ECMAScript Language Specification

262

Edition 3 Final

2010-4-10

目　录

目　录.....2
简史.....5
1 作用范围.....6
2 前言.....7
3 参考文献.....8
4 概述.....9
4.1 网页脚本.....9
4.2 语言概述.....10
4.2.1 对象.....10
4.3 定义.....12
4.3.1 类型.....12
4.3.2 原语值.....12
4.3.3 对象.....12
4.3.4 构造函数.....12
4.3.5 原型.....12
4.3.6 本地对象.....12
4.3.7 内置对象.....12
4.3.8 宿主对象.....13
4.3.9 未定义值.....13
4.3.10 Undefined 类型.....13
4.3.11 空值.....13
4.3.12 Null 类型.....13
4.3.13 布尔值.....13
4.3.14 Boolean 类型.....13
4.3.15 Boolean 对象.....13
4.3.16 字符串值.....13
4.3.17 String 类型.....14
4.3.18 String 对象.....14
4.3.19 数值.....14
4.3.20 Number 类型.....14
4.3.21 Number 对象.....14
4.3.22 Infinity（无穷）.....14
4.3.23 NaN.....14
5 记法约定.....15
5.1 语法和词法文法.....15

5.1.1 上下文无关文法(Context-Free Grammars)....15
5.1.2 词法文法和正则表达式文法....15
5.1.3 数字化字符串文法....16
5.1.4 语法文法....16
5.1.5 文法记法....16
5.2 算法约定....19
6 源代码文本....21
7 词法约定....22
7.1 Unicode 格式控制字符....22
7.2 White Space....23
7.3 行结束符....23
7.4 注释....24
7.5 托肯....25
7.5.1 保留字....25
7.5.2 关键字....25
7.5.3 未来保留字....26
7.6 标识符....26
7.7 标点符号....27
7.8 常量....28
7.8.1 空值常量....28
7.8.3 布尔值常量....28
7.8.3 数值常量....28
7.8.4 字符串常量....31
7.8.5 正则表达式常量....33
7.9 自动分号插入....34
7.9.1 自动分号插入的规则....34
7.9.2 自动分号插入的例子....35
8 类型....38
8.1 未定义类型....38
8.2 空值类型....38
8.3 布尔值类型....38
8.4 字符串类型....38
8.5 数值类型....39
8.6 对象类型....40
8.6.1 属性的特征....40
8.6.2 内部属性和方法....40
8.6.2.1 [[Get]](P)....42
8.6.22 [[Put]](P,V)....42
8.6.2.3 [[CanPut]](P)....42

8.6.2.4 [[HasProperty]](P)....43
8.6.2.5 [[Delete]](P)....43
8.6.2.6 [[DefaultValue]](hint)....43
8.7 引用类型....44
8.7.1 GetValue(V)....44
8.7.1 GetValue(V)....44
8.8 列表类型....45
8.9 完结类型....45
9 类型转换....46
9.1 ToPrimitive....46
9.2 ToBoolean....46
9.3 ToNumber....47
9.3.1 对字符串类型应用 ToNumber....47
9.4 ToInteger....50
9.5 ToInt32：（32 位有符号整数）....51
9.6 ToUint32：（32 位无符号整数）....51
9.6 ToUint16：（16 位无符号整数）....51
9.8 ToString....52
9.8.1 对数值类型应用 ToString....52
9.8 ToObject....53
10 执行上下文....55
10.1 定义....55
10.1.1 函数对象....55
10.1.2 可执行代码的类型....55
10.1.3 变量实例化....55
10.1.4 作用域链和标识符判定....56
10.1.5 全局对象....56
10.1.6 活动对象....57
10.1.7 This....57
10.1.8 参数对象....57
10.2 进入执行上下文....58
10.2.1 全局代码....58
10.2.2 求值代码....58
10.2.3 函数代码....58

简史

本 ECMA 标准基于一系列原创技术，其中最著名的是 Netscape 公司的 JavaScript 和 Microsoft 公司的 JScript。该语言由 Netscape 公司的 Brendan Eich 发明并首先出现在该公司的 Navigator 2.0 浏览器中。该语言至今已经在所有来自 Netscape 的全部子系列浏览器和从 Internet Explorer3.0 开始所有来自 Microsoft 的浏览器中出现。

该标准的研发起始于 1996 年 11 月。ECMA 标准的首个版本于 1997 年 6 月被 ECMA 公共协会采用。

此 ECMA 标准曾被提交 ISO/IECJTC1 以“快速通道”审批过程的方式进行审批，并于 1998 年 8 月被批准成为 ISO/IEC16262 国际标准。1998 年 6 月，ECMA 公共协会批准了 ECMA-262 标准的第二版以保持它与 ISO/IEC16262 的完全协调。第一版和第二版之间存在着性质上的重大变化。

目前的文档定义了该标准的第三版，这包含了强大的正则表达式，更优秀的字符串处理，新的流程控制语句，try/catch 异常处理，更严密的出错定义，数字式输出格式以及一些次要变更，为即将到来的国际化语言设施和语言未来发展的作提前准备。

关于这个语言的工作并没有结束。技术委员会正在致力于作出意义重大的优化，这包括脚本在互联网上创建和使用的机制，以及与其它标准实 体（如万维网联盟(W3C)及无线应用协议论坛的等团体）间更紧密的配合。

1 作用范围

此标准定义了 ECMAScript 脚本语言。

2 前言

符合标准的 ECMAScript 实现必须提供并支持本规范中所描述的所有类型，值，对象，属性，函数，程序语法和语义。

符合这个国际标准的实现应当能解释前言中提到的字符集：Unicode 标准，2.1 版或更新版本，使用 UCS-2 或 UTF-16 作为被采纳的编码格式的 ISO/IEC 10646-1 第 3 级实现。如果被采纳的 ISO/IEC 10646-1 子集没有额外指明，则被认定为组号是 300 的 BMP 子集。如果被采纳的编码格式没有额外指明，则被认定为 UTF-16 编码格式。

符合标准的 ECMAScript 实现被允许提供额外的、在此规范描述之外的类型，值，对象，属性和函数。特别的，符合标准的 ECMAScript 实现被允许给那些此规范中所描述的对象提供此规范所没有描述的属性，以及那些属性的值。

3 参考文献

ISO/IEC 9899:1996 Programming Languages-C, including amendment 1 and technical corrigenda 1 and 2.

ISO/IEC 10646-1:1993 Information Technology--Universal Multiple-Octet Coded Character Set(UCS) plus its amendments and corrigenda.

UnicodeInc.(1996), The Unicode Standard, Version2.0. ISBN: 0-201-48345-9, Addison-Wesley Publishing Co.,MenloPark, California.

UnicodeInc.(1998), Unicode Technical Report #8: The Unicode Standard, Version2.1.

UnicodeInc.(1998), Unicode Technical Report #15: Unicode Normalization Forms.

ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic. Institute of Electrical and Electronic Engineers, NewYork(1985).

4 概述

这一节包含一个 ECMAScript 语言的非正式概述。

ECMAScript 是面向对象的编程语言，被用来演示计算技术和操控宿主环境下的计算机对象。这里定义的 ECMAScript 没有被设计成计算性自足的语言；事实上，在这篇规范中，没有为输入外部数据或输出计算结果给出任何条款。取而代之的是，我们期望 ECMAScript 程序的计算机环境可提供除了这篇规范中所描述的对象和其它语言设施之外的、某些特定环境下的 *宿主(host)* 对象，它们的描述和行为将超出此规范的所指出的范围，即它们可提供某些可被访问的属性和某些可从 ECMAScript 程序中调用的函数。

脚本语言(script language)是一类被 用于操控、自定义和自动控制现有系统设施的编程语言。在这些系统中，实用的功能可通过一个用户界面来使用，脚本语言就是一种通过程序控制那些功能的机制。 于是，我们就说此系统为对象和设施提供了一个宿主环境，它们使得脚本语言的能力变得完备。脚本语言是为了能被专业或非专业程序员所使用而设计的。为了适应 非专业程序员，语言的一些方面会多少有些不严格。

ECMAScript 从一开始就被设计成一种**网页脚本语言(Web scripting language)**作为基于网页的、客户端—服务器端(C/S)构架的建筑师，它能提供一 种机制，用来使浏览器中的网站页面更加活跃，并展示服务器端的处理情况。ECMAScript 能够为各种主机环境，以及这个文档所描述的核心脚本编程语言之外任何特定的主机环境，提供基本的脚本编程能力。

ECMAScript 中的有些语言设施类似其它编程语言；尤其是 Java 和 Self，下列文献描述了它们：

- Gosling,James,Bill Joy and GuySteele. The Java Language Specification. Addison Wesley Publishing Co.,1996.
- Ungar,David,and Smith,Randall B. Self: The Power of Simplicity. OOPSLA '87 Conference Proceedings,pp. 227-C241,Orlando,FL,October 1987.

4.1 网页脚本

网页浏览器为 ECMAScript 提供了一个宿主环境以进行客户端的计算，例如：代表窗口、菜单、浮动条、对话框、文本区域、锚点、框架、历史、cookies，以及输入/输出功能的对 象。深入说来，主机环境提供了一套向事件上附加脚本代码的方法，这些事件有焦点的改变，页面和图像的载入、卸载，出错和异常终止，点选，提交表单以及鼠标 动作等。脚本代码在 HTML 之间出现，被显示的页面是一个用户界面元素和固定或处理后的文本和图像的组合。这种脚本语言会响应用户的互交从而不需要主程序。

与浏览器不同，网页服务器为服务器端提供了另一种宿主环境，这包括代表请求、客户端、文件的对象；以及锁定或共享数据的机制。通过同时使用浏览器端和客户端脚本，在客户端与

服务器之间分配计算将成为可能，这样就能为基于网页的应用提供一个自定义化的用户界面。

4.2 语言概述

接下来是一个 ECMAScript 语言的非正式概述——这个语言的所有部分的描述。严格 说来，这个概述不是标准的一部分。

ECMAScript 是基于对象的：语言的基本部分和宿主设施由对象提供，一个 ECMAScript 程序是一组可通信的对象。ECMAScript 对象(object)都是未排序的属性 (properties)集合，零个或多个特征(attributes)来分别确定其中的各属性应如何使用——举个例子：当某个 属性的 ReadOnly（只读）特征被设为**真(true)**时， 任何通过执行 ECMAScript 代码来改变这个属性值的企图都不会生效。属性是装载其它对象的容器，如**原语值 (primitive values)**或**方法 (methods)**。原语值是下列内置类型 的一个成员：**未定义(Undefined)**，**空值(Null)**，**数 值 (Number)**，**布尔值(Boolean)**和**字符串(String)**；对象是保持内置类型**对象(Object)**的一个成员；方法是一种通过属性来访问对象的函数。

ECMAScript 定义了一集勾勒出 ECMAScript 定义实体的**内置对象(built-in object)**，它们包括 **Global**（全局）对象、**Object** 对象、**Function** （函数）对象、**Array**（数组）对象、**String** 对象、**Number** 对 象、**Math**（数学库）对象、**Date**（日期）对象、**RegExp**（正 则表达式）对象以及其它 Error 类对象：**Error**，**EvalError**（求 值错误），**RangeError**（越界错误），**ReferenceError**（引 用错误），**SyntaxError**（语法错误），**TypeError**（类 型错误），**URIError**（唯一资源定位符错误）。

ECMAScript 还定义了一个内置**运算符(operators)**集合。严格地说，它们可能不是函数或方法。ECMAScript 运算符包含了各种各样的操作：乘法运算符，加法运算符，位移运算符，关系运算符，相等关系运算符，二元位操作运算符，二元逻辑运算符，分配运算符，逗号运算符。

ECMAScript 语法被特意设计成类似 Java 的语法。ECMAScript 的语法设计能使其作为一种易于使用的脚本语言提供服务。例如，一个变量不需要拥有自己的类型声明或与其关联的属性的类型，定义函数时不需要在它们被调用的上文中出现它们的声明。

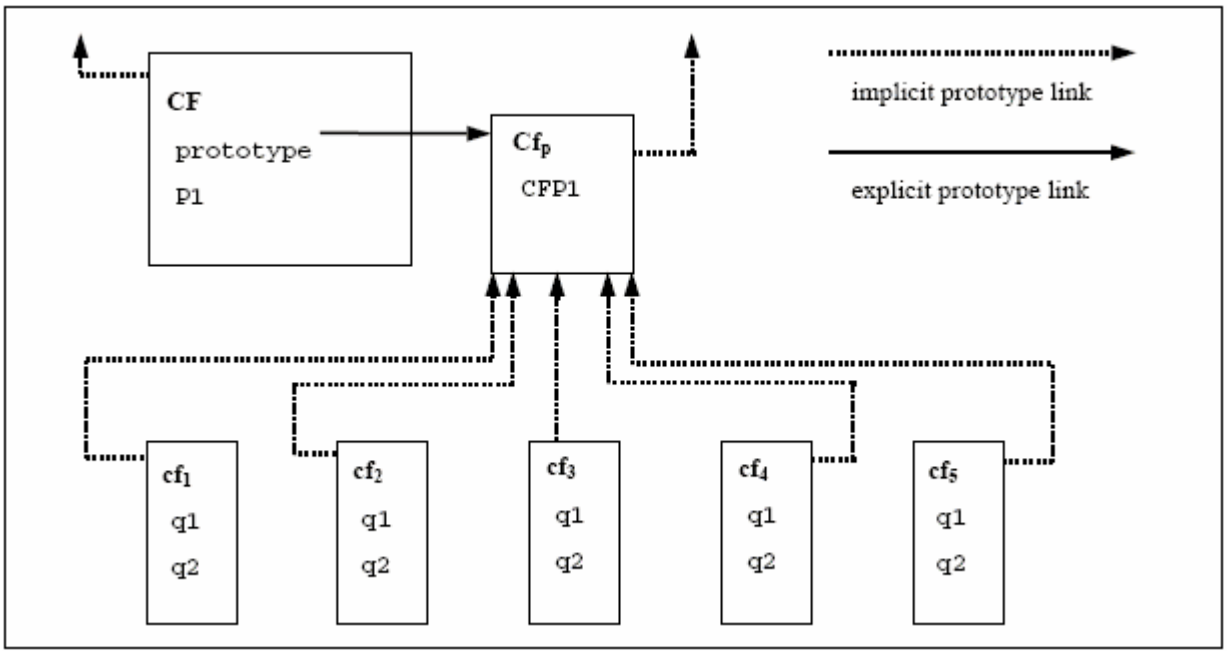
4.2.1 对象

ECMAScript 并没有严格意义上的类，在这一点上不同于 C++、Smalltalk 或者 Java，不过作为替代，它支持**构造函数(constructors)**，利用它，可通过执行代码创建对象：给对象 分配 存储，然后通过赋初始值来初始化对象属性的全部或部分。所有的构造函数都是对象，但并非所有的对象都是构造函数。每个构造函数都有一个 Prototype（原 型）属性，被用于实现**基于原型继承(prototype-based inheritance)**和**共 享属性(shared properties)**。使用构造函数来创建对象要用到 `new`（新 建）表达句；举个例子，`new String("A String")` 创建了一个新的字符串对象。不使用 `new` 调用构造函数的后果取决于这个构造函数本身。举个例子，`String("A String")` 产生一个原语字符串而非对象。

ECMAScript 支持*基于原型继承*。每一个构造函数都有一个相关联的原型，它创建的所有对象都拥有一个隐含的引用指向那个与其构造函数相关联的原型（所谓的*对象的原型*）。更深一步讲，原型还可能会拥有 到它的原型的隐含的非空引用，依此类推；这被称为*原型链(prototypechain)*。若为某对象中的一个属 性建立引用，那么此引用指向该对象的原型链中，最先拥有同名属性的对象所包含的这个属性。换句话说，首先检查被直接提及的对象是否包含某个属性；如果那 个对象包含同名属性，被引用指向的就是这个属性；如果那个对象并没有包含同名属性，接下来检查它的原型，如此继续下去。

通常，在基于类的面向对象语言中，实例装载状态，类装载方法，且继承的仅仅是结构和行为。而在 ECMAScript 中，状态和方法均由对象装载，且结构、行为、状态都会被继承。

所有没有直接包含某个它们的原型所包含的特定属性的对象，与它们的原型共享那个属性和它的值。下面的图表阐述了一切：



CF 是一个构造函数（当然也是一个对象）。使用 new 表达式，我们创建了五个对象：cf1，cf2， cf3，cf4 和 cf5。 这些对象中的每一个都包含了名为 q1 和 q2 的属性。虚线表示隐含的原型关系；比方说，cf3 的原型是 CFp。构造函数 CF 自己拥有两个属性，名为 P1 和 P2，它们对于 CFp，cf1，cf2， cf3，cf4 或 cf5 而言都是不可见的。CFp 中名为 CFP1 的属性被 cf1，cf2， cf3，cf4 和 cf5（除 了 CF）共享，这样，CFp 的隐含原型链中的所有属性没有名为 q1，q2 或 CFP1 的。需要注意的是，CF 和 CFp 之间没有隐含的原型关联。

不同于基于类的面向对象语言，ECMAScript 中的属性可以通过给它们赋值的方式，把它们动态添加给对象。也就是说，构造函数不需要给所构造的对象的部分或全部属性命名或赋值。在上面的图表中，通过给 CFp 中的属性赋新值就可以给 cf1，cf2， cf3，cf4 和 cf5 添加新的共享属性。

4.3 定义

下面给出了与 ECMAScript 有关的关键术语的非正式定义。

4.3.1 类型

类型是数据取值的集合。

4.3.2 原语值

原语值是类型 **Undefined**，**Null**，**Number**，**Boolean** 或 **String** 的一个成员。原语值是直接表示语言实现的底层数据。

4.3.3 对象

对象是未 排序的属性的集合，其中每个属性包含一个原语值、对象或函数。被作为属性保存的函数被称为方法。

4.3.4 构造函数

构造函数是 一种创建并初始化对象的函数对象。每个工作；构造函数都拥有一个相关联的原型对象，用它来实现继承和共享属性。

4.3.5 原型

原型是一 种对象，被用在 ECMAScript 中实现继承结构、状态和行为。当构造函数创建对象时，那个对象隐含引用构造函数的关联原型，以此分解属性引用。通过程序中的表达式 *constructor*.prototype 可以引用到构造函数的关联原型，通过继承，添加给对象的属性会被所有共享此原型的对象共享。

4.3.6 本地对象

本地对象指 的是由 ECMAScript 实现提供并独立于宿主环境的任何对象。这篇规范定义了标准本地对象。有些本地对象是内置的；其余的可能在 ECMAScript 程序执行的过程中被构造。

4.3.7 内置对象

内置对象指 的是由 ECMAScript 实现提供的，独立于宿主环境的，并在 ECMAScript 程序刚开始执行时就出现的对象。这篇规范定义了标准内置对象，一个 ECMAScript 实现也可能指明并定义其它的内置对象。所有的内置对象都是本地对象。

4.3.8 宿主对象

宿主对象指的是由 ECMAScript 实现提供的，使 ECMAScript 的执行环境变得完备的对象。所有非本地对象都是宿主对象。

4.3.9 未定义值

未定义值是 一种原语值，当一个变量未被赋值时被使用。

4.3.10 Undefined 类型

类型 Undefined 仅有一个值，叫做 undefined。

4.3.11 空值

空值是一种原语值，用来代表 0，空或不存在的引用。

4.3.12 Null 类型

类型 Null 仅有一个值，叫做 **null**。

4.3.13 布尔值

布尔值是 Boolean 的成员，它是两个特殊的值之一，true 和 false。

4.3.14 Boolean 类型

Boolean 类型仅用两个特殊的值表示逻辑实体。其一被称为 true 另一个被称为 false。

4.3.15 Boolean 对象

Boolean 对象是 Object 类型的一个成员，内置 Boolean 对象的实例。也就是说，在 new 表达式中使用 Boolean 构造函数提供一个布尔值参数创建出 Boolean 对象。结果对象拥有一个隐含（无命名的）属性是那个布尔值。一个 Boolean 对象可被强制为一个布尔值。

4.3.16 字符串值

字符串值是 String 类型的一个成员，它是一个有序的、长度有限的的列表，包含零个或更多 16 位无符号整数。

NOTE 即使每个值常常表示单个 16 位 UTF-16 文本，语言也不给出关于期望这些值被表示为 16 位无符号整数的限制或要求。

4.3.17 String 类型

String 类型是所有字符串值的集合。

4.3.18 String 对象

String 对象是 Object 类 型的一个成员，内置 String 对象的实例。也就是说，在 new 表达式中使用 String 构造函数提供一个字符串参数创建出 String 对象。结果对象拥有一个隐含（无命名的）属性是那个字符串值。通过以函数的方式调用 String 构造函数可以把一个 String 对象强制为一个字符串值。

4.3.19 数值

数值是 Number 类型的一个成员，是数字的直接表示。

4.3.20 Number 类型

Number 类型是表示数字的值的集合。在 ECMAScript 中，表示双 精度 64 位格式（IEEE 754）的值的集合包含特殊值“Not-a-Number”（非数字，NaN），正无穷和负无穷。

4.3.21 Number 对象

Number 对象是 Object 类 型的一个成员，内置 Number 对象的实例。也就是说，在 new 表达式中使用 Number 构造函数提供一个数值参数创建出 Number 对象。结果对象拥有一个隐含（无命名的）属性是那个数值。通过以函数的方式调用 Number 构造函数可以把一个 Number 对象强制为一个数值(15.7.1)。

4.3.22 Infinity（无穷）

原语值 **Infinity** 表示正无穷数值。这个值是 Number 类型的一个成员。

4.3.23 NaN

原语值 **NaN** 表示 IEEE 标准“非数字”值。这个值是 Number 类型的一个成员。

5 记法约定

5.1 语法和词法文法

这一节描述了本规范中使用的上下文无关文法，这是 ECMAScript 程序语法结构的定义。

5.1.1 上下文无关文法(Context-Free Grammars)

一个*上下文无关文法*由许多*产生式(productions)*组 成。每个产生式都拥有一个抽象符号作为其*左式(left-hand side)*，被称为*非终结 符(nonterminal)*，以及一个由零个或多个非终结符和*终结符(terminal)*组 成的*右式(right-hand side)*。对于每个文法而言，终结符是从一个特定的字母表中抽取的。

句子的开始由单个显式的非终结符组成，被称为*目标符(goal symbol)*，一个给定的上 下文无关文法规定了这样一种*语言*，它被表示为可能的终结符序列组成的集合（有可能是无限集），这是用产生式左式重 复替换序列中一切非终结符得到的，替换时所用的产生式的左式要和被替换的非终结符一致。

5.1.2 词法文法和正则表达式文法

ECMAScript 的*词法文法(lexical grammar)*在条款 7 中 给出。此文法 以 Unicode 字符集中的字符作为其终结符，定义了一个产生式集合，其中的产生式以目标符*输入分隔元素(InputElementDiv)*或*输入正则表达式元素(InputElementRegExp)*开 始，这描述了 Unicode 字符是如何被翻译成输入元素序列的。

为 ECMAScript 语法文法提供的除空白和注释之外的输入元素被称为 ECMAScript *托肯(token)*。这些托肯即 ECMAScript 语言中的保留字、标识符、常量和标点符号。此外，行结束符即使不被认为是托肯，它同样是输入的元素流中的一部分，并指导自动插入分号的过程(7.8.5)。普通的空白和单行注释被丢弃，且不出现在为语法文法提供的输入元素流中。对于*多行注释 (MultiLineComment)*（即形如 "/*...*/"的注释，不论它跨越了多少行），如果它没有包含行结束符，就被简单地丢弃；假使*多行注释*包含一 个以上行结束符，则用单个行结束符替换它，使之成为为语法文法提供的输入元素流的一部分。

ECMAScript 的*正则表达式文法(RegExp grammar)*在 15.10 中给出。此文法同样 以 Unicode 字符集中的字符作为其终结符，定义了一个产生式集合，其中的产生式以目标符的*模式(Pattern)*为 起始，描述了 Unicode 字符是如何被翻译成正则表达式模式的。

词法文法和正则表达式文法的产生式被识别为被两个冒号"::"分割的产生式。词法文法和正则表达式文法共享某些产生式。

5.1.3 数字化字符串文法

第二个文法被用于将字符串翻译为数值量。此文法类似词法文法中与数字常量有关的部分，以 Unicode 字符集中的字符作为其终结符。此文法在 9.3.1 中出现。数字化字符串文法的产生式被识 别为被三个冒号":::"分割的产生式。

5.1.4 语法文法

ECMAScript 的*语法文法*在条款 11，12，13 和 14 中 给出。此文法以词法文法定义的 ECMAScript 托肯作为其终结符(5.1.2)，定义了一个产生式 集合，其中的产生式以目标符 *Program* 开始，描述了托肯序列是如何构成语法正确的 ECMAScript 程序的。

若将一个 Unicode 字符流被解析为一个 ECMAScript 程序，首先，通过重复应用词法文法，它被转化为一个输入元素流；通过一次应用语法文法，这个输入元素流继续被解析。如果输入元素流中已不再留有托肯，而托 肯仍无法被解析为单个目标非终结符 *Program*，则这个程序语法出 错。

语法文法的产生式被识别为仅被一个冒号":"分割的产生式。

语法文法在章节 11，12，13，14 中 体现①，它事实上不是能被正确的 ECMAScript 程序接受的托肯序列中的一员。一个确定的额外托肯序列同样会被接受，也就是说，即使只有冒号被加入序列中的某个位置（比如在行结束字符之前），这些托肯也 会被 文法所描述。深入说来，即便终结字符出现在某些“尴尬”的位置上，被文法描述的、确定的托肯序列也不考虑其是否能被接受。

5.1.5 文法记法

词法和字符串文法的终结符，以及有些语法文法的终结符，将以等宽`(fixed width)`字体显示在文法的 产生式中，贯穿本规范中该文本被直接引用为一个终结符 的全过程。它们可出现在写成的程序中。所有的非终结字符以这种特定的方式，作为恰当的、ASCII 范围内的 Unicode 字符被识别，以区别其它 Unicode 范围内看上去相似的 Unicode 字符。

非终结符以*斜体(italic)*显示。非终结符的定义由其被定义的名字后跟一个或更多冒号引入（冒 号的数量指出产生式所属的文法）。非终结符中一个或多个可变的右式紧跟在下一行。例如，文法语法的定义：

With 语句：
`with (` *表达 式* `)` *语句*

这说明非终结符 *With 语句*表示 `with` 托肯，后跟一个左括号托肯，再跟随一个*表达式*，其后是一个右括号托肯，再后面是一个*语句*。 出现的*表达式*和*语句*它们自身都是非终结符。作为另一个例子，有文法定义：

参数列表：

赋值表达式
参数列表,*赋值表达式*

这说明一个*参数列表*可以表现为单个*赋值表达式*，或一个参 *数列表*后跟一个逗号，再跟一个*赋值表达式*。*参数列表*的 定义是*递归的(recursive)*，也就是说，它的定义中借用了它本身。结果是*参数列表*可 以包含任意数量为正的参数，以逗号隔开，每个参数表达式都是一个*赋值表达式*。像这样的非终结符的递归定义很常见。

出现在终结符或非终结符后面的下标后缀"opt"，指出这是一个*可选符号(optional symbol)*。包含可选符号的可变部分其实可以细分为两个右式，其一忽略可选元素，而另一个包括它。这意味着：

变量声明：
标识符 初始化器opt

是下述产生式的简略形式：

变量声明：
标识符
标识符 初始化器

再如：

迭代语句：
for (*入口表 达 式opt* ; *表 达式opt* ; *表 达式opt*) *语 句*

是下述产生式的简略形式：

迭代语句：
for (; *表达式opt* ; *表达式opt*)
for (*入口表 达式* ; *表达式opt* ; *表达式opt*) *语句*

它是由下面的产生式的简略形式转变得到的：

迭代语句：
for (; ; *表达式opt*) *语句*
for (; *表达式* ; *表达式opt*) *语句*
for (*入口表 达式* ; ; *表达式opt*) *语句*
for (*入口表 达式* ; *表达式* ; *表 达式opt*) *语 句*

它是由下面的产生式的简略形式转变得到的：

迭代语句：
for (; ;) *语句*
for (; ; *表达式*) *语 句*
for (; *表达式* ;) *语 句*
for (; *表达式* ; *表达式*) *语句*

for (*入口表 达式* ; ;) *语 句*
for (*入口表 达式* ; ; *表达式*) *语句*
for (*入口表 达式* ; *表达式* ;) *语句*
for (*入口表 达式* ; *表达式* ; *表 达式*) *语句*

所以说，*迭代语句*其实有八个可变右式。

如果产生式右式出现了短语"[lookahead ∉ set]"，指的是如果紧接着的输入终结符不是所给集合 *set* 的 成员，则不使用此产生式。 集合 *set* 可以被写成克里闭包中的非闭合终结符组成的列表。为方便起见，这个集合也可写成一个非终结符，这使它代表除去此非终结符的所有终结符构成的集合。例如，给 出下列定义：

十进制数字 :: one of
0 1 2 3 4 5 6 7 8 9

十进制数 ::
十进制数字
十进制数 十进制数字

定义

Lookahead 示例 ::
n [lookahead ∉ {1,3,5,7,9}] *十进制数*
十进制数字 [lookahead ∉ *十 进制数*]

当字母 n 后跟一个或多个首位是偶数的十进制数，或后跟一个后面没有另一个十进制数字的数字时，完成匹配。

如果语法文法的产生式右式出现了短语"[no *LineTerminator* here]"，它指出了该产生式是一个*受约束产生式(restricted production)*： 如果*行结束符*出现在输入流的指定位置上，这个产生式不被使用。例如，下面的产生式：

返回语句 ::
return [no *LineTerminator* here] *表达式 opt* ;

这里指出了，如果在 return 托肯和 *表达式* 之间出现了*行结束符*，则不使用此产生式。

直到出现的*行结束符*被受限产生式禁止之前，在输入的元素流中，*行结束符*允 许在两个连续托肯之间出现任意次数但对程序的语法可接受性没有影响。

如果在语法定义的冒号之后出现了单词"one of"，它表示随后的行中的所有终结符被当 作一个可变定义。例如，在 ECMAScript 的词法文法中包含下面的产生式：

非零数字 :: one of
1 2 3 4 5 6 7 8 9

这只不过是下面的产生式的简略形式：

非零数字 ::②

```
1
2
3
4
5
6
7
8
9
```

当词法文法或数字化字符串文法的产生式可变部分中出现了多字符托肯，表示这个字符序列构成一个托肯。

使用短语"`but not`"可具体指明产生式的某些特定扩充是不允许的，这样可以把指定的 扩充排除在外。例如，下面的产生式：

标识符 **::**
标识符名 `but not` *保 留字*

意思是，非终结符*标识符*可以被任意可以替换*标识符名*的 字符序列替换，同一序列不能被*保留字*替换。

最后，对于实际上不可能列出全部可变元的少量非终结符，我们用普通字体写出描述性的短语来描述它们：

源文件字符 **::**
任何 Unicode 字符

---

① 此句原为 in sections 0, 0, 0 and 0，被认为有误。
② 原书此处有 `one of`，错误。

5.2 算法约定

本规范中常会用到一种标有数字序号的列表来详细指明一个算法的步骤。这些算法是用来说明语义的。实际操作中，具体实现可能会对给定的特 性给出更有效率的算法。

若算法要产生一个值作为结果，使用说明"返回 *x*"来指明算法的结果是 *x* 的值，此时算法结束。记法 Result(*n*) 被用作"第 *n* 步的结果" 的缩写。Type(*x*) 被用作"*x* 的类型"的缩写。

加法、减法、取负、乘法、除法这些数学运算，以及这一节稍后定义的数学函数应被理解为总是使用实数做精确的数学计算，这不包括无穷大或 负零。本标准中的算法在适当的地方会建模浮点数运算，描述其步骤，处理无穷大和有符号零并进行舍入。如果数学运算或函数应用于一个浮点数，应被理解为应用 于此浮点数所代表的精确的数学值；比如，浮点数必须是有限

的，若为+0 或-0 则简单地取与之相符的数学值 0。

数学函数 abs(*x*) 返回 *x* 的绝对值，即如果 *x* 为负（小于零）则是 -*x*， 否则是 *x* 本身。

若为 *x* 正，数学函数 sign(*x*) 返回 1；为负则返回-1。在本标准中，对于 *x* 为零的情况，不使用函数 sign。

记法"*x* modulo *y*"（*y* 必须为有限的非零值）计算 *k* 值，它与 *y* 同号（或同为零），使得 abs(*k*) < abs(*y*) 且对于同样的整数 *q* 有 *x*-*k* = *q* × *y*。

数学函数 floor(*x*) 返回不大于 *x* 的最 大的整数（可接近正无穷）。

NOTE
floor(x) = x-(x modulo 1).

若定义一个"抛出异常"的算法，执行此算法结束后没有返回结果。调用它的算法也会结束，除非算法步骤到达了明确地处理这个异常的地方，处理异常的术语有"如果一个异常被抛出..."等等。只要有一个算法步骤遭遇异常，就不再认为此异常发生过。

6 源代码文本

ECMAScript 源代码文本被表示为一个 Unicode 编码的字符序列，Unicode 版本 2.1 或更新，使用 UTF-16 转换格式。我们期望文本已被 Unicode 规范化形式 C(canonical composition)规范化，Unicode 技术报告 #15 中描述了它。合乎标准的 ECMAScript 实现不要求展示任何文本的规范化形式，做出它们展示规范化文本的行为，只需展示文本自身。

源文件字符 **::**
任何 Unicode 字符

ECMAScript 源文件文本可以包含任何 Unicode 字符。所有的 Unicode 空白字符都被视为空格，所有 Unicode 行、段分隔符都被视为行分隔符。Unicode 非拉丁文字符允许出现在标识符，字符串常量，正则表达式常量和注释中。

贯穿此文档的剩余部分，短语"代码点"和单词"字符"将被用于代指 16 位无符号值，用来呈现单个 UTF-16 的 16 位单元。短语"Unicode 字符"将被用于代指抽象语言学或排版单元，呈现为单个 Unicode 标量值（可能长于 16 位，将被呈现为多于一个的代码点）。这儿仅代指单个以 Unicode 标量值形式呈现的实体：组合字符得到的字符序列仍是独立的“Unicode 字符”，即使用户认为整个序列是单个字符。

在字符串常量、正则表达式常量和标识符中，任何字符（代码点）也可以被表示为一个由六个字符组成的 Unicode 转义序列，形式为 `\u` 加上四个十六进制数字。事实上，在注释中，这样的转义序列被当作注释的一部分忽略 掉。在字符串常量或正则表达式常量中，Unicode 转义序列向常量的值提供一个字符。在标识符中，转义序列给标识符提供一个字符。

NOTE 1
虽然这篇文档常常提到在“字符串”中的“字符”和 16 位无符号整数（该字符的 UTF-16 编码）之间的“转换”，但事实上，这种转换是不存在的，因为“字符串& rdquo;中的“字符”其实正是用那个 16 位无符号整数表示的。
NOTE 2
在对待 Unicode 转义序列的行为方面，ECMAScript 语言与 Java 语言不同。例如，在 Java 程序中，Unicode 转义序列 `\u000A` 在单行注释中出现，它将被解释为一个行结束符（Unicode 字符 `000A` 代 表换行），因此下一个字符不再是注释的一部分。类似地，如果在 Java 程序中，Unicode 转义序列 `\u000A` 出现在字符串常量中，将被同样解释为一个行结束符，所以它不能允许出现在字符串常量中——要把换行作为字符串常 量的值的一部分，必须写成 `\n` 而不是 `\u000A`。 在 ECMAScript 程序中，在注释中出现的 Unicode 转义序列决不会被解释，因此不能使注释结束。类似地，在 ECMAScript 程序中，在字符串常量中出现的 Unicode 转义序列总是为字符串常量的值提供字符，而决不会被解释成行结束符或引号，这使 Unicode 转义序列不可能结束一个字符串常量。

7 词法约定

ECMAScript 程序的源代码文本首先被转换为一个输入元素的序列，其中的每一项是托肯、行结束符、注释或空白中的一个。源代码文本被从左到右扫描，重复地把最长的可能的 字符序列作为下一个输入元素。

词法文法中有两个目标符。符号*输入分隔元素*在其它语法文法中上下文中可以是一个除号(/) 或除法赋值(/=)运算符。符号*输入元素正则表达式*在其它语法文法上 下文中使用。

需要注意的是，除号和*正则表达式常量*存在于语法文法中的上下文中，且都被语法文法允许；不论如 何，在斜线号不被识别为正则表达式常量的开始这一情况下，词法文法使用*输入分隔元素*目标符。为了使它能在任何情况 下正常工作，可以把正则表达式常量用括号括起。

语法

输入分隔元素 ::

空白
行结束符
注释
托肯
分隔符

输入元素正则表达式 ::

空白
行结束符
注释
托肯
正则表达式常量

7.1 Unicode 格式控制字符

Unicode 格式控制字符（Unicode 字符数据库"Cf"分类，比如 LEFT-TO-RIGHT 标记、RIGHT-TO-LEFT 标记）被用于控制范围内的文本格式化，以使文本被相对于此的高阶协议（如标记语言）忽略。在源代码文本中允许出现这些字符也是有用的，便于查看和编辑。

格式控制字符可以在 ECMAScript 程序的任何位置出现。这些字符将在应用词法文法之前从源代码文本中被移除。因为这些字符将在处理字符串常量和正则表达式常量之前被移除，所以必须使用 Unicode 转义序列(7.6)，以使字符串和正则表达式常量中包含 Unicode 格式控制字符。

7.2 White Space

空白字符被用于提升源代码文本的可读性，以及把托肯（不可分割的词法单元）互相分隔开来，在其它方面是无关紧要的。空白字符允许出现在 两个托肯之间，也允许出现在字符串中（在这个地方，它们会被看成是有意义的字符，成为字符串常量的一部分）。但空白字符不能出现在其它任何托肯之间。

下面的字符被认为是空白字符：

代码点值	名称	正式名称
\u0009	制表符	<TAB>
\u000B	垂直制表符	<VT>
\u000B	表单结束符	<FF>
\u0020	空格	<SP>
\u00A0	非阻断空格	<NBSP>
其它类别"Zs"	其它任何 Unicode“空白分隔符”	<USP>

语法

空白字符 **::**
<TAB>
<VT>
<FF>
<SP>
<NBSP>
<USP>

7.3 行结束符

与空白字符类似，行结束字符被用于提升源代码文本的可读性，以及把托肯（不可分割的词法单元）互相分隔开来。不过，与空白字符不同的 是，行结束符对语法文法的行为有影响。一般说来，行结束符可以出现在两个托肯之间，不过有少数地方，语法文法禁止它的出现。行结束符不能出现在任何托肯之 间，即使是字符串也不行。行结束符还对自动分号插入过程有影响。

下面的字符被认为是行结束符：

代码点值	名称	正式名称
\u000A	换行	<LF>
\u000D	行末	<CR>
\u2028	行分隔符	<LS>
\u2029	段落分隔符	<PS>

语法

行结束符 **::**
<LF>
<CR>
<LS>
<PS>

7.4 注释

描述

注释可以是单行或多行的。多行注释不能自包含。

由于单行注释可以包含除*行结束符*之外的任何字符①。根据常规，托肯总是进行最长匹配，所以单行 注释总是由从 **//** 开始到行末结束的所有字符组成。不过，行尾的*行结束符*不被识别为单行注释的一部分；它被词法文法识别为分隔符，并 成为给语法文法提供的输入元素流的一部分。这一点非常重要，因为这意味着单行注释的存在与否对自动分号插入的过程没有影响。

除非*多行注释*中包含有一个行结束字符，导致整个注释被语法文法解析为一个*行 结束符*，否则抛弃注释，行为类似空白字符。

语法

*注释***::**
多行注释
单行注释

*多行注释***::**
/* *多行注释字符集* $_{opt}$ ***/**

*多行注释字符集***::**
多行注释非星号字符 多行注释字符集 $_{opt}$
***** *后置星号注释字符* $_{opt}$

*后置星号注释字符***::**
多行非正斜 杠或星号字符 多行注释字符集 $_{opt}$
***** *后星号注释字符* $_{opt}$

*多行非星号字符***::**
源代码字符 **but not** *星号* ***

*多行非正斜杠或星号字符***::**
源代码字符 **but not** *正斜杠* **/or** *星号****

*单行注释***::**

// *单行注释字符集* *opt*

单行注释字符集::
单行注释字符 *单 行注释字符集* *opt*

单行注释字符::
源代码字符 **but not** *行结束符*

---

① 此句有歧义，正确意思是"单行注释直到行末"。

7.5 托肯

语法

托肯 ::
保留字
标识符
标点符号
数字常量
字符串常量

7.5.1 *保留字*

描述

保留字不能被用作标识符。

语法

保留字 ::
关键字
未来保留字
空值常量
布尔值常量

7.5.2 关键字

下列托肯是 ECMAScript 关键字，且不能被用作 ECMAScript 程序中的标识符。

语法

关键字 :: **one of**

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

```
instanceof typeof
```

7.5.3 未来保留字

下面的单词被用作将来可能出现的扩展的关键字，因此，为了让它们被将来采用的这些扩展所允许，它们被保留。

语法

未来保留字 **:: one of**

```
 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
```

7.6 标识符

描述

解释标识符的文法根据在章节 5.16 中给出，对于 Unicode 标准的升级版 3.0 有一些小更正。此文法同时基于 Unicode 标准所具体指明的常规及非常规字符类别。在 Unicode 标准 2.1 版中具体指明的类别中的字符，必须被所有符合标准的 ECMAScript 实现认作是其它类别中的字符；不过，符合标准的 ECMAScript 实现可以允许附加那些基于 Unicode 早期版本中的类别分配的、合法的标识符字符集。

本标准具体指明了一个背离给定的 Unicode 标准的文法：允许美元号($)和下划线 (_)出现在标识符的任何位置。美元号仅应在机械生成的代码中使用。

标识符中同样允许有 Unicode 转义序列，它们在此处给标识符提供单个字符，由 *Unicode 转义序列*的 CV 算法计算（参见 7.8.4）。在 *Unicode 转义序列*之 前提前出现的 \ 不为标识符提供字符。*Unicode 转义序列*不 能被用于向标识符中放置一个字符，否则是违法的。换句话说，如果 \ *Unicode 转义序列的序列*被它的 *Unicode 转义序列*的 CV 替换，结果必定仍是合法的*标识符*，与原始的*标识符*有完全相同的字符序列。

根据 Unicode 标准来判断两个标识符不相等时，不考虑它们是否被呈现为完全相同的代码点序列（换句话说，仅要求符合标准的 ECMAScript 实现对标识符作以比特为单位的比较）。这意味着，所得到的源代码文本在到达编译器之前就已经被转化成常规的形式 C。

语法

标识符 **::**
标识符名 **but not** *保留 字*

标识符名 **::**

标识符开始
标识符名 标识符体

标识符开始 **::**
Unicode 字母
$
_
*****Unicode 转义序列*

标识符体 **::**
标识符开始
Unicode 联合标记
Unicode 数字
Unicode 连接标点符号
*****Unicode 转义序列*

Unicode 字母
以下 Unicode 分类中的任何字符： "Uppercase letter(Lu，大写字母)", "Lower caseletter(Ll，小写字母)", "Titlecase letter(Lt，标题字母)", "Modifier letter (Lm，修饰字母)","Other etter(Lo，其它字母)", 或 "Letter number(Nl，字母数字)"。

Unicode 联合标记
以下 Unicode 分类中的任何字符： "Non-spacing mark(Mn，非空格标记)" 或 "Combining spacing mark(Mc，联合空格标记)"。

Unicode 数字
以下 Unicode 分类中的任何字符："Decimal number(Nd，十进制数)"。

Unicode 连接标点符号
以下 Unicode 分类中的任何字符："Connector punctuation (Pc，连接标点符号)"。

Unicode 转义序列
参见 7.8.4

十六进制数字:: **one of**
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

7.7 标点符号

语法

标点符号 **:: one of**
```
{  }  (  )  [  ]  .  ;  ,  <  >  <=  >=  ==  !=  ===  !==  +  -  *
 %  ++  --  <<  >>  >>>  &  |  ^  !  ~  &&  ||  ?  :  =  +=  -=
 *=  %=  <<=  >>=  >>>=  &=  |=  ^=
```

除号标点符号 **:: one of**
`/ /=`

7.8 常量

语法

常量 **::**
空值常量
布尔值常量
数值常量
字符串常量
正则表达式常量①

---

① 原书此处无此句，有误。

7.8.1 空值常量

语法

空值常量 **::**
```
null
```

语义

空值常量 `null` 的值是空类型唯一的值，也就是 null。

7.8.3 布尔值常量

语法

布尔值常量 **::**
```
true
false
```

语义

布尔值常量值 `true` 是布尔值类型的值，也就是 true。

布尔值常量值 `false` 是布尔值类型的值，也就是 false。

7.8.3 数值常量

语法

数值常量 **::**

十进制常量
十六进制整数常量

十进制常量 **::**
十进制整数常量 **.** *十进制数* *opt* *指数部分* *opt*
. *十进 制数 指数部分* *opt*
十进制整数常量 指数部分 *opt*

十进制整数常量 **::**
0
非零数字 十进制数 *opt*

十进制数 **::**
十进制数字
十进制数 十进制数字

十进制数字 **:: one of**
0 1 2 3 4 5 6 7 8 9

非零数字 **:: one of**
1 2 3 4 5 6 7 8 9

指数部分 **::**
指数指示符 有符号整数

指数指示符 **:: one of**
e E

有符号整数 **::**
十进制数
+ *十进制数*
- *十进 制数*

十六进制整数常量 **::**
0x *十六进制数字*
0X *十六进制数字*
十六进制整数常量 十六进制数字

紧跟在*数值常量*后的源代码字符既不是*标识符开始*也不是*十 进制数*。

NOTE

例如：`3in`
不是两个输入元素`3`和`in`，而是一个错误。

语义

数值常量代表数值类型的值。用两个步骤来确定这个值：首先，从常量中获得数学值；然

后，用下面所描述的方式舍入这个数学值。

- *数值常量*的 MV :: *十 进制常量*的 MV 是 *十进制常量*的 MV。
- *数值常量*的 MV :: *十六进制整数常量*的 MV 是 *十六进制整数常量*的 MV。
- *十进制常量*的 MV :: *十进制整数常量* **.** 是 *十进制整数常量*的 MV。
- *十进制常量*的 MV :: *十进制整数常量* **.** *十进制数* 是 *十进制整数常量*的 MV 加 (*十进制数*的 MV 乘以 10^{-n}), 这里的 *n* 是 *十进制数*中的字符数。
- *十进制常量*的 MV :: *十进制整数常量* **.** *指数部分* 是 *十进制整数常量*的 MV 乘以 $10^{e\text{的 MV}}$,这里的 *e* 的 MV 是 *指数部分*的 MV。
- *十进制常量*的 MV :: *十进制整数常量* **.** *十进制数 指数部分* 是 (*十进制整数常量*的 MV 加 (*十进制数*的 MV 乘以 10^{-n})) 乘以 10^{e}, 这里的 *n* 是 *十进制数*中的字符数 和 *e* 的 MV 是 *指数部分*的 MV。
- *十进制常量*的 MV :: **.** *十进制数* 是 *十进制数*的 MV 乘以 10^{-n}, 这里的 *n* 是 *十进制数*中 的字符数。
- *十进制常量*的 MV :: **.** *十 进制数 指数部分*的 MV 是 *十进制数*的 MV 乘以 10^{e-n},这里的 *n* 是 *十进制数*中的字符数 且 *e* 是 *指数部分*的 MV。
- *十进制常量*的 MV :: *十进制整数常量* 是 *十进制整数常量*的 MV。
- *十进制常量*的 MV :: *十进制整数常量 指数部分* 是 *十进制整数常量*的 MV 乘以 $10^{e\text{的 MV}}$, 这里的 e 是 *指数部分*的 MV。
- *十进制整数常量*的 MV :: **0** 是 0。
- *十进制整数常量*的 MV :: *非 零数字 十进制数* 是 (*非零数字*的 MV 乘以 $10^{n\text{的 MV}}$) 加 *十进制数*的 MV, 这里的 *n* 是 *十进制数*中的字符数。
- *十进制数*的 MV :: *十 进制数字* 是 *十进制数字*的 MV.
- *十进制数*的 MV :: *十 进制数 十进制数字* 是 (*十进制数*的 MV 乘以 10) 加 *十进制数字*的 MV。
- *指数部分*的 MV :: *指 数指示符 有符号整数* 是 *有符号整数*的 MV。
- *有符号整数*的 MV :: *十 进制数* 是 *十进制数*的 MV.
- *有符号整数*的 MV :: **+** *十进制数* 是 *十进制数*的 MV.
- *有符号整数*的 MV :: **-** *十进制数*的 MV 是 *十进制数*的负 MV.
- *十进制数字*的 MV :: **0** or of *十六进制数字* :: **0** 是 0.
- *十进制数字*的 MV :: **1** or of *非零数字* :: **1** or of *十六进制数字* :: **1** 是 1. *十进制数字*的 MV :: **2** or of *非零数字* :: **2** or of *十六进制数字* :: **2** 是 2.
- *十进制数字*的 MV :: **3** or of *非零数字* :: **3** or of *十六进制数字* :: **3** 是 3.
- *十进制数字*的 MV :: **4** or of *非零数字* :: **4** or of *十六进制数字* :: **4** 是 4.
- *十进制数字*的 MV :: **5** or of *非零数字* :: **5** or of *十六进制数字* :: **5** 是 5. *十进制数字*的 MV :: **6** or of *非零数字* :: **6** or of *十六进制数字* :: **6** 是 6.
- *十进制数字*的 MV :: **7** or of *非零数字* :: **7** or of *十六进制数字* :: **7** 是 7.
- *十进制数字*的 MV :: **8** or of *非零数字* :: **8** or of *十六进制数字* :: **8** 是 8.

- *十进制数字*的 MV **::** **9** or of *非零数字* **::** **9** or of *十六进制数字* **::** **9** 是 9. *十六进制数字*的 MV **::** **a** or of *十六进制数字* **::** **A** 是 10.
- *十六进制数字*的 MV **::** **b** or of *十六进制数字* **::** **B** 是 11.
- *十六进制数字*的 MV **::** **c** or of *十六进制数字* **::** **C** 是 12.
- *十六进制数字*的 MV **::** **d** or of *十六进制数字* **::** **D** 是 13.
- *十六进制数字*的 MV **::** **e** or of *十六进制数字* **::** **E** 是 14.
- *十六进制数字*的 MV **::** **f** or of *十六进制数字* **::** **F** 是 15.
- *十六进制整数常量*的 MV **::** **0x** *十六进制数字* 是 *十六进制数字*的 MV。
- *十六进制整数常量*的 MV **::** **0X** *十六进制数字十六进制数字*的 MV。
- *十六进制整数常量*的 MV **::** *十 六进制整数常量 十六进制数字* 是 (*十六进制整数常量*的 MV 乘以 16) 加 *十六进制数字*的 MV。

一旦数值常量的 MV 被精确地确定，接下来就会被舍入为数值类型的一个值。如果 MV 是 0，那么舍入值为 **+0**； 否则，舍入值必须*正是* MV 的值（在 8.5 中定义），除非该常量是 一个有效数字超过 20 位的*十进制常量*——在这种情况下，此数字的值是下 面两种之一：一是将其 20 位之后的每个有效数字用 0 替换，产生此常量的 MV；二是将其 20 位之后的每个有效数字用 0 替换，并在 20 位有效数字之后增加数字位，产生此常量的 MV 值。判断一个数字是否为*有效数字*，首先它不能是*指数部分*的一部分，且

- 它不是 0；或
- 它的左边是一个非零值，右边是一个不在*指数部分*中的非零值。

7.8.4 字符串常量

字符串常量是被单引号或双引号括起的零个或多个字符。每个字符都可以呈现为转义序列。

语法

字符串常量 **::** **"** *双字符串字符集* $_{opt}$ **"**
' *单字符串字符* s_{opt} **'**

双字符串字符集 **::** *双字符串字符 双字符串字符集* $_{opt}$

单字符串字符 s **::** *单字符串字符 单字符串字符* s_{opt}

双字符串字符 **::** *源代码字符* **but not** *双引号* **"** **or** *反斜杠* **** **or** *行结束符*
**** *转义序列*

单字符串字符 **::** *源代码字符* **but not** *单引号* **'** **or** *反斜杠* **** **or** *行结束符*
**** *转义序列*

转义序列 **::** *字符转义序列*
0 [lookahead ? *十进制数字]*
十六进制转义序列

Unicode 转义序列

字符转义序列 **::** *单个转义字符*
非转义字符

单个转义字符 **:: one of** **' " \ b f n r t v**

非转义字符 **::** *源代码字符* **but not** *转义字符* **or** *行结束符*

转义字符 **::** *单个转义字符*
十进制数字
x
u

十六进制转义序列 **::** **x** *十六进制数字 十六进制数字*

Unicode 转义序列 **::** **u** *十六进制数字 十六进制数字 十六进制数字 十六进制数字*

非终止*十六进制数字*的定义在 7.8.3 节中给出。*源 代码字符*的描述在章节 2、6 中。

字符串常量代表字符串类型的值。该常量的字符串值(SV)的描述，取决于字符串常量各个部分提供的字符值(CV)。作为此过程的一部 分，一些字符串常量中的字符被解释为拥有数学值(MV)，正如下文和章节 7.8.3 中所描述的。

- *字符串常量*的 SV **::** **""** 是 空字符序列
- *字符串常量*的 SV **::** **''** 是 空字符序列
- *字符串常量*的 SV **::** **"** *双字符串字符集* **"** 是 the SV of *双字符串字符集*.
- *字符串常量*的 SV **::** **'** *单字符串字符s* **'** 是 *单字符串字符s* 的 SV.
- *双字符串字符集*的 SV **::** *双字符串字符* 是一个字符的序列,*双字符串字符*的 SV.
- *双字符串字符集*的 SV **::** *双字符串字符 双字符串字符集* 是 *双字符串字符*的 SV 的序列 后跟所有*双字符串字符集*的 SV 中的有序字符
- *单字符串字符集*的 SV **::** *单字符串字符* 是一个字符的序列, *单字符串字符*的 SV.
- *单字符串字符s* **::** *单字符串字符 单字符串字符s* 是 *单字符串字符*后跟所有*单字符串字符集*的 SV 中的有序字符
- *双字符串字符* **::** *源代码字符* **but not** *双引号* **"** **or** *反斜杠* **** **or** *行结束符* 是 the *源代码字符* 字符本身。
- *双字符串字符* **::** **** *转义序列* 是 *转义序列. 单字符串字符* **::** *源代码字符* **but not** *单引号* **'** **or** *反斜杠* **** **or** *行结束符* 是 *源代码字符* 字符本身。
- *单字符串字符* **::** **** *转义序列* 是 *转 义序列.*
- *转义序列* **::** *字符转义序列* 是 the *字符转义序列.*
- *转义序列* **::** **0** [lookahead ? *十进制数字]* 是 a <NUL> 字符 (Unicode 值 0000).
- *转义序列* **::** *十六进制转义序列* 是 *十六进制转义序列.*
- *转义序列* **::** *Unicode 转义序列* 是 *Unicode 转义序列.*
- *字符转义序列* **::** *单个转义字符* 是 由*单个转义字符*根据下表确定的代码点值的字符:

转义序列	代码点	名称	符号
\b	\u0008	退格	<BS>
\t	\u0009	水平制表符	<HT>
\n	\u000A	换行	<LF>
\v	\u000B	垂直制表符	<VT>
\f	\u000C	表单结束	<FF>
\r	\u000D	回车	<CR>
\"	\u0022	双引号	"
\'	\u0027	单引号	'
\\	\u005C	反斜杠	\

- *字符转义序列* **::** *非转义字符* 是 the *非转义字符*.
- *非转义字符* **::** *源代码字符* **but not** *转义字符* **or** *行结束符* 是 the *源代码字符* 字符本身。
- *十六进制转义序列* **::** **x** *十六进制数字 十六进制数字* 是以 (第一个*十六进制数字*的 MV 值的 16 倍) 加 (第二个 *十六进制数字*的 MV 值) 为代码点值的字符。
- *Unicode 转义序列* **::** **u** *十六进制数字 十六进制数字 十六进制数字 十六进制数字* 是以 (第一个 *十六进制数字*)的 MV 的 (4096 (即, 16^3)倍) 加 (第二个 *十六进制数字*)的 MV 的(256 (即 , 16^2)倍) 加 (第三个 *十六进制数字*的 16 倍) 加 (第四个 *十六进制数字*的 MV) 为代码点值的字符。

NOTE

“行结束”字符不能出现在字符串常量中，即使在它前面有一个反斜杠 \。 使行结束字符成为字符串常量的值一部分的正确方法是，使用转义序列，如 \n 或 \u000A。

7.8.5 正则表达式常量

正则表达式常量被扫描后，转换为 RegExp 对象的输入元素（章节 15.10）。 此对象在包含它的程序或函数开始求值之前被创建。求值此常量产生一个到那个对象的引用；这不产生新对象。决不能用 === 比较被程序求值为正则表达式对象的两个正则表达式常量，即使两个常量内容相同。在运行时创建 RegExp 对象，可以用 new RegExp（章节 15.10.4），或者以函数的方式调用 RegExp 构造函数（章节 15.10.3）。

下面的产生式描述了正则表达式常量的语法，以及输入元素扫描器查找正则表达式常量结尾的方法。给正则表达式构造函数传递包含*正 则表达式体*和*正则表达式徽标*的、不被解释的字符串，将以更严厉的文法，根据它们自己解释它 们。实现可以扩展正则表达式构造函数的文法，但不应扩展*正 则表达式体*和*正则表达式徽标*的产生式，以及被它们使用的产生式。

语法

正则表达式常量 **::** **/** *正则表达式体* **/** *正则表达式徽标*

正则表达式体 **::** *正则表达式首字符 正则表达式字符集*

正则表达式字符集 **::** [empty]
正则表达式字符集 正则表达式字符

正则表达式首字符 **::** *非结束符* **but not * or \ or /**
反斜杠序列

正则表达式字符 **::** *非结束符* **but not \ or /**
反斜杠序列

反斜杠序列 **::** **** *非结束符*

非结束符 **::** *源代码字符* **but not** *行结束符*

正则表达式徽标 **::** [empty]
正则表达式徽标 标识部分

NOTE
正则表达式常量不可为空；`//` 开始一个单行注释而非表示空正则表达式。描述空正则表达式，使用 `/(?:)/`。

语义

正则表达式常量代表一个 Object 类型的值。确定该值要两个步骤：首先，包含和的产生式延伸的字符，只收集、不解释地转为两个字符串，分别为 Pattern 和 Flags。然后，用参数 Pattern 和 Flags 两个参数调用 `new RegExp` 构造函数，结果是 *正则表达式常量* 的值。如果调用 `new RegExp` 产生了错误，可能的实现是，作为其判断力，在扫描程序时立即报错，或推迟报错，直到该正则表达式常量在出现执行的轨迹中被求值。

7.9 自动分号插入

空语句，变量语句，表达式语句，`do-while` 语句，`continue` 语句，`break` 语句，`return` 语句，以及 `throw` 语句，这些确定的 ECMAScript 语句必须以分号结束。这些分号可以总是明确地出现在源代码文本中。为方便起见，在特定的情况下，源代码文本中的这些分号可以被省略。在下面描述的这些情况中，分号被自动插入源代码托肯流。

7.9.1 自动分号插入的规则

- 从左到右解析程序时，若遇到不被任何产生式允许的托肯（被称为 *违规托肯*），于是，在下列情况一个或多个为真的违规托肯之前自动插入一个分号：
    1. 该违规托肯与前一个托肯之间以至少一个*行结束符*分隔开。
    2. 该违规托肯是 `}`。

- 从左到右解析程序时，若遇到输入托肯流的结尾，且解析器无法把此输入的托肯流解析为单个完整的 ECMAScript，*程序*，于是在这个输入流的结束处自动插入一个分号。
- 从左到右解析程序时，若遇到的托肯被一些文法的产生式允许，但该产生式是*非严格产生式*，且此托肯是终结符或非终结符的第一个托肯，此终结符或非终结符后紧跟着的是非严格产生式中的记法"[no *行结束 符* here]"（因此这样的托肯被称为非严格托肯）。当非严格托肯与前一个托肯之间以至少一个*行结束符*分 隔开时，在此非严格托肯前自动插入一个分号。

不过，有一种附加的情况凌驾于优先规则：如果此分号将被解析为空语句，或那个分号将成为一个 `for` 语句头中的两个分号之一，决不自动插入分号（参见 12.6.3）。

NOTE
这些是文法中仅有的非严格产生式。

后缀表达式：
左侧表达式 [no *LineTerminator* here] **++**
左侧表达式 [no *行结束符* here] **--**

Continue 语句：
continue [no *行结束符* here] *标识符*$_{opt}$ **;**

Break 语句：
break [no *行结 束符* here] *标识符*$_{opt}$ **;**

Return 语句：
return [no *行 结束符* here] *表达式*$_{opt}$ **;**

Throw 语句：
throw [no *行结 束符* here] *表达式* **;**

这些非严格产生式的实际效果如下所示：

- 若遇到托肯 **++** 或 **--**，解析器将视其为一个后缀运算符时，且在提前托肯与托肯 **++** 或 **--** 之间有至少一个*行结束符*，则在托肯 **++** 或 **--** 前自动插入一个分号。
- 若遇到托肯 **continue, break, return,** 或 **throw**，且在下一个托肯之前遇到一个*行 结束符*，在 **continue, break, return,** 或 **throw** 之后自动插入一个分号。

最后给 ECMAScript 程序员一些忠告：

- **++** 或 **--** 应和其操作数出现在同一行。
- `return` 或 `throw` 语句中的*表达式*应和托肯 `return` 或 `throw` 出现在同一行。
- `break` 或 `continue` 语句中的标签应和托肯 `break` 或 `continue` 出现在同一行。

7.9.2 自动分号插入的例子

源代码

```
{ 1 2 } 3
```

即使应用自动分号插入规则，它也不是 ECMAScript 文法中的合法句子。作为对比，源代码

```
{ 1

2 } 3
```

同样不是 ECMAScript 中的合法句子，却会被自动分号插入变形为下面的形式：

```
{ 1

;2 ;} 3;
```

这就是一个合法的 ECMAScript 句子了。

源代码

```
for (a; b

)
```

不是合法的 ECMAScript 句子，且因 `for` 语句头需要分号，句子无法被自动分号插入调整。自动分号插入决不插入 `for` 语句头中的两个分号之一。

源代码

```
return



a + b
```

被自动分号插入变形为下面的形式：

```
return;

a + b;
```

NOTE
这里的表达式 `a + b` 不被作为 `return` 语句的值返回，因为'*行结束符*'把它和托肯 `return` 分隔开了。

源代码

```
a = b

++c
```

被自动分号插入变形为下面的形式：

```
a = b;
```

```
++c;
```

NOTE

这里的托肯 ++ 不被视为应用于变量 b 的后缀运算符，因为在 b 和 ++ 之间存在一个'*行 结束符*'。

源代码

```
if (a > b)
```

```
else c = d
```

不是合法的 ECMAScript 句子，且即使没有文法产生式引用于此，在 else 托肯之前的句子无法被自动分号插入调整，因为被自动插入的分号将被解析为一个空语句。

源代码

```
a = b + c
```

```
(d + e).print()
```

*不*被自动分号插入变形，因为第二行开始的括号表达式可以解释为函数调用的参数列表：

```
a = b +c(d + e).print()
```

在这种状况下，赋值表达式必须以一个左括号为开始。在提前的语句结尾处提供一个清晰的分号而不依赖于自动分号插入，对于程序员来说是个 好想法。

8 类型

值是九种类型之一的实体。这九种类型是：未定义，空值，布尔值，字符串，数值，对象，引用，列表和完结。类型引用，列表和完结的值仅被用于表达式求值的中间值，且不能被对象的属性存储。

8.1 未定义类型

未定义类型仅有一个值，称为 **undefined**。未必赋值的变量的值即 **undefined**。

8.2 空值类型

空值类型仅有一个值，称为 **null**。

8.3 布尔值类型

布尔值类型表示逻辑实体，有两个值，称为 **true** 和 **false**。

8.4 字符串类型

字符串类型是所有有限的零个或多个 16 位无符号整数值（“元素”）的有序序列。在运行的 ECMAScript 程序中，字符串类型常被用于表示文本数据，此时字符串中的每个元素都被视为一个代码点（参看章节 6）。 每个元素都被认为占有此序列中的一个位置。用非负数值索引这些位置。任何时候，第一个元素在位置 0，下一个元素在位置 1，依此类推。字符串的长度即其中元 素的个数（比如，16 位值）。空字符串长度为零，因而不包含任何元素。

若一个字符串包含实际的文本数据，每个元素都被认为是一个单独的 UTF-16 单元。无论这是不是 `String` 实际的存储格式，`String` 中的字符都被当作表示为 UTF-16 来计数。除非特别声明，作用在字符串上的所有操作都视它们为无差别的 16 位无符号整数；这些操作不保证结果字符串仍为常规化的形式，也不保证语言敏感结 果。

NOTE

这些决议背后的原理是尽可能地保持字符串的实现简单而高效。这意味着，在运行中的程序读到从外部进入执行环境的文本数据（即，用户输入，从文件读取文本， 或从网络上接收文本，等等）之前，它们已被转为 Unicode 常规化形式 C。通常情况下，这个转化在进入的文本被从其原始字符编码转为 Unicode 的同时进行（且强制去除头部附加信息）。因此，建议 ECMAScript 程序源代码为常规化形式 C，（如果保证源代码文本是常规化的）保证字符串常量是常规化的，即便它们不包含任何 Unicode 转义序列。

8.5 数值类型

精确地，数值类型拥有 18437736874454810627（即，$2^{64}-2^{53}+3$）个值，表示为 IEEE-754 格式 64 位双精度数值（IEEE 二进制浮点数算术中描述了它），除了 IEEE 标准中的 9007199254740990（即，$2^{53}-2$）个明显的“非数字”值；在 ECMAScript 中，它们被表示为一个单独的特殊值：**NaN**。（请注意，**NaN** 值由程序表达式 `NaN` 产生，并假设执行程序不能调整定义的全局变量 `NaN`。）在某些实现中，外部代码也许有能力探测出众多非数字值之间的不同，但此类行为依赖于具体实现；对于 ECMAScript 代码而言，NaN 值相互之间无法区别。

还有另外两个特殊值，称为**正无穷**和**负无穷**。为简洁起 见，在说明目的时，用符号 +∞ 和 -∞ 分别代表它们。（请注意，两个无限数值由程序表达式 `+Infinity`（简作 `Infinity`）和 `-Infinity` 产生，并假设执行程序不能调整定义的全局变量 `Infinity`。）

另外 18437736874454810624（即，$2^{64}-2^{53}$）个值被称为有些数值。其中的一半是正数，另一半是负数，对于每个正数而言，都有一个与之对应的、相同规模的负数。

请注意，还有一个**正零**和**负数**。为简洁起见，类似地，在 说明目的时，分别用用符号 **+0** 和 **-0** 代表这些值。（请注意，这两个数字零由程序表达式 `+0`（简作 `0`）和 `-0` 产生。）

这 18437736874454810622（即，$2^{64}-2^{53}-2$）个有限非零值分为两种：

其中 18428729675200069632（即，$2^{64}-2^{54}$）个是常规值，形如

$s * m * 2^e$

这里的 *s* 是+1 或-1，*m* 是一个小于 2^{53} 但 不小于 2^{52} 的正整数，e 是一个闭区间 -1074 到 971 中的整数。

剩下的 9007199254740990（即，$2^{53}-2$）个值是非常规的，形如

$s * m * 2^e$

这里的 *s* 是+1 或-1，*m* 是一个小于 2^{52} 的 正整数，e 为 -1074

请注意，所有规模不超过 2^{53} 的正整数和负整数都可被数值类型表示（不过，整数 0 有两个呈现形式，`+0` 和 `0`）。

如果一个有限的数值非零且用来表达它（上文两种形式之一）的整数 *m* 是奇数，则该数值有 *奇数标记(odd significand)*。否则，它有 *偶数标记(even significand)*。

在本规范中，当 *x* 表示一个精确的非零实数数学量（甚至可以是无理数，比如 π）时，短语*"the number value for x"*意为，以下面的方式选择一个数字 值。考虑数值类型的所有有限值的集合（不包括 **-0** 和两个被加入在数值类型中但不可呈现的值，即 2^{1024}（那 是 $+1 * 2^{53} * 2^{971}$）和 -2^{1024}（那是 $-1 * 2^{53} * 2^{971}$）。选择此集合 中值最接近 *x* 的一员，若集合中的两值近似相等，那么选择有偶数标记的那个；为此，2^{1024} 和 -2^{1024} 这两个超额值被认为有偶数标记。最终，若选择 2^{1024}， 用+∞替换它；若选择-2^{1024}， 用-∞替换它；若选择+0，有且只有 *x* 小于零时，用-0 替换它；其它任何被选取的值都不用改变。结果就是 *x* 的数字值。（此过程正是 IEEE-

754"round to nearest"模式对应的行为。）

某些 ECMAScript 运算符仅涉及闭区间-2^{31} 到 $2^{31}-1$ 的整数，或闭区间 0 到 $2^{32}-1$。这些运算符接受任何数值类型的值，不过，数值首先被转换为 2^{32} 整 数值中的一个。参见 ToInt32 和 ToUint32 的 描述，分别在章节 9.5 和 9.6 中。

8.6 对象类型

对象是未排序是属性容器。每个属性由名字、值和一个特征的集合组成。

8.6.1 属性的特征

属性可以拥有下表中零个或多个特征。

特征	描述
ReadOnly	该属性是一个只读属性。使用 ECMAScript 代码写入该属性的企图将被忽略。（不过，要注意的是，在某些情况下，由于宿主环境做出的动作，有 ReadOnly 特征的属性可能会随着时间的推移改变；"ReadOnly" 不代表“一成不变”！）
DontEnum	该属性不被 **`for-in`** 枚举。（章节 12.6.4）
DontDelete	删除该属性的企图将被忽略。参见章节 11.4.1 所描述的 **`delete`** 运算符。
Internal	内部属性没有名字，且不可通过属性访问运算符直接访问。如何访问这些属性取决于实现特性。在何时何地使用这些属性由语言 规范明确指出。

8.6.2 内部属性和方法

内部属性和方法不是该语言的一部分。本规范定义它们纯粹是为了说明要旨。ECMAScript 实现的产生和操作内部属性的行为必须是 这里所描述的方法。对于本文档中的要旨，内部属性的名字以双方括号 [[]] 括起。若一个算法使用对象的内部属性，但该对象未实现此指明的内部属性，抛出 **TypeError** 异常。

对于常规（非内部）属性有两类访问操作：*get* 和 *put*， 分别用来取值和赋值。

本地 ECMAScript 对象有一个称为[[Prototype]]的 内部属性。该属性的值是 `null` 或某个用于实现继承的对象。[[Prototype]]对象的属性对于其子对象是可见的，但仅是 get 访问而非 put 访问。

下面的表格总结了本规范中所用的内部属性。这个描述指明了本地 ECMAScript 对象的行为。宿主对象可以根据任何依赖实现的行为实现这些内部方法，也可只实现其中的一部分。

属性	变参	描述
[[Prototype]]	无	该对象的原型。

属性	变参	描述
[[Class]]	无	指示这种对象的字符串值。
[[Value]]	无	与这个对象相关联的内部状态信息。
[[Get]]	(属性名)	返回该属性的值。
[[Put]]	(属性名, *值*)	把这个特定属性设为 *值*。
[[CanPut]]	(属性名)	返回一个布尔值，指示 [[Put]] 操作 *属性名* 是否会成功。
[[HasProperty]]	(属性名)	返回一个布尔值，指示该对象是否有给定名字的属性。
[[Delete]]	(属性名)	从该对象中删除指定属性。
[[DefaultValue]]	(暗示)	返回该对象的默认值，这应当是一个原语值（不是对象或引用）。
[[Construct]]	由调用者提供的参数值列表	构造一个对象。 通过 **new** 运算符启用。 实现这个内部方法的对象被称为 *构造函数*。
[[Call]]	由调用者提供的参数值列表	执行与此对象相关联的代码。通过函数调用表达式启用它。实现这个内部方法的对象被称为 *函数*。
[[HasInstance]]	(值)	返回一个布尔值，指示 *值* 是否委派此对象的行为。对于本地 ECMAScript 对象来说，只有 Function 对象实现 [[HasInstance]].
[[Scope]]	无	Function 对象所在的，定义执行环境的作用域链。
[[Match]]	(字符串, *下标*)	测试正则表达式匹配，返回一个 MatchResult 值（参见章节 15.10.2.1）。

每个对象（包括宿主对象）必须实现[[Prototype]]和[[Class]] 属性，以及[[Get]]、[[Put]]、[[CanPut]]、[[HasProperty]]、[[Delete]]、[[DefaultValue]]方法。（不过，请注意，对于某些对象，[[DefaultValue]]方法可以简单地抛出一个 **TypeError** 异常。）

[[Prototype]]属性的值必 须是一个对象或 **null**，且每个[[Prototype]]链 长度必须有限（也就是说，从一个对象开始，递归地访问[[Prototype]]属 性必须以一个 **null** 值结束。）本地对象是否可以把宿主对象作为其[[Prototype]]，这取决于具体实现。

对于所有种类的内置对象，本规范都定义了[[Class]]属 性的值。宿主对象的[[Class]]属性的值可以是任何值，甚至 可以是一个内置对象用作其[[Class]]属性的值。[[Class]]属性的值在内部被用于区别不同种类的内置对象。需要注意的 是，本规范不给程序提供任何访问该值的方式，除了 `Object.prototype.toString`（参 见 15.2.4.2）。

对于本地对象，[[Get]]、[[Put]]、[[CanPut]]、[[HasProperty]]、[[Delete]]、[[DefaultValue]]方法的行为分别在 8.6.2.1 8.6.2.2 8.6.2.3 8.6.2.4 8.6.2.5 8.6.2.6 中描述，除了 Array 对象。它们在[[Put]]方法的实现上有稍许不同（参见 15.4.5.1）。 宿主对象可以以除特定情况之外的任何方式

实现这些方法；例如，一种可能是，对于特定的宿主对象，[[Get]]和[[Put]]可以获取和存储属性值，但[[HasProperty]]总是产生 **false**。

在下面的算法描述中，假定 *O* 是一个本地 ECMAScript 对象，*P* 是一个字符串。

8.6.2.1 [[Get]](P)

若 *O* 的[[Get]]方 法以属性名 *P* 调用，采取下面的步骤：

1. 如果 *O* 没有以 *P* 命名的属性，转到步 骤 4.
2. 获取属性的值。
3. 返回 Result(2)。
4. 如果 *O* 的[[Prototype]]为 **null**，返回 **undefined**。
5. 以属性名 *P* 调用[[Prototype]]的[[Get]]方法。
6. 返回 Result(5)。

8.6.22 [[Put]](P,V)

若以属性名 *P* 和值 *V* 调用 *O* 的[[Put]]方法，采取下面的步 骤：

1. 以名字 *P* 调用 *O* 的[[CanPut]]方法。
2. 如果 Result(1) 为 **false**，返回。
3. 如果 *O* 没有以 *P* 为 名的属性，转到步骤 6.
4. 将该属性的值设为 *V* 。不改变该属性的特征。
5. 返回。
6. 创建以 *P* 为名属性，将其值设为 *V* ， 并给予它空特征。
7. 返回。

不过，值得注意的是，假如 *O* 是一个 Array 对象，关于[[Put]]方法有更多详细叙述（ 15.4.5.1 ）。

8.6.2.3 [[CanPut]](P)

[[CanPut]]方法仅被方法[[Put]]使用。

若以属性名 *P* 调用 *O* 的[[CanPut]]方 法，采取下面的步骤：

1. 如果 *O* 没有以 *P* 命 名的属性，转到步骤 4.
2. 如果该属性有 ReadOnly 特征，返回 **false**。
3. 返回 **true**。
4. 如果 *O* 的[[Prototype]]为 **null**，返回 **true**。
5. 以属性名 *P* 调用 *O* 的[[Prototype]]的[[CanPut]]方法。
6. 返回 Result(5)。

8.6.2.4 [[HasProperty]](P)

若以属性名 *P* 调用 *O* 的[[HasProperty]]方法，采取下面的步骤：

1. 如果 *O* 有以 *P* 命 名的属性，返回 **true**。
2. 如果 *O* 的[[Prototype]]为 **null**，返回 **false**。
3. 以属性名 *P* 调用[[Prototype]]的[[HasProperty]]方法。
4. 返回 Result(3)。

8.6.2.5 [[Delete]](P)

若以属性名 *P* 调用 *O* 的[[Delete]]方法，采取下面的步骤：

1. 如果 *O* 没有以 *P* 命 名的属性，返回 **true**。
2. 如果该属性有 DontDelete 特征，返回 **false**。
3. 从 *O* 中删除以 *P* 命 名的属性。
4. 返回 **true**。

8.6.2.6 [[DefaultValue]](hint)

若以暗示字符串调用 *O* 的[[DefaultValue]]方 法，采取下面的步骤：

1. 以参数 "toString" 调用对象 *O* 的[[Get]]方法。
2. 如果 Result(1) 不是对象，转到步骤 5。
3. 调用 Result(1) 的[[Get]]方 法，使用空参数列表，把 *O* 作为 **this** 值。
4. 如果 Result(3) 是原语值，返回 Result(3)。
5. 以参数 "valueOf" 调用对象 *O* 的[[Call]]方法。
6. 如果 Result(5) 不是对象，转到步骤 9。
7. 调用 Result(5) 的[[Call]]方 法，使用空参数列表，把 *O* 作为 **this** 值。
8. 如果 Result(7) 是原语值，返回 Result(7)。
9. 抛出 **TypeError** 异常。

若以暗示数值调用 *O* 的[[DefaultValue]]方 法，采取下面的步骤：

1. 以参数 "valueOf" 调用对象 *O* 的[[Get]]方法。
2. 如果 Result(1) 不是对象，转到步骤 5。
3. 调用 Result(1) 的[[Call]]方 法，使用空参数列表，把 *O* 作为 **this** 值。
4. 如果 Result(3) 是原语值，返回 Result(3)。
5. 以参数 "toString" 调用对象 *O* 的[[Get]]方法。
6. 如果 Result(5) 不是对象，转到步骤 9。
7. 调用 Result(5) 的[[Call]]方 法，使用空参数列表，把 *O* 作为 **this** 值。
8. 如果 Result(7) 是原语值，返回 Result(7)。
9. 抛出 **TypeError** 异常。

若不以暗示调用 *O* 的[[DefaultValue]]方 法，接下来的行相当于于暗示是数值，除非 *O* 是 Date 对象（参见 15.9），在这种情况下，它的行为相当于暗示是字符串。

在上文的规范中，本地对象的[[DefaultValue]]只能返回原语值。如果是实现其自己的[[DefaultValue]]方法宿主的对象，必须保证[[DefaultValue]]只能返回原语值。

8.7 引用类型

内部引用类型不是语言的数据类型。在本规范中定义它是为了说明要旨。 ECMAScript 实现的产生和操作内部属性的行为必须是 这里所描述的方法。而且，类型 **Reference** 的值仅被 用于表达式求值的中间值，且不能被对象的属性存储。

引用类型被用于解释诸如 `delete`、`typeof` 和 赋值运算符等操作。例如，赋值的左侧操作数被期望产生一个引用。作为替代，对于赋值运算符的左侧操作数，赋值的行为被用于解释其整个句法形式的套用分析， 但这对于解释允许函数调用返回引用这个问题有些困难。我们纯粹是为了宿主对象而允许这种可能性。本规范中定义的非内置的 ECMAScript 函数返回一个引用，但这里没有为用户定义函数返回引用作准备。（另一个理由是，不使用句法套用分析将会变得冗长且尴尬，对本规范中许多部分造成影响。）

Reference 是到对象的属性的引用。一个引用由两部分组件组成，*基 对象(base object)*和*属性名(property name)*。

本规范使用下面的抽象操作来访问引用的组件。

- GetBase(*V*)。返回引用 *V* 的基对象组件。
- GetPropertyName(*V*)。返回引用 *V* 的属性名组件。

在本规范中使用这些抽象操作来操作引用。

8.7.1 GetValue(V)

1. 如果 Type(*V*) 不是引用，返回 *V*。
2. 调用 GetBase(*V*)。
3. 如果 Result(2) 是 **null**，抛出 **ReferenceError** 异常。
4. 调用 Result(2) 的[[Get]]方法，传递 GetPropertyName(*V*) 作为属性名。
5. 返回 Result(4)。

8.7.1 GetValue(V)

1. 如果 Type(*V*) 不是引用，抛出 **ReferenceError** 异常。
2. 调用 GetBase(*V*)。
3. 如果 Result(2) 是 **null**，转到步骤 6。

4. 调用 Result(2) 的[[Put]]方法，传递 GetPropertyName(*V*) 作为属性名，*W* 作为值。
5. 返回。
6. 调用全局对象的[[Put]]方法，传递 GetPropertyName(*V*) 作为属性名，*W* 作为值。
7. 返回。

8.8 列表类型

内部列表类型不是语言的数据类型。在本规范中定义它是为了说明要旨。 ECMAScript 实现的产生和操作内部属性的行为必须是 这里所描述的方法。而且，类型 **List** 的值仅被 用于表达式求值的中间值，且不能被对象的属性存储。

列表类型被用于解释参数列表的求值（参见 11.2.4）。列表类的值是简单的已排序值的序列。这些序列可以是任何长度。

8.9 完结类型

内部完结类型不是语言的数据类型。在本规范中定义它是为了说明要旨。 ECMAScript 实现的产生和操作内部属性的行为必须是 这里所描述的方法。而且，类型 Completion 的值仅被用于表达式求值的中间值，且不能被对象的属性存储。

完结类型被用于解释非本地控制调度的语句的行为（`break, continue, return, throw`）， 这里的*类型*是 **normal, break, continue, return, throw** 之一，*值*是一个 ECMAScript 值或 **empty**，*目标*是任何 ECMAScript 标识符或 **empty**。

限制词 "突然完结" 指的是任何除 **normal** 之外有类型的完结。

9 类型转换

ECMAScript 运行时系统会在需要时从事自动类型转换。为了阐明某些结构的语义，定义一集转换运算符是很有用的。这些运算符不是语言的一部分；在这里定义它们是为了协助 语言语义的规范。转换运算符是多态的。也就是说，它们可以接受任何标准类型的值，除了引用，列表，完结类型这些内部类型之外。

9.1 ToPrimitive

ToPrimitive 运算符接受一个值，和一个可选的 *期望类型* 作参数。ToPrimitive 运算符把其值参数转换为非对象类型。如果对象有能力被转换为不止一种原语类型，可以使用可选的 *期 望类型* 来暗示那个类型。根据下表完成转换：

输入类型	结果
未定义	结果等于输入的参数（不转换）。
空值	结果等于输入的参数（不转换）。
布尔值	结果等于输入的参数（不转换）。
数值	结果等于输入的参数（不转换）。
字符串	结果等于输入的参数（不转换）。
对象	返回该对象的默认值。调用该对象的内部方法 [[DefaultValue]] 来恢复这个默认值， 调用时传递暗示 *期望类型*。对于所有 ECAMScript 本地对象，[[DefaultValue]]方法的行为在此规范(8.6.2.6) 中定义。

9.2 ToBoolean

ToBoolean 运算符根据下表将其参数转换为布尔值类型的值：

输入类型	结果
未定义	**false**
空值	**false**
布尔值	结果等于输入的参数（不转换）。
数值	如果参数是 **+0, -0,** 或 NaN，结果为 **false** ；否则结果为 **true.**
字符串	如果参数参数是空字符串（其长度为零），结果为 **false**，否则结果为 **true.**
对象	**true**

9.3 ToNumber

ToNumber 运算符根据下表将其参数转换为数值类型的值：

输入类型	结果
未定义	**NaN**
空值	**+0**
布尔值	如果参数是 **1**，结果为 **true**。 如果参数是**+0**，此结果为 **false**。
数字	结果等于输入的参数（不转换）。
字符串	参见下文的文法和注释。
对象	应用下列步骤： 1. 调用 ToPrimitive(输 入参数, 暗示数值类型). 2. 调用 ToNumber(Result(1)). 3. 返回 Result(2).

9.3.1 对字符串类型应用 ToNumber

对字符串应用 ToNumber 时，对输入字符串应用如下文法。如果此文法无法将字符串解释为*字符串数值常量*的 扩展，那么 ToNumber 的结果为 **NaN**。

字符串数值常量 **:::**
串空白 $_{opt}$
串空白 $_{opt}$ *串数值常量 串空白* $_{opt}$

串空白 **:::**
串空白字符 串空白 $_{opt}$

串空白字符 **:::**
<*TAB*>
<*SP*>
<*NBSP*>
<*FF*>
<*VT*>
<*CR*>
<*LF*>
<*LS*>
<*PS*>
<*USP*>

串数值常量 **:::**
串十进制常量

十六进制整数常量

串十进制常量 **:::**
串无符号十进制常量
+ *串无符号十进制常量*
- *串无符号十进制常量*

串无符号十进制常量 **:::**
Infinity
十进制数 **.** *十进制数* *opt* *指数部分* *opt*
. *十进制数* *指数部分* *opt*
十进制数 *指数部分* *opt*

十进制数 **:::**
十进制数字
十进制数 *十进制数字*

十进制数字 **::: one of**
0 1 2 3 4 5 6 7 8 9

指数部分 **:::**
幂指示符 *有符号整数*

幂指示符 **::: one of**
e E

有符号整数 **:::**
十进制数
+ *十进制数*
- *十进制数*

十六进制整数常量 **:::**
0x *十六进制数字*
0X *十六进制数字*
十六进制整数常量 *十六进制数字*

十六进制数字 **::: one of**
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

需要注意到*字符串数值常量*和*数值常量*语法上的不同（参 见 7.8.3）：

- *字符串数值常量*之前和、或之后可以有空白和、或行结束符。
- 在数字 0 之前有任何数字，则此*字符串数值常量*是十进制的。
- 用 + 或 - 前缀指示其符号的*字符串数值常量*是十进制的。
- 空的，或只包含空白的字符串常量被转换为 **+0**。

字符串到数字值的转换，大体上类似于判定数值常量的数字值（参见 7.8.3），不过有些细节上的不同，所以， 这里给出了把字符串数值常量转换为数值类型的值的全部过程。这个值分两步来判定：首先，从字符串数值常量中导出数学值(MV)；第二步，以下面所描述的方 式对该数学值进行舍入。

- *字符串整数常量*的 MV **:::** [empty] 是 0.
- *字符串整数常量*的 MV **:::** *串 空白* 是 0.
- *字符串整数常量*的 MV **:::** *串 空白* $_{opt}$ *串数值常量 串空白* $_{opt}$ 是 *串数值常量* 的 MV, no matter whether white space 是 present or not.
- *串数值常量*的 MV **:::** *串 十进制常量* 是 *串十进制常量*的 MV.
- *串数值常量*的 MV **:::** *十 六进制整数常量* 是 *十六进制整数常量*的 MV.
- *串十进制常量*的 MV **:::** *串 无符号整数常量* 是 *串无符号整数常量*的 MV.
- *串十进制常量*的 MV **::: +** *串无符号整数常量* 是 *串无符号整数常量*的 MV.
- *串十进制常量*的 MV **::: -** *串无符号整数常量* 是 the negative of *串无符号整数常量*的 负 MV.（需要注意的是，如果 *串无符号整数常量*的 MV 是 0, 其负 MV 也是 0。 下午中描述的舍入规则 会合适地处理小于数学零到浮点数 **+0** 或 **-0** 的变换。）
- *串无符号整数常量*的 MV **:::** **`Infinity`** 是 10^{10000}（过大的值会返回为 +∞）。
- *串无符号整数常量*的 MV **:::** *十 进制数. 是 十进制数*的 MV. *串无符号整数常量*的 MV **:::** *十进制数. 十进制数* 是 the 第一个 *十进制数*的 MV 加 (the 第二个 *十进制数*的 MV 乘以 10^{-n}), 这里的 *n* 是 the number of characters in the 第二个 *十进制数*.
- *串无符号整数常量*的 MV **:::** *十 进制数. 指数部分* 是 *十进制数*的 MV 乘以 10^{e}, 这里的 *e* 是 *指数部分*的 MV.
- *串无符号整数常量*的 MV **:::** *十 进制数. 十进制数 指数部分* 是 (the 第一个 *十进制数*的 MV 加 (the 第二个 *十进制数*的 MV 乘以 10^{-n})) 乘以 10^{e}, 这里的 *n* 是 第二个 *十进制数* 中的字符个数， *e* 是 *指 数部分*的 MV.
- *串无符号整数常量*的 MV **:::** . *十进制数* 是 *十进制数*的 MV 乘以 10^{-n}, 这里的 *n* 是 *十进制数* 中的字符个数。
- *串无符号整数常量*的 MV **:::** . *十进制数 指数部分* 是 *十进制数*的 MV 乘以 10^{e-n}, 这里的 *n* 是 *十进制数* 中的字符个数， *e* 是 *指数 部分*的 MV.
- *串无符号整数常量*的 MV **:::** *十 进制数* 是 *十进制数*的 MV.
- *串无符号整数常量*的 MV **:::** *十 进制数 指数部分* 是 *十进制数*的 MV 乘以 10^{e}, 这里的 *e* 是 *指数部分*的 MV.
- *十进制数*的 MV **:::** *十 进制数字* 是 *十进制数字*的 MV. *十进制数*的 MV **:::** *十进制数 十进制数字* 是 (*十 进制数* 乘以 10) 加 *十进制数字*的 MV.
- *指数部分*的 MV **:::** *幂 指示符 有符号整数* 是 *有符号整数*的 MV.
- *有符号整数*的 MV **:::** *十 进制数* 是 *十进制数*的 MV.
- *有符号整数*的 MV **::: +** *十进制数* 是 *十进制数*的 MV.
- *有符号整数*的 MV **::: -** *十进制数* 是 *十进制数*的负 MV.
- *十进制数字*的 MV **::: `0`**， 对于 *十六进制数字* **::: `0`** 是 0。

- *十进制数字*的 MV ::: **1**，对于 *十六进制数字* ::: **1** 是 1。
- *十进制数字*的 MV ::: **2**，对于 *十六进制数字* ::: **2** 是 2。
- *十进制数字*的 MV ::: **3**，对于 *十六进制数字* ::: **3** 是 3。
- *十进制数字*的 MV ::: **4**，对于 *十六进制数字* ::: **4** 是 4。
- *十进制数字*的 MV ::: **5**，对于 *十六进制数字* ::: **5** 是 5。
- *十进制数字*的 MV ::: **6**，对于 *十六进制数字* ::: **6** 是 6。
- *十进制数字*的 MV ::: **7**，对于 *十六进制数字* ::: **7** 是 7。
- *十进制数字*的 MV ::: **8**，对于 *十六进制数字* ::: **8** 是 8。
- *十进制数字*的 MV ::: **9**，对于 *十六进制数字* ::: **9** 是 9。
- *十六进制数字*的 MV ::: **a**，对于 *十六进制数字* ::: **A** 是 10。
- *十六进制数字*的 MV ::: **b**，对于 *十六进制数字* ::: **B** 是 11。
- *十六进制数字*的 MV ::: **c**，对于 *十六进制数字* ::: **C** 是 12。
- *十六进制数字*的 MV ::: **d**，对于 *十六进制数字* ::: **D** 是 13。
- *十六进制数字*的 MV ::: **e**，对于 *十六进制数字* ::: **E** 是 14。
- *十六进制数字*的 MV ::: **f**，对于 *十六进制数字* ::: **F** 是 15。
- *十六进制整数常量*的 MV ::: **0x** *十六进制数字* 是 *十六进制数字*的 MV。
- *十六进制整数常量*的 MV ::: **0X** *十六进制数字* 是 *十六进制数字*的 MV。
- *十六进制整数常量*的 MV ::: *十六进制整数常量 十六进制数字* 是 (*十六进制整数常量*的 MV 乘以 16) 加 *十六进制数字*的 MV。

字符串数值常量的精确 MV 被确定之后，接着被舍入为数值类型的值。如果 MV 是 0，那么其舍入值为 +0，除非字符串数值常量中 第一个非空白字符是 '-'——在这种情况下，舍入值为 -0。否则，舍入值必须是 MV 的数字值（章节 8.5 有 定义），除非该常量包括一个*串无符号十进制常量*，且该常量多于 20 位有效数字，在这种情况下，数字值可以是下面两 值之一，

一旦字符串数值常量的 MV 被精确地确定，接下来就会被舍入为数值类型的一个值。如果 MV 是 0，那么舍入值为 **+0**，除非字符串数值常量中 第一个非空白字符是 '-'——在这种情况下，舍入值为 -0。否则，舍入值必须是 MV 的数字值（在 8.5 中定义），除非该常量包括一个*串无符号十进 制常量*，且此常量多于 20 位有效数字——在这种情况下，此数字的值是下 面两种之一：一是将其 20 位之后的每个有效数字用 0 替换，产生此常量的 MV；二是将其 20 位之后的每个有效数字用 0 替换，并在 20 位有效数字之后增加数字位，产生此常量的 MV 值。判断一个数字是否为*有 效数字*，首先它不能是*指数部分*的一部分，且

- 它不是 0；或
- 它的左边是一个非零值，右边是一个不在*指数部分*中的非零值。

9.4 ToInteger

ToInteger 运算符将其参数转换为整数值。此运算符功能如下所示：

1. 对输入参数调用 ToNumber。

2. 如果 Result(1) 是 **NaN**，返回 **+0**。
3. 如果 Result(1) 是 **+0**， **-0**，+∞， 或 -∞，返回 Result(1)。
4. 计算 sign(Result(1)) * floor(abs(Result(1)))。
5. 返回 Result(4)。

9.5 ToInt32：（32 位有符号整数）

ToInt32 运算符将其在-2^{31} 到 2^{31}-1 闭区间内的参数转换为 2^{32} 个整数值之一。此运算符功能如下所示：

1. 对输入参数调用 ToNumber。
2. 如果 Result(1) 是 **+0**， **-0**，+∞， 或 -∞，返回 +0。
3. 计算 sign(Result(1)) * floor(abs(Result(1)))。
4. 计算 Result(3) modulo 2^{32} ；也就是说，数值类型的有限整数值 k 为正，且小于 2^{32} ，规模相对于 Result(3) 的数学值差异， 2^{32} 是 k 的整数倍。
5. 如果 Result(4) 是大于等于 2^{31} 的整数，返回 Result(4)-2^{32}， 否则返回 Result(4)。

9.6 ToUint32：（32 位无符号整数）

ToUint32 运算符将其在 0 到 2^{32}-1 闭区间内的参数转换为 2^{32} 个 整数值之一。此运算符功能如下所示：

1. 对输入参数调用 ToNumber。
2. 如果 Result(1) 是 **+0**， **-0**，+∞， 或 -∞，返回 +0。
3. 计算 sign(Result(1)) * floor(abs(Result(1)))。
4. 计算 Result(3) modulo 2^{32} ；也就是说，数值类型的有限整数值 k 为正，且小于 2^{32} ，规模相对于 Result(3) 的数学值差异， 2^{32} 是 k 的整数倍。
5. 返回 Result(4)。

NOTE
上面给出的 ToUint32 的定义中：
ToUint32 和 ToInt32 唯 一的不同在于第 5 步。
ToUint32 的操作具有鉴一性：如果应用于一个已经产生的结果，第二次应用保持值不变。
对于 x 的所有值，ToUint32(ToInt32(x)) 与 ToUint32 相等。（这是为了保证后来的属性 +∞ 和 -∞ 被映射为 +0。）
ToUint32 把 -0 映射为 +0。

9.6 ToUint16：（16 位无符号整数）

ToUint32 运算符将其在 0 到 2^{16}-1 闭区间内的参数转换为 2^{16} 个 整数值之一。此运算符功能如下所示：

1. 对输入参数调用 ToNumber。
2. 如果 Result(1) 是 **+0**， **-0**，+∞， 或 -∞，返回 +0。
3. 计算 sign(Result(1)) * floor(abs(Result(1)))。
4. 计算 Result(3) modulo 2^{16} ；也就是说，数值类型的有限整数值 k 为正，且小于 2^{16} ，规模相对于 Result(3) 的数学值差异， 2^{16} 是 k 的整数倍。
5. 返回 Result(4)。

NOTE

上面给出的 ToUint32 的定义中：

ToUint32 ToUint16 之间唯一的不同是第 4 步中，2^{16} 代替了 2^{32}。

ToUint32 把 -0 映射为 +0。

9.8 ToString

ToString 运算符根据下表将其参数转换为字符串类型的值：

输入类型	结果
未定义	**"undefined"**
空值	**"null"**
布尔值	如果参数是 **true**，那么结果为 **"true"**。 如果参数是 **false**，那么结果为 **"false"**。
数值	参见下面的注释。
字符串	返回输入的参数（不转换）。
对象	应用下列步骤： 1. 调用 ToPrimitive(输 入参数, 暗示字符串类型). 2. 调用 ToString(Result(1))。 3. 返回 Result(2)。

9.8.1 对数值类型应用 ToString

ToString 运算符将数字 *m* 转换为字符串格式的给出如下所示：

1. 如果 *m* 是 **NaN**，返回字符串 **"NaN"**。
2. 如果 *m* 是 **+0** 或 **-0**， 返 回字符串 **"0"**。
3. 如果 *m* 小于零，返回连接 **"-"** 和 ToString(-*m*) 的字符串。
4. 如果 *m* 无限大，返回字符串 **"Infinity"**。
5. 否则，令 *n, k,* 和 *s* 是整数，使得 *k* >= 1, 10^{k-1} <= *s* <10^{k}，*s* * 10^{n-k} 的数字值是 *m*，且 *k* 足够小。要注意的是，*k* 是 *s* 在十进制表示中的数字的个数。*s* 不被 10 整除，且 *s* 的至少要求的有效数字位数不一定要被这些标准唯一确定。

6. 如果 *k* <= *n* <= 21，返回由 *k* 个 *s* 在十进制表示中的数字组成的字符串（有序的，开头没有零），后面跟随字符 '0' 的 *n* *k* 次出现。
7. 如果 0 < n <= 21，返回由 *s* 在十进制表示中的、最多 *n* 个有效数字组成的字符串，后面跟随一个小数点 '.'，再后面是余下的 *k*-*n*& nbsp;个 *s* 在十进制表示中的数字。
8. 如果 -6 <n <= 0，返回由字符 '0' 组成的字符串，后面跟随一个小数点 '.'，再后面是字符 '0' 的 -*n* 次出现，再往后是 *k* 个 *s* 在十进制表示中的数字。
9. 否则，如果 *k* = 1，返回由单个数字 *s* 组成的字符串，后面跟随小写字母 'e'，根据 *n*-1 是正或负，再后面是一个加号 '+' 或减号 '-'，再往后是整数 abs(*n*-1) 的十进制表示（没有前置的零）。
10. 返回由 *s* 在十进制表示中的、最多的有效数字组成的字符串，后面跟随一个小数点 '.'，再后面是余下的是 *k*-1 个 *s* 在十进制表示中的数字，再往后是小写字母 'e'，根据 *n*-1 是正或负，再后面是一个加号 '+' 或减号 '-'，再往后是整数 abs(*n*-1) 的十进制表示（没有前置的零）。

NOTE

下面的评语可能对指导实现有用，但不是本标准的常规要求。

如果 x 是除 **-0** 以外的任一数字值，那么 ToNumber(ToString(x)) 与 x 是完全相同的数字值。

s 至少要求的有效数字位数并非总是由步骤 5 中所列的要求唯一确定。

对于那些提供了比上面的规则所要求的更精确的转换的实现，我们推荐下面这个步骤 5 的可选版本，作为指导：

否则，令 *n*, *k*, 和 *s* 是整数，使得 *k* >= 1, 10^{k-1} <= *s* <10^{k}，*s* * 10^{n-k} 的数字值是 *m*，且 *k* 足够小。如果有数倍于 *s* 的可能性，选择 s * 10^{n-k} 最接近于 m 的值作为 *s* 的值。如果 *s* 有两个这样可能的值，选择是偶数的那个。要注意的 是，*k* 是 *s* 在十进制表示中的数字的个数，且 *s* 不被 10 整除。

ECMAScript 的实现者们可能会发现，David M 所写的关于浮点数进行二进制到十进制转换方面的文章和代码很有用：

Gay, David M. Correctly Rounded Binary-Decimal and Decimal-Binary Conversions. Numerical Analysis Manuscript 90-10. AT&T Bell Laboratories (Murray Hill, New Jersey). November 30, 1990. 在这里取得 http://cm.bell-labs.com/cm/cs/doc/90/4-10.ps.gz。 有关的代码在这里 http://cm.bell-labs.com/netlib/fp/dtoa.c.gz 还有 http://cm.bell-labs.com/netlib/fp/g_fmt.c.gz。这些都可在众多的 netlib 镜像站点上找到。

9.8 ToObject

ToObject 运算符根据下表将其参数转换为对象类型的值：

输入类型	结果
未定义	抛出 **TypeError** 异常。
空值	抛出 **TypeError** 异常。

输入类型	结果
布尔值	创建一个新的 Boolean 对象，其 [[value]] 属性被设为该布尔值的值。参见 15.6 中 Boolean 对象的描述。
数值	创建一个新的 Number 对象，其 [[value]] 属性被设为该布尔值的值。参见 15.7 中 Number 对象的描述。
字符串	创建一个新的 String 对象，其 [[value]] 属性被设为该布尔值的值。参见 15.8 中 String 对象的描述。
对象	结果是输入的参数（不转换）。

10 执行上下文

当控制器转换到 ECMAScript 可执行代码时，控制器就进入了*执行上下文*。活动的执行上下文组成一个逻辑上的栈。逻辑栈上的顶层执行上下文即正在运行的执行上下文。

10.1 定义

10.1.1 函数对象

有两种类型的函数对象：

- 在源代码文本中用*函数声明*定义的，和使用*函数表达式*或使用函数对象作为构造器动态创建的程序中的函数。
- 内置函数，它们是语言的内建对象，例如 `parseInt` 和 `Math.exp`。实现也可能会提供独立于实现且在此规范中未被描述的内置函数。这些函数不包含 ECMAScript 语法所定义的可执行代码，所以它们不在讨论范围之内。

10.1.2 可执行代码的类型

ECMAScript 的可执行代码有三种类型：

- *全局代码*，被视为 ECMAScript *程序*的源代码文本。对于一个*程序*而言，其全局代码不包含可被解析为*函数体*的一部分的源代码文本。
- *求值代码*，被提供给内置函数 `eval` 的源代码文本。更确切地说，如果给内置函数 `eval` 的参数是字符串，它就被视为一个为 ECMAScript *程序*。对于一段用于调用 `eval` 的求值代码而言，它是这个字符串参数的全局代码。
- *函数代码*，被解析为*函数体*的一部分的源代码文本。对于一个*函数体*的函数而言，其*代码*不包含任何一段可被解析为一个嵌套的*函数体*的一部分的源代码文本。当使用函数对象作为构造器时，提供的源代码文本也可以用*函数代码*表示。更确切地说，最后一个提供给函数构造器的参数被转换为一个字符串，将其视为*函数体*。如果给函数构造器给出了多于一个的参数，那么，除最后一个参数之外，所有的参数被转换为字符串并连接（以逗号分隔）。结果字符串被解释为最后一个参数所定义的*函数体*的*形式参数列表*。对于一个*函数*的实例而言，其*函数代码*不包含任何一段可被解析为一个嵌套的*函数体*的一部分的源代码文本。

10.1.3 变量实例化

所有的执行上下文都与一个可变对象相关联，在源代码文本中声明的变量和函数都被作为属性添加给这个可变对象。对于函数代码，参数被作为属性添加给这个可变对象。

被当作这个可变对象使用的对象和这些属性所要使用的特征取决于代码的类型，不过该行为的其余部分是泛用的。进入执行上下文时，按照下面的顺序把属性绑定到可变对象：

- 对于函数代码：对每个*形式参数列表*中所定义的形式参数创建一个该可变对象的属性，其名称为相应的*标识符*，特征由代码类型决定，参数的值在调用者[[Call]]时由实参给出。如果调用者给出了个数少于已有形式参数的参数值，多余的形式参数的值即为`undefined`。如果有两个或更多形式参数重名（会使用同一属性），最后一个使用此名称的参数给对应的属性提供值。若最后一个参数未被调用者给出，对应的属性的值即为`undefined`。
- 对于代码中的每个*函数声明*，按照源代码文本的顺序创建该可变对象的属性，其名称为该*函数声明*中的*标识符*，值为创建一个函数对象的返回结果（在第 13 章中被描述），特征由代码类型决定。如果这个可变对象已经拥有了与此同名的属性，替换这个属性的值和特征。从语义上来看，这个步骤必须紧随*形式参数列表*属性的创建。
- 对于代码中的每个*变量声明*或*变量声明（不在 In 中）*，创建该可变对象的属性，其名称为*变量声明*或*变量声明（不在 In 中）*中的*标识符*，值为`undefined`，特征由代码类型决定。如果这个可变对象已经拥有了与被声明变量的同名的属性，该属性的值和特征不变。从语义上来看，这个步骤必须紧随*形式参数列表*和*函数声明*属性的创建。特别要注意到是，如果被声明的变量与已声明的函数或形式参数同名，变量声明并不干扰已经存在的属性。

10.1.4 作用域链和标识符判定

每一个执行上下文都与一个作用域链相关联。作用域链是一个对象组成的链表，求值 *标识符* 的时候会搜索它。当控制进入执行上下文时，就根据代码类型创建一个作用域链，并用初始化对象填充。执行一个上下文的时候，其作用域链只会被`with`声明（见 12.10）和`catch`语句（见 12.14）所影响。

执行过程中，使用下面的算法求值语义产生式 *初级表达式 : 标识符*:

1. 获取作用域链中的下一个对象。如果没有，转到步骤 5。
2. 调用 Result(1) 的 [[HasProperty]] 方法，把 *标识符* 作为属性名传递。
3. 如果 Result(2) 为 **true**，返回一个引用类型的值，其基对象是 Result(1)，属性名为 *标识符*。
4. 转到步骤 1。
5. 返回引用类型的值，基对象为`null`，属性名为 *标识符*。
6.

求值标识符的结果总是一个引用类型的值，其成员名字组件与标识符字符串相等。

10.1.5 全局对象

全局对象比较特别，它在控制进入任何执行上下文之前被创建。初始化后的全局对象拥有

如下属性：

- 内置对象，如 Math，String，Date，parseInt 等等。它们拥有 { DontEnum } 特征。
- 宿主所定义的附加属性。这可能会包括一个值为全局对象自身的属性，例如在 HTML 文档模型中，全局对象的 `window` 属性就是全局对象本身。

在控制进入执行上下文执行 ECMAScript 之后，全局变量就可能会被附加属性或改变初始属性。

10.1.6 活动对象

当控制进入函数代码的执行上下文时，创建一个活动对象并将它与该执行上下文相关联，并使用一个名为 `arguments`、特征为 { DontDelete } 的属性初始化该对象。该属性的初始值是稍后将要描述的一个参数对象。

接下来，这个活动对象将被用作变量初始化的可变对象。

活动对象纯粹是一种规范性机制，在 ECMAScript 访问它是不可能的。只能访问其成员而非该活动对象本身。对一个基对象为活动对象的引用值应用调用运算符时，这次调用的 **this** 值为 `null`。

10.1.7 This

对于每个执行上下文，都有一个 **this** 值与其相关联。在控制进入执行上下文时，根据调用者和被执行代码的类型决定这个值。与执行上下文相关联的 **this** 值是非可变的。

10.1.8 参数对象

当控制进入函数代码的执行上下文时，创建一个参数对象，初始化之如下：

- 参数对象内部属性[[Prototype]]的值是原始的 Object 原型对象，即 `Object.prototype` 的初始值（见 15.2.3.1）。
- 创建名为 `callee` 的属性，特征是 { DontEnum }，初始值为被执行的函数对象。这样就允许了匿名函数的递归。
- 创建名为 `length` 的属性，特征是 { DontEnum }，初始值为提供给调用者的实参值的个数。
- 对每个小于 `length` 属性的非负整数 *arg*，创建名为 ToString(*arg*) 的属性，特征是 { DontEnum }，初始值为对应的提供给调用者的实际参数。第一个实参值对应 *arg* = 0，第二个对应 *arg* = 1，依此类推。对于 *arg* 小于函数对象的形参个数的情况，该属性与对应活动对象的属性同值。这就意味着改变这个属性也会改变对应活动对象的属性，反之亦然。

10.2 进入执行上下文

每个函数和构造器调用都要进入一个新的执行上下文，即使是函数在递归地调用自身。每次返回都会退出执行上下文。未被 catch 的异常抛出有可能退出一个或多个执行上下文。

当控制进入执行上下文时，创建并初始化作用域链，进行变量初始化，并决定 **this** 值。

作用域链的初始化，变量的初始化和 **this** 值的决定取决于进入的代码类型。

10.2.1 全局代码

- 被创建并初始化的作用域链只包含全局代码。
- 进行变量初始化时，把全局对象作为可变对象，属性特征为 { DontDelete }。
- **this** 值为全局对象。

10.2.2 求值代码

当控制进入求值代码的执行上下文时，把前一个活动的执行上下文引用为*调用上下文*，用它决定作用域链、可变对象和 **this** 值。若调用上下文不存在，就把它当作全局对象，进行作用域链和变量的初始化及 **this** 值的决定。

- 被创建并初始化的作用域链与调用上下文包含相同的对象，顺序也一样；使用 `with` 声明或 `catch` 语句给调用上下文的作用域链添加的对象也包括在内。
- 进行变量初始化时，使用调用上下文的可变对象，属性特征为空。
- **this** 值与调用上下文的 **this** 值相同。

10.2.3 函数代码

- 被创建并初始化的作用域链包含一个活动对象，该对象之后是函数对象的[[Scope]]属性存储的作用域链中的对象。
- 进行变量初始化时，把该活动对象作为可变对象，属性特征为 { DontDelete }。
- **this** 值由调用者提供。若调用者提供的 **this** 值不是一个对象（注意，`null` 不是对象），则 **this** 值为全局对象。