**JS进阶课外总结**

# ch01.JavaScript发展历程及语言特点

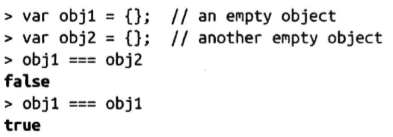
## 深入理解JavaScript第一部分

1、原始值和对象

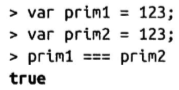
-原始值包括布尔值、数字、字符串、null和undefined。

-其他的值都是对象。

这两者之间最主要的区别在于他们的比较方式，每个对象都有唯一的标识且只（严格的）等于自己：

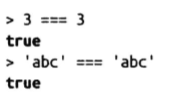


相反，所有原始值，只要编码相同，则被认为相等：

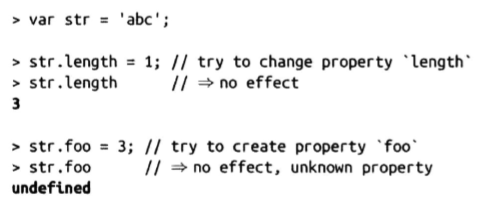


2、原始值特点

（1）按值进行比较



（2）其属性不能被改变、添加或移除，读取一个未知属性时总会返回undefined

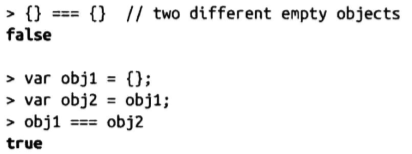


3、对象特点

所有非原始值都是对象。

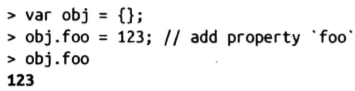
（1）按引用进行比较：

比较身份标识；每个值都有各自的身份标识



（2）默认可变

对象属性可以自由地改变、添加和移除

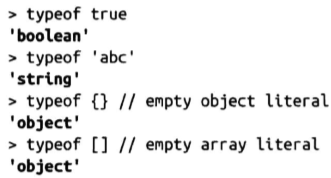


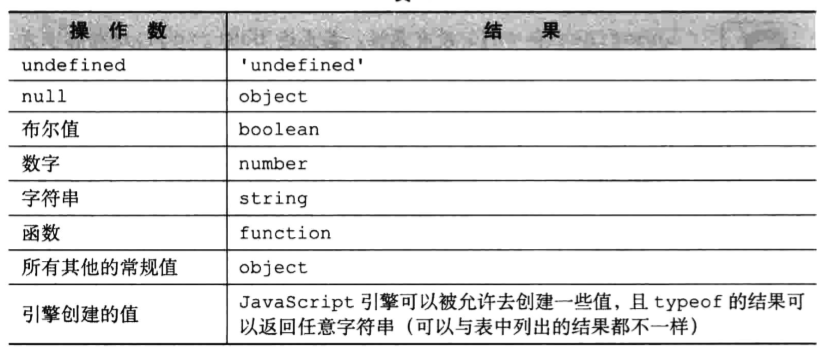
4、typeof和instanceof

typeof主要用于原始值，instanceof用于对象。

（1）typeof

typeof value：它的返回值是一个表示这个值“类型”的字符串：

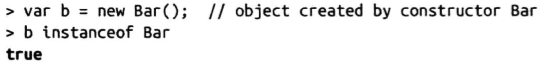
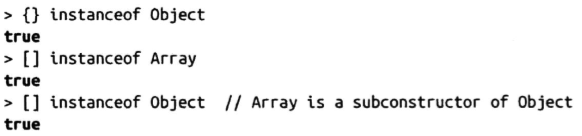




typeof null返回object是一个不能去修正的bug，因为这会破坏现有的代码，但这并不表示null是一个对象。

（2）instanceof

value instanceof Constr：如果value是一个通过Constr构造器创建的对象，则返回true：





5、等式运算符

-常规的，或“宽松的”相等（或不相等）：==，!=。

-严格的相等（或不相等）：===，!==。

6、被解释为false的值：

-undefined、null

-布尔值：false

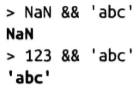
-数字：**-0**、NaN

-字符串：’’

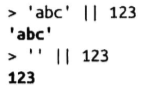
7、二元逻辑运算符（与假或真）

短路。**如果第一个运算数就足以确定结果的话，则永运不会第二个运算数做评估。**

（1）与（&&）：如果第一个运算数是假值，返回它。否则返回第二个运算数。



（2）或（||）：如果第一个运算数是真值，返回它。否则返回第二个运算数。



8、数字

JS中所有的数字都是浮点数：

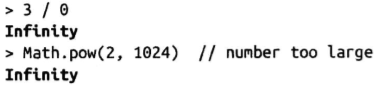


也包含一些特殊的数字：

-NaN（”not a number”） 一个错误的值：



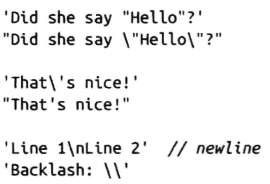
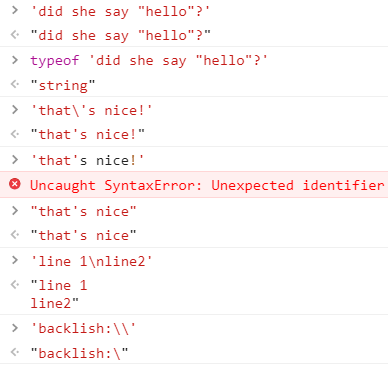
-Infinity 多数情况下也是一个错误的值：



Infinity比任何一个数都要大（NaN除外）。同样的，-Infinity比任何一个数都要小（NaN除外）。这使得这两个数字常用来作为默认值（比如，当你需要一个最大值和最小值的时候）。

9、 字符串-转义字符（\）

反斜杠（\）用于转义字符及产生一些控制字符：

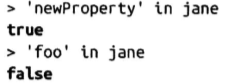
 

10、break：跳离循环

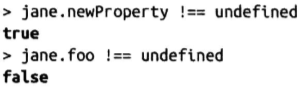
continue：开始一个新的迭代循环

11、对象

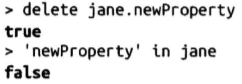
（1）使用in运算符检查属性是否存在：



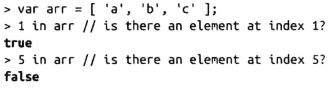
（2）如果读取一个不存在的属性，会得到undefined：



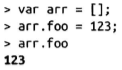
（3）使用delete运算符移除属性：



12、数组in操作符

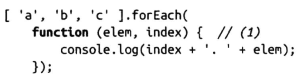


注意数组是对象，可以拥有对象属性。



13、遍历数组

（1）forEach迭代数组并且将当前的元素和元素的index扔到一个函数中：



输出结果：



(1)处的函数可以被随意地忽略参数。例如，它可以只包含一个elem参数。

（2）map通过一个函数映射到现有的数组的每个已经存在的元素创建一个新的数组：

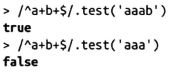
14、正则表达式

JavaScript内置的支持正则表达式，它们使用斜线分割：

/^abc$/

/[A-Za-z0-9]+/

（1）test()方法：匹配吗



（2）exec()方法：匹配以及捕获分组



返回的数组包含完整的匹配结果，它的**索引从0开始，第一组捕获的内容的索引是1**，以此类推。

（3）replace()方法：搜索和替换



replace的第一个参数必须是一个带着/g标志的正则表达式；否则将只替换第一次出现的内容。

# ch02.数据类型以及存储和转换

## 《你不知道的JavaScript（中卷）》第1/2/4章

1-1、JavaScript七种内置类型：

-空值（null）

-未定义（undefined）

-布尔值（boolean）

-数字（number）

-字符串（string）

-对象（object）

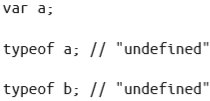
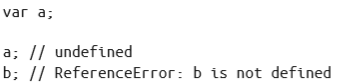
-符号（symbol，ES6中新增）

除对象之外，其他统称为“基本类型”。

1-2、JS中变量没有类型，只有值才有。变量可以随时持有任何类型的值。

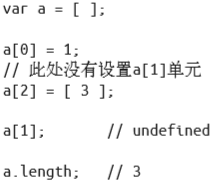
1-3、undefined：已在作用域中声明但还没有赋值的变量。

undeclared：还没有在作用域中声明过得变量。



2-1、稀疏数组

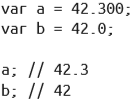
在创建“稀疏”数组（sparse array，即含有空白或空缺单元的数组）时要特别注意：



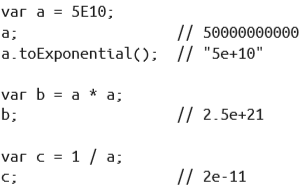
上方代码中a[1]的值为undefined，但这与将其显示赋值为undefined（a[1]=undefined）有所区别。

2-2、数字位数

默认情况下大部分数字都以十进制显示，小数部分最后面的0被省略：

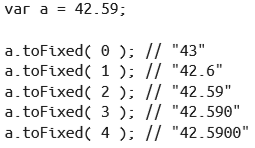


特别大和特别小的数字默认用指数格式显示，与toExponential()函数的输出结果相同：

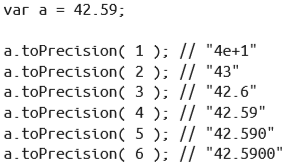


由于数字值可以使用 Number 对象进行封装，因此数字值可以调用 Number.prototype 中的方法。例如，

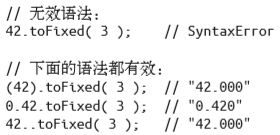
tofixed(..) 方法可指定小数部分的显示位数：

（输出结果为字符串形式）

toPrecision(..)方法用来指定有效数位的显示数位：



注意：





2-3、较小的数值

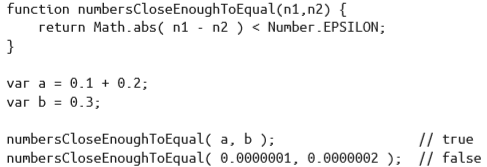
二进制浮点数最大问题：

简单来说，二进制浮点数中的 0.1 和 0.2 并不是十分精确，它们相加的结果并非刚好等于 0.3，而是一个比较接近的数字 0.30000000000000004，所以条件判断结果为 false。

机器精度值通常是：2^-52（2.220446049250313e-16）。从ES6开始，该值定义在Number.EPSILON中，可以直接用，也可以为ES6之前的版本写polyfill：



可以使用Number.EPSILON来比较两个数字是否相等（在指定误差范围内）：



2-4、整数安全范围

能被“安全”呈现的最大整数：2^53-1，即9007199254740991，在ES6中被定义为Number.MAX\_SAFE\_INTEGER；

最小整数：-9007199254740991，在ES6中被定义为Number.MIN\_SAFE\_INTEGER。

JS的数字类型无法精确呈现64位数值，所以必须将他们保存（转换）为字符串。

2-5、null和undefined

null指空值（曾赋过值，但目前没有指）

undefined指没有指（从未赋值）

null是一个特殊关键字，不是标识符，不能将其当作变量来赋值。然而undefined是一个标识符，可以被当作变量来使用和赋值。

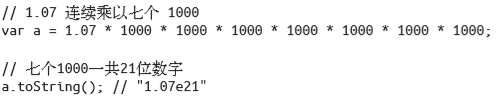
2-6、void运算符

表达式void \_没有返回值，因此返回结果是undefined。void并不改变表达式的结果，只是让表达式不返回值：



4-1、ToString：处理非字符串到字符串的强制类型转换。

数字的字符串化则遵循通用规则，不过极大和极小的数字使用指数形式：



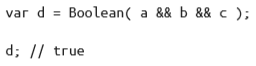
数组的默认toString()方法经过了重新定义，将所有单元字符串化以后再用“,”链接起来：



4-2、假值对象

假值对象并非封装了假值的对象：

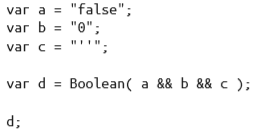


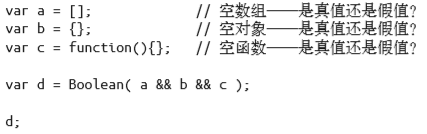


d为true说明a、b、c都为true。

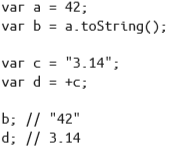
假值对象看起来和普通对象并无二致（都有属性，等等），但将它们强制类型转换为布尔值时结果为false。

4-3、真值（假值列表之外的值）

d为true

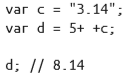
d为true

4-4、



上例中+c是+运算符的一元形式（即只有一个操作数）。+运算符显示地将c转换为数字，而非数字加法运算（也不是字符串拼接）。

在 JavaScript 开源社区中，一元运算 + 被普遍认为是显式强制类型转换。不过这样有时候也容易产生误会。例如：



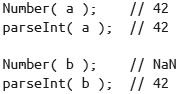
一元运算符 - 和 + 一样，并且它还会反转数字的符号位。由于 -- 会被当作递减运算符来处 理，所以我们不能使用 -- 来撤销反转，而应该像 - -"3.14" 这样，在中间加一个空格，才 能得到正确结果 3.14。

特例：

尽量不要把一元运算符 +（还有 -）和其他运算符放在一起使用。上面的代码可以运行， 但非常糟糕。此外 d = +c（还有 d =+ c）也容易和 d += c 搞混，两者天壤之别。

4-5、显示解析数字字符串

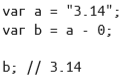




解析允许字符串中含有非数字字符，解析按从左到右的顺序，如果遇到非数字字符就停 止。而转换不允许出现非数字字符，否则会失败并返回 NaN。

解析和转换之间不是相互替代的关系。它们虽然类似，但各有各的用途。如果字符串右边 的非数字字符不影响结果，就可以使用解析。而转换要求字符串中所有的字符都是数字， 像 "42px" 这样的字符串就不行。

4-6、字符串强制类型转换为数字

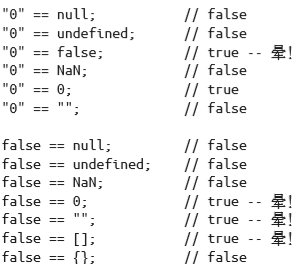


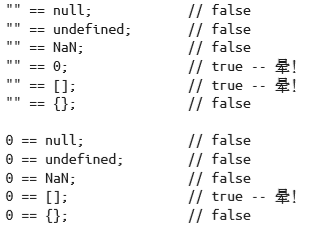
- 是数字减法运算符，因此 a - 0 会将 a 强制类型转换为数字。也可以使用 a \* 1 和 a / 1，因为这两个运算符也只适用于数字，只不过这样的用法不太常见。

4-7、NaN不等于NaN。

+0等于-0。

4-8、假值相等比较



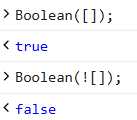


以上 24 种情况中有 17 种比较好理解。比如我们都知道 "" 和 NaN 不相等，"0" 和 0 相等。

然而有 7 种我们注释了“晕！”，因为它们属于假阳（false positive）的情况，里面坑很多。 "" 和 0 明显是两个不同的值，它们之间的强制类型转换很容易搞错。请注意这里不存在假 阴（false negative）的情况。

极端情况：

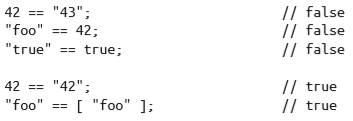


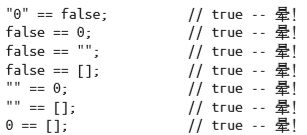


根据 ToBoolean 规则，它会进行布尔值的显式强制类型转换（同时反转奇偶校验位）。所以 [] == ![] 变成了 [] == false。又因为false == []，所以[] == ![]。



""、"\n"（或者 " " 等其他空格组合）等空字符串被 ToNumber 强制类型转换 为 0。







# ch03.包装对象和数据类型转换

## 《JavaScript权威指南》第三章

1、除了十进制的整型直接量，JavaScript同样能识别十六进制（以16为基数）值。所谓十六进制的直接量是以“0x”或“0X”为前缀，其后跟随十六进制数串的直接量。



2、Math



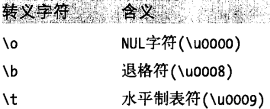
3、在BCMAScript 3中，字符串直接量必须写在一行中，而在ECMAScript 5中，字符串直

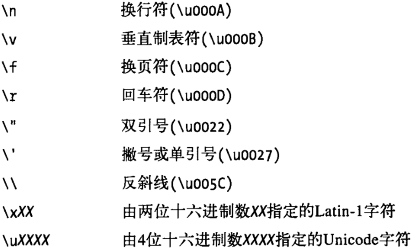
量可以拆分成数行，每行必须以反斜线(\) 结束，反斜线和行结束符都不算是字符串直接量的内容。如果希望在字符串直接量中另起一行，可以使用转义字符\n。



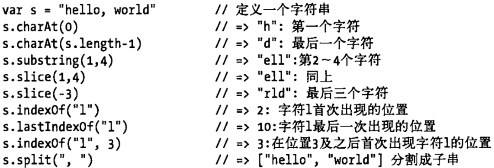


4、转义字符





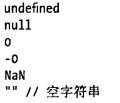
5、字符串属性





s.length

6、假值



false和上面6个可以转换成false的值有时称作“假值”，其他值称作真值。

7、全局对象

全局对象的属性时全局定义的符号，JS程序可以直接使用。当JS解释器启动时（或者任何Web浏览器加载新页面的时候），它将创建一个新的全局对象，并给它一组定义的初始属性：

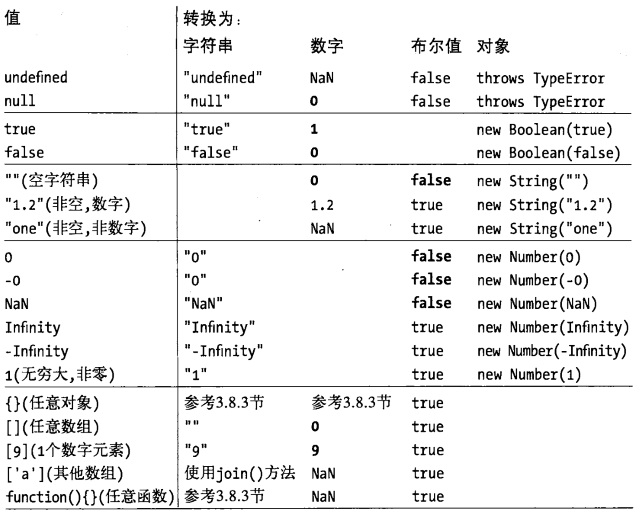
全局属性，比如undefined、Infinity和NaN

全局函数，比如isNaN()、parseInt()和eval()

构造函数，比如Data()、RegExp()、String()、Object()和Array()

全局对象，比如Math和JSON

8、类型转换



## 《深入理解JavaScript》第八章

1、undefined和null出现场景

（1）undefined出现场景

未初始化的变量



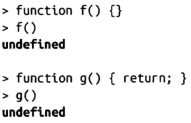
缺失的参数



访问一个不存在的属性会返回undefined



如果函数中没有显示地返回任何值，函数会隐式返回undefined



（2）null出现的场景

null是原型链最顶端的元素



当字符串中没有匹配到正则表达式的结果是，RegExp.prototype.exec()会返回null



2、算法：ToPrimitive()——将值转换为原始值

要将任意值转换为数字或者字符串，首先会被转换为任意的原始值，然后再转换为

最终的结果。



可选参数PreferredType表明转换后的类型:它可以是Number或String,具体取决于ToPrimitive的结果是希望转换成数字还是字符串。

如果PreferredType是Number,会执行以下步骤。

(1)如果input是原始值，返回这个值(没有其他需要做的)。

(2)否则，如果input是对象，调用input. value0f()。 如果结果是原始值，

返回结果。

(3)否则，调用input. toString()。如果结果是原始值，返回结果。

(4)否则，抛出一个TypeError (说明将输入转换为原始值出错了)。

如果PreferredType 是字符串，第二步和第三步会进行交换。PreferredType 也可以

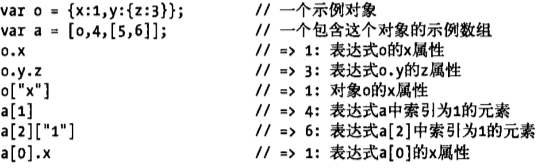
被省略，这种情况下，日期会被认为是String而其他值会被认为是Number。因此，

+运算符和==运算符可以操作ToPrimitive()。

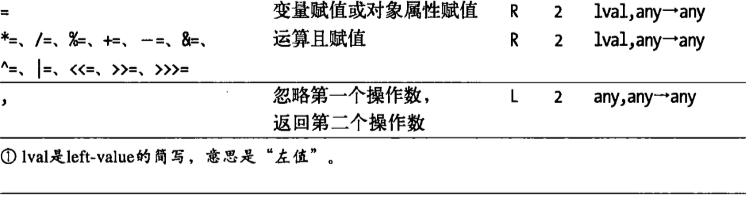
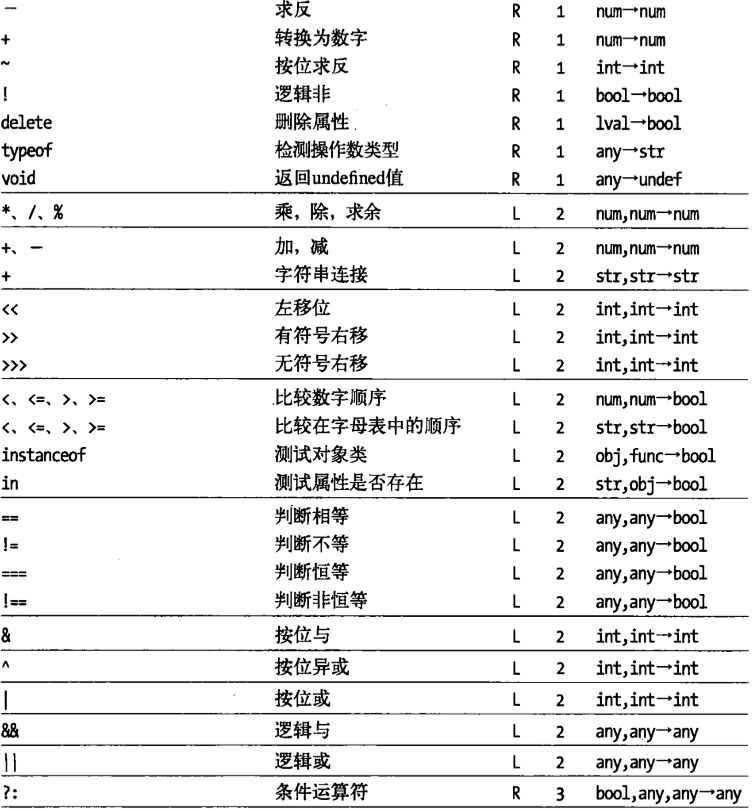
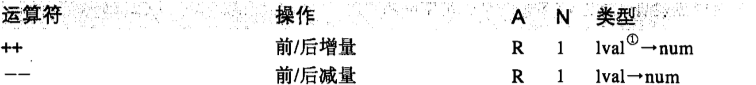
# ch04.标识符与表达式

## 《JavaScript权威指南》第4、5章

4-1、属性访问



4-2、JS运算符



4-3、一元操作符、赋值和三元条件运算符都具有从左至右的结核性：

等同于

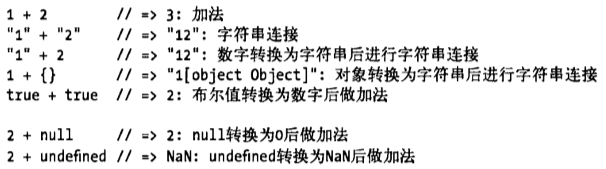
4-4、%运算符

结果的符号和第一个操作数（被除数）的符号保持一致。例如，5%2=1，-5%2=-1。

求余运算符的操作数通常都是整数，但也是用于浮点数。例如6.5%2.1=0.2。

4-5、+运算符

如果其中一个操作数是对象，则对象会遵循对象到原始值的转换规则转换为原始类值：日期对象通过toString()方法执行转换，其他对象则通过valueOf()方法执行转换（如果valueOf()方法返回一个原始值的话）。由于多数对象都不具备可用的valueOf()方法，因此它们会通过toString()方法来执行转换。



4-6、位运算符

（1）按位与(&)

位运算符“&”对它的整型操作数**逐位**执行布尔与(AND)操作。只有两个操作数中相对应的位都是1，结果中的这一位才是1。例如，0x1234 & 0x00FF = 0x0034。

（2）按位或(|)

位运算符“|”对它的整型操作数**逐位**执行布尔或(OR)操作。如果其中一个操作数相应的位为1，或者两个操作数相应位都是1,那么结果中的这一位就为1。例如: 0x1234| 0X00FF = 0x12FF。

（3）按位异或(^)

位运算符“I”对它的整型操作数**逐位**执行布尔异或(XOR)操作。异或是指第一个操作数t rue或第二个操作数为true,但两者不能同时为true。如果两个操作数中只有一个相应位为1 (不能同时为1)，那么结果中的这一位就是1。例如，0xFF00 ^ 0xFOF0 = 0x0FF0。

（4）按位非(~)

运算符“~”是一元运算符，位于一个整型参数之前，它将操作数的所有位取反。根据JavaScript中带符号的整数的表示方法，对一个值使用“~”运算符相当于改变它的符号并减1。例如，~0x0F = FFFFFFF0或-16。

（5）左移(<<)

将第一个操作数的所有二进制位进行左移操作，移动的位数由第二个操作数指定，移动的位数是0~ 31之间的一个整数。例如，在表达式a<<1中， a的第一位变成了第二位，a的第二位变成了它的第三位，以此类推。新的第一位 用0来补充，舍弃第32位。将一个值左移1位相当于它乘以2，左移两位相当于乘以4，以此类推。例如，7<<2=28。

（6）带符号右移(>>)

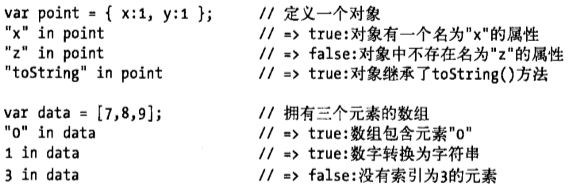
运算符“>>” 将第一个操作数的所有位进行右移操作，移动的位数由第二个操作数指定，移动的位数是0~31之间的一个整数。右边溢出的位将忽略。填补在左边的位由原操作数的符号决定，以便保持结果的符号与原操作数一致。如果第一个操作数是正数，移位后用0填补最高位；如果第一个操作数是负的，移位后就用1填补高位。将一个值右移1位，相当于用它除以2 (忽略余数)，右移两位，相当于它除以4，以此类推，例如，7>>1=3， -7>>1=-4。

（7）无符号右移(>>>)

运算符“>>>” 和运算符“>>” 一样，只是左边的高位总是填补0，与原来的操作数符号无关，例如，-1>>4=-1, 但是-1>>>4=0FFFFFF。

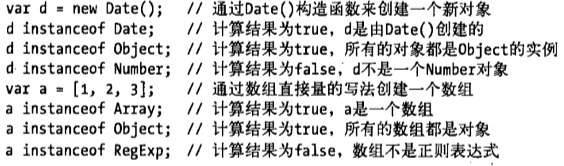
4-7、in运算符

in运算符希望它的左操作数是一个字符串或可以转换为字符串，希望它的右操作数是一个对象。如果右侧的对象拥有一个名为左操作数值的属性名，那么表达式返回true，例如：



4-8、instanceof运算符

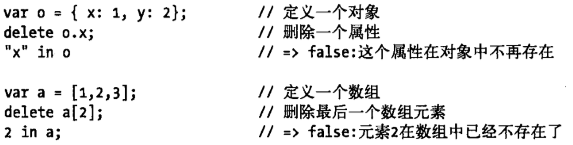
instanceof运算符希望左操作数是一个对象，右操作数标识对象的类。如果左侧的对象是右侧类的实例，则表达式返回true;否则返回false。JavaScript中对象的类是通过初始化它们的构造函数来定义的。这样的话，instanceof的右操作数应当是一个函数。比如：



需要注意的是，所有的对象都是Object的实例。当通过instanceof判断一个对象是否是一个类的实例的时候，这个判断也会包含对“父类”(superclass) 的检测。如果instanceof的左操作数不是对象的话，instanceof返回false。如果右操作数不是函数，则抛出一个类型错误异常。

4-9、delete运算符

delete是一元操作符，它用来删除对象属性或者数组元素。就像赋值、递增、递减运算符一样，delete也是具有副作用的，它是用来做删除操作的，不是用来返回一个值的，例如：

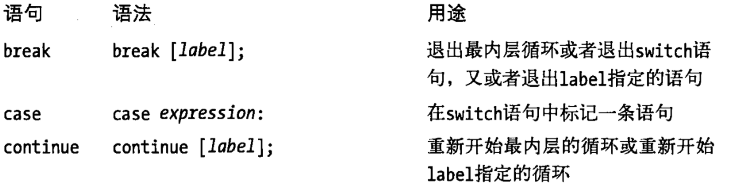




需要注意的是，删除属性或者删除数组元素不仅仅是设置了一个undefined的值。当删除一个属性时，这个属性将不再存在。读取一个不存在的属性将返回undefined,但是可以通过in运算符来检测这个属性是否在对象中存在。

delete希望他的操作数是一个左值，如果它不是左值，那么delete将不进行任何操作同时返回true。否则，delete将试图删除这个指定的左值。如果删除成功，delete将返回true。然而并不是所有的属性都可删除，一些内置核心和客户端属性是不能删除的，用户通过var语句声明的变量不能删除。同样，通过function语句定义的函数和函数参数也不能删除。

5-1、JS语句语法



## 《深入理解JavaScript》第9、13章

9-1、宽松（普通）相等（==，!=）

如果两个运算数的类型相同(六种规范类型：Undefined，Null，Boolean，Number，String和Object其中之一)， 则使用严格相等比较它们。

否则，如果运算数是如下类型：

1. undefined和null，则它们被认为是宽松相等的。



2.一个字符串和一个数字，则将字符串转换为一个数字，使用严格相等比较两个运算数。

3.一个布尔值和一个非布尔值，则将布尔值转换为一个数字，然后(再次)进行宽松比较。

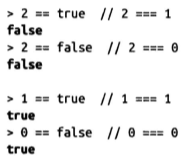
4.一个对象和一个数字或者一个字符串，则尝试转换此对象为一个原始值，然后(再次)进

行宽松比较。

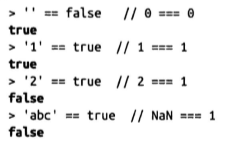
不符合上述提到的情况，宽松比较的结果是false。

9-2、宽松相等与布尔转换不同

相等和转换为布尔值的工作原理是不同的。若数字大于1，则转换为布尔值true (如if语句)。但这些数字不是宽松地等于true：

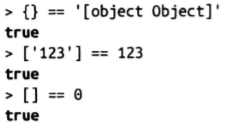


类似的，虽然空字符串等于false，但不是所有空字符串都等于true：

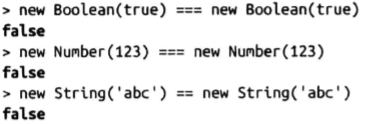


9-3、宽松相等中的对象

如果比较对象和非对象，它们会被转换为原始值：



然而，只有当两个对象是同一个对象时才会相等。这意味着无法真正地比较两个包装对象：



9-4、逗号运算符

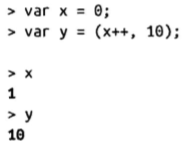


逗号运算符会执行两边的运算数并返回right 部分的结果。简单来讲，它的效果等同于块之间的分号。

例，第二个运算数会作为运算的结果返回：



例，两个运算数的执行效果：



9-5、对象运算符

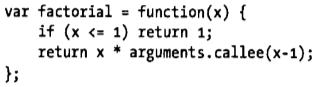
new，delete，in

# ch05.函数定义和调用形式

## 《JavaScript权威指南》第8章

1、callee和caller属性

除了数组元素，实参对象还定义了callee和caller属性。在ECMAScript 5严格模式中，对这两个属性的读写操作都会产生一个类型错误。而在非严格模式下，ECMAScript标准规范规定callee属性指代当前正在执行的函数。caller是非标准的，但大多数浏览器都实现了这个属性，它指代调用当前正在执行的函数的函数。通过caller 属性可以访问调用栈。callee属性在某些时候会非常有用，比如在匿名函数中通过callee来递归地调用自身。



2、Function()构造函数

不管是通过函数定义语句还是函数直接量表达式，函数的定义都要使用function关键字。但函数还可以通过Function()构造函数来定义，比如：



这一行代码创建一个新的函数，这个函数和通过下面代码定义的函数几乎等价：



Function()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，每两条语句之间用分号分隔。传入构造函数的其他所有的实参字符串是指定函数的形参名字的字符串。如果定义的函数不包含任何参数，只须给构造函数简单地传入一个字符串——函数体——即可。

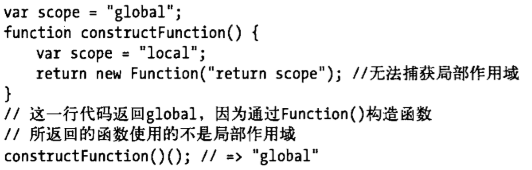
注意，Function()构造函数并不需要通过传人实参以指定函数名。就像函数直接量一样，Function()构造函数创建一个匿名函数。

关于Function()构造函数有几点需要特别注意:

-Function()构造函数允许JavaScript在运行时动态地创建并编译函数。

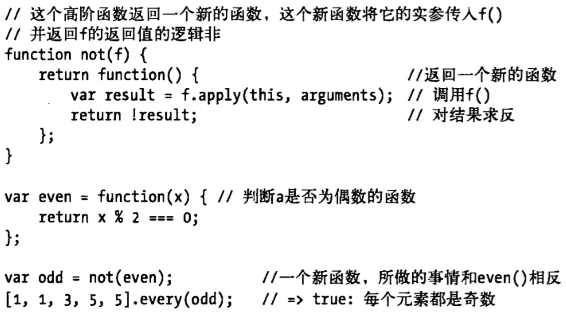
-每次调用Function()构造函数都会解析函数体，并创建新的函数对象。如果是在一个循环或者多次调用的函数中执行这个构造函数，执行效率会受影响。相比之下，循环中的嵌套函数和函数定义表达式则不会每次执行时都重新编译。

-最后一点，也是关于Function()构造函数非常重要的一点，就是它所创建的函数并**不是使用词法作用域**，相反，函数体代码的编译总是会在顶层函数执行：



3、高阶函数

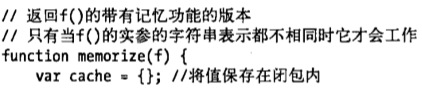
所谓高阶函数(higher-order function)就是操作函数的函数，它接收一个或多个函数作为参数，并返回一个新函数：



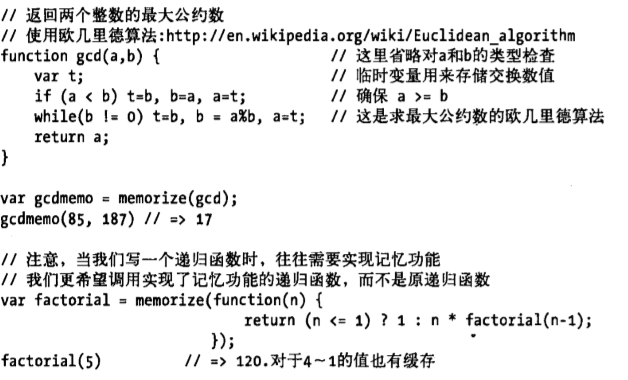
4、记忆

在一个阶乘函数，它可以将上次的计算结果缓存起来。在函数式编程当中，这种缓存技巧叫做“记忆”(memorization) 。下面的代码展示了一个高阶函数，memorize()接收一个函数作为实参，并返回带有记忆能力的函数。

需要注意的是，记忆只是一种编程技巧，本质上是牺牲算法的空间复杂度以换取更优的时间复杂度，在客户端JavaScript中代码的执行时间复杂度往往成为瓶颈，因此在大多数场景下，这种牺牲空间换取时间的做法以提升程序执行效率的做法是非常可取的。



memorize()函数创建一个新的对象，这个对象被当做缓存(的宿主)并赋值给-一个局部变量，因此对于返回的函数来说它是私有的(在闭包中)。所返回的函数将它的实参数组转换成字符串，并将字符串用做缓存对象的属性名。如果在缓存中存在这个值，则直接返回它。否则，就调用既定的函数对实参进行计算，将计算结果缓存起来并返回，下面的代码展示了如何使用memorize():

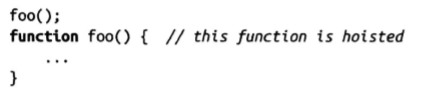


## 《深入理解JavaScript》第15章

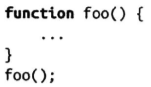
1、函数提升

函数提升表示“将函数的声明放到作用域的开始”。函数声明是做了完全提升的，而变量声明则是部分提升。

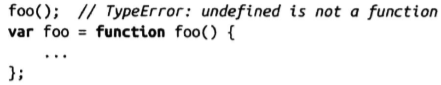
由于函数声明是完全提升的。这允许在函数声明前进行函数的调用：



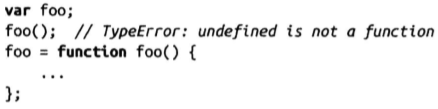
上面的代码是可以正常工作的，这是由于JavaScript引擎会将foo函数的声明放到其作用域最前面。因此代码实际上是这样执行的：



使用var的定义也是会进行代码提升的，但是只对于声明有效，对于赋值过程是无效的。因此，使用var来定义会导致下面的错误：



我们可以看到只有变量的定义被提升了。引擎实际会像这样解析上面的代码:



2、函数的名称

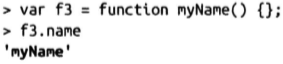
大多数的JavaScript引擎对于函数对象都会提供一个非标准的name属性。函数的声明会创建这样的一个属性：



而匿名函数表达式的name则是一个空字符串：



具名函数表达式也有一个name:



函数的名称对于debug是非常有用的。正因如此，有一些开发者往往会给函数表达式加上名字。

3、函数声明相比函数表达式有两个优势:

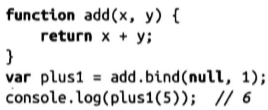
-函数声明会做代码提升，可以在源码中先于函数的定义来调用函数。

-它们具有名字。不过，JavaScript 引擎也在对匿名函数表达式的名字引用做优化。

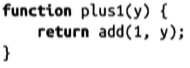
4、func.bind(thisValue,arg1,...,argN)

这个方法会执行部分的函数功能,它会创建一个新的函数，这个函数会调用func,并会将thisValue指定为this,同时应用以下参数: arg1 直到argN,紧随其后的是新函数的实际参数。在下面的非面向对象的调用中，我们将thisValue设定为了null。

下面，使用bind ()来创建一个新的plus1()函数，它类似add()函数，但是由于指定x始终为1,只需要传入参数y:



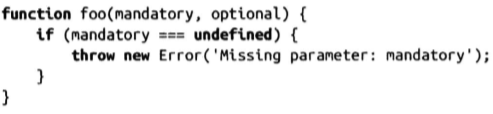
事实上，我们用类似以下代码创建了一个新的函数:



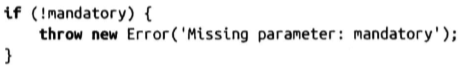
5、判断参数是否缺失

要判断一个参数是否缺失，有三种方式。

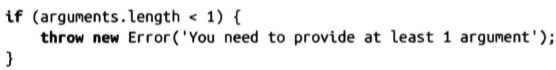
第一种，可以检测它的值是否为undefined：



第二种方式，将参数转化为布尔值。undefined会被认为false。不过这种方式有一个漏洞:还有一些其他的值也会被认为是false，因此这种方式不能区分例如0和一个缺失参数的情况:



最后一种方式，用arguments.length来检测并强制指定参数数量的最小值:



最后一种方式相比前两种有--些区别:

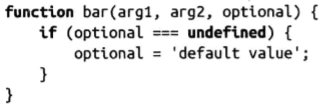
前两种方式无法区分foo()和foo (undefined)。它们对于两种情况会抛一样的错误。

第三种方式会对foo()抛错,而对于foo(undefined)会正常使用undefined传入函数并调用。

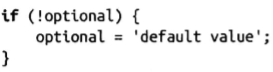
6、可选参数

如果参数是可选的，这意味着当没有指定这个参数，它将会使用默认值。类似于强制性参数，我们有四种处理可选参数的选择。

第一种，检测undefined:



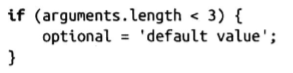
第二种，将可选参数转换为布尔值:



第三种，使用或运算符，如果左侧操作数不是false值，则返回左侧的操作数，否则将返回右侧的操作数:



第四种，使用arguments.length来检测函数支持的最小数量的参数。



同样的，最后一-种方式有别于其他的方式:

前三种方式无法区分bar(1,2)和bar(1,2,undefined)。这两种情况下，optional的值都是默认值。

第四种方式对于bar(1,2)会使用默认值，而对于bar(1,2,undefined)会保留undefined的传入。

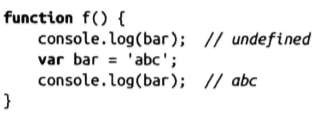
# ch06.函数定义和调用过程

## 《JavaScript权威指南》第8章

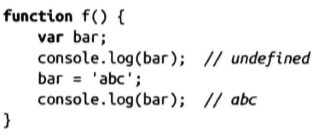
## 《深入理解JavaScript》第15、16章

16-1、变量声明提前

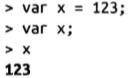
JavaScript会提前所有的变量声明，它会把所有的声明移到直接作用域的最前面。很明显当我们试图在变量声明前来访问它会发生什么:



可以看到变量bar在函数f()的第一行已经存在了，只是还没有值：因此，被提前的是变量的声明，而非变量的赋值。实际上，JavaScript 中是这样执行f()的:



如果定义一个已经被定义过的变量，那么什么也不会发生(包括变量的值，也不会变化)：



函数的声明也会被提前，但是与变量截然不同，函数声明会被整体提前，而不仅仅是变量的创建的那部分。

# ch07.闭包和this

## 你不知道的JavaScript（上卷）第一部分1-5章

1-1、var a = 2;

编译器会进行如下处理:

（1）遇到 var a，编译器会询问作用域是否已经有一个该名称的变量存在于同一个作用域的集合中。如果是，编译器会忽略该声明，继续进行编译；否则它会要求作用域在当前作用域的集合中声明一个新的变量，并命名为 a。

（2）接下来编译器会为引擎生成运行时所需的代码，这些代码被用来处理 a = 2 这个赋值操作。引擎运行时会首先询问作用域，在当前的作用域集合中是否存在一个叫作 a 的变量。如果是，引擎就会使用这个变量；如果否，引擎会继续查找该变量。如果引擎最终找到了 a 变量，就会将 2 赋值给它。否则引擎就会举手示意并抛出一个异常！



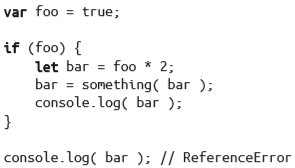
总结：变量的赋值操作会执行两个动作，首先编译器会在当前作用域中声明一个变量（如果之前没有声明过），然后在运行时引擎会在作用域中查找该变量，如果能够找到就会对它赋值。

2-1、JavaScript 中有两个机制可以“欺骗”词法作用域：eval(..) 和 with。前者可以对一段包 含一个或多个声明的“代码”字符串进行演算，并借此来修改已经存在的词法作用域（在 运行时）。后者本质上是通过将一个对象的引用当作作用域来处理，将对象的属性当作作 用域中的标识符来处理，从而创建了一个新的词法作用域（同样是在运行时）。

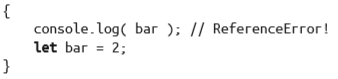
3-1、let

ES6引入了新的 let 关键字，提供了除 var 以外的另一种变量声明方式。

let 关键字可以将变量绑定到所在的任意作用域中（通常是 { .. } 内部）。换句话说，let 为其声明的变量隐式地限制在了所在的块作用域。

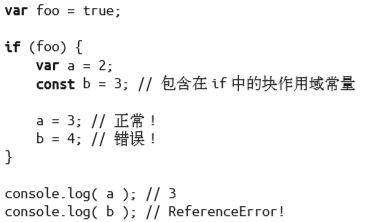


let 进行的声明不会在块作用域中进行提升。声明的代码被运行之前，声明并不 “存在”。

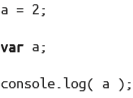
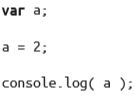


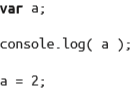
3-2、const

除了 let 以外，ES6 还引入了 const，同样可以用来创建块作用域变量，但其值是固定的 （常量）。之后任何试图修改值的操作都会引起错误。

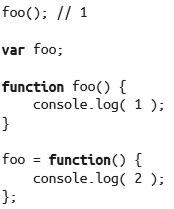
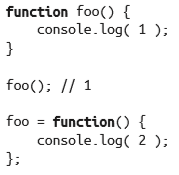


4-1、

===》

===》

4-2、

===》

注意，var foo 尽管出现在 function foo()... 的声明之前，但它是重复的声明（因此被忽 略了），因为函数声明会被提升到普通变量之前。

尽管重复的 var 声明会被忽略掉，但出现在后面的函数声明还是可以覆盖前面的。

4-3、我们习惯将 var a = 2; 看作一个声明，而实际上 JavaScript 引擎并不这么认为。它将 var a 和 a = 2 当作两个单独的声明，第一个是编译阶段的任务，而第二个则是执行阶段的任务。

这意味着无论作用域中的声明出现在什么地方，都将在代码本身被执行前首先进行处理。 可以将这个过程形象地想象成所有的声明（变量和函数）都会被“移动”到各自作用域的 最顶端，这个过程被称为提升。

声明本身会被提升，而包括函数表达式的赋值在内的赋值操作并不会提升。

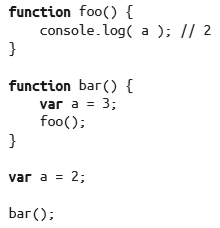
5-1、当函数可以记住并访问所在的词法作用域，即使函数是在当前词法作用域之外执行，这时产生了闭包。

5-2、模块有两个主要特征：

（1）为创建内部作用域而调用了一个包装函数；

（2）包装函数的返回值必须至少包括一个对内部函数的引用，这样就会创建涵盖整个包装函数内部作用域的闭包。

A-1、



词法作用域让 foo() 中的 a 通过 RHS 引用到了全局作用域中的 a，因此会输出 2。