# JS 高级程序设计

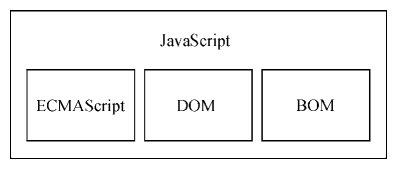
[第 1 章] 什么是 JavaScript

[1.1] JavaScript 历史回顾

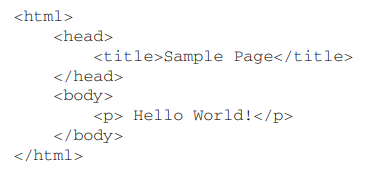
* + ECMA-262 就是 ECMAScript 脚本语言标准

[1.2] JavaScript 实现部分

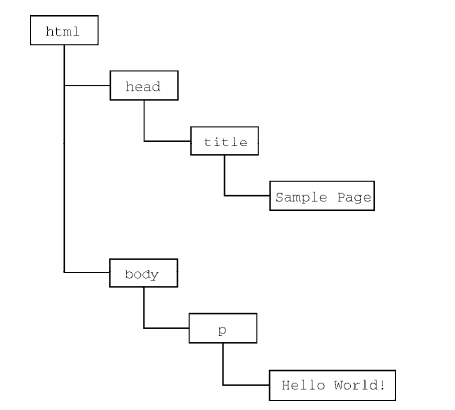
* + 分支主题 1



* + 核心（ECMAScript）
    - 基本层面定义
      * 语法
      * 类型
      * 语句
      * 关键字
      * 保留字
      * 操作符
      * 全局对象
    - ECMAScript 只是对实现 ECMA-262 规范描述的所有方面的一门语言的称呼
    - ECMA-262 将 ECMAScript 作为一个基准来定义，以便在它之上再构建更稳健的脚本语言
    - 宿主环境提供 ECMAScript 的基准实现和与环境自身交互必须的扩展
    - 扩展使用 ECMAScript 核心类型和语法，提供特定于环境的额外功能
    - 宿主环境（host environment）
      * Web 浏览器
      * Node.js
      * Adobe Flash
  + 文档对象模型（DOM）
    - Document Object Model
    - 是一个应用编程接口（API），用于在 HTML 中使用扩展的 XML
    - DOM 将整个页面抽象为一组分层节点
    - 分支主题 4



* + - * 分支主题 1



* + - W3C 指定 DOM 规范
      * DOM Core
        + 提供映射 XML 文档，从而方便访问和操作文档任意部分的方式
      * DOM HTML
        + 对 DOM Core 进行扩展，增加了特定于 HTML 的对象和方法
    - DOM 并非只能通过 JS 访问，很多语言都实现了 DOM，不过对于浏览器来说，DOM 是使用 ECMAScript 实现的
  + 浏览器对象模型（BOM）
    - Browser Object Model
    - 用于支持访问和操作浏览器的窗口
      * 借用 BOM，开发者可以操作浏览器显示页面之外的部分。
    - HTML5 以正式规范的形式涵盖了尽可能多的 BOM 特性
    - BOM 主要针对浏览器窗口和子窗口（frame）
    - BOM 也包括特定于浏览器的扩展
      * 弹出新浏览器窗口的能力
      * 移动、缩放和关闭浏览器的能力
      * navigator 对象
        + 提供关于浏览器的详尽信息
      * location 对象
        + 提供浏览器加载页面的详尽信息
      * screen 对象
        + 提供关于用户屏幕分辨率的详尽信息
      * performance 对象
        + 提供浏览器内存占用、导航行为和时间统计的详尽信息
      * 对 cookie 的支持
      * 其他自定义对象
        + XMLHttpRequest
        + [IE]ActiveXObject

[第 2 章] HTML 中的 JavaScript

[2.1] <script> 元素

* + 属性
    - async
      * 可选
      * 只对外部脚本有效
      * 立即开始下载脚本，不能阻止其他页面动作
      * 其他页面动作
        + 资源下载
        + 等待其他脚本加载
    - defer
      * 可选
      * 支队外部脚本有效
      * 脚本可以延迟到文档完全被解析和显示之后再执行
    - src
      * 可选
      * 包含要执行的代码的外部文件
    - charset
      * 可选
      * 使用 src 属性指定的代码字符集
      * 很少使用
    - crossorigin
      * 可选
      * 配置相关请求的 CORS（跨域资源共享）设置
      * 默认不适用 CORS
      * crossorigin="anonymous"
        + 配置文件请求不必设置凭据
      * crossorigin="use-credentials"
        + 设置凭据标志，意味着出站请求会包含凭据
    - integrity
      * 可选
      * 允许对比接收到的资源和指定的加密签名以验证子资源完整性（SRI，Subresource Integrity）
      * 如果接收到的资源的签名与这个属性指定的签名不匹配，则页面会报错，脚本不会执行
      * 这个属性可以防止内容分发网络（CDN，Content Delivery Network）提过恶意内容。
    - type
      * 可选
      * 代替 language，表示代码块中脚本语言的内容类型（MIME 类型）
      * 属性值
        + text/javascript
        + text/ecmascript
        + application/javascript
        + application/ecmasript
        + mddule

代码会被当做 ES6 模块

只有此种情况，代码中才能出现 import 和 export 关键字

* + - language - 废弃
  + 推迟执行脚本
    - <script> 元素声明 defer 属性
    - 浏览器会立即下载外部脚本，但是延迟执行
    - 执行时机
      * 在浏览器解析完 HTML 之后
      * DOMContentLoaded 事件之前执行
  + 异步执行脚本
    - <script> 元素声明 async 属性
    - 不必等待脚本下载和执行完毕之后再加载页面
    - 不必等待该异步脚本下载和执行后再加载其他脚本
    - 执行时机
      * 保证在页面的 load 事件前执行
      * 可能在 DOMContentLoaded 事件之前或之后
    - 异步脚本不应该在加载期间加载 DOM
  + 动态加载脚本
    - JS 使用 DOM API
      * 向 DOM 中动态添加 script 元素加载指定的脚本
    - 默认以异步方式加载
    - 这种方式获取的资源对浏览器预加载器是不可见的
      * 会影响它们在资源获取队列中的优先级，会影响性能
      * 若想要使其知晓，可以在文档头部显式声明
        + <link rel="preload" href="gibberish.js">

[2.2] 行内代码和外部文件

* + 最佳实践是尽可能将 JS 代码放在外部文件中
  + 理由
    - 可维护性
    - 缓存
      * 浏览器会根据特性的设置缓存所有外部链接的 JS 文件
    - 适应未来

[2.3] 文档模式

* + 使用 doctype 切换文档模式
  + 分类
    - 混杂模式（quirks mode）
    - 标准模式（standards mode）
    - 准标准模式（almost standards mode）

[2.4] <noscript> 元素

* + 作为禁用 JS 的浏览器的降级处理方案
  + 元素中可以包含除 <script> 元素外的任何 HTML 元素
  + 浏览器显示包含在 <noscript> 中内容的情况
    - 浏览器不支持脚本
    - 浏览器对脚本的支持被关闭

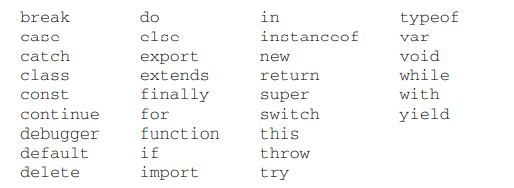
[第 3 章] 语言基础

[3.1] 语法

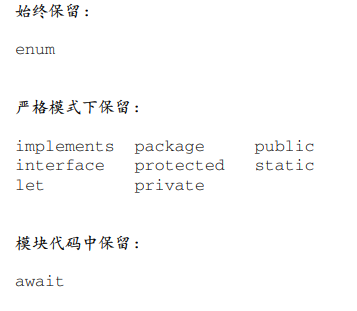
* + 区分大小写
    - ECMAScript 中的一切都区分大小写
  + 标识符
    - 所谓标识符，就是变量、函数、属性或函数参数的名称
    - 规则
      * 第一个字符必须是一个字母、下划线（ \_ ）、美元符号（ $ ）
      * 其余字符可以是字母、下划线、美元符号、数字
    - 标识符中的字母可以是扩展 ASCII（Extended ASCII）中的字母
      * 也可以是 Unicode 的字母字符，但不推荐使用
    - 按照惯例，ECMAScript 标识符使用驼峰小写形式
      * 这种形式与 ECMAScript 内置函数和对象的命名方式一致
  + 注释
    - 采用 C 语言风格的注释
    - 单行注释
      * // 单行注释
    - 块注释
      * /\* 块注释 \*/
  + 严格模式
    - ECMAScript 5 增加了严格模式（ strict mode ）的概念
    - 对脚本启用严格模式
      * 脚本开头声明 "use strict"
    - 对函数启用严格模式
      * 函数开头声明 "use strict"
    - 所有现在浏览器均支持严格模式
  + 语句
    - ECMAScript 中的语句以分号（ ; ）结尾
    - 推荐语句末尾加分号
      * 可以避免输入内容不完整
      * 便于开发者通过删除空行来压缩代码
      * 在某些情况下提升性能
        + 解析器会尝试在合适的位置补上分号以纠正语法错误
    - 多条语句可以合并到一个 C 语言风格的代码块中
      * 代码块由 { } 包裹
    - 对于 if 之类的控制语句只在执行多条语句时要求必须有代码块
      * 最佳实践：始终在控制语句中使用代码块

[3.2] 关键字和保留字

* + 保留字和关键字不能用作标识符或属性名
  + 关键字
    - 分支主题 1

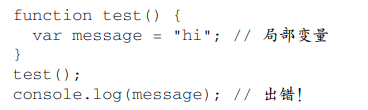


* + 保留字
    - 分支主题 1

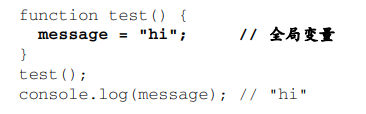


[3.3] 变量

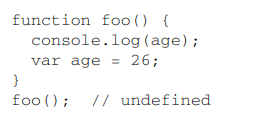
* + ECMAScript 变量是松散类型的
    - 即变量可以用于保存任何类型的数据
    - 每个变量只不过是一个用于保存任意值的命名占位符
  + var 关键字
    - var message
      * 定义了一个名为 message 的变量
      * 可以用它保存任何类型的值
      * 未初始化的情况下，变量值是 undefined
    - var 声明作用域
      * 使用 var 操作符定义的变量会成为包含它的函数的局部变量
        + 分支主题 1



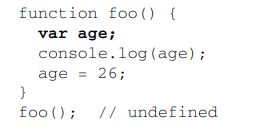
* + - * + test 函数调用后即被销毁，变量 message 也随之销毁
      * 在函数内定义变量时省略 var 操作符，可以创建一个全局变量
        + 分支主题 1



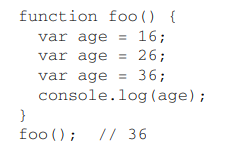
* + - * + 在严格模式下，如上述给未声明的变量赋值，会抛出 ReferenceError
    - var 声明提升
      * 使用 var 声明的变量会自动提升到函数作用域顶部
      * 对于 ECMAScript 两者代码等价
        + 分支主题 1



* + - * + 分支主题 2

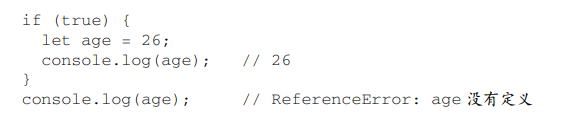


* + - * 所谓"提升"（hoist）会将所有变量声明都拉到函数作用域的顶部
        + 分支主题 1



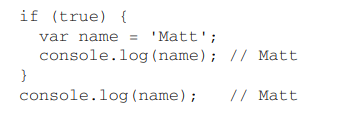
* + - * + 反复多次使用 var 声明同一个变量也没有问题
  + let 声明
    - 与 var 区别
      * 作用域区别
        + let 声明的范围是 块作用域

分支主题 1



* + - * + ver 声明的范围是 函数作用域

分支主题 1



* + - * + 块作用域是函数作用域的子集，因此适用于 var 的作用域限制同样也适用于 let
      * 重复声明区别
        + var 可以重复声明，不限制作用域

分支主题 1



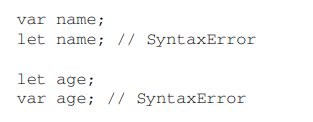
* + - * + let 不允许同一块作用域出现冗余声明

分支主题 1



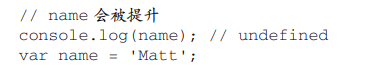
* + - * + 对声明冗余报错不会因混用 let 和 var 而受影响

分支主题 1



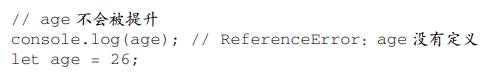
* + - * 声明提升区别
        + var 声明变量会有声明提升

分支主题 1



* + - * + let 声明变量没有声明提升，存在“暂时性死区”

分支主题 1



在 let 声明之前的执行瞬间被称为“暂时性死区”（temporal dead zone）

在暂时性死区阶段，引用任何后面才声明的变量的操作都会抛出 ReferenceError

* + - * 全局声明区别
        + 在全局作用域使用 var 声明变量，该变量会成为 window 对象的属性

分支主题 1



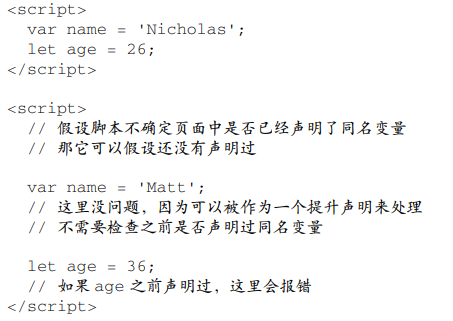
* + - * + 在全局作用域使用 let 声明变量，该变量不会成为 window 对象的属性

分支主题 1



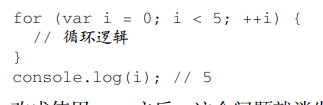
let 声明仍然是在全局作用域中发生的，相应变量会在页面的生命周期内存续

* + - * 条件声明区别
        + 在使用 var 声明变量时，由于声明会被提升，JS 引擎会自动将多余的声明在作用域顶部合并为一个声明。
        + 因为 let 的作用域是块，所以不可能检查前面是否已经用 let 声明过同名变量，同时也就不可能在没有声明的情况下声明它
        + 分支主题 3



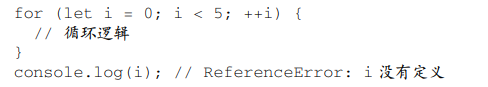
* + - * for 循环中的迭代变量区别
        + for 循环中使用 var 定义的迭代变量会渗透到循环体外部

分支主题 1

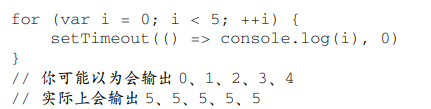


* + - * + for 循环中使用 let 定义的迭代变量仅限于 for 循环体内部

分支主题 1



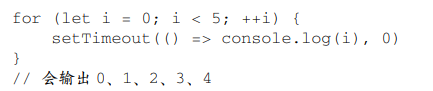
* + - * for 循环中嵌套 setTimeout() 区别
        + 分支主题 1



退出循环时，迭代变量保存的是导致循环退出的值：5

在之后执行宏任务 setTimeout() 逻辑时，所有的 i 都是同一个变量，因而输出的都是同一个最终值

* + - * + 分支主题 2



JS 引擎在后台会为每个迭代循环声明一个新的迭代变量

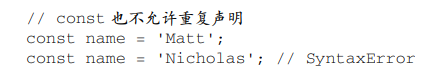
每个 setTimeout 引用的都是不同的变量实例，所以 console。log() 输出的是执行过程中每个迭代变量的值。

每次迭代，声明一个独立变量实例的行为适用于所有风格的 for 循环

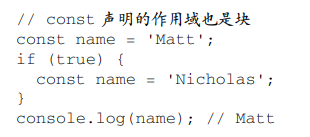
for-in

for-of

* + const 声明
    - const 的行为与 let 基本相同
      * 不允许重复声明
        + 分支主题 1



* + - * 声明作用域为块作用域
        + 分支主题 1



* + - 唯一一个重要的区别
      * 使用 const 声明变量时必须同时初始化变量
      * 尝试修改 const 声明的变量会导致运行时错误
        + 分支主题 1



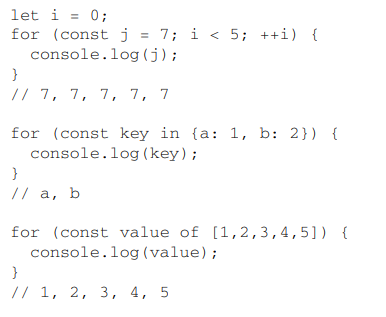
* + - const 声明的限制只适用于它所指向的变量的引用
      * 如果 const 变量引用的是一个对象，那么修改这个对象内部的属性并不违反 const 限制
      * 分支主题 2



* + - 不能用于声明迭代变量
      * 因为迭代变量会自增
      * 分支主题 2



* + - 可以使用 const 声明一个不会被修改的 for 循环变量
      * 分支主题 1



* + 声明风格及最佳实践
    - 不使用 var
    - const 优先，let 次之

[3.4] 数据类型

* + ECMAScript 有 7 种简单数据类型（也称为原始类型）
    - Undefined
    - Null
    - Boolean
    - Number
    - String
    - Symbol
    - BigInt
  + 一种复杂数据类型
    - Object（对象）
      * 无序名值对的集合
  + ECMAScript 中不能定义自己的数据类型
  + typeof 操作符
    - 因为 ECMAScript 的类型系统是松散的，所以需要一种手段来确定任意变量的数据类型
    - 对一个值使用 typeof 操作符会返回下列字符串之一
      * "undefined"
        + 表示值未定义
      * "boolean"
        + 表示值为布尔值
      * "string"
        + 表示值为字符串
      * "number"
        + 表示值为数值
      * "object"
        + 表示值为对象（而不是函数）
        + null

被认为是一个对空对象的引用

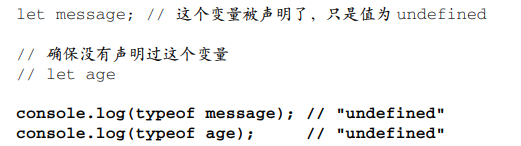
* + - * "function"
        + 表示值为函数

严格来讲，函数在 ECMAScript 被认为是对象，并不代表一种数据类型

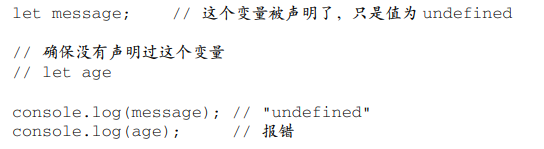
但由于其拥有自己特殊的属性

有必要通过 typeof 操作符来区分函数和其他对象

* + - * "symbol"
        + 表示值为符号
      * "bigint"
        + 表示值为大数
  + Undefined 类型
    - Undefined 类型只有一个值，就是特殊值 undefined
    - 当使用 var 或 let 声明变量但没有初始化时
      * 就相当于给变量赋予了 undefined 值
    - 对没有声明过的变量调用 typeof，返回"undefined"
      * 分支主题 1



* + - 注意：包含 undefined 值的变量与未定义变量的区别
      * 分支主题 1



* + - * + 对于未声明的变量，只能执行下述有用的操作

对它调用 typeof

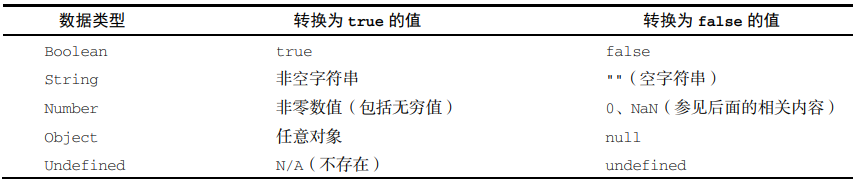
对它调用 delete

严格模式下报错

* + Null 类型
    - Null 类型只有一个值，就是特殊值 null
    - 逻辑上讲，null 值表示一个空对象指针
      * typeof null = "object"
      * 在定义将来要保存对象值的变量时，建议使用 null 进行初始化
    - undefined 值由 null 值派生而来
      * 因此 ECMA-262 将它们定义为表面上相等
        + 分支主题 1



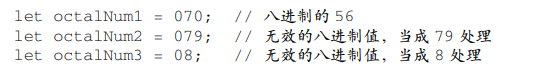
* + - * + 操作符“==”会为了比较而转换它的操作数
  + Boolean 类型
    - 两个字面值
      * true
      * false
    - 布尔值字面量 true 和 false 是区分大小写的，因此 True 和 False（及其他大小混写形式） 是有效的标识符，但不是布尔值
    - 不同类型与布尔值之间的转换规则
      * 分支主题 1



* + - * if 等流控制语句会自动执行其他类型值到布尔值的转换
  + Number 类型
    - Number 类型使用 IEEE 754 格式表示整数和浮点数（在某些语言中也叫双精度值）
    - 不同进制的 number 初始化
      * 分支主题 1



* + - * 分支主题 2



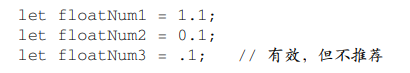
* + - * + 在严格模式下无效
        + 会导致 JS 引擎抛出语法错误
        + ES6 中的八进制通过前缀 0o 来表示

这是合法的

* + - * 分支主题 3



* + - * 使用八进制和十六进制格式创建的数值在所有数学操作中都被视为十进制数值
    - 由于 JS 保存数值的方式，实际中可能存在正零（ +0 ）和负零（ -0 ）
      * 正零和负零在所有情况下都被认为是等同的
    - 浮点值
      * 分支主题 1



* + - * + 要定义浮点值，数值中必须包含小数点，而且小数点后面必须有一个数字。
        + 因为存储浮点值使用的内存空间是存储整数值的两倍，所以 ECMAScript 总是想法设法把值转换为整数值

分支主题 1



* + - * 对于非常大或非常小的数值，浮点值可以用科学记数法来表示
        + 分支主题 1



* + - * + 0.000 000 000 000 000 03 = 3e-17
      * 浮点值的精度最高可达 17 为小数，但在算术计算中远不如整数精确
        + 0.1 + 02 ≠ 0.3（0.300 000 000 000 000 04）
    - 值的范围
      * ECMAScript 并不支持表示这个世界上的所有数值
        + 可以表示的最小值保存在 Number.MIN\_VALUE

在多数浏览器中是 5e-324

* + - * + 可以表示的最大值保存在 Number.MAX\_VALUE

在多数浏览器中是 1.797 693 134 862 315 7e+308

* + - * 如果某个计算得到的数值结果超出了 JS 可以表示的范围
        + 任何无法表示的正数以 Infinity(正无穷大)表示

Number.NEGATIVE\_INFINITY

* + - * + 任何无法表示的负数以 -Infinity(负无穷大)表示

Number.POSITIVE\_INFINITY

* + - NaN
      * 不是数值（Not a Number）
        + 表示本来要返回数值的操作失败了（而不是抛出错误）
      * 分支主题 2



* + - * + 在其他语言中通常会导致错误，从而终止代码执行
      * 分支主题 3

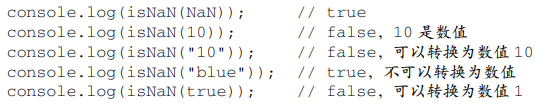


* + - * 特性
        + 任何涉及 NaN 的操作符始终返回 NaN
        + NaN 不等于包括 NaN 在内的任何值

为此，ECMAScript 提供了 isNaN() 函数

接收一个参数，可以是任意数据类型

分支主题 2



isNaN() 可以用于测试对象

首先调用对象的 valueOf() 方法

确定返回的值是否可以转换为数值

再调用 toString() 方法

并测试其返回值

* + - 数值转换
      * 有 3 个函数可以将非数值转换为数值
      * Number()
        + 转型函数
        + 可以用于任何数据类型
        + 规则

布尔值

true 转换为 1

false 转换为 0

数值

直接返回

null

返回 0

undefined

返回 NaN

字符串

如果字符串仅由数值字符构成，包括数值字符前带加、减号的情况

转换为一个十进制数值

Number("1") == 1

Number("123") == 123

Number("011") == 11

忽略前面的零

如果字符串仅由有效的浮点值格式如"1.1" 构成

转换为相应的浮点值

忽略前面的零

如果字符串包含有效的十六进制格式如"0xf"

转换为与该十六进制值对应的十进制整数值

如果是空字符串（不包括字符）

返回 0

如果字符串包含除上述情况之外的其他字符

返回 NaN

对象

调用 valueOf() 方法

按照上述规则转换返回的值

如果转换结果是 NaN

调用 toString() 方法

按照转换字符串的规则进行转换

* + - * + + 操作符与 Number() 函数遵循相同的转换规则
      * parseInt()
        + 通常再需要得到整数时可以优先使用 parseInt() 函数，而不是 Number()
        + parseInt() 函数更专注于字符串是否包含数值
        + 字符串最前面的空格会被忽略，从第一个非空格字符开始转换
        + 如果第一个字符不是数值字符、加号或减号

立即返回 NaN

空字符也会返回 NaN

* + - * + 如果第一个字符是数值字符、加号或减号

继续依次检测每个字符

直到字符串末尾

直到碰到非数值字符

parseInt("1234blue") == 1234

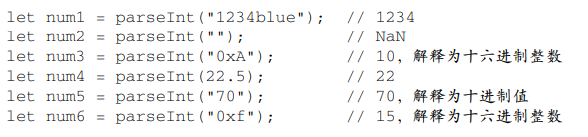
parseInt("22.5") == 22

* + - * + 如果第一个字符是数值字符，parseInt() 函数能识别不同的整数格式

"0x" 开头则解释为十六进制整数

"0o"/"0" 开头则解释为八进制整数

* + - * + 分支主题 7

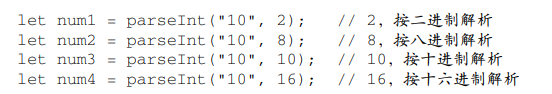


* + - * + 第二个参数使用

分支主题 1



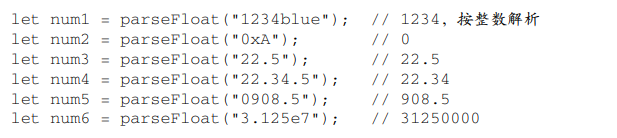
分支主题 2



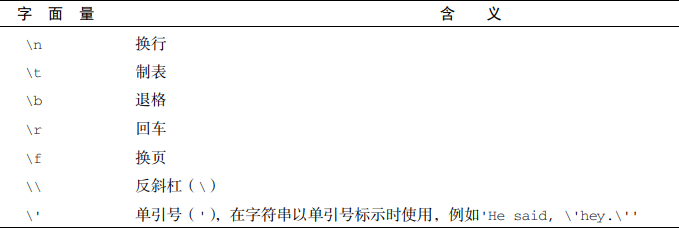
* + - * parseFloat()
        + 工作方式与 parseInt() 函数类似
        + 始终忽略字符串开头的零
        + 解析到字符产结尾或者解析到一个无效的浮点数值字符为止
        + 只解析十进制数

不可以指定底数（第二个参数）

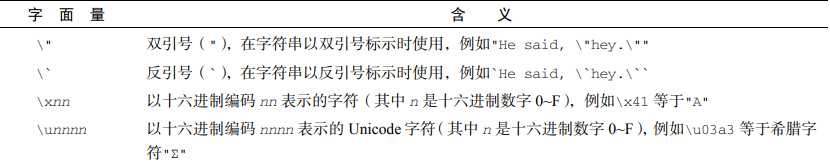
* + - * + 分支主题 5



* + String 类型
    - String 数据类型表示零或多个 16 位 Unicode 字符序列
      * 可以使下述符号标识
        + 双引号（"）
        + 单引号（'）
        + 反引号（`）
      * 与某些语言中使用不同的引号会改变字符串的解释方式不同
        + ECMAScript 语法中表示字符串的引号没有区别
    - 字符字面量
      * 用于表示非打印字符或者有其他用途的字符
        + 分支主题 1



* + - * + 分支主题 2



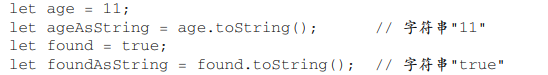
* + - * 转义序列表示一个字符，所以只算一个字符
    - 字符串的特点
      * ECMAScript 中的字符串是不可变的（immutable）
      * 要修改某个变量中的字符串值，必须先销毁原先的字符串，然后将包含新值的另一个字符串保存到该变量
    - 转换为字符串
      * toString() 方法
        + 几乎所有值都具有该方法

toString() 可见于数值、布尔值、对象和字符串值（返回自身的一个副本）

null 和 undefined 值没有 toString() 方法

* + - * + 该方法的唯一用途就是返回当前值的字符串等价物

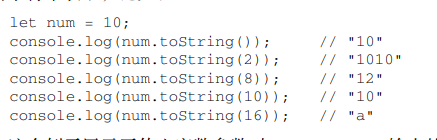
分支主题 1



* + - * + 在对数值调用该方法时，可以接收一个底数参数

即以什么底数来输出数值的字符串表示

分支主题 2



* + - * + 使用该方法确定值的类型

如果值有 toString() 方法

则调用该方法（不传参数）并返回结果

如果值是 null

返回 "null"

如果值是 undefined

返回 "undefined"

* + - * 使用 + 操作符给一个值加上一个空字符串 " "，亦可以将其转换为字符串
    - 模板字面量
      * ECMAScript 6 新增了使用模板字面量定义字符串的能力
      * 模板字面量保留换行字符，可以跨行定义字符串
      * 顾名思义，模板字面量在定义模板时特别有用
    - 字符串插值
      * 模板字面量最常用的一个特性：支持字符串插值
      * 技术上讲，模板字面量不是字符串，而是一种特殊的 JS 句法表达式
        + 求值得到的是字符串
        + 模板字面量在定义时立即求值并转换为字符串实例

任何插入的变量也会从它们最接近的作用域中取值

* + - * 任何插入的值都会使用 toString() 强制转换为字符串类型，任何 JS 表达式都可以用于插值
        + 嵌套的模板字符串无须转义

分支主题 1



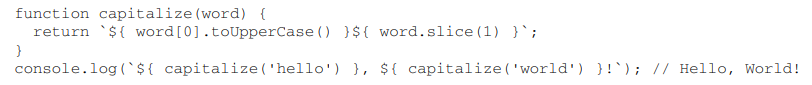
* + - * + 将表达式转换为字符串时会调用 toString() 方法

分支主题 1



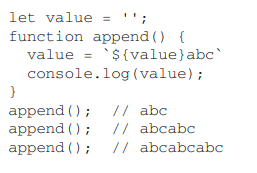
* + - * + 在插值表达式中可以调用函数和方法

分支主题 1

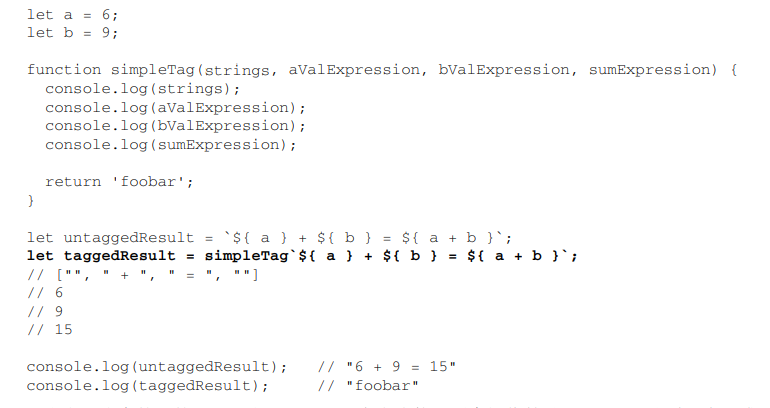


* + - * + 模板也可以插入自己之前的值

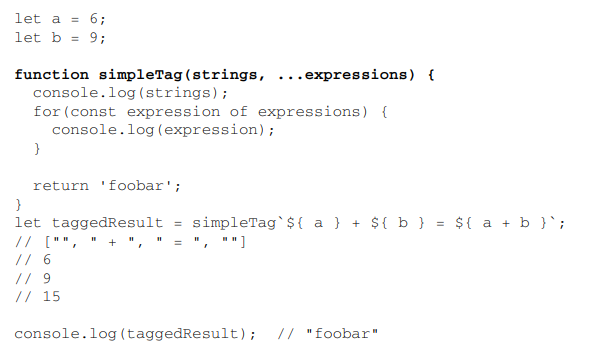
分支主题 1



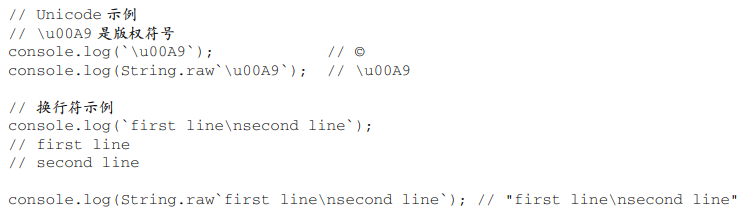
* + - 模板字面量标签函数
      * 模板字面量支持自定义 标签函数（ tag function ）
        + 标签函数可以自定义插值行为
        + 分支主题 2



* + - * + 分支主题 3

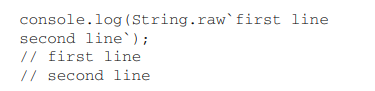


* + - 原始字符串
      * 使用模板字面量可以直接获取原始的模板字面量内容（如换行符或 Unicode 字符）
      * 使用默认的 String.raw 标签函数可以做到上述需求
        + 分支主题 1

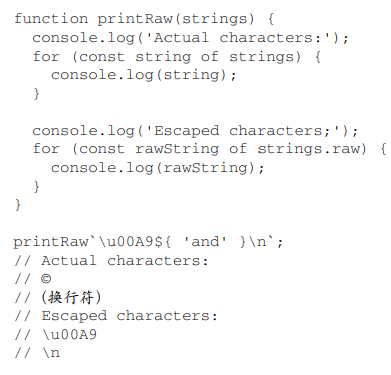


* + - * + 对于实际的换行符来说是无效的

分支主题 1



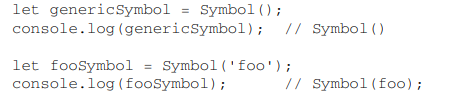
* + - * 可以通过标签函数的第一个参数，即字符串数组的 .raw 属性获取每个字符串的的原始内容
        + 分支主题 1



* + Symbol 类型
    - Symbol（符号）是 ECMAScript 6 新增的数据类型。
    - Symbol 属于原始值，其 Symbol 实例是唯一的、不可变的
    - Symbol 的用途是确保对象属性使用唯一标识符，不会发生属性冲突的危险
      * Symbol 并不是为了提供私有属性的行为才增加的
      * Symbol 就是用来创建唯一记号，进而用作非字符串形式的对象属性
    - Symbol 的基本用法
      * 符号需要使用 Symbol() 函数进行初始化。
        + 分支主题 1

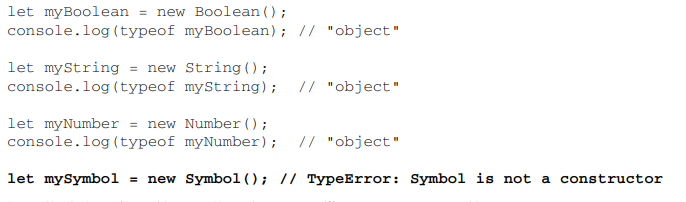


* + - * 调用 Symbol() 函数时，可以传入一个字符串参数作为对符号的描述（description）
        + 将来可以通过该字符串调试代码
        + 但是这个字符串参数与符号定义或标识完全无关
      * 符号没有字面量语法
        + 只要创建 Symbol() 实例并将其用作对象的新属性，就可以保证其不会覆盖已有的对象属性
        + 分支主题 2



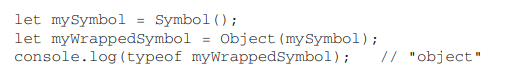
* + - * Symbol() 函数不能作为构造函数使用
        + 目的是避免创建符号包装对象

分支主题 1



* + - * + 若确实想使用符号包装对象，可以借用 Object() 函数

分支主题 1



* + - 使用全局符号注册表
      * 如果运行时的不同部分需要共享和重用符号实例
        + 可以用一个字符串作为键
        + 在全局符号注册表中创建并重用符号
      * 需要使用 Symbol.for() 方法
        + 分支主题 1



* + - * + Symbol.for() 对每个字符串键都执行幂等操作

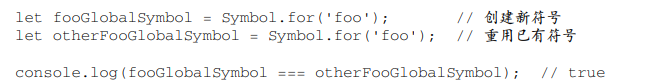
若全局注册表不存在该字符串键对应的符号

则生成新符号，并添加到全局注册表中

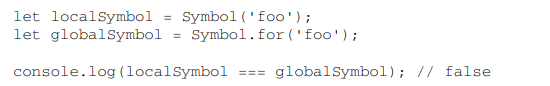
若全局注册表存在该字符串键对应的符号

则直接该符号实例

分支主题 3



* + - * 即使采用相同的符号描述，在全局注册表中定义的符号与使用 Symbol() 定义的符号并不等同
        + 分支主题 1



* + - * 全局注册表中的符号必须使用字符串键来创建
        + 作为参数传给 Symbol.for() 的任何值都会被转换为字符串
        + 注册表中使用的键同时也会被用作符号描述
        + 分支主题 3

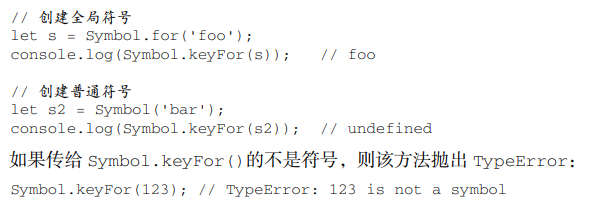


* + - * 可以使用 Symbol.keyFor() 查询全局注册表
        + 该方法接收符号

返回该全局符号对应的字符串键

若该符号不是全局符号，则返回 undefined

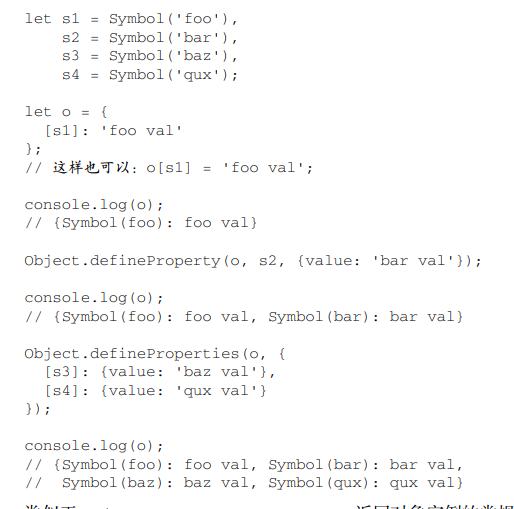
* + - * + 分支主题 2



* + - 使用符号作为属性
      * 凡是可以使用字符串或数值作为属性的地方，都可以使用符号
        + 包括 对象字面量属性

对象字面量只能在计算属性语法中使用符号作为属性

* + - * + 包括 Object.defineProperty() / Object.defineProperties() 定义的属性
        + 分支主题 3



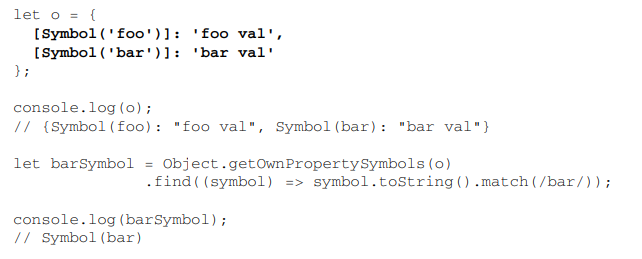
* + - * Object.getOwnPropertyNames()
        + 返回对象实例的常规属性数组
      * Object.getOwnPropertySymbols()
        + 返回对象实例的符号属性数组
      * 上述两个方法的返回值彼此互斥
      * Object.getOwnPropertyDescriptors()
        + 返回同时包含常规和符号属性描述符的对象属性
      * Reflect.ownKeys()
        + 返回两种类型的键
      * 分支主题 7



* + - * 由于符号属性是对内存中符号的一个引用
        + 直接创建并用作属性的符号不会丢失
        + 但若没有显式地保存对这些属性的引用

那么必须遍历对象的所有符号属性才能找到对应的属性键

* + - * + 分支主题 3

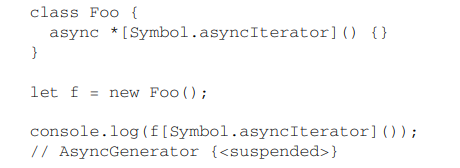


* + - 常用内置符号
      * ECMAScript 6 引入了一批常用内置符号（ well-know symbol）
        + 用于暴露语言内部行为
        + 开发者可以直接访问、重写或模拟这些行为
      * 内置符号最重要的用途之一就是重新定义它们
        + 从而改变原生结构的行为
        + 例如，for-of 循环会在相关对象上使用 Symbol.iterator 属性

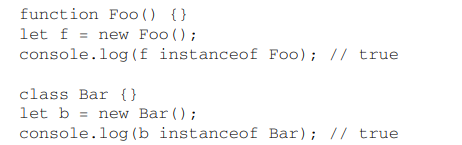
可以通过在自定义对象上重新定义 Symbol.iterator 值

以改变 for-of 在迭代该对象时的行为

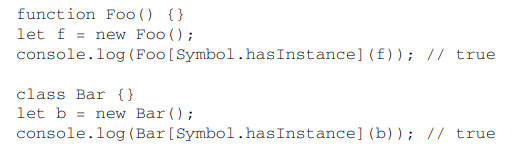
* + - * 在 ECMAScript 规范中，@@iterator 指的是 Symbol.iterator
    - Symbol.asyncIterator
      * 作为属性表示一个方法
      * 该方法返回对象默认的 AsyncIterator。由 for-wait-of 语句使用
        + 这个符号表示实现异步迭代器 API
      * for-wait-of 循环会利用该函数执行异步迭代操作
        + 循环时，它们会调用以 Symbol.asyncIterator 为键的函数
        + 期望返回一个实现迭代器 API 的对象



* + - Symbol.hasInstance
      * 作为属性表示一个方法
      * 该方法决定一个构造器对象是否认可一个对象是它的实例
        + 由 instanceof 操作符使用
        + iinstanceof 操作符可以用于确定一个对象实例的原型链上是否有原型



* + - * + 在 ES6 中，instanceof 操作符会使用 Symbol.hasInstance() 函数确定关系

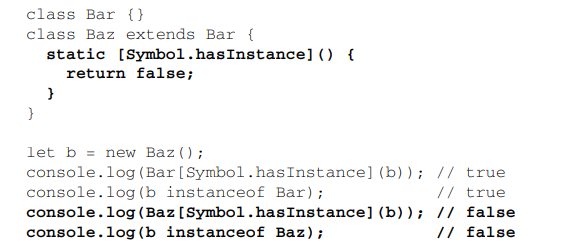


该属性定义在 Function 的原型上

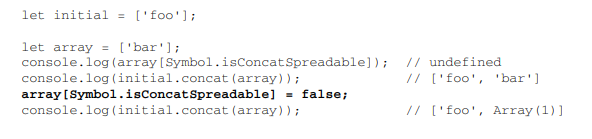
因此默认在所有函数和类上都可以调用

由于 instanceof 操作符会在原型链上寻找该属性定义

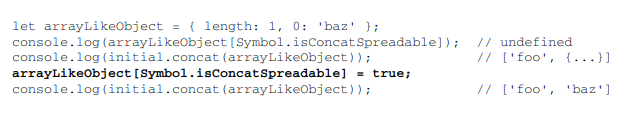
因此可以在继承的类上通过静态方法重新定义该函数



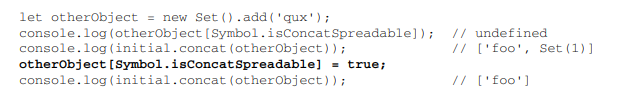
* + - Symbol.isConcatSpreadable
      * 作为属性表示：一个布尔值
        + 如果值为 true，则意味着对象应该用 Array.protptype.concat() 打平其数组元素
        + ES6 的 Array.prototype.concat() 方法会根据接收到的对象类型选择如何将一个类数组对象拼接成数组实例
      * 与上述介绍的属性一样，覆盖 Symbol.isConcatSpreadable 的值可以修改该行为
      * 数组对象默认情况下会被打平到已有的数组
        + 该属性返回 false 或 假值，会导致整个对象被追加到数组末尾



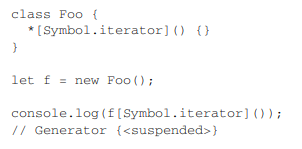
* + - * 类数组对象默认情况下会被追加到数组末尾
        + 该属性返回 true 或 真值，会导致该类数组对象被打平到数组实例



* + - * 其他非类数组对象在 Symbol.isConcatSpreadable 被设置为 true 的情况下，将被忽略



* + - Symbol.iterator
      * 作为属性表示一个方法
      * 该方法返回对象默认的迭代器
        + 由 for-of 语句使用
        + 该方法符号表示实现迭代器 API 的函数
        + 类似 for-of 循环的语言结构会利用该函数执行迭代操作，
          会调用以 Symbol.iterator 为键的函数，
          并默认这个函数会返回一个实现迭代器 API 的对象
        + 返回对象是实现该 API 的 Generator

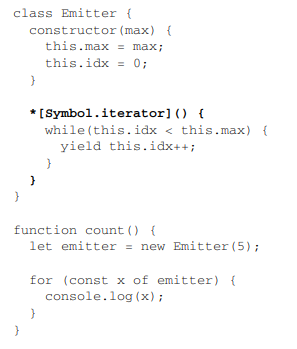


* + - * + 技术上，由 Symbol.iterator 函数生成的对象应该通过其 next() 方法陆续返回值

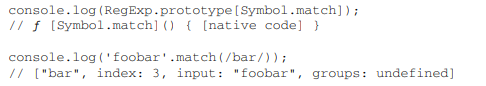
可以通过显式调用 next() 返回值

可以隐式通过生成器函数返回值

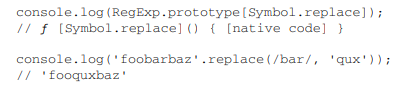




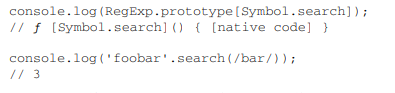
* + - Symbol.match
      * 作为符号表示一个方法
      * 正则表达式方法，使用正则表达式去匹配字符串
        由 String.prototype.match() 方法使用
        + String.prototype.match() 方法会使用以 Symbol.match 为键的函数来对正则表达式求值。
        + 正则表达式的原型上默认有该函数的定义，因此所有正则表达式实例默认是这个 String 方法的有效参数



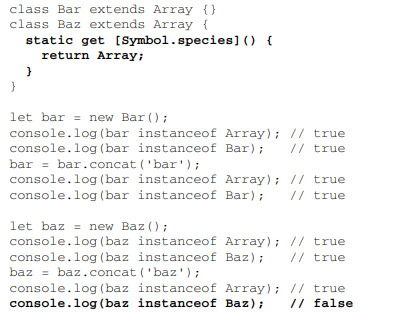
* + - * 给这个方法传入非正则表达式值会导致该值被转换为 RegExp 对象。
        如果想改变这种行为，让方法直接使用参数，可以重新定义 Symbol.search 函数以取代默认对正则表达式求值的行为，
        从而让search()方法使用非正则表达式实例
    - Symbol.replace
      * 作为符号表示一个方法
      * 正则表达式方法，该方法替换一个字符串中匹配的子串
        由 String.prototype.replace() 方法使用
        + String.prototype.replace() 方法会使用以 Symbol.replace 为键的函数来对正则表达式求值
        + 正则表达式的原型上默认有这个函数的定义，因此所有正则表达式实例默认是这个 String 方法的有效参数



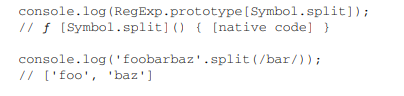
* + - * 给这个方法传入非正则表达式值会导致该值被转换为 RegExp 对象。
        如果想改变这种行为，让方法直接使用参数，可以重新定义 Symbol.search 函数以取代默认对正则表达式求值的行为
        从而让search()方法使用非正则表达式实例
    - Symbol.search
      * 作为符号表示一个方法
      * 正则表达式方法，该方法返回字符串中匹配正则表达式的索引
        由 String.protptype.search() 方法使用
        + String.prototype.search() 方法会使用以 Symbol.search 为键的函数来对正则表达式求值
        + 正则表达式的原型上默认有这个函数的定义，因此所有正则表达式实例默认是这个 String 方法的有效参数



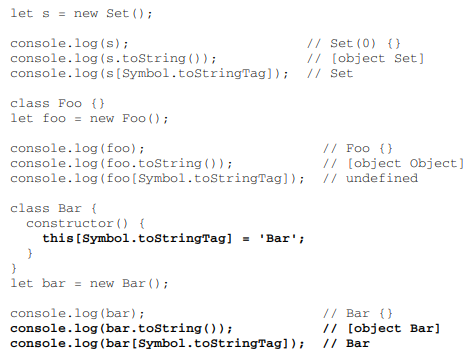
* + - * 给这个方法传入非正则表达式值会导致该值被转换为 RegExp 对象。
        如果想改变这种行为，让方法直接使用参数，可以重新定义 Symbol.search 函数以取代默认对正则表达式求值的行为
        从而让search()方法使用非正则表达式实例
    - Symbol.species
      * 作为符号表示一个方法
      * 该函数作为创建派生对象的构造函数
      * 该属性在内置类型中最常用，
        用于内置类型实例方法的返回值暴露实例化派生对象的方法
      * 用 Symbol.species 定义静态的获取器（getter）方法，
        可以覆盖新创建实例的原型定义



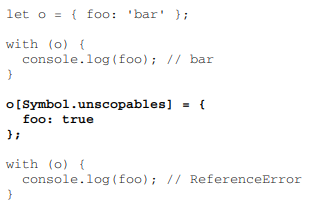
* + - Symbol.split
      * 作为符号表示一个方法
      * 一个正则表达式方法，该方法在匹配正则表达式的索引位置拆分字符串
        由 String.prototype.split()方法使用
        + String.prototype.split() 方法会使用以 Symbol.split 为键的函数来对正则表达式求值
        + 正则表达式的原型上默认有这个函数的定义，因此所有正则表达式实例默认是这个 String 方法的有效参数



* + - * 给这个方法传入非正则表达式值会导致该值被转换为 RegExp 对象。
        如果想改变这种行为，让方法直接使用参数，可以重新定义 Symbol.search 函数以取代默认对正则表达式求值的行为
        从而让search()方法使用非正则表达式实例
    - Symbol.toPrimitive
      * 作为符号表示一个方法
      * 该方法将对象转换为相应的原始值。
        由 ToPrimitive 抽象操作使用
      * 很多内置操作都会尝试强制将对象转换为相应的原始值
        包括 字符串、数值和未指定的原始类型
      * 对于一个自定义对象实例，通过这个实例的 Symbol.toPrimitive 属性上定义一个函数改变默认行为
    - Symbol.toStringTag
      * 作为符号表示一个字符串
      * 该字符串用于创建对象的默认字符串描述
        由内置方法 Object.prototype.toString() 使用
      * 通过 toString()方法获取对象标识时，
        会检索由 Symbol.toStringTag 指定的实例标识符，
        默认为"Object"
      * 内置类型已经指定了这个值，但自定义类实例还需要明确定义



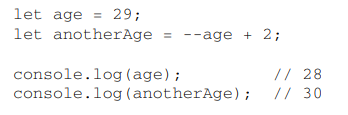
* + - Symbol.unscopables
      * 作为符号表示一个对象
      * 该对象所有的以及继承的属性，都会从关联对象的 with 环境绑定中排除
      * 设置这个符号并让其映射对应属性的键值为 true，
        就可以阻止该属性出现在 with 环境绑定中



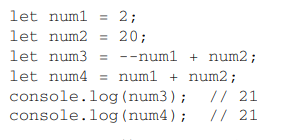
* + Object 类型
    - ECMAScript 中的对象是一组数据和功能的集合
    - ECMAScript 中的 Object 是派生其他对象的基类，
      Object 类型的所有属性和方法在派生的对象上同样存在
    - 每个 Object 实例都有如下属性和方法
      * constructor
        + 用于创建当前对象的函数
      * hasOwnProperty(propertyName)
        + 用于判断当前对象实例（不是原型）上是否存在给定的属性
        + 待检查的属性名必须是字符串类型或符号类型
      * isPrototypeOf(object)
        + 用于判断当前对象是否为另一个对象的原型
      * propertyIsEnumerable(propertyName)
        + 用于判断给定的属性是否可以使用 for-in 语句枚举
        + 属性名必须是字符串类型或符号类型
      * toLocalString()
        + 返回对象的字符串表示，
          该字符串反映对象所在的本地化执行环境
      * toString()
        + 返回对象的字符串表示
      * valueOf()
        + 返回对象对应的字符串、数值或者布尔值表示
        + 通常与 toString() 的返回值相同
    - ECMA-262 中对象的行为不一定适合所有 JS 中的其他对象，如宿主环境定义和提供的 BOM 和 DOM 对象。
      宿主对象不受 ECMA-262 约束

[3.5] 操作符

* + ECMA-262 描述了一组可用于操作数据值的操作符
    - 数学操作符
    - 位操作符
    - 关系操作符
    - 相等操作符
    - ...
  + ECMAScript 中的操作符可以用于各种值，
    包括字符串、数值、布尔值、对象
    - 操作符应用于对象时，
      通常会调用 valueOf() 或 toString()，
      以取得可以计算的值
  + 一元操作符（unaryt operator）
    - 只操作一个值的操作符
    - 递增/递减操作符
      * 前缀版
        + 变量使用前缀操作符，其值都会在语句被求值之前改变



* + - * + 前缀递增和递减在语句中的优先级是相等的



* + - * 后缀版
        + 变量使用后缀操作符，其值都会在语句被求值之后改变
        + 后缀递增和递减在语句中的优先级是相等的
      * 规则
        + 字符串

有效的数值形式

转换为数值后，
对数值进行操作

变量类型从 string 变为 number

无效的数值形式

将变量值设置为 NaN

变量类型从 string 变为 number

* + - * + 布尔值

fasle

转换为 0，
对数值进行操作

变量类型从布尔值变为数值

true

转换为 1，
对数值进行操作

变量类型从布尔值变为数值

* + - * + 浮点值

+ 1 / -1

* + - * + 对象

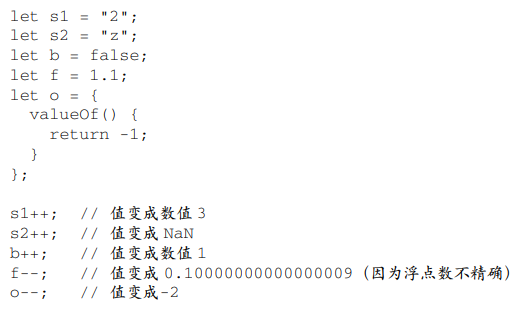
调用其 valueOf() 方法取得可以操作的值

对得到的值应用上述规则

若该值为 NaN，调用 toString() 后，在此应有其他规则

变量类型从 object 变为 number





* + - 一元加和减
      * 一元加（+）置于变量前，对数值没有任何影响
      * 一元减（-）置于变量前，对数值取相反值
      * 应用于非 number 类型时，执行与使用 Number() 转型函数一样的类型转换
  + 位操作符
    - 用于数值的底层操作，
      操作内存中表示数据的比特（位）
    - ECMAScript 中所有数值都以 IEEE 754 64位格式存储，
      位操作会先将数值转换为 32 位整数，再进行位操作，之后再将结果转换为64位
      * 有符号整数
        + 使用前 31 位表示整数值
        + 第32位表示数值的符号，即符号位（ sign bit ）

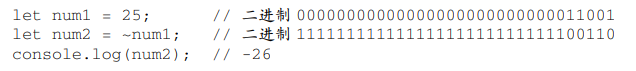
0 表示 正

正值以真正的二进制格式存储，
31 位中的每一位都代表 2 的幂

1 表示 负

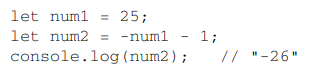
负值以补码的二进制编码存储

* + - * 副作用
        + 特殊值 NaN 和 Infinity 在位操作中都会被当作 0 处理
    - 使用位操作符操作非数值
      * 首先，JS 自动使用 Number() 将该值转换为数值
      * 再应用位操作
      * 最终结果是数值
    - 按位非
      * 操作符使用波浪符（~）表示
      * 返回数值的一补数

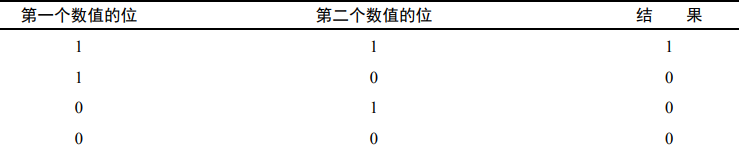


在数值的底层表示上完成，速度更快

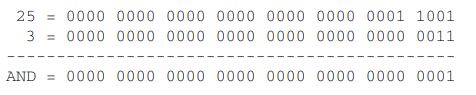
* + - * + 最终效果为：对数值取反并减 1



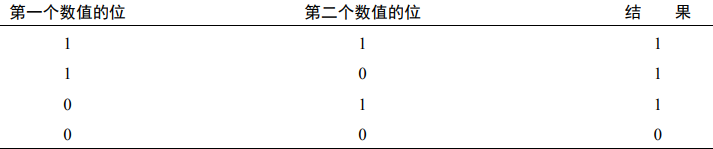
* + - 按位与
      * 操作符使用和号（&）表示
      * 需要两个操作数
        + 将两个操作数的每一位对齐，
          然后基于真值表中的规则，
          对每一位执行相应的与操作





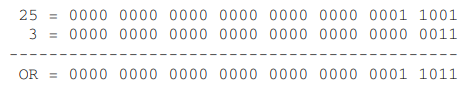


* + - 按位或
      * 操作符使用管道符（|）表示
      * 需要两个操作数

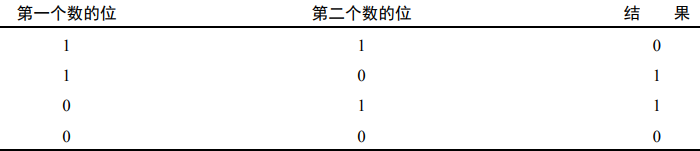






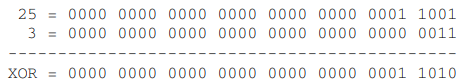


* + - 按位异或
      * 使用脱字符（^）表示
      * 需要两个操作数









* + - 左移
      * 使用符号（<<）表示
      * 按照指定的位数将数值的所有32位向左移动



* + - * + 左移会保留操作数的符号
    - 右移
      * 使用符号（>>）表示
      * 按照指定的位数将数值的所有32位向右移动



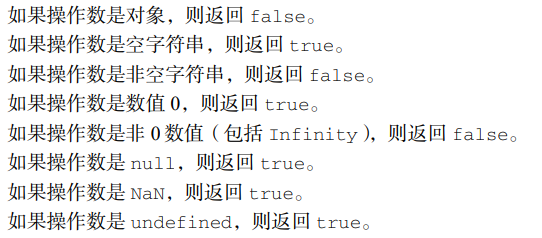
* + - * + 右移会保留操作数的符号
    - 无符号右移
      * 使用符号（>>>）表示
      * 将数值的所有32位都向右移
        + 对于正值，与右移操作符无差别

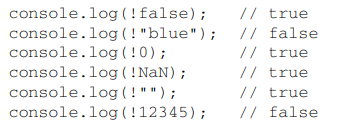


* + - * + 对于负值，有时与右移操作符差别很大

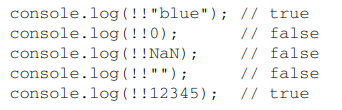


* + 布尔操作符
    - 逻辑非
      * 操作符使用叹号（!）表示
      * 可应用给 ECMAScript 中的任何值
        + 始终返回布尔值
      * 逻辑非操作符首先将操作数转换布尔值，
        然后再对其取反
      * 规则

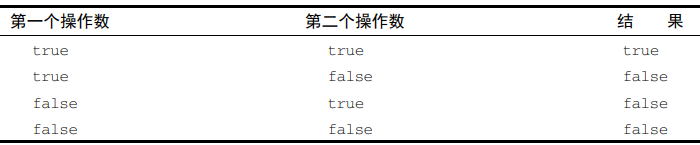




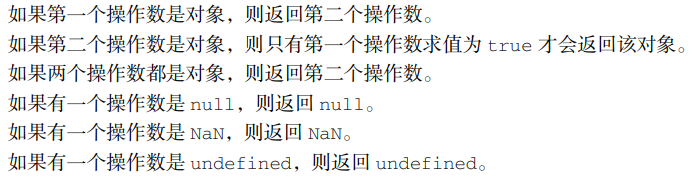
* + - * 可用于将任意值转换为布尔值
        + 同时使用（!!），相当于调用转型函数 Boolean()



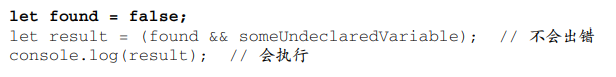
* + - 逻辑与
      * 操作符使用符号（&&）表示
      * 需要两个操作数（任何类型）
        + 遵循真值表



* + - * 规则

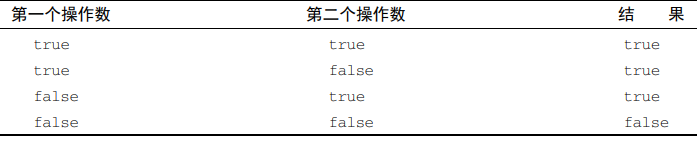


* + - * 逻辑与操作符是一种短路操作符
        + 如果第一个操作数求值为 false，
          第二个操作数就不再会被求值

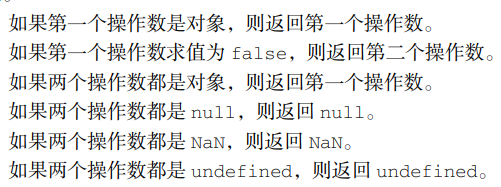


尽管第二个操作数未定义，但由于 JS 引擎不会执行，所以并没有报错

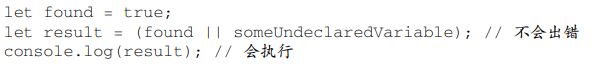
* + - * + 如果第一个操作数是 false，
          无论第二个操作数是什么值，
          结果不可能等于 true
    - 逻辑或
      * 操作符使用符号（||）表示
      * 需要两个操作数（任何类型）
        + 遵循真值表



* + - * 规则



* + - * 逻辑或操作符是一种短路操作符
        + 如果第一个操作数求值为 true，
          第二个操作数就不再会被求值



* + 乘性操作符
    - 若乘性操作符有不是数值的操作数，则该操作数会在后台被使用 Number() 转换为数值
    - 乘法操作符
      * 操作符使用符号（\*）表示
      * 可用于计算两个数值的乘积
      * 规则
        + 操作数都是数值

执行正常见的乘法操作

如果 ECMAScript 不能表示乘积，则返回 Infinity 或 -Infinity

* + - * + 任一操作数为 NaN

返回 NaN

* + - * + Infinity \* 0

返回 0

* + - * + Infinity \* （非 0 的有限数值）

根据第二个操作数的符号，
返回 Infinity 或 -Infinity

* + - * + Infinity \* Infinity

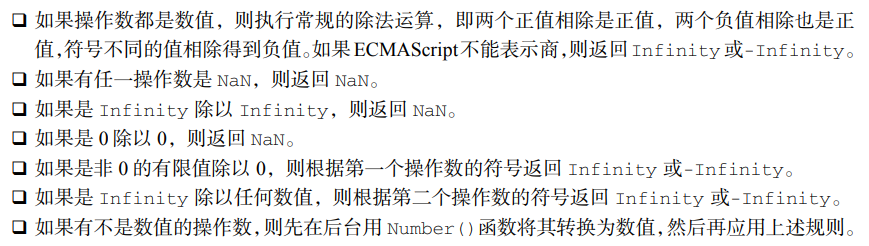
返回 Infinity

* + - * + 若操作数都不是数值

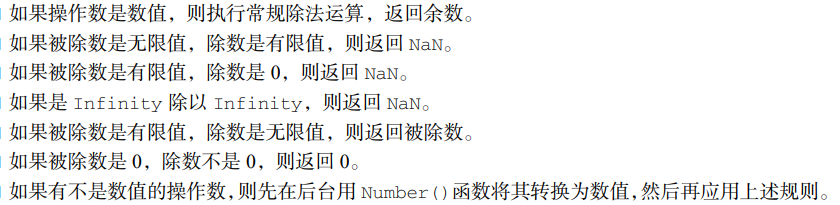
先在后台使用 Number() 进行类型转换

再应用上述规则

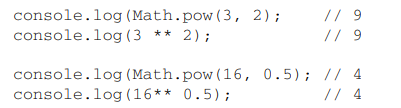
* + - 除法操作符
      * 操作符使用符号（/）表示
      * 规则



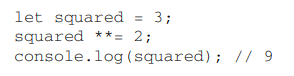
* + - 取模操作符
      * 操作符使用符号（%）表示
      * 规则



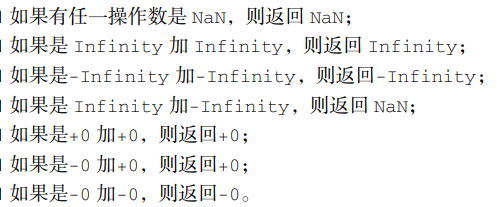
* + 指数操作符
    - 操作符使用符号（\*\*）表示
    - 与 Math.pow() 效果一样



* + - 指数赋值操作符（\*\*=）
      * 执行指数运算和结果的赋值操作



* + 加性操作符
    - 加法操作符
      * 两个操作数都是数值



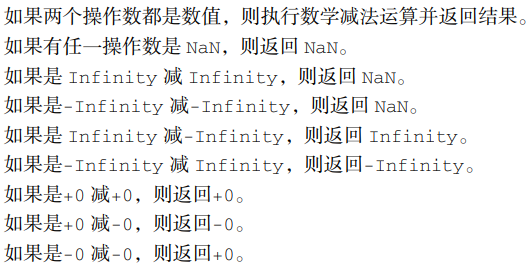
* + - * 操作数含有字符串
        + 两个操作数都是字符串

将第二个字符串拼接到第一个字符串后面

* + - * + 只有一个字符串

将另一个操作数转换为字符串，
再将两个字符串拼接在一起

* + - * 任一操作数是对象、数值或布尔值
        + 调用其 toString() 方法以获取字符串，
          然后再应用上述关于字符串的规则
      * 任一操作数是 undefined 或 null
        + 调用 String() 函数，
          分别获取 ”undefined“ 和 ”null“
    - 减法操作符
      * 两个操作数都是数值



* + - * 任一操作数是字符串、布尔值、null 或 undefined
        + 先在后台使用 Number() 将其转换为数值

根据上述规则执行数学运算

数值为 NaN，
则减法计算的结果是 NaN

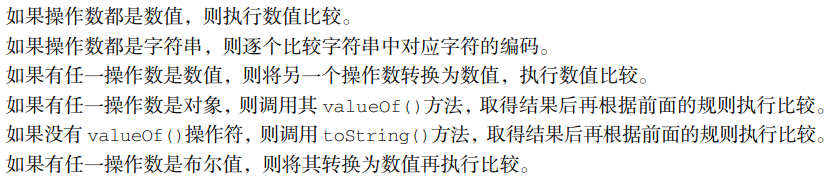
* + - * 任一操作数是对象
        + 调用其 valueOf() 方法，取得表示它的数值

数值为 NaN，
则减法计算的结果是 NaN

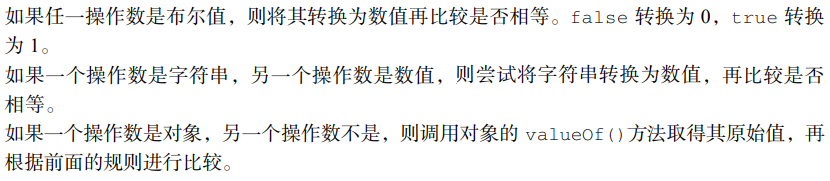
* + - * + 若对象没有 valueOf() 方法，则调用其 toString() 方法

再将得到的字符串转换为数值

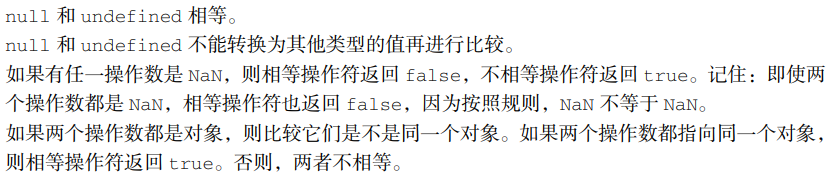
* + 关系操作符
    - 执行比较两个值的操作
      * 返回布尔值
    - 包括
      * 小于（<）
      * 大于（>）
      * 小于等于（<=）
      * 大于等于（>=）
    - 规则



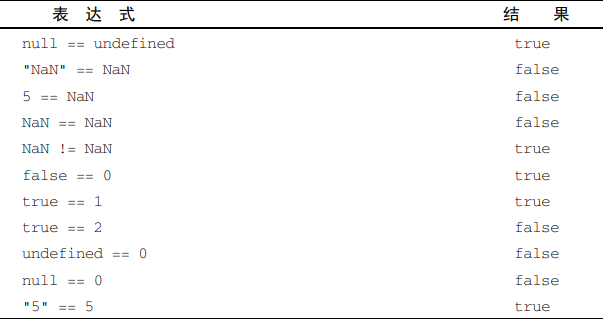
* + 相等操作符
    - 等于和不等于
      * 等于操作符（==）
      * 不等操作符（!=）
      * 会进行强制类型转换
      * 操作数类型转换规则



* + - * 比较规则



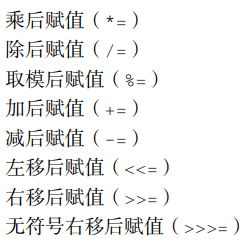
* + - * 特殊情况的比较



* + - 全等和不全等
      * 全等操作符（===）
      * 不全等操作符（!==）
      * 不进行强制类型转换
      * null === false = false
  + 条件操作符
    - 语法



* + 赋值操作符
    - 简单赋值操作符（=）
    - 符合赋值操作符，
      仅是简写语法，不会提升性能



* + 逗号操作符
    - 可以用于在一条语句中执行多个操作符
    - 常用于同时声明多个变量



* + - 也可用于辅助赋值



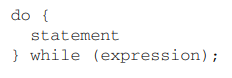
* + - * 在赋值时使用逗号操作符分隔值，最终会返回表达式中最后一个值

[3.6] 语句

* + ECMAScript 描述了一些语句（也称为流程控制语句）
  + if 语句
    - 语法



* + - 注意事项
      * 条件（condition）可以是任何表达式，
        求值结果并不一定是布尔值
        + ECMAScript 会自动调用 Boolean() 函数将表达式结果转换为布尔值
  + do-while 语句
    - 一种后测试循环语句，
      即循环体中的代码执行后才会对退出条件进行求值
    - 语法



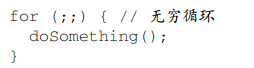
* + - 常用场景
      * 循环体内代码在退出前至少要执行一次
  + while 语句
    - 一种先测试循环语句，
      即先检测退出条件，再执行循环体内的代码
    - 语法



* + for 语句
    - 先测试语句
      * 具有进入循环之前的初始化代码，
        循环执行后要执行的表达式
    - 语法



* + - for 循环初始化代码
      * 可以不使用变量声明关键字
      * 最清晰的写法是使用 let 声明迭代器变量，
        可以将变量的作用域限定在循环中
    - 初始化、条件表达式和循环后表达式都不是必须的



* + for-in 语句
    - 一种严格的迭代语句，
      用于枚举对象中的非符号键属性
    - 语法



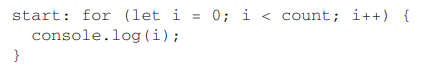
* + - ECMAScript 中对象的属性是无序的，
      因此 for-in 语句不能保证返回对象属性的顺序
      * 对象属性的返回顺序因游览器而异
    - 如果 for-in 循环要迭代的变量是 null 或 undefined，
      则不执行循环体
  + for-of 语句
    - 一种严格的迭代语句，
      用于遍历可迭代对象的元素
    - 语法



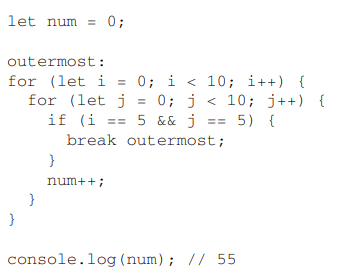
* + - for-of 循环会按照可迭代对象的 next() 方法产生值的顺序迭代元素
    - 若尝试迭代的变量不支持迭代，
      则 for-of 语句会抛出错误
  + 标签语句
    - 用于给语句加标签
    - 语法



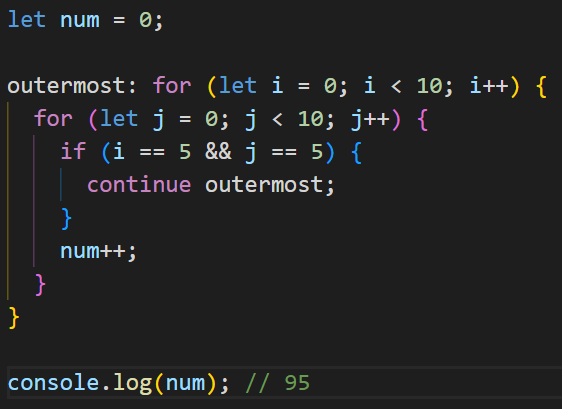
* + - 示例



* + - * start 即为一个标签，
        可在后面通过 break 或 continue 语句引用
    - 典型应用场景
      * 嵌套循环
  + break 和 continue 语句
    - break 和 continue 语句为执行循环代码提供更严格的控制手段
    - break 语句
      * 用于立即退出循环，
        强制执行循环后的下一条语句
      * 配合标签使用



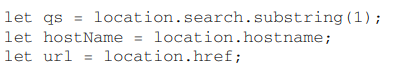
* + - * + 由于 outermost 指向最外圈循环，
          break outermost 将退出双层循环，而不仅是单层循环体
    - continue 语句
      * 用于立即退出循环，
        再次从循环顶部开始执行
      * 配合标签使用



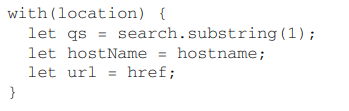
* + with 语句
    - 将代码作用域设置为特定的对象
    - 语法



* + - 主要场景
      * 针对一个对象反复操作
      * 示例

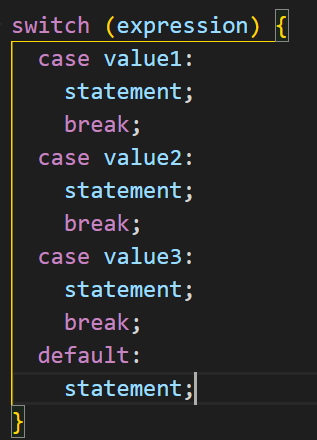






这里 with 语句用于连接 location 对象，
在该语句内部，每个变量首先会被认为是一个局部变量，
如果没有找到该局部变量，则会搜素 location 对象

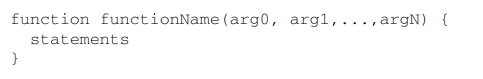
* + - 严格模式下不允许使用 with 语句，
      否则抛出错误
    - 警告 由于 with 语句影响性能且难于调试其中的代码，通常不推荐在产品代码中使用 with 语句
  + switch 语句
    - 与 if 语句紧密相关的一种流控制语句
    - 语法



* + - switch 语句可以用于所有数据类型
    - 条件的值不需要是常量，
      也可以是变量或表达式

[3.7] 函数

* + ECMAScript 中的函数使用 function 关键字声明，
    后跟一组参数，然后是函数体
  + 基本语法

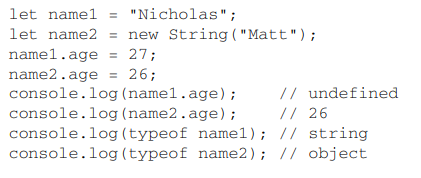


* + ECMAScript 函数不需要指定是否返回值
    - 任何函数可以在任何时候返回任何值
    - 不指定返回值的函数实际上会返回特殊值 undefined
  + 严格模式对函数的一些限制
    - 函数不能以 eval 或 arguments 作为名称
    - 函数的参数不能以 eval 或 arguments 命名
    - 两个命名参数不能拥有同一个名称

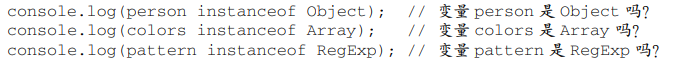
[第 4 章] 变量、作用域与内存

[4.1] 原始值与引用值

* + ECMAScript 变量可以包含两种类型的数据
    - 原始值（primitive value）
      * 最简单的数据
      * 保存原始值的变量为按值访问（by value ），
        操作存储在变量中的实际值
      * 很多语言中，字符串使用对象表示，
        因此被认为是引用类型，ECMAScript 打破了该惯例
    - 引用值（reference value）
      * 多个值构成的对象
      * 保存引用值的变量为按引用访问（by reference），
        实际操作对象的引用（reference）而非实际的对象本身
  + 动态属性
    - 对于引用值
      * 可以随时添加、修改和删除其属性和方法
    - 对于原始值
      * 不能用属性
      * 可以添加属性，且不会报错，但无意义
      * 原始类型的初始化
        + 可以只使用原始字面量形式
        + 使用 new 关键字，则 JS 会创建一个 Object 类型的实例，
          行为类似原始值



* + 传递参数
    - ECMAScript 中所有函数的参数都为按值传递
      * 函数外的值会被复制到函数内部的参数中，
        就像从一个变量复制到另一个变量中
    - ECMAScript 中函数的参数是局部变量
  + 确定类型
    - typeof 操作符
      * 适合判断一个变量是否为原始类型，
        null 会被判断为 ”object”
      * ECMA-262 规定，任何实现内部 [[Call]] 方法的对象都应该在 typeoif 检测时返回 “function”
    - instanceof 操作符
      * 适合判断引用值类型
      * 如果变量是给定引用类型的实例，
        由变量的原型链决定，
        则 instanceof 操作符返回 true



* + - * + 所有引用值都是 Object 的实例，
          通过 instanceof 操作符检测任何引用值和 Object 构造函数都会返回 true
        + instanceof 检测原始值，始终返回 false

[4.2] 执行上下文与作用域

* + 变量或函数的执行上下文决定其可以访问哪些数据，以及它们的行为
  + 每个执行上下文都有一个关联的变量对象（variable object）,
    该执行上下文中定义的所有变量和函数都保存在该变量对象中
    - 无法通过代码访问变量对象
    - 后台处理数据会使用变量对象
  + 全局上下文是最外层的上下文。
    根据 ECMAScript 实现的宿主环江，表示全局上下文的对象可能不一样
    - 浏览器中，window 对象就是全局上下文
      * 所有通过 var 定义的全局变量和函数都会称为 window 对象的属性和方法
      * 使用 let 和 const 的顶级声明不会定义在全局上下文中，但在作用域链解析上效果是一样的
  + 上下文在其所有代码都执行完毕后会被销毁，
    包括定义在其上面的所有变量和函数
    - 全局上下文在应用程序退出前才会被销毁，
      比如关闭网页或退出浏览器