际上，每个函数都返回 10。因为每个函数的作用域链中 都保存着createFunctions() 函数的活动对象，所以它们引用的都是同一个变量 i 。 当createFunctions()函数返回后，变量 i 的值是 10，此时每个函数都引用着保存变量 i 的同一个变量对象，所以在每个函数内部 i 的值都是 10。

我们可以通过创建另一个匿名函数强制让闭包的行为 符合预期，

*function* createFunctions(){

*var* result = new Array();

for (*var* i=0; i < 10; i++){

result[i] = *function*(*num*){

return *function*(){

return num;

} ;

}(i);

}

return result;

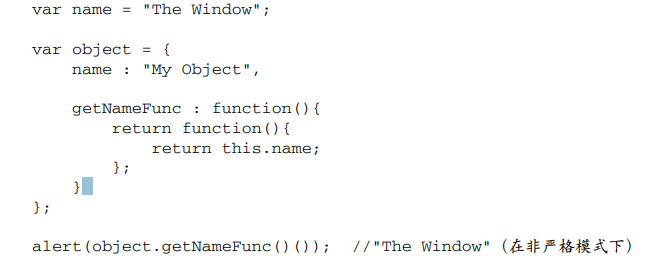
}

在重写了前面的 createFunctions()函数后，每个函数就会返回各自不同的索引值了。在这个版 本中，我们没有直接把闭包赋值给数组，而是定义了一个匿名函数，并将立即执行该匿名函数的结果赋 给数组。这里的匿名函数有一个参数 num，也就是最终的函数要返回的值。在调用每个匿名函数时，我 们传入了变量 i。由于函数参数是按值传递的，所以就会将变量 i 的当前值复制给参数 num。而在这个 匿名函数内部，又创建并返回了一个访问 num 的闭包。这样一来，result 数组中的每个函数都有自己 num 变量的一个副本，因此就可以返回各自不同的数值了。

### 5.2.2 关于this对象

在闭包中使用this可能会导致一些问题。this对象是基于运行时函数的运行环境绑定的:在全局中this等于window；当函数被当做对象的某个方法执行时，this指向当前对象。

匿名函数的执行环境具有全局性，其this通常指向window。但在通过 call()或 apply()改变函数执行环境的情况下，this 就会指向其他对象。

这个例子中的最终返回的是name = The Window，即全局的name属性。

当返回为函数时，不会自动执行，可转换为自执行函数。

上例为什么this指向的是window全局，而不是当前函数？

每个函数被调用时都会自动取得两个特殊变量：this和arguments对象。内部函数在搜索这两个变量时，只会搜索到活动对象为止，永远不可能直接访问外部函数中的this和arguments。但可以把外部作用域的this对象保存到一个闭包能够访问到变量里。

this 和 arguments 也存在同样的问题。如果想访问作用域中的 arguments 对 象，必须将对该对象的引用保存到另一个闭包能够访问的变量中。

内部函数无法直接访问到外部函数的this和arguments对象。

### 5.2.3 内存泄漏

IE9之前对Jscript和COM对象执行不同的垃圾收集例程.如果闭包保存着一个HTML元素,那么这个元素将无法被销毁

*function* assignHandler(){

*var element* = *document*.getElementById("someElement");

*element*.onclick = *function*(){

alert(*element*.id);

};

}

上例中由于在匿名函数中始终保存对element的引用只要匿名函数存在element的引用数就至少为1,内存因此永远无法被收回.

我们可以将*element*.id赋给一个变量，但不会消除内存泄漏。

闭包保存着包含函数的整个活动对象。为了便于进行垃圾回收，应当手动将不再使用的变量设置为null.

## 5.3 模仿块级作用域

JavaScript中没有块级作用域概念。因此在块语句中创建的变量实际包含在函数中而非语句中。即，在函数语句块中创建的变量可以在函数中语句块之外使用。若想重新使用同变量名则需要重新定义变量名。

因此，匿名函数可以用来模拟块级作用域。

用作块级作用域（通常称为私有作用域）的匿名函数语法如下：

**(function(){**

**//这里是块级作用域**

**})();**

上面语法可以：将一个函数表达式用小括号包含，紧跟一个小括号，表示这个函数会被立即执行。未使用小括号包含的函数表达式会报错，因为JavaScript会认为是一个函数声明，函数声明后面不可以使用小括号，而函数表达式可以，为防止解析错误，使用小括号包含函数表达式。

在自执行函数中定义的变量无法在自执行函数之外调用，否则会报错。因为在自执行函数中的变量在函数执行完毕会自动销毁，那么变量便不存在。

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。 一般来说，我们都应该尽量少向全局作用域中添加变量和函数。在一个由很多开发人员共同参与的大型应用程序中，过多的全局变量和函数很容易导致命名冲突。而通过创建私有作用域，每个开发人员既可以使用自己的变量，又不必担心搞乱全局作用域。

自执行函数可以减少闭包占用内存的问题，因为没有指向匿名函数的引用，只需要函数执行完毕，就可以立即销毁其作用域链。

## 5.4 私有变量

严格讲，JavaScript中没有私有成员的概念，所有对象属性都是共有的。但有一个私有变量的概念任何在函数中定义的变量都可以认为是私有变量，因为不能再函数外部调用。

私有变量 包括函数参数、局部变量和在函数内部定义的其他函数。

在函数外部不能访问函数的私有变量，但可以通过闭包的形式去访问私有变量。闭包可以访问包含函数的所有变量。

特权方法：把有权访问私有变量和私有函数的公有方法。

在对象上创建特权方法的方式有两种:

1.在构造函数中定义特权方法，模式如：

*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();

};

}

该模式在构造函数内部定义了私有变量和函数，继而又创建了能够访问这些私有变量的特权方法。之所有能够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。

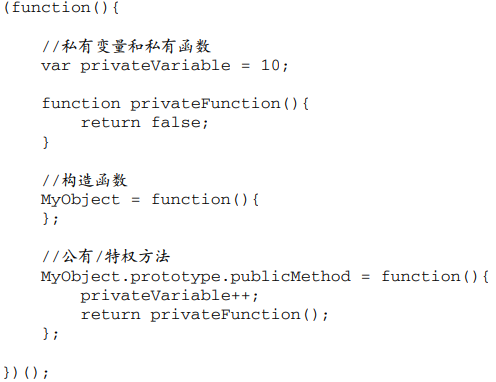
利用私有和特权成员，可以隐藏一些不应该被修改的数据。

该模式将闭包做为特权方法并赋给函数内变量名，

构造函数定义特权方法，必须使用构造函数模式。但构造函数模式会针对每个实例创建相同的方法，静态私有变量可以解决这个问题。

### 5.4.1 静态私有变量

通过在私有作用域中创建私有变量或函数，也可以创建特权方法。模式：



在这个模式中，创建了一个私有作用域，并封装了一个构造函数和相对应的方法，在私有作用域中首先定义私有变量和函数，然后创建了构造函数和公有方法。公有方法是以原型模式创建的。同时定义构造函数时，并没有使用函数声明而是使用了函数表达式，因为函数声明只能创建局部函数，所以使用函数表达式且未使用var操作符。因为：初始化未经声明的变量时，总会创建一个全局变量。因此，构造函数能够被外部访问。但严格模式下，给未经声明的变量赋值会报错。

该模式与在构造函数中定义特权方法的区别在于：私有变量和函数是由实例所共享的。

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。

以这种方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。到底是使用实例变量，还是静态私有变量，最终还是要视你的具体需求而定。

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。而这正是使用 闭包和私有变量的一个显明的不足之处

### 5.4.2 模块模式

上述模式是为自定义类型创建私有变量和特权方法的。而道格拉斯的模块模式是为单例创建私有变量和特权方法。

单例，指的是只有一个实例的对象。按照惯例，JavaScript是以对象字面量的方式创建单例对象的。语法：

var singleton = {

name : value,

method : function(){

//代码

}

}

模式模块通过为单例添加私有变量和特权方法可使其增强。

*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();

}

};

}();

在这个模式中使用了返回了一个对象的匿名函数。在这个匿名函数内部，首先定义了私有变量和函数，然后将一个对象字面量作为匿名函数的返回值返回。而返回的对象字面量只包含公有属性和方法。因为这个对象字面量是在函数内部定义的，所以公有方法有权访问包含函数的私有变量和函数 从本质上而言，这个对象字面量定义的是单例的公共接口。

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有用的。

如果必须创建一个对象并以某些数据对其进行初始化，同时还要公开一些方法能够访问这些私有数据的方法，可使用模块模式。 以模块模式创建的单例都是Object的实例，因最终要通过一个对象字面量来表示它。

单例通常都是作为全局对象存在的，

### 5.4.3 增强的模块模式

即，在返回对象之前加入对其增强的代码。增强的模块模式适用于那些单例必须是某种类型的实例，同时还必须添加某些属性和方法对其进行增强的情况。例如：

*var* singleton = *function*(){

//私有变量和私有函数

*var* privateVariable = 10;

*function* privateFunction(){

return false;

}

//创建对象

*var object* = new CustomType(); ---必须是某种类型的实例

//添加特权/公有属性和方法

*object*.publicProperty = true;

*object*.publicMethod = *function*(){

privateVariable++;

return privateFunction();

};

//返回这个对象

return *object*;

}();

## 5.5 小结

在JavaScript中，使用函数表达式无须对函数命名，从而实现动态编程。匿名函数，也称拉姆达表达式lambda。总结函数表达式的特点：

* 函数表达式不同于函数声明 函数声明需要命名，但函数表达式无须命名，没有名字的函数表达式也成为匿名函数；
* 在无法确定如何引用函数的情况下，递归函数会变得复杂；
* 递归函数应该始终使用arguments.callee来递归调用自身，不要使用函数名，因为函数名可能会发生变化；

当函数内部定义了其他函数就形成了闭包。闭包有权访问另一个作用域（包含函数）中变量，原理如下：

* 在后台执行环境中，闭包的作用域链包含着它自己的作用域、包含函数的作用域以及全局作用域;
* 通常，函数的作用域及其所有变量都会在函数执行完毕后结束；
* 当一个函数返回闭包时，这个函数会一直保存在内存中，直至闭包不再存在

使用闭包可以在JavaScript中模仿块级作用域（因为JavaScript中没有块级作用域的概念），要点如下：

* 创建并立即调用一个函数，这样既可以执行其中的代码，又不会在内存中留下对函数的引用；
* 结果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（即外 部作用域）中的变量。

闭包可以在对象中创建私有变量：

* JavaScript中没有私有对象属性的概念，但可以通过闭包来实现公有方法，并通过公有方法访问包含作用域中的变量;
* 有权访问私有变量的方法，称为特权方法;
* 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，也可以用模块模式、增强模块模式来实现单例的特权方法;

函数表达式和闭包有极其重要的作用，但创建闭包必须维护额外的作用域，过度使用他们会导致内存大量占用。

# 六、 BOM

ECMAScript是JavaScript的核心，但在web中使用JavaScript，那么BOM（浏览器对象模型）是核心。W3C把JavaScript中最基本的部分规范化。

## 6.1 window对象

BOM的核心对象是window，它代表浏览器的一个实例。window对象有着双重角色，它既是JavaScript访问浏览器窗口的接口，又是ECMAScript规定的Global对象。因此，这就意味着在网页中定义的任何一个对象、变量和函数都以window作为其Global对象，因而可以访问parseInt()方法。

### 6.1.1 全局作用域

由于window同时扮演着ECMAScript中的Global对象，在全局作用域中声明的变量、函数都会变成window对象的属性和方法。

除了全局变量都会成为window对象的属性，定义全局变量和在window对象上直接定义属性的区别：全局变量不能通过delete操作符删除，而直接在window对象上定义的属性可以使用delete删除。

delete 操作符：

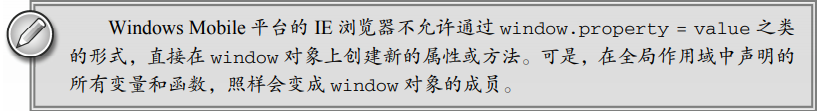
delete操作符是一元运算符，用来删除数组元素或对象属性。Delete操作符是由副作用的，它是用来删除的，不是用来返回值的。

没有使用var声明的变量，可以使用delete操作符删除；而使用**var**操作符声明的变量不能通过delete删除。（才使用 var 语句添加的 window 属性有一个名为[[Configurable]]的特性，这个特性的值被 设置为false，因此这样定义的属性不可以通过delete操作符删除。）

通过function语句定义的函数和函数参数也不能删除

使用delete删除属性时，删除的是实例属性；但不会删除prototype上的属性，如果要删除prototype上的属性需使用delete Object.prototyp.name 的方式。

访问一个未声明的变量会抛错，但可以通过查询window对象，来判断某个可能未声明的变量是否存在。使用window.property的方式，当property未声明不存在时，window.property会显示为undefined。



### 6.1.2 窗口关系及框架

如果页面中包含框架，那么每个框架都拥有自己的window对象，并且保存在frames集合中。在frames集合中，可以通过数值索引（从0开始，从左向右，从上至下）或者通过框架名称来访问相对应的window对象。每个window对象都有一个name属性，其中包含框架的名称。

# 七、客户端检测

# 八、DOM

# 九、DOM拓展

# 十、DOM2和DOM3

# 十一、事件

# 十二、表单脚本

# 十三、使用Canvas绘图

# 十四、HTML5脚本编程

# 十五、错误处理与调试

# 十六、JavaScript与xml

# 十七、E4X

# 十八、JSON

JSON是JavaScript的一个严格的子集，表示一种结构化数据。是一种数据格式，而不是编程语言。

## 18.1语法

JSON语法可以表示为三种类型的值：

* 简单值：使用与JavaScript相同的语法，可以在JSON中表示字符、数值、布尔值或null，但JSON不支持JavaScript中的undefined；
* 对象：是一种复杂的数据类型，表示的是一组无序的键值对，每个键值对的值可以是简单值，也可以是复杂数据类型；
* 数组：一种复杂数据类型，表示一组有序的键值对，可以通过索引来访问值。数组的值可以是任意类型---简单值、数组或对象。

JSON不支持变量、函数或实例对象，其就是一种结构化数据的格式。

### 18.1.1简单值

最简单的JSON形式就是简单值。简单值可以由 字符串、数值、布尔值或null表示。

JSON字符串与JavaScript字符串最大的区别在于：JSON字符串必须使用双引号否则会报错。

布尔值和null也是有效的JSON形式。

### 18.1.2 对象

与在JavaScript中创建对象字面有所不同的是，JSON中创建对象需要给JSON对象的键值对中的键加上双引号。如：

{

"name": "Nicholas",

"age": 29

}

JSON对象中没有声明变量，也没有使用分号结尾。在JSON对象中的键必须使用双引号，且键具有唯一性；值可以是简单值，也可以是复杂的数据类型。

与JavaScript不同，在JSON对象中属性名必须使用加双引号。否则为错。

### 18.1.3 数组

JSON数据格式中的另一种复杂数据结构。JSON数组采用的就是JavaScript中的数组字面量形式。*语法形式*：

[25, "hi", true]

在JSON数组中也不声明变量，不使用分号。对象与数组可以结合使用。

数组和对象通常是JSON数据格式的最外层形式（非强制）。

## 18.2解析与序列化

在JavaScript中可以将JSON数据结构解析为JavaScript对象。与XML 相比，在JavaScript中JSON数据更容易获取。

序列化是指将JavaScript数据转换成JSON字符串；

解析是指将JSON字符串转换成相对应的JavaScript值。

### 18.2.1 JOSN对象

早期的JSON解析器基本上使用的是JavaScript中的eval()函数。由于JSON是JavaScript的子集，因此eval()函数可以解析、解释并返回JavaScript对象和数组。ECMAScript 5 对解析JSON的行为进行了规范，定义了全局对象JSON。因为使用eval()函数存在风险，可能执行恶意代码。

JSON对象有两个方法：stringify()与parse();在最简单的情况下，这两个方法分别用于把 JavaScript 对象序列化为 JSON 字符串和把 JSON 字符串解析为原生 JavaScript 值。

**stringify()**将JavaScript对象序列化为一个JSON字符串。默认情况下，JSON.stringify（）输出的JSON字符串不包含任何空格字符或缩进。在对JavaScript对象进行序列化时，所有的函数以及原型成员都会被有意忽略；此外，值为undefined的任何属性也会被跳过，即最终都是值为有效JSON数据类型的实例属性。

**JSON.parse()**方法用于将JSON字符串转换成相对应的JavaScript值;如果传给 JSON.parse()的字符串不是有效的 JSON数据格式，该方法会抛出错误。

### 18.2.2 序列化选项

**JSON.stringify() 除了序列化JavaScript对象外，还可以接受两个参数，这两个参数指定以不同的方法序列化JavaScript对象；**第一个参数是过滤器，可以是一个数组，也可以是个函数；第二个参数是个选项，表示是否在JSON字符串中保留缩进。这两个参数可以单独或组合使用。

#### ****1.过滤结果****

如果过滤器参数是数组，那么JSON.stringify()的输出结果中只包含数组中列出的属性，即，通过将数组属性与键值对的键比较进行过滤，最终输出只包含在数组中的键值对。例如：

*var* book = {

"title": "Professional JavaScript",

"authors": [

"Nicholas C. Zakas"

],

edition: 3,

year: 2011

};

*var* jsonText = JSON.stringify(book, ["title", "edition"]);

上例中第二个参数为数组，数组中的属性与将要序列化的对象中的属性是对应的。所以，输出的结果中只会包含对应的属性：{"title":"Professional JavaScript","edition":3}。

如果传入的第二个参数为函数，则传入的函数接收两个参数：属性名与属性值（键值对）；根据属性名可以知道应该如何处理要序列化的对象中的属性。属性名只能是字符串，而在值并非键值对结构的值时，键名可以为空字符串。 通过传入的第一个参数为函数，可以对JSON中的键值对通过键名来相对应修改值，对于不需要修改的键的值，使用default项来返回原值，当返回值为undefined时，输出会忽略键值对。

为改变序列化对象的结果，函数返回的值就应该是相对应键的值。但如果函数返回了undefined，那么相应的属性就会被忽略。如例子：

*var* book = {

"title": "Professional JavaScript",

"authors": [

"Nicholas C. Zakas"

],

edition: 3,

year: 2011

};

*var* jsonText = JSON.stringify(book, *function*(*key*, *value*){

switch(*key*){

case "authors":

return value.join(",")

case "year":

return 5000;

case "edition":

return undefined;

default:

return value;

}

});

上例可以看出，函数过滤器根据传入的键决定结果。通过例子可以看出，case后的常量属性为键值对结构中的键，通过switch—case语句来修改键对应的值，对于不需要修改的键值属性，通过default项返回，即在函数过滤器中default项是必须有的，以便其他键对应的属性能够出现在结果中。实际上第一次调用函数过滤器传入的是个空字符串，值为book对象。上例结果为：

{"title":"Professional JavaScript","authors":"Nicholas C. Zakas","year":5000}

要序列化的对象中的每一个对象都要经过过滤器，因此数组中的每个带有这些属性的对象经过过滤之后，每个对象都只会包含"title"、"authors"和"year"属性。

#### 2.字符串缩进

**JSON.stringify()方法的第三个参数用于控制输出结果中的*空白符和缩进*。**

如果第三个参数为数值，表示每个级别缩进的空格数；同时，我们可以知道，只要传入有效控制缩进的参数值，结果就会包含换行符。注意：最大缩进数为10，所有大于10的值都会被自动转换为10 。

如果第三个参数是一个字符串而非数值，那么这个字符串将在JSON字符串中被用作缩进字符（不再使用空格）。在缩进参数为字符串时，可以将缩进参数设置为制表符或者两个短下划线之类的任意字符。同样缩进字符串最长不能超过10个字符长。当超过10个字符长时将会只出现前10个字符。

#### 3.toJSON()方法

在JSON.stringify()方法不能满足对某些对象进行自定义序列化的需求情况下，可以给对象定义toJSON()方法，返回其自身的JSON格式。。原生 Date 对象有一个 toJSON()方法， 能够将JavaScript的Date 对象自动转换成ISO 8601日期字符串（与在Date 对象上调用toISOString() 的结果完全一样）。可以为任何对象添加toJSON()方法,如：

*var* book = {

"title": "Professional JavaScript",

"authors": [

"Nicholas C. Zakas"

],

edition: 3,

year: 2011,

toJSON: *function*(){

return *this*.title;

}

};

*var* jsonText = JSON.stringify(book);

以上代码在 book 对象上定义了一个 toJSON()方法，该方法返回图书的书名。与 Date 对象类似，这个对象也将被序列化为一个简单的字符串而非对象。可以让toJSON()方法返回任何值，它都能正常工作。比如，可以让这个方法返回 undefined，此时如果包含它的对象嵌入在另一个对象中，会导致 它的值变成 null，而如果它是顶级对象，结果就是 undefined。toJSON()可以作为函数过滤器的补充，因此理解序列化的内部顺序十分重要。

假设把一个对象传入 JSON.stringify()，序列化该对象的顺序如下。

(1) 如果存在 toJSON()方法而且能通过它取得有效的值，则调用该方法。否则，返回对象本身。

(2) 如果提供了第二个参数，应用这个函数过滤器。传入函数过滤器的值是第(1)步返回的值。

(3) 对第(2)步返回的每个值进行相应的序列化。

(4) 如果提供了第三个参数，执行相应的格式化。

无论是考虑定义 toJSON()方法，还是考虑使用函数过滤器，亦或需要同时使用两者，理解这个顺 序都是至关重要的。

### 18.2.3 解析选项

JSON.parse()方法也可以接受另外一个参数，该参数是个函数，将会在每个键值对上调用。为区别stringify()的过滤函数，该函数被称为还原函数，但实际上这两个函数的签名是相通的，他们都需要接收两个参数，一个是键，一个是值，并且都返回一个值。

如果还原函数返回undefined，则表示从结果中删除相对应的键；如果返回其他值，则将该值插入到结果中。在将日期字符串转换成Date对象时，需要用到还原函数。

待完善（588页）

## 18.3 小结

JSON 是一个轻量级的数据格式，可以简化表示复杂数据结构的工作量。JSON 使用 JavaScript 语法的子集表示对象、数组、字符串、数值、布尔值和 null。即使 XML 也能表示同样复杂的数据结果，但 JSON 没有那么烦琐，而且在 JavaScript 中使用更便利。 ECMAScript 5 定义了一个原生的 JSON 对象，可以用来将对象序列化为 JSON 字符串或者将 JSON 数据解析为 JavaScript 对象。JSON.stringify()和 JSON.parse()方法分别用来实现上述两项功能。 这两个方法都有一些选项，通过它们可以改变过滤的方式，或者改变序列化的过程。

原生的 JSON 对象也得到了很多浏览器的支持，比如 IE8+、Firefox 3.5+、Safari 4+、Opera 10.5 和 Chrome。

# 十九、Ajax与Comet

Ajax技术的核心是XMLHttpRequest（简称XHR）。Ajax技术可以异步从服务器获取信息，而无需刷新页面。这种技术就是无须刷新页面即可从服务器取得数 据，但不一定是 XML 数据。

远程脚本

## 19.1 XMLhttpRequest对象(XHR)

IE5是第一款引入XHR对象的浏览器。在 IE5 中，XHR 对象是通过 MSXML 库中的一个 ActiveX 对象实现的。因此，在 IE 中可能会遇到三种不同版本的 XHR 对象，即 MSXML2.XMLHttp、 MSXML2.XMLHttp.3.0 和 MXSML2.XMLHttp.6.0。要使用 MSXML 库中的 XHR 对象，需编写一个函数如：

//适用于 IE7 之前的版本

*function* createXHR(){

if (typeof *arguments*.callee.activeXString != "string"){ *var* versions = ["MSXML2.XMLHttp.6.0",

"MSXML2.XMLHttp.3.0",

"MSXML2.XMLHttp"],

i, len;

for (i=0,len=versions.length; i < len; i++){

try {

new ActiveXObject(versions[i]);

*arguments*.callee.activeXString = versions[i];

break;

} catch (ex){

//跳过

}

}

}

return new ActiveXObject(*arguments*.callee.activeXString);

}

创建XHR对象的语法：var xhr = new XMLHttpRequest();

在这个 createXHR()函数中加入对原生 XHR 对象的支持。例子：

*function* createXHR() {

if(typeof XMLHttpRequest != "undefined") {

return new XMLHttpRequest();

} else if(typeof ActiveXObject != "undefined") {

if(typeof *arguments*.callee.activeXString != "string") {

*var* versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",

"MSXML2.XMLHttp"

],

i, len;

for(i = 0, len = versions.length; i < len; i++) {

try {

new ActiveXObject(versions[i]);

*arguments*.callee.activeXString = versions[i];

break;

} catch(ex) {

//跳过

}

}

}

return newActiveXObject(*arguments*.callee.activeXString);

} else {

throw new Error("No XHR object available.");

}

}

var xhr = new createXHR();

通过调用以上方法，我们可以调用createXHR()方法。由于其他浏览器中对 XHR 的实现与 IE 最早的实现是兼容的，因此就可以在所有浏览器中都以相同方式使用上面创建的 xhr 对象。

### 19.1.1 XHR 用法

在使用XHR对象时，首先调用的第一个方法是open()，此方法接收三个参数：要发送的请求的类型、请求的URL和表示是否异步发送请求的布尔值。

例如：xhr.open("get", "example.php", false);

说明两点：一是 URL 相对于执行代码的当前页面（当然也可以使用绝对路径）；二是调用 open()方法并不会真正发送请求， 而只是启动一个请求以备发送。

只能向同一个域中使用相同端口和协议的 URL 发送请求。如果 URL 与启动请求 的页面有任何差别，都会引发安全错误。

open() 方法接收三参数，请求类型、请求URL和是否异步标志。

open()方法只能启动请求，XHR 对象要发送请求则需调用send()方法。send() 方法接收一个参数，即作为请求主体发送的数据。如果不需要通过请求主体发送数据，则必须传入null，这对个别浏览器是必须的。调用send()方法，请求被分配。

请求是同步的，JavaScript 代码会等到服务器响应之后再继续执行。在收到响应后，响应 的数据会自动填充 XHR 对象的属性，相关的属性简介如下。

* responseText：作为响应主体被返回的文本；
* responseXML：如果相应的内容类型是：“text/xml”或“application/xml”，这个属性中将包含着相应数据的XML DOM数据；
* status：响应的http状态；
* statusText：HTTP状态说明。

在接收到响应后，首先检查status属性，以确定响应是否成功返回。一般来说，可以将HTTP状态代码为200作为成功的标志。此时responseText属性的内容已经就绪，在内容类型正确的情况下，responseXML也应该能访问了状态码为304表示请求的资源并未被修改。可直接使用浏览器缓存的版本。

假如我们通过检测status来决定下一步操作，不要依赖statusText。无论内容类型是什么，响应主体的内容都会保存在responseText属性中；对于非XML数据而言，responseXML属性的值将为null。

一般情况下，我们需要发送异步请求，才能让JavaScript继续执行而不必响应

待。在异步情况下，我们可以检测XHR对象的readyState属性，该属性表示请求/响过程

的当前活动阶段。readyState属性有如下可取值：

* 0 ：未初始化。尚未调用open()方法；
* 1 ：启动。 已调用open()方法，尚未调用send()方法；
* 2 ：发送。已调用send()方法，但尚未接收到响应；
* 3 ：接收。已接收到部分响应数据；
* 4 ：完成。已接收全部响应数据，并且可以在客户端使用。

只要readyState属性的值由一个值变为另一个值，都会触发一次readystatechange事

件。可通过检测这个readystatechange事件来检测每次状态变化后的readyState属性的值。

通常，我们只对 readyState 值为 4 的阶 段感兴趣，因为这时所有数据都已经就绪。不过，必须在调用 open()之前指定 onreadystatechange 事件处理程序才能确保跨浏览器兼容性。

*var* xhr = createXHR();

xhr.onreadystatechange = *function*(){

if (xhr.readyState == 4){

if ((xhr.status >= 200 && xhr.status < 300)

|| xhr.status == 304){

alert(xhr.responseText);

} else {

alert("Request was unsuccessful: " + xhr.status);

}

}

};

xhr.open("get", "example.txt", true); xhr.send(null);

以上代码利用 DOM 0 级方法为 XHR 对象添加了事件处理程序，原因是并非所有浏览器都支持 DOM 2 级方法。与其他事件处理程序不同，这里没有向 onreadystatechange 事件处理程序中传递 event 对象； 必须通过 XHR 对象本身来确定下一步该怎么做。

这个例子在 onreadystatechange 事件处理程序中使用了 xhr 对象，没有使用 this 对象，原因是 onreadystatechange 事件处理程序的作用域问题。如果使用 this 对象，在有的浏览器中会导致函数执行失败，或者导致错误发生。因此，使用 实际的 XHR 对象实例变量是较为可靠的一种方式。

另外，在接收到响应之前还可以调用 abort()方法来取消异步请求，如下所示：

xhr.abort();

调用这个方法后，XHR 对象会停止触发事件，而且也不再允许访问任何与响应有关的对象属性。在终止请求之后，还应该对 XHR 对象进行解引用操作。由于内存原因，不建议重用 XHR 对象。

### 19.1.2 HTTP头部信息

每个HTTP请求或响应都会带有相应的头部信息。XHR对象提供了操作这两种头部（请求头部和响应头部）信息的方法。

默认情况下，发送XHR请求时还会发送的头部信息

* Accept：浏览器能够处理的内容类型；
* Accept-Charset :浏览器能显示的字符集；
* Accept-Ecoding:浏览器能够处理的压缩编码；
* Accept-Language：浏览器当前设置的语言；
* Connection：浏览器与服务器之间的连接类型；
* Cookie：当前页面设置的任何Cookie；
* Host：发出请求的页面的所在域；
* Referer:发出请求的页面的URI；注意，HTTP 规范将这个头部字段拼写错了，而为保证与规 范一致，也只能将错就错了。（这个英文单词的正确拼法应该是 referrer。）
* User-Agent：浏览器的用户代理字符串

使用setRequestHeader()方法可以设置自定义的请求头部信息。该方法接收两个参数：头部字段的名称和头部字段的值。要成功发送请求头部信息，就必须在调用open()方法之后且在调用send()方法之前调用setRequestHeader()方法。

服务器在接收到自定义的头部信息后，可以进行相应的后续操作。一般最好使用自定义的头部信息，不建议使用浏览器正常发送的头部字段。否则有可能影响服务器的响应。

**调用getResponseHeader()方法并传入头部字段名称，可以取得相应的响应头部信息。调用getAllResponseHeaders()可以获得一个包含所有头部信息的长字符串。**

在服务端也可以利用头部信息向浏览器发送额外的、结构化的数据。。在没有自定义信息的情况 下，getAllResponseHeaders()方法通常会返回的多行文本内容

### 19.1.3 Get请求

Get是最常见的请求类型，最常用于向服务器查询某些信息。必要时可以将查询字符串参数追加到URL的末尾。对XHR而言，位于传入open()方法的URL末尾的查询字符串必须经过正确编码。即**通过open()方法传参必须经过正确编码。**

使用GET请求经常会发生一个错误，就是查询的字符串格式有问题。查询字符串中的每个参数名称和值都必须经过encodeURIComponent()方法进行编码。然后才能方法到URL的末尾。且所有的名-值之间有 & 分隔，而路径和参数间使用问号分隔。可以使用下列方法进行编码组合向URL末尾追加查询参数：

**function addURLParam(url, name, value) {**

**url += (url.indexOf("?") == -1 ? "?" : "&");**

**url += encodeURIComponent(name) + "=" + encodeURIComponent(value);**

**return url;**

**}**

### 19.1.4 POST请求

使用频率仅次于Get的是POST请求，通常用于向服务器发送应该被保存的数据。POST请求应当把数据作为请求的主体提交，这点与Get请求传统上是不一样的。POST请求的主体可以包含非常多的数据，且格式不限。在open()方法的第一个参数传入“post”，就可以初始化一个POST请求。

POST请求初始化成功后，通过send()方法发送信息，可以向该方法中传入XML DOM文档，传入的文档经序列化后将作为请求的主体被提交到服务器。也可以向服务器发送字符串。

默认情况下，服务器对POST请求和提交Web表单的请求是不同的。服务端必须有程序来读取发送过来的原始数据，并从中提取有用的数据。

我们可以使用 XHR 来模仿表单提交：首先将 Content-Type 头部信息设置为 application/x-www-form-urlencoded，也就是表单提交时的内容类型，其次是以适当的格式创建一个字符串。POST数据的格式与查询字符串格式相同。如果需要将页面中表单的数据进行序列化，然后再通过 XHR 发送到服务器，那么就可以使用**serialize()**函数来创建这个字符串：

如果不设置 Content-Type 头部信息，那么发送给服务器的数据就不会出现在$\_POST 超级全局变量中。这时候，要访问同样的数据，就必须借助$HTTP\_RAW\_POST\_DATA。

与 GET 请求相比，POST 请求消耗的资源会更多一些。从性能角度来看，以发送 相同的数据计，GET 请求的速度最多可达到 POST 请求的两倍。

## 19.2 XMLHttpRequest 2级

XMLHttpRequest 1级只是把已有的XHR对象的实现细节描述了出来。XMLHttpRequest2级则进一步发展了 XHR。并非所有浏览器都完整地实现了 XMLHttpRequest 2 级规范，但所有浏览器都实现了它规定的部分内容。

### 19.2.1 FormData

表单数据的序列化。XMLHttpRequest 2 级定义了FormData类型。FormData 为序列化表单以及创建与表单格式相同的数据（用于通过 XHR 传输）提供了便利。

var data = new FormData();

append()方法，该方法接收两个参数：键和值；分别对应表单字段的名字和字段中包含的值。通过这种方法可以向表单添加任意多键值对。通过向FormData构造函数中传入表单元素也可以用表单元素的预先填入键值对,即可以预先向FormData的构造方法传入表单元素初始化表单：

var data = new FormData(document.forms[0]);

创建了FormData的实例之后，可以直接传给XHR的send()方法。

xhr.send(data);

使用FormData可以不必明确在XHR对象上设置请求头部。XHR对象能够识别传入的数据类型是FormData的实例，并配置适当的头部信息。

### 19.2.2 超时设定

IE8 为XHR对象添加了一个timeout属性，表示请求在等待响应多少毫秒之后就会终止。再给timeout设置一个数值后如果浏览器在规定的时间内还没有接受到响应就会触发timeout事件，进而调出ontimeout事件处理程序。

xhr.timeout = 1000; //将超时设置为 1 秒钟（仅适用于 IE8+）

xhr.ontimeout = function(){ alert("Request did not return in a second."); };

请求终止时，会调用 ontimeout 事件处理程序。但此时 readyState 可能已经改变为 4 了，这意味着会调用 onreadystatechange 事件处理程序。可是，如果在超时终止 请求之后再访问 status 属性，就会导致错误。为避免浏览器报告错误，可以将检查 status 属性的语句封装在一个 try-catch 语句当中。

### 19.2.3 overrideMineType() 方法

overrideMimeType()方法用于重写XHR响应的Mime类型。这个方法后来也被纳入了 XMLHttpRequest 2 级规范。因为返回响应的 MIME 类型决定了 XHR 对象如何处理它，所 以提供一种方法能够重写服务器返回的 MIME 类型是很有用的。

服务器返回的MIME类型text/plain，但数据中实际包含的是XML。根据MIME类型，即使数据是 XML，responseXML属性中仍然是null。通过调用 overrideMimeType()方法，可以保证把响应当作 XML 而非纯文本来处理。

var xhr = createXHR();

xhr.open("get", "text.php", true);

xhr.overrideMimeType("text/xml");

xhr.send(null);

这个例子强迫XHR对象将响应当作XML而非纯文本来处理。调用overrideMimeType()必须在 send()方法之前，才能保证重写响应的 MIME 类型。

支持 overrideMimeType()方法的浏览器有 Firefox、Safari 4+、Opera 10.5 和 Chrome。

## 19.3 进度事件 (待完善。。。)

Progress Events 规范是 W3C 的一个工作草案，定义了与客户端服务器通信有关的事件。这些事件最 早其实只针对 XHR 操作，但目前也被其他 API 借鉴。

有以下 6 个进度事件。

* loadstart：在接收到响应数据的第一个字节时触发。
* progress：在接收响应期间持续不断地触发。
* error：在请求发生错误时触发。
* abort：在因为调用 abort()方法而终止连接时触发。
* load：在接收到完整的响应数据时触发。
* loadend：在通信完成或者触发 error、abort 或 load 事件后触发。

每个请求都从触发 loadstart 事件开始，接下来是一或多个 progress 事件，然后触发 error、 abort 或 load 事件中的一个，最后以触发 loadend 事件结束。 支持前 5 个事件的浏览器有 Firefox 3.5+、Safari 4+、Chrome、iOS 版 Safari 和 Android 版 WebKit。 Opera（从第 11 版开始）、IE 8+只支持 load 事件。目前还没有浏览器支持 loadend 事件。

## 19.4 跨资源共享（CORS）待完善

## 19.5 其他跨域技术(待完善)

### 19.5.1 图像Ping

# 二十、高级技巧

使用JavaScript编程，要么使用过程方法，要么使用面向对象方式。

## 20.1 高级函数

所有函数都是对象，使用函数指针变得非常简单。以下函数的高级方法。

### 20.1.1 安全的类型检查

JavaScript的类型检测机制并非完全可靠。 在任何值上调用Object原生的toString()方法，都会返回一个[object NativeConstructorName]格式的字符串。每个类在内部都有一个[[Class]]属性，这个属性指定了上述字符串的构造函数名。下面方式，较多应用于检测原生对象。

例如：**Object.prototype.toString.call(value)；**

该方法返回[Object NativeConstructorName]

若是数组则[Object Array]；

若是正则表达式则[Object RegExp]；

若是函数则[Object Function]’

要注意，对于在 IE 中以 COM 对象形式实现的任何函数，isFunction()都将返回 false（因 为它们并非原生的 JavaScript 函数）；

Object 的 toString()方法不能检测非原生构造函数的构造函数名。因此，开发人员定义的任何构造函数都将返回[object Object]。

请注意，Object.prototpye.toString()本身也可能会被修改。本节讨论的 技巧假设 Object.prototpye.toString()是未被修改过的原生版本。

### 20.1.2 作用域安全的构造函数

构造函数一般是使用new操作符来调用的函数。当使用new来调用时，构造函数内会用到的this对象会指向新创建的的对象实例。

*function* Person(*name*, *age*, *job*){

*this*.name = name;

*this*.age = age;

*this*.job = job;

}

*var* person = new Person("Nicholas", 29, "Software Engineer");

如上例，当使用new操作时，this指向当前创建的对象实例；但当只通过Person()来调用则this指向全局对象window上，错误的增加了对象的属性。为了能安全增加对象的属性，就需要创建一个作用域安全的构造函数。

作用域安全的构造函数在进行任何更改前，首先确认this对象是正确类型的实例。如果不是，那么会创建新的实例并返回。如下：

*function* Person(*name*, *age*, *job*){

if (*this* instanceof Person){

*this*.name = name;

*this*.age = age;

*this*.job = job;

} else {

return new Person(name, age, job);

}

}

从上例可以看出，调用构造函数时，无论是否使用new操作符都会返回一个对象的新实例，避免在全局对象上意外增加属性。

需要注意的是，使用作用域安全的构造函数模式，就锁定了可以调用的构造函数的环境。如果使用构造函数窃取模式的继承且不使用原型链，这个继承可能会被破坏。

*function* Polygon(*sides*){

if (*this* instanceof Polygon) {

*this*.sides = sides;

*this*.getArea = *function*(){

return 0;

};

} else {

return new Polygon(sides);

}

}

*function* Rectangle(*width*, *height*){

Polygon.call(*this*, 2);

*this*.width = width;

*this*.height = height;

*this*.getArea = *function*(){

return *this*.width \* *this*.height;

};

}

*var rect* = new Rectangle(5, 10);

alert(*rect*.sides); //undefined

ScopeSafeConstructorsExample03.htm

在这段代码中，Polygon 构造函数是作用域安全的，然而 Rectangle 构造函数则不是。新创建一个 Rectangle 实例之后，这个实例应该通过 Polygon.call()来继承 Polygon 的 sides 属性。但是，由于 Polygon 构造函数是作用域安全的，*this* 对象并非 Polygon 的实例，所以会创建并返回一个新的 Polygon 对象。Rectangle 构造函数中的 *this* 对象并没有得到增长，同时 Polygon.call()返回

的值也没有用到，所以 Rectangle 实例中就不会有 sides 属性。

如果构造函数窃取结合原型链或寄生组合则可以解决破坏继承这个问题。

*function* Polygon(*sides*){

if (*this* instanceof Polygon) {

*this*.sides = sides;

*this*.getArea = *function*(){

return 0;

};

} else {

return new Polygon(sides);

}

}

*function* Rectangle(*width*, *height*){

Polygon.call(*this*, 2);

*this*.width = width;

*this*.height = height;

*this*.getArea = *function*(){

return *this*.width \* *this*.height;

};

}

Rectangle.prototype = new Polygon();

*var rect* = new Rectangle(5, 10);

alert(*rect*.sides); //2

上面这段重写的代码中，一个Rectangle实例也同时是一个Polygon实例，所以Polygon.call() 会照原意执行，最终为 Rectangle 实例添加了 sides 属性。

多个程序员在同一个页面中写JavaScript代码的环境中，作用域构造函数就有用，除非单纯的基于构造函数窃取来实现继承，推荐作用域安全构造函数作为最佳实践

### 20.1.3 惰性载入函数（适用于执行一次便可确定最终结构，无须重复执行相同代码）

惰性载入表示函数执行的分支仅会执行一次。有两种实现惰性载入的方式：

第一种是在函数被调用时再处理函数。在第一次执行的过程中，该函数会被覆盖为另外一种按合适的方式执行的函数，这样任何对原函数的调用都不用在经过执行的分支。

function createXHR() {

if(typeof XMLHttpRequest != "undefined") {

createXHR = function() {

return new XMLHttpRequest();

};

} else if(typeof ActiveXObject != "undefined") {

createXHR = function() {

if(typeof arguments.callee.activeXString != "string") {

var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",

"MSXML2.XMLHttp"

],

i, len;

for(i = 0, len = versions.length; i < len; i++) {

try {

new ActiveXObject(versions[i]);

arguments.callee.activeXString = versions[i];

break;

} catch(ex) {

//skip

}

}

}

return new ActiveXObject(arguments.callee.activeXString);

};

} else {

createXHR = function() {

throw new Error("No XHR object available.");

};

}

return createXHR();

}

以createXHR（）函数为例，在条件语句的每个分支都给createXHR变量赋值function(){};进而覆盖原有函数，下一次调用就会直接执行调用被分配的函数，而不用在执行if语句。

第二种实现惰性载入的方式是在函数声明时就指定适当的函数。第一次调用就不会损失性能，在代码在加载时会损失一点性能。

*var* createXHR = (*function*() {

if(typeof XMLHttpRequest != "undefined") {

return *function*() {

return new XMLHttpRequest();

};

} else if(typeof ActiveXObject != "undefined") {

return *function*() {

if(typeof *arguments*.callee.activeXString != "string") {

*var* versions = ["MSXML2.XMLHttp.6.0",

"MSXML2.XMLHttp.3.0",

"MSXML2.XMLHttp"

],

i, len;

for(i = 0, len = versions.length; i < len; i++) {

try {

new ActiveXObject(versions[i]);

*arguments*.callee.activeXString = versions[i];

break;

} catch(ex) {

//skip

}

}

}

return new ActiveXObject(*arguments*.callee.activeXString);

};

} else {

return *function*() {

throw new Error("No XHR object available.");

};

}

})();

此例是创建一个匿名的自执行函数，用以确定应该使用哪个函数实现。实现的逻辑是不一样的，不一样的地方就是第一行代码（使用 var 定义函数）、新增了自执行的匿名函数，每个分支都返回正确的函数定义，以便赋值给createXHR()。

惰性载入函数的优点是只在执行分支代码是损耗一点性能。具体使用哪种惰性方法要具体对待。

### 20.1.4 函数绑定（不太熟悉需要深入看下…）

函数绑定会创建另外一个函数，可以在特定的环境中以指定参数调用另外一个函数。该技巧常常和回调函数与事件处理程序一起使用，以便将函数作为变量传递的同时保留代码的执行环境。如：

*var* handler = {

message: "Event handled",

handleClick: *function*(*event*) {

alert(*this*.message);

}

};

*var* btn = *document*.getElementById("my-btn");

EventUtil.addHandler(btn, "click", handler.handleClick);

创建了一个叫做 handler 的对象。handler.handleClick()方法被分配为 一个 DOM 按钮的事件处理程序。当按下该按钮时，就调用该函数，显示一个警告框。虽然貌似警告框 应该显示 Event handled ，然而实际上显示的是 undefiend 。这个问题在于没有保存 handler.handleClick()的环境，所以 this 对象最后是指向了 DOM 按钮而非 handler（在 IE8 中，this指向 window。）

*var* handler = {

message: "Event handled",

handleClick: *function*(*event*) {

alert(*this*.message);

}

};

*var* btn = *document*.getElementById("my-btn");

EventUtil.addHandler(btn, "click", *function*(*event*) {

handler.handleClick(*event*);

});

这个解决方案在 onclick 事件处理程序内使用了一个闭包直接调用 handler.handleClick()。 当然，这是特定于这段代码的解决方案。创建多个闭包可能会令代码变得难于理解和调试。

JavaScript库实现了一个可以将函数绑定到指定环境的函数，此函数一般叫bind()。

一个简单的bind()函数接收一个函数和一个环境，并返回一个在给定环境中调用给定函数的函数，并且**将所有参数原封不动传递过去**。语法如下：

function bind(fn, context){

return function(){

return fn.apply(context, arguments);

};

}

在bind()中创建了一个闭包，闭包使用apply()函数调用传入的函数，并给apply（）函数传递context对象和参数。需要注意的是，此处的arguments对象是内部对象的，而不是bind()方法的。当调用返回的函数时，它会在给定的环境中执行被传入的函数并给出所有参数。bind()函数按以下方式使用

*var* handler = {

message: "Event handled",

handleClick: *function*(*event*) {

alert(*this*.message);

}

};

*var* btn = *document*.getElementById("my-btn");

*EventUtil.addHandler(btn,"click",bind(handler.handleClick,handler));*

此例，我们用 bind()函数创建了一个保持了执行环境的函数，并将其传给 EventUtil. addHandler()。event 对象也被传给了该函数

ECMAScript 5 为所有函数定义了一个原生的 bind()方法，是可以直接在函数上调用这个方法。例如：

*var* handler = {

message: "Event handled",

handleClick: *function*(*event*) {

alert(*this*.message + ":" + *event*.type);

}

};

*var* btn = *document*.getElementById("my-btn");

EventUtil.addHandler(btn,"click",handler.handleClick.bind(handler));

原生bind()方法与自定义bind()方法类似，都要传入作为this值的对象。

只要是将某个函数指针以值的形式进行传递，同时该函数必须在特定环境中进行执行，凸显被绑定函数的效用。他们主要用于事件处理程序以及setTimeout()和setInterval()。绑定函数比普通函数有更多开销，需要更多内存，也因多重函数调用而缓慢一些，最好在必要时使用。

### 20.1.5 函数柯里化(622页)

与函数绑定紧密相关的主题是函数柯里化。它用于创建已经设置好了一个或多个参数的函数。函数柯里化的基本方法与函数绑定是一样的：使用一个闭包返回一个函数。两者区别在于：当函数被调用时，返回的函数还需要设置一些传入的参数。

柯里化函数通常由以下步骤动态创建：调用另一个函数，并为它传入要柯里化的函数和必要的参数。创建柯里化函数的通用方式：

*function* curry(*fn*) {

*var* args = Array.prototype.slice.call(*arguments*, 1);

return *function*() {

*var* innerArgs = Array.prototype.slice.call(*arguments*);

*var* finalArgs = args.concat(innerArgs);

return fn.apply(null, finalArgs);

};

}

## 20.2、[防篡改对象](#_1._尊重对象所有权)

任何对象都可以被在同一环境中运行的代码修改。因此要避免被篡改。ECMAScript5中定义了防篡改对象（tamper-proof object）。

通过手动设置对象的属性的[[Configurable]]、 [[Writable]]、 [[Enumerable]]、[[Value]]、[[Get]]以及[[Set]]特性，以改变属性的行为。 类似地，ECMAScript 5也增加了几个方法，通过它们可以指定对象的行为。

一旦把对象定义为防篡改，就无法撤销了。

# 二十一、离线应用与客户端存储

# 二十二、最佳实践

## 22.1 可维护性

在早期的网站中，JavaScript 主要是用于小特效或者是表单验证。

### 22.1.1 什么是可维护性代码

可维护性代码的特点:

* 可理解性：其他人可以接手代码并理解它的意图和一般途径，而无需原开发人员的完整解释。
* 直观性：—代码中的东西一看就能明白，不管其操作过程多么复杂。
* 可适应性：代码以一种数据上的变化不要求完全重写的方法撰写。
* 可拓展性：在代码架构上已考虑到在未来允许对核心功能进行扩展。
* 可调试性：当有地方出错时，代码可以给予你足够的信息来尽可能直接地确定问题所在。

### 22.1.2 代码约定

1.可读性

2.变量和函数命名

3.变量类型透明

有三种表示变量数据类型的方式：

1.初始化

2.使用匈牙利标记法来指定变量类型

3. 是使用类型注释。类型注释放在变量名右边，但是在初始化前面。这种方式是在变量旁边放一段指定类型的注释

### 22.1.3 松散耦合

只要应用的某个部分过分依赖另外一个部分，代码就是耦合过紧，难以维护。

#### 1.解耦HTML/JavaScript

一种常见的耦合类型就是HTML/JavaScript。在 Web 上，HTML 和 JavaScript 各自代表了解决 方案中的不同层次：HTML 是数据，JavaScript 是行为。直接写在HTML中的JavaScript，使用包含内联代码的<script>元素或者使用HTML属性来分配事件处理程序。

HTML 呈现应该尽可能与 JavaScript 保持分离。当 JavaScript 用于插入数据时，尽量不要直接插入 标记。一般可以在页面中直接包含并隐藏标记，然后等到整个页面渲染好之后，，就可以用 JavaScript 显 示该标记，而非生成它。另一种方法是进行 Ajax 请求并获取更多要显示的 HTML，这个方法可以让同 样的渲染层（PHP、JSP、Ruby 等等）来输出标记，而不是直接嵌在 JavaScript 中。

#### 2.解耦CSS/JavaScript

另一个 Web 层则是 CSS，它主要负责页面的显示。JavaScript 和 CSS 也是非常紧密相关的：他们都 是 HTML 之上的层次，因此常常一起使用。但是，和 HTML 与 JavaScript 的情况一样，CSS 和 JavaScript 也可能会过于紧密地耦合在一起。最常见的紧密耦合的例子是使用 JavaScript 来更改某些样式。

由于 CSS 负责页面的显示，当显示出现任何问题时都应该只是查看 CSS 文件来解决。然而，当使 用了 JavaScript 来更改某些样式的时候，比如颜色，就出现了第二个可能已更改和必须检查的地方。结果是JavaScript也在某种程度上负责了页面的显示，并与CSS紧密耦合了。如果未来需要更改样式表，CSS和JavaScript文件可能都需要修改。这就给开发人员造成了维护上的噩梦。所以在这两个层次之间必须有清晰的划分。

第二类紧密耦合仅会在 IE 中出现（但运行于标准模式下的 IE8 不会出现），它可以在 CSS 中通过表 达式嵌入 JavaScript。通常要避免使用表达式，因为它们不能跨浏览器兼容，还因为它们所引入的 JavaScript 和 CSS 之间 的紧密耦合。如果使用了表达式，那么可能会在 CSS 中出现 JavaScript 错误。由于 CSS 表达式而追踪过 JavaScript 错误的开发人员，会告诉你在他们决定看一下 CSS 之前花了多长时间来查找错误。 再次提醒，好的层次划分是非常重要的。显示问题的唯一来源应该是 CSS，行为问题的唯一来源应 该是 JavaScript。在这些层次之间保持松散耦合可以让你的整个应用更加易于维护。

#### 3. 解耦应用逻辑／事件处理程序

较好的方法是将应用逻辑和事件处理程序相分离，这样两者分别处理各自的东西。一个事件处理程 序应该从事件对象中提取相关信息，并将这些信息传送到处理应用逻辑的某个方法中。

从事件处理程序中分离应用逻辑有几个好处。首先，可以让你更容易更改触发特定过程的事件。如 果最开始由鼠标点击事件触发过程，但现在按键也要进行同样处理，这种更改就很容易。其次，可以在不附加到事件的情况下测试代码，使其更易创建单元测试或者是自动化应用流程。 以下是要牢记的应用和业务逻辑之间松散耦合的几条原则：

* + 勿将 event 对象传给其他方法；只传来自 event 对象中所需的数据；
  + 任何可以在应用层面的动作都应该可以在不执行任何事件处理程序的情况下进行；
  + 任何事件处理程序都应该处理事件，然后将处理转交给应用逻辑。

### 22.1.4、编程实践(680页)

#### 1. 尊重对象所有权

JavaScript的动态性质使得几乎所有的任何东西在任何时候都可以修改。在JavaScript中引入防止修改的机制，三种锁定级别：

阻止修改，三种锁定级别，逐渐

* 防止拓展
  + 禁止为对象添加属性和方法，但修改和删除已存在的属性和方法;
* 密封
  + 类似防止拓展，而且禁止为对象删除已存在的属性和方法,被密封的对象也是不可拓展的;
* 冻结
  + 类似密封。而且禁止为对象修改已存在的方法和属性（所有字段均为只读),冻结的对象也是不可拓展和密封的;

每种锁定级别都有两个方法：

一种是用于实时操作；

一种是检测是否应用了相应操作

防止拓展的方法：

1. Object.preventExtension()

2. Object.isExtensible()

密封的方法：

1. Object.seal()

2. Object.isSealed()

冻结的方法：

1. Object.freeze()

2. Object.isFrozen()

一旦一个对象被锁定，将无法解锁！！

在非严格模式下，对相应锁定的对象进行相应操作，只会悄无声息的失败，

建议使用严格模式，会抛出错误

在JavaScript中，默认情况下。所有对象都是可以修改的。在其它语言中，当没有实际源代码的时候，对象和类是不可变的。

尊重对象所有权也就是不能修改不属于你的代码。即，如果你不负责创建或维护某个对象、它的对象或者它的方法，那么你就不能对它们进行修改。具体而言：

不要为实例或原型添加属性；

不要为实例或原型添加方法；

不要重定义已存在的方法。

最佳的方法便是永远不修改不是由你所有的对象。

拥有对象，就是说这个对象是你创建的。为对象创建新的功能：

创建包含所需功能的新对象，并用它与相关对象进行交互；

创建自定义类型，继承需要修改的类型，并可以为自定义类型创建额外的功能。

#### 2. 避免全局变量

与尊重对象所有权密切相关的就是尽可能避免全局变量和函数。最多创建一个全局变量，让其他对象和函数存在其中。

单一的全局量的延伸便是命名空间的概念。**命名空间**包括创建一个用于放置功能的对象。

#### 3.避免与null进行比较

JavaScript不作任何的自动类型检查。在JavaScript代码中很少进行类型检测。最常见的类型检测就是查看某个值是否为null。

现实中与null比较很少适合情况而被使用。必须按照所期望进行类型检查，而不是按照不被期望的进行类型检测。例如，希望某个值为数组，应该检测值为数组，而不是不为null。

如果看到了与null进行比较的代码，需尝试进行技术替换：

* 如果值应为一个引用类型，使用instanceof操作符检查其构造函数；
* 如果值应为基本类型，使用typeof操作符检查其类型；
* 如果希望某个对象包含某个特定的方法名，则使用typeof操作符确保指定名字的方法存在于对象上。

代码中的 null 比较越少，就越容易确定代码的目的，并消除不必要的错误。

#### 4.使用常量

显示在用户界面上的字符串应该以允许 进行语言国际化的方式抽取出来。URL 也应被抽取出来，因为它们有随着应用成长而改变的倾向。基本 上，有着可能由于这样那样原因会变化的这些数据，那么都会需要找到函数并在其中修改代码 。可以通过将数据抽取出来变成单独定义的常量的方式，将应 用逻辑与数据修改隔离开来。

需将数据和使用它的逻辑进行分离的值的类型：

* **重复值**：任何在多处用到的值都应抽取为一个常量。
* **用户界面字符串：**任何用于显示给用户的字符串，都应被抽取出来以方便国际化。
* **URLs：**在 Web 应用中，资源位置很容易变更，所以推荐用一个公共地方存放所有的 URL。
* **任意可能会更改的值：** 每当你在用到字面量值的时候，你都要问一下自己这个值在未来是不 是会变化。如果答案是“是”，那么这个值就应该被提取出来作为一个常量。

## 22.2性能

因为JavaScript是一个解释型语言，执行速度要比一般的编译型语言要慢。

Chrome 是第一款内置优化引擎，将 JavaScript 编译成本地代码的浏览器。

### 22.2.1、注意作用域

随着作用域链中的作用域数量的增加，访问当前作用域以外的变量的时间也在增加。访问全局变量总是要比访问局部变量慢，因为需要遍历作用域链。只要能减少花费在作用域链上的时间，就能增加脚本的整体性能。

#### 1.避免全局查找

可能优化脚本性能最重要的就是注意全局查找。使用全局变量和函数都会比局部的开销大，因为要涉及作用域链上的查找。

将在一个函数中会用到多次的全局对象存储为局部变量总是没错的。

#### 2.避免with语句

with语句的原本用意是为逐级的对象访问提供命名空间式的速写方式，即在指定的区域直接通过节点名称调用对象。例如：

with(document.forms[]){

name.value = "lee king";

address.value = "Peking";

zipcode.value = "";

}

与之对应的是传统书写方式：

document.forms[].name.value = "lee king";

document.forms[].address.value = "Peking";

document.forms[].zipcode.value = "";

但在性能非常重要的地方必须避免使用with语句。和函数类似，with语句会创建自己的作用域，因此会增加其中执行的代码的作用域链的长度。由于额外的作用域链查找，在with语句中执行的代码比外面执行的代码慢。

必须使用with语句的情况很少，因为它主要用于消除额外的字符。在大多数情况下，可以用局部变量完成相同的事情而不引入新的作用域，即优先使用局部变量的方式替换with语句。这将避免全局查找带来的性能低下。

### 22.2.2、选择正确的方法

#### 1、避免不必要的属性查找

在计算机科学中，算法的复杂度使用O符号来表示。最简单、最快捷的算法是常数值即O(1)。之后算法变得越来越复杂并花费更长的时间。

下表展示常见的算法类型：

|  |  |  |
| --- | --- | --- |
| 标记 | 名称 | 描述 |
| O(1) | 常数 | 不管有多少值，执行的时间都是恒定的。一般表示简单值和存储在变量中的值。 |
| O() | 对数 | 总的执行时间和值的数量相关。但是要完成算法，并不一定要获取每个值 |
| O(n) | 线性 | 总执行时间和值的数量直接相关。如：遍历数组中的所有元素 |
| O() | 平方 | 总执行时间和值的数量有关，每个值至少要获取n次。 |

常数值，即O(1),指代字面值和存储在变量中的值。符号O(1)表示无论有多少值，需要获取常量值的时间都一样。获取常量值是非常高效的过程。

使用变量和数组要比访问对象上的属性更有效率，后者是一个O(n)操作。对象上的任何属性查找都要比访问变量或者数组花费更长时间，因为必须在原型链中对拥有该名称的属性进行一次搜索。简而言之，属性查找越多执行时间就越长。

注意获取单个值的多重属性查找。

一旦多次用到对象属性，应该将其存储在局部变量中。第一次访问该值会是O(n),而后续的访问都会是O(1)，

一般来说，只要能减少算法的复杂度，就尽量减少。尽可能地使用局部变量将属性查找替换为值查找。进一步讲，如果即可以用数字化的数组位置进行访问，也可以使用命名属性，那么使用数字位置。

#### 2、优化循环

循环是编程中最常见的结构。优化循环有助于提供性能，一个循环的基本优化步骤：

* 减值迭代：大多数循环使用一个从0开始，增加到某个特定值的迭代器。在大多数情况下，从最大值开始，在循环中不断减值的迭代器更加高效。
* 简化终止条件：由于每次循环都会计算终止条件，所以必须保证它尽可能快。即避免属性查找或其他O(n)操作。
* 简化循环体：循环体是执行最多的，所以要最大限度的优化，确保没有某些可以被很容易移除循环的密集运算。
* 使用后测试循环：最常用的for循环和while循环都是前测试循环，而do-while 这种后测试循环可以避免最初终止条件的计算。

使用“后测试”循环时必须确保要处理的值至少有一个。空数组会导致多余的一次循环而“前测试”循环则可以避免。

#### 3、展开循环

当循环次数是确定的，消除循环并使用多次函数调用往往更快。

当循环次数比较少的情况下，可以使用多次调用函数，这样可以减少建立循环和终止循环的开销。

如果循环的迭代次数不能事先确定，可以考虑使用一种叫作Duff装置的技术。Duff装置的基本概念是通过计算迭代的次数是否为8的倍数将一个循环展开为一系列语句。例如：

//credit: Jeff Greenberg for JS implementation of Duff’s Device

//假设 values.length > 0

*var* iterations = Math.ceil(values.length / 8);

*var* startAt = values.length % 8;

*var* i = 0;

do {

switch(startAt) {

case 0:

process(values[i++]);

case 7:

process(values[i++]);

case 6:

process(values[i++]);

case 5:

process(values[i++]);

case 4:

process(values[i++]);

case 3:

process(values[i++]);

case 2:

process(values[i++]);

case 1:

process(values[i++]);

}

startAt = 0;

} while (--iterations > 0);

Duff 装置的实现是通过将 values 数组中元素个数除以 8 来计算出循环需要进行多少次迭代的。然后使用取整的上限函数确保结果是整数。如果完全根据除 8 来进行迭代，可能会有一些不能被处理到的元素，这个数量保存在 startAt 变量中。首次执行该循环时，会检查 StartAt 变量看有需要多少额外调用。例如，如果数组中有 10 个值，startAt 则等于 2，那么最开始的时候 process()则只会被调用 2 次。在接下来的循环中，startAt 被重置为 0，这样之后的每次循环都会调用 8 次 process()。展开 循环可以提升大数据集的处理速度。

更快的 Duff 装置技术，这个版本相当于把数组长度除以8的余数数量的循环提取出来

//credit: Speed Up Your Site (New Riders, 2003)

*var* iterations = Math.floor(values.length / 8);

*var* leftover = values.length % 8;

*var* i = 0;

if(leftover > 0) {

do {

process(values[i++]);

} while (--leftover > 0);

}

do {

process(values[i++]);

process(values[i++]);

process(values[i++]);

process(values[i++]);

process(values[i++]);

process(values[i++]);

process(values[i++]);

process(values[i++]);

} while (--iterations > 0);

在这个实现中，剩余的计算部分不会在实际循环中处理，而是在一个初始化循环中进行除以 8 的操 作。当处理掉了额外的元素，继续执行每次调用 8 次 process()的主循环。这个方法几乎比原始的 Duff 装置实现快上 40%。

如果是大数据集使用展开循环可以节省很多时间，但对于小数据集额外的开销可能会得不偿失。如果处理的不是大数据集一般来说并不值得。

#### 4、避免双重解释

当JavaScript代码想解析JavaScript的时候就会存在双重解释惩罚。当使用eval()函数或者Function构造函数以及使用setTimeout（）传入一个字符串参数时会发生这种情况。

//某些代码求值——避免!!

eval("alert('Hello world!')");

//创建新函数——避免!!

*var* sayHi = new Function("alert('Hello world!')");

//设置超时——避免!!

setTimeout("alert('Hello world!')", 500);

以上例子，都要解析包含了JavaScript代码的字符串。这个操作是不能在初始的解析过程中完成的，因为代码是包含在字符串中的，也就是说在JavaScript代码运行的同时必须新启动一个解析器来解析新的代码。实例化一个新的解析器有不容忽视的开销，所以这种代码要比直接解析慢得多。

只有在极少的情况下，eval()是绝对必需的，所以尽可能避免使用。

如果要提高代码性能，尽可能避免出现需要按照JavaScript解释的字符串。

#### 5、性能的其他注意事项

* 原生方法较快——只要有可能，使用原生方法而不是自己用 JavaScript 重写一个。原生方法是用 诸如 C/C++之类的编译型语言写出来的，所以要比 JavaScript 的快很多很多。JavaScript 中最容 易被忘记的就是可以在 Math 对象中找到的复杂的数学运算；这些方法要比任何用 JavaScript 写 的同样方法如正弦、余弦快的多。
* Switch 语句较快 —— 如果有一系列复杂的 if-else 语句，可以转换成单个 switch 语句则可 以得到更快的代码。还可以通过将 case 语句按照最可能的到最不可能的顺序进行组织，来进一 步优化 switch 语句。
* 位运算符较快 —— 当进行数学运算的时候，位运算操作要比任何布尔运算或者算数运算快。选 择性地用位运算替换算数运算可以极大提升复杂计算的性能。诸如取模，逻辑与和逻辑或都可 以考虑用位运算来替换。

### 22.2.3、最小化语句数

JavaScript中的代码数量影响所执行的操作的速度。完成多个操作的单个语句比完成单个操作的多个语句要快。所以要找出可以组合在一起的语句，以减少代码执行的时间。

#### 1、多个变量声明

在强语言类型中，不同的数据类型的变量必须在不同的语句中声明；但在JavaScript中所有的变量都可以使用的一个*var*语句来声明。JavaScript是弱语言类型。在JavaScript中声明多个变量使用逗号分隔。

#### 2、插入迭代值

在使用迭代值（即在不同的位置上增加或减少的值）的时候，尽可能合并语句。例如：

var name = values[i]; i++;

可以写成 ：var name = values[i++];

#### 3、使用数组和对象字面量

创建数组和对象的方法有：使用构造函数或字面量。使用构造函数总是用到更多的语句来插入或定义属性；而使用字面量可以将这些操作放到一个语句中完成。

只要有可能，尽可能使用数组或对象的字面量表达式来消除不必要的语句。

### 22.2.4、优化DOM交互

DOM 操作与交互要消耗大量时间， 因为它们往往需要重新渲染整个页面或者某一部分。

#### 1、最小化现场更新

一旦你需要访问的 DOM 部分是已经显示的页面的一部分，那么你就是在进行一个现场更新。之所 以叫现场更新，是因为需要立即（现场）对页面对用户的显示进行更新。每一个更改，不管是插入单个 字符，还是移除整个片段，都有一个性能惩罚，因为浏览器要重新计算无数尺寸以进行更新。现场更新 进行得越多，代码完成执行所花的时间就越长；完成一个操作所需的现场更新越少，代码就越快。

*var* list = *document*.getElementById("myList"),

item,

i;

for (i=0; i < 10; i++) {

item = *document*.createElement("li");

list.appendChild(item);

item.appendChild(*document*.createTextNode("Item " + i));

}

上例更新次数较多。要修正这个性能瓶颈，需要减少现场更新的数量。一般有 2 种方法。第一种是将列表从页面上移除， 最后进行更新，最后再将列表插回到同样的位置。这个方法不是非常理想，因为在每次页面更新的时候 它会不必要的闪烁。第二个方法是使用文档片段来构建 DOM 结构，接着将其添加到 List 元素中。这 个方式避免了现场更新和页面闪烁问题。如：

*var* list = *document*.getElementById("myList"),

fragment = *document*.createDocumentFragment(),

item,

i;

for (i=0; i < 10; i++) {

item = *document*.createElement("li");

fragment.appendChild(item);

item.appendChild(*document*.createTextNode("Item " + i));

}

list.appendChild(fragment);

在这个例子中只有一次现场更新，它发生在所有项目都创建好之后。文档片段用作一个临时的占位 符，放置新创建的项目。然后使用 appendChild()将所有项目添加到列表中。记住，当给 appendChild() 传入文档片段时，只有片段中的子节点被添加到目标，片段本身不会被添加的。

一旦需要更新 DOM，请考虑使用文档片段来构建 DOM 结构，然后再将其添加到现存的文档中。

#### 2、使用innerHTML

有两种在页面上创建 DOM 节点的方法：使用诸如 createElement()和 appendChild()之类的 DOM 方法，以及使用 innerHTML。对于小的 DOM 更改而言，两种方法效率都差不多。然而，对于大 的 DOM 更改，使用 innerHTML 要比使用标准 DOM 方法创建同样的 DOM 结构快得多。

当把 innerHTML 设置为某个值时，后台会创建一个 HTML 解析器，然后使用内部的 DOM 调用来 创建 DOM 结构，而非基于 JavaScript 的 DOM 调用。由于内部方法是编译好的而非解释执行的，所以执 行快得多。前面的例子还可以用 innerHTML 改写如下：

*var* list = *document*.getElementById("myList"),

html = "",

i;

for (i=0; i < 10; i++) {

html += "<li>Item " + i + "</li>";

}

list.innerHTML = html;

这段代码构建了一个 HTML 字符串，然后将其指定到 list.innerHTML，便创建了需要的 DOM 结 构。虽然字符串连接上总是有点性能损失，但这种方式还是要比进行多个 DOM 操作更快。

**使用 innerHTML 的关键在于（和其他 DOM 操作一样）最小化调用它的次数**。

调用 innerHTML 实际上就是一次现场更新，所以也要如此对待。构建好一个字符串然后一次性调用 innerHTML 要比调用 innerHTML 多次快得多。

#### 3、使用事件代理

大多数 Web 应用在用户交互上大量用到事件处理程序。页面上的事件处理程序的数量和页面响应 用户交互的速度之间有个负相关。为了减轻这种惩罚，最好使用事件代理.

事件代理，事件冒泡。任何可以冒泡的事件都不仅仅可以在事 件目标上进行处理，目标的任何祖先节点上也能处理。使用这个知识，就可以将事件处理程序附加到更 高层的地方负责多个目标的事件处理。如果可能，在文档级别附加事件处理程序，这样可以处理整个页 面的事件。

#### 4、注意 HTMLCollection

HTMLCollection 对象的陷阱已经在本书中讨论过了，因为它们对于 Web 应用的性能而言是巨大 的损害。记住，任何时候要访问 HTMLCollection，不管它是一个属性还是一个方法，都是在文档上进 行一个查询，这个查询开销很昂贵。最小化访问 HTMLCollection 的次数可以极大地改进脚本的性能。

避免在循环体内多次调用 HTMLCollection。

编写 JavaScript 的时候，一定要知道何时返回 HTMLCollection 对象，这样你就可以最小化对他们 的访问。发生以下情况时会返回 HTMLCollection 对象：

* 进行了对 getElementsByTagName() 的调用；
* 获取了元素的 childNodes 属性；
* 获取了元素的 attributes 属性；
* 访问了特殊的集合，如 document.forms、document.images 等。

要了解当使用 HTMLCollection 对象时，合理使用会极大提升代码执行速度。

## 22.3、部署

### 22.3.1、构建过程

软件开发的典型模式就是写代码-编译-测试，即首先写好代码，然后通过编译，运行并确保其正常通过。但JavaScript并非一个编译型语言，因此模式就是写代码-测试，

写的代码不应该原封不动地放入浏览器中：

* 知识产权问题 —— 如果把带有完整注释的代码放到线上，那别人就更容易知道你的意图，对它 再利用，并且可能找到安全漏洞。
* 文件大小 —— 书写代码要保证容易阅读，才能更好地维护，但是这对于性能是不利的。浏览器 并不能从额外的空白字符或者是冗长的函数名和变量名中获得什么好处。
* 代码组织 —— 组织代码要考虑到可维护性并不一定是传送给浏览器的最好方式。

基于这些原因，最好给 JavaScript 文件定义一个构建过程。

构建过程始于在源控制中定义用于存储文件的逻辑结构。最好避免使用一个文件存放所有的 JavaScript，遵循以下面向对象语言中的典型模式：将每个对象或自定义类型分别放入其单独的文件中。 这样可以确保每个文件包含最少量的代码，使其在不引入错误的情况下更容易修改。另外，在使用像 CVS 或 Subversion 这类并发源控制系统的时候，这样做也减少了在合并操作中产生冲突的风险。

记住将代码分离成多个文件只是为了提高可维护性，并非为了部署。要进行部署的时候，需要将这 些源代码合并为一个或几个归并文件。推荐 Web 应用中尽可能使用最少的 JavaScript 文件，是因为 HTTP 请求是 Web 中的主要性能瓶颈之一。记住通过<script>标记引用 JavaScript 文件是一个阻塞操作，当代码下载并运行的时候会停止其他所有的下载。因此，尽量从逻辑上将 JavaScript 代码分组成部署文件。

一旦组织好文件和目录结构，并确定哪些要出现在部署文件中，就可以创建构建系统了。**Ant 构建工具**（http://ant.apache.org）是为了自动化 Java 构建过程而诞生的，不过因为其易用性和应用广泛，而 在 Web 应用开发人员中也颇流行.

Ant 由于其简便的文件处理能力而非常适合 JavaScript 编译系统。可以很方便地获得目录中 的所有文件的列表，然后将其合并为一个文件，如下所示：

<project name="JavaScript Project" default="js.concatenate">

<!-- 输出的目录 -->

<property name="build.dir" value="./js" />

<!-- 包含源文件的目录 -->

<property name="src.dir" value="./dev/src" />

<!-- 合并所有 JS 文件的目标 -->

<!-- Credit: Julien Lecomte, http://www.julienlecomte.net/blog/2007/09/16/ -->

<target name="js.concatenate">

<concat destfile="${build.dir}/output.js">

<filelist dir="${src.dir}/js" files="a.js, b.js"/>

<fileset dir="${src.dir}/js" includes="\*.js" excludes="a.js, b.js"/>

</concat>

</target>

</project>

该 build.xml 文件定义了两个属性：输出最终文件的构建目录，以及 JavaScript 源文件所在的源目录。 目标 js.concatenate 使用了元素来指定需要进行合并的文件的列表以及结果文件所要输 出的位置。元素用于指定 a.js 和 b.js 要首先出现在合并的文件中，元素指定了 之后要添加到目录中的其他所有文件，a.js 和 b.js 除外。结果文件最后输出到/js/output.js。

### 22.3.2、验证

JavaScript的验证工具JSLint。JSLint可以查找JavaScript代码中的语法错误及常见的编码错误。挖掘一些潜在问题：

* + eval()的使用；
  + 未声明变量的使用；
  + 遗漏的分号；
  + 不恰当的换行；
  + 错误的逗号使用；
  + 语句周围遗漏的括号；
  + switch 分支语句中遗漏的 break；
  + 重复声明的变量；
  + with 的使用；
  + 错误使用的等号（替代了双等号或三等号）；
  + 无法到达的代码。

可以使用 Ant 将 JSLint 作为构建过程的一部分运行，添加如下一个目标：

<target name="js.verify">

<apply executable="java" parallel="false">

<fileset dir="${build.dir}" includes="output.js"/>

<arg line="-jar"/>

<arg path="${rhino.jar}"/>

<arg path="${jslint.js}" />

<srcfile/>

</apply>

</target>

给开发周期添加代码验证这个环节有助于避免将来可能出现的一些错误。建议开发人员给构建过程 加入某种类型的代码验证作为确定潜在问题的一个方法，防患于未然。

### 22.3.3、压缩

有关压缩的两个方面：**代码长度和配重**。**代码长度**指的是浏览器所需解析的字节数；**配重**指的是实际从服务器传送到浏览器的字节数。。在 Web 开发的早期， 这两个数字几乎是一样的，因为从服务器端到客户端原封不动地传递了源文件。而在今天的 Web 上， 这两者很少相等，实际上也不应相等。

#### 1.文件压缩

因为 JavaScript 并非编译为字节码，而是按照源代码传送的，代码文件通常包含浏览器执行所不需 要的额外的信息和格式。。注释，额外的空白，以及长长的变量名和函数名虽然提高了可读性，但却是传 送给浏览器时不必要的字节。不过，我们可以使用压缩工具减少文件的大小。

压缩器一般进行的步骤：

* 删除额外的空白（包括换行）；
* 删除所有注释；
* 缩短变量名。

JavaScript 有不少压缩工具

#### 2.HTTP压缩

**配重**指的是实际从服务器传送到浏览器的字节数。因为现在的服务器和浏览器都有压缩功能，这个字节数不一定和代码长度一样。所有的五大 Web 浏览器（IE、Firefox、Safari、Chrome 和 Opera）都支持对所接收的资源进行客户端解压缩。这样服务器端就可以使用服务器端相关功能来压缩 JavaScript 文件。一个指定了文件使用了给定格式进行了压缩的 HTTP 头包含在了服务器响应中。接着浏览器会查看 该 HTTP 头确定文件是否已被压缩，然后使用合适的格式进行解压缩。结果是和原来的代码量相比在网络中传递的字节数量大大减少了。

## 22.4、小结

随着 JavaScript 开发的成熟，也出现了很多最佳实践。过去一度认为只是一种爱好的东西现在变成

了正当的职业，同时还需要经历过去其他编程语言要做的一些研究，如可维护性、性能和部署。

JavaScript 中的可维护性部分涉及到下面的代码约定。

* 来自其他语言中的代码约定可以用于决定何时进行注释，以及如何进行缩进，不过 JavaScript需要针对其松散类型的性质创造一些特殊的约定。
* 由于 JavaScript 必须与 HTML 和 CSS 共存，所以让各自完全定义其自己的目的非常重要：JavaScript 应该定义行为，HTML 应该定义内容，CSS 应该定义外观。
* 这些职责的混淆会导致难以调试的错误和维护上的问题。
* 随着 Web 应用中的 JavaScript 数量的增加，性能变得更加重要，因此，你需要牢记以下事项。
* JavaScript 执行所花费的时间直接影响到整个 Web 页面的性能，所以其重要性是不能忽略的。
* 针对基于 C 的语言的很多性能的建议也适用于 JavaScript，如有关循环性能和使用 switch 语句替代 if 语句。
* 还有一个要记住的重要事情，即 DOM 交互开销很大，所以需要限制 DOM 操作的次数。

流程的最后一步是部署。本章讨论了以下一些关键点。

* 为了协助部署，推荐设置一个可以将 JavaScript 合并为较少文件（理想情况是一个）的构建过程。
* 有了构建过程也可以对源代码自动运行额外的处理和过滤。例如，你可以运行 JavaScript 验证器来确保没有语法错误或者是代码没有潜在的问题。
* 在部署前推荐使用压缩器将文件尽可能变小。
* 和 HTTP 压缩一起使用可以让 JavaScript 文件尽可能小，因此对整体页面性能的影响也会最小。

# 二十三、新兴的API.

# 附录A、ECMAScript Harmony（待续…）

## A.1、一般性变化

### A.1.1、常量

增加了用const关键字声明常量的语法。const关键字的用法与var类型，但使用const声明的变量在赋初始值后就不能再重新赋值。例如：const MAX\_NUM = 12；

可以像声明变量一样在任何地方声明常量。但在同一作用域常量名不能与其他变量或函数名重名，否则会报错。

注意：使用const关键字后，修改常量的值不会有任何效果。

### A.1.2、块级作用域及其他作用域

新增定义块级作用域的语法：使用let关键字。

与const与var关键字类似，可以在任何地方使用let关键字定义变量并未变量赋值。并且使用let关键字定义的变量属于块级作用域，在块级作用域之外使用将会报错。

同时，也可以使用let关键字创建语句。在其中定义只能在后续代码中使用的变量，例如：let （name=“1”，age=100）{

//代码块

}

let语句创建了自身的作用域，该作用域内的变量与外部变量无关。

使用let也可以创建let表达式，其中的变量只在表达式中有定义；例如

var result = let(num=10, multiplier=2) num \* multiplier;

alert(result); //20

let表达式中的变量只在该表达式存在，执行表达式之后便不再存在。

## A.2、函数

### A.2.1、剩余参数与分布参数

Harmony中不再有arguments对象，因此无法通过该对象来获取未声明的参数。因而引入剩余参数语法，来表示传入的可变变量的参数。

剩余参数语法的形式：三个点后跟一个标识符。

通过这种语法传进来的多余出的参数会被放入一个数组中，即使没有多余的参数，剩余参数的对象也是一个数组Array的实例。因此可以以数组的方法对剩余参数的对象进行操作。

与剩余参数语法相关的是分布参数，通过分布参数可以向函数中传入一个数组，然后数组中的元素会映射到函数的每个参数上。分布参数的语法与剩余参数的语法相同，在标识符前面加三个点。 两者区别之处在于：分布参数在函数调用的时候使用，而剩余参数在定义函数的时候使用。

var result = sum(...[1, 2, 3, 4, 5, 6]);

### A.2.2、默认参数值

ECMAScript函数中所有的参数都是可选的，所以函数的调用不是检查传入参数的数量。对于那些没有传参，但我们不希望为空的参数，可以使用默认参数值替代。要为参数指定默认值，可以在参数名后加等号和指定值，例如：

function sum(num1, num2=0){ return num1 + num2; }

但在为引入默认参数值的写法前，使用num2 = num2 || 0;的写法

### A.2.3、生成器

生成器，就是一个对象，每次能生成系列值中的一个。对于在Harmony中，创建生成器可以让函数通过yield操作符返回某个特殊的值。对于使用yield操作符返回值的函数，调用它时会创建并返回一个新的Generator实例。然后在这个实例上调用next()方法，就会取得生成器的第一个值。此时执行的是原来的函数，但是执行流到yield语句就会停止，只返回特定的值；再次调用next()方法，原来函数中位于 yield 语句后的代码会继续执行，直到再次遇见 yield 语句时停止执行，此时再返回一个新值。从形式上可以看出，yield与return类似。例如：

*function* myNumbers(){

for (*var* i=0; i < 10; i++){

yield i \* 2;

}

}

*var* generator = myNumbers();

**var num= generator.next();**

**generator.close();**

在 myNumbers()函数完成退出而没有执行 yield 语句时（最后一次循环判断 i 不小于 10 的时候），生成器会抛出 StopIteration 错误。因此，为了输出生成器能产生的所有数值，这里用一个 try-catch 结构包装了一个 while 循环，以避免出错时中断代码执行。

如果不再需要某个生成器，最好调用它的close()方法，这样会执行原始函数的其他部分，包括try-catch相关的finally语句块。

* 在需要一系列值，而每一个值又与前一个值存在某种关系，可以使用生成器。

## A.3、数组及其他结构（待完善）

### A.3.1、迭代器

## A.4、新对象类型

### A.4.1、代理对象

Harmony为JavaScript引入了代理的概念。代理（proxy），就是表示接口的对象，对它的操作不一定作用在代理对象本身。代理是一种抽象机制，能够通过API只公开部分信息，同时还能对数据源进行全面控制。

在JavaScript中创建代理对象，可以使用Proxy.create()方法，并传入一个handler（处理程序）和一个可选的prototype（原型）对象。例如：

var proxy = Proxy.create(handler);

//创建一个以 myObject 为原型的代理对象

var proxy = Proxy.create(handler, myObject);

其中，handler对象包含用于定义捕捉器（trap）的属性。捕捉器本身是函数，用于处理（捕捉）原生功能，以便该功能能够以另外一种方式处理。要确保代理对象能按照预期工作，至少要实现以下7中基本的捕捉器：

* getOwnPropertyDescriptor：当在代理对象上调用Object.getOwnPropertyDescriptor() 时调用的函数。这个函数以接收到的属性名作为参数，返回属性描述符，或者在属性不存在时返 回 null。
* getPropertyDescriptor：当在代理对象上调用 Object.getPropertyDescriptor()时调 用的函数。（这是 Harmony 中的新方法。）这个函数以接收到的属性名作为参数，返回属性描述 符，或者在属性不存在时返回 null。
* getOwnPropertyNames：当在代理对象上调用 Object.getOwnPropertyNames ()时调用的 函数。这个函数以接收到的属性名作为参数，应该返回一个字符串数组。
* getPropertyNames：当在代理对象上调用 Object.getPropertyNames ()时调用的函数。 （这是 Harmony 中的新方法。）这个函数以接收到的属性名作为参数，应该返回一个字符串数组。
* defineProperty：当在代理对象上调用 Object.defineProperty()时调用的函数。这个函 数以接收到的属性名和属性描述符作为参数。
* delete：定义在对象属性上使用 delete 操作符时调用的函数。属性名以参数形式传进来，如 果删除成功则返回 true，删除失败返回 false。
* fix：当调用 Object.freeze()、Object.seal()或 Object.preventExtensions()时调 用的函数。当在代理对象上调用这几个方法时，返回 undefined 以抛出错误。

除了这 7 个基本的捕捉器，还有 6 个派生的捕捉器（derived trap）。与基本捕捉器不同，少定义一个 或几个派生捕捉器不会导致错误。每个派生的捕捉器都会覆盖一种默认的 JavaScript 行为。

* has 在对象上使用 in 操作符（例如"name" in object）时调用的函数。以接收到的属性名作 为参数，返回 true 表示对象包含该属性，否则返回 false。
* hasOwn：在代理对象上调用 hasOwnProperty()方法时调用的函数。以接收到的属性名作为参 数，返回 true 表示对象包含该属性，否则返回 false。
* get：在读取属性时调用的函数。这个函数接收两个参数，即包含被读属性的对象的引用及属性 名。这个对象引用可能是代理对象本身，也可能是继承了代理对象的对象。
* set：在写入属性时调用的函数。这个函数接收三个参数，即包含被写属性的对象的引用、属性名 和属性值。与 get 类似，这个对象引用可能是代理对象本身，也可能是继承了代理对象的对象。
* enumerate：当代理对象被放在 for-in 循环中时调用的函数。这个函数必须返回一个字符串 数组，其中包含在 for-in 循环中使用的相应属性名。
* keys：当在代理对象上调用 Object.keys()时调用的函数。与 enumerate 类似，这个函数也 必须返回一个字符串数组。

**在需要公开 API，而同时又要避免使用者直接操作底层数据的时候，可以使用代理**。

### A.4.2、代理函数

Harmony支持创建代理函数，代理函数与代理对象的区别在于，代理函数可以执行。创建代理函数可以通过Proxy.createFunction()方法，传入一个 handler （处理程序）对象、一个调用捕捉器函数和一个可选的构造函数捕捉器函数。例如：

var proxy = Proxy.createFunction(handler, function(){}, function(){});

与代理对象一样，handler 对象也有同样多的捕捉器。调用捕捉器函数是在代理函数执行（如 proxy()）时运行的代码。构造函数捕捉器是在用 new 操作符调用代理函数（如 new proxy()）时运 行的代码。如果没有指定构造函数捕捉器，则使用调用捕捉器作为构造函数。

### A.4.3、映射与几何

Map类型，也称为简单映射，只有一个目的：保存一组键值对儿。开发人员通常都使用普通对象来保存键值对儿，但问题是那样做会导致键容易与原生属性混淆。简单映射能做到键和值与对象属性分离，从而保证对象属性的安全存储。例如：

*var map* = new Map();

*map*.set("name", "Nicholas");

*map*.set("book", "Professional JavaScript");

*console*.log(*map*.has("name")); //true

*console*.log(*map*.get("name")); //"Nicholas"

*map*.delete("name");

简单映射的基本 API 包括 get()、set()和 delete()，每个方法的作用看名字就知道了。键可以是原始值，也可是引用值。

与简单映射相关的是 Set 类型。集合就是一组不重复的元素。与简单映射不同的是，集合中只有键，没有与键关联的值。在集合中，添加元素要使用 add()方法，检查元素是否存在要使用 has()方法，而删除元素要使用 delete()方法。以下是基本的使用示例。

*var* set = new Set();

set.add("name");

*console*.log(set.has("name")); //true

set.delete("name");

*console*.log(set.has("name")); //false

### A.4.4、WeakMap

WeakMap 是 ECMAScript 中唯一一个能让你知道什么时候对象已经完全解除引用的类型。WeakMap 与简单映射很相似，也是用来保存键值对儿的。它们的主要区别在于，WeakMap 的键必须是对象，而 在对象已经不存在时，相关的键值对儿就会从 WeakMap 中被删除。例如：

var key = {},

map = new WeakMap();

map.set(key, "Hello!");

//解除对键的引用，从而删除该值

key = null;

### A.4.5、StructType（待完善）

ES6引入结构化类型。对于JavaScript的结构类型，通过指定属性及其保存的数据类型，也可以创建类似的数据结构。早期实现的有

* + uint8：无符号 8 位整数。
  + int8：有符号 8 位整数。
  + uint16：无符号 16 位整数。
  + int16：有符号 16 位整数。
  + uint32：无符号 32 位整数。
  + int32：有符号 32 位整数。
  + float32：32 位浮点数。
  + float64：64 位浮点数。

要创建结构类型的对象，可以使用new关键字调用StructType传入对象字面量形式的属性定义。

var Size = new StructType({ width: uint32, height: uint32 });

### A.4.6、ArrayType（待完善）

与结构类型紧密相关的是数组类型，通过数组类型（ArrayType）可以创建一个数组，并限制数组的值必须是某种特定的类型（与 WebGL 中的类型化数组很相似）。要创建新的数组类型，可以调用 ArrayType 构造函数，并传入它应该保存的数据类型以及应该保存的元素数目。例如：

var SizeArray = new ArrayType(Size, 2);

var boxes = new BoxArray([ { width: 80, height: 60 }, { width: 50, height: 50 } ]);

以上代码创建了一个名为 SizeArray 的数组类型，这个数组类型只能保存 Size 的实例，同时也给数组分配了两个该实例的位置。要实例化数组类型，可以传入一个数组，其中包含应该转换的数据。 数据可以是字面量，只要该字面量能提升为正确的数据类型即可（比如在这个例子中，传入的字面量可以提升为结构类型）。

## A.5、类

ES6增加了一种关于类的语法，但JavaScript中的类其实就是一种语法糖，覆盖在目前基于构造函数和基于原型的方法和类型上。

class Person {

constructor(name, age){

public name = name;

public age = age;

}

sayName(){

alert(this.name);

}

getOlder(years){

this.age += years;

}

}

创建类的新语法以关键字class为开头，然后就是类名，在花括号中定义的是属性和方法，在类中定义方法不必再使用关键字function，有方法名和圆括号就可以，当方法名为constructor时，方法为类的构造函数。

在构造函数中，public 和 private 关键字用于创建对象的实例属性。这个例子中的 name 和 age 都是公有属性。

### A.5.1、私有成员

关于类语法的建议是默认支持私有成员的，包括实例中的私有成员和原型中的私有成员。private 关键字表示成员是私有的，不能在类方法之外访问。要访问私有成员，可以使用一种特殊的语法，即**调用 private()函数并传入 this 对象，然后再访问私有成员**。

class Person {

constructor(name, age){

public name = name;

private age = age;

}

sayName(){

alert(this.name);

}

getOlder(years){

private(this).age += years;

}

}

这种用于访问私有成员的语法还没有定论，将来很可能会改变。

### A.5.2、getter和setter

新的类语法支持直接为属性定义 getter 和 setter，从而避免了调用 Object.defineProperty() 的麻烦。为属性定义 getter 和 setter 与定义方法类似，只不过要在方法名前加上 get 和 set 关键字。

class Person {

constructor(name, age){

public name = name;

public age = age;

private innerTitle = "";

get title(){

return innerTitle;

}

set title(value){

innerTitle = value;

}

}

sayName(){

alert(this.name);

}

getOlder(years){

this.age += years;

}

}

这个 Person 类为 title 属性定义了一个 getter 和一个 setter。这两个操作 innerTitle 变量的函数都定义在了构造函数中。要为原型属性定义 getter 和 setter，语法相同，但要在构造函数外部定义。

### A.5.3、继承

使用类语法而不是过去那种 JavaScript 语法，最大的好处是容易实现继承。有了类语法，只要使用与其他语言相同的 extends 关键字就能实现继承，而不必去考虑借用构造函数或者原型连缀。

class Employee extends Person {

constructor(name, age){

super(name,age);

}

}

在简单的语法背后，已经自动实现了 原型连缀，而且通过使用 super()函数，也正式支持了借用构造函数。类语法还允许直接将对象指定为其原型，方法就是用 prototype 关键字代 替 extends：

## A.6、模块…ECMAScript6

模块（“命名空间”，“包”）是组织JavaScript应用代码的重要方法。每个模块都是独立的拥有特定的功能。

模块在其自己的顶级环境中运行，因而不会污染导入他的全局执行环境。默认情况下，模块中声明的所有变量、函数、类等都是私有的。**对于应该向外部公开的成员，可以在前面加上export关键字**。创建模块使用module关键字，如：

module MyModule {

//公开这些成员

export let myobject = {};

export function hello(){ alert("hello"); };

//隐藏这些成员

function goodbye(){

//...

}

}

可以在页面或其他模块中使用该模块；也可以导入模块中的一个或者多个成员。导入模块需要使用import命令。

//只导入 myobject import myobject from MyModule;

console.log(myobject);

//导入所有公开的成员

import \* from MyModule;

//列出要导入的成员名

import {myobject, hello} from MyModule;

//不导入，直接使用模块

console.log(MyModule.myobject);

在执行环境能够直接访问到模块的情况下，可以直接调用模块中对外公开的成员。

### 外部模块

通过提供模块所在外部文件的URL，也可以动态加载和导入模块。为此，首先要在模块声明后面加上外部文件的URL在导入模块成员：

module MyModule from "mymodule.js";

import myobject from MyModule;

以上声明会通知 JavaScript 引擎下载 mymodule.js 文件，然后从中加载名为 MyModule 的模块。 请读者注意，这个调用会阻塞进程。换句话说，JavaScript 引擎在下载完外部文件并对其求值之前，不 会处理后面的代码。

如果你只想包含模块中对外公开的某些成员，不想把整个模块都加载进来，可以像下面这样使用 import 指令：

import myobject from "mymodule.js";

总之，模块就是一种组织相关功能的手段，而且能够保护全局作用域不受污染。

1. [[Prototype]] 表示原型对象 [↑](#footnote-ref-0)