

第 3 章

基本概念

本章内容

- 语法
- 数据类型
- 流控制语句
- 理解函数

任何语言的核心都必然会描述这门语言最基本的工作原理。而描述的内容通常都要涉及这门语言的语法、操作符、数据类型、内置功能等用于构建复杂解决方案的基本概念。如前所述，ECMA-262 通过叫做 ECMAScript 的“伪语言”为我们描述了 JavaScript 的所有这些基本概念。

目前，ECMA-262 第 3 版中定义的 ECMAScript 是各浏览器实现最多的一个版本。ECMA-262 第 5 版是浏览器接下来实现的版本，但截止到 2011 年底，还没有浏览器完全实现了这个版本。为此，本章将主要按照第 3 版定义的 ECMAScript 介绍这门语言的基本概念，并就第 5 版的变化给出说明。

3.1 语法

ECMAScript 的语法大量借鉴了 C 及其他类 C 语言（如 Java 和 Perl）的语法。因此，熟悉这些语言的开发人员在接受 ECMAScript 更加宽松的语法时，一定会有一种轻松自在的感觉。

3.1.1 区分大小写

要理解的第一个概念就是 ECMAScript 中的一切（变量、函数名和操作符）都区分大小写。这也就意味着，变量名 `test` 和变量名 `Test` 分别表示两个不同的变量，而函数名不能使用 `typeof`，因为它是一个关键字（3.2 节介绍关键字），但 `typeOf` 则完全可以是一个有效的函数名。

3.1.2 标识符

所谓标识符，就是指变量、函数、属性的名字，或者函数的参数。标识符可以是按照下列格式规则组合起来的一或多个字符：

- 第一个字符必须是一个字母、下划线（`_`）或一个美元符号（`$`）；
- 其他字符可以是字母、下划线、美元符号或数字。

标识符中的字母也可以包含扩展的 ASCII 或 Unicode 字母字符（如 `À` 和 `Æ`），但我们不推荐这样做。

按照惯例，ECMAScript 标识符采用驼峰大小写格式，也就是第一个字母小写，剩下的每个有意义的单词的首字母大写，例如：

```
firstSecond  
myCar  
doSomethingImportant
```

虽然没有谁强制要求必须采用这种格式，但为了与 ECMAScript 内置的函数和对象命名格式保持一致，可以将其当作一种最佳实践。



不能把关键字、保留字、true、false 和 null 用作标识符。3.2 节将介绍更多相关内容。

3.1.3 注释

ECMAScript 使用 C 风格的注释，包括单行注释和块级注释。单行注释以两个斜杠开头，如下所示：

```
// 单行注释
```

块级注释以一个斜杠和一个星号（/*）开头，以一个星号和一个斜杠（*/）结尾，如下所示：

```
/*  
 * 这是一个多行  
 * （块级）注释  
*/
```

虽然上面注释中的第二和第三行都以一个星号开头，但这不是必需的。之所以添加那两个星号，纯粹是为了提高注释的可读性（这种格式在企业级应用程序中极其常见）。

3.1.4 严格模式

ECMAScript 5 引入了严格模式（strict mode）的概念。严格模式是为 JavaScript 定义了一种不同的解析与执行模型。在严格模式下，ECMAScript 3 中的一些不确定的行为将得到处理，而且对某些不安全的操作也会抛出错误。要在整个脚本中启用严格模式，可以在顶部添加如下代码：

```
"use strict";
```

这行代码看起来像是字符串，而且也没有赋值给任何变量，但其实它是一个编译指示（pragma），用于告诉支持的 JavaScript 引擎切换到严格模式。这是为不破坏 ECMAScript 3 语法而特意选定的语法。在函数内部的上方包含这条编译指示，也可以指定函数在严格模式下执行：

```
function doSomething(){  
    "use strict";  
    // 函数体  
}
```

严格模式下，JavaScript 的执行结果会有很大不同，因此本书将会随时指出严格模式下的区别。支持严格模式的浏览器包括 IE10+、Firefox 4+、Safari 5.1+、Opera 12+ 和 Chrome。

3.1.5 语句

ECMAScript 中的语句以一个分号结尾；如果省略分号，则由解析器确定语句的结尾，如下例所示：

```
var sum = a + b           // 即使没有分号也是有效的语句——不推荐
var diff = a - b;         // 有效的语句——推荐
```

虽然语句结尾的分号不是必需的,但我们建议任何时候都不要省略它。因为加上这个分号可以避免很多错误(例如不完整的输入),开发人员也可以放心地通过删除多余的空格来压缩 ECMAScript 代码(代码行结尾处没有分号会导致压缩错误)。另外,加上分号也会在某些情况下增进代码的性能,因为这样解析器就不必再花时间推测应该在哪里插入分号了。

可以使用 C 风格的语法把多条语句组合到一个代码块中,即代码块以左花括号 ({) 开头,以右花括号 (}) 结尾:

```
if (test){
    test = false;
    alert(test);
}
```

虽然条件控制语句(如 if 语句)只在执行多条语句的情况下才要求使用代码块,但最佳实践是始终在控制语句中使用代码块——即使代码块中只有一条语句,例如:

```
if (test)
    alert(test);           // 有效但容易出错,不要使用

if (test){                 // 推荐使用
    alert(test);
}
```

在控制语句中使用代码块可以让编码意图更加清晰,而且也能降低修改代码时出错的几率。

3.2 关键字和保留字

ECMA-262 描述了一组具有特定用途的关键字,这些关键字可用于表示控制语句的开始或结束,或者用于执行特定操作等。按照规则,关键字也是语言保留的,不能用作标识符。以下就是 ECMAScript 的全部关键字(带*号上标的是第 5 版新增的关键字):

break	do	instanceof	typeof
case	else	new	var
catch	finally	return	void
continue	for	switch	while
debugger*	function	this	with
default	if	throw	
delete	in	try	

ECMA-262 还描述了另外一组不能用作标识符的保留字。尽管保留字在这门语言中还没有任何特定的用途,但它们有可能在将来被用作关键字。以下是 ECMA-262 第 3 版定义的全部保留字:

abstract	enum	int	short
boolean	export	interface	static
byte	extends	long	super
char	final	native	synchronized
class	float	package	throws
const	goto	private	transient
debugger	implements	protected	volatile
double	import	public	

第5版把在非严格模式下运行时的保留字缩减为下列这些：

class	enum	extends	super
const	export	import	

在严格模式下，第5版还对以下保留字施加了限制：

implements	package	public
interface	private	static
let	protected	yield

注意，let 和 yield 是第5版新增的保留字；其他保留字都是第3版定义的。为了最大程度地保证兼容性，建议读者将第3版定义的保留字外加 let 和 yield 作为编程时的参考。

在实现 ECMAScript 3 的 JavaScript 引擎中使用关键字作标识符，会导致“Identifier Expected”错误。而使用保留字作标识符可能会也可能不会导致相同的错误，具体取决于特定的引擎。

第5版对使用关键字和保留字的规则进行了少许修改。关键字和保留字虽然仍然不能作为标识符使用，但现在可以用作对象的属性名。一般来说，最好都不要使用关键字和保留字作为标识符和属性名，以便与将来的 ECMAScript 版本兼容。

除了上面列出的保留字和关键字，ECMA-262 第5版对 eval 和 arguments 还施加了限制。在严格模式下，这两个名字也不能作为标识符或属性名，否则会抛出错误。

3.3 变量

ECMAScript 的变量是松散类型的，所谓松散类型就是可以用来保存任何类型的数据。换句话说，每个变量仅仅是一个用于保存值的占位符而已。定义变量时要使用 var 操作符（注意 var 是一个关键字），后跟变量名（即一个标识符），如下所示：

```
var message;
```

这行代码定义了一个名为 message 的变量，该变量可以用来保存任何值（像这样未经过初始化的变量，会保存一个特殊的值——undefined，相关内容将在 3.4 节讨论）。ECMAScript 也支持直接初始化变量，因此在定义变量的同时就可以设置变量的值，如下所示：

```
var message = "hi";
```

在此，变量 message 中保存了一个字符串值“hi”。像这样初始化变量并不会把它标记为字符串类型；初始化的过程就是给变量赋一个值那么简单。因此，可以在修改变量值的同时修改值的类型，如下所示：

```
var message = "hi";  
message = 100; // 有效，但不推荐
```

在这个例子中，变量 message 一开始保存了一个字符串值“hi”，然后该值又被一个数字值 100 取代。虽然我们不建议修改变量所保存值的类型，但这种操作在 ECMAScript 中完全有效。

有一点必须注意，即使用 var 操作符定义的变量将成为定义该变量的作用域中的局部变量。也就是说，如果在函数中使用 var 定义一个变量，那么这个变量在函数退出后就会被销毁，例如：

```
function test(){  
    var message = "hi"; // 局部变量  
}  
test();  
alert(message); // 错误！
```

这里, 变量 `message` 是在函数中使用 `var` 定义的。当函数被调用时, 就会创建该变量并为其赋值。而在此之后, 这个变量又会立即被销毁, 因此例子中的下一行代码就会导致错误。不过, 可以像下面这样省略 `var` 操作符, 从而创建一个全局变量:

```
function test(){  
    message = "hi"; // 全局变量  
}  
test();  
alert(message); // "hi"
```

这个例子省略了 `var` 操作符, 因而 `message` 就成了全局变量。这样, 只要调用过一次 `test()` 函数, 这个变量就有了定义, 就可以在函数外部的任何地方被访问到。



虽然省略 `var` 操作符可以定义全局变量, 但这也不是我们推荐的做法。因为在局部作用域中定义的全局变量很难维护, 而且如果有意地忽略了 `var` 操作符, 也会由于相应变量不会马上就有定义而导致不必要的混乱。给未经声明的变量赋值在严格模式下会导致抛出 `ReferenceError` 错误。

可以使用一条语句定义多个变量, 只要像下面这样把每个变量 (初始化或不初始化均可) 用逗号分隔开即可:

```
var message = "hi",  
    found = false,  
    age = 29;
```

这个例子定义并初始化了 3 个变量。同样由于 ECMAScript 是松散类型的, 因而使用不同类型初始化变量的操作可以放在一条语句中来完成。虽然代码里的换行和变量缩进不是必需的, 但这样做可以提高可读性。

在严格模式下, 不能定义名为 `eval` 或 `arguments` 的变量, 否则会导致语法错误。

3.4 数据类型

ECMAScript 中有 5 种简单数据类型 (也称为基本数据类型): `Undefined`、`Null`、`Boolean`、`Number` 和 `String`。还有 1 种复杂数据类型——`Object`, `Object` 本质上是由一组无序的名值对组成的。ECMAScript 不支持任何创建自定义类型的机制, 而所有值最终都将是上述 6 种数据类型之一。乍一看, 好像只有 6 种数据类型不足以表示所有数据; 但是, 由于 ECMAScript 数据类型具有动态性, 因此的确没有再定义其他数据类型的必要了。


3.4.1 typeof 操作符

鉴于 ECMAScript 是松散类型的, 因此需要有一种手段来检测给定变量的数据类型——`typeof` 就是负责提供这方面信息的操作符。对一个值使用 `typeof` 操作符可能返回下列某个字符串:

- ❑ `"undefined"`——如果这个值未定义;
- ❑ `"boolean"`——如果这个值是布尔值;
- ❑ `"string"`——如果这个值是字符串;

- "number"——如果这个值是数值;
- "object"——如果这个值是对象或 null;
- "function"——如果这个值是函数。

下面是几个使用 `typeof` 操作符的例子:



```
var message = "some string";
alert(typeof message);      // "string"
alert(typeof(message));    // "string"
alert(typeof 95);           // "number"
```

typeofExample01.htm

这几个例子说明, `typeof` 操作符的操作数可以是变量 (`message`), 也可以是数值字面量。注意, `typeof` 是一个操作符而不是函数, 因此例子中的圆括号尽管可以使用, 但不是必需的。


有些时候, `typeof` 操作符会返回一些令人迷惑但技术上却正确的值。比如, 调用 `typeof null` 会返回 "object", 因为特殊值 `null` 被认为是一个空的对象引用。Safari 5 及之前版本、Chrome 7 及之前版本在对正则表达式调用 `typeof` 操作符时会返回 "function", 而其他浏览器在这种情况下会返回 "object"。



从技术角度讲, 函数在 ECMAScript 中是对象, 不是一种数据类型。然而, 函数也确实有一些特殊的属性, 因此通过 `typeof` 操作符来区分函数和其他对象是有必要的。

3.4.2 Undefined 类型

Undefined 类型只有一个值, 即特殊的 `undefined`。在使用 `var` 声明变量但未对其加以初始化时, 这个变量的值就是 `undefined`, 例如:



```
var message;
alert(message == undefined); //true
```

UndefinedExample01.htm

这个例子只声明了变量 `message`, 但未对其进行初始化。比较这个变量与 `undefined` 字面量, 结果表明它们是相等的。这个例子与下面的例子是等价的:

```
var message = undefined;
alert(message == undefined); //true
```

UndefinedExample02.htm

这个例子使用 `undefined` 值显式初始化了变量 `message`。但我们没有必要这么做, 因为未经初始化的值默认就会取得 `undefined` 值。



一般而言, 不存在需要显式地把一个变量设置为 `undefined` 值的情况。字面值 `undefined` 的主要目的是用于比较, 而 ECMA-262 第 3 版之前的版本中并没有规定这个值。第 3 版引入这个值是为了正式区分空对象指针与未经初始化的变量。

不过, 包含 `undefined` 值的变量与尚未定义的变量还是不一样的。看看下面这个例子:

```
var message; // 这个变量声明之后默认取得了 undefined 值

// 下面这个变量并没有声明
// var age


alert(message);    // "undefined"
alert(age);        // 产生错误
```

UndefinedExample03.htm

3

运行以上代码, 第一个警告框会显示变量 `message` 的值, 即 `"undefined"`。而第二个警告框——由于传递给 `alert()` 函数的是尚未声明的变量 `age`——则会导致一个错误。对于尚未声明过的变量, 只能执行一项操作, 即使用 `typeof` 操作符检测其数据类型 (对未经声明的变量调用 `delete` 不会导致错误, 但这样做没什么实际意义, 而且在严格模式下确实会导致错误)。

然而, 令人困惑的是: 对未初始化的变量执行 `typeof` 操作符会返回 `undefined` 值, 而对未声明的变量执行 `typeof` 操作符同样也会返回 `undefined` 值。来看下面的例子:



```
var message; // 这个变量声明之后默认取得了 undefined 值

// 下面这个变量并没有声明
// var age

alert(typeof message);    // "undefined"
alert(typeof age);        // "undefined"
```

UndefinedExample04.htm

结果表明, 对未初始化和未声明的变量执行 `typeof` 操作符都返回了 `undefined` 值; 这个结果有其逻辑上的合理性。因为虽然这两种变量从技术角度看有本质区别, 但实际上无论对哪种变量也不可能执行真正的操作。



即便未初始化的变量会自动被赋予 `undefined` 值, 但显式地初始化变量依然是明智的选择。如果能够做到这一点, 那么当 `typeof` 操作符返回 `"undefined"` 值时, 我们就知道被检测的变量还没有被声明, 而不是尚未初始化。

3.4.3 Null 类型

`Null` 类型是第二个只有一个值的数据类型, 这个特殊的值是 `null`。从逻辑角度来看, `null` 值表示一个空对象指针, 而这也正是使用 `typeof` 操作符检测 `null` 值时会返回 `"object"` 的原因, 如下面的例子所示:

```
var car = null;
alert(typeof car);    // "object"
```

NullExample01.htm

如果定义的变量准备在将来用于保存对象,那么最好将该变量初始化为 `null` 而不是其他值。这样一来,只要直接检查 `null` 值就可以知道相应的变量是否已经保存了一个对象的引用,如下面的例子所示:

```
if (car != null){  
    // 对 car 对象执行某些操作  
}
```

实际上, `undefined` 值是派生自 `null` 值的,因此 ECMA-262 规定对它们的相等性测试要返回 `true`:

```
alert(null == undefined);    //true
```

NullExample02.htm

这里,位于 `null` 和 `undefined` 之间的相等操作符 (`==`) 总是返回 `true`,不过要注意的是,这个操作符出于比较的目的会转换其操作数(本章后面将详细介绍相关内容)。

尽管 `null` 和 `undefined` 有这样的关系,但它们的用途完全不同。如前所述,无论在什么情况下都没有必要把一个变量的值显式地设置为 `undefined`,可是同样的规则对 `null` 却不适用。换句话说,只要意在保存对象的变量还没有真正保存对象,就应该明确地让该变量保存 `null` 值。这样做不仅可以体现 `null` 作为空对象指针的惯例,而且也有助于进一步区分 `null` 和 `undefined`。

3.4.4 Boolean 类型

Boolean 类型是 ECMAScript 中使用得最多的一种类型,该类型只有两个字面值: `true` 和 `false`。这两个值与数字值不是一回事,因此 `true` 不一定等于 1,而 `false` 也不一定等于 0。以下是为变量赋 Boolean 类型值的例子:

```
var found = true;  
var lost = false;
```

需要注意的是,Boolean 类型的字面值 `true` 和 `false` 是区分大小写的。也就是说, `True` 和 `False` (以及其他的混合大小写形式)都不是 Boolean 值,只是标识符。

虽然 Boolean 类型的字面值只有两个,但 ECMAScript 中所有类型的值都有与这两个 Boolean 值等价的值。要将一个值转换为其对应的 Boolean 值,可以调用转型函数 `Boolean()`,如下例所示:

```
var message = "Hello world!";  
var messageAsBoolean = Boolean(message);
```

BooleanExample01.htm

在这个例子中,字符串 `message` 被转换成了一个 Boolean 值,该值被保存在 `messageAsBoolean` 变量中。可以对任何数据类型的值调用 `Boolean()` 函数,而且总会返回一个 Boolean 值。至于返回的这个值是 `true` 还是 `false`,取决于要转换值的数据类型及其实际值。下表给出了各种数据类型及其对应的转换规则。

数据类型	转换为true的值	转换为false的值
Boolean	true	false
String	任何非空字符串	"" (空字符串)

(续)

数据类型	转换为true的值	转换为false的值
Number	任何非零数值（包括无穷大）	0和NaN（参见本章后面有关NaN的内容）
Object	任何对象	null
Undefined	n/a ^①	undefined

这些转换规则对理解流控制语句（如 if 语句）自动执行相应的 Boolean 转换非常重要，请看下面的代码：



```
var message = "Hello world!";
if (message){
    alert("Value is true");
}
```

[BooleanExample02.htm](#)

运行这个示例，就会显示一个警告框，因为字符串 message 被自动转换成了对应的 Boolean 值（true）。由于存在这种自动执行的 Boolean 转换，因此确切地知道在流控制语句中使用的是什么变量至关重要。错误地使用一个对象而不是一个 Boolean 值，就有可能彻底改变应用程序的流程。

3.4.5 Number 类型

Number 类型应该是 ECMAScript 中最令人关注的数据类型了，这种类型使用 IEEE754 格式来表示整数和浮点数值（浮点数值在某些语言中也被称为双精度数值）。为支持各种数值类型，ECMA-262 定义了不同的数值字面量格式。

最基本的数值字面量格式是十进制整数，十进制整数可以像下面这样直接在代码中输入：

```
var intNum = 55;           // 整数
```

除了以十进制表示外，整数还可以通过八进制（以 8 为基数）或十六进制（以 16 为基数）的字面值来表示。其中，八进制字面值的第一位必须是零（0），然后是八进制数字序列（0~7）。如果字面值中的数值超出了范围，那么前导零将被忽略，后面的数值将被当作十进制数值解析。请看下面的例子：

```
var octalNum1 = 070;       // 八进制的 56
var octalNum2 = 079;       // 无效的八进制数值——解析为 79
var octalNum3 = 08;        // 无效的八进制数值——解析为 8
```

八进制字面量在严格模式下是无效的，会导致支持的 JavaScript 引擎抛出错误。

十六进制字面值的前两位必须是 0x，后跟任何十六进制数字（0~9 及 A~F）。其中，字母 A~F 可以大写，也可以小写。如下面的例子所示：

```
var hexNum1 = 0xA;         // 十六进制的 10
var hexNum2 = 0x1f;        // 十六进制的 31
```

在进行算术计算时，所有以八进制和十六进制表示的数值最终都将被转换成十进制数值。

^① n/a（或 N/A），是 not applicable 的缩写，意思是“不适用”。



鉴于 JavaScript 中保存数值的方式，可以保存正零 (+0) 和负零 (-0)。正零和负零被认为相等，但为了读者更好地理解上下文，这里特别做此说明。

1. 浮点数值

所谓浮点数值，就是该数值中必须包含一个小数点，并且小数点后面必须至少有一位数字。虽然小数点前面可以没有整数，但我们不推荐这种写法。以下是浮点数值几个例子：

```
var floatNum1 = 1.1;
var floatNum2 = 0.1;
var floatNum3 = .1;           // 有效，但不推荐
```

由于保存浮点数值需要的内存空间是保存整数值的两倍，因此 ECMAScript 会不失时机地将浮点数值转换为整数值。显然，如果小数点后面没有跟任何数字，那么这个数值就可以作为整数值来保存。同样地，如果浮点数值本身表示的就是一个整数（如 1.0），那么该值也会被转换为整数，如下面的例子所示：

```
var floatNum1 = 1.;           // 小数点后面没有数字——解析为 1
var floatNum2 = 10.0;         // 整数——解析为 10
```

对于那些极大或极小的数值，可以用 e 表示法（即科学计数法）表示的浮点数值表示。用 e 表示法表示的数值等于 e 前面的数值乘以 10 的指数次幂。ECMAScript 中 e 表示法的格式也是如此，即前面是一个数值（可以是整数也可以是浮点数），中间是一个大写或小写的字母 E，后面是 10 的幂中的指数，该幂值将用来与前面的数相乘。下面是一个使用 e 表示法表示数值的例子：

```
var floatNum = 3.125e7;       // 等于 31250000
```

在这个例子中，使用 e 表示法表示的变量 floatNum 的形式虽然简洁，但它的实际值则是 31250000。在此，e 表示法的实际含义就是“3.125 乘以 10^7 ”。

也可以使用 e 表示法表示极小的数值，如 0.00000000000000003，这个数值可以使用更简洁的 3e-17 表示。在默认情况下，ECMAScript 会将那些小数点后面带有 6 个零以上的浮点数值转换为以 e 表示法表示的数值（例如，0.0000003 会被转换成 3e-7）。

浮点数值的最高精度是 17 位小数，但在进行算术计算时其精确度远远不如整数。例如，0.1 加 0.2 的结果不是 0.3，而是 0.30000000000000004。这个小小的舍入误差会导致无法测试特定的浮点数值。例如：

```
if (a + b == 0.3){           // 不要做这样的测试！
    alert("You got 0.3.");
}
```

在这个例子中，我们测试的是两个数的和是不是等于 0.3。如果这两个数是 0.05 和 0.25，或者是 0.15 和 0.15 都不会有问题。而如前所述，如果这两个数是 0.1 和 0.2，那么测试将无法通过。因此，永远不要测试某个特定的浮点数值。



关于浮点数值计算会产生舍入误差的问题，有一点需要明确：这是使用基于 IEEE754 数值的浮点计算的通病，ECMAScript 并非独此一家；其他使用相同数值格式的语言也存在这个问题。

2. 数值范围

由于内存的限制, ECMAScript 并不能保存世界上所有的数值。ECMAScript 能够表示的最小数值保存在 `Number.MIN_VALUE` 中——在大多数浏览器中, 这个值是 `5e-324`; 能够表示的最大数值保存在 `Number.MAX_VALUE` 中——在大多数浏览器中, 这个值是 `1.7976931348623157e+308`。如果某次计算的结果得到了一个超出 JavaScript 数值范围的值, 那么这个数值将被自动转换成特殊的 `Infinity` 值。具体来说, 如果这个数值是负数, 则会被转换成 `-Infinity` (负无穷), 如果这个数值是正数, 则会被转换成 `Infinity` (正无穷)。

如上所述, 如果某次计算返回了正或负的 `Infinity` 值, 那么该值将无法继续参与下一次的计算, 因为 `Infinity` 不是能够参与计算的数值。要想确定一个数值是不是有穷的 (换句话说, 是不是位于最小和最大的数值之间), 可以使用 `isFinite()` 函数。这个函数在参数位于最小与最大数值之间时会返回 `true`, 如下面的例子所示:

```
var result = Number.MAX_VALUE + Number.MAX_VALUE;  
alert(isFinite(result)); //false
```

尽管在计算中很少出现某些值超出表示范围的情况, 但在执行极小或极大数值的计算时, 检测监控这些值是可能的, 也是必需的。



访问 `Number.NEGATIVE_INFINITY` 和 `Number.POSITIVE_INFINITY` 也可以得到负和正 `Infinity` 的值。可以想见, 这两个属性中分别保存着 `-Infinity` 和 `Infinity`。

3. NaN

`NaN`, 即非数值 (Not a Number) 是一个特殊的数值, 这个数值用于表示一个本来要返回数值的操作数未返回数值的情况 (这样就不会抛出错误了)。例如, 在其他编程语言中, 任何数值除以 0 都会导致错误, 从而停止代码执行。但在 ECMAScript 中, 任何数值除以 0 会返回 `NaN`, 因此不会影响其他代码的执行。

`NaN` 本身有两个非同寻常的特点。首先, 任何涉及 `NaN` 的操作 (例如 `NaN/10`) 都会返回 `NaN`, 这个特点在多步计算中有可能导致问题。其次, `NaN` 与任何值都不相等, 包括 `NaN` 本身。例如, 下面的代码会返回 `false`:

```
alert(NaN == NaN); //false
```

针对 `NaN` 的这两个特点, ECMAScript 定义了 `isNaN()` 函数。这个函数接受一个参数, 该参数可以是任何类型, 而函数会帮我们确定这个参数是否 “不是数值”。`isNaN()` 在接收到一个值之后, 会尝试将这个值转换为数值。某些不是数值的值会直接转换为数值, 例如字符串 `"10"` 或 `Boolean` 值。而任何不能被转换为数值的值都会导致这个函数返回 `true`。请看下面的例子:

```
alert(isNaN(NaN)); //true  
alert(isNaN(10)); //false (10 是一个数值)  
alert(isNaN("10")); //false (可以被转换成数值 10)  
alert(isNaN("blue")); //true (不能转换成数值)  
alert(isNaN(true)); //false (可以被转换成数值 1)
```



这个例子测试了5个不同的值。测试的第一个值是NaN本身，结果当然会返回true。然后分别测试了数值10和字符串"10"，结果这两个测试都返回了false，因为前者本身就是数值，而后者可以被转换成数值。但是，字符串"blue"不能被转换成数值，因此函数返回了true。由于Boolean值true可以转换成数值1，因此函数返回false。



尽管有点儿不可思议，但 `isNaN()` 确实也适用于对象。在基于对象调用 `isNaN()` 函数时，会首先调用对象的 `valueOf()` 方法，然后确定该方法返回的值是否可以转换为数值。如果不能，则基于这个返回值再调用 `toString()` 方法，再测试返回值。而这个过程也是 ECMAScript 中内置函数和操作符的一般执行流程，更详细的内容请参见 3.5 节。

4. 数值转换

有3个函数可以把非数值转换为数值：`Number()`、`parseInt()`和`parseFloat()`。第一个函数，即转型函数 `Number()` 可以用于任何数据类型，而另两个函数则专门用于把字符串转换成数值。这3个函数对于同样的输入会有返回不同的结果。

`Number()` 函数的转换规则如下。

- ❑ 如果是 Boolean 值，true 和 false 将分别被转换为 1 和 0。
- ❑ 如果是数字值，只是简单的传入和返回。
- ❑ 如果是 null 值，返回 0。
- ❑ 如果是 undefined，返回 NaN。
- ❑ 如果是字符串，遵循下列规则：
 - 如果字符串中只包含数字（包括前面带正号或负号的情况），则将其转换为十进制数值，即"1"会变成 1，"123"会变成 123，而"011"会变成 11（注意：前导的零被忽略了）；
 - 如果字符串中包含有效的浮点格式，如"1.1"，则将其转换为对应的浮点数值（同样，也会忽略前导零）；
 - 如果字符串中包含有效的十六进制格式，例如"0xf"，则将其转换为相同大小的十进制整数值；
 - 如果字符串是空的（不包含任何字符），则将其转换为 0；
 - 如果字符串中包含除上述格式之外的字符，则将其转换为 NaN。
- ❑ 如果是对象，则调用对象的 `valueOf()` 方法，然后依照前面的规则转换返回的值。如果转换的结果是 NaN，则调用对象的 `toString()` 方法，然后再次依照前面的规则转换返回的字符串值。

根据这么多的规则使用 `Number()` 把各种数据类型转换为数值确实有点复杂。下面还是给出几个具体的例子吧。

```
var num1 = Number("Hello world!"); //NaN
var num2 = Number(""); //0
var num3 = Number("000011"); //11
var num4 = Number(true); //1
```

首先, 字符串 "Hello world!" 会被转换为 NaN, 因为其中不包含任何有意义的数字值。空字符串会被转换为 0。字符串 "000011" 会被转换为 11, 因为忽略了其前导的零。最后, true 值被转换为 1。



一元加操作符 (3.5.1 节将介绍) 的操作与 Number() 函数相同。

由于 Number() 函数在转换字符串时比较复杂而且不够合理, 因此在处理整数的时候更常用的是 parseInt() 函数。parseInt() 函数在转换字符串时, 更多的是看其是否符合数值模式。它会忽略字符串前面的空格, 直至找到第一个非空格字符。如果第一个字符不是数字字符或者负号, parseInt() 就会返回 NaN; 也就是说, 用 parseInt() 转换空字符串会返回 NaN (Number() 对空字符串返回 0)。如果第一个字符是数字字符, parseInt() 会继续解析第二个字符, 直到解析完所有后续字符或者遇到了一个非数字字符。例如, "1234blue" 会被转换为 1234, 因为 "blue" 会被完全忽略。类似地, "22.5" 会被转换为 22, 因为小数点并不是有效的数字字符。

如果字符串中的第一个字符是数字字符, parseInt() 也能够识别出各种整数格式 (即前面讨论的十进制、八进制和十六进制数)。也就是说, 如果字符串以 "0x" 开头且后跟数字字符, 就会将其当作一个十六进制整数; 如果字符串以 "0" 开头且后跟数字字符, 则会将其当作一个八进制数来解析。

为了更好地理解 parseInt() 函数的转换规则, 下面给出一些例子:



```
var num1 = parseInt("1234blue"); // 1234
var num2 = parseInt(""); // NaN
var num3 = parseInt("0xA"); // 10 (十六进制数)
var num4 = parseInt(22.5); // 22
var num5 = parseInt("070"); // 56 (八进制数)
var num6 = parseInt("70"); // 70 (十进制数)
var num7 = parseInt("0xf"); // 15 (十六进制数)
```

NumberExample05.htm

在使用 parseInt() 解析像八进制字面量的字符串时, ECMAScript 3 和 5 存在分歧。例如:

```
//ECMAScript 3 认为是 56 (八进制), ECMAScript 5 认为是 0 (十进制)
var num = parseInt("070");
```

在 ECMAScript 3 JavaScript 引擎中, "070" 被当成八进制字面量, 因此转换后的值是十进制的 56。而在 ECMAScript 5 JavaScript 引擎中, parseInt() 已经不具有解析八进制值的能力, 因此前导的零会被认为无效, 从而将这个值当成 "0", 结果就得到十进制的 0。在 ECMAScript 5 中, 即使是在严格模式下也会如此。

为了消除在使用 parseInt() 函数时可能导致的上述困惑, 可以为这个函数提供第二个参数: 转换时使用的基数 (即多少进制)。如果知道要解析的值是十六进制格式的字符串, 那么指定基数 16 作为第二个参数, 可以保证得到正确的结果, 例如:

```
var num = parseInt("0xAF", 16); //175
```

实际上, 如果指定了 16 作为第二个参数, 字符串可以不带前面的 "0x", 如下所示:

```
var num1 = parseInt("AF", 16); //175
var num2 = parseInt("AF"); //NaN
```

NumberExample06.htm

这个例子中的第一个转换成功了，而第二个则失败了。差别在于第一个转换传入了基数，明确告诉 `parseInt()` 要解析一个十六进制格式的字符串；而第二个转换发现第一个字符不是数字字符，因此就自动终止了。

指定基数会影响到转换的输出结果。例如：



```
var num1 = parseInt("10", 2);    //2   (按二进制解析)
var num2 = parseInt("10", 8);    //8   (按八进制解析)
var num3 = parseInt("10", 10);   //10  (按十进制解析)
var num4 = parseInt("10", 16);   //16  (按十六进制解析)
```

NumberExample07.htm

不指定基数意味着让 `parseInt()` 决定如何解析输入的字符串，因此为了避免错误的解析，我们建议无论在什么情况下都明确指定基数。



多数情况下，我们要解析的都是十进制数值，因此始终将 10 作为第二个参数是非常必要的。

与 `parseInt()` 函数类似，`parseFloat()` 也是从第一个字符（位置 0）开始解析每个字符。而且也是一直解析到字符串末尾，或者解析到遇见一个无效的浮点数字字符为止。也就是说，字符串中的第一个小数点是有效的，而第二个小数点就是无效的了，因此它后面的字符串将被忽略。举例来说，“22.34.5”将会被转换为 22.34。

除了第一个小数点有效之外，`parseFloat()` 与 `parseInt()` 的第二个区别在于它始终都会忽略前导的零。`parseFloat()` 可以识别前面讨论过的所有浮点数值格式，也包括十进制整数格式。但十六进制格式的字符串则始终会被转换成 0。由于 `parseFloat()` 只解析十进制值，因此它没有用第二个参数指定基数的用法。最后还要注意一点：如果字符串包含的是一个可解析为整数的数（没有小数点，或者小数点后都是零），`parseFloat()` 会返回整数。以下是使用 `parseFloat()` 转换数值的几个典型示例。

```
var num1 = parseFloat("1234blue");    //1234   (整数)
var num2 = parseFloat("0xA");         //0
var num3 = parseFloat("22.5");        //22.5
var num4 = parseFloat("22.34.5");     //22.34
var num5 = parseFloat("0908.5");      //908.5
var num6 = parseFloat("3.125e7");     //31250000
```

NumberExample08.htm

3.4.6 String 类型

`String` 类型用于表示由零或多个 16 位 Unicode 字符组成的字符序列，即字符串。字符串可以由双引号 (") 或单引号 (') 表示，因此下面两种字符串的写法都是有效的：

```
var firstName = "Nicholas";
var lastName = 'Zakas';
```

与 PHP 中的双引号和单引号会影响对字符串的解释方式不同，ECMAScript 中的这两种语法形式没有什么区别。用双引号表示的字符串和用单引号表示的字符串完全相同。不过，以双引号开头的字符串

也必须以双引号结尾，而以单引号开头的字符串必须以单引号结尾。例如，下面这种字符串表示法会导致语法错误：

```
var firstName = 'Nicholas'; // 语法错误（左右引号必须匹配）
```

1. 字符字面量

String 数据类型包含一些特殊的字符字面量，也叫转义序列，用于表示非打印字符，或者具有其他用途的字符。这些字符字面量如下表所示：

字 面 量	含 义
\n	换行
\t	制表
\b	空格
\r	回车
\f	进纸
\\	斜杠
\'	单引号（'），在用单引号表示的字符串中使用。例如：'He said, \'hey.\''
\"	双引号（"），在用双引号表示的字符串中使用。例如："He said, \"hey.\""
\xnn	以十六进制代码nn表示的一个字符（其中n为0~F）。例如，\x41表示"A"
\unnnn	以十六进制代码nnnn表示的一个Unicode字符（其中n为0~F）。例如，\u03a3表示希腊字符Σ

这些字符字面量可以出现在字符串中的任意位置，而且也将被作为一个字符来解析，如下面的例子所示：

```
var text = "This is the letter sigma: \u03a3.";
```

这个例子中的变量 text 有 28 个字符，其中 6 个字符长的转义序列表示 1 个字符。

任何字符串的长度都可以通过访问其 length 属性取得，例如：

```
alert(text.length); // 输出 28
```

这个属性返回的字符数包括 16 位字符的数目。如果字符串中包含双字节字符，那么 length 属性可能不会精确地返回字符串中的字符数目。

2. 字符串的特点

ECMAScript 中的字符串是不可变的，也就是说，字符串一旦创建，它们的值就不能改变。要改变某个变量保存的字符串，首先要销毁原来的字符串，然后再用另一个包含新值的字符串填充该变量，例如：

```
var lang = "Java";  
lang = lang + "Script";
```

以上示例中的变量 lang 开始时包含字符串"Java"。而第二行代码把 lang 的值重新定义为"Java"与"Script"的组合，即"JavaScript"。实现这个操作的过程如下：首先创建一个能容纳 10 个字符的新字符串，然后在这个字符串中填充"Java"和"Script"，最后一步是销毁原来的字符串"Java"和字符串"Script"，因为这两个字符串已经没用了。这个过程是在后台发生的，而这也是在某些旧版本的

浏览器（例如版本低于 1.0 的 Firefox、IE6 等）中拼接字符串时速度很慢的原因所在。但这些浏览器后来的版本已经解决了这个低效率问题。

3. 转换为字符串

要把一个值转换为一个字符串有两种方式。第一种是使用几乎每个值都有的 `toString()` 方法（第 5 章将讨论这个方法的特点）。这个方法唯一要做的就是返回相应值的字符串表现。来看下面的例子：

```
var age = 11;
var ageAsString = age.toString();           // 字符串"11"
var found = true;
var foundAsString = found.toString();       // 字符串"true"
```

StringExample01.htm

数值、布尔值、对象和字符串值（没错，每个字符串也都有一个 `toString()` 方法，该方法返回字符串的一个副本）都有 `toString()` 方法。但 `null` 和 `undefined` 值没有这个方法。

多数情况下，调用 `toString()` 方法不必传递参数。但是，在调用数值的 `toString()` 方法时，可以传递一个参数：输出数值的基数。默认情况下，`toString()` 方法以十进制格式返回数值的字符串表示。而通过传递基数，`toString()` 可以输出以二进制、八进制、十六进制，乃至其他任意有效进制格式表示的字符串值。下面给出几个例子：

```
var num = 10;
alert(num.toString());           // "10"
alert(num.toString(2));          // "1010"
alert(num.toString(8));          // "12"
alert(num.toString(10));         // "10"
alert(num.toString(16));         // "a"
```

StringExample02.htm

通过这个例子可以看出，通过指定基数，`toString()` 方法会改变输出的值。而数值 10 根据基数的不同，可以在输出时被转换为不同的数值格式。注意，默认的（没有参数的）输出值与指定基数 10 时的输出值相同。

在不知道要转换的值是不是 `null` 或 `undefined` 的情况下，还可以使用转型函数 `String()`，这个函数能够将任何类型的值转换为字符串。`String()` 函数遵循下列转换规则：

- 如果值有 `toString()` 方法，则调用该方法（没有参数）并返回相应的结果；
- 如果值是 `null`，则返回 `"null"`；
- 如果值是 `undefined`，则返回 `"undefined"`。

下面再看几个例子：

```
var value1 = 10;
var value2 = true;
var value3 = null;
var value4;

alert(String(value1));           // "10"
alert(String(value2));           // "true"
alert(String(value3));           // "null"
alert(String(value4));           // "undefined"
```

StringExample03.htm

这里先后转换了 4 个值：数值、布尔值、null 和 undefined。数值和布尔值的转换结果与调用 toString() 方法得到的结果相同。因为 null 和 undefined 没有 toString() 方法，所以 String() 函数就返回了这两个值的字面量。



要把某个值转换为字符串，可以使用加号操作符（3.5 节讨论）把它与一个字符串（""）加在一起。

3.4.7 Object 类型

ECMAScript 中的对象其实就是一组数据和功能的集合。对象可以通过执行 new 操作符后跟要创建的对象类型的名称来创建。而创建 Object 类型的实例并为其添加属性和（或）方法，就可以创建自定义对象，如下所示：

```
var o = new Object();
```

这个语法与 Java 中创建对象的语法相似；但在 ECMAScript 中，如果不给构造函数传递参数，则可以省略后面的那一对圆括号。也就是说，在像前面这个示例一样不传递参数的情况下，完全可以省略那对圆括号（但这不是推荐的做法）：

```
var o = new Object; // 有效，但不推荐省略圆括号
```

仅仅创建 Object 的实例并没有什么用处，但关键是要理解一个重要的思想：即在 ECMAScript 中，（就像 Java 中的 java.lang.Object 对象一样）Object 类型是所有它的实例的基础。换句话说，Object 类型所具有的任何属性和方法也同样存在于更具体的对象中。

Object 的每个实例都具有下列属性和方法。

- ❑ Constructor：保存着用于创建当前对象的函数。对于前面的例子而言，构造函数（constructor）就是 Object()。
- ❑ hasOwnProperty(propertyName)：用于检查给定的属性在当前对象实例中（而不是在实例的原型中）是否存在。其中，作为参数的属性名（propertyName）必须以字符串形式指定（例如：o.hasOwnProperty("name")）。
- ❑ isPrototypeOf(object)：用于检查传入的对象是否是另一个对象的原型（第 5 章将讨论原型）。
- ❑ propertyIsEnumerable(propertyName)：用于检查给定的属性是否能够使用 for-in 语句（本章后面将会讨论）来枚举。与 hasOwnProperty() 方法一样，作为参数的属性名必须以字符串形式指定。
- ❑ toLocaleString()：返回对象的字符串表示，该字符串与执行环境的地区对应。
- ❑ toString()：返回对象的字符串表示。
- ❑ valueOf()：返回对象的字符串、数值或布尔值表示。通常与 toString() 方法的返回值相同。

由于在 ECMAScript 中 Object 是所有对象的基础，因此所有对象都具有这些基本的属性和方法。第 5 章和第 6 章将详细介绍 Object 与其他对象的关系。



从技术角度讲, ECMA-262 中对象的行为不一定适用于 JavaScript 中的其他对象。浏览器环境中的对象, 比如 BOM 和 DOM 中的对象, 都属于宿主对象, 因为它们是由宿主实现提供和定义的。ECMA-262 不负责定义宿主对象, 因此宿主对象可能会也可能不会继承 Object。

3.5 操作符

ECMA-262 描述了一组用于操作数据值的操作符, 包括算术操作符 (如加号和减号)、位操作符、关系操作符和相等操作符。ECMAScript 操作符的与众不同之处在于, 它们能够适用于很多值, 例如字符串、数字值、布尔值, 甚至对象。不过, 在应用于对象时, 相应的操作符通常都会调用对象的 `valueOf()` 和 (或) `toString()` 方法, 以便取得可以操作的值。

3.5.1 一元操作符

只能操作一个值的操作符叫做一元操作符。一元操作符是 ECMAScript 中最简单的操作符。

1. 递增和递减操作符

递增和递减操作符直接借鉴自 C, 而且各有两个版本: 前置型和后置型。顾名思义, 前置型应该位于要操作的变量之前, 而后置型则应该位于要操作的变量之后。因此, 在使用前置递增操作符给一个数值加 1 时, 要把两个加号 (++) 放在这个数值变量前面, 如下所示:

```
var age = 29;
++age;
```

在这个例子中, 前置递增操作符把 `age` 的值变成了 30 (为 29 加上了 1)。实际上, 执行这个前置递增操作与执行以下操作的效果相同:

```
var age = 29;
age = age + 1;
```

执行前置递减操作的方法也类似, 结果会从一个数值中减去 1。使用前置递减操作符时, 要把两个减号 (--) 放在相应变量的前面, 如下所示:

```
var age = 29;
--age;
```

这样, `age` 变量的值就减少为 28 (从 29 中减去了 1)。

执行前置递增和递减操作时, 变量的值都是在语句被求值以前改变的。(在计算机科学领域, 这种情况通常被称作副效应。)请看下面这个例子。




```
var age = 29;
var anotherAge = --age + 2;

alert(age);           // 输出 28
alert(anotherAge);    // 输出 30
```


这个例子中变量 `anotherAge` 的初始值等于变量 `age` 的值前置递减之后加 2。由于先执行了减法操作，`age` 的值变成了 28，所以再加上 2 的结果就是 30。

由于前置递增和递减操作与执行语句的优先级相等，因此整个语句会从左至右被求值。再看一个例子：



```
var num1 = 2;
var num2 = 20;
var num3 = --num1 + num2;    // 等于 21
var num4 = num1 + num2;      // 等于 21
```

[IncrementDecrementExample02.htm](#)

3

在这里，`num3` 之所以等于 21 是因为 `num1` 先减去了 1 才与 `num2` 相加。而变量 `num4` 也等于 21 是因为相应的加法操作使用了 `num1` 减去 1 之后的值。

后置型递增和递减操作符的语法不变（仍然分别是++和--），只不过要放在变量的后面而不是前面。后置递增和递减与前置递增和递减有一个非常重要的区别，即递增和递减操作是在包含它们的语句被求值之后才执行的。这个区别在某些情况下不是什么问题，例如：

```
var age = 29;
age++;
```

把递增操作符放在变量后面并不会改变语句的结果，因为递增是这条语句的唯一操作。但是，当语句中还包含其他操作时，上述区别就会非常明显了。请看下面的例子：

```
var num1 = 2;
var num2 = 20;
var num3 = num1-- + num2;    // 等于 22
var num4 = num1 + num2;      // 等于 21
```

[IncrementDecrementExample03.htm](#)

这里仅仅将前置递减改成了后置递减，就立即可以看到差别。在前面使用前置递减的例子中，`num3` 和 `num4` 最后都等于 21。而在这个例子中，`num3` 等于 22，`num4` 等于 21。差别的根源在于，这里在计算 `num3` 时使用了 `num1` 的原始值（2）完成了加法计算，而 `num4` 则使用了递减后的值（1）。

所有这 4 个操作符对任何值都适用，也就是它们不仅适用于整数，还可以用于字符串、布尔值、浮点数值和对象。在应用于不同的值时，递增和递减操作符遵循下列规则。

- ❑ 在应用于一个包含有效数字字符的字符串时，先将其转换为数字值，再执行加减 1 的操作。字符串变量变成数值变量。
- ❑ 在应用于一个不包含有效数字字符的字符串时，将变量的值设置为 NaN（第 4 章将详细讨论）。字符串变量变成数值变量。
- ❑ 在应用于布尔值 `false` 时，先将其转换为 0 再执行加减 1 的操作。布尔值变量变成数值变量。
- ❑ 在应用于布尔值 `true` 时，先将其转换为 1 再执行加减 1 的操作。布尔值变量变成数值变量。
- ❑ 在应用于浮点数值时，执行加减 1 的操作。
- ❑ 在应用于对象时，先调用对象的 `valueOf()` 方法（第 5 章将详细讨论）以取得一个可供操作的值。然后对该值应用前述规则。如果结果是 NaN，则在调用 `toString()` 方法后再应用前述规则。对象变量变成数值变量。

以下示例展示了上面的一些规则：



```
var s1 = "2";
var s2 = "z";
var b = false;
var f = 1.1;
var o = {
  valueOf: function() {
    return -1;
  }
};

s1++; // 值变成数值 3
s2++; // 值变成 NaN
b++; // 值变成数值 1
f--; // 值变成 0.10000000000000009 (由于浮点舍入错误所致)
o--; // 值变成数值 -2
```

IncrementDecrementExample04.htm

2. 一元加和减操作符

绝大多数开发人员对一元加和减操作符都不会陌生，而且这两个 ECMAScript 操作符的作用与数学书上讲的完全一样。一元加操作符以一个加号 (+) 表示，放在数值前面，对数值不会产生任何影响，如下面的例子所示：

```
var num = 25;
num = +num; // 仍然是 25
```

不过，在对非数值应用一元加操作符时，该操作符会像 `Number()` 转型函数一样对这个值执行转换。换句话说，布尔值 `false` 和 `true` 将被转换为 0 和 1，字符串值会被按照一组特殊的规则进行解析，而对象是先调用它们的 `valueOf()` 和 (或) `toString()` 方法，再转换得到的值。

下面的例子展示了对不同数据类型应用一元加操作符的结果：



```
var s1 = "01";
var s2 = "1.1";
var s3 = "z";
var b = false;
var f = 1.1;
var o = {
  valueOf: function() {
    return -1;
  }
};

s1 = +s1; // 值变成数值 1
s2 = +s2; // 值变成数值 1.1
s3 = +s3; // 值变成 NaN
b = +b; // 值变成数值 0
f = +f; // 值未变，仍然是 1.1
o = +o; // 值变成数值 -1
```

UnaryPlusMinusExample01.htm

一元减操作符主要用于表示负数，例如将 1 转换成 -1。下面的例子演示了这个简单的转换过程：

```
var num = 25;
num = -num; // 变成了 -25
```

在将一元减操作符应用于数值时，该值会变成负数（如上面的例子所示）。而当应用于非数值时，一元减操作符遵循与一元加操作符相同的规则，最后再将得到的数值转换为负数，如下面的例子所示：

```
var s1 = "01";
var s2 = "1.1";
var s3 = "z";
var b = false;
var f = 1.1;
var o = {
    valueOf: function() {
        return -1;
    }
};

s1 = -s1;           // 值变成了数值-1
s2 = -s2;           // 值变成了数值-1.1
s3 = -s3;           // 值变成了NaN
b = -b;             // 值变成了数值0
f = -f;             // 变成了-1.1
o = -o;             // 值变成了数值1
```

UnaryPlusMinusExample02.htm

一元加和减操作符主要用于基本的算术运算，也可以像前面示例所展示的一样用于转换数据类型。

3.5.2 位操作符

位操作符用于在最基本的层次上，即按内存中表示数值的位来操作数值。ECMAScript 中的所有数值都以 IEEE-754 64 位格式存储，但位操作符并不直接操作 64 位的值。而是先将 64 位的值转换成 32 位的整数，然后执行操作，最后再将结果转换回 64 位。对于开发人员来说，由于 64 位存储格式是透明的，因此整个过程就像是只存在 32 位的整数一样。

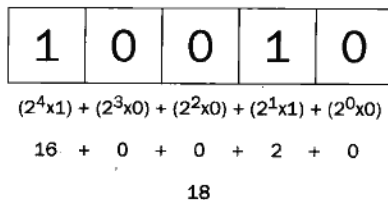
[illegible]

图 3-1

负数同样以二进制码存储，但使用的格式是二进制补码。计算一个数值的二进制补码，需要经过下列 3 个步骤：

这里，对 25 执行按位非操作，结果得到了-26。这也验证了按位非操作的本质：操作数的负值减 1。因此，下面的代码也能得到相同的结果：

```
var num1 = 25;  
var num2 = -num1 - 1;  
alert(num2); // "-26"
```

虽然以上代码也能返回同样的结果，但由于按位非是在数值表示的最底层执行操作，因此速度更快。

2. 按位与 (AND)

按位与操作符由一个和号字符 (&) 表示，它有两个操作数。从本质上讲，按位与操作就是将两个数值的每一位对齐，然后根据下表中的规则，对相同位置上的两个数执行 AND 操作：

第一个数值的位	第二个数值的位	结 果
1	1	1
1	0	0
0	1	0
0	0	0

简而言之，按位与操作只在两个数值的对应位都是 1 时才返回 1，任何一位是 0，结果都是 0。

下面看一个对 25 和 3 执行按位与操作的例子：



```
var result = 25 & 3;  
alert(result); //1
```

[BitwiseAndExample01.htm](#)

可见，对 25 和 3 执行按位与操作的结果是 1。为什么呢？请看其底层操作：

```
25 = 0000 0000 0000 0000 0000 0000 0001 1001  
3  = 0000 0000 0000 0000 0000 0000 0000 0011  
-----  
AND = 0000 0000 0000 0000 0000 0000 0000 0001
```

原来，25 和 3 的二进制码对应位上只有一位同时是 1，而其他位的结果自然都是 0，因此最终结果等于 1。

3. 按位或 (OR)

按位或操作符由一个竖线符号 (|) 表示，同样也有两个操作数。按位或操作遵循下面这个真值表。

第一个数值的位	第二个数值的位	结 果
1	1	1
1	0	1
0	1	1
0	0	0

由此可见，按位或操作在有一个位是 1 的情况下就返回 1，而只有在两个位都是 0 的情况下才返回 0。如果在前面按位与的例子中对 25 和 3 执行按位或操作，则代码如下所示：



```
var result = 25 | 3;  
alert(result); //27
```

[BitwiseOrExample01.htm](#)

25 与 3 按位或的结果是 27:

```
25 = 0000 0000 0000 0000 0000 0000 0001 1001
3  = 0000 0000 0000 0000 0000 0000 0000 0011
-----
OR = 0000 0000 0000 0000 0000 0000 0001 1011
```

这两个数值的都包含 4 个 1, 因此可以把每个 1 直接放到结果中。二进制码 11011 等于十进制值 27。

4. 按位异或 (XOR)

按位异或操作符由一个插入符号 (^) 表示, 也有两个操作数。以下是按位异或的真值表。

第一个数值的位	第二个数值的位	结 果
1	1	0
1	0	1
0	1	1
0	0	0

按位异或与按位或的不同之处在于, 这个操作在两个数值对应位上只有一个 1 时才返回 1, 如果对应的两位都是 1 或都是 0, 则返回 0。

对 25 和 3 执行按位异或操作的代码如下所示:

```
var result = 25 ^ 3;
alert(result);    //26
```

BitwiseXorExample01.htm

25 与 3 按位异或的结果是 26, 其底层操作如下所示:

```
25 = 0000 0000 0000 0000 0000 0000 0001 1001
3  = 0000 0000 0000 0000 0000 0000 0000 0011
-----
XOR = 0000 0000 0000 0000 0000 0000 0001 1010
```

这两个数值都包含 4 个 1, 但第一位上则都是 1, 因此结果的第一位变成了 0。而其他位上的 1 在另一个数值中都没有对应的 1, 可以直接放到结果中。二进制码 11010 等于十进制值 26 (注意这个结果比执行按位或时小 1)。

5. 左移

左移操作符由两个小于号 (<<) 表示, 这个操作符会将数值的所有位向左移动指定的位数。例如, 如果将数值 2 (二进制码为 10) 向左移动 5 位, 结果就是 64 (二进制码为 1000000), 代码如下所示:

```
var oldValue = 2;           // 等于二进制的 10
var newValue = oldValue << 5; // 等于二进制的 1000000, 十进制的 64
```

LeftShiftExample01.htm

注意, 在向左移位后, 原数值的右侧多出了 5 个空位。左移操作会以 0 来填充这些空位, 以便得到的结果是一个完整的 32 位二进制数 (见图 3-2)。





```
alert(!!"blue");      //true
alert(!!0);           //false
alert(!!NaN);         //false
alert(!!"");          //false
alert(!!12345);       //true
```

LogicalNotExample02.htm

2. 逻辑与

逻辑与操作符由两个和号 (&&) 表示, 有两个操作数, 如下面的例子所示:

```
var result = true && false;
```

逻辑与的真值表如下:

第一个操作数	第二个操作数	结 果
true	true	true
true	false	false
false	true	false
false	false	false

逻辑与操作可以应用于任何类型的操作数, 而不仅仅是布尔值。在有一个操作数不是布尔值的情况下, 逻辑与操作就不一定返回布尔值; 此时, 它遵循下列规则:

- ❑ 如果第一个操作数是对象, 则返回第二个操作数;
- ❑ 如果第二个操作数是对象, 则只有在第一个操作数的求值结果为 true 的情况下才会返回该对象;
- ❑ 如果两个操作数都是对象, 则返回第二个操作数;
- ❑ 如果有一个操作数是 null, 则返回 null;
- ❑ 如果有一个操作数是 NaN, 则返回 NaN;
- ❑ 如果有一个操作数是 undefined, 则返回 undefined。

逻辑与操作属于短路操作, 即如果第一个操作数能够决定结果, 那么就不会再对第二个操作数求值。

对于逻辑与操作而言, 如果第一个操作数是 false, 则无论第二个操作数是什么值, 结果都不再可能是 true 了。来看下面的例子:



```
var found = true;
var result = (found && someUndefinedVariable);    // 这里会发生错误
alert(result);    // 这一行不会执行
```

LogicalAndExample01.htm

在上面的代码中, 当执行逻辑与操作时会发生错误, 因为变量 someUndefinedVariable 没有声明。由于变量 found 的值是 true, 所以逻辑与操作符会继续对变量 someUndefinedVariable 求值。但 someUndefinedVariable 尚未定义, 因此就会导致错误。这说明不能在逻辑与操作中使用了未定义的值。如果像下面这个例中一样, 将 found 的值设置为 false, 就不会发生错误了:

```
var found = false;
var result = (found && someUndefinedVariable);    // 不会发生错误
alert(result);    // 会执行 ("false")
```

LogicalAndExample02.htm

在这个例子中，警告框会显示出来。无论变量 `someUndefinedVariable` 有没有定义，也永远不会对它求值，因为第一个操作数的值是 `false`。而这就意味着逻辑与操作的结果必定是 `false`，根本用不着再对 `&&` 右侧的操作数求值了。在使用逻辑与操作符时要始终铭记它是一个短路操作符。

3. 逻辑或

逻辑或操作符由两个竖线符号 (`||`) 表示，有两个操作数，如下面的例子所示：

```
var result = true || false;
```


逻辑或的真值表如下：

第一个操作数	第二个操作数	结 果
True	true	true
True	false	true
false	true	true
false	false	false

与逻辑与操作相似，如果有一个操作数不是布尔值，逻辑或也不一定返回布尔值；此时，它遵循下列规则：

- 如果第一个操作数是对象，则返回第一个操作数；
- 如果第一个操作数的求值结果为 `false`，则返回第二个操作数；
- 如果两个操作数都是对象，则返回第一个操作数；
- 如果两个操作数都是 `null`，则返回 `null`；
- 如果两个操作数都是 `NaN`，则返回 `NaN`；
- 如果两个操作数都是 `undefined`，则返回 `undefined`。

与逻辑与操作符相似，逻辑或操作符也是短路操作符。也就是说，如果第一个操作数的求值结果为 `true`，就不会对第二个操作数求值了。下面看一个例子：



```
var found = true;
var result = (found || someUndefinedVariable);    // 不会发生错误
alert(result);    // 会执行 ("true")
```

LogicalOrExample01.htm

这个例子跟前面的例子一样，变量 `someUndefinedVariable` 也没有定义。但是，由于变量 `found` 的值是 `true`，而变量 `someUndefinedVariable` 永远不会被求值，因此结果就会输出 `"true"`。如果像下面这个例子一样，把 `found` 的值改为 `false`，就会导致错误：

```
var found = false;
var result = (found || someUndefinedVariable);    // 这里会发生错误
alert(result);    // 这一行不会执行
```

LogicalOrExample02.htm

我们可以利用逻辑或的这一行为来避免为变量赋 `null` 或 `undefined` 值。例如：

```
var myObject = preferredObject || backupObject;
```

在这个例子中，变量 `myObject` 将被赋予等号后面两个值中的一个。变量 `preferredObject` 中包含优先赋给变量 `myObject` 的值，变量 `backupObject` 负责在 `preferredObject` 中不包含有效值的

情况下提供后备值。如果 preferredObject 的值不是 null, 那么它的值将被赋给 myObject; 如果是 null, 则将 backupObject 的值赋给 myObject。ECMAScript 程序的赋值语句经常会使用这种模式, 本书也将采用这种模式。

3.5.4 乘性操作符

ECMAScript 定义了 3 个乘性操作符: 乘法、除法和求模。这些操作符与 Java、C 或者 Perl 中的相应操作符用途类似, 只不过在操作数为非数值的情况下会执行自动的类型转换。如果参与乘法计算的某个操作数不是数值, 后台会先使用 Number() 转型函数将其转换为数值。也就是说, 空字符串将被当作 0, 布尔值 true 将被当作 1。

1. 乘法

乘法操作符由一个星号 (*) 表示, 用于计算两个数值的乘积。其语法类似于 C, 如下面的例子所示:

```
var result = 34 * 56;
```

在处理特殊值的情况下, 乘法操作符遵循下列特殊的规则:

- ❑ 如果操作数都是数值, 执行常规的乘法计算, 即两个正数或两个负数相乘的结果还是正数, 而如果只有一个操作数有符号, 那么结果就是负数。如果乘积超过了 ECMAScript 数值的表示范围, 则返回 Infinity 或 -Infinity;
- ❑ 如果有一个操作数是 NaN, 则结果是 NaN;
- ❑ 如果是 Infinity 与 0 相乘, 则结果是 NaN;
- ❑ 如果是 Infinity 与非 0 数值相乘, 则结果是 Infinity 或 -Infinity, 取决于有符号操作数的符号;
- ❑ 如果是 Infinity 与 Infinity 相乘, 则结果是 Infinity;
- ❑ 如果有一个操作数不是数值, 则在后台调用 Number() 将其转换为数值, 然后再应用上面的规则。

2. 除法

除法操作符由一个斜线符号 (/) 表示, 执行第二个操作数除第一个操作数的计算, 如下面的例子所示:

```
var result = 66 / 11;
```

与乘法操作符类似, 除法操作符对特殊的值也有特殊的处理规则。这些规则如下:

- ❑ 如果操作数都是数值, 执行常规的除法计算, 即两个正数或两个负数相除的结果还是正数, 而如果只有一个操作数有符号, 那么结果就是负数。如果商超过了 ECMAScript 数值的表示范围, 则返回 Infinity 或 -Infinity;
- ❑ 如果有一个操作数是 NaN, 则结果是 NaN;
- ❑ 如果是 Infinity 被 Infinity 除, 则结果是 NaN;
- ❑ 如果是零被零除, 则结果是 NaN;
- ❑ 如果是非零的有限数被零除, 则结果是 Infinity 或 -Infinity, 取决于有符号操作数的符号;
- ❑ 如果是 Infinity 被任何非零数值除, 则结果是 Infinity 或 -Infinity, 取决于有符号操作数的符号;

□ 如果有一个操作数不是数值，则在后台调用 `Number()` 将其转换为数值，然后再应用上面的规则。

3. 求模

求模（余数）操作符由一个百分号（%）表示，用法如下：

```
var result = 26 % 5; // 等于1
```

与另外两个乘性操作符类似，求模操作符会遵循下列特殊规则来处理特殊的值：

- 如果操作数都是数值，执行常规的除法计算，返回除得的余数；
- 如果被除数是无穷大值而除数是有限大的数值，则结果是 NaN；
- 如果被除数是有限大的数值而除数是零，则结果是 NaN；
- 如果是 Infinity 被 Infinity 除，则结果是 NaN；
- 如果被除数是有限大的数值而除数是无穷大的数值，则结果是被除数；
- 如果被除数是零，则结果是零；
- 如果有一个操作数不是数值，则在后台调用 `Number()` 将其转换为数值，然后再应用上面的规则。

3.5.5 加性操作符

加法和减法这两个加性操作符应该说是编程语言中最简单的算术操作符了。但是在 ECMAScript 中，这两个操作符却都有一系列的特殊行为。与乘性操作符类似，加性操作符也会在后台转换不同的数据类型。然而，对于加性操作符而言，相应的转换规则还稍微有点复杂。

1. 加法

加法操作符（+）的用法如下所示：

```
var result = 1 + 2;
```

如果两个操作符都是数值，执行常规的加法计算，然后根据下列规则返回结果：

- 如果有一个操作数是 NaN，则结果是 NaN；
- 如果是 Infinity 加 Infinity，则结果是 Infinity；
- 如果是 -Infinity 加 -Infinity，则结果是 -Infinity；
- 如果是 Infinity 加 -Infinity，则结果是 NaN；
- 如果是 +0 加 +0，则结果是 +0；
- 如果是 -0 加 -0，则结果是 -0；
- 如果是 +0 加 -0，则结果是 +0。

不过，如果有一个操作数是字符串，那么就要应用如下规则：

- 如果两个操作数都是字符串，则将第二个操作数与第一个操作数拼接起来；
- 如果只有一个操作数是字符串，则将另一个操作数转换为字符串，然后再将两个字符串拼接起来。

如果有一个操作数是对象、数值或布尔值，则调用它们的 `toString()` 方法取得相应的字符串值，然后再应用前面关于字符串的规则。对于 `undefined` 和 `null`，则分别调用 `String()` 函数并取得字符串 `"undefined"` 和 `"null"`。

下面来举几个例子：

```
var result1 = 5 + 5; // 两个数值相加  
alert(result1); // 10
```



```
var result2 = 5 + "5";    // 一个数值和一个字符串相加  
alert(result2);           // "55"
```

AddExample01.htm

以上代码演示了加法操作符在两种模式下的差别。第一行代码演示了正常的情况，即 5+5 等于 10（数值）。但是，如果将一个操作数改为字符串“5”，结果就变成了“55”（字符串值），因为第一个操作数也被转换成了“5”。

忽视加法操作中的数据类型是 ECMAScript 编程中最常见的一个错误。再来看一个例子：

```
var num1 = 5;  
var num2 = 10;  
var message = "The sum of 5 and 10 is " + num1 + num2;  
alert(message);    // "The sum of 5 and 10 is 510"
```

AddExample02.htm

在这个例子中，变量 message 的值是执行两个加法操作之后的结果。有人可能以为最后得到的字符串是“The sum of 5 and 10 is 15”，但实际的结果却是“The sum of 5 and 10 is 510”。之所以会这样，是因为每个加法操作是独立执行的。第一个加法操作将一个字符串和一个数值（5）拼接了起来，结果是一个字符串。而第二个加法操作又用这个字符串去加另一个数值（10），当然也会得到一个字符串。如果想先对数值执行算术计算，然后再将结果与字符串拼接起来，应该像下面这样使用圆括号：

```
var num1 = 5;  
var num2 = 10;  
var message = "The sum of 5 and 10 is " + (num1 + num2);  
alert(message);    // "The sum of 5 and 10 is 15"
```

AddExample03.htm

在这个例子中，一对圆括号把两个数值变量括在了一起，这样就会告诉解析器先计算其结果，然后再将结果与字符串拼接起来。因此，就得到了结果“The sum of 5 and 10 is 15”。

2. 减法

减法操作符（-）是另一个极为常用的操作符，其用法如下所示：

```
var result = 2 - 1;
```

与加法操作符类似，ECMAScript 中的减法操作符在处理各种数据类型转换时，同样需要遵循一些特殊规则，如下所示：

- ❑ 如果两个操作符都是数值，则执行常规的算术减法操作并返回结果；
- ❑ 如果有一个操作数是 NaN，则结果是 NaN；
- ❑ 如果是 Infinity 减 Infinity，则结果是 NaN；
- ❑ 如果是 -Infinity 减 -Infinity，则结果是 NaN；
- ❑ 如果是 Infinity 减 -Infinity，则结果是 Infinity；
- ❑ 如果是 -Infinity 减 Infinity，则结果是 -Infinity；
- ❑ 如果是 +0 减 +0，则结果是 +0；
- ❑ 如果是 +0 减 -0，则结果是 -0；

- ❑ 如果是-0减-0, 则结果是+0;
- ❑ 如果有一个操作数是字符串、布尔值、null 或 undefined, 则先在后台调用 Number() 函数将其转换为数值, 然后再根据前面的规则执行减法计算。如果转换的结果是 NaN, 则减法的结果就是 NaN;
- ❑ 如果有一个操作数是对象, 则调用对象的 valueOf() 方法以取得表示该对象的数值。如果得到的值是 NaN, 则减法的结果就是 NaN。如果对象没有 valueOf() 方法, 则调用其 toString() 方法并将得到的字符串转换为数值。

下面几个例子展示了上面的规则:

```
var result1 = 5 - true;    // 4, 因为 true 被转换成了 1
var result2 = NaN - 1;    // NaN
var result3 = 5 - 3;      // 2
var result4 = 5 - "";     // 5, 因为 "" 被转换成了 0
var result5 = 5 - "2";    // 3, 因为 "2" 被转换成了 2
var result6 = 5 - null;   // 5, 因为 null 被转换成了 0
```

SubtractExample01.htm

3.5.6 关系操作符

小于 (<)、大于 (>)、小于等于 (<=) 和大于等于 (>=) 这几个关系操作符用于对两个值进行比较, 比较的规则与我们在数学课上所学的一样。这几个操作符都返回一个布尔值, 如下面的例子所示:

```
var result1 = 5 > 3;      //true
var result2 = 5 < 3;      //false
```

RelationalOperatorsExample01.htm 中包含本节所有的代码片段

与 ECMAScript 中的其他操作符一样, 当关系操作符的操作数使用了非数值时, 也要进行数据转换或完成某些奇怪的操作。以下就是相应的规则。

- ❑ 如果两个操作数都是数值, 则执行数值比较。
- ❑ 如果两个操作数都是字符串, 则比较两个字符串对应的字符编码值。
- ❑ 如果一个操作数是数值, 则将另一个操作数转换为一个数值, 然后执行数值比较。
- ❑ 如果一个操作数是对象, 则调用这个对象的 valueOf() 方法, 用得到的结果按照前面的规则执行比较。如果对象没有 valueOf() 方法, 则调用 toString() 方法, 并用得到的结果根据前面的规则执行比较。
- ❑ 如果一个操作数是布尔值, 则先将其转换为数值, 然后再执行比较。

在使用关系操作符比较两个字符串时, 会执行一种奇怪的操作。很多人都会认为, 在比较字符串值时, 小于的意思是“在字母表中的位置靠前”, 而大于则意味着“在字母表中的位置靠后”, 但实际上完全不是那么回事。在比较字符串时, 实际比较的是两个字符串中对应位置的每个字符的字符编码值。经过这么一番比较之后, 再返回一个布尔值。由于大写字母的字符编码全部小于小写字母的字符编码, 因此我们会看到如下所示的奇怪现象:

```
var result = "Brick" < "alphabet";    //true
```

在这个例子中，字符串"Brick"被认为小于字符串"alphabet"。原因是字母 B 的字符编码为 66，而字母 a 的字符编码是 97。如果要真正按字母表顺序比较字符串，就必须把两个操作数转换为相同的大小写形式（全部大写或全部小写），然后再执行比较，如下所示：

```
var result = "Brick".toLowerCase() < "alphabet".toLowerCase(); //false
```

通过将两个操作数都转换为小写形式，就可以得出"alphabet"按字母表顺序排在"Brick"之前的正确判断了。

另一种奇怪的现象发生在比较两个数字字符串的情况下，比如下面这个例子：

```
var result = "23" < "3"; //true
```

确实，当比较字符串"23"是否小于"3"时，结果居然是 true。这是因为两个操作数都是字符串，而字符串比较的是字符编码（"2"的字符编码是 50，而"3"的字符编码是 51）。不过，如果像下面例子中一样，将一个操作数改为数值，比较的结果就正常了：

```
var result = "23" < 3; //false
```

此时，字符串"23"会被转换成数值 23，然后再与 3 进行比较，因此就会得到合理的结果。在比较数值和字符串时，字符串都会被转换成数值，然后再以数值方式与另一个数值比较。当然，这个规则对前面的例子是适用的。可是，如果那个字符串不能被转换成一个合理的数值呢？比如：

```
var result = "a" < 3; // false, 因为"a"被转换成了NaN
```

由于字母"a"不能转换成合理的数值，因此就被转换成了 NaN。根据规则，任何操作数与 NaN 进行关系比较，结果都是 false。于是，就出现了下面这个有趣的现象：

```
var result1 = NaN < 3; //false  
var result2 = NaN >= 3; //false
```

按照常理，如果一个值不小于另一个值，则一定是大于或等于那个值。然而，在与 NaN 进行比较时，这两个比较操作的结果都返回了 false。

3.5.7 相等操作符

确定两个变量是否相等是编程中的一个非常重要的操作。在比较字符串、数值和布尔值的相等性时，问题还比较简单。但在涉及到对象的比较时，问题就变得复杂了。最早的 ECMAScript 中的相等和不等操作符会在执行比较之前，先将对象转换成相似的类型。后来，有人提出了这种转换到底是否合理的质疑。最后，ECMAScript 的解决方案就是提供两组操作符：相等和不相等——先转换再比较，全等和不全等——仅比较而不转换。

1. 相等和不相等

ECMAScript 中的相等操作符由两个等于号(==)表示，如果两个操作数相等，则返回 true。而不相等操作符由叹号后跟等于号(!=)表示，如果两个操作数不相等，则返回 true。这两个操作符都会先转换操作数（通常称为强制转型），然后再比较它们的相等性。

在转换不同的数据类型时，相等和不相等操作符遵循下列基本规则：

- 如果有一个操作数是布尔值，则在比较相等性之前先将其转换为数值——false 转换为 0，而 true 转换为 1；

- 如果一个操作数是字符串，另一个操作数是数值，在比较相等性之前先将字符串转换为数值；
- 如果一个操作数是对象，另一个操作数不是，则调用对象的 `valueOf()` 方法，用得到的基本类型值按照前面的规则进行比较；

这两个操作符在进行比较时则要遵循下列规则。

- `null` 和 `undefined` 是相等的。
- 要比较相等性之前，不能将 `null` 和 `undefined` 转换成其他任何值。
- 如果有一个操作数是 `NaN`，则相等操作符返回 `false`，而不相等操作符返回 `true`。重要提示：即使两个操作数都是 `NaN`，相等操作符也返回 `false`；因为按照规则，`NaN` 不等于 `NaN`。
- 如果两个操作数都是对象，则比较它们是不是同一个对象。如果两个操作数都指向同一个对象，则相等操作符返回 `true`；否则，返回 `false`。

下表列出了一些特殊情况及比较结果：

表达式	值	表达式	值
<code>null == undefined</code>	<code>true</code>	<code>true == 1</code>	<code>true</code>
<code>"NaN" == NaN</code>	<code>false</code>	<code>true == 2</code>	<code>false</code>
<code>5 == NaN</code>	<code>false</code>	<code>undefined == 0</code>	<code>false</code>
<code>NaN == NaN</code>	<code>false</code>	<code>null == 0</code>	<code>false</code>
<code>NaN != NaN</code>	<code>true</code>	<code>"5"==5</code>	<code>true</code>
<code>false == 0</code>	<code>true</code>		

2. 全等和不全等

除了在进行比较之前不转换操作数之外，全等和不全等操作符与相等和不相等操作符没有什么区别。全等操作符由 3 个等于号 (`===`) 表示，它只在两个操作数未经转换就相等的情况下返回 `true`，如下面的例子所示：



```
var result1 = ("55" == 55);    //true, 因为转换后相等
var result2 = ("55" === 55);  //false, 因为不同的数据类型不相等
```

[EqualityOperatorsExample02.htm](#)

在这个例子中，第一个比较使用的是相等操作符比较字符串 `"55"` 和数值 `55`，结果返回了 `true`。如前所述，这是因为字符串 `"55"` 先被转换成了数值 `55`，然后再与另一个数值 `55` 进行比较。第二个比较使用了全等操作符以不转换数值的方式比较同样的字符串和值。在不转换的情况下，字符串当然不等于数值，因此结果就是 `false`。

不全等操作符由一个叹号后跟两个等于号 (`!==`) 表示，它在两个操作数未经转换就不相等的情况下返回 `true`。例如：



```
var result1 = ("55" !== 55);   //false, 因为转换后相等
var result2 = ("55" !== 55);  //true, 因为不同的数据类型不相等
```

[EqualityOperatorsExample03.htm](#)

在这个例子中，第一个比较使用了不相等操作符，而该操作符会将字符串 `"55"` 转换成 `55`，结果就与第二个操作数（也是 `55`）相等了。而由于这两个操作数被认为相等，因此就返回了 `false`。第二个比较使用了不全等操作符。假如我们这样想：字符串 `55` 与数值 `55` 不相同吗？，那么答案一定是：是的（`true`）。

记住：`null == undefined` 会返回 `true`，因为它们是类似的值；但 `null === undefined` 会返

回 false，因为它们是不同类型的值。



由于相等和不相等操作符存在类型转换问题，而为了保持代码中数据类型的完整性，我们推荐使用全等和不全等操作符。

3.5.8 条件操作符

条件操作符应该算是 ECMAScript 中最灵活的一种操作符了，而且它遵循与 Java 中的条件操作符相同的语法形式，如下面的例子所示：

```
variable = boolean_expression ? true_value : false_value;
```

本质上，这行代码的含义就是基于对 boolean_expression 求值的结果，决定给变量 variable 赋什么值。如果求值结果为 true，则给变量 variable 赋 true_value 值；如果求值结果为 false，则给变量 variable 赋 false_value 值。再看一个例子：

```
var max = (num1 > num2) ? num1 : num2;
```

在这个例子中，max 中将会保存一个最大的值。这个表达式的意思是：如果 num1 大于 num2（关系表达式返回 true），则将 num1 的值赋给 max；如果 num1 小于或等于 num2（关系表达式返回 false），则将 num2 的值赋给 max。

3.5.9 赋值操作符

简单的赋值操作符由等于号 (=) 表示，其作用就是把右侧的值赋给左侧的变量，如下面的例子所示：

```
var num = 10;
```

如果在等于号 (=) 前面再添加乘性操作符、加性操作符或位操作符，就可以完成复合赋值操作。这种复合赋值操作相当于是对下面常规表达式的简写形式：

```
var num = 10;  
num = num + 10;
```

其中的第二行代码可以用一个复合赋值来代替：

```
var num = 10;  
num += 10;
```

每个主要算术操作符（以及个别的其他操作符）都有对应的复合赋值操作符。这些操作符如下所示：

- 乘/赋值 (*=);
- 除/赋值 (/=);
- 模/赋值 (%=);
- 加/赋值 (+=);
- 减/赋值 (-=);
- 左移/赋值 (<<=);
- 有符号右移/赋值 (>>=);

□ 无符号右移/赋值 (\gg 或 $\gg=$)。

设计这些操作符的主要目的就是简化赋值操作。使用它们不会带来任何性能的提升。

3.5.10 逗号操作符

使用逗号操作符可以在一条语句中执行多个操作，如下面的例子所示：

```
var num1=1, num2=2, num3=3;
```

逗号操作符多用于声明多个变量；但除此之外，逗号操作符还可以用于赋值。在用于赋值时，逗号操作符总会返回表达式中的最后一项，如下面的例子所示：

```
var num = (5, 1, 4, 8, 0); // num 的值为 0
```

由于 0 是表达式中的最后一项，因此 num 的值就是 0。虽然逗号的这种使用方式并不常见，但这个例子可以帮助我们理解逗号的这种行为。

3.6 语句

ECMA-262 规定了一组语句（也称为流控制语句）。从本质上看，语句定义了 ECMAScript 中的主要语法，语句通常使用一或多个关键字来完成给定任务。语句可以很简单，例如通知函数退出；也可以比较复杂，例如指定重复执行某个命令的次数。

3.6.1 if 语句

大多数编程语言中最为常用的一个语句就是 if 语句。以下是 if 语句的语法：

```
if (condition) statement1 else statement2
```

其中的 condition（条件）可以是任意表达式；而且对这个表达式求值的结果不一定是布尔值。ECMAScript 会自动调用 Boolean() 转换函数将这个表达式的结果转换为一个布尔值。如果对 condition 求值的结果是 true，则执行 statement1（语句 1），如果对 condition 求值的结果是 false，则执行 statement2（语句 2）。而且这两个语句既可以是一行代码，也可以是一个代码块（以一对花括号括起来的多行代码）。请看下面的例子。

```
if (i > 25)
    alert("Greater than 25.");           // 单行语句
else {
    alert("Less than or equal to 25."); // 代码块中的语句
}
```

[IfStatementExample01.htm](#)

不过，业界普遍推崇的最佳实践是始终使用代码块，即使要执行的只有一行代码。因为这样可以消除人们的误解，否则可能让人分不清在不同条件下要执行哪些语句。

另外，也可以像下面这样把整个 if 语句写在一行代码中：

```
if (condition1) statement1 else if (condition2) statement2 else statement3
```

但我们推荐的做法则是像下面这样：

```
if (i > 25) {  
    alert("Greater than 25.");  
} else if (i < 0) {  
    alert("Less than 0.");  
} else {  
    alert("Between 0 and 25, inclusive.");  
}
```

IfStatementExample02.htm

3

3.6.2 do-while 语句

do-while 语句是一种后测试循环语句，即只有在循环体中的代码执行之后，才会测试出口条件。换句话说，在对条件表达式求值之前，循环体内的代码至少会被执行一次。以下是 do-while 语句的语法：

```
do {  
    statement  
} while (expression);
```

下面是一个示例：

```
var i = 0;  
do {  
    i += 2;  
} while (i < 10);  
  
alert(i);
```

DoWhileStatementExample01.htm

在这个例子中，只要变量 i 的值小于 10，循环就会一直继续下去。而且变量 i 的值最初为 0，每次循环都会递增 2。



像 do-while 这种后测试循环语句最常用于循环体中的代码至少要被执行一次的情形。

3.6.3 while 语句

while 语句属于前测试循环语句，也就是说，在循环体内的代码被执行之前，就会对出口条件求值。因引，循环体内的代码有可能永远不会被执行。以下是 while 语句的语法：

```
while(expression) statement
```

下面是一个示例：

```
var i = 0;  
while (i < 10) {
```

```
i += 2;  
}
```

[WhileStatementExample01.htm](#)

在这个例子中，变量 *i* 开始时的值为 0，每次循环都会递增 2。而只要 *i* 的值小于 10，循环就会继续下去。

3.6.4 for 语句

for 语句也是一种前测试循环语句，但它具有在执行循环之前初始化变量和定义循环后要执行的代码的能力。以下是 for 语句的语法：

```
for (initialization; expression; post-loop-expression) statement
```

下面是一个示例：

```
var count = 10;  
for (var i = 0; i < count; i++){  
    alert(i);  
}
```

[ForStatementExample01.htm](#)

以上代码定义了变量 *i* 的初始值为 0。只有当条件表达式 (*i*<*count*) 返回 true 的情况下才会进入 for 循环，因此也有可能不会执行循环体中的代码。如果执行了循环体中的代码，则一定会对循环后的表达式 (*i*++) 求值，即递增 *i* 的值。这个 for 循环语句与下面的 while 语句的功能相同：

```
var count = 10;  
var i = 0;  
while (i < count){  
    alert(i);  
    i++;  
}
```

使用 while 循环做不到的，使用 for 循环同样也做不到。也就是说，for 循环只是把与循环有关的代码集中在了一个位置。

有必要指出的是，在 for 循环的变量初始化表达式中，也可以不使用 var 关键字。该变量的初始化可以在外部执行，例如：

```
var count = 10;  
var i;  
for (i = 0; i < count; i++){  
    alert(i);  
}
```

[ForStatementExample02.htm](#)

以上代码与在循环初始化表达式中声明变量的效果是一样的。由于 ECMAScript 中不存在块级作用域（第 4 章将进一步讨论这一点），因此在循环内部定义的变量也可以在外访问到。例如：

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



```
    alert(i);  
  }  
  alert(i);    //10
```

ForStatementExample03.htm

在这个例子中，会有一个警告框显示循环完成后变量 *i* 的值，这个值是 10。这是因为，即使 *i* 是在循环内部定义的一个变量，但在循环外部仍然可以访问到它。

此外，for 语句中的初始化表达式、控制表达式和循环后表达式都是可选的。将这两个表达式全部省略，就会创建一个无限循环，例如：

```
for (;;) {    // 无限循环  
    doSomething();  
}
```

而只给出控制表达式实际上就把 for 循环转换成了 while 循环，例如：

```
var count = 10;  
var i = 0;  
for (; i < count; ){  
    alert(i);  
    i++;  
}
```

ForStatementExample04.htm

由于 for 语句存在极大的灵活性，因此它也是 ECMAScript 中最常用的一个语句。

3.6.5 for-in 语句

for-in 语句是一种精准的迭代语句，可以用来枚举对象的属性。以下是 for-in 语句的语法：

```
for (property in expression) statement
```

下面是一个示例：

```
for (var propName in window) {  
    document.write(propName);  
}
```

ForInStatementExample01.htm

在这个例子中，我们使用 for-in 循环来显示了 BOM 中 window 对象的所有属性。每次执行循环时，都会将 window 对象中存在的一个属性名赋值给变量 propName。这个过程会一直持续到对象中的所有属性都被枚举一遍为止。与 for 语句类似，这里控制语句中的 var 操作符也不是必需的。但是，为了保证使用局部变量，我们推荐上面例子中的这种做法。

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。



Safari 3 以前版本的 for-in 语句中存在一个 bug, 该 bug 会导致某些属性被返回两次。

3.6.6 label 语句

使用 label 语句可以在代码中添加标签, 以便将来使用。以下是 label 语句的语法:

```
label: statement
```

下面是一个示例:

```
start: for (var i=0; i < count; i++) {  
    alert(i);  
}
```

这个例子中定义的 start 标签可以在将来由 break 或 continue 语句引用。加标签的语句一般都要与 for 语句等循环语句配合使用。

3.6.7 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。请看下面的例子:



```
var num = 0;  
  
for (var i=1; i < 10; i++) {  
    if (i % 5 == 0) {  
        break;  
    }  
    num++;  
}  
  
alert(num);    //4
```

[BreakStatementExample01.htm](#)

这个例子中的 for 循环会将变量 i 由 1 递增至 10。在循环体内, 有一个 if 语句检查 i 的值是否可以被 5 整除 (使用求模操作符)。如果是, 则执行 break 语句退出循环。另一方面, 变量 num 从 0 开始, 用于记录循环执行的次数。在执行 break 语句之后, 要执行的下一行代码是 alert() 函数, 结果显示 4。也就是说, 在变量 i 等于 5 时, 循环总共执行了 4 次; 而 break 语句的执行, 导致了循环在 num 再次递增之前就退出了。如果在这里把 break 替换为 continue 的话, 则可以看到另一种结果:



```
var num = 0;  
  
for (var i=1; i < 10; i++) {  
    if (i % 5 == 0) {  
        continue;  
    }  
}
```

```
        num++;  
    }  
  
    alert(num);    //8
```

[ContinueStatementExample01.htm](#)


例子的结果显示 8，也就是循环总共执行了 8 次。当变量 *i* 等于 5 时，循环会在 *num* 再次递增之前退出，但接下来执行的是下一次循环，即 *i* 的值等于 6 的循环。于是，循环又继续执行，直到 *i* 等于 10 时自然结束。而 *num* 的最终值之所以是 8，是因为 *continue* 语句导致它少递增了一次。

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，如下面的例子所示：

```
var num = 0;  
  
outermost:  
for (var i=0; i < 10; i++) {  
    for (var j=0; j < 10; j++) {  
        if (i == 5 && j == 5) {  
            break outermost;  
        }  
        num++;  
    }  
}  
  
alert(num);    //55
```

[BreakStatementExample02.htm](#)

在这个例子中，*outermost* 标签表示外部的 *for* 语句。如果每个循环正常执行 10 次，则 *num++* 语句就会正常执行 100 次。换句话说，如果两个循环都自然结束，*num* 的值应该是 100。但内部循环中的 *break* 语句带了一个参数：要返回到的标签。添加这个标签的结果将导致 *break* 语句不仅会退出内部的 *for* 语句（即使用变量 *j* 的循环），而且也会退出外部的 *for* 语句（即使用变量 *i* 的循环）。为此，当变量 *i* 和 *j* 都等于 5 时，*num* 的值正好是 55。同样，*continue* 语句也可以像这样与 *label* 语句联合，如下面的例子所示：



```
var num = 0;  
  
outermost:  
for (var i=0; i < 10; i++) {  
    for (var j=0; j < 10; j++) {  
        if (i == 5 && j == 5) {  
            continue outermost;  
        }  
        num++;  
    }  
}  
  
alert(num);    //95
```

[ContinueStatementExample02.htm](#)

在这种情况下, `continue` 语句会强制继续执行循环——退出内部循环, 执行外部循环。当 `j` 是 5 时, `continue` 语句执行, 而这就意味着内部循环少执行了 5 次, 因此 `num` 的结果是 95。

虽然联用 `break`、`continue` 和 `label` 语句能够执行复杂的操作, 但如果使用过度, 也会给调试带来麻烦。在此, 我们建议如果使用 `label` 语句, 一定要使用描述性的标签, 同时不要嵌套过多的循环。

3.6.8 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;  
}
```

[WithStatementExample01.htm](#)

在这个重写后的例子中, 使用 `with` 语句关联了 `location` 对象。这意味着在 `with` 语句的代码块内部, 每个变量首先被认为是一个局部变量, 而如果在局部环境中找不到该变量的定义, 就会查询 `location` 对象中是否有同名的属性。如果发现了同名属性, 则以 `location` 对象属性的值作为变量的值。

严格模式下不允许使用 `with` 语句, 否则将视为语法错误。



由于大量使用 `with` 语句会导致性能下降, 同时也会给调试代码造成困难, 因此在开发大型应用程序时, 不建议使用 `with` 语句。

3.6.9 switch 语句

`switch` 语句与 `if` 语句的关系最为密切, 而且也是在其他语言中普遍使用的一种流控制语句。ECMAScript 中 `switch` 语句的语法与其他基于 C 的语言非常接近, 如下所示:

```
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)。

从根本上讲, switch 语句就是为了让开发人员免于编写像下面这样的代码:

```
if (i == 25){  
    alert("25");  
} else if (i == 35) {  
    alert("35");  
} else if (i == 45) {  
    alert("45");  
} else {  
    alert("Other");  
}
```

而与此等价的 switch 语句如下所示:

```
switch (i) {  
    case 25:  
        alert("25");  
        break;  
    case 35:  
        alert("35");  
        break;  
    case 45:  
        alert("45");  
        break;  
    default:  
        alert("Other");  
}
```

[SwitchStatementExample01.htm](#)

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, 如下所示:

```
switch (i) {  
    case 25:  
        /* 合并两种情形 */  
    case 35:  
        alert("25 or 35");  
        break;  
    case 45:  
        alert("45");  
        break;  
    default:  
        alert("Other");  
}
```

[SwitchStatementExample02.htm](#)

虽然 ECMAScript 中的 switch 语句借鉴自其他语言,但这个语句也有自己的特色。首先,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。请看下面这个例子:

```
switch ("hello world") {  
    case "hello" + " world":  
        alert("Greeting was found.");  
        break;  
    case "goodbye":  
        alert("Closing was found.");  
        break;  
    default:  
        alert("Unexpected message was found.");  
}
```

SwitchStatementExample03.htm

在这个例子中, switch 语句使用的就是字符串。其中,第一种情形实际上是一个对字符串拼接操作求值的表达式。由于这个字符串拼接表达式的结果与 switch 的参数相等,因此结果就会显示 "Greeting was found."。而且,使用表达式作为 case 值还可以实现下列操作:



```
var num = 25;  
switch (true) {  
    case num < 0:  
        alert("Less than 0.");  
        break;  
    case num >= 0 && num <= 10:  
        alert("Between 0 and 10.");  
        break;  
    case num > 10 && num <= 20:  
        alert("Between 10 and 20.");  
        break;  
    default:  
        alert("More than 20.");  
}
```

SwitchStatementExample04.htm

这个例子首先在 switch 语句外面声明了变量 num。而之所以给 switch 语句传递表达式 true,是因为每个 case 值都可以返回一个布尔值。这样,每个 case 按照顺序被求值,直到找到匹配的值或者遇到 default 语句为止(这正是这个例子的最终结果)。



switch 语句在比较值时使用的是全等操作符,因此不会发生类型转换(例如,字符串 "10" 不等于数值 10)。

3.7 函数

函数对任何语言来说都是一个核心的概念。通过函数可以封装任意多条语句,而且可以在任何地方、任何时候调用执行。ECMAScript 中的函数使用 function 关键字来声明,后跟一组参数以及函数体。函数的基本语法如下所示:

```
function functionName(arg0, arg1,...,argN) {  
    statements  
}
```

以下是一个函数示例:

```
function sayHi(name, message) {  
    alert("Hello " + name + ", " + message);  
}
```

FunctionExample01.htm

3

这个函数可以通过其函数名来调用,后面还要加上一对圆括号和参数(圆括号中的参数如果有多个,可以用逗号隔开)。调用 sayHi() 函数的代码如下所示:

```
sayHi("Nicholas", "how are you today?");
```

这个函数的输出结果是 "Hello Nicholas, how are you today?". 函数中定义中的命名参数 name 和 message 被用作了字符串拼接的两个操作数,而结果最终通过警告框显示了出来。

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。请看下面的例子:



```
function sum(num1, num2) {  
    return num1 + num2;  
}
```

FunctionExample02.htm

这个 sum() 函数的作用是把两个值加起来返回一个结果。我们注意到,除了 return 语句之外,没有任何声明表示该函数会返回一个值。调用这个函数的示例代码如下:

```
var result = sum(5, 10);
```

这个函数会在执行完 return 语句之后停止并立即退出。因此,位于 return 语句之后的任何代码都永远不会执行。例如:

```
function sum(num1, num2) {  
    return num1 + num2;  
    alert("Hello world");    // 永远不会执行  
}
```


在这个例子中,由于调用 alert() 函数的语句位于 return 语句之后,因此永远不会显示警告框。当然,一个函数中也可以包含多个 return 语句,如下面这个例子中所示:

```
function diff(num1, num2) {  
    if (num1 < num2) {  
        return num2 - num1;  
    } else {  
        return num1 - num2;  
    }  
}
```

FunctionExample03.htm


这个例子中定义的 `diff()` 函数用于计算两个数值的差。如果第一个数比第二个小，则用第二个数减第一个数；否则，用第一个数减第二个数。代码中的两个分支都具有自己的 `return` 语句，分别用于执行正确的计算。

另外，`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。比如在下面这个例子中，就不会显示警告框：



```
function sayHi(name, message) {  
    return;  
    alert("Hello " + name + ", " + message);    //永远不会调用  
}
```

FunctionExample04.htm



推荐的做法是要么让函数始终都返回一个值，要么永远都不要返回值。否则，如果函数有时候返回值，有时候有不返回值，会给调试代码带来不便。

严格模式对函数有一些限制：


- ☐ 不能把函数命名为 `eval` 或 `arguments`；
- ☐ 不能把参数命名为 `eval` 或 `arguments`；
- ☐ 不能出现两个命名参数同名的情况。

如果发生以上情况，就会导致语法错误，代码无法执行。

3.7.1 理解参数

ECMAScript 函数的参数与大多数其他语言中函数的参数有所不同。ECMAScript 函数不介意传递进来多少个参数，也不在乎传进来参数是什么数据类型。也就是说，即便你定义的函数只接收两个参数，在调用这个函数时也未必一定要传递两个参数。可以传递一个、三个甚至不传递参数，而解析器永远不会有什么怨言。之所以会这样，原因是 ECMAScript 中的参数在内部是用一个数组来表示的。函数接收到的始终都是这个数组，而不关心数组中包含哪些参数（如果有参数的话）。如果这个数组中不包含任何元素，无所谓；如果包含多个元素，也没有问题。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。

其实，`arguments` 对象只是与数组类似（它并不是 `Array` 的实例），因为可以使用方括号语法访问它的每一个元素（即第一个元素是 `arguments[0]`，第二个元素是 `arguments[1]`，以此类推），使用 `length` 属性来确定传递进来多少个参数。在前面的例子中，`sayHi()` 函数的第一个参数的名字叫 `name`，而该参数的值也可以通过访问 `arguments[0]` 来获取。因此，那个函数也可以像下面这样重写，即不显式地使用命名参数：



```
function sayHi() {  
    alert("Hello " + arguments[0] + ", " + arguments[1]);  
}
```

FunctionExample05.htm

这个重写后的函数中不包含命名的参数。虽然没有使用 `name` 和 `message` 标识符，但函数的功能依旧。这个事实说明了 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。另外，在命名参数方面，其他语言可能需要事先创建一个函数签名，而将来的调用必须与该签名一致。但在 ECMAScript 中，没有这些条条框框，解析器不会验证命名参数。

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。下面这个函数会在每次被调用时，输出传入其中的参数个数：

```
function howManyArgs() {  
    alert(arguments.length);  
}  
  
howManyArgs("string", 45); //2  
howManyArgs();             //0  
howManyArgs(12);           //1
```

FunctionExample06.htm


执行以上代码会依次出现 3 个警告框，分别显示 2、0 和 1。由此可见，开发人员可以利用这一点让函数能够接收任意参数并分别实现适当的功能。请看下面的例子：

```
function doAdd() {  
    if(arguments.length == 1) {  
        alert(arguments[0] + 10);  
    } else if (arguments.length == 2) {  
        alert(arguments[0] + arguments[1]);  
    }  
}  
  
doAdd(10);           //20  
doAdd(30, 20);       //50
```

FunctionExample07.htm

函数 `doAdd()` 会在只有一个参数的情况下给该参数加上 10；如果是两个参数，则将那个参数简单相加并返回结果。因此，`doAdd(10)` 会返回 20，而 `doAdd(30, 20)` 则返回 50。虽然这个特性算不上完美的重载，但也足够弥补 ECMAScript 的这一缺憾了。

另一个与参数相关的重要方面，就是 `arguments` 对象可以与命名参数一起使用，如下面的例子所示：



```
function doAdd(num1, num2) {  
    if(arguments.length == 1) {  
        alert(num1 + 10);  
    } else if (arguments.length == 2) {  
        alert(arguments[0] + num2);  
    }  
}
```

FunctionExample08.htm

在重写后的这个 `doAdd()` 函数中，两个命名参数都与 `arguments` 对象一起使用。由于 `num1` 的值与 `arguments[0]` 的值相同，因此它们可以互换使用（当然，`num2` 和 `arguments[1]` 也是如此）。

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。例如：

```
function doAdd(num1, num2) {  
    arguments[1] = 10;  
    alert(arguments[0] + num2);  
}
```

FunctionExample09.htm

每次执行这个 `doAdd()` 函数都会重写第二个参数，将第二个参数的值修改为 10。因为 `arguments` 对象中的值会自动反映到对应的命名参数，所以修改 `arguments[1]`，也就修改了 `num2`，结果它们的值都会变成 10。不过，这并不是说读取这两个值会访问相同的内存空间；它们的内存空间是独立的，但它们的值会同步。但这种影响是单向的：修改命名参数不会改变 `arguments` 中对应的值。另外还要记住，如果只传入了一个参数，那么为 `arguments[1]` 设置的值不会反应到命名参数中。这是因为 `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。

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，像前面例子中那样的赋值会变得无效。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。

3.7.2 没有重载

ECMAScript 函数不能像传统意义上那样实现重载。而在其他语言（如 Java）中，可以为一个函数编写两个定义，只要这两个定义的签名（接受的参数的类型和数量）不同即可。如前所述，ECMAScript 函数没有签名，因为其参数是由包含零或多个值的数组来表示的。而没有函数签名，真正的重载是不可能做到的。

如果在 ECMAScript 中定义了两个名字相同的函数，则该名字只属于后定义的函数。请看下面的例子：

```
function addSomeNumber(num) {  
    return num + 100;  
}  
  
function addSomeNumber(num) {  
    return num + 200;  
}  
  
var result = addSomeNumber(100);    //300
```

FunctionExample10.htm

在此，函数 `addSomeNumber()` 被定义了两次。第一个版本给参数加 100，而第二个版本给参数加 200。由于后定义的函数覆盖了先定义的函数，因此当在最后一行代码中调用这个函数时，返回的结果就是 300。

如前所述，通过检查传入函数中参数的类型和数量并作出不同的反应，可以模仿方法的重载。

3.8 小结

JavaScript 的核心语言特性在 ECMA-262 中是以名为 ECMAScript 的伪语言的形式来定义的。ECMAScript 中包含了所有基本的语法、操作符、数据类型以及完成基本的计算任务所必需的对象，但没有对取得输入和产生输出的机制作出规定。理解 ECMAScript 及其纷繁复杂的各种细节，是理解其在 Web 浏览器中的实现——JavaScript 的关键。目前大多数实现所遵循的都是 ECMA-262 第 3 版，但很多也已经着手开始实现第 5 版了。以下简要总结了 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。