

ECMA-262 中文版

摘自：<http://www.ecmascript.cn>

目录

ECMA-262 中文版	1
摘自： http://www.ecmascript.cn	1
介绍	16
范围	16
一致性	17
参考文献	17
概述	17
Web 脚本语言	18
语言概述	19
对象	19
ECMAScript 的严格模式变体	20
术语定义	21
类型 (type)	21
原始值 (primitive value)	21
对象 (object)	21
构造器 (constructor)	21
原型 (prototype)	21

原生对象 (native object).....	21
内置对象 (built-in object)	22
宿主对象 (host object)	22
未定义值 (undefined value)	22
未定义类型 (Undefined type).....	22
空值 (null value)	22
空类型 (Null type)	22
布尔值 (Boolean value)	22
布尔类型 (Boolean type)	23
布尔对象 (Boolean object).....	23
字符串值 (String value).....	23
字符串类型 (String type)	23
字符串对象 (String object).....	23
数字值 (Number value)	23
数字类型 (Number type)	24
数字对象 (Number object)	24
无穷 (Infinity)	24
NaN	24
函数 (function).....	24
内置函数 (built-in function).....	24
属性 (property).....	24
方法 (method).....	25

内置方法 (built-in method)	25
特性 (attribute)	25
自身属性 (own property)	25
继承属性 (inherited property)	25
记法约定	25
语法和词法的文法	25
上下文无关文法	25
词法和正则的文法	26
数字字符串文法	26
语作文法	26
JSON 文法	27
文法标记法	27
算法约定	31
源代码文本	32
词法	34
Unicode 格式控制字符	35
空白字符	35
行终结符	36
注释	37
Tokens	38
标识符名和标识符	38
保留字	40

标点符号.....	41
字面量.....	42
空值字面量.....	42
布尔值字面量.....	42
数值字面量.....	43
字符串字面量.....	45
正则表达式字面量.....	48
自动分号插入.....	50
自动分号插入规则.....	50
自动分号插入的例子.....	52
类型	53
Undefined 类型.....	54
Null 类型	54
Boolean 类型	54
String 类型	54
Number 类型	55
Object 类型	56
Property 特性	56
Object 内部属性及方法.....	58
引用规范类型.....	62
GetValue(v)	62
PutValue(v,w).....	63

列表规范类型.....	64
完结规范类型.....	64
属性描述符及属性标识符规范类型.....	64
IsAccessorDescriptor (Desc)	65
IsDataDescriptor (Desc)	65
IsGenericDescriptor (Desc)	66
FromPropertyDescriptor (Desc)	66
ToPropertyDescriptor (Obj).....	66
词法环境和环境记录项规范类型.....	67
对象内部方法的算法.....	67
[[GetOwnProperty]](P)	68
[[GetProperty]] (P)	68
[[Get]] (P)	68
[[CanPut]] (P)	69
[[Put]] (P, V, Throw)	69
[[HasProperty]] (P)	70
[[Delete]] (P, Throw)	70
[[DefaultValue]] (hint)	70
[[DefineOwnProperty]] (P, Desc, Throw).....	71
类型转换与测试.....	73
ToPrimitive	73
ToBoolean	73

ToNumber	74
对字符串类型应用 ToNumber	74
ToInteger	77
ToInt32 : (32 位有符号整数)	78
ToUint32 : (32 位无符号整数)	78
ToUint16 : (16 位无符号整数)	79
ToString	79
对数值类型应用 ToString	79
ToObject	81
CheckObjectCoercible	81
IsCallable	81
SameValue 算法	82
可执行代码与执行环境	82
可执行代码类型	82
严格模式下的代码	83
词法环境	83
环境记录项	84
词法环境的运算	89
全局环境	90
执行环境	91
标识符解析	91
建立执行环境	92

进入全局代码.....	92
进入 eval 代码.....	92
进入函数代码.....	93
定义绑定初始化.....	93
Arguments 对象	95
表达式	99
主值表达式.....	99
this 关键字	99
标识符引用.....	99
字面量引用.....	100
数组初始化.....	100
对象初始化.....	102
分组表达式.....	105
左值表达式.....	105
属性访问.....	106
new 运算符	107
函数调用.....	107
参数列表.....	108
函数表达式.....	109
后缀表达式.....	109
后缀自增运算符.....	109
后缀自减运算符.....	109

一元运算符.....	110
delete 运算符.....	110
void 运算符.....	111
typeof 运算符.....	111
前自增运算符.....	111
前自减运算符.....	112
一元 + 运算符.....	112
一元 - 运算符.....	112
按位非运算符.....	112
逻辑非运算符.....	113
乘法运算符.....	113
使用 * 运算符.....	113
使用 / 运算符.....	114
使用 % 运算符.....	114
加法运算符.....	115
加号运算符 (+).....	115
减号运算符 (-).....	116
加法作用于数字.....	116
位运算移位运算符.....	117
左移运算符.....	117
带符号右移运算符.....	117
无符号右移运算符.....	118

比较运算符.....	118
The Less-than Operator (<).....	119
The Greater-than Operator (>).....	120
The Less-than-or-equal Operator (<=).....	120
The Greater-than-or-equal Operator (>=).....	120
抽象关系比较算法.....	120
The instanceof operator	122
The in operator.....	122
等值运算符.....	122
The Equals Operator (==).....	123
The Does-not-equals Operator (!=).....	123
抽象相等比较算法.....	124
严格等于运算符 (===).....	125
The Strict Does-not-equal Operator (!==).....	125
严格等于比较算法.....	125
二进制位运算符.....	126
二元逻辑运算符.....	127
条件运算符.....	128
赋值运算符.....	129
简单赋值.....	130
组合赋值.....	130
逗号运算符.....	131

语句	131
块	132
变量语句.....	133
严格模式的限制.....	135
空语句.....	135
表达式语句.....	135
if 语句.....	136
迭代语句.....	137
do-while 语句.....	137
while 语句.....	137
for 语句.....	138
for-in 语句	139
continue 语句.....	140
break 语句	141
return 语句	141
with 语句.....	142
严格模式的限制.....	143
switch 语句.....	143
标签语句.....	145
throw 语句.....	146
try 语句	146
严格模式的限制.....	148

debugger 语句	148
函数定义	148
严格的模式的限制	150
创建函数对象	150
[[call]]	151
[[Construct]]	151
[[ThrowTypeError]] 函数对象	152
程序	152
指令序言和严格模式指令	154
标准 ECMAScript 内置对象	154
全局对象	155
全局对象的值属性	156
全局对象的函数属性	156
处理 URI 的函数属性	159
全局对象的构造器属性	165
全局对象的其他属性	166
Object 对象	167
作为函数调用 Object 构造器	167
Object 构造器	167
Object 构造器的属性	168
Object 的 prototype 对象的属性	172
Object 的实例的属性	174

Function 对象	174
作为函数调用 Function 构造器	174
Function 构造器	175
Function 构造器的属性	176
Function 的 prototype 对象的属性	176
Function 的实例的属性	180
Array 对象	181
作为函数调用 Array 构造器	182
Array 构造器	182
Array 构造器的属性	183
数组原型对象的属性	184
Array 实例的属性	206
String 对象	208
作为函数调用 String 构造器	208
String 构造器	208
String 构造器的属性	209
字符串原型对象的属性	209
String 实例的属性	222
布尔对象	223
作为函数调用布尔构造器	223
布尔构造器	223
布尔构造器的属性	224

布尔原型对象的属性.....	224
布尔实例的属性.....	225
Number 对象.....	225
作为函数调用的 Number 构造器	225
Number 构造器	225
Number 构造器的属性	226
数字原型对象的属性.....	227
数字实例的属性.....	232
Math 对象.....	232
Math 对象的值属性	232
Math 对象的函数属性	234
Date 对象	240
Date 对象的概述和抽象操作的定义	240
作为函数调用 Date 构造器	248
Date 构造器	248
Date 构造器的属性	250
Date 原型对象的属性	251
Date 实例的属性	262
RegExp (正则表达式) 对象	262
模式.....	263
模式语义.....	265
The RegExp Constructor Called as a Function	282

The RegExp Constructor	282
Properties of the RegExp Constructor	283
Properties of the RegExp Prototype Object	283
Properties of RegExp Instances	286
Error Objects	286
The Error Constructor Called as a Function	287
The Error Constructor	287
Properties of the Error Constructor	287
Properties of the Error Prototype Object	288
Error实例的属性	289
Native Error Types Used in This Standard	289
NativeError对象结构	290
JSON 对象	292
JSON 语法	292
parse (text [, reviver])	294
stringify (value [, replacer [, space]])	296
错误	300
文法摘要	301
词法	301
数字转换	303
表达式	304
语句	306

函数和程序.....	306
统一资源定位符字符分类.....	307
正则表达式.....	307
JSON.....	307
JSON词法.....	308
JSON语法.....	308
兼容性.....	308
附加语法.....	308
数字直接量.....	308
字符串直接量.....	309
附加属性.....	310
escape(string)	310
unescape(string)	311
String.prototype.substr(start, length)	311
Date.prototype.getYear()	312
Date.prototype.setYear(year)	312
Date.prototype.toGMTString()	313
ECMAScript 的严格模式.....	313
第 5 版中可能会对第 3 版产生兼容性影响的更正及澄清.....	314
第 5 版内容的增加与变化，介绍第 3 版不兼容问题.....	315
5.1 版中技术上的重大更正和阐明.....	318
参考书目.....	320

介绍

这一 Ecma 标准建立在一些原本的技术上，最为著名的是 JavaScript(网景)和 JScript (微软)。语言由网景的 Brendan Eich 发明而第一次出现在这个公司的 Navigator 2.0 浏览器上。它出现在所有 Netscape 后来的浏览器以及微软从 Internet Explorer 3.0 之后的所有浏览器上。

这一标准的编制自 1996 年十一月开始。这一 Ecma 标准的第一个版本被 1997 年六月的 Ecma General Assembly 采纳。

上述 Ecma 标准被以快速跟进流程提交至 ISO/IEC JTC 1，并作为于 1998 年四月作为 ISO/IEC 16262 通过。1998 年六月 Ecma General Assembly 通过了 ECMA-262 第二版以保持它与 ISO/IEC 16262 的完全一致性。第一版到第二版的变更仅仅是编辑性质的。

第三版标准引入了强大的正则表达式，更佳的字符串处理，新的控制语句，try/catch 异常处理，更严密地错误定义，格式化的数字输出以及一些为国际化和未来语言成长预留的小变更。ECMAScript 标准的第三版 1999 年十二月的 Ecma General Assembly 采纳并于 2002 年六月作为 ISO/IEC 16262:2002 发布。

自第三版发布以来，ECMAScript 因其与万维网的关联而获得了广泛采用，它已经成为所有 web 浏览器实质上都提供的一种编程语言。为了编制第四版 ECMAScript，有很多有意义的工作。尽管这工作没能完成而且也没有作为 ECMAScript 的第四版发布，它促进了语言的进化。ECMAScript 第五版（发布为 ECMA-262 5th edition）纸面化了很多事实上已经在浏览器形成共识的语言规范解析并且增加了对自第三版发布以来的新功能的支持。这些功能包括访问器属性，反射创建以及对象检测，属性特性的程序控制，新增的数组操作函数，JSON 对象编码格式，以及提供了改进的错误检查以及程序安全性的严格模式。

这一 ECMAScript 5.1 版本标准完全与国际标准 ISO/IEC 16262:2011 的第三版本一致。

ECMAScript 是一个充满活力的语言，而且语言的演进尚未完成。有意义的技术性增强将会在未来版本的规范中持续进行。

这一 Ecma 规范由 2011 年六月 Ecma General Assembly 采纳。

范围

此标准定义了 ECMAScript 脚本语言。

一致性

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

符合标准的 ECMAScript 实现，应当能解释符合 Unicode 标准 3.0 或更高版本以 UCS-2 或 UTF-16 作为编码格式的 ISO/IEC 10646-1 第 3 级实现里的字符。如果没有额外指明采用的 ISO/IEC 10646-1 子集，则假定组号为 300 的 BMP 子集。如果没有额外指明采用的编码格式，则假定编码格式为 UTF-16。

符合标准的 ECMAScript 实现，允许提供超出本规范描述的额外类型、值、对象、属性、函数。尤其是本规范中描述的对象，是允许提供未在本规范中描述的属性和属性值的。

符合标准的 ECMAScript 实现，允许支持本规范未描述的程序语法和正则表达式语法。尤其是本规范 7.6.1.2 列出的“未来保留字”，是允许作为程序语法的。

参考文献

为了实现符合本规范的应用程序，下列引用文档是不可或缺的。对于标注了日期的文档，仅适用标注的那个版本。对于未标注日期的文档，以文档的最新版为准（包括任何修订版）。

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

概述

本节包含对 ECMAScript 语言非规范性的概述。

ECMAScript 是在宿主环境中执行计算，处理对象的面向对象编程语言。这里定义的 ECMAScript 并未打算要计算性自足；事实上，本规范没有任何针对输入外部数据或输出计算结果的条文。相反，我们期望 ECMAScript 程序的计算环境可提供本规范中描述的对象和其它设施之外的、某些特定环境下的 宿主

(host) 对象，除了说明宿主对象应该提供可被 **ECMAScript** 程序访问的某些属性，调用的某些方法外，关于它的其他描述和行为超出了本规范涉及的范围。

脚本语言 是一种用于操作，自定义，自动化现有系统设施的编程语言。在这种系统中，已经可以通过一个用户界面使用可用功能，脚本语言是一种机制，暴露这些功能给程序控制。这样，现有系统可以说给完善脚本语言能力需要的对象和设施提供了一个宿主环境。脚本语言被设计成专业和非专业程序员都能使用。

ECMAScript 最初被设计为 **Web** 脚本语言，提供了一种机制，使浏览器里的网页更加活跃，成为基于 **Web** 的客户 - 服务器架构的一部分执行服务器计算。**ECMAScript** 可以为各种宿主环境提供核心的脚本功能，因此本文档为不依赖特定宿主环境的核心脚本语言作出规范。

ECMAScript 的一些机能和其他编程语言的类似：特别是 **Java™**，**Self**，和 **Scheme**。以下文献描述了他们：

Gosling, James, Bill Joy and Guy Steele. 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. 227–241, Orlando, FL, October 1987.

IEEE Standard for the Scheme Programming Language. IEEE Std 1178-1990.

Web 脚本语言

WEB 浏览器为引入客户端计算能力而提供 **ECMAScript** 宿主环境，例如，它提供的对象有：windows, menus, pop-ups, dialog boxes, text areas, anchors, frames, history, cookies 及输入 / 输出等等。进一步来说，**WEB** 浏览器中提供的这种宿主环境，它提供了一种方式，使得脚本代码可以去处理诸如改变焦点、页面和图片的加载、卸载、错误和放弃，选择，表单提交和鼠标交互等等事件。脚本代码出现在 **HTML** 中，显示出来的页面是一个用户接口元素与固定的和计算出来的文本和图片的集合。脚本代码根据用户的交互而做出反应，并不需要存在一个主程序。

WEB 服务器为了服务端的计算则提供了一个完全不一样的宿主环境，包括的对象有：requests, clients, files 以及数据锁定和分享的机制。通过浏览器端脚本及服务端脚本的配合使用，在为基于 **WEB** 方式的应用程序提供定制的用户接口时，可以将计算分布到客户端和服务端进行。

每一种支持 **ECMAScript** 的 **WEB** 浏览器和服务器都将它们自身的宿主环境作为 **ECMAScript** 的补充，以使得 **ECMAScript** 的执行环境变得完整。

语言概述

下面是非正式的 ECMAScript 概述 -- 并未描述语言的所有部分。此概述并非标准的一部分。

ECMAScript 是基于对象的：基本语言和宿主设施都由对象提供，ECMAScript 程序是一组可通信的对象。ECMAScript 对象 (objects) 是 属性 (properties) 的集合，每个属性有零个或多个 特性 (attributes)，它确定怎样使用此属性。例如，当设置一个属性的 Writable 特性为 false 时，任何试图更改此属性值的 ECMAScript 代码的都会运行失败。属性是持有其他 对象 (objects)，原始值 (primitive values)，函数 (functions) 的容器。原始值是以下内置类型之一的成员：Undefined, Null, Boolean, Number, String；对象是剩下的内置类型 Object 的成员；函数是可调用对象 (callable object)。方法 (method) 是通过属性与对象关联的函数。

ECMAScript 定义一组内置对象 (built-in objects)，勾勒出 ECMAScript 实体的定义。这些内置对象包括 全局对象 (global object)，Object 对象，Function 对象，Array 对象，String 对象，Boolean 对象，Number 对象，Math 对象，Date 对象，RegExp 对象，JSON 对象，和 Error 对象：Error，EvalError，RangeError，ReferenceError，SyntaxError，TypeError，URIError。

ECMAScript 中还定义一组内置运算符 (operators)。ECMAScript 运算符包括一元运算符，乘法运算符，加法运算符，按位移位运算符，关系运算符，相等运算符，二进制位运算符，二进制逻辑运算符，赋值运算符，逗号运算符。

ECMAScript 语法有意设计成与 Java 语法类似。ECMAScript 的语法是松散的，使其能够作为一个易于使用的脚本语言。例如，一个变量不需要有类型声明，属性也不需要与类型关联，定义的函数也不需要声明在函数调用词句的前面。

对象

ECMAScript 不使用诸如 C++，Smalltalk，Java 中的类。相反，对象可以通过各种方式创建，包括字面符号，或通过 构造器 创建对象然后运行代码初始化其全部或部分属性值，为这些属性分配初始值。每个构造器是一个拥有名为“prototype”的属性的函数。此属性用于实现基于原型的继承 和 共享属性。构造器通过 new 表达式创建对象：例如，new Date(2009,11) 创建一个新 Date 对象。不使用 new 调用一个构造器的结果，依赖构造器本身。例如，Date() 产生一个表示当前日期时间的字符串，而不是一个对象。

每个由构造器创建的对象，都有一个隐式引用（叫做对象的原型）链接到构造器的“prototype”属性值。再者，原型可能有一个非空 (non-null) 隐式引用链接到它自己的原型，以此类推，这叫做 原型链。当向对象的一个属性提出引用，引

用会指向原型链中包含此属性名的第一个对象的此属性。换句话说，首先检查直接提及的对象的同名属性，如果对象包含同名的属性，引用即指向此属性，如果该对象不包含同名的属性，则下一步检查对象的原型；以此类推。

对象/原型关系

一般情况下基于类的面向对象语言的实例拥有状态，类拥有方法，并且只能继承结构和行为。在 **ECMAScript** 中，对象拥有状态和方法，并且结构，行为，状态全都可继承。

不直接包含原型中包含的特定属性的所有对象会共享此属性及属性值。图 1 说明了这一点：

CF 是一个构造器（也是一个对象）。五个对象已用 **new** 表达式创建：**cf1**, **cf2**, **cf3**, **cf4**, **cf5**。每个对象都有名为 **q1** 和 **q2** 的属性。虚线表示隐式原型关系；例如：**cf3** 的原型是 **CFp**。构造器 **CF** 自己有名为 **P1** 和 **P2** 的两个属性，这对 **CFp**, **cf1**, **cf2**, **cf3**, **cf4**, **cf5** 是不可见的。**CFp** 的名为 **CFP1** 的属性共享给 **cf1**, **cf2**, **cf3**, **cf4**, **cf5**（没有 **CF**），以及在 **CFp** 的隐式原型链中找不到任何名为 **q1**, **q2**, 或 **CFP1** 的属性。请注意，**CF** 和 **CFp** 之间没有隐式原型链接。

不同于基于类的对象语言，属性可以通过赋值的方式动态添加给对象。也就是说，构造器并不是非要对构造的对象的全部或任何属性命名或赋值。上图中，可以给 **CFp** 添加新属性值的方式为 **cf1**, **cf2**, **cf3**, **cf4**, **cf5** 添加一个新的共享属性。

ECMAScript 的严格模式变体

ECMAScript 语言认可有些用户希望限制使用语言中某些功能的可能性。他们这样做可能是为了安全考虑，避免他们认为是容易出错的功能，获得增强的错误检查，或其他原因。为了支持这种可能性，**ECMAScript** 中定义了语言的严格变体。语言的严格变体，排除了 **ECMAScript** 语言的某些特定的语法和语义特征，还修改了某些功能的详细语义。严格变体还指定了必须抛出错误异常报告的额外错误条件，即使在非严格的语言形式下这些条件不属于错误。

ECMAScript 的严格变体通常被称为语言的 严格模式 (**strict mode**)。严格模式选择使用的 **ECMAScript** 严格模式的语法和语义，明确地适用于个别 **ECMAScript** 代码单元级别。由于严格模式适用于选择的语法代码单元级别，严格模式仅在这个代码单元内施加有局部效果的限制。严格模式不限制或修改任何必须运行在多个代码单元的 **ECMAScript** 语义层面。一个 **ECMAScript** 程序可由严格模式和非严格模式的代码单元组成。在这种情况下，严格的模式只适用于严格模式代码单元内实际执行的代码。

要符合这一规范，**ECMAScript** 的实现必须同时实现未限制的 **ECMAScript** 语言和按照这个规范定义的 **ECMAScript** 的严格模式变体。此外，实现还必须支持未限制的和严格模式代码单元的在同一个程序中混用。。

术语定义

本文档将使用下列术语和定义。

类型 (type)

本规范第 8 章定义数据的集合。

原始值 (primitive value)

在本规范的第 8 章定义的未定义，空，布尔，数字，字符串类型之一的成员。

原始值直接代表语言实现的最底层的数据。

对象 (object)

对象类型的成员。

对象是属性的集合，并有一个原型对象。原型可以是空值。

构造器 (constructor)

创建和初始化对象的函数对象。

构造器的“prototype”属性值是一个原型对象，它用来实现继承和共享属性。

原型 (prototype)

为其他对象提供共享属性的对象。

当构造器创建一个对象，为了解决对象的属性引用，该对象会隐式引用构造器的“prototype”属性。通过程序表达式 `constructor.prototype` 可以引用到构造器的“prototype”属性，并且添加到对象原型里的属性，会通过继承与所有共享此原型的对象共享。另外，可使用 `Object.create` 内置函数，通过明确指定原型来创建一个新对象。

原生对象 (native object)

ECMAScript 实现中，并非由宿主环境，而是完全由本规范定义其语义的对象。

标准的原生对象由本规范定义。一些原生对象是内置的，其他的可在 ECMAScript 程序执行过程中构建。

内置对象 (built-in object)

由 ECMAScript 实现提供，独立于宿主环境的对象，ECMAScript 程序开始执行时就存在。

标准的内置对象由本规范定义，ECMAScript 实现可以指定和定义其他的。所有内置对象是原生对象。一个内置构造器 (built-in constructor) 是个内置对象，也是个构造器。

宿主对象 (host object)

由宿主环境提供的对象，用于完善 ECMAScript 执行环境。

任何对象，不是原生对象就是宿主对象。

未定义值 (undefined value)

说明一个变量没有被分配值的一个原始值。

未定义类型 (Undefined type)

拥有唯一值“未定义值”的类型。

空值 (null value)

代表对象值故意留空的一个原始值。

空类型 (Null type)

拥有唯一值“空值”的类型。

布尔值 (Boolean value)

布尔类型的成员。

只有两个布尔值，true 和 false 。

布尔类型 (Boolean type)

由原始值 `true` 和 `false` 组成的类型。

布尔对象 (Boolean object)

对象类型的成员，它是标准内置构造器 `Boolean` 的一个实例。

通过使用 `new` 表达式，以一个布尔值作为参数调用 `Boolean` 构造器来创建布尔对象。由此产生的对象包含一个值为此布尔值的内部属性。一个 `Boolean` 对象可以强制转换为布尔值。

字符串值 (String value)

原始值，它是零个或多个 16 位无符号整数组成的有限有序序列。

一个字符串值是字符串类型的成员。通常序列中的每个整数值代表 UTF-16 文本的单个 16 位单元。然而，对于其值，ECMAScript 只要求必须是 16 位无符号整数，除此之外没有任何限制或要求。

字符串类型 (String type)

所有可能的字符串值的集合。

字符串对象 (String object)

对象类型的成员，它是标准内置构造器 `String` 的一个实例。

通过使用 `new` 表达式，以一个字符串值为参数调用 `String` 构造器来创建字符串对象。由此产生的对象包含一个值为此字符串值的内部属性。将 `String` 构造器作为一个函数来调用，可将一个字符串对象强制转换为一个字符串值（15.5.1）。

数字值 (Number value)

原始值，对应一个 64 位双精度二进制 IEEE754 值。

一个数字值是数字类型的成员，直接代表一个数字。

数字类型 (Number type)

所有可能的数字值的集合，包括特殊的“Not-a-Number”(NaN) 值，正无穷，负无穷。

数字对象 (Number object)

对象类型的成员，它是标准内置构造器 `Number` 的一个实例。

通过使用 `new` 表达式，以一个数字值为参数调用 `Number` 构造器来创建数字对象。由此产生的对象包含一个值为此数字值的内部属性。将 `Number` 构造器作为一个函数来调用，可将一个 `Number` 对象强制转换为一个数字值(15.7.1)。

无穷 (Infinity)

正无穷数字值。

NaN

值为 IEEE 754“Not-a-Number”的数字值

函数 (function)

对象类型的成员，标准内置构造器 `Function` 的一个实例，并且可做为子程序被调用。

函数除了拥有命名的属性，还包含可执行代码、状态，用来确定被调用时的行为。函数的代码不限于 ECMAScript。

内置函数 (built-in function)

作为函数的内置对象。

如 `parseInt` 和 `Math.exp` 就是内置函数。一个实现可以提供本规范没有描述的依赖于实现的内置函数。

属性 (property)

作为对象的一部分联系名和值。

属性可能根据属性值的不同表现为直接的数据值（原始值，对象，或一个函数对象）或间接的一对访问器函数。

方法 (method)

作为属性值的函数。

当一个函数被作为一个对象的方法调用，此对象将作为 **this** 值传递给函数。

内置方法 (built-in method)

作为内置函数的方法。

标准内置方法由本规范定义，一个 ECMAScript 实现可指定，提供其他额外的内置方法。

特性 (attribute)

定义一个属性的一些特性的内部值。

自身属性 (own property)

对象直接拥有的属性。

继承属性 (inherited property)

不是对象的自身属性，但是是对象原型的属性（原型的自身属性或继承属性）。

记法约定

语法和词法的文法

上下文无关文法

一个 上下文无关文法 由一定数量的 产生式 (productions) 组成。每个产生式的 左边 (left-hand side) 是一个被称为非终结符 (nonterminal) 的抽象符号，

右边 (right-hand side) 是零或多个非终结符和 终结符 (terminal symbols) 的有序排列。任何文法，它的终结符都来自指定的字母集。

当从一个叫做 目标符 (goal symbol) 的特殊非终端符组成的句子起始，那么给出的上下文无关文法就表示 语言 (language)，即，将产生式右边序列的非终结符当作左边，进行反复替换的结果就成为可能的终结符序列集合（可能无限）。

词法和正则的文法

第 7 章给出了 ECMAScript 的 词法文法 (lexical grammar)。作为此文法的终结符字符 (Unicode 代码单元) 符合第 6 章定义的 SourceCharacter 的规则。它定义了一套产生式，从目标符 InputElementDiv 或 InputElementRegExp 起始，描述了如何将这样的字符序列翻译成一个输入元素序列。

空白和注释之外的输入元素构成 ECMAScript 语法的终结符，它们被称为 ECMAScript 的 tokens。这些 tokens 是，ECMAScript 语言的保留字，标识符，字面量，标点符号。此外，行结束符虽然不被视为 tokens，但会成为输入元素流的一部分，用于引导处理自动插入分号（7.9）。空白和单行注释会被简单的丢弃，不会出现在语法的输入元素的流中。如果一个多行注释 (MultiLineComment)（即形式为“/ ... /”的注释，不管是否跨越多行）不包含行结束符也会简单地丢弃，但如果一个 多行注释包含一个或多个结束符，那么，注释会被替换为一个行结束符，成为语法输入元素流的一部分。

15.10 给出了 ECMAScript 的 正则文法 (RegExp grammar)。此文法的终结符字符也由 SourceCharacter 定义。它定义了一套产生式，从目标符 Pattern 起始，描述了如何将这样的字符序列翻译成一个正则表达式模式。

两个冒号“::”作为分隔符分割词法和正则的文法产生式。词法和正则的文法共享某些产生式。

数字字符串文法

用于转换字符串为数字值的一种文法。此文法与词法文法的一部分（与数字字面量有关的）类似，并且有终结符 SourceCharacter。此文法出现在 9.3.1。

三个冒号“:::”作为分隔符分割数字字符串文法的产生式。

语法规法

第 11, 12, 13, 14 章给出了 ECMAScript 的 语法规法。词法文法定义的 ECMAScript tokens 是此文法的终结符（5.1.2）。它定义了一组起始于

Program 目标符的产生式，描述了语法正确的 **ECMAScript** 程序应该怎样排列 **tokens**。

当一个字符流被解析为 **ECMAScript** 程序，它首先通过词法文法应用程序反复转换为一个输入元素流；然后再用一个语法文法应用程序解析这个输入元素流。当输入元素流没有更多 **tokens** 时，如果 **tokens** 不能解析为 **Program** 目标非终结符的单一实例，那么程序在语法上存在错误。

只用一个冒号“:”作为分隔符分割语法词法的产生式。

事实上第 11, 12, 13 和 14 章给出的语法规则，并不能完全说明一个正确的 **ECMAScript** 程序能接受的 **token** 序列。一些额外的 **token** 序列也被接受，即某些特殊位置（如行结束符前）加入分号可以被文法接受。此外，文法描述的某些 **token** 序列不被文法接受，如一个行结束符出现在“尴尬”的位置。

JSON 文法

JSON 文法用于将描述 **ECMAScript** 对象的字符串转换为实际的对象。15.12.1 给出了 **JSON** 文法。

JSON 文法由 **JSON** 词法文法和 **JSON** 语法文法组成。**JSON** 词法文法用于将字符序列转换为 **tokens**，类似 **ECMAScript** 词法文法。**JSON** 语法文法说明 **JSON** 词法文法给出怎样的 **tokens** 序列才能转换为语法上是正确的 **JSON** 对象。

两个冒号“::”作为分隔符分割 **JSON** 词法文法的产生式。**JSON** 词法文法使用某些 **ECMAScript** 词法文法的产生式。**JSON** 语法文法与 **ECMAScript** 语法文法类似。**JSON** 语法文法产生式被一个冒号“:”作为分隔符分割。

文法标记法

词法文法和字符串文法的终结符，以及一些语法文法的终结符，无论是在文法的产生式还是贯穿本规范的所有文本直接给出的终结符，都用 等宽 (**fixed width**) 字体显示。他们表示程序书写正确。所有以这种方式指定的终结符字符，可以理解为 **Unicode** 字符的完整的 **ASCII** 范围，不是任何其他类似的 **Unicode** 范围字符。

非终结符以 斜体 (**italic**) 显示。一个非终结符的定义由非终结符名称和其后定义的一个或多个冒号给出。（冒号的数量表示产生式所属的文法。）非终结符的右侧有一个或多个替代子紧跟在下一行。例如，语法定义：

```
WhileStatement  
    while ( Expression ) Statement
```

表示这个非终结符 **WhileStatement** 代表 **while token**，其后跟左括号 **token**，其后跟 **Expression**，其后跟右括号 **token**，其后跟 **Statement**。这里出现的 **Expression** 和 **Statement** 本身是非终结符。另一个例子，语法定义：

ArgumentList :

AssignmentExpression

ArgumentList , AssignmentExpression

表示这个 **ArgumentList** 可以代表一个 **AssignmentExpression**，或 **ArgumentList**，其后跟一个逗号，其后跟一个 **AssignmentExpression**。这个 **ArgumentList** 的定义是递归的，也就是说，它定义它自身。其结果是，一个 **ArgumentList** 可能包含用逗号隔开的任意正数个参数，每个参数表达式是一个 **AssignmentExpression**。这样，非终结符共用了递归的定义。

终结符或非终结符可能会出现后缀下标“**opt**”，表示它是可选符号。实际上包含可选符号的替代子包含两个右边，一个是省略可选元素的，另一个是包含可选元素的。这意味着：

VariableDeclaration :

Identifier Initialiseropt

是以下的一种缩写：

VariableDeclaration :

Identifier

Identifier Initialiser

并且：

IterationStatement :

for (ExpressionNoInopt ; Expressionopt ; Expressionopt) Statement

是以下的一种缩写：

IterationStatement :

for (; Expressionopt ; Expressionopt) Statement

for (ExpressionNoIn ; Expressionopt ; Expressionopt) Statement

是以下的一种缩写：

```

IterationStatement :
    for ( ; ; Expressionopt ) Statement
    for ( ; Expression ; Expressionopt ) Statement
    for ( ExpressionNoIn ; ; Expressionopt ) Statement
    for ( ExpressionNoIn ; Expression ; Expressionopt ) Statement

```

是以下的一种缩写：

```

IterationStatement :
    for ( ; ; ) Statement
    for ( ; ; Expression ) Statement
    for ( ; Expression ; ) Statement
    for ( ; Expression ; Expression ) Statement
    for ( ExpressionNoIn ; ; ) Statement
    for ( ExpressionNoIn ; ; Expression ) Statement
    for ( ExpressionNoIn ; Expression ; ) Statement
    for ( ExpressionNoIn ; Expression ; Expression ) Statement

```

因此，非终结 `IterationStatement` 实际上有 8 个右侧变体。

如果文法定义的冒号后面出现文字“one of”，那么其后续一行或多行出现的每个终结符都是一个选择定义。例如，ECMAScript 包含的词法文法生产器：

```

NonZeroDigit :: one of
1 2 3 4 5 6 7 8 9

```

这仅仅下面写法的一种缩写：

```

NonZeroDigit ::
    1
    2
    3

```

4
5
6
7
8
9

如果产生式的右侧是出现“[empty]”，它表明，产生式的右侧不包含终结符或非终结符。

如果产生式的右侧出现“[lookahead \notin set]”，它表明，给定 **set** 的成员不得成为产生式紧随其后的 **token**。这个 **set** 可以写成一个中括号括起来的终结符列表。为方便起见，**set** 也可以写成一个非终结符，在这种情况下，它代表了这个非终结符 **set** 可扩展所有终结符。例如，给出定义

```
DecimalDigit :: one of
0 1 2 3 4 5 6 7 8 9
DecimalDigits ::
DecimalDigit
DecimalDigits DecimalDigit
```

在定义

```
LookaheadExample ::
n [lookahead  $\notin$  {1 , 3 , 5 , 7 , 9}]DecimalDigits
DecimalDigit [lookahead  $\notin$  DecimalDigit ]
```

能匹配字母 **n** 后跟随由偶数起始的一个或多个十进制数字，或一个十进制数字后面跟随一个非十进制数字。

如果产生式的右侧出现“[no LineTerminator here]”，那么它表示此产生式是个受限的产生式：如果 **LineTerminator** 在输入流的指定位置出现，那么此产生式将不会被适用。例如，产生式：

```
ThrowStatement :
throw [no LineTerminator here] Expression ;
```

表示如果程序中 `return token` 和 `Expression` 之间的出现 `LineTerminator`，那么不得使用此产生式。

`LineTerminator` 除了禁止出现在受限的产生式，可以在输入元素流的任何两个 `tokens` 之间出现任意次数，而不会影响程序的语法验证。

当一个词法文法产生式或数字字符串文法中出现多字符 `token`，它表示此字符序列将注册一个 `token`。

使用词组“**but not**”可以指定某些不允许在产生式右侧的扩展，它说明排除这个扩展。例如，产生式：

Identifier ::

IdentifierName but not ReservedWord

此非终结符 `Identifier` 可以由可替换成 `IdentifierName` 的字符序列替换，相同的字符序列不能替换 `ReservedWord`。

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：

SourceCharacter ::

any Unicode code unit

算法约定

此规范通常使用带编号的列表来指定算法的步骤。这些算法是用来精确地指定 **ECMAScript** 语言结构所需的语义。该算法无意暗示任何具体实现使用的技术。在实践中，也许可用更有效的算法实现一个给定功能。

为了方便其使用本规范的多个部分，叫做 抽象操作 (**abstract operations**) 的一些算法编写成带名称的可传参函数化形式，所以在其他算法里可以通过名称引用它们。

当一个算法产生返回值，**“return x”** 指令说明该算法的返回值是 **x**，并且算法应该终止。“第 **n** 步的结果”的简写是 **Result(n)**。

为了表达清晰，算法的步骤可细分为有序的子步骤。子步骤被缩进，可以将自身进一步划分为缩进子步骤。大纲编号约定用于识别分步骤，第一层次的子步骤适用小写字母标记，第二层次的子步骤使用小写罗马数字标记。如果需要超过三个层次，则重复这些规则，第四层次使用数字标记。例如：

1. Top-level step

1. Substep.
2. Substep
 - i. Subsubstep.
 - ii. Subsubstep.
 1. Subsubsubstep
 1. Subsubsubsubstep

步骤或子步骤可写“if”谓词作为它的子步骤的条件。在这种情况下，当谓词为真时子步骤才适用。如果一个步骤或子步骤由单词“else”开始，那么它是一个谓词，否定前面的同一层级的“if”谓词。

步骤可以表示其子步骤的迭代应用可能指定其子步的迭代应用程序。

步骤可能断言其算法中的某一不变条件。这样的断言可以让算法中隐含的不变条件变成显式的。这种断言不会添加额外的语义要求，实现没有一定去检查的必要性。它们只是用来让算法更清晰。

数学运算，如加法，减法，取反，乘法，除法，还有稍后在本节中定义的数学函数应该总是被理解为对数学实数计算精确的数学结果，其中不包括无穷大，不包括负零区别于正零。本标准中的浮点运算算法模型，包括明确的步骤，在必要情况下处理无穷大和有符号零和执行四舍五入。如果一个数学运算或函数应用一个浮点数，它应该被应用为代表此浮点数的确切的数学值，一个浮点数必须是有限的，如果是 $+0$ 或 -0 ，则相应的数学值就是 0 。

数学函数 $\text{abs}(x)$ 产生 x 的绝对值，如果 x 是负数（小于零），它是 $-x$ ，否则是 x 本身。

如果 x 是正数，数学函数 $\text{sign}(x)$ 产生 1 ，如果 x 是负数产生 -1 。此标准中 x 为零的情况下不使用 sign 函数。

符号 “ x modulo y ” (y 必须有限且非零) 计算一个满足以下条件的 k 值，与 y 同号（或是零）， $\text{abs}(k) < \text{abs}(y)$ ，对一些整数 q 满足 $x - k = q \times y$ 。

数学函数 $\text{floor}(x)$ 产生不大于 x 的最大整数（最大可为正无穷）。

$\text{floor}(x) = x - (x \text{ modulo } 1)$.

如果算法定义“抛出一个异常”，算法的执行将被终止，且没有返回结果。已调用的算法也被终止，直到算法步骤使用术语“如果一个异常被抛出 ...”明确指出异常处理。一旦遇到这种算法步骤，异常将不再被视已发生过。

源代码文本

用 3.0 或更高版本 Unicode 字符编码的一个字符序列来表示 ECMAScript 源文本。该文本预期已经正常化为 Unicode Technical Report #15 中描述的 Unicode 正常化形式 C (canonical composition)。符合 ECMAScript 的实现不要求对文本执行正常化,也不要求将其表现为像执行了正常化一样。为了目的,此规范 ECMAScript 的源文本被假定为一个 16 位代码单元,本规范的目的序列。这样的包含 16 位代码单元序列的源文本可能不是有效的 UTF-16 字符编码。如果实际的源文本没有用 16 位代码单元形式编码,那么必须把它看作已经转换为 UTF-16 一样处理。

语法

SourceCharacter ::

any Unicode code unit

贯穿本文档,短语“代码单元 (code unit)”和单词“字符 (character)”特指表示文本的单个 16 位单元的 16 位无符号值。短语“Unicode 字符 (Unicode character)”特指单个 Unicode 标量值(这可能大于 16 位,因此它可能代表多个代码单元)表示的语言或排版上的抽象单位。短语“代码点 (code point)”是指这样一个 Unicode 标量值。“Unicode 字符”仅指由单一的 Unicode 标量值表示的实体:组合字符序列的每个组成部分都是单个“Unicode 字符”,尽管用户可能会认为整个序列是单个字符。

在字符串字面量,正则表达式字面量,标识符中的任意字符(代码单元),可以是由六个字符组成的 Unicode 转义序列,即 \u 加上四个 16 进制数字。在注释中,这样的转义序列被当作注释的一部分忽略掉。在字符串字面量或正则表达式字面量中,Unicode 转义序列会给字面量值贡献一个字符。在标识符中,转义序列给标识符贡献一个字符。

虽然本文档有时会提到“字符串 (string)”里的“字符 (character)”和代表字符代码单元的 16 位无符号整数间的“变换 (transformation)”。事实上并没有变换,因为实际上就是用 16 位无符号值代表“字符串”里的“字符”。

ECMAScript 与 Java 编程语言对 Unicode 转义序列有不同的解释。在 Java 程序中,如果 Unicode 转义序列 \u000A 出现在单行注释中,它会被解释成行终结符(Unicode 字符 000A 是换行),因此接下来的一个字符不是注释的一部分。与此类似,如果 Java 程序中的字符串字面量里出现 Unicode 转义序列 \u000A,它同样会被解释成行终结符,字符串字面量里不允许出现行终结符,不得不将作为字符串字面量字符值的换行符的 \u000A 替换成 \n。在 ECMAScript 程序中,始终不会解释注释里出现的 Unicode 转义序列,因此无法给注释贡献终止符。与此类似,如果 ECMAScript 程序中的字符串字面量里出现 Unicode 转义序列,它始终会贡献一个字符给字面量值,并且始终不会解释成有可能终止字符串字面量的行终结符或引号标记。

词法

ECMAScript 程序的源文本首先转换成一个输入元素序列；tokens，行终结符，注释，空白构成输入元素序列。从左到右扫描源文本，反复获取作为下一个输入元素的尽可能长的字符序列。

词法文法有两个目标符。InputElementDiv 目标符用在允许除法 (/) 或除赋值 (/=) 运算符开始的语法文法上下文中。InputElementRegExp 目标符用在其他语法文法上下文中。

没有允许除法或除赋值运算符开头，同时又允许 **RegularExpressionLiteral** 开头的语法文法上下文。这不会被分号插入（见 7.9）影响；如下面的例子：

```
a = b /hi/g.exec(c).map(d);
```

其中 **LineTerminator** 后的第一个非空白，非注释字符是斜线 (/)，并且这个语法上下文允许除法或除赋值运算符，所以不会在这个 **LineTerminator** 位置插入分号。也就是说，上面的例子解释为：

```
a = b / hi / g.exec(c).map(d);
```

语法：

InputElementDiv ::

WhiteSpace

LineTerminator

Comment

Token

DivPunctuatorInputElementRegExp ::

WhiteSpace

LineTerminator

Comment

Token

RegularExpressionLiteral

Unicode 格式控制字符

Unicode 格式控制字符（即，Unicode 字符数据库中“Cf”分类里的字符，如“左至右符号 (left-to-right mark)”或“右至左符号 (left-to-right mark)”）是用来控制被更高层级协议（如标记语言）忽略的文本范围的格式的控制代码。

允许在源文本中出现控制字符是有用的，以方便编辑和显示。所有格式控制字符可写入到注释，字符串字面量，正则表达式字面量中。

在某些语言中和控制字符用于创建必要的分隔符分割词或短语。在 ECMAScript 源文本里，和还可以用在标识符后的第一个字符。

控制字符主要出现的文本的开头，标记它是 Unicode，并允许检测文本的编码和字节顺序。用于这一目的字符，有时也可能出现在文本开始的后面，例如，一个合并的文件。字符被视为空白字符（见 [7.2]）。

表 1 总结了一些在注释，字符串字面量，正则表达式字面量之外被特殊对待的格式控制字符。

控制字符的使用

字符编码值	名称	正式名称	用途
\u200C	零宽非连接符	<ZWJ>	IdentifierPart
\u200D	零宽连接符	<ZWJ>	IdentifierPart
\uFEFF	位序掩码	<BOM>	Whitespace

空白字符

空白字符用来改善源文本的可读性和分割 **tokens**（不可分割的词法单位），此外就无关紧要。空白字符可以出现的两个 **token** 之间还可以出现在输入的开始或结束位置。空白字符，还可以出现在字符串字面量 (**StringLiteral**) 或正则表达式字面量 (**RegularExpressionLiteral**) (在这里它表示组成字面量的字符) 或注释 (**Comment**) 中，但是不能出现的其他任何 **token** 内。

表 2 中列出了 ECMAScript 空白字符。

空白字符

字符编码值	名称	正式名称
\u0009	制表符	<TAB>
\u000B	纵向制表符	<VT>
\u000C	进纸符	<FF>
\u0020	空格	<SP>
\u00A0	非断空格	<NBSP>

\uFEFF	位序掩码	<BOM>
其它分类“Zs”	其它任何 Unicode"空白分隔符"	<USP>

ECMAScript 实现必须认可 Unicode 3.0 中定义的所有空白字符。后续版本的 Unicode 标准可能定义其他空白字符。ECMAScript 实现可以认可更高版本 Unicode 标准里的空白字符。

语法：

WhiteSpace ::

<tab>

<vt>

<ff>

<sp>

<nbsp>

<bom>

<usp>

行终结符

像空白字符一样，行终止字符用于改善源文本的可读性和分割 **tokens**（不可分割的词法单位）。然而，不像空白字符，行终结符对语法文法的行为有一定的影响。一般情况下，行终结符可以出现在任何两个 **token** 之间，但也有少数地方，语法文法禁止这样做。行终结符也影响自动插入分号过程（7.9）。行终结符不能出现在 **StringLiteral** 之外的任何 **token** 内。行终结符只能出现在作为 **LineContinuation** 一部分的 **StringLiteral token** 里。

行终结符可以出现在 **MultiLineComment**（7.4）内，但不能出现在 **SingleLineComment** 内。

正则表达式的 **\s** 类匹配的空白字符集中包含行终结符。

表 3 列出了 ECMAScript 的行终止字符。

行终止字符

字符编码值	名称	正式名称
\u000A	进行符	<LF>
\u000D	回车符	<CR>

\u2028	行分隔符	<LS>
\u2029	段分隔符	<PS>

只有表 3 中的字符才被视为行终结符。其他新行或折行字符被视为空白，但不作为行终结符。字符序列 作一个行终结符。计算行数时它应该被视为一个字符。

语法：

```
LineTerminator :: LineTerminatorSequence ::
```

```
[lookahead ∉ ]
```

注释

注释可以是单行或多行。多行注释不能嵌套。

因为单行注释可以包含除了 `LineTerminator` 字符之外的任何字符，又因为有一般规则：一个 `token` 总是尽可能匹配更长，所以一个单行注释总是包含从 `//` 到行终结符之间的所有字符。然而，在该行末尾的 `LineTerminator` 不被看成是单行注释的一部分，它被词法文法识别成语法文法输入元素流的一部分。这一点非常重要，因为这意味着是否存在单行注释都不影响自动分号插入进程（见 7.9）。

像空白一样，注释会被语法文法简单丢弃，除了 `MultiLineComment` 包含行终结符字符的情况，这种情况下整个注释会当作一个 `LineTerminator` 提供给语法文法解析。

语法：

```
Comment ::
```

```
MultiLineComment
```

```
SingleLineCommentMultiLineComment ::
```

```
/* MultiLineCommentCharsopt*/MultiLineCommentChars ::
```

```
MultiLineNotAsteriskChar MultiLineCommentCharsopt
```

```
* PostAsteriskCommentCharsoptPostAsteriskCommentChars ::
```

```
MultiLineNotForwardSlashOrAsteriskChar
```

```
MultiLineCommentCharsopt
```

```
* PostAsteriskCommentCharsoptMultiLineNotAsteriskChar ::
```

```

SourceCharacter but not asterisk
*MultiLineNotForwardSlashOrAsteriskChar ::

SourceCharacter but not forward-slash / or asterisk
*SingleLineComment ::

// SingleLineCommentCharsoptSingleLineCommentChars ::

SingleLineCommentChar
SingleLineCommentCharsoptSingleLineCommentChar ::

```

SourceCharacter but not LineTerminator

Tokens

语法:

```

Token ::

IdentifierName

Punctuator

NumericLiteral

```

StringLiteral

DivPunctuator 和 RegularExpressionLiteral 产生式定义 tokens，但 Token 的产生式不包含它们。

标识符名和标识符

标识符名是 tokens，Unicode 标准第 5 章的“标识符”节给出的文法加入了一些小的修改来解释它。Identifier 是一个 IdentifierName 但不是一个 ReservedWord(见 7.6.1)。Unicode 标识符文法基于 Unicode 标准指出的 normative 和 informative 字符分类。所有符合 ECMAScript 的实现必须能够正确处理 Unicode 标准 3.0 版本中指定的分类里的字符的分类。

本标准增加了个别字符：在 IdentifierName 的任何位置允许出现美元符 (\$) 和下划线 (_)。

IdentifierName 还允许出现 Unicode 转义序列，它们被 UnicodeEscapeSequence 的 CV 计算成单个字符贡献给 IdentifierName (见 7.8.4)。UnicodeEscapeSequence 前面的 \ 不给 IdentifierName 贡献字符。UnicodeEscapeSequence 不能提供单个字符给将要成为非法字符的

IdentifierName。换句话说，如果一个 \UnicodeEscapeSequence 序列被 UnicodeEscapeSequence 的 CV 替换，结果必须仍是有效的包含与原 IdentifierName 精确相同字符序列的 IdentifierName。本规范说明的所有标识符是根据它的实际字符，不管转义序列贡献特定字符与否。

根据 Unicode 标准两个规范的 IdentifierName 相等，是说除非他们的代码单元序列准确相等，否则不同（换句话说，符合 ECMAScript 的实现只需要按位比较 IdentifierName 值）。其目的是为了传入编译器之前就把源文本转换为正常化形式 C。

ECMAScript 实现可以识别后续版本 Unicode 标准定义的标识符字符。如果考虑可移植性，程序员应该只采用 Unicode 3.0 中定义的标识符字符。

语法：

```
Identifier ::  
IdentifierName but not ReservedWordIdentifierName ::  
IdentifierStart  
IdentifierName IdentifierPartIdentifierStart ::  
UnicodeLetter  
$  
_  
\ UnicodeEscapeSequenceIdentifierPart ::  
IdentifierStart  
UnicodeCombiningMark  
UnicodeDigit  
UnicodeConnectorPunctuation UnicodeLetter  
any character in the Unicode categories  
“Uppercase letter (Lu)”, “Lowercase letter (Ll)”,  
“Titlecase letter (Lt)”, “Modifier letter (Lm)”,  
“Other letter (Lo)”, or “Letter number (Nl)”.  
UnicodeCombiningMark
```

any character in the Unicode categories “Non-spacing mark (Mn)”\\

or “Combining spacing mark (Mc)” UnicodeDigit

any character in the Unicode category “Decimal number (Nd)”
UnicodeConnectorPunctuation

any character in the Unicode category “Connector punctuation (Pc)” UnicodeEscapeSequence

see 7.8.4.

保留字

保留字不能作为 Identifier 的 IdentifierName。

语法

ReservedWord ::

Keyword

FutureReservedWord

NullLiteral

BooleanLiteral

关键词

下列 token 是 ECMAScript 的关键词，不能用作 ECMAScript 程序的 Identifiers。

语法

Keyword :: one of

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
```

未来保留字

下列词被用作建议扩展关键字，因此保留，以便未来可能采用这些扩展。

语法

```
FutureReservedWord :: one of  
  
class enum extends super  
  
const export import
```

当下列 `tokens` 出现在 严格模式代码 (strict mode code) (见 10.1.1) 里，将被当成是 `FutureReservedWords`。任意这些 `tokens` 出现在任意上下文中的严格模式代码 (strict mode code) 中，如果 `FutureReservedWord` 出现的位置会产生错误，那么必须抛出对应的异常：

```
implements let private public yield  
  
interface package protected static
```

标点符号

语法

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

/ /=

字面量

语法

```
Literal ::
```

```
NullLiteral
```

```
BooleanLiteral
```

```
NumericLiteral
```

```
StringLiteral
```

```
RegularExpressionLiteral
```

空值字面量

语法:

```
NullLiteral ::
```

```
null
```

语义:

空值字面量的值 `null`，是 `Null` 类型的唯一值。

布尔值字面量

语法:

```
BooleanLiteral ::
```

```
true
```

```
false
```

语义:

布尔值字面量的值 `true` 是个布尔类型值，即 `true`。

布尔值字面量的值 `false` 是个布尔类型值，即 `false`。

数值字面量

语法:

```
NumericLiteral ::  
    DecimalLiteral  
    HexIntegerLiteral  
    DecimalLiteral ::  
        DecimalIntegerLiteral . DecimalDigitsopt ExponentPartopt  
        . DecimalDigits ExponentPartopt  
    DecimalIntegerLiteral  
    ExponentPartoptDecimalIntegerLiteral ::  
        0  
    NonZeroDigit DecimalDigitsoptDecimalDigits ::  
        DecimalDigit  
    DecimalDigits DecimalDigit DecimalDigit :: one of  
        0 1 2 3 4 5 6 7 8 9 NonZeroDigit :: one of  
        1 2 3 4 5 6 7 8 9  
    ExponentPart ::  
        ExponentIndicator SignedInteger ExponentIndicator :: one  
        of  
        e ESignedInteger ::  
        DecimalDigits  
        + DecimalDigits  
        - DecimalDigits  
    HexIntegerLiteral ::  
        0x HexDigit  
        0X HexDigit  
    HexIntegerLiteral HexDigit HexDigit :: one of  
        0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

源字符中的 `NumericLiteral` 后面不允许紧跟着 `IdentifierStart` 或 `DecimalDigit`。

例如：

`3in`

是错误的，不存在两个输入元素 `3` 和 `in`。

语义：

一个数值字面量代表一个 `Number` 类型的值。此值取决于两个步骤：第一，由字面量得出的数学值 (mathematical value) (MV)；第二，这个数学值按照后面描述的规则舍入。

- `NumericLiteral::DecimalLiteral` 的 MV 是 `DecimalLiteral` 的 MV。
- `NumericLiteral::HexIntegerLiteral` 的 MV 是 `HexIntegerLiteral` 的 MV。
- `DecimalLiteral::DecimalIntegerLiteral` 的 MV 是 `DecimalIntegerLiteral` 的 MV。
- `DecimalLiteral::DecimalIntegerLiteral.DecimalDigits` 的 MV 是 `DecimalIntegerLiteral` 的 MV 加上 (`DecimalDigits` 的 MV 乘 10^{-n})，这里的 n 是 `DecimalDigits` 的字符个数。
- `DecimalLiteral::DecimalIntegerLiteral.ExponentPart` 的 MV 是 `DecimalIntegerLiteral` 的 MV 乘 10^e ，这里的 e 是 `ExponentPart` 的 MV。
- `DecimalLiteral::DecimalIntegerLiteral.DecimalDigits.ExponentPart` 的 MV 是 (`DecimalIntegerLiteral` 的 MV 加 (`DecimalDigits` 的 MV 乘 10^{-n})) 乘 10^e ，这里的 n 是 `DecimalDigits` 的字符个数， e 是 `ExponentPart` 的 MV。
- `DecimalLiteral::DecimalDigits` 的 MV 是 `DecimalDigits` 的 MV 乘 10^{-n} ，这里的 n 是 `DecimalDigits` 的字符个数。
- `DecimalLiteral::DecimalDigits.ExponentPart` 的 MV 是 `DecimalDigits` 的 MV 乘 10^{e-n} ，这里的 n 是 `DecimalDigits` 的字符个数， e 是 `ExponentPart` 的 MV。
- `DecimalLiteral::DecimalIntegerLiteral` 的 MV 是 `DecimalIntegerLiteral` 的 MV。
- `DecimalLiteral::DecimalIntegerLiteral.ExponentPart` 的 MV 是 `DecimalIntegerLiteral` 的 MV 乘 10^e ，这里的 e 是 `ExponentPart` 的 MV。
- `DecimalIntegerLiteral::0` 的 MV 是 0。
- `DecimalIntegerLiteral::NonZeroDigit.DecimalDigits` 的 MV 是 (`NonZeroDigit` 的 MV 乘 10^n) 加 `DecimalDigits` 的 MV，这里的 n 是 `DecimalDigits` 的字符个数。
- `DecimalDigits::DecimalDigit` 的 MV 是 `DecimalDigit` 的 MV。
- `DecimalDigits::DecimalDigits.DecimalDigit` 的 MV 是 (`DecimalDigits` 的 MV 乘 10) 加 `DecimalDigit` 的 MV。

- `ExponentPart :: ExponentIndicator SignedInteger` 的 MV 是 `SignedInteger` 的 MV。
- `SignedInteger :: DecimalDigits` 的 MV 是 `DecimalDigits` 的 MV。
- `SignedInteger :: + DecimalDigits` 的 MV 是 `DecimalDigits` 的 MV。
- `SignedInteger :: - DecimalDigits` 的 MV 是 `DecimalDigits` 的 MV 取负。
- `DecimalDigit :: 0` 或 `HexDigit :: 0` 的 MV 是 0。
- `DecimalDigit :: 1` 或 `NonZeroDigit :: 1` 或 `HexDigit :: 1` 的 MV 是 1。
- `DecimalDigit :: 2` 或 `NonZeroDigit :: 2` 或 `HexDigit :: 2` 的 MV 是 2。
- `DecimalDigit :: 3` 或 `NonZeroDigit :: 3` 或 `HexDigit :: 3` 的 MV 是 3。
- `DecimalDigit :: 4` 或 `NonZeroDigit :: 4` 或 `HexDigit :: 4` 的 MV 是 4。
- `DecimalDigit :: 5` 或 `NonZeroDigit :: 5` 或 `HexDigit :: 5` 的 MV 是 5。
- `DecimalDigit :: 6` 或 `NonZeroDigit :: 6` 或 `HexDigit :: 6` 的 MV 是 6。
- `DecimalDigit :: 7` 或 `NonZeroDigit :: 7` 或 `HexDigit :: 7` 的 MV 是 7。
- `DecimalDigit :: 8` 或 `NonZeroDigit :: 8` 或 `HexDigit :: 8` 的 MV 是 8。
- `DecimalDigit :: 9` 或 `NonZeroDigit :: 9` 或 `HexDigit :: 9` 的 MV 是 9。
- `HexDigit :: a` 或 `HexDigit :: A` 的 MV 是 10。
- `HexDigit :: b` 或 `HexDigit :: B` 的 MV 是 11。
- `HexDigit :: c` 或 `HexDigit :: C` 的 MV 是 12。
- `HexDigit :: d` 或 `HexDigit :: D` 的 MV 是 13。
- `HexDigit :: e` 或 `HexDigit :: E` 的 MV 是 14。
- `HexDigit :: f` 或 `HexDigit :: F` 的 MV 是 15。
- `HexIntegerLiteral :: 0x HexDigit` 的 MV 是 `HexDigit` 的 MV。
- `HexIntegerLiteral :: 0X HexDigit` 的 MV 是 `HexDigit` 的 MV。
- `HexIntegerLiteral :: HexIntegerLiteral HexDigit` 的 MV 是 (`HexIntegerLiteral` 的 MV 乘 16) 加 `HexDigit` 的 MV。

数值字面量的确切 MV 值一旦被确定，它就会舍入成 `Number` 类型的值。如果 MV 是 0，那么舍入值是 +0；否则，舍入值必须是 MV 对应的准确数字值（8.5 中定义），除非此字面量是有效数字超过 20 位的 `DecimalLiteral`，这种情况下，数字值可以用下面两种方式产生的 MV 值确定：一，将 20 位后的每个有效数字用 0 替换后产生的 MV，二，将 20 位后的每个有效数字用 0 替换，并且递增第 20 位有效数字位置的数字值，产生的 MV。如果一个数字是 `ExponentPart` 的一部分，并且：

- 它不是 0；或
- 它的左侧是非零数字，它的右侧是不在 `ExponentPart` 的非零数字。

符合标准的实现，在处理严格模式代码（见 10.1.1）时，按照 B.1.1 的描述，不得扩展 `NumericLiteral` 包含 `OctalIntegerLiteral` 的语法。

字符串字面量

一个字符串字面量是关闭的单引号或双引号里的零个或多个字符。每个字符都可以用一个转义序列代表。除了闭合引号字符，反斜杠，回车，行分隔符，段落分

隔符，换行符之外的所有字符都可以直接出现的字符串字面量里。任何字符都可以通过转移序列的形式出现。

语法

```
StringLiteral ::  
    " DoubleStringCharactersopt "  
    ' SingleStringCharactersopt 'DoubleStringCharacters ::  
DoubleStringCharacter  
DoubleStringCharactersoptSingleStringCharacters ::  
SingleStringCharacter  
SingleStringCharactersoptDoubleStringCharacter ::  
  
SourceCharacter but not double-quote " or backslash \ or  
LineTerminator  
  
\ EscapeSequence  
  
LineContinuationSingleStringCharacter ::  
  
SourceCharacter but not single-quote ' or backslash \ or  
LineTerminator  
  
\ EscapeSequence  
  
LineContinuationLineContinuation ::  
  
\ LineTerminatorSequenceEscapeSequence ::  
  
CharacterEscapeSequence  
  
0 [lookahead &#x2190; DecimalDigit]  
  
HexEscapeSequence  
  
UnicodeEscapeSequenceCharacterEscapeSequence ::  
  
SingleEscapeCharacter  
  
NonEscapeCharacter SingleEscapeCharacter :: one of  
  
' " \ b f n r t vNonEscapeCharacter ::  
  
SourceCharacter but not EscapeCharacter or  
LineTerminatorEscapeCharacter ::
```

SingleEscapeCharacter

DecimalDigit

x

uHexEscapeSequence ::

x HexDigit HexDigitUnicodeEscapeSequence ::

u HexDigit HexDigit HexDigit HexDigit

7.6 给出了 HexDigit 非终结符的定义。第 6 章 定义了 SourceCharacter。

语义

一个字符串字面量代表一个 String 类型的值。字面量的字符串值 (SV) 由字符串字面量各部分贡献的字符值 (CV) 描述。作为这一过程的一部分，字符字面量里的某些字符会被解释成包含数学值 (MV)，如 7.8.3 和下面描述的。

- StringLiteral :: "" 的 SV 是空字符序列。
- StringLiteral :: 的 SV 是空字符序列。
- StringLiteral :: " DoubleStringCharacters " 的 SV 是 DoubleStringCharacters 的 SV。
- StringLiteral :: ' SingleStringCharacters ' 的 SV 是 SingleStringCharacters 的 SV。
- DoubleStringCharacters :: DoubleStringCharacter 的 SV 是包含一个字符的序列，此字符的 CV 是 DoubleStringCharacter 的 CV。
- DoubleStringCharacters :: DoubleStringCharacter DoubleStringCharacters 的 SV 是 (DoubleStringCharacter 的 CV 后面跟着 DoubleStringCharacters 的 SV 里所有字符的) 序列。
- SingleStringCharacters :: SingleStringCharacter 的 SV 是包含一个字符的序列，此字符的 CV 是 SingleStringCharacter 的 CV。
- SingleStringCharacters :: SingleStringCharacter SingleStringCharacters 的 SV 是 (SingleStringCharacter 的 CV 后面跟着 SingleStringCharacters 的 SV 里所有字符的) 序列。
- LineContinuation :: \ LineTerminatorSequence 的 SV 是空字符序列。
- DoubleStringCharacter :: SourceCharacter but not double-quote " or backslash \ or LineTerminator 的 CV 是 SourceCharacter 字符自身。
- DoubleStringCharacter :: \ EscapeSequence 的 CV 是 EscapeSequence 的 CV。
- DoubleStringCharacter :: LineContinuation 的 CV 是空字符序列。
- SingleStringCharacter :: SourceCharacter but not single-quote ' or backslash \ or LineTerminator 的 CV 是 SourceCharacter 字符自身。
- SingleStringCharacter :: \ EscapeSequence 的 CV 是 EscapeSequence 的 CV。

- `SingleStringCharacter :: LineContinuation` 的 CV 是空字符序列。
- `EscapeSequence :: CharacterEscapeSequence` 的 CV 是 `CharacterEscapeSequence` 的 CV。
- `EscapeSequence :: 0 [lookahead \notin DecimalDigit]` 的 CV 是 字符 (Unicode 值 0000)。
- `EscapeSequence :: HexEscapeSequence` 的 CV 是 `HexEscapeSequence` 的 CV。
- `EscapeSequence :: UnicodeEscapeSequence` 的 CV 是 `UnicodeEscapeSequence` 的 CV。
- `CharacterEscapeSequence :: SingleEscapeCharacter` 的 CV 是表格 4 里的 `SingleEscapeCharacter` 确定的代码单元值字符：字符串单字符转义序列

转义序列	字符编码值	名称	符号
<code>\b</code>	<code>\u0008</code>	回格	<code><BS></code>
<code>\t</code>	<code>\u0009</code>	水平制表符	<code><HT></code>
<code>\n</code>	<code>\u000A</code>	进行 (新行)	<code><LF></code>
<code>\v</code>	<code>\u000B</code>	竖直制表符	<code><VT></code>
<code>\f</code>	<code>\u000C</code>	进纸	<code><FF></code>
<code>\r</code>	<code>\u000D</code>	回车	<code><CR></code>
<code>\"</code>	<code>\u0022</code>	双引号	<code>"</code>
<code>\'</code>	<code>\u0027</code>	单引号	<code>'</code>
<code>\\</code>	<code>\u005C</code>	反斜杠	<code>\</code>

- `CharacterEscapeSequence :: NonEscapeCharacter` 的 CV 是 `NonEscapeCharacter` 的 CV。
- `NonEscapeCharacter :: SourceCharacter but not EscapeCharacter or LineTerminator` 的 CV 是 `SourceCharacter` 字符自身。
- `HexEscapeSequence :: x HexDigit HexDigit` 的 CV 是 ((16 乘第一个 `HexDigit` 的 MV) 加第二个 `HexDigit` 的 MV) 代码单元确定的字符。
- `UnicodeEscapeSequence :: u HexDigit HexDigit HexDigit HexDigit` 的 CV 是 (4096 乘第一个 `HexDigit` 的 MV) 加 (256 乘第二个 `HexDigit` 的 MV) 加 (16 乘第三个 `HexDigit` 的 MV) 加 (第四个 `HexDigit` 的 MV) 代码单元确定的字符。

符合标准的实现，在处理严格模式代码（见 10.1.1）时，按照 B.1.2 的描述，不得扩展 `EscapeSequence` 包含 `OctalEscapeSequence` 的语法。

行终结符不能出现在字符串字面量里，除非它成为 `LineContinuation` 的一部分产生空字符序列。让字符串字面量的字符串值包含行终结符的正确方法是使用转义序列，如 `\n` 或 `\u000A`。

正则表达式字面量

正则表达式字面量是输入元素，每当字面量被评估时会转换为 **RegExp** 对象（见 15.10）。当一个程序中有两个正则表达式字面量评估成正则表达式对象，不能用 **===** 比较他们是否相等，即使两个字面量包含相同内容。**RegExp** 对象也可以在运行时使用 **new RegExp**（见 15.10.4）或以函数方式调用 **RegExp** 构造器来创建（见 15.10.3）。

下面的产生式描述了正则表达式字面量的语法，输入元素扫描器还用它搜索正则表达式字面量的结束位置。**RegularExpressionBody** 和 **RegularExpressionFlags** 包含的字符组成的字符串会直接传递给正则表达式构造器，在那里用更严格文法进行解析。一个实现可以扩展正则表达式构造器的文法。但它不能扩展 **RegularExpressionBody** 和 **RegularExpressionFlags** 产生式或使用这些产生式的产生式。

语法

```
RegularExpressionLiteral ::  
  
/ RegularExpressionBody /  
RegularExpressionFlagsRegularExpressionBody ::  
  
RegularExpressionFirstChar  
RegularExpressionCharsRegularExpressionChars ::  
  
[empty]  
  
RegularExpressionChars  
RegularExpressionCharRegularExpressionFirstChar ::  
  
RegularExpressionNonTerminator but not *or \or / or [  
  
RegularExpressionBackslashSequence  
  
RegularExpressionClassRegularExpressionChar ::  
  
RegularExpressionNonTerminator but not \or / or [  
  
RegularExpressionBackslashSequence  
  
RegularExpressionClassRegularExpressionBackslashSequenc  
e ::  
  
\  
RegularExpressionNonTerminatorRegularExpressionNonTermi  
nator ::  
  
SourceCharacter but not  
LineTerminatorRegularExpressionClass ::
```

```
[ RegularExpressionClassChars ]RegularExpressionClassChars  
ars ::
```

```
[empty] RegularExpressionClassChars  
RegularExpressionClassChar
```

```
RegularExpressionClassChar ::
```

```
RegularExpressionNonTerminator but not ]or \
```

```
RegularExpressionBackslashSequenceRegularExpressionFlags  
s ::
```

```
[empty]
```

RegularExpressionFlags IdentifierPart

正则表达式字面量不能为空;并不是说正则表达式字面量不能代表空,字符 `//` 会启动一个单行注释。要指定一个空正则,使用: `/(?:)/`。

语义

正则表达式字面量会评估为一个 `Object` 类型值,它是标准内置构造器 `RegExp` 的一个实例。此值取决于两个步骤:首先,展开组成正则表达式产生式 `RegularExpressionBody` 和 `RegularExpressionFlags` 的字符,将其以未解析形式分别存成两个字符串 `Pattern` 和 `Flags`。然后,在每次评估字面量时创建新对象,仿佛使用 `new RegExp(Pattern, Flags)` 一样,这里的 `RegExp` 是标准内置构造器名。新构造的对象将成为 `RegularExpressionLiteral` 的值。如果调用 `new RegExp` 会产生 15.10.4.1 指定的错误,那么必须把错误当作是早期错误(见第 16 章)。

自动分号插入

必须用分号终止某些 ECMAScript 语句(空语句,变量声明语句,表达式语句,do-while 语句,continue 语句,break 语句,return 语句,throw 语句)。这些分号总是明确的显示在源文本里。然而,为了方便起见,某些情况下这些分号可以在源文本里省略。描述这种情况会说:这种情况下给源代码的 token 流自动插入分号。

自动分号插入规则

分号插入有三个基本规则:

1. 左到右解析程序，当遇到一个不符合任何文法产生式的 **token**（叫做 违规 **token(offending token)**），那么只要满足下面条件之一就在违规 **token** 前面自动插入分号。
 - 至少一个 **LineTerminator** 分割了违规 **token** 和前一个 **token**。
 - 违规 **token** 是 **}**。
2. 左到右解析程序，**tokens** 输入流已经结束，当解析器无法将输入 **token** 流解析成单个完整 **ECMAScript** 程序，那么就在输入流的结束位置自动插入分号。
3. 左到右解析程序，遇到一个某些文法产生式允许的 **token**，但是此产生式是受限产生式，受限产生式的里紧跟在 **no LineTerminator here** 后的第一个终结符或非终结符的 **token** 叫做受限的 **token**，当至少一个 **LineTerminator** 分割了受限的 **token** 和前一个 **token**，那么就在受限 **token** 前面自动插入分号。

然而，上述规则有一个附加的优先条件：如果插入分号后解析结果是空语句，或如果插入分号后它成为 **for** 语句头部的两个分号之一（见 12.6.3），那么不会自动插入分号。

文法里的受限产生式只限以下：

PostfixExpression :

LeftHandSideExpression [no LineTerminator here] ++

LeftHandSideExpression [no LineTerminator here] --

ContinueStatement :

continue [no LineTerminator here] Identifier;

BreakStatement :

break [no LineTerminator here] Identifier;

ReturnStatement :

return [no LineTerminator here] Expression;

ThrowStatement :

throw [no LineTerminator here] Expression;

这些受限产生式的实际效果如下：

当遇到的 **++** 或 **--token** 将要被解析器当作一个后缀运算符，并且至少有一个 **LineTerminator** 出现 **++** 或 **--token** 和它之前的 **token** 之间，那么在 **++** 或 **--token** 前面自动插入一个分号。

当遇到 `continue`, `break`, `return`, `throw token`, 并且在下一个 `token` 前面遇到 `LineTerminator`, 那么在 `continue`, `break`, `return`, `throw token` 后面自动插入一个分号。

这对 ECMAScript 程序员的实际影响是：

后缀运算符 `++` 或 `--` 和它的操作数应该出现在同一行。

`return` 或 `throw` 语句的表达式开始位置应该和 `return` 或 `throw token` 同一行。

`break` 或 `continue` 语句的标示符应该和 `break` 或 `continue token` 同一行。

自动分号插入的例子

源代码：

```
{ 1 2 } 3
```

即使在自动分号插入规则下，它也不符合 ECMAScript 文法。做为对比，源代码：

```
{ 1 2 } 3
```

它还是不符合 ECMAScript 文法，但是它会被自动分号插入成为一下形式：

```
{ 1 ; 2 ; } 3;
```

这符合 ECMAScript 文法。

源代码：

```
for (a; b )
```

不符合 ECMAScript 文法，并且不会被自动分号插入所更改，因为 `for` 语句头部需要分号。自动分号插入从来不会插入成 `for` 语句头部的两个分号之一。

源代码：

```
return a + b
```

会被自动分号插入转换成以下形式：

```
return; a + b;
```

表达式 `a + b` 不会被当做是 `return` 语句要返回的值，因为有一个 `LineTerminator` 分割了它和 `return token`。

源代码：

```
a = b ++c
```

会被自动分号插入转换成以下形式：

```
a = b; ++c;
```

`++token` 不会被当做应用于变量 `b` 的后缀运算符，因为 `b` 和 `++` 之间出现了一个 `LineTerminator`。

源代码：

```
if (a > b) else c = d
```

它不符合 `ECMAScript` 文法，`else token` 前面不会被自动分号插入改变，即使没有文法产生式适用这一位置，因为自动插入分号后会解析成空语句。

源代码：

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

它不会被自动分号插入改变，因为第二行开始位置的括号表达式可以解释成函数调用的参数列表：

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

在赋值语句必须用左括号开头的情况下，程序员在前面语句的结束位置明确的提供一个分号是个好主意，而不是依赖于自动分号插入。

类型

本规范的算法操作各个有类型的值，可处理的类型在算法相关叙述中定义。类型又再分为 `ECMAScript` 语言类型 与 规范类型 。

`ECMAScript` 语言类型 是 `ECMAScript` 程序员使用 `ECMAScript` 语言直接操作的值对应的类型。`ECMAScript` 语言类型包括未定义（`Undefined`）、空值（`Null`）、布尔值（`Boolean`）、字符串（`String`）、数值（`Number`）、对象（`Object`）。

规范类型 是描述 ECMAScript 语言构造与 ECMAScript 语言类型语意的算法所用的元值对应的类型。规范类型包括 引用 、 列表 、 完结 、 属性描述式 、 属性标示 、 词法环境(Lexical Environment)、环境纪录(Environment Record)。规范类型的值是不一定对应 ECMAScript 实现里任何实体的虚拟对象。规范类型可用来描述 ECMAScript 表式运算的中途结果，但是这些值不能存成对象的变量或是 ECMAScript 语言变量的值。

在本规范中，我们将「 x 的类型」简写为 $\text{Type}(x)$ ，而类型指的就是上述的 ECMAScript 语言类型 与 规范类型。

Undefined 类型

Undefined 类型有且只有一个值，称为 *undefined*。任何没有被赋值的变量都有 *undefined* 值。

Null 类型

Null 类型有且只有一个值，称为 *null*。

Boolean 类型

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

String 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（若存在）在位置 0，下一个元素（若存在）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。

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。

Number 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 $+\infty$ 和 $-\infty$ 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， $2^{64}-2^{53}$ ）个值被称为有限数值。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** 和一个 **负零**。为简洁起见，类似地，在说明目的时，分别用符号 ***+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} （即 $+1 * 2^{53} * 2^{971}$ ）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 $+\infty$ 替换它；若选择 -2^{1024} ，用 $-\infty$ 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} 个整数值中的一个。参见 ToInt32 和 ToUint32 的描述，分别在 章节 9.5 和 9.6 中。

Object 类型

Object 是一个属性的集合。每个属性既可以是一个命名的数据属性，也可以是一个命名的访问器属性，或是一个内部属性：

- 命名的数据属性 (named data property) 由一个名字与一个 ECMAScript 语言值和一个 Boolean 属性集合组成
- 命名的访问器属性 (named accessor property) 由一个名字与一个或两个访问器函数，和一个 Boolean 属性集合组成。访问器函数用于存取一个与该属性相关联的 ECMAScript 语言值
- 内部属性 (internal property) 没有名字，且不能直接通过 ECMAScript 语言操作。内部属性的存在纯粹为了规范的目的。

有两种带名字的访问器属性（非内部属性）：`get` 和 `put`，分别对应取值和赋值。

Property 特性

本规范中的特性 (Attributes) 用于定义和解释命名属性 (properties) 的状态。命名的数据属性由一个名字关联到一个下表中列出的特性 (attributes)

命名的数据属性的特性

特性名称	取值范围	描述
[[Value]]	任何 ECMAScript 语	通过读 property 来取到该值

	言类型	
[[Writable]]	Boolean	如果为 false ，试图通过 ECMAScript 代码 [[Put]] 去改变该属性的 [[Value]]，将会失败
[[Enumerable]]	Boolean	如果为 true ，则该属性可被 for-in 枚举出来（参见 12.6.4），否则，该属性不可枚举。
[[Configurable]]	Boolean	如果为 false ，试图删除该属性，改变该属性为访问器属性，或改变它的 attributes（和 [[Value]] 不同），都会失败。

命名的访问器属性由一个名字关联到一个下表中列出的特性 (attributes)

命名的访问器属性的特性

特性名称	取值范围	描述
[[Get]]	Object 或 Undefined	如果该值为一个 Object 对象，那么它必须是一个函数对象。每次有对该属性进行 get 访问时，该函数的内部方法 [[Call]]（8.6.2）会被一个空参数列表调用，以返回该属性值
[[Set]]	Object 或 Undefined	如果该值为一个 Object 对象，那么它必须是一个函数对象。每次有对该属性进行 set 访问时，该函数的内部方法 [[Call]]（8.6.2）会被一个参数列表调用，这个参数列表包含分配的值作为唯一的参数。property 的内部方法 [[Set]] 产生的影响可能会，但不是必须的，对随后的 property 内部方法 [[Get]] 的调用返回结果产生影响。
[[Enumerable]]	Boolean	如果为 true ，则该属性可被 for-in 枚举出来（参见 12.6.4），否则，该属性不可枚举。
[[Configurable]]	Boolean	如果为 false ，试图删除该属性，改变该属性为访问器属性，或改变它的 attributes（和 [[Value]] 不同），都会失败。

如果某个命名属性的特性值没有在此规范中明确给出，那么它的默认值将使用下表的定义。

默认特性值

特性名称	默认值
[[Value]]	undefined
[[Get]]	undefined
[[Set]]	undefined
[[Writable]]	false
[[Enumerable]]	false
[[Configurable]]	false

Object 内部属性及方法

本规范使用各种内部属性来定义对象值的语义。这些内部属性不是 ECMAScript 语言的一部分。本规范中纯粹是以说明为目的定义它们。ECMAScript 实现必须表现为仿佛它被这里描述的内部属性产生和操作。内部属性的名字用闭合双方括号括起来。如果一个算法使用一个对象的一个内部属性，并且此对象没有实现需要的内部属性，那么抛出 **TypeError** 异常。

表 8 总结了本规范中适用于所有 ECMAScript 对象的内部属性。表 9 总结了本规范中适用于某些 ECMAScript 对象的内部属性。这些表中的描述如果没有特别指出是特定的原生 ECMAScript 对象，那么就说明了其在原生 ECMAScript 对象中的行为。宿主对象的内部属性可以支持任何依赖于实现的行为，只要其与本文档说的宿主对象的个别限制一直。

下面表的“值的类域”一列定义了内部属性关联值的类型。类型名称参考第 8 章定义的类型，作为增强添加了一下名称：“any”指值可以是任何 ECMAScript 语言类型；“primitive”指 Undefined, Null, Boolean, String, , Number；“SpecOp”指内部属性是一个内部方法，一个抽象操作规范定义一个实现提供它的步骤。“SpecOp”后面跟着描述性参数名的列表。如果参数名和类型名一致那么这个名字用于描述参数的类型。如果“SpecOp”有返回值，那么这个参数列表后跟着“→”符号和返回值的类型。

所有对象共有的内部属性

内部属性	取值范围	说明
[[Prototype]]	Object 或 Null	此对象的原型
[[Class]]	String	说明规范定义的对象分类的一个字符串值
[[Extensible]]	Boolean	如果是 true，可以向对象添加自身属性。
[[Get]]	SpecOp(propertyName) → any	返回命名属性的值
[[GetOwnProperty]]	SpecOp (propertyName) → Undefined 或 Property Descriptor	返回此对象的自身命名属性的属性描述，如果不存在返回 undefined
[[GetProperty]]	SpecOp (propertyName) → Undefined 或 Property Descriptor	返回此对象的完全填入的自身命名属性的属性描述，如果不存在返回 undefined
[[Put]]	SpecOp (propertyName, any, Boolean)	将指定命名属性设为第二个参数的值。flog 控制失败处理。
[[CanPut]]	SpecOp (propertyName) →	返回一个 Boolean 值，

	Boolean	说明是否可以在 PropertyName 上执行 [[Put]] 操作。
[[HasProperty]]	SpecOp (propertyName) → Boolean	返回一个 Boolean 值，说明对象是否含有给定名称的属性。
[[Delete]]	SpecOp (propertyName, Boolean) → Boolean	从对象上删除指定的自身命名属性。 flag 控制失败处理。
[[DefaultValue]]	SpecOp (Hint) → primitive	Hint 是一个字符串。返回对象的默认值
[[DefineOwnProperty]]	SpecOp (propertyName, PropertyDescriptor, Boolean) → Boolean	创建或修改自身命名属性为拥有属性描述里描述的状态。 flag 控制失败处理。

所有对象（包括宿主对象）必须实现表 8 中列出的所有内部属性。然而，对某些对象的 **[[DefaultValue]]** 内部方法，可以简单的抛出 **TypeError** 异常。

所有对象都有一个叫做 **[[Prototype]]** 的内部属性。此对象的值是 **null** 或一个对象，并且它用于实现继承。一个原生属性是否可以把宿主对象作为它的 **[[Prototype]]** 取决于实现。所有 **[[Prototype]]** 链必须是有限长度（即，从任何对象开始，递归访问 **[[Prototype]]** 内部属性必须最终到头，并且值是 **null**）。从 **[[Prototype]]** 对象继承来的命名数据属性（作为子对象的属性可见）是为了 **get** 请求，但无法用于 **put** 请求。命名访问器属性会把 **get** 和 **put** 请求都继承。

所有 ECMAScript 对象都有一个 **Boolean** 值的 **[[Extensible]]** 内部属性，它控制是否可以给对象添加命名属性。如果 **[[Extensible]]** 内部属性的值是 **false** 那么不得给对象添加命名属性。此外，如果 **[[Extensible]]** 是 **false** 那么不得更改对象的 **[[Class]]** 和 **[[Prototype]]** 内部属性的值。一旦 **[[Extensible]]** 内部属性的值设为 **false** 之后无法再更改为 **true**。

本规范的定义中没有 ECMAScript 语言运算符或内置函数允许一个程序更改对象的 **[[Class]]** 或 **[[Prototype]]** 内部属性或把 **[[Extensible]]** 的值从 **false** 更改成 **true**。实现中修改 **[[Class]]**, **[[Prototype]]**, **[[Extensible]]** 的个别扩展必须不违反前一段定义的不变量。

本规范的每种内置对象都定义了 **[[Class]]** 内部属性的值。宿主对象的 **[[Class]]** 内部属性的值可以是除了 "Arguments", "Array", "Boolean", "Date", "Error", "Function", "JSON", "Math", "Number", "Object", "RegExp", "String" 的任何字符串。**[[Class]]** 内部属性的值用于内部区分对象的种类。注，本规范中除了通过 **Object.prototype.toString** (见 15.2.4.2) 没有提供任何手段使程序访问此值。

除非特别指出，原生 ECMAScript 对象的公共内部方法的行为描述在 8.12。Array 对象的 `[[DefineOwnProperty]]` 内部方法有稍不同的实现（见 15.4.5.1），又有 String 对象的 `[[GetOwnProperty]]` 内部方法有稍不同的实现（见 15.5.5.2）。Arguments 对象（10.6）的 `[[Get]]`，`[[GetOwnProperty]]`，`[[DefineOwnProperty]]`，`[[Delete]]` 有不同的实现。Function 对象（15.3）的 `[[Get]]` 的有不同的实现。

除非特别指出，宿主对象可以以任何方式实现这些内部方法，一种可能是一个特别的宿主对象的 `[[Get]]` 和 `[[Put]]` 确实可以存取属性值，但 `[[HasProperty]]` 总是产生 `false`。然而，如果任何对宿主对象内部属性的操作不被实现支持，那么当试图操作时必须抛出 `TypeError` 异常。

宿主对象的 `[[GetOwnProperty]]` 内部方法必须符合宿主对象每个属性的以下不变量：

- 如果属性是描述过的数据属性，并随着时间的推移，它可能返回不同的值，那么即使没有暴露提供更改值机制的其他内部方法，`[[Writable]]` 和 `[[Configurable]]` 之一或全部必须是 `true`。
- 如果属性是描述过的数据属性，并且其 `[[Writable]]` 和 `[[Configurable]]` 都是 `false`。那么所有对 `[[GetOwnProperty]]` 的呼出，必须返回作为属性 `[[Value]]` 特性的 `SameValue`(9.12)。
- 如果 `[[Writable]]` 特性可以从 `false` 更改为 `true`，那么 `[[Configurable]]` 特性必须是 `true`。
- 当 ECMAScript 代码监测到宿主对象的 `[[Extensible]]` 内部属性值是 `false`。那么如果调用 `[[GetOwnProperty]]` 描述一个属性是不存在，那么接下来所有调用这个属性必须也描述为不存在。

如果 ECMAScript 代码监测到宿主对象的 `[[Extensible]]` 内部属性是 `false`，那么这个宿主对象的 `[[DefineOwnProperty]]` 内部方法不允许向宿主对象添加新属性。

如果 ECMAScript 代码监测到宿主对象的 `[[Extensible]]` 内部属性是 `false`，那么它以后必须不能再改为 `true`。

只在某些对象中定义的内部属性 Object

内部属性	取值范围	说明
<code>[[PrimitiveValue]]</code>	primitive	与此对象的内部状态信息关联。对于标准内置对象只能用 Boolean, Date, Number, String 对象实现 <code>[[PrimitiveValue]]</code> 。
<code>[[Construct]]</code>	SpecOp(a List of any) → Object	通过 new 运算符调，创建对象。SpecOp 的参数是通过 new 运算符传的参数。实现了这个内部方法的对象叫做 构造器。

[[Call]]	SpecOp(any, a List of any) → any or Reference	运行与此对象关联的代码。通过函数调用表达式调用。SpecOp 的参数是一个 this 对象和函数调用表达式传来的参数组成的列表。实现了这个内部方法的对象是 可调用 的。只有作为宿主对象的可调用对象才可能返回 引用 值。
[[HasInstance]]	SpecOp(any) → Boolean	返回一个表示参数对象是否可能是由本对象构建的布尔值。在标准内置 ECMAScript 对象中只有 Function 对象实现 [[HasInstance]]。
[[Scope]]	Lexical Environment	一个定义了函数对象执行的环境的词法环境。在标准内置 ECMAScript 对象中只有 Function 对象实现 [[Scope]]。
[[FormalParameters]]	List of Strings	一个包含 Function 的 FormalParameterList 的标识符字符串的可能是空的列表。在标准内置 ECMAScript 对象中只有 Function 对象实现 [[FormalParameterList]]。
[[Code]]	ECMAScript code	函数的 ECMAScript 代码。在标准内置 ECMAScript 对象中只有 Function 对象实现 [[Code]]。
[[TargetFunction]]	Object	使用标准内置的 Function.prototype.bind 方法创建的函数对象的目标函数。只有使用 Function.prototype.bind 创建的 ECMAScript 对象才有 [[TargetFunction]] 内部属性。
[[BoundThis]]	any	使用标准内置的 Function.prototype.bind 方法创建的函数对象的预绑定的 this 值。只有使用 Function.prototype.bind 创建的 ECMAScript 对象才有 [[BoundThis]] 内部属性。
[[BoundArguments]]	List of any	使用标准内置的 Function.prototype.bind 方法创建的函数对象的预绑定的参数值。只有使用 Function.prototype.bind 创建的 ECMAScript 对象才有 [[BoundArguments]] 内部属性。
[[Match]]	SpecOp(String, index) → MatchResult	测试正则匹配并返回一个 MatchResult 值（见 15.10.2.1）。在标准内置 ECMAScript 对象中只有 RegExp 对象实现 [[Match]]。

[[ParameterMap]]	提供参数对象的属性和函数关联的形式参数之间的映射。只有参数对象才有 [[ParameterMap]] 内部属性。
------------------	--

引用规范类型

引用类型用来说明 delete，typeof，赋值运算符这些运算符的行为。例如，在赋值运算中左边的操作数期望产生一个引用。通过赋值运算符左侧运算符的语法规则分析可以但不能完全解释赋值行为，还有个难点：函数调用允许返回引用。承认这种可能性纯粹是为了宿主对象。本规范没有定义返回引用的内置 ECMAScript 函数，并且也不提供返回引用的用户定义函数。（另一个不使用语法规则分析的原因是，那样将会影响规范的很多地方，冗长并且别扭。）

一个 引用 (Reference) 是个已解决的命名绑定。一个引用由三部分组成， 基 (base) 值， 引用名称 (referenced name) 和布尔值 严格引用 (strict reference) 标志。基值是 undefined， 一个 Object， 一个 Boolean， 一个 String， 一个 Number， 一个 environment record 中的任意一个。基值是 undefined 表示此引用可以不解决一个绑定。引用名称是一个字符串。

本规范中使用以下抽象操作接近引用的组件：

- **GetBase(V)**。 返回引用值 V 的基值组件。
- **GetReferencedName(V)**。 返回引用值 V 的引用名称组件。
- **IsStrictReference(V)**。 返回引用值 V 的严格引用组件。
- **HasPrimitiveBase(V)**。 如果基值是 Boolean, String, Number， 那么返回 true。
- **IsPropertyReference(V)**。 如果基值是个对象或 HasPrimitiveBase(V) 是 true， 那么返回 true； 否则返回 false。
- **IsUnresolvableReference(V)**。 如果基值是 undefined 那么返回 true， 否则返回 false。

本规范使用以下抽象操作来操作引用：

GetValue(v)

1. 如果 Type(V) 不是引用， 返回 V。
2. 令 base 为调用 GetBase(V) 的返回值。
3. 如果 IsUnresolvableReference(V)， 抛出一个 ReferenceError 异常。
4. 如果 IsPropertyReference(V)， 那么
 1. 如果 HasPrimitiveBase(V) 是 false， 那么令 get 为 base 的 [[Get]] 内部方法， 否则令 get 为下面定义的特殊s的 [[Get]] 内部方法。
 2. 将 base 作为 this 值， 传递 GetReferencedName(V) 为参数， 调用 get 内部方法， 返回结果。

5. 否则, base 必须是一个 environment record。
6. 传递 GetReferencedName(V) 和 IsStrictReference(V) 为参数调用 base 的 GetBindingValue(见 10.2.1) 具体方法, 返回结果。

GetValue 中的 V 是原始基值的 属性引用 时使用下面的 [[Get]] 内部方法。它用 base 作为他的 this 值, 其中属性 P 是它的参数。采用以下步骤:

1. 令 O 为 ToObject(base)。
2. 令 desc 为用属性名 P 调用 O 的 [[GetProperty]] 内部方法的返回值。
3. 如果 desc 是 undefined, 返回 undefined。
4. 如果 IsDataDescriptor(desc) 是 true, 返回 desc.[[Value]]。
5. 否则 IsAccessorDescriptor(desc) 必须是 true, 令 getter 为 desc.[[Get]]。
6. 如果 getter 是 undefined, 返回 undefined。
7. 提供 base 作为 this 值, 无参数形式调用 getter 的 [[Call]] 内部方法, 返回结果。

上述方法之外无法访问在第一步创建的对象。实现可以选择真的创建这个对象。使用这个内部方法给实际属性访问产生可见影响的情况只有在调用访问器函数时。

PutValue(v,w)

1. 如果 Type(V) 不是引用, 抛出一个 ReferenceError 异常。
2. 令 base 为调用 GetBase(V) 的结果。
3. 如果 IsUnresolvableReference(V), 那么
 1. 如果 IsStrictReference(V) 是 true, 那么
 - i. 抛出 ReferenceError 异常。
2. 用 GetReferencedName(V), W, false 作为参数调用全局对象的 [[Put]] 内部方法。
4. 否则如果 IsPropertyReference(V), 那么
 1. 如果 HasPrimitiveBase(V) 是 false, 那么令 put 为 base 的 [[Put]] 内部方法, 否则令 put 为下面定义的特殊的 [[Put]] 内部方法。
2. 用 base 作为 this 值, 用 GetReferencedName(V), W, IsStrictReference(V) 作为参数调用 put 内部方法。
5. 否则 base 必定是 environment record 作为 base 的引用。所以,
 1. 用 GetReferencedName(V), W, IsStrictReference(V) 作为参数调用 base 的 SetMutableBinding (10.2.1) 具体方法。
6. 返回。

PutValue 中的 V 是原始基值的属性引用时使用下面的 [[Put]] 内部方法。用 base 作为 this 值, 用属性 P, 值 W, 布尔标志 Throw 作为参数调用它。采用以下步骤:

1. 令 O 为 ToObject(base)。

2. 如果用 **P** 作为参数调用 **O** 的 **[[CanPut]]** 内部方法的结果是 **false**，那么
 1. 如果 **Throw** 是 **true**，那么抛出一个 **TypeError** 异常。
 2. 否则返回。
3. 令 **ownDesc** 为用 **P** 作为参数调用 **O** 的 **[[GetOwnProperty]]** 内部方法的结果。
4. 如果 **IsDataDescriptor(ownDesc)** 是 **true**，那么
 1. 如果 **Throw** 是 **true**，那么抛出一个 **TypeError** 异常。
 2. 否则返回。
5. 令 **desc** 为用 **P** 作为参数调用 **O** 的 **[[GetProperty]]** 内部方法的结果。这可能是一个自身或继承的访问器属性描述或是一个继承的数据属性描述。
6. 如果 **IsAccessorDescriptor(desc)** 是 **true**，那么
 1. 令 **setter** 为 **desc.Set**，他不能是 **undefined**。
 2. 用 **base** 作为 **this** 值，用只由 **W** 组成的列表作为参数调用 **setter** 的 **[[Call]]** 内部方法。
7. 否则，这是要在临时对象 **O** 上创建自身属性的请求。
 1. 如果 **Throw** 是 **true**，抛出一个 **TypeErroe** 异常。
 8. 返回。

上述方法之外无法访问在第一步创建的对象。实现可以选择不真的创建这个临时对象。使用这个内部方法给实际属性访问产生可见影响的情况只有在调用访问器函数时，或 **Throw** 未通过提前错误检查。当 **Throw** 是 **true**，试图在这个临时对象上创建新属性的任何属性分配操作会抛出一个错误。

列表规范类型

列表类型用于说明 **new** 表达式，函数调用，其他需要值的简单列表的算法 -- 里的参数列表的计算。列表类型的值是简单排序的一些值的序列。此序列可以是任意长度。

完结规范类型

完结类型用于说明执行将控制转移到外部的声明 (**break**, **continue**, **return**, **throw**) 的行为。完结类型的值是由三部分组成，形如 (**type**, **value**, **target**)，其中 **type** 是 **normal**, **break**, **continue**, **return**, **throw** 之一，**value** 是任何 ECMAScript 语言值或 **empty**，**target** 是任何 ECMAScript 标识符或 **empty**。

术语“突然完结 (**abrupt completion**)”是指任何非正常完成的完成类型。

属性描述符及属性标识符规范类型

属性描述符类型是用来解释命名属性的具体的操作的特性集。属性描述符类型的值是记录项，由命名字段组成，每个字段的名称是一个特性名并且它的值是一个相应的特性值，这些特性指定在 8.6.1。此外，任何字段都可能存在或不存在。

根据是否存在或使用了某些字段，属性描述符的值可进一步划分为数据属性描述符和访问器属性描述符。一个数据属性描述符里包括叫做 `[[Value]]` 或 `[[Writable]]` 的字段。一个访问器属性描述符里包括叫做 `[[Get]]` 或 `[[Set]]` 的字段。任何属性描述都可能名为 `[[Enumerable]]` 和 `[[Configurable]]` 的字段。一个属性描述符不能同时是数据属性描述符和访问器属性描述符；但是，它可能二者都不是。一个通用属性描述符是，既不是数据属性描述符也不是访问器属性描述符的属性描述符值。一个完全填充属性描述符是访问器属性描述符或数据属性描述符，并且拥有 8.6.1 Table 5 或 Table 6 里定义的所有属性特性对应的字段。

本规范中为了便于标记，使用一种类似对象字面量的语法来定义属性描述符。例如，属性描述符 `{[[Value]]: 42, [[Writable]]: false, [[Configurable]]: true}`，就定义了一个数据属性描述符。字段名称的顺序并不重要。任何没有明确列出的字段被认为是不存在的。

在规范中的文本和算法里，可用点符号来指明一个属性描述符的特定字段。例如，如果 `D` 是一个属性描述符，那么 `D. [[Value]]` 是“`D` 的 `[[Value]]` 字段”的简写。

属性标识符类型用于关联属性名称与属性描述符。属性标识符类型的值是 `(name, descriptor)` 形式的一对值，其中 `name` 是一个字符串和 `descriptor` 是一个属性描述符值。

在本规范中使用以下的抽象操作来操作属性描述符值：

IsAccessorDescriptor (Desc)

当用属性描述 `Desc` 调用抽象操作 `IsAccessorDescriptor`，采用以下步骤：

1. 如果 `Desc` 是 `undefined`，那么返回 `false`。
2. 如果 `Desc. [[Get]]` 和 `Desc. [[Set]]` 都不存在，则返回 `false`。
3. 返回 `true`。

IsDataDescriptor (Desc)

当用属性描述 `Desc` 调用抽象操作 `IsDataDescriptor`，采用以下步骤：

1. 如果 `Desc` 是 `undefined`，那么返回 `false`。
2. 如果 `Desc. [[Value]]` 和 `Desc. [[Writable]]` 都不存在，则返回 `false`。
3. 返回 `true`。

IsGenericDescriptor (Desc)

当用属性描述 Desc 调用抽象操作 IsDataDescriptor, 采用以下步骤:

1. 如果 Desc 是 undefined, 那么返回 false。
2. 如果 IsAccessorDescriptor(Desc) 和 IsDataDescriptor(Desc) 都是 false, 则返回 true。
3. 返回 false。

FromPropertyDescriptor (Desc)

当用属性描述 Desc 调用抽象操作 FromPropertyDescriptor, 采用以下步骤:

假定以下算法的 Desc 是 `[[GetOwnProperty]]`(见 8.12.1) 返回的完全填充的属性描述。

1. 如果 Desc 是 undefined, 那么返回 false。
2. 令 obj 为仿佛使用 `new Object()` 表达式创建的新对象, 这里的 Object 是标准内置构造器名。
3. 如果 IsDataDescriptor(Desc) 是 true, 则
 1. 用参数 "value", 属性描述符 `{[[Value]]: Desc. [[Value]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, false 调用 obj 的 `[[DefineOwnProperty]]` 内部方法。
 2. 用参数 "writable", 属性描述符 `{[[Value]]: Desc. [[Writable]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, false 调用 obj 的 `[[DefineOwnProperty]]` 内部方法。
4. 否则, IsAccessorDescriptor(Desc) 必定是 true, 所以
 1. 用参数 "get", 属性描述符 `{[[Value]]: Desc. [[Get]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, false 调用 obj 的 `[[DefineOwnProperty]]` 内部方法。
 2. 用参数 "set", 属性描述符 `{[[Value]]: Desc. [[Set]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, false 调用 obj 的 `[[DefineOwnProperty]]` 内部方法。
5. 用参数 "enumerable", 属性描述符 `{[[Value]]: Desc. [[Enumerable]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, false 调用 obj 的 `[[DefineOwnProperty]]` 内部方法。
6. 用参数 "configurable", 属性描述符 `{[[Value]]: Desc. [[Configurable]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, false 调用 obj 的 `[[DefineOwnProperty]]` 内部方法。
7. 返回 obj。

ToPropertyDescriptor (Obj)

当用对象 Desc 调用抽象操作 FromPropertyDescriptor, 采用以下步骤:

1. 如果 Type(Obj) 不是 Object, 抛出一个 TypeError 异常。
2. 令 desc 为创建初始不包含字段的新属性描述的结果。
3. 如果用参数 "enumerable" 调用 Obj 的 [[HasProperty]] 内部方法的结果是 true, 则
 1. 令 enum 为用参数 "enumerable" 调用 Obj 的 [[Get]] 内部方法的结果。
 2. 设定 desc 的 [[Enumerable]] 字段为 ToBoolean(enum)。
4. 如果参数 "configurable" 调用 Obj 的 [[HasProperty]] 内部方法的结果是 true, 则
 1. 令 conf 为用参数 "enumerable" 调用 Obj 的 [[Get]] 内部方法的结果。
 2. 设定 desc 的 [[Configurable]] 字段为 ToBoolean(conf)。
5. 如果参数 "value" 调用 Obj 的 [[HasProperty]] 内部方法的结果是 true, 则
 1. 令 value 为用参数 "value" 调用 Obj 的 [[Get]] 内部方法的结果。
 2. 设定 desc 的 [[Value]] 字段为 value。
6. 如果参数 "writable" 调用 Obj 的 [[HasProperty]] 内部方法的结果是 true, 则
 1. 令 writable 为用参数 "writable" 调用 Obj 的 [[Get]] 内部方法的结果。
 2. 设定 desc 的 [[Writable]] 字段为 ToBoolean(writable)。
7. 如果参数 "get" 调用 Obj 的 [[HasProperty]] 内部方法的结果是 true, 则
 1. 令 getter 为用参数 "get" 调用 Obj 的 [[Get]] 内部方法的结果。
 2. 如果 IsCallable(getter) 是 false 并且 getter 不是 undefined, 则抛出一个 TypeError 异常。
 3. 设定 desc 的 [[Get]] 字段为 getter。
8. 如果参数 "set" 调用 Obj 的 [[HasProperty]] 内部方法的结果是 true, 则
 1. 令 setter 为用参数 "set" 调用 Obj 的 [[Get]] 内部方法的结果。
 2. 如果 IsCallable(setter) 是 false 并且 setter 不是 undefined, 则抛出一个 TypeError 异常。
 3. 设定 desc 的 [[Set]] 字段为 Setter。
9. 如果存在 desc. [[Get]] 或 desc. [[Set]], 则
 1. 如果存在 desc. [[Value]] 或 desc. [[Writable]], 则抛出一个 TypeError 异常。
10. 返回 desc

词法环境和环境记录项规范类型

词法环境和环境记录项类型用于说明在嵌套的函数或块中的名称解析行为。这些类型和他们的操作定义在第 10 章。

对象内部方法的算法

在以下算法说明中假定 **O** 是一个原生 ECMAScript 对象，**P** 是一个字符串，**Desc** 是一个属性说明记录，**Throw** 是一个布尔标志。

[[GetOwnProperty]](P)

当用属性名 **P** 调用 **O** 的 [[GetOwnProperty]] 内部方法，采用以下步骤：

1. 如果 **O** 不包含名为 **P** 的自身属性，返回 **undefined**。
2. 令 **D** 为无字段的新建属性描述。
3. 令 **X** 为 **O** 的名为 **P** 的自身属性。
4. 如果 **X** 是数据属性，则
 1. 设定 **D**.[[Value]] 为 **X** 的 [[Value]] 特性值。
 2. 设定 **D**.[[Writable]] 为 **X** 的 [[Writable]] 特性值。
5. 否则 **X** 是访问器属性，所以
 1. 设定 **D**.[[Get]] 为 **X** 的 [[Get]] 特性值。
 2. 设定 **D**.[[Set]] 为 **X** 的 [[Set]] 特性值。
6. 设定 **D**.[[Enumerable]] 为 **X** 的 [[Enumerable]] 特性值。
7. 设定 **D**.[[Configurable]] 为 **X** 的 [[Configurable]] 特性值。
8. 返回 **D**。

然而，如果 **O** 是一个字符串对象，关于其 [[GetOwnProperty]] 的更多阐述定义在 15.5.5.2。

[[GetProperty]] (P)

当用属性名 **P** 调用 **O** 的 [[GetProperty]] 内部方法，采用以下步骤：

1. 令 **prop** 为用属性名 **P** 调用 **O** 的 [[GetOwnProperty]] 内部方法的结果。
2. 如果 **prop** 不是 **undefined**，返回 **prop**。
3. 令 **proto** 为 **O** 的 [[Prototype]] 内部属性值。
4. 如果 **proto** 是 **null**，返回 **undefined**。
5. 用参数 **P** 调用 **proto** 的 [[GetProperty]] 内部方法，返回结果。

[[Get]] (P)

当用属性名 **P** 调用 **O** 的 [[Get]] 内部方法，采用以下步骤：

1. 令 **desc** 为用属性名 **P** 调用 **O** 的 [[GetProperty]] 内部方法的结果。
2. 如果 **desc** 是 **undefined**，返回 **undefined**。
3. 如果 **IsDataDescriptor(desc)** 是 **true**，返回 **desc**.[[Value]]。
4. 否则，**IsAccessorDescriptor(desc)** 必定是真，所以，令 **getter** 为 **desc**.[[Get]]。

5. 如果 `getter` 是 `undefined`，返回 `undefined`。
6. 用 `O` 作为 `this`，无参数调用 `getter` 的 `[[Call]]` 内部方法，返回结果。

`[[CanPut]] (P)`

当用属性名 `P` 调用 `O` 的 `[[CanPut]]` 内部方法，采用以下步骤：

1. 令 `desc` 为用参数 `P` 调用 `O` 的 `[[GetOwnProperty]]` 内部方法的结果。
2. 如果 `desc` 不是 `undefined`，则
 1. 如果 `IsAccessorDescriptor(desc)` 是 `true`，则
 - i. 如果 `desc[[Set]]` 是 `undefined`，则返回 `false`。
 - ii. 否则返回 `true`。
 2. 否则，`desc` 必定是 `DataDescriptor`，所以返回 `desc[[Writable]]` 的值。
3. 令 `proto` 为 `O` 的 `[[Prototype]]` 内部属性。
4. 如果 `proto` 是 `null`，则返回 `O` 的 `[[Extensible]]` 内部属性的值。
5. 令 `inherited` 为用属性名 `P` 调用 `proto` 的 `[[GetProperty]]` 内部方法的结果。
6. 如果 `inherited` 是 `undefined`，返回 `O` 的 `[[Extensible]]` 内部属性的值。
7. 如果 `IsAccessorDescriptor(inherited)` 是 `true`，则
 1. 如果 `inherited[[Set]]` 是 `undefined`，则返回 `false`。
 2. 否则返回 `true`。
8. 否则，`inherited` 必定是 `DataDescriptor`
 1. 如果 `O` 的 `[[Extensible]]` 内部属性是 `false`，返回 `false`。
 2. 否则返回 `inherited[[Writable]]` 的值。

宿主对象可以定义受额外约束的 `[[Put]]` 操作。如果可能，宿主对象不应该在 `[[CanPut]]` 返回 `false` 的情况下允许 `[[Put]]` 操作。

`[[Put]] (P, V, Throw)`

当用属性名 `P`，值 `V`，布尔值 `Throw` 调用 `O` 的 `[[Put]]` 内部方法，采用以下步骤：

1. 如果用参数 `P` 调用 `O` 的 `[[CanPut]]` 内部方法的结果是 `false`，则
 1. 如果 `Throw` 是 `true`，则抛出一个 `TypeError` 异常。
 2. 否则返回。
2. 令 `ownDesc` 为用参数 `P` 调用 `O` 的 `[[GetOwnProperty]]` 内部方法的结果。
3. 如果 `IsDataDescriptor(ownDesc)` 是 `true`，则
 1. 令 `valueDesc` 为属性描述 `{[[Value]]: V}`。
 2. 用参数 `P`，`valueDesc`，`Throw` 调用 `O` 的 `[[DefineOwnProperty]]` 内部方法。
3. 返回。

4. 令 desc 为用参数 P 调用 O 的 `[[GetProperty]]` 内部方法的结果。这可能是自身或继承的访问器属性描述或者是继承的数据属性描述。
5. 如果 `IsAccessorDescriptor(desc)` 是 true, 则
 1. 令 setter 为不是 undefined 的 desc.`[[Set]]`。
 2. 用 O 作为 this, V 作为唯一参数调用 setter 的 `[[Call]]` 内部方法。
6. 否则, 按照以下步骤在对象 O 上创建名为 P 的命名数据属性。
 1. 令 newDesc 为属性描述 `{[[Value]]: V, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`。
 2. 用参数 P, newDesc, Throw 调用 O 的 `[[DefineOwnProperty]]` 内部方法。
7. 返回。

`[[HasProperty]] (P)`

当用属性名 P 调用 O 的 `[[HasProperty]]` 内部方法, 采用以下步骤:

1. 令 desc 为用属性名 P 调用 O 的 `[[GetProperty]]` 内部方法的结果。
2. 如果 desc 是 undefined, 则返回 false。
3. 否则返回 true。

`[[Delete]] (P, Throw)`

当用属性名 P 和布尔值 Throw 调用 O 的 `[[Delete]]` 内部方法, 采用以下步骤:

1. 令 desc 为用属性名 P 调用 O 的 `[[GetOwnProperty]]` 内部方法的结果。
2. 如果 desc 是 undefined, 则返回 true。
3. 如果 desc.`[[Configurable]]` 是 true, 则
 1. 在 O 上删除名为 P 的自身属性。
 2. 返回 true。
4. 否则如果 Throw, 则抛出一个 `TypeError` 异常。
5. 返回 false。

`[[DefaultValue]] (hint)`

当用字符串 hint 调用 O 的 `[[DefaultValue]]` 内部方法, 采用以下步骤:

1. 令 toString 为用参数 "toString" 调用对象 O 的 `[[Get]]` 内部方法的结果。
2. 如果 `IsCallable(toString)` 是 true, 则
 1. 令 str 为用 O 作为 this 值, 空参数列表调用 toString 的 `[[Call]]` 内部方法的结果。
 2. 如果 str 是原始值, 返回 str。
3. 令 valueOf 为用参数 "valueOf" 调用对象 O 的 `[[Get]]` 内部方法的结果。

4. 如果 `IsCallable(valueOf)` 是 `true`, 则
 1. 令 `val` 为用 `O` 作为 `this` 值, 空参数列表调用 `valueOf` 的 `[[Call]]` 内部方法的结果。
 2. 如果 `val` 是原始值, 返回 `val`。
 5. 抛出一个 `TypeError` 异常。

当用数字 `hint` 调用 `O` 的 `[[DefaultValue]]` 内部方法, 采用以下步骤:

1. 令 `valueOf` 为用参数 `"valueOf"` 调用对象 `O` 的 `[[Get]]` 内部方法的结果。
2. 如果 `IsCallable(valueOf)` 是 `true`, 则
 1. 令 `val` 为用 `O` 作为 `this` 值, 空参数列表调用 `valueOf` 的 `[[Call]]` 内部方法的结果。
 2. 如果 `val` 是原始值, 返回 `val`。
3. 令 `toString` 为用参数 `"toString"` 调用对象 `O` 的 `[[Get]]` 内部方法的结果。
4. 如果 `IsCallable(toString)` 是 `true`, 则
 1. 令 `str` 为用 `O` 作为 `this` 值, 空参数列表调用 `toString` 的 `[[Call]]` 内部方法的结果。
 2. 如果 `str` 是原始值, 返回 `str`。
 5. 抛出一个 `TypeError` 异常。

当不用 `hint` 调用 `O` 的 `[[DefaultValue]]` 内部方法, `O` 是 `Date` 对象的情况下仿佛 `hint` 是字符串一样解释它的行为, 除此之外仿佛 `hint` 是数字一样解释它的行为。

上面说明的 `[[DefaultValue]]` 在原生对象中只能返回原始值。如果一个宿主对象实现了它自身的 `[[DefaultValue]]` 内部方法, 那么必须确保其 `[[DefaultValue]]` 内部方法只能返回原始值。

[[DefineOwnProperty]] (P, Desc, Throw)

在以下算法中, 术语“拒绝”指代“如果 `Throw` 是 `true`, 则抛出 `TypeError` 异常, 否则返回 `false`”。算法包含测试具体值的属性描述 `Desc` 的各种字段的步骤。这种方式测试的字段事实上不需要真的在 `Desc` 里。如果一个字段不存在则将其值看作是 `false`。

当用属性名 `P`, 属性描述 `Desc`, 布尔值 `Throw` 调用 `O` 的 `[[DefineOwnProperty]]` 内部方法, 采用以下步骤:

1. 令 `current` 为用属性名 `P` 调用 `O` 的 `[[GetOwnProperty]]` 内部属性的结果。
2. 令 `extensible` 为 `O` 的 `[[Extensible]]` 内部属性值。
3. 如果 `current` 是 `undefined` 并且 `extensible` 是 `false`, 则拒绝。
4. 如果 `current` 是 `undefined` 并且 `extensible` 是 `true`, 则
 1. 如果 `IsGenericDescriptor(Desc)` 或 `IsDataDescriptor(Desc)` 是 `true`, 则

- i. 在 O 上创建名为 P 的自身数据属性，Desc 描述了它的 [[Value]], [[Writable]], [[Enumerable]], [[Configurable]] 特性值。如果 Desc 的某特性字段值不存在，那么设定新建属性的此特性为默认值。
2. 否则，Desc 必定是访问器属性描述，所以
- i. 在 O 上创建名为 P 的自身访问器属性，Desc 描述了它的 [[Get]], [[Set]], [[Enumerable]], [[Configurable]] 特性值。如果 Desc 的某特性字段值不存在，那么设定新建属性的此特性为默认值。
5. 如果 Desc 不存在任何字段，返回 true。
6. 如果 Desc 的任何字段都出现在 current 中，并且用 SameValue 算法比较 Desc 中每个字段值和 current 里对应字段值，结果相同，则返回 true。
7. 如果 current 的 [[Configurable]] 字段是 false，则
 1. 如果 Desc 的 [[Configurable]] 字段是 true，则拒绝。
 2. 如果 Desc 有 [[Enumerable]] 字段，并且 current 和 Desc 的 [[Enumerable]] 字段相互布尔否定，则拒绝。
8. IsGenericDescriptor(Desc) 是 true，则不需要进一步验证。
9. 否则，如果 IsDataDescriptor(current) 和 IsDataDescriptor(Desc) 的结果不同，则
 1. 如果 current 的 [[Configurable]] 字段是 false，则拒绝。
 2. 如果 IsDataDescriptor(current) 是 true，则
- i. 将对象 O 的名为 P 的数据属性转换为访问器属性。保留转换属性的 [[Configurable]] 和 [[Enumerable]] 特性的现有值，并且设定属性的其余特性为其默认值。
3. 否则
- i. 将对象 O 的名为 P 的访问器属性转换为数据属性。保留转换属性的 [[Configurable]] 和 [[Enumerable]] 特性的现有值，并且设定属性的其余特性为其默认值。
10. 否则，如果 IsDataDescriptor(current) 和 IsDataDescriptor(Desc) 都是 true，则
 1. 如果 current 的 [[Configurable]] 字段是 false，则
- i. 如果 current 的 [[Writable]] 字段是 false 并且 Desc 的 [[Writable]] 字段是 true，则拒绝。
- ii. 如果 current 的 [[Writable]] 字段是 false，则
 1. 如果 Desc 有 [[Value]] 字段，并且 SameValue(Desc.[[Value]], current.[[Value]]) 是 false，则拒绝。
 2. 否则，current 的 [[Configurable]] 字段是 true，所以可接受任何更改。
11. 否则 IsAccessorDescriptor(current) 和 IsAccessorDescriptor(Desc) 都是 true，所以
 1. 如果 current 的 [[Configurable]] 字段是 false，则
- i. 如果 Desc 有 [[Set]] 字段，并且 SameValue(Desc.[[Set]], current.[[Set]]) 是 false，则拒绝。
- ii. 如果 Desc 有 [[Get]] 字段，并且 SameValue(Desc.[[Set]], current.[[Get]]) 是 false，则拒绝。
12. Desc 拥有所有特性字段，设定对象 O 的名为 P 的属性的对性特性为这些字段值。

13. 返回 true。

然而，如果 O 是一个 Array 对象，其 `[[DefineOwnProperty]]` 内部方法的更多阐述定义在 15.4.5.1

如果 current 的 `[[Configurable]]` 字段是 true，那么步骤 10.b 允许 Desc 的任何字段与 current 对应的字段不同。这甚至可以改变 `[[Writable]]` 特性为 false 的属性的 `[[Value]]`。允许这种情况是因为值是 true 的 `[[Configurable]]` 特性会允许按照：首先设定 `[[Writable]]` 为 true，然后设定新 `[[Value]]`，`[[Writable]]` 设为 false 的顺序调用。

类型转换与测试

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 — 它们可以接受任何 ECMAScript 语言类型 的值，但是不接受 规范类型 。

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：

ToPrimitive 转换

输入类型	结果
Undefined	结果等于输入的参数（不转换）。
Null	结果等于输入的参数（不转换）。
Boolean	结果等于输入的参数（不转换）。
Number	结果等于输入的参数（不转换）。
String	结果等于输入的参数（不转换）。
Object	返回该对象的默认值。（调用该对象的内部方法 <code>[[DefaultValue]]</code> 一样）。

ToBoolean

ToBoolean 运算符根据下表将其参数转换为布尔值类型的值：

ToBoolean 转换

输入类型	结果
Undefined	false

Null	false
Boolean	结果等于输入的参数（不转换）。
Number	如果参数是 +0, -0, 或 NaN, 结果为 false ; 否则结果为 true。
String	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。
Object	true

ToNumber

ToNumber 运算符根据下表将其参数转换为数值类型的值：

ToNumber 转换

输入类型	结果
Undefined	NaN
Null	+0
Boolean	如果参数是 true, 结果为 1。如果参数是 false, 此结果为 +0。
Number	结果等于输入的参数（不转换）。
String	参见下文的文法和注释。
Object	应用下列步骤： . 設 原始值 為 ToPrimitive(輸入参数 , 暗示 数值类型)。 . 返回 ToNumber(原始值)。

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。

语法

```
StringNumericLiteral :::
```

```
StrWhiteSpaceopt
```

```
StrWhiteSpaceoptStrNumericLiteral
```

```
StrWhiteSpaceoptStrWhiteSpace :::
```

```
StrWhiteSpaceChar StrWhiteSpaceoptStrWhiteSpaceChar :::
```

```
WhiteSpace
```

```
LineTerminatorStrNumericLiteral :::
```

```

StrDecimalLiteral
HexIntegerLiteralStrDecimalLiteral :::

StrUnsignedDecimalLiteral
+ StrUnsignedDecimalLiteral
- StrUnsignedDecimalLiteralStrUnsignedDecimalLiteral :::

Infinity

DecimalDigits . DecimalDigitsopt ExponentPartopt
. DecimalDigits ExponentPartopt

DecimalDigits ExponentPartoptDecimalDigits :::

DecimalDigit

DecimalDigits DecimalDigit DecimalDigit ::: 以下之一
0 1 2 3 4 5 6 7 8 9ExponentPart :::

ExponentIndicator SignedInteger ExponentIndicator ::: 以
下之一

e ESignedInteger :::

DecimalDigits
+ DecimalDigits
- DecimalDigitsHexIntegerLiteral :::

0x HexDigit

0X HexDigit

HexIntegerLiteral HexDigit HexDigit ::: 以下之一
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：

- 「字符串数值常量」之前和、或之后可以有空白和 / 或行结束符。
- 十进制的「字符串数值常量」可有任意位数的 0 在前头。
- 十进制的「字符串数值常量」可有指示其符号的 + 或 - 前缀。
- 空的，或只包含空白的「字符串值常量」会被转换为 +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，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；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 $::: [\text{empty}]$ 」的数学值是 0。
- 「字符串整数常量 $::: \text{串空白}$ 」的数学值是 0。
- 不管有没有空白「字符串整数常量 $::: \text{串空白}_{\text{opt}} \text{串数值常量} \text{串空白}_{\text{opt}}$ 」的数学值是「串数值常量」的数学值
- 「串数值常量 $::: \text{串十进制常量}$ 」的数学值是「串十进制常量」的数学值
- 「串数值常量 $::: \text{十六进制整数常量}$ 」的数学值是「十六进制整数常量」的数学值
- 「串十进制常量 $::: \text{串无符号整数常量}$ 」的数学值是「串无符号整数常量」的数学值
- 「串十进制常量 $::: + \text{串无符号整数常量}$ 」的数学值是「串无符号整数常量」的数学值。
- 「串十进制常量 $::: - \text{串无符号整数常量}$ 」的数学值是「串无符号整数常量」的数学值的负数。（需要注意的是，如果「串无符号整数常量」的数学值是 0，其负数也是 0。下面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 $::: \text{Infinity}$ 」的数学值是 10^{10000} （一个大到会舍入为 $+\infty$ 的值过大的值会返回为 $+$ ）。
- 「串无符号整数常量 $::: \text{十进制数} .$ 」的数学值是「十进制数」的数学值。
- 「串无符号整数常量 $::: \text{十进制数} . \text{十进制数}$ 」的数学值是第一个「十进制数」的数学值加（第二个「十进制数」的数学值乘以 10^{-n} ），这里的 n 是 **the number of characters in the** 第二个「十进制数」字符数。
- 「串无符号整数常量 $::: \text{十进制数} . \text{指数部分}$ 」的数学值是「十进制数」的数学值乘以 10^e ，这里的 e 是「指数部分」的数学值
- 「串无符号整数常量 $::: \text{十进制数} . \text{十进制数} \text{指数部分}$ 」的数学值是（第一个「十进制数」的数学值加（第二个「十进制数」的数学值乘以 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」的数学值。
- 「串无符号整数常量 $::: . \text{十进制数}$ 」的数学值是「十进制数」的数学值乘以 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 $::: . \text{十进制数} \text{指数部分}$ 」的数学值是「十进制数」的数学值乘以 10^{-en} ，这里的 n 是「十进制数」中的字符个数， e 是「指数部分」的数学值
- 「串无符号整数常量 $::: \text{十进制数}$ 」的数学值是「十进制数」的数学值
- 「串无符号整数常量 $::: \text{十进制数} \text{指数部分}$ 」的数学值是「十进制数」的数学值乘以 10^e ，这里的 e 是「指数部分」的数学值
- 「十进制数 $::: \text{十进制数字}$ 」是「十进制数字」的数学值
- 「十进制数 $::: \text{十进制数} \text{十进制数字}$ 」的数学值是（「十进制数」的数学值乘以 10）加「十进制数字」的数学值

- 「指数部分 :: 幂指示符 有符号整数」的数学值是「有符号整数」的数学值
- 「有符号整数 :: 十进制数」的数学值是「十进制数」的数学值
- 「有符号整数 :: + 十进制数」的数学值是「十进制数」的数学值
- 「有符号整数 :: - 十进制数」是「十进制数」的数学值的负数。
- 「十进制数字 :: 0」或「十六进制数字 :: 0」的数学值是 0。
- 「十进制数字 :: 1」或「十六进制数字 :: 1」的数学值是 1。
- 「十进制数字 :: 2」或「十六进制数字 :: 2」的数学值是 2。
- 「十进制数字 :: 3」或「十六进制数字 :: 3」的数学值是 3。
- 「十进制数字 :: 4」或「十六进制数字 :: 4」的数学值是 4。
- 「十进制数字 :: 5」或「十六进制数字 :: 5」的数学值是 5。
- 「十进制数字 :: 6」或「十六进制数字 :: 6」的数学值是 6。
- 「十进制数字 :: 7」或「十六进制数字 :: 7」的数学值是 7。
- 「十进制数字 :: 8」或「十六进制数字 :: 8」的数学值是 8。
- 「十进制数字 :: 9」或「十六进制数字 :: 9」的数学值是 9。
- 「十六进制数字 :: a」或「十六进制数字 :: A」的数学值是 10。
- 「十六进制数字 :: b」或「十六进制数字 :: B」的数学值是 11。
- 「十六进制数字 :: c」或「十六进制数字 :: C」的数学值是 12。
- 「十六进制数字 :: d」或「十六进制数字 :: D」的数学值是 13。
- 「十六进制数字 :: e」或「十六进制数字 :: E」的数学值是 14。
- 「十六进制数字 :: f」或「十六进制数字 :: F」的数学值是 15。
- 「十六进制整数常量 :: 0x 十六进制数字」的数学值是「十六进制数字」的数学值。
- 「十六进制整数常量 :: 0X 十六进制数字」的数学值是「十六进制数字」的数学值。
- 「十六进制整数常量 :: 十六进制整数常量 十六进制数字」的数学值是（「十六进制整数常量」的数学值乘以 16）加「十六进制数字」的数学值。

一旦字符串数值常量的数学值被精确地确定，接下来就会被舍入为数值类型的一个值。如果数学值是 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，舍入值必须是数学值的 数字值，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 重要数字 — 在这种情况下，此数字的值是下面两种之一：一是将其 20 位之后的每个重要数字用 0 替换，产生此字符串解析出的数学值的数字值；二是将其 20 位之后的每个有效数字用 0 替换，并在第 20 位重要数字加一，产生此字符串解析出的数学值的数字值 。判断一个数字是否为 重要数字，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。

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 $\text{sign}(\text{Result}(1)) * \text{floor}(\text{abs}(\text{Result}(1)))$ 。
5. 返回 Result(4)。

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 $\text{sign}(\text{Result}(1)) * \text{floor}(\text{abs}(\text{Result}(1)))$ 。
4. 计算 $\text{Result}(3) \bmod 2^{32}$ ；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 $\text{Result}(4) - 2^{32}$ ，否则返回 Result(4)。

按照 ToInt32 以上的定义：ToInt32 抽象操作是幂等的：如果作用于它产生的结果，第二次程序将会保持值不变，对任何值 x ， $\text{ToInt32}(\text{ToUint32}(x))$ 都与 $\text{ToInt32}(x)$ 相等。（将 +∞ 和 -∞ 映射到 +0 就是为了保持这一特性。） ToInt32 将 -0 映射为 +0。

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 $\text{sign}(\text{Result}(1)) * \text{floor}(\text{abs}(\text{Result}(1)))$ 。
4. 计算 $\text{Result}(3) \bmod 2^{32}$ ；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)。

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 $+\infty$ 和 $-\infty$ 被映射为 $+0$ 。）
- `ToUint32` 把 -0 映射为 $+0$ 。

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 ， $+\infty$ ，或 $-\infty$ ，返回 $+0$ 。
3. 计算 `sign(Result(1)) * floor(abs(Result(1)))`。
4. 计算 `Result(3) modulo 216`；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)`。

上面给出的 `ToUint16` 的定义中：

- `ToUint32` 和 `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 $+0$ 。

ToString

`ToString` 运算符根据下表将其参数转换为字符串类型的值：

`ToString` 转换

输入类型	结果
Undefined	"undefined"
Null	"null"
Boolean	如果参数是 <code>true</code> ，那么结果为 <code>"true"</code> 。 如果参数是 <code>false</code> ，那么结果为 <code>"false"</code> 。
Number	结果等于输入的参数（不转换）。
String	参见下文的文法和注释。
Object	应用下列步骤： <ul style="list-style-type: none"> • 调用 <code>ToPrimitive(输入参数 , 暗示 字符串类型)</code>。 • 调用 <code>ToString(Result(1))</code>。 • 返回 <code>Result(2)</code>。

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 $k \geq 1, 10^{k-1} \leq s < 10^k, s \times 10^{n-k}$ 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 $k \leq n \leq 21$ ，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 $0 < n \leq 21$ ，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 $k-n$ 个 s 在十进制表示中的数字。
8. 如果 $-6 < n \leq 0$ ，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 $\text{abs}(n-1)$ 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 $\text{abs}(n-1)$ 的十进制表示（没有前置的零）。

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 $\text{ToNumber}(\text{ToString}(x))$ 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 $k \geq 1, 10^{k-1} \leq s < 10^k, s \times 10^{n-k}$ 的数字值是 m ，且 k 足够小。如果有数倍于 s 的可能性，选择 $s \times 10^{n-k}$ 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。

ToObject

ToObject 运算符根据下表将其参数转换为对象类型的值：

ToObject 转换

输入类型	结果
Undefined	抛出 TypeError 异常。
Null	>抛出 TypeError 异常。
Boolean	创建一个新的 Boolean 对象，其 [[PrimitiveValue]] 属性被设为该布尔值的值。
Number	创建一个新的 Number 对象，其 [[PrimitiveValue]] 属性被设为该布尔值的值。
String	创建一个新的 String 对象，其 [[PrimitiveValue]] 属性被设为该布尔值的值。
Object	结果是输入的参数（不转换）。

CheckObjectCoercible

根据表 15 定义，抽象操作 CheckObjectCoercible 在其参数无法用 ToObject 转换成对象的情况下抛出一个异常：

CheckObjectCoercible 结果

输入类型	结果
Undefined	抛出一个 TypeError 异常
Null	>抛出 TypeError 异常。
Boolean	返回
Number	返回
String	返回
Object	返回

IsCallable

根据表 16，抽象操作 IsCallable 确定其必须是 ECMAScript 语言值的参数是否是一个可调用对象：

IsCallable 结果

输入类型	结果
Undefined	返回 false
Null	>返回 false
Boolean	返回 false
Number	返回 false
String	返回 false
Object	如果参数对象包含一个 Call 内部方法，则返回 true，否则返回 false

SameValue 算法

内部严格比较操作 SameValue(x,y)，x 和 y 为 ECMAScript 语言中的值，需要产出 true 或 false 📄。比较过程如下：

1. 如果 Type(x) 与 Type(y) 的结果不一致，返回 false，否则
2. 如果 Type(x) 结果为 Undefined，返回 true
3. 如果 Type(x) 结果为 Null，返回 true
4. 如果 Type(x) 结果为 Number，则
 1. 如果 x 为 NaN，且 y 也为 NaN，返回 true
 2. 如果 x 为 +0，y 为 -0，返回 false
 3. 如果 x 为 -0，y 为 +0，返回 false
 4. 如果 x 与 y 为同一个数字，返回 true
 5. 返回 false
5. 如果 Type(x) 结果为 String，如果 x 与 y 为完全相同的字符序列（相同的长度和相同的字符对应相同的位置），返回 true，否则，返回 false
6. 如果 Type(x) 结果为 Boolean，如果 x 与 y 都为 true 或 false，则返回 true，否则，返回 false
7. 如果 x 和 y 引用到同一个 Object 对象，返回 true，否则，返回 false

可执行代码与执行环境

可执行代码类型

一共有 3 种 ECMA 脚本可执行代码：

- 全局代码 是指被作为 ECMA 脚本 程序 处理的源代码文本。一个特定 程序 的全局代码不包括作为 函数体 被解析的源代码文本。
- Eval 代码 是指提供给 eval 内置函数的源代码文本。更精确地说，如果传递给 eval 内置函数的参数为一个字符串，该字符串将被作为 ECMA 脚本

程序 进行处理。在特定的一次对 `eval` 的调用过程中, `eval` 代码作为该 程序 的 `#global-code` 部分。

- 函数代码 是指作为 函数体 被解析的源代码文本。一个 函数体 的 函数代码 不包括作为其嵌套函数的 函数体 被解析的源代码文本。函数代码 同时还特指 以构造器方式调用 `Function` 内置对象 时所提供的源代码文本。更精确地说, 调用 `Function` 构造器时传递的最后一个参数将被转换为字符串并作为函数体 使用。如果调用 `Function` 构造器时, 传递了一个以上的参数, 除最后一个参数以外的其他参数都将转换为字符串, 并以逗号作为分隔符连接在一起成为一个字符串, 该字符串被解析为形参列表 供由最后一个参数定义的 函数体 使用。初始化 `Function` 对象时所提供的函数代码, 并不包括作为其嵌套函数的 函数体被解析的源代码文本。

严格模式下的代码

一个 `ECMA` 脚本程序的语法单元可以使用非严格或严格模式下的语法及语义进行处理。当使用严格模式进行处理时, 以上三种代码将被称为严格全局代码、严格 `eval` 代码和严格函数代码。当符合以下条件时, 代码将被解析为严格模式下的代码:

- 当 全局代码 以指令序言开始, 且该指令序言包含一个使用严格模式的指令序言 (参考 14.1 章) 时, 即为严格全局代码。
- 当 全局代码 以指令序言开始, 且该指令序言包含一个使用严格模式的指令序言时; 或者在 严格模式下的代码 中通过直接调用 `eval` 函数 (参考 15.1.2.1.1 章) 时, 即为严格 `eval` 代码。
- 当一个 函数声明 、 函数表达式 或 函数赋值 访问器处在一段 严格模式下的代码 中, 或其函数代码以指令序言开始, 且该指令序言包含一个使用严格模式的指令序言时, 该函数代码即为严格函数代码。
- 当调用内置的 `Function` 构造器时, 如果最后一个参数所表达的字符串在作为 函数体 处理时以指令序言开始, 且该指令序言包含一个使用严格模式的指令序言, 则该函数代码即为严格函数代码。

词法环境

词法环境 是一个用于定义特定变量和函数标识符在 `ECMAScript` 代码的词法嵌套结构上关联关系的规范类型。一个词法环境由一个环境记录项和可能为空的外部词法环境引用构成。通常词法环境会与特定的 `ECMAScript` 代码诸如 `FunctionDeclaration`, `WithStatement` 或者 `TryStatement` 的 `Catch` 块这样的语法结构相联系, 且类似代码每次执行都会有一个新的语法环境被创建出来。

环境记录项记录了在它的关联词法环境域内创建的标识符绑定情形。

外部词法环境引用用于表示词法环境的逻辑嵌套关系模型。(内部)词法环境的外部引用是逻辑上包含内部词法环境的词法环境。外部词法环境自然也可能有多

个内部词法环境。例如，如果一个 **FunctionDeclaration** 包含两个嵌套的 **FunctionDeclaration**，那么 每个内嵌函数的词法环境都是外部函数本次执行所产生的词法环境。

词法环境和环境记录项是纯粹的规范机制，而不需要 **ECMAScript** 的实现保持一致。**ECMAScript** 程序不可能直接访问或者更改这些值。

环境记录项

在本标准中，共有 2 类环境记录项：声明式环境记录项 和 对象式环境记录项 。声明式环境记录项用于定义那些将 标识符 与语言值直接绑定的 **ECMA** 脚本语法元素，例如 函数定义 ， 变量定义 以及 **Catch** 语句。对象式环境记录项用于定义那些将标识符 与具体对象的属性绑定的 **ECMA** 脚本元素，例如 程序 以及 **With** 表达式 。

出于标准规范的目的，可以将环境记录项理解为面向对象中的一个简单继承结构，其中环境记录项是一个抽象类花前月下 有 2 个具体实现类，分别为声明式环境记录项和对象式环境记录项。抽象类包含了表 17 所描述的抽象方法定义，针对每一个具体实现类，每个抽象方法都有不同的具体算法。

环境记录项的抽象方法

方法	作用
HasBinding(N)	判断环境记录项是否包含对某个标识符的绑定。如果包含该绑定则返回 true ，反之返回 false 。其中字符串 N 是标识符文本。
CreateMutableBinding(N, D)	在环境记录项中创建一个新的可变绑定。其中字符串 N 指定绑定名称。如果可选参数 D 的值为 true ，则该绑定在后续操作中可以被删除。
SetMutableBinding(N,V, S)	在环境记录项中设置一个已经存在的绑定的值。其中字符串 N 指定绑定名称。 V 用于指定绑定的值，可以是任何 ECMA 脚本语言的类型。 S 是一个布尔类型的标记，当 S 为 true 并且该绑定不允许赋值时，则抛出一个 TypeError 异常。 S 用于指定是否为严格模式。
GetBindingValue(N,S)	返回环境记录项中一个已经存在的绑定的值。其中字符串 N 指定绑定的名称。 S 用于指定是否为严格模式。如果 S 的值为 true 并且该绑定不存在或未初始化，则抛出一个 ReferenceError 异常。
DeleteBinding(N)	从环境记录项中删除一个绑定。其中字符串 N 指定绑定的名称。如果 N 指定的绑定存在，将其删除并返回 true 。如果绑定存在但无法删除则返回 false 。如果绑定不存在则返回 true 。

<code>ImplicitThisValue()</code>	当从该环境记录项的绑定中获取一个函数对象并且调用时，该方法返回该函数对象使用的 <code>this</code> 对象的值。
----------------------------------	---

声明式环境记录项

每个声明式环境记录项都与一个包含变量和（或）函数声明的 **ECMA** 脚本的程序作用域相关联。声明式环境记录项用于绑定作用域内定义的一系列标识符。

除了所有环境记录项都支持的可变绑定外，声明式环境记录项还提供不可变绑定。在不可变绑定中，一个标识符与它的值之间的关联关系建立之后，就无法改变。创建和初始化不可变绑定是两个独立的过程，因此类似的绑定可以处在已初始化阶段或者未初始化阶段。除了环境记录项定义的抽象方法外，声明式环境记录项还支持表 18 中列出的方法：

声明式环境记录项的额外方法

方法	作用
<code>CreateImmutableBinding(N)</code>	在环境记录项中创建一个未初始化的不可变绑定。其中字符串 <code>N</code> 指定绑定名称。
<code>InitializeImmutableBinding(N,V)</code>	在环境记录项中设置一个已经创建但未初始化的不可变绑定的值。其中字符串 <code>N</code> 指定绑定名称。 <code>V</code> 用于指定绑定的值，可以是任何 ECMA 脚本语言的类型。

环境记录项定义的方法的具体行为将由以下算法给予描述。

HasBinding (N)

声明式环境记录项的 `HasBinding` 具体方法用于简单地判断作为参数的标识符是否是当前对象绑定的标识符之一：

1. 令 `envRec` 为函数调用时对应的声明式环境记录项。
2. 如果 `envRec` 有一个名称为 `N` 的绑定，返回 `true`。
3. 如果没有该绑定，返回 `false`。

CreateMutableBinding (N, D)

声明式环境记录项的 `CreateMutableBinding` 具体方法会创建一个名称为 `N` 的绑定，并初始化其值为 `undefined`。方法调用时，当前环境记录项中不能存在 `N` 的绑定。如果调用时提供了布尔类型的参数 `D` 且其值为 `true`，则新建的绑定被标记为可删除。

1. 令 `envRec` 为函数调用时对应的声明式环境记录项。

2. 执行断言: `envRec` 没有 `N` 的绑定。
3. 在 `envRec` 中为 `N` 创建一个可变绑定, 并将绑定的值设置为 `undefined`。
如果 `D` 为 `true` 则新创建的绑定可在后续操作中通过调用 `DeleteBinding` 删除。

SetMutableBinding (N,V,S)

声明式环境记录项的 `SetMutableBinding` 具体方法尝试将当前名称为参数 `N` 的绑定的值修改为参数 `V` 指定的值。方法调用时, 必须存在 `N` 的绑定。如果该绑定为不可变绑定, 并且 `S` 的值为 `true`, 则抛出一个 `TypeError` 异常。

1. 令 `envRec` 为函数调用时对应的声明式环境记录项。
2. 执行断言: `envRec` 必须有 `N` 的绑定。
3. 如果 `envRec` 中 `N` 的绑定为可变绑定, 则将其值修改为 `V`。
4. 否则该操作会尝试修改一个不可变绑定的值, 因此如果 `S` 的值为 `true`, 则抛出一个 `TypeError` 异常。

GetBindingValue (N,S)

声明式环境记录项的 `GetBindingValue` 具体方法简单地返回名称为参数 `N` 的绑定的值。方法调用时, 该绑定必须存在。如果 `S` 的值为 `true` 且该绑定是一个未初始化的不可变绑定, 则抛出一个 `ReferenceError` 异常。

1. 令 `envRec` 为函数调用时对应的声明式环境记录项。
2. 执行断言: `envRec` 必须有 `N` 的绑定。
3. 如果 `envRec` 中 `N` 的绑定是一个未初始化的不可变绑定, 则:
 1. 如果 `S` 为 `false`, 返回 `undefined`, 否则抛出一个 `ReferenceError` 异常。
4. 否则返回 `envRec` 中与 `N` 绑定的值。

DeleteBinding (N)

声明式环境记录项的 `DeleteBinding` 具体方法只能删除显示指定可被删除的那些绑定。

1. 令 `envRec` 为函数调用时对应的声明式环境记录项。
2. 如果 `envRec` 不包含名称为 `N` 的绑定, 返回 `true`。
3. 如果 `envRec` 中 `N` 的绑定不能删除, 返回 `false`。
4. 移除 `envRec` 中 `N` 的绑定。
5. 返回 `true`。

ImplicitThisValue()

声明式环境记录项永远将 `undefined` 作为其 `ImplicitThisValue` 返回。

1. 返回 `undefined`。

CreateImmutableBinding (N)

声明式环境记录项的 `CreateImmutableBinding` 具体方法会创建一个不可变绑定，其名称为 `N` 且初始化其值为 `undefined`。调用方法时，该环境记录项中不得存在 `N` 的绑定。

1. 令 `envRec` 为函数调用时对应的声明式环境记录项。
2. 执行断言：`envRec` 不存在 `N` 的绑定。
3. 在 `envRec` 中为 `N` 创建一个不可变绑定，并记录为未初始化。

InitializeImmutableBinding (N,V)

声明式环境记录项的 `InitializeImmutableBinding` 具体方法用于将当前名称为参数 `N` 的绑定的值修改为参数 `V` 指定的值。方法调用时，必须存在 `N` 对应的未初始化的不可变绑定。

1. 令 `envRec` 为函数调用时对应的声明式环境记录项。
2. 执行断言：`envRec` 存在一个与 `N` 对应的未初始化的不可变绑定。
3. 在 `envRec` 中将 `N` 的绑定的值设置为 `V`。
4. 在 `envRec` 中将 `N` 的不可变绑定记录为已初始化。

对象式环境记录项

每一个对象式环境记录项都有一个关联的对象，这个对象被称作 **绑定对象**。对象式环境记录项直接将一系列标识符与其绑定对象的属性名称建立一一对应关系。不符合 `IdentifierName` 的属性名不会作为绑定的标识符使用。无论是对象自身的，还是继承的属性都会作为绑定，无论该属性的 `[[Enumerable]]` 特性的值是什么。由于对象的属性可以动态的增减，因此对象式环境记录项所绑定的标识符集合也会隐匿地变化，这是增减绑定对象的属性而产生的副作用。通过以上描述的副作用而建立的绑定，均被视为可变绑定，即使该绑定对应的属性的 `Writable` 特性的值为 `false`。对象式环境记录项没有不可变绑定。

对象式环境记录项可以通过配置的方式，将其绑定对象合为函数调用时的隐式 `this` 对象的值。这一功能用于规范 `With` 表达式（12.10 章）引入的绑定行为。该行为通过对象式环境记录项中布尔类型的 `provideThis` 值控制，默认情况下，`provideThis` 的值为 `false`。

环境记录项定义的方法的具体行为将由以下算法给予描述。

HasBinding(N)

对象式环境记录项的 **HasBinding** 具体方法判断其关联的绑定对象是否有名为 **N** 的属性：

1. 令 **envRec** 为函数调用时对应的声明式环境记录项。
2. 令 **bindings** 为 **envRec** 的绑定对象。
3. 以 **N** 为属性名，调用 **bindings** 的 **[[HasProperty]]** 内部方法，并返回调用的结果。

CreateMutableBinding (N, D)

对象式环境记录项的 **CreateMutableBinding** 具体方法会在其关联的绑定对象上创建一个名称为 **N** 的属性，并初始化其值为 **undefined**。调用方法时，绑定对象不得包含名称为 **N** 的属性。如果调用方法时提供了布尔类型的参数 **D** 且其值为 **true**，则设置新创建的属性的 **[[Configurable]]** 特性的值为 **true**，否则设置为 **false**。

1. 令 **envRec** 为函数调用时对应的声明式环境记录项。
2. 令 **bindings** 为 **envRec** 的绑定对象。
3. 执行断言：以 **N** 为属性名，调用 **bindings** 的 **[[HasProperty]]** 内部方法，调用的结果为 **false**。
4. 如果 **D** 的值为 **true**，则令 **configValue** 的值为 **true**，否则令 **configValue** 的值为 **false**。
5. 以 **N**、属性描述符 **{[[Value]]:undefined, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: configValue}** 和布尔值 **true** 为参数，调用 **bindings** 的 **[[DefineOwnProperty]]** 内部方法。

SetMutableBinding (N,V,S)

对象式环境记录项的 **SetMutableBinding** 具体方法会尝试设置其关联的绑定对象中名为 **N** 的属性的值为 **V**。方法调用时，绑定对象中应当存在该属性，如果该属性不存在或属性不可写，则根据 **S** 参数的值来执行错误处理。

1. 令 **envRec** 为函数调用时对应的声明式环境记录项。
2. 令 **bindings** 为 **envRec** 的绑定对象
3. 以 **N**、**V** 和 **S** 为参数，调用 **bindings** 的 **[[Put]]** 内部方法。

GetBindingValue(N,S)

对象式环境记录项的 **GetBindingValue** 具体方法返回其关联的绑定对象中名为 **N** 的属性的值。方法调用时，绑定对象中应当存在该属性，如果该属性不存在，则方法的返回值由 **S** 参数决定：

1. 令 **envRec** 为函数调用时对应的声明式环境记录项。
2. 令 **bindings** 为 **envRec** 的绑定对象
3. 以 **N** 为属性名，调用 **bindings** 的 **[[HasProperty]]** 内部方法，并令 **value** 为调用的结果。
4. 如果 **value** 的值为 **false**，则：
 1. 如果 **S** 的值为 **false**，则返回 **undefined**，否则抛出一个 **ReferenceError** 异常。
5. 以 **N** 为参数，调用 **bindings** 的 **[[Get]]** 内部方法，并返回调用的结果。

DeleteBinding (N)

对象式环境记录项的 **DeleteBinding** 具体方法只能用于删除其关联的绑定对象上 **[[Configurable]]** 特性的值为 **true** 的属性所对应的绑定。

1. 令 **envRec** 为函数调用时对应的声明式环境记录项。
2. 令 **bindings** 为 **envRec** 的绑定对象
3. 以 **N** 和布尔值 **false** 为参数，调用 **bindings** 的 **[[Delete]]** 内部方法。

ImplicitThisValue()

对象式环境记录项的 **ImplicitThisValue** 通常返回 **undefined**，除非其 **provideThis** 标识的值为 **true**。

1. 令 **envRec** 为函数调用时对应的声明式环境记录项。
2. 如果 **envRec** 的 **provideThis** 标识的值为 **true**，返回 **envRec** 的绑定对象。
3. 否则返回 **undefined**。

词法环境的运算

在本标准中，以下抽象运算将被用于操作环境记录项：

GetIdentifierReference (lex, name, strict)

当调用 **GetIdentifierReference** 抽象运算时，需要指定一个 词法环境 **lex**，一个标识符字符串 **name** 以及一个布尔型标识 **strict**。**lex** 的值可以为 **null**。当调用该运算时，按以下步骤进行：

1. 如果 `lex` 的值为 `null`，则：
 1. 返回一个类型为 引用 的对象，其基值为 `undefined`，引用的名称为 `name`，严格模式标识的值为 `strict`。
2. 令 `envRec` 为 `lex` 的环境数据。
3. 以 `name` 为参数 `N`，调用 `envRec` 的 `HasBinding(N)` 具体方法，并令 `exists` 为调用的结果。
4. 如果 `exists` 为 `true`，则：
 1. 返回一个类型为 引用 的对象，其基值为 `envRec`，引用的名称为 `name`，严格模式标识的值为 `strict`。
5. 否则：
 1. 令 `outer` 为 `lex` 的外部环境引用。
 2. 以 `outer`、`name` 和 `strict` 为参数，调用 `GetIdentifierReference`，并返回调用的结果。

NewDeclarativeEnvironment (E)

当调用 `NewDeclarativeEnvironment` 抽象运算时，需指定一个 词法环境 `E`，其值可以为 `null`，此时按以下步骤进行：

1. 令 `env` 为一个新建的 词法环境。
2. 令 `envRec` 为一个新建的 声明式环境数据，该环境数据不包含任何绑定。
3. 令 `env` 的环境数据为 `envRec`。
4. 令 `env` 的外部词法环境引用至 `E`。
5. 返回 `env`。

NewObjectEnvironment (O, E)

当调用 `NewObjectEnvironment` 抽象运算时，需指定一个对象 `O` 及一个 词法环境 `E`（其值可以为 `null`），此时按以下步骤进行：

1. 令 `env` 为一个新建的 词法环境。
2. 令 `envRec` 为一个新建的 对象环境数据，该环境数据包含 `O` 作为绑定对象。
3. 令 `env` 的环境数据为 `envRec`。
4. 令 `env` 的外部词法环境引用至 `E`。
5. 返回 `env`。

全局环境

全局环境 是一个唯一的 词法环境，它在任何 `ECMA` 脚本的代码执行前创建。全局环境的环境数据 是一个 `#object-environment-record` 对象环境数据，该

环境数据使用 全局对象 (15.1) 作为 绑定对象 。全局环境的 外部环境引用 为 null。

在 ECMA 脚本的代码执行过程中，可能会向 全局对象 添加额外的属性，也可能修改其初始属性的值。

执行环境

当控制器转入 ECMA 脚本的可执行代码时，控制器会进入一个执行环境。当前活动的多个执行环境在逻辑上形成一个栈结构。该逻辑栈的最顶层的执行环境称为当前运行的执行环境。任何时候，当控制器从当前运行的执行环境相关的可执行代码转入与该执行环境无关的可执行代码时，会创建一个新的执行环境。新建的这个执行环境会推入栈中，成为当前运行的执行环境。

执行环境包含所有用于追踪与其相关的代码的执行进度的状态。精确地说，每个执行环境包含如表 19 列出的组件。

执行环境的状态组件

组件	作用目的
词法环境	指定一个词法环境对象，用于解析该执行环境内的代码创建的标识符引用。
变量环境	指定一个词法环境对象，其环境数据用于保存由该执行环境内的代码通过 变量表达式 和 函数表达式 创建的绑定。
>This 绑定	指定该执行环境内的 ECMA 脚本代码中 this 关键字所关联的值。

其中执行环境的词法环境和变量环境组件始终为 词法环境 对象。当创建一个执行环境时，其词法环境组件和变量环境组件最初是同一个值。在该执行环境相关联的代码的执行过程中，变量环境组件永远不变，而词法环境组件有可能改变。

在本标准中，通常情况下，只有正在运行的执行环境（执行环境栈里的最顶层对象）会被算法直接修改。因此当遇到“词法环境”，“变量环境”和“>This 绑定”这三个术语时，指的是正在运行的执行环境的对应组件。

执行环境是一个纯粹的标准机制，并不代表任何 ECMA 脚本实现的工件。在 ECMA 脚本程序中是不可能访问到执行环境的。

标识符解析

标识符解析是指使用正在运行的执行环境中的词法环境，通过一个 标识符 获得其对应的绑定的过程。在 ECMA 脚本代码执行过程中，PrimaryExpression : Identifier 这一语法产生式将按以下算法进行解释执行：

1. 令 `env` 为正在运行的执行环境的 词法环境 。
2. 如果正在解释执行的语法产生式处在 严格模式下的代码 中，则仅 `strict` 的值为 `true`，否则令 `strict` 的值为 `false`。
3. 以 `env`，`Identifier` 和 `strict` 为参数，调用 `GetIdentifierReference` 函数，并返回调用的结果。

解释执行一个标识符得到的结果必定是 引用 类型的对象，且其引用名属性的值与 `Identifier` 字符串相等。

建立执行环境

解释执行 全局代码 或使用 `eval` 函数（15.1.2.1）输入的代码会创建并进入一个新的执行环境。每次调用 `ECMA` 脚本代码定义的函数（13.2.1）也会建立并进入一个新的执行环境，即便函数是自身递归调用的。每一次 `return` 都会退出一个执行环境。抛出异常也可退出一个或多个执行环境。

当控制流进入一个执行环境时，会设置该执行环境的 `this` 绑定，定义变量环境和初始词法环境，并执行定义绑定初始化过程（10.5）。以上这些步骤的严格执行方式由进入的代码的类型决定。

进入全局代码

当控制流进入 全局代码 的执行环境时，执行以下步骤：

1. 按 10.4.1.1 描述的方案，使用 全局代码 初始化执行环境。
2. 按 10.5 描述的方案，使用 全局代码 执行定义绑定初始化步骤。

10.4.1.1 初始化全局执行环境

以下步骤描述 `ECMA` 脚本的全局执行环境 `C` 的创建过程：

1. 将变量环境设置为 全局环境 。
2. 将词法环境设置为 全局环境 。
3. 将 `this` 绑定设置为 全局对象 。

进入 eval 代码

当控制流进入 `eval` 代码 的执行环境时，执行以下步骤：

1. 如果没有调用环境，或者 `eval` 代码 并非通过直接调用（15.1.2.1.1）`eval` 函数进行评估的，则

1. 按 (10.4.1.1) 描述的初始化全局执行环境的方案，以 `eval` 代码 作为 `C` 来初始化执行环境。
2. 否则
 1. 将 `this` 绑定设置为当前执行环境下的 `this` 绑定。
 2. 将词法环境设置为当前执行环境下的 词法环境 。
 3. 将变量环境设置为当前执行环境下的变量环境。
3. 如果 `eval` 代码 是 严格模式下的代码 ， 则
 1. 令 `strictVarEnv` 为以词法环境为参数调用 `NewDeclarativeEnvironment` 得到的结果。
 2. 设置词法环境为 `strictVarEnv`。
 3. 设置变量环境为 `strictVarEnv`。
4. 按 10.5 描述的方案，使用 `eval` 代码 执行定义绑定初始化步骤。

严格模式下的限制

如果调用环境的代码或 `eval` 代码 是 严格模式下的代码 ， 则 `eval` 代码不能在调用环境的变量环境中 初始化变量及函数绑定 。与之相对的，变量及函数绑定将在一个新的环境变量中被初始化，该环境变量仅可被 `eval` 代码 访问。

进入函数代码

当控制流根据一个函数对象 `F`、调用者提供的 `thisArg` 以及调用者提供的 `argumentList`，进入 函数代码的执行环境时，执行以下步骤：

1. 如果 函数代码 是 严格模式下的代码 ， 设 `this` 绑定为 `thisArg`。
2. 否则如果 `thisArg` 是 `null` 或 `undefined`，则设 `this` 绑定为 全局对象 。
3. 否则如果 `Type(thisArg)` 的结果不为 `Object`，则设 `this` 绑定为 `ToObject(thisArg)`。
4. 否则设 `this` 绑定为 `thisArg`。
5. 以 `F` 的 `[[Scope]]` 内部属性为参数调用 `NewDeclarativeEnvironment`，并令 `localEnv` 为调用的结果。
6. 设词法环境为 `localEnv`。
7. 设变量环境为 `localEnv`。
8. 令 `code` 为 `F` 的 `[[Code]]` 内部属性的值。
9. 按 [10.5](#10.5) 描述的方案，使用 函数代码 `code` 和 `argumentList` 执行定义绑定初始化步骤。

定义绑定初始化

每个执行环境都有一个关联的变量环境。当在一个执行环境下评估一段 **ECMA** 脚本时，变量和函数定义会以绑定的形式添加到这个变量环境的 环境记录 中。

对于函数 函数代码，参数也同样会以绑定的形式添加到这个变量环境的环境记录 中。

选择使用哪一个、哪一类型的 环境记录 来绑定定义，是由执行环境下执行的 ECMA 脚本的类型决定的，而其它部分的逻辑是相同的。当进入一个执行环境时，会按以下步骤在变量环境上创建绑定，其中使用到调用者提供的代码设为 `code`，如果执行的是 函数代码 ，则设 参数列表 为 `args`：

1. 令 `env` 为当前运行的执行环境的环境变量的 环境记录 。
2. 如果 `code` 是 `eval` 代码 ，则令 `configurableBindings` 为 `true`，否则令 `configurableBindings` 为 `false`。
3. 如果代码是 严格模式下的代码 ，则令 `strict` 为 `true`，否则令 `strict` 为 `false`。
4. 如果代码为 函数代码 ，则：
 1. 令 `func` 为通过 `[[Call]]` 内部属性初始化 `code` 的执行的函数对象。令 `names` 为 `func` 的 `[[FormalParameters]]` 内部属性。
 2. 令 `argCount` 为 `args` 中元素的数量。
 3. 令 `n` 为数字类型，其值为 0。
 4. 按列表顺序遍历 `names`，对于每一个字符串 `argName`：
 - i. 令 `n` 的值为 `n` 当前值加 1。
 - ii. 如果 `n` 大于 `argCount`，则令 `v` 为 `undefined`，否则令 `v` 为 `args` 中的第 `n` 个元素。
 - iii. 以 `argName` 为参数，调用 `env` 的 `HasBinding` 具体方法，并令 `argAlreadyDeclared` 为调用的结果。
 - iv. 如果 `argAlreadyDeclared` 的值为 `false`，以 `argName` 为参数调用 `env` 的 `CreateMutableBinding` 具体方法。
 - v. 以 `argName`、`v` 和 `strict` 为参数，调用 `env` 的 `SetMutableBinding` 具体方法。
 5. 按源码顺序遍历 `code`，对于每一个 `FunctionDeclaration` 表达式 `f`：
 1. 令 `fn` 为 `FunctionDeclaration` 表达式 `f` 中的 标识符 。
 2. 按 第 13 章 中所述的步骤初始化 `FunctionDeclaration` 表达式 ，并令 `fo` 为初始化的结果。
 3. 以 `fn` 为参数，调用 `env` 的 `HasBinding` 具体方法，并令 `argAlreadyDeclared` 为调用的结果。
 4. 如果 `argAlreadyDeclared` 的值为 `false`，以 `fn` 和 `configurableBindings` 为参数调用 `env` 的 `CreateMutableBinding` 具体方法。
 5. 否则如果 `env` 是全局环境的 环境记录 对象，则：
 - i. 令 `go` 为全局对象。
 - ii. 以 `fn` 为参数，调用 `go` 和 `[[GetProperty]]` 内部方法，并令 `existingProp` 为调用的结果。
 - iii. 如果 `existingProp.{{Configurable}}` 的值为 `true`，则：
 1. 以 `fn`、由 `{{Value}}: undefined, {{Writable}}: true, {{Enumerable}}: true, {{Configurable}}: configurableBindings` } 组成的 属性描述符 和 `true` 为参数，调用 `go` 的 `[[DefineOwnProperty]]` 内部方法。

- iv. 否则如果 `IsAccessorDescriptor(existingProp)` 的结果为真，或 `existingProp` 的特性中没有 `[[Writable]]: true, [[Enumerable]]: true`，则：
 - 1. 抛出一个 `TypeError` 异常。
- v. 以 `fn`、`fo` 和 `strict` 为参数，调用 `env` 的 `SetMutableBinding` 具体方法。
- 6. 以 `arguments` 为参数，调用 `env` 的 `HasBinding` 具体方法，并令 `argumentsAlreadyDeclared` 为调用的结果。
- 7. 如果 `code` 是 函数代码，并且 `argumentsAlreadyDeclared` 为 `false`，则
 - 1. 以 `fn`、`names`、`args`、`env` 和 `strict` 为参数，调用 `CreateArgumentsObject` 抽象运算函数，并令 `argsObj` 为调用的结果。
 - 2. 如果 `strict` 为 `true`，则：
 - i. 以字符串 `"arguments"` 为参数，调用 `env` 的 `CreateImmutableBinding` 具体方法。
 - ii. 以字符串 `"arguments"` 和 `argsObj` 为参数，调用 `env` 的 `InitializeImmutableBinding` 具体函数。
 - 3. 否则：
 - i. 以字符串 `"arguments"` 为参数，调用 `env` 的 `CreateMutableBinding` 具体方法。
 - ii. 以字符串 `"arguments"`、`argsObj` 和 `false` 为参数，调用 `env` 的 `SetMutableBinding` 具体函数。
- 8. 按源码顺序遍历 `code`，对于每一个 `VariableDeclaration` 和 `VariableDeclarationNoIn` 表达式：
 - 1. 令 `dn` 为 `d` 中的标识符。
 - 2. 以 `dn` 为参数，调用 `env` 的 `HasBinding` 具体方法，并令 `varAlreadyDeclared` 为调用的结果。
 - 3. 如果 `varAlreadyDeclared` 为 `false`，则：
 - i. 以 `dn` 和 `configurableBindings` 为参数，调用 `env` 的 `CreateMutableBinding` 具体方法。
 - ii. 以 `dn`、`undefined` 和 `strict` 为参数，调用 `env` 的 `SetMutableBinding` 具体方法。

Arguments 对象

当控制器进入到函数代码的执行环境时，将创建一个 `arguments` 对象，除非它作为标识符 `"arguments"` 出现在该函数的形参列表中，或者是该函数代码内部的变量声明标识符或函数声明标识符。

`Arguments` 对象通过调用抽象方法 `CreateArgumentsObject` 创建，调用时将以下参数传入：`func`、`names`、`args`、`env`、`strict`。将要执行的函数对象作为 `func` 参数，将该函数的所有形参名加入一个 `List` 列表，作为 `names` 参数，将所有传给内部方法 `[[Call]]` 的实际参数，作为 `args` 参数，将该函数代码的环境变量作为 `env` 参数，将该函数代码是否为严格代码作为 `strict` 参数。当 `CreateArgumentsObject` 调用时，按照以下步骤执行：

- 1. 令 `len` 为参数 `args` 的元素个数

2. 令 obj 为一个新建的 ECMAScript 对象
3. 按照 8.12 章节中的规范去设定 obj 对象的所有内部方法
4. 将 obj 对象的内部属性 `[[Class]]` 设置为 "Arguments"
5. 令 Object 为标准的内置对象的构造函数 (15.2.2)
6. 将 obj 对象的内部属性 `[[Prototype]]` 设置为标准的内置对象的原型对象
7. 调用 obj 的内部方法 `[[DefineOwnProperty]]`, 将 "length" 传递进去, 属性描述符为: `{[[Value]]: len, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}`, 参数为 false
8. 令 map 为表达式 `new Object()` 创建的对象, 就是名为 Object 的标准的内置构造函数
9. 令 mappedNames 为一个空的 List 列表
10. 令 `indx = len - 1`
11. 当 `indx >= 0` 的时候, 重复此过程:
 1. 令 val 为 args (维度从 0 开始的 list 列表) 的第 indx 维度所在的元素
 2. 调用 obj 的内部方法 `[[DefineOwnProperty]]`, 将 `ToString(indx)` 传递进去, 属性描述符为: `{[[Value]]: val, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, 参数为 false
 3. 如果 indx 小于 names 的元素个数, 则
 - i. 令 name 为 names (维度从 0 开始的 list 列表) 的第 indx 维度所在的元素
 - ii. 如果 strict 值为 false, 且 name 不是一个 mappedNames 元素, 则
 1. 将 name 添加到 mappedNames 列表中, 作为它的一个元素
 2. 令 g 为调用抽象操作 MakeArgGetter 的结果, 其参数为 name 和 env
 3. 令 p 为调用抽象操作 MakeArgSetter 的结果, 其参数为 name 和 env
 4. 调用 map 对象的内部方法 `[[DefineOwnProperty]]`, 将 `ToString(indx)` 传递进去, 属性描述符为: `{[[Set]]: p, [[Get]]: g, [[Configurable]]: true}`, 参数为 false
 4. 令 `indx = indx - 1`
12. 如果 mappedNames 不为空, 则
 1. 将 obj 对象的内部属性 `[[ParameterMap]]` 设置为 map
 2. 将 obj 对象的内部方法 `[[Get]]`, `[[GetOwnProperty]]`, `[[DefineOwnProperty]]`, `[[Delete]]` 按下面给出的定义进行设置。
13. 如果 strict 值为 false, 则
 1. 调用 obj 对象的内部方法 `[[DefineOwnProperty]]`, 将 "callee" 传递进去, 属性描述符为: `{[[Value]]: func, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}`, 参数为 false
14. 否则, strict 值为 true, 那么
 1. 令 thrower 为 `[[ThrowTypeError]]` 函数对象 (13.2.3)
 2. 调用 obj 对象的内部方法 `[[DefineOwnProperty]]`, 将 "caller" 传递进去, 属性描述符为: `{[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}`, 参数为 false
 3. 调用 obj 对象的内部方法 `[[DefineOwnProperty]]`, 将 "callee" 传递进去, 属性描述符为: `{[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}`, 参数为 false

15. 返回 obj

抽象操作 **MakeArgGetter** 以字符串 **name** 和环境记录 **env** 作为参数被调用时，会创建一个函数对象，当执行完后，会返回在 **env** 中绑定的 **name** 的值。执行步骤如下：

1. 令 **body** 为字符 "return ", **name**, ";" 的连接字符串
2. 返回一个按照 13.2 章节中描述的方式创建的函数对象，它不需要形参列表，以 **body** 作为它的 **FunctionBody**，以 **env** 作为它的 **Scope**，并且 **Strict** 值为 **true**

抽象操作 **MakeArgSetter** 以字符串 **name** 和环境记录 **env** 作为参数被调用时，会创建一个函数对象，当执行完后，会给在 **env** 中绑定的 **name** 设置一个值。执行步骤如下：

1. 令 **param** 为 **name** 和字符串 "_arg" 的连接字符串
2. 令 **body** 为字符串 "=:"; 将 替换为 **name** 的值，将 替换为 **param** 的值
3. 返回一个按照 13.2 章节中描述的方式创建的函数对象，以一个只包含字符串 **param** 的 **list** 列表作为它的形参，以 **body** 作为它的函数体 (**FunctionBody**)，以 **env** 作为它的 **Scope**，并且 **Strict** 值为 **true**

当 **arguments** 对象的内部方法 **[[Get]]** 在一个非严格模式下带有形参的函数中，在一个属性名为 **P** 的条件下被调用时，其执行步骤如下：

1. 令 **map** 为 **arguments** 对象的内部属性 **[[ParameterMap]]**
2. 令 **isMapped** 为 **map** 对象的内部方法 **[[GetOwnProperty]]** 传入参数 **P** 的执行结果
3. 如果 **isMapped** 值为 **undefined**，则
 1. 令 **v** 为 **arguments** 对象的内部默认的 **[[Get]]** 方法 (8.12.3)，传入参数 **P** 的执行结果
 2. 如果 **P** 为 "caller"，且 **v** 为严格模式下的 **Function** 对象，则抛出一个 **TypeError** 的异常
 3. 返回 **v**
4. 否则，**map** 包含一个 **P** 的形参映射表
 1. 返回 **map** 对象的内部方法 **[[Get]]** 传入参数 **P** 的执行结果

当 **arguments** 对象的内部方法 **[[GetOwnProperty]]** 在一个非严格模式下带有形参的函数中，在一个属性名为 **P** 的条件下被调用时，其执行步骤如下：

1. 令 **desc** 为 **arguments** 对象的内部方法 **[[GetOwnProperty]]** (8.12.1) 传入参数 **P** 的执行结果
2. 如果 **desc** 为 **undefined**，返回 **desc**
3. 令 **map** 为 **arguments** 对象内部属性 **[[ParameterMap]]** 的值
4. 令 **isMapped** 为 **map** 对象的内部方法 **[[GetOwnProperty]]** 传入参数 **P** 的执行结果

5. 如果 `isMapped` 的值不是 `undefined`，则
 1. 将 `desc` 的值设置为 `map` 对象的内部方法 `[[Get]]` 传入参数 `P` 的执行结果
6. 返回 `desc`

当 `arguments` 对象的内部方法 `[[DefineOwnProperty]]` 在一个非严格模式下带有形参的函数中，在一个属性名为 `P`，属性描述符为 `Desc`，布尔标志为 `Throw` 的条件下被调用时，其执行步骤如下：

1. 令 `map` 为 `arguments` 对象的内部属性 `[[ParameterMap]]` 的值
2. 令 `isMapped` 为 `map` 对象的内部方法 `[[GetOwnProperty]]` 传入参数 `P` 的执行结果
3. 令 `allowed` 为 `arguments` 对象的内部方法 `[[DefineOwnProperty]]` (8.12.9) 传入参数 `P`，`Desc`，`false` 的执行结果
4. 如果 `allowed` 为 `false`，则
 1. 如果 `Throw` 为 `true`，则抛出一个 `TypeError` 的异常，否则，返回 `false`
5. 如果 `isMapped` 的值不为 `undefined`，则
 1. 如果 `IsAccessorDescriptor(Desc)` 为 `true`，则
 - i. 调用 `map` 对象的内部方法 `[[Delete]]`，传入 `P` 和 `false` 作为参数
 2. 否则
 - i. 如果 `Desc.[[Value]]` 存在，则
 1. 调用 `map` 对象的内部方法 `[[Put]]`，传入 `P`，`Desc.[[Value]]` 和 `Throw` 作为参数
 - ii. 如果 `Desc.[[Writable]]` 存在，且其值为 `false`，则
 1. 调用 `map` 对象的内部方法 `[[Delete]]`，传入 `P` 和 `false` 作为参数
 6. 返回 `true`

当 `arguments` 对象的内部方法 `[[Delete]]` 在一个非严格模式下带有形参的函数中，在一个属性名为 `P`，布尔标志为 `Throw` 的条件下被调用时，其执行步骤如下：

1. 令 `map` 为 `arguments` 对象的内部属性 `[[ParameterMap]]` 的值
2. 令 `isMapped` 为 `map` 对象的内部方法 `[[GetOwnProperty]]` 传入参数 `P` 的执行结果
3. 令 `result` 为 `arguments` 对象的内部方法 `[[Delete]]` (8.12.7) 传入参数 `P` 和 `Throw` 的执行结果
4. 如果 `result` 为 `true`，且 `isMapped` 不为 `undefined`，则
 1. 调用 `map` 对象的内部方法 `[[Delete]]`，传入 `P` 和 `false` 作为参数
5. 返回 `result`

非严格模式下的函数，`arguments` 对象以数组索引（参见 15.4 的定义）作为数据属性的命名，其数字名字的值少于对应的函数对象初始时的形参数量，它们与绑定在该函数执行环境中对应的参数共享值。这意味着，改变该属性将改变这些对应的、绑定的参数的值，反之亦然。如果其中一个属性被删除然后再对其重定义，或者其中一个属性在某个访问器属性内部被更改，则这种对应关系将被打破。

破。严格模式下的函数，`arguments` 对象的属性值就是传入该函数的实际参数的简单拷贝，它们与形参之间的值不存在动态的联动关系。

`ParameterMap` 对象和它的属性值被作为说明 `arguments` 对象对应绑定参数的装置。`ParameterMap` 对象和它的属性值对象不能直接被 `ECMAScript` 代码访问。作为 `ECMAScript` 的实现，不需要实际创建或使用这些对象去实现指定的语义。

严格模式下函数的 `Arguments` 对象定义的非可配置的访问器属性，`"caller"` 和 `"callee"`，在它们被访问时，将抛出一个 `TypeError` 的异常。在非严格模式下，`"callee"` 属性具有非常明确的意义，`"caller"` 属性有一个历史问题，它是否被提供，视为一个由实作环境决定的，在具体的 `ECMAScript` 实作进行扩展。在严格模式下对这些属性的定义的出现是为了确保它们俩谁也不能在规范的 `ECMAScript` 实作中以任何方式被定义。

表达式

主值表达式

语法：

`PrimaryExpression` :

`this`

`Identifier`

`Literal`

`ArrayLiteral`

`ObjectLiteral`

`(Expression)`

this 关键字

`this` 关键字执行为当前执行环境的 `ThisBinding`。

标识符引用

Identifier 的执行遵循 10.3.1 所规定的标识符查找。标识符执行的结果总是一个 Reference 类型的值。

字面量引用

Literal 按照 7.8 所描述的方式执行。

数组初始化

数组初始化是一个以字面量的形式书写的描述数组对象的初始化的表达式。它是一个零个或者多个表达式的序列，其中每一个表示一个数组元素，并且用方括号括起来。元素并不一定要是字面量，每次数组初始化执行时它们都会被执行一次。

数组元素可能在元素列表的开始、结束，或者中间位置被省略。当元素列表中的一个逗号没有被 AssignmentExpression 优先处理（如，一个逗号在另一个逗号之前。）的情况下，缺失的数组元素仍然会对数组长度有贡献，并且增加后续元素的索引值。省略数组元素是没有定义的。假如元素在数组末尾被省略，那么元素不会贡献数组长度。

语法：

```
ArrayLiteral :  
  [ Elisionopt ]  
  [ ElementList ]  
  [ ElementList , Elisionopt ]ElementList :  
    ElisionoptAssignmentExpression  
    ElementList , ElisionoptAssignmentExpressionElision :  
      ,  
Elision ,
```

语义：

产生式 ArrayLiteral : [Elision_{opt}] 按照下面的过程执行：

1. 令 array 为以表达式 new Array() 完全一致的方式创建一个新对象的结果，其中 Array 是一个标准的内置构造器
2. 令 pad 为解释执行 Elision 的结果；如果不存在的话，使用数值 0.
3. 以参数 "length", pad, 和 false 调用 array 的 [[Put]] 内置方法

4. 返回 array.

产生式 `ArrayLiteral : [ElementList]` 按照下面的过程执行：

1. 返回解释执行 `ElementList` 的结果 .

产生式 `ArrayLiteral : '[ElementList , 'Elisionopt]` 按照下面的过程执行：

1. 令 `array` 为解释执行 `ElementList` 的结果 .
2. 令 `pad` 为解释执行 `Elision` 的结果；如果不存在的话，使用数值 0.
3. 令 `len` 为以参数 "length". 调用 `array` 的 `[[Get]]` 内置方法的结果
4. 以参数 "length", `ToUint32(pad+len)`, 和 `false` 调用 `array` 的 `[[Put]]` 内置方法
5. 返回 `array`.

产生式 `ElementList : Elisionopt AssignmentExpression` 按照下面的过程执行：

1. 令 `array` 为以表达式 `new Array()` 完全一致的方式创建一个新对象的结果，其中 `Array` 是一个标准的内置构造器
2. 令 `firstIndex` 为解释执行 `Elision` 的结果；如果不存在的话，使用数值 0.
3. 令 `initResult` 为解释执行 `AssignmentExpression` 的结果 .
4. 令 `initValue` 为 `GetValue(initResult)`.
5. 以参数 `ToString(firstIndex)`, 属性描述对象 `{ [[Value]]: initValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, 和 `false` 调用 `array` 的 `[[DefineOwnProperty]]` 内置方法
6. 返回 `array`.

产生式 `ElementList : ElementList , Elisionopt AssignmentExpression` 按照下面的过程执行：

1. 令 `array` 为解释执行 `ElementList` 的结果 .
2. 令 `pad` 为解释执行 `Elision` 的结果；如果不存在的话，使用数值 0.
3. 令 `initResult` 为解释执行 `AssignmentExpression` 的结果 .
4. 令 `initValue` 为 `GetValue(initResult)`.
5. 令 `len` 为以参数 "length". 调用 `array` 的 `[[Get]]` 内置方法的结果
6. 以参数 `ToString(ToUint32((pad+len)))` 和 属性描述对象 `{ [[Value]]: initValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, 和 `false` 调用 `array` 的 `[[DefineOwnProperty]]` 内置方法
7. 返回 `array`.

产生式 `Elision :`，按照下面的过程执行：

1. 返回数值 1.

产生式 `Elision : Elision ,` 按照下面的过程执行：

1. 令 preceding 为解释执行 Elision 的结果 .
2. 返回 preceding+1.

`[[DefineOwnProperty]]` 用于确保即使默认的原型对象被更改的情况下自身属性也会被定义，可以杜绝用 `[[put]]` 创建一个新的自身属性。

对象初始化

对象初始化是一个以直接量的方式描述对象的初始化过程的表达式。它是用花括号括起来的由零或者多对属性名 / 关联值组成的列表，值不需要是直接量，每次对象初始化被执行到时他们会执行一次。

语法：

```
ObjectLiteral :  
  
{ }  
  
{ PropertyNameAndValueList }  
  
{ PropertyNameAndValueList , }PropertyNameAndValueList :  
  
PropertyAssignment  
  
PropertyNameAndValueList ,  
PropertyAssignmentPropertyAssignment :  
  
PropertyName : AssignmentExpression  
  
get PropertyName() { FunctionBody }  
  
set PropertyName( PropertySetParameterList )  
{ FunctionBody }PropertyName :  
  
IdentifierName  
  
StringLiteral  
  
NumericLiteralPropertySetParameterList :
```

Identifier

语义：

产生式 `ObjectLiteral : { }` 按照下面的过程执行：

1. 返回 a new object created as if by the expression `new Object()` where `Object` is the `st` 和 `ard` built-in constructor with that name.

产生式 `s ObjectLiteral : { PropertyNameAndValueList }` 以及 `ObjectLiteral : { PropertyNameAndValueList , }` 按照下面的过程执行 :

1. 返回解释执行 `PropertyNameAndValueList` 的结果 .

产生式 `PropertyNameAndValueList : PropertyAssignment` 按照下面的过程执行 :

1. 令 `obj` 为以表达式 `new Object()` 完全一致的方式创建一个新对象的结果, 其中 `Object` 是一个标准的内置构造器
2. 令 `propId` 为解释执行 `PropertyAssignment` 的结果 .
3. 以参数 `propId.name`, `propId.descriptor`, 和 `false` 调用 `obj` 的 `[[DefineOwnProperty]]` 内置方法
4. 返回 `obj`.

产生式 `PropertyNameAndValueList : PropertyNameAndValueList , PropertyAssignment` 按照下面的过程执行 :

1. 令 `obj` 为解释执行 `PropertyNameAndValueList` 的结果 .
2. 令 `propId` 为解释执行 `PropertyAssignment` 的结果 .
3. 令 `previous` 为以参数 `propId.name`. 调用 `obj` 的 `[[GetOwnProperty]]` 内置方法的结果
4. 如果 `previous` 不是 `undefined`, 且当以下任意一个条件为 `true` 时, 则抛出一个 `SyntaxError` 异常:
 - 产生式包含在严格模式下并且 `IsDataDescriptor(previous)` 为 `true` 并且 `IsDataDescriptor(propId.descriptor)` 为 `true`
 - `IsDataDescriptor(previous)` 为 `true` 并且 `IsAccessorDescriptor(propId.descriptor)` 为 `true`.
 - `IsAccessorDescriptor(previous)` 为 `true` 并且 `IsDataDescriptor(propId.descriptor)` 为 `true`.
 - `IsAccessorDescriptor(previous)` 为 `true` 并且 `IsAccessorDescriptor(propId.descriptor)` 为 `true` 并且 `previous` 和 `propId.descriptor` 都有 `[[Get]]` 字段 或者 `previous` 和 `propId.descriptor` 都有 `[[Set]]` 字段。

如果以上步骤抛出一个语法错误, 那么实现应该把这个错误视为早期语法错误。

产生式 `PropertyAssignment : PropertyName : AssignmentExpression` 按照下面的过程执行 :

1. 令 `propName` 为解释执行 `PropertyName` 的结果 .
2. 令 `exprValue` 为解释执行 `AssignmentExpression` 的结果 .

3. 令 `propValue` 为 `GetValue(exprValue)`.
4. 令 `desc` 为属性描述对象 `{[[Value]]: propValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`
5. 返回 `Property Identifier (propName, desc)`.

产生式 `PropertyAssignment : get PropertyName () { FunctionBody }` 按照下面的过程执行：

1. 令 `propName` 为解释执行 `PropertyName` 的结果。
2. 令 `closure` 为按照 13.2 规定，以空的参数列表和 `body` 代表的 `FunctionBody` 创建的一个新的函数对象。传入当前执行中的执行环境的 `LexicalEnvironment` 作为 `Scope`。假如 `PropertyAssignment` 包含在严格模式代码中或者 `FunctionBody` 是严格模式代码，传入 `true` 为严格模式标志。
3. 令 `desc` 为属性描述对象 `{[[Get]]: closure, [[Enumerable]]: true, [[Configurable]]: true}`
4. 返回 `Property Identifier (propName, desc)`.

产生式 `PropertyAssignment : set PropertyName (PropertySetParameterList) { FunctionBody }` is evaluated as follows:

1. 令 `propName` 为解释执行 `PropertyName` 的结果。
2. 令 `closure` 为按照 13.2 规定，以 `PropertySetParameterList` 作为参数列表和 `body` 代表的 `FunctionBody` 创建的一个新的函数对象。传入当前执行中的执行环境的 `LexicalEnvironment` 作为 `Scope`。假如 `PropertyAssignment` 包含在严格模式代码中或者 `FunctionBody` 是严格模式代码，传入 `true` 为严格模式标志。
3. 令 `desc` 为属性描述对象 `{[[Set]]: closure, [[Enumerable]]: true, [[Configurable]]: true}`
4. 返回属性标识符 `(propName, desc)`.

假如 `FunctionBody` 是严格模式或者被包含在严格模式代码内，`PropertyAssignment` 中的 `PropertySetParameterList`, `"eval"` 或者 `"arguments"` 作为标识符将会是一个语法错误。

产生式 `PropertyName : IdentifierName` 按照下面的过程执行：

1. 返回一个包含跟 `IdentifierName` 完全相同的字符序列的字符串值

产生式 `PropertyName : StringLiteral` 按照下面的过程执行：

1. 返回 the SV of the `StringLiteral`.

产生式 `PropertyName : NumericLiteral` 按照下面的过程执行：

1. 令 `nbr` 为求 `NumericLiteral` 值的结果
2. 返回 `ToString(nbr)`.

分组表达式

产生式 `PrimaryExpression : (Expression)` 按照下面的过程执行：

1. 返回执行 `Expression` 的结果，它可能是 `Reference` 类型。

这一算法并不会作用 `GetValue` 于执行 `Expression` 的结果。这样做的原则是确保 `delete` 和 `typeof` 这样的运算符可以作用于括号括起来的表达式。

左值表达式

语法：

`MemberExpression :`

`PrimaryExpression`

`FunctionExpression`

`MemberExpression [Expression]`

`MemberExpression . IdentifierName`

`new MemberExpression ArgumentsNewExpression :`

`MemberExpression`

`new NewExpressionCallExpression :`

`MemberExpression Arguments`

`CallExpression Arguments`

`CallExpression [Expression]`

`CallExpression . IdentifierNameArguments :`

`()`

`(ArgumentList)ArgumentList :`

`AssignmentExpression`

ArgumentList ,
AssignmentExpressionLeftHandSideExpression :

NewExpression

CallExpression

属性访问

属性是通过 `name` 来访问的，可以使用点表示法访问

`MemberExpression . IdentifierName`

`CallExpression . IdentifierName`

或者括号表示法访问

`MemberExpression [Expression]`

`CallExpression [Expression]`

点表示法是根据以下的语法转换解释

`MemberExpression . IdentifierName`

这会等同于下面这个行为

`MemberExpression [<identifier-name-string>]`

类似地，

`CallExpression . IdentifierName`

是等同于下面的行为

`CallExpression [<identifier-name-string>]`

是一个字符串字面量，它与 Unicode 编码后的 `IdentifierName` 包含相同的字符序列。

产生式 `MemberExpression : MemberExpression [Expression]` is evaluated as follows:

1. 令 `baseReference` 为解释执行 `MemberExpression` 的结果 .
2. 令 `baseValue` 为 `GetValue(baseReference)`.
3. 令 `propertyNameReference` 为解释执行 `Expression` 的结果 .

4. 令 `propertyNameValue` 为 `GetValue(propertyNameReference)`.
5. 调用 `CheckObjectCoercible(baseValue)`.
6. 令 `propertyNameString` 为 `ToString(propertyNameValue)`.
7. 如果正在执行中的语法产生式包含在严格模式代码当中, 令 `strict` 为 `true`, 否则令 `strict` 为 `false`.
8. 返回一个值类型的引用, 其基值为 `baseValue` 且其引用名为 `propertyNameString`, 严格模式标记为 `strict`.

产生式 `CallExpression : CallExpression [Expression]` 以完全相同的方式执行, 除了第 1 步执行的是其中的 `CallExpression`。

new 运算符

产生式 `NewExpression : new NewExpression` 按照下面的过程执行：

1. 令 `ref` 为解释执行 `NewExpression` 的结果。
2. 令 `constructor` 为 `GetValue(ref)`.
3. 如果 `Type(constructor) is not Object`, 抛出一个 `TypeError` 异常。
4. 如果 `constructor` 没有实现 `[[Construct]]` 内置方法, 抛出一个 `TypeError` 异常。
5. 返回调用 `constructor` 的 `[[Construct]]` 内置方法的结果, 传入按无参数传入参数列表 (就是一个空的参数列表)。

产生式 `MemberExpression : new MemberExpression Arguments` 按照下面的过程执行：

1. 令 `ref` 为解释执行 `MemberExpression` 的结果。
2. 令 `constructor` 为 `GetValue(ref)`.
3. 令 `argList` 为解释执行 `Arguments` 的结果, 产生一个由参数值构成的内部列表类型 (11.2.4)。
4. 如果 `Type(constructor) is not Object`, 抛出一个 `TypeError` 异常。
5. 如果 `constructor` 没有实现 `[[Construct]]` 内置方法, 抛出一个 `TypeError` 异常。
6. 返回以 `argList` 为参数调用 `constructor` 的 `[[Construct]]` 内置方法的结果。

函数调用

产生式 `CallExpression : MemberExpression Arguments` 按照下面的过程执行：

1. 令 `ref` 为解释执行 `MemberExpression` 的结果。
2. 令 `func` 为 `GetValue(ref)`.

3. 令 `argList` 为解释执行 `Arguments` 的结果，产生参数值们的内部列表 (see 11.2.4).
4. 如果 `Type(func) is not Object`，抛出一个 `TypeError` 异常。
5. 如果 `IsCallable(func) is false`，抛出一个 `TypeError` 异常。
6. 如果 `Type(ref)` 为 `Reference`，那么 如果 `IsPropertyReference(ref)` 为 `true`，那么 令 `thisValue` 为 `GetBase(ref)`。否则，`ref` 的基值是一个环境记录项 令 `thisValue` 为调用 `GetBase(ref)` 的 `ImplicitThisValue` 具体方法的结果
7. 否则，假如 `Type(ref)` 不是 `Reference`。令 `thisValue` 为 `undefined`。
8. 返回调用 `func` 的 `[[Call]]` 内置方法的结果，传入 `thisValue` 作为 `this` 值和列表 `argList` 作为参数列表

产生式 `CallExpression : CallExpression Arguments` 以完全相同的方式执行，除了第 1 步执行的是其中的 `CallExpression`。

假如 `func` 是一个原生的 `ECMAScript` 对象，返回的结果永远不会是 `Reference` 类型，调用一个宿主对象是否返回一个 `Reference` 类型的值由实现决定。若一 `Reference` 值返回，则它必须是一个非严格的属性引用。

参数列表

The evaluation of an argument list produces a List of values (see 8.8).

产生式 `Arguments : ()` 按照下面的过程执行：

1. 返回一个空列表。

产生式 `Arguments : (ArgumentList)` 按照下面的过程执行：

1. 返回解释执行 `ArgumentList` 的结果。

产生式 `ArgumentList ':' AssignmentExpression` 按照下面的过程执行：

1. 令 `ref` 为解释执行 `AssignmentExpression` 的结果。
2. 令 `arg` 为 `GetValue(ref)`。
3. 返回 a List whose sole item is `arg`。

产生式 `ArgumentList ':' ArgumentList , AssignmentExpression` 按照下面的过程执行：

1. 令 `precedingArgs` 为解释执行 `ArgumentList` 的结果。
2. 令 `ref` 为解释执行 `AssignmentExpression` 的结果。
3. 令 `arg` 为 `GetValue(ref)`。

4. 返回一个列表，长度比 `precedingArgs` 大 1 且 它的 `items` 为 `precedingArgs` 的 `items`，按顺序在后面跟 `arg`，`arg` 是这个新的列表的最后一个 `item`。

函数表达式

产生式 `MemberExpression : FunctionExpression` 按照下面的过程执行：

1. 返回解释执行 `FunctionExpression` 的结果。

后缀表达式

语法：

`PostfixExpression :`

`LeftHandSideExpression`

`LeftHandSideExpression` [此处无换行 `LineTerminator`] `++`

`LeftHandSideExpression` [此处无换行 `LineTerminator`] `--`

后缀自增运算符

产生式 `PostfixExpression : LeftHandSideExpression` [此处无换行 `LineTerminator`] `++` 按照下面的过程执行：

1. 令 `lhs` 为解释执行 `LeftH` 和 `SideExpression` 的结果。
2. 假如以下所有条件都为 `true`，抛出一个 `SyntaxError` 异常：
 - `Type(lhs)` 为 `Reference`
 - `IsStrictReference(lhs)` 为 `true`
 - `Type(GetBase(lhs))` 为环境记录项
 - `GetReferencedName(lhs)` 为 `"eval"` 或 `"arguments"`

后缀自减运算符

产生式 `PostfixExpression : LeftHandSideExpression` [此处无换行 `LineTerminator`] `--` 按照下面的过程执行：

1. 令 `lhs` 为解释执行 `LeftH` 和 `SideExpression` 的结果。
2. 假如以下所有条件都为 `true`，抛出一个 `SyntaxError` 异常：
 - `Type(lhs)` 为 `Reference`

- IsStrictReference(lhs) 为 true
- Type(GetBase(lhs)) 为环境记录项
- GetReferencedName(lhs) 为 "eval" 或 "arguments"

一元运算符

语法:

```
UnaryExpression :
    PostfixExpression
    delete UnaryExpression
    void UnaryExpression
    typeof UnaryExpression
    ++ UnaryExpression
    -- UnaryExpression
    + UnaryExpression
    - UnaryExpression
    ~ UnaryExpression  ! UnaryExpression
```

delete 运算符

产生式 `UnaryExpression : delete UnaryExpression` 按照下面的过程执行：

1. 令 ref 为解释执行 `UnaryExpression` 的结果。
2. 如果 `Type(ref)` 不是 `Reference`，返回 `true`。
3. 若 `IsUnresolvableReference(ref)` 则，如果 `IsStrictReference(ref)` 为 `true`，抛出一个 `SyntaxError` 异常。否则，返回 `true`。
4. 如果 `IsPropertyReference(ref)` 为 `true` 则：返回以 `GetReferencedName(ref)` 和 `IsStrictReference(ref)` 做为参数调用 `ToObject(GetBase(ref))` 的 `[[Delete]]` 内置方法的结果。
5. 否则，ref 是到环境记录项绑定的 `Reference`，所以：如果 `IsStrictReference(ref)` 为 `true`，抛出一个 `SyntaxError` 异常。令 `bindings` 为 `GetBase(ref)`。返回以 `GetReferencedName(ref)` 为参数调用绑定的 `DeleteBinding` 具体方法的结果。

当 `delete` 运算符出现在 `strict` 模式代码中的时候, 若 `UnaryExpression` 是到变量, 函数形参或者函数名的直接引用则抛出一个 `SyntaxError` 异常。此外若 `delete` 运算符出现在严格模式代码中且要删除的属性具有特性 `{ [[Configurable]]: false }`, 抛出一个 `TypeError` 异常。

void 运算符

产生式 `UnaryExpression : void UnaryExpression` 按照下面的过程执行：

1. 令 `expr` 为解释执行 `UnaryExpression` 的结果。
2. 调用 `GetValue(expr)`.
3. 返回 `undefined`.

`GetValue` 一定要调用, 即使它的值无用, 但是可能会有可见的附加效果。

typeof 运算符

产生式 `UnaryExpression : typeof UnaryExpression` 按照下面的过程执行：

1. 令 `val` 为解释执行 `UnaryExpression` 的结果。
2. 如果 `Type(val)` 为 `Reference`, 则: 如果 `IsUnresolvableReference(val)` 为 `true`, 返回 `"undefined"`。令 `val` 为 `GetValue(val)`.
3. 返回根据表 20 由 `Type(val)` 决定的字符串。

typeof 运算符结果

val 类型	结果
Undefined	"undefined"
Null	"null"
Boolean	"boolean"
Number	"number"
String	"string"
Object (原生, 且没有实现 <code>[[call]]</code>)	"object"
Object (原生或者宿主且实现了 <code>[[call]]</code>)	"function"
Object (宿主且没实现 <code>[[call]]</code>)	由实现定义, 但不能是 <code>"undefined"</code> , <code>"boolean"</code> , <code>"number"</code> , or <code>"string"</code>

前自增运算符

产生式 `UnaryExpression : ++ UnaryExpression` 按照下面的过程执行：

1. 令 `expr` 为解释执行 `UnaryExpression` 的结果 .
2. 抛出一个 `SyntaxError` 异常当以下条件全部为真 :
 - `Type(expr)` 为 `Reference`
 - `IsStrictReference(expr)` 为 `true`
 - `Type(GetBase(expr))` 为环境记录项
 - `GetReferencedName(expr)` 是 `"eval"` 或 `"arguments"`

前自减运算符

产生式 `UnaryExpression : -- UnaryExpression` 按照下面的过程执行 :

1. 令 `expr` 为解释执行 `UnaryExpression` 的结果 .
2. 抛出一个 `SyntaxError` 异常, 当以下条件全部为真 :
 - `Type(expr)` 为 `Reference`
 - `IsStrictReference(expr)` 为 `true`
 - `Type(GetBase(expr))` 为环境记录项
 - `GetReferencedName(expr)` 是 `"eval"` 或 `"arguments"`

一元 + 运算符

一元+运算符将其操作数转换为 `Number` 类型。

产生式 `UnaryExpression : + UnaryExpression` 按照下面的过程执行 :

1. 令 `expr` 为解释执行 `UnaryExpression` 的结果 .
2. 返回 `ToNumber(GetValue(expr))`.

一元 - 运算符

一元+运算符将其操作数转换为 `Number` 类型并反转其正负。注意负的+0产生-0, 负的-0产生+0。

产生式 `UnaryExpression : - UnaryExpression` 按照下面的过程执行 :

1. 令 `expr` 为解释执行 `UnaryExpression` 的结果 .
2. 令 `oldValue` 为 `ToNumber(GetValue(expr))`.
3. 如果 `oldValue` is NaN , return NaN.
4. 返回 `oldValue` 取负 (即, 算出一个数字相同但是符号相反的值) 的结果。

按位非运算符

产生式 `UnaryExpression : ~ UnaryExpression` 按照下面的过程执行 :

1. 令 `expr` 为解释执行 `UnaryExpression` 的结果 .
2. 令 `oldValue` 为 `ToInt32(GetValue(expr))`.
3. 返回 `oldValue` 按位取反的结果。结果为 32 位有符号整数。

逻辑非运算符

产生式 `UnaryExpression : ! UnaryExpression` 按照下面的过程执行：

1. 令 `expr` 为解释执行 `UnaryExpression` 的结果 .
2. 令 `oldValue` 为 `ToBoolean(GetValue(expr))`.
3. 如果 `oldValue` 为 `true`，返回 `false`.
4. 返回 `true`.

乘法运算符

语法：

`MultiplicativeExpression :`

`UnaryExpression`

`MultiplicativeExpression * UnaryExpression`

`MultiplicativeExpression / UnaryExpression`

`MultiplicativeExpression % UnaryExpression`

语义：

产生式 `MultiplicativeExpression : 'MultiplicativeExpression'@ 'UnaryExpression`, 其中 `@` 表示上面定义中的运算符之一，按照下面的过程执行：

1. 令 `left` 为解释执行 `MultiplicativeExpression` 的结果 .
2. 令 `leftValue` 为 `GetValue(left)`.
3. 令 `right` 为解释执行 `UnaryExpression` 的结果 .
4. 令 `rightValue` 为 `GetValue(right)`.
5. 令 `leftNum` 为 `ToNumber(leftValue)`.
6. 令 `rightNum` 为 `ToNumber(rightValue)`.
7. 返回将特定运算符 (`*`, `/`, or `%`) 作用于 `leftNum` 和 `rightNum` 的结果。参见 11.5.1, 11.5.2, 11.5.3 后的注解。

使用 * 运算符

*运算符表示乘法，产生操作数的乘积。乘法运算满足交换律。因为精度问题，乘法不总是满足结合律。

浮点数的乘法遵循 IEEE 754 二进制双精度幅度浮点算法规则：

- 若两个操作数之一为 NaN，结果为 NaN。
- 假如两个操作数的正负号相同，结果就是正的，如果不同就是负的。
- 无穷大被零乘结果是 NaN。
- 无穷大被无穷大乘结果就是无穷大。符号按照前面说过的规则决定。
- 无穷大被有穷的非零值乘结果是带正负号的无穷大。符号仍然按照前面说过的规则决定。
- 其它情况下，既没有无穷大也没有 NaN 参与运算，结果计算出来后会按照 IEEE 754 round-to-nearest 模式取到最接近的能表示的数。如果值过大不能表示，则结果为相应的正负无穷大。如果值过小不能表示，则结果为相应的正负零。ECMAScript 要求支持 IEEE 754 规定的渐进下溢。

使用 / 运算符

/运算符表示除法，产生操作数的商。左操作数是被除数，右操作数是除数。ECMAScript 不支持整数除法。所有除法运算的操作数和结果都是双精度浮点数。浮点数的除法遵循 IEEE 754 二进制双精度幅度浮点算法规则：

- 若两个操作数之一为 NaN，结果为 NaN。
- 假如两个操作数的正负号相同，结果就是正的，如果不同就是负的。
- 无穷大被零乘结果是 NaN。
- 无穷大被无穷大除结果是 NaN。
- 无穷大被零除结果是无穷大。符号按照前面说过的规则决定。
- 无穷大被非零有穷的值除结果是有正负号的无穷大。符号按照前面说过的规则决定。
- 有穷的非零值被无穷大除结果是零。符号按照前面说过的规则决定。
- 零被零除结果是 NaN；零被其它有穷数除结果是零，符号按照前面说过的规则决定。
- 有穷的非零值被零除结果是有正负号的无穷大。符号按照前面说过的规则决定。
- 其它情况下，既没有无穷大也没有 NaN 参与运算，结果计算出来后会按照 IEEE 754 round-to-nearest 模式取到最接近的能表示的数。如果值过大不能表示，则结果为相应的正负无穷大。如果值过小不能表示，则结果为相应的正负零。ECMAScript 要求支持 IEEE 754 规定的渐进下溢。

使用 % 运算符

%运算符产生其运算符在除法中的余数。左操作数是被除数，右操作数是除数。

在 C 和 C++ 中，余数运算符只接受整数为操作数；在 ECMAScript，它还接受浮点操作数。

浮点数使用 % 运算符的余数运算与 IEEE 754 所定义的 "remainder" 运算不完全相同。IEEE 754 "remainder" 运算做邻近取整除法的余数计算，而不是舍尾除法，这样它的行为跟通常意义上的整数余数运算符行为不一致。而 ECMAScript 语言定义浮点操作 % 为与 Java 取余运算符一致；可以参照 C 库中的函数 fmod。

ECMAScript 浮点数的取余法遵循 IEEE 754 二进制双精度幅度浮点算法规则：

- 若两个操作数之一为 NaN，结果为 NaN。
- 结果的符号等于被除数。
- 若被除数是无穷大或者除数是零，或者两者皆是，结果就是 NaN。
- 若被除数有穷而除数为无穷大，结果为被除数。
- 若被除数为零且除数非零且有穷，结果与被除数相同。
- 其它情况下，既没有 0，无穷大也没有 NaN 参与运算，从被除数 n 和除数 d 得到浮点数余数 r 以数学关系式 $r = n - (d \times q)$ 定义，其中 q 是个整数，在 n/d 为负时为负，在 n/d 为正时为正，它应该在不超过 n 和 d 的商的前提下尽可能大。结果计算出来后会按照 IEEE 754 round-to-nearest 模式取到最接近的能表示的数。

加法运算符

语法：

AdditiveExpression :

MultiplicativeExpression

AdditiveExpression + MultiplicativeExpression

AdditiveExpression - MultiplicativeExpression

加号运算符 (+)

The addition operator either performs string concatenation or numeric addition.

产生式 AdditiveExpression : AdditiveExpression + MultiplicativeExpression 按照下面的过程执行：

1. 令 lref 为解释执行 AdditiveExpression 的结果。
2. 令 lval 为 GetValue(lref)。
3. 令 rref 为解释执行 MultiplicativeExpression 的结果。
4. 令 rval 为 GetValue(rref)。

5. 令 lprim 为 ToPrimitive(lval).
6. 令 rprim 为 ToPrimitive(rval).
7. 如果 Type(lprim) 为 String 或者 Type(rprim) 为 String, 则: 返回由 ToString(lprim) 和 ToString(rprim) 连接而成的字符串
8. 返回将加法运算作用于 ToNumber(lprim) 和 ToNumber(rprim) 的结果。
参见 11.6.3 后的注解。

在步骤 5 和 6 中的 ToPrimitive 调用没有提供 hint, 除了 Date 对象之外所有 ECMAScript 对象将缺少 hint 的情况当做 Number 处理; Date 对象将缺少 hint 的情况当做 hint 为字符串。宿主对象可能将缺少 hint 的情况当做别的处理。

步骤 7 与关系运算符比较算法中的步骤 3 不同, 它使用逻辑或运算符而不是逻辑与运算符

减号运算符 (-)

产生式 AdditiveExpression : AdditiveExpression - MultiplicativeExpression 按照下面的过程执行 :

1. 令 lref 为解释执行 AdditiveExpression 的结果 .
2. 令 lval 为 GetValue(lref).
3. 令 rref 为解释执行 MultiplicativeExpression 的结果 .
4. 令 rval 为 GetValue(rref).
5. 令 lnum 为 ToNumber(lval).
6. 令 rnum 为 ToNumber(rval).
7. 返回返回将减法运算作用于 ToNumber(lprim) 和 ToNumber(rprim) 的结果。参见 11.6.3 后的注解。

加法作用于数字

+运算符作用于两个数字类型的操作数时表示加法, 产生两个操作数之和。-运算符表示减法, 产生两个数字之差。

加法是满足交换律的运算, 但是不总满足结合律。

加法遵循 IEEE 754 二进制双精度幅度浮点算法规则:

- 两个正负号相反的无穷之和为 NaN。
- 两个正负号相同的无穷大之和是具有相同正负的无穷大。
- 无穷大和有穷值之和等于操作数中的无穷大。
- 两个负零之和为-0。
- 两个正零, 或者两个正负号相反的零之和为+0。
- 零与非零有穷值之和等于非零的那个操作数。

- 两个大小相等，符号相反的非零有穷值之和为+0。
- 其它情况下，既没有无穷大也没有 NaN 或者零参与运算，并且操作数要么大小不等，要么符号相同，结果计算出来后会按照 IEEE 754 round-to-nearest 模式取到最接近的能表示的数。如果值过大不能表示，则结果为相应的正负无穷大。如果值过小不能表示，则结果为相应的正负零。ECMAScript 要求支持 IEEE 754 规定的渐进下溢。

-运算符作用于两个数字类型时表示减法，产生两个操作数之差。左边操作数是被减数右边是减数。给定操作数 a 和 b，总是有 $a-b$ 产生与 $a + (-b)$ 产生相同结果。

位运算移位运算符

语法：

ShiftExpression :

AdditiveExpression

ShiftExpression << AdditiveExpression

ShiftExpression >> AdditiveExpression

ShiftExpression >>> AdditiveExpression

左移运算符

表示对左操作数做右操作数指定次数的按位左移操作。

产生式 ShiftExpression : ShiftExpression << AdditiveExpression 按照下面的过程执行：

1. 令 lref 为解释执行 ShiftExpression 的结果。
2. 令 lval 为 GetValue(lref)。
3. 令 rref 为解释执行 AdditiveExpression 的结果。
4. 令 rval 为 GetValue(rref)。
5. 令 lnum 为 ToInt32(lval)。
6. 令 rnum 为 ToUint32(rval)。
7. 令 shiftCount 为用掩码算出 rnum 的最后五个比特位，即计算 $rnum \& 0x1F$ 的结果。
8. 返回 lnum 左移 shiftCount 比特位的结果。结果是一个有符号 32 位整数。

带符号右移运算符

filling bitwise right shift operation on the left operand by the amount specified by the right operand.

产生式 `ShiftExpression : ShiftExpression >> AdditiveExpression` 按照下面的过程执行：

1. 令 `lref` 为解释执行 `ShiftExpression` 的结果。
2. 令 `lval` 为 `GetValue(lref)`。
3. 令 `rref` 为解释执行 `AdditiveExpression` 的结果。
4. 令 `rval` 为 `GetValue(rref)`。
5. 令 `Inum` 为 `ToInt32(lval)`。
6. 令 `rnum` 为 `ToUInt32(rval)`。
7. 令 `shiftCount` 为用掩码算出 `rnum` 的最后五个比特位，即计算 `rnum & 0x1F` 的结果。
8. 返回 `Inum` 带符号扩展的右移 `shiftCount` 比特位的结果。The most significant bit is propagated. 结果是一个有符号 32 位整数。

无符号右移运算符

Performs a zero-filling bitwise right shift operation on the left operand by the amount specified by the right operand.

产生式 `ShiftExpression : ShiftExpression >>> AdditiveExpression` 按照下面的过程执行：

1. 令 `lref` 为解释执行 `ShiftExpression` 的结果。
2. 令 `lval` 为 `GetValue(lref)`。
3. 令 `rref` 为解释执行 `AdditiveExpression` 的结果。
4. 令 `rval` 为 `GetValue(rref)`。
5. 令 `Inum` 为 `ToUInt32(lval)`。
6. 令 `rnum` 为 `ToUInt32(rval)`。
7. 令 `shiftCount` 为用掩码算出 `rnum` 的最后五个比特位，即计算 `rnum & 0x1F` 的结果。
8. 返回 `Inum` 做 0 填充右移 `shiftCount` 比特位的结果。缺少的比特位填 0。结果是一个无符号 32 位整数。

比较运算符

语法：

`RelationalExpression :`

`ShiftExpression`

```

RelationalExpression < ShiftExpression
RelationalExpression > ShiftExpression
RelationalExpression <= ShiftExpression
RelationalExpression >= ShiftExpression
RelationalExpression instanceof ShiftExpression

RelationalExpression in
ShiftExpressionRelationalExpressionNoIn :
ShiftExpression

RelationalExpressionNoIn < ShiftExpression
RelationalExpressionNoIn > ShiftExpression
RelationalExpressionNoIn <= ShiftExpression
RelationalExpressionNoIn >= ShiftExpression

RelationalExpressionNoIn instanceof ShiftExpression

```

“NoIn”变体用以避免混淆关系表达式中的 **in** 运算符和 **for** 语句中的 **in** 运算符。

语义：

执行关系比较运算符的结果总是 **Boolean** 类型。表示是否由运算符指定的关系对两操作数成立。

RelationalExpressionNoIn 跟 **RelationalExpression** 完全按相同的方式执行，出了 **RelationalExpressionNoIn** 要代替 **RelationalExpression** 被执行。

The Less-than Operator (<)

产生式 **RelationalExpression : RelationalExpression < ShiftExpression** 按照下面的过程执行：

1. 令 **lref** 为解释执行 **RelationalExpression** 的结果。
2. 令 **lval** 为 **GetValue(lref)**。
3. 令 **rref** 为解释执行 **ShiftExpression** 的结果。
4. 令 **rval** 为 **GetValue(rref)**。
5. 令 **r** 为抽象关系比较算法 **lval < rval**(参见 11.8.5) 的结果
6. 如果 **r** 为 **undefined**，返回 **false**。否则，返回 **r**。

The Greater-than Operator (>)

产生式 $\text{RelationalExpression} : \text{RelationalExpression} > \text{ShiftExpression}$ 按照下面的过程执行：

1. 令 $lref$ 为解释执行 $\text{RelationalExpression}$ 的结果。
2. 令 $lval$ 为 $\text{GetValue}(lref)$.
3. 令 $rref$ 为解释执行 ShiftExpression 的结果。
4. 令 $rval$ 为 $\text{GetValue}(rref)$.
5. 令 r 为为抽象关系比较算法 $lval < rval$ (参见 11.8.5) 的结果, 参数 LeftFirst 设为 $false$
6. 如果 r 为 $undefined$, 返回 $false$. 否则, 返回 r .

The Less-than-or-equal Operator (<=)

产生式 $\text{RelationalExpression} : \text{RelationalExpression} <= \text{ShiftExpression}$ 按照下面的过程执行：

1. 令 $lref$ 为解释执行 $\text{RelationalExpression}$ 的结果。
2. 令 $lval$ 为 $\text{GetValue}(lref)$.
3. 令 $rref$ 为解释执行 ShiftExpression 的结果。
4. 令 $rval$ 为 $\text{GetValue}(rref)$.
5. 令 r 为为抽象关系比较算法 $rval < lval$ (参见 11.8.5) 的结果, 参数 LeftFirst 设为 $false$
6. 如果 r 为 $true$ 或者 $undefined$, 返回 $false$. 否则, 返回 $true$.

The Greater-than-or-equal Operator (>=)

产生式 $\text{RelationalExpression} : \text{RelationalExpression} >= \text{ShiftExpression}$ 按照下面的过程执行：

1. 令 $lref$ 为解释执行 $\text{RelationalExpression}$ 的结果。
2. 令 $lval$ 为 $\text{GetValue}(lref)$.
3. 令 $rref$ 为解释执行 ShiftExpression 的结果。
4. 令 $rval$ 为 $\text{GetValue}(rref)$.
5. 令 r 为抽象关系比较算法 $lval < rval$ (参见 11.8.5) 的结果
6. 如果 r 为 $true$ 或者 $undefined$, 返回 $false$. 否则, 返回 $true$.

抽象关系比较算法

以 x 和 y 为值进行小于比较 ($x < y$ 的比较)，会产生的结果可为 `true`, `false` 或 `undefined` (这说明 x 、 y 中最少有一个操作数是 `NaN`)。除了 x 和 y ，这个算法另外需要一个名为 `LeftFirst` 的布尔值标记作为参数。这个标记用于解析顺序的控制，因为操作数 x 和 y 在执行的时候会有潜在可见的副作用。

`LeftFirst` 标志是必须的，因为 `ECMAScript` 规定了表达式是从左到右顺序执行的。`LeftFirst` 的默认值是 `true`，这表明在相关的表达式中，参数 x 出现在参数 y 之前。如果 `LeftFirst` 值是 `false`，情况会相反，操作数的执行必须是先 y 后 x 。这样的一个小于比较的执行步骤如下：

1. 如果 `LeftFirst` 标志是 `true`，那么
 1. 让 px 为调用 `ToPrimitive(x, hint Number)` 的结果。
 2. 让 py 为调用 `ToPrimitive(y, hint Number)` 的结果。
 2. 否则解释执行的顺序需要反转，从而保证从左到右的执行顺序
 1. 让 py 为调用 `ToPrimitive(y, hint Number)` 的结果。
 2. 让 px 为调用 `ToPrimitive(x, hint Number)` 的结果。
 3. 如果 `Type(px)` 和 `Type(py)` 得到的结果不都是 `String` 类型，那么
 1. 让 nx 为调用 `ToNumber(px)` 的结果。因为 px 和 py 都已经是基本数据类型 (`primitive values` 也作原始值)，其执行顺序并不重要。
 2. 让 ny 为调用 `ToNumber(py)` 的结果。
 3. 如果 nx 是 `NaN`，返回 `undefined`
 4. 如果 ny 是 `NaN`，返回 `undefined`
 5. 如果 nx 和 ny 的数字值相同，返回 `false`
 6. 如果 nx 是 `+0` 且 ny 是 `-0`，返回 `false`
 7. 如果 nx 是 `-0` 且 ny 是 `+0`，返回 `false`
 8. 如果 nx 是 `+\infty`，返回 `false`
 9. 如果 ny 是 `+\infty`，返回 `true`
 10. 如果 ny 是 `-\infty`，返回 `false`
 11. 如果 nx 是 `-\infty`，返回 `true`
 12. 如果 nx 数学上的值小于 ny 数学上的值 (注意这些数学值都不能是无限的且不能都为 0)，返回 `true`。否则返回 `false`。
 4. 否则， px 和 py 都是 `Strings` 类型
 1. 如果 py 是 px 的一个前缀，返回 `false`。(当字符串 q 的值可以是字符串 p 和一个其他的字符串 r 拼接而成时，字符串 p 就是 q 的前缀。注意：任何字符串都是自己的前缀，因为 r 可能是空字符串。)
 2. 如果 px 是 py 的前缀，返回 `true`。
 3. 让 k 成为最小的非负整数，能使得在 px 字符串中位置 k 的字符与字符串 py 字符串中位置 k 的字符不相同。(这里必须有一个 k ，使得互相都不是对方的前缀)
 4. 让 m 成为字符串 px 中位置 k 的字符的编码单元值。
 5. 让 n 成为字符串 py 中位置 k 的字符的编码单元值。
 6. 如果 $n < m$ ，返回 `true`。否则，返回 `false`。

使用或代替的时候要注意，这里的步骤 3 和加号操作符 `+` 算法 (11.6.1) 的步骤 7 的区别。

String 类型的比较使用了其编码单元值的作为一个简单的词法表序列去比较。这里不打算使用更复杂的、语义化的字符或字符串序列，和 **Unicode** 规范的整理序列进行比较。因此，字符串的值和其对应的 **Unicode** 标准的值是不相同的。实际上，这个算法假定了所有字符串已经是正常化的格式。同时要注意，对于字符串拼接追加的字符的时候，**UTF-16** 编码单元值的词法表序列是不同于代码点值的序列的。

The instanceof operator

产生式 **RelationalExpression: RelationalExpression instanceof ShiftExpression** 按照下面的过程执行：

1. 令 **lref** 为解释执行 **RelationalExpression** 的结果。
2. 令 **lval** 为 **GetValue(lref)**。
3. 令 **rref** 为解释执行 **ShiftExpression** 的结果。
4. 令 **rval** 为 **GetValue(rref)**。
5. 如果 **Type(rval)** 不是 **Object**，抛出一个 **TypeError** 异常。
6. 如果 **rval** 没有 **[[HasInstance]]** 内置方法，抛出一个 **TypeError** 异常。
7. 返回以参数 **lval** 调用 **rval** 的 **[[HasInstance]]** 内置方法的结果

The in operator

产生式 **RelationalExpression : RelationalExpression in ShiftExpression** 按照下面的过程执行：

1. 令 **lref** 为解释执行 **RelationalExpression** 的结果。
2. 令 **lval** 为 **GetValue(lref)**。
3. 令 **rref** 为解释执行 **ShiftExpression** 的结果。
4. 令 **rval** 为 **GetValue(rref)**。
5. 如果 **Type(rval)** 不是 **Object**，抛出一个 **TypeError** 异常。
6. 返回以参数 **ToString(lval)** 调用 **rval** 的 **[[HasProperty]]** 内置方法的结果

等值运算符

语法：

EqualityExpression :

RelationalExpression

EqualityExpression == RelationalExpression

EqualityExpression != RelationalExpression

```

EqualityExpression == RelationalExpression

EqualityExpression !=
RelationalExpressionEqualityExpressionNoIn :

RelationalExpressionNoIn

EqualityExpressionNoIn == RelationalExpressionNoIn

EqualityExpressionNoIn != RelationalExpressionNoIn

EqualityExpressionNoIn === RelationalExpressionNoIn

EqualityExpressionNoIn != RelationalExpressionNoIn

```

语义：

执行相等比较运算符的结果总是 **Boolean** 类型。表示是否由运算符指定的关系对两操作数成立。

EqualityExpressionNoIn 跟 **EqualityExpression** 完全按相同的方式执行，出了 **RelationalExpressionNoIn** 要代替 **RelationalExpression** 被执行。

The Equals Operator (==)

产生式 **EqualityExpression** : **EqualityExpression** == **RelationalExpression** 按照下面的过程执行：

1. 令 **lref** 为解释执行 **EqualityExpression** 的结果。
2. 令 **lval** 为 **GetValue(lref)**。
3. 令 **rref** 为解释执行 **RelationalExpression** 的结果。
4. 令 **rval** 为 **GetValue(rref)**。
5. 返回做用相等比较算法于 **rval == lval**(参见 11.9.3) 的结果

The Does-not-equals Operator (!=)

产生式 **EqualityExpression** : **EqualityExpression** != **RelationalExpression** 按照下面的过程执行：

1. 令 **lref** 为解释执行 **EqualityExpression** 的结果。
2. 令 **lval** 为 **GetValue(lref)**。
3. 令 **rref** 为解释执行 **RelationalExpression** 的结果。
4. 令 **rval** 为 **GetValue(rref)**。
5. 令 **r** 为做用相等比较算法于 **rval == lval**(参见 11.9.3) 的结果
6. 如果 **r** 为 **true**，返回 **false**。否则，返回 **true**。

抽象相等比较算法

比较运算 $x==y$, 其中 x 和 y 是值, 产生 **true** 或者 **false**。这样的比较按如下方式进行:

1. 若 $Type(x)$ 与 $Type(y)$ 相同, 则
 1. 若 $Type(x)$ 为 **Undefined**, 返回 **true**。
 2. 若 $Type(x)$ 为 **Null**, 返回 **true**。
 3. 若 $Type(x)$ 为 **Number**, 则
 - i. 若 x 为 **NaN**, 返回 **false**。
 - ii. 若 y 为 **NaN**, 返回 **false**。
 - iii. 若 x 与 y 为相等数值, 返回 **true**。
 - iv. 若 x 为 $+0$ 且 y 为 -0 , 返回 **true**。
 - v. 若 x 为 -0 且 y 为 $+0$, 返回 **true**。
 - vi. 返回 **false**。
 4. 若 $Type(x)$ 为 **String**, 则当 x 和 y 为完全相同的字符序列 (长度相等且相同字符在相同位置) 时返回 **true**。否则, 返回 **false**。
 5. 若 $Type(x)$ 为 **Boolean**, 当 x 和 y 为同为 **true** 或者同为 **false** 时返回 **true**。否则, 返回 **false**。
 6. 当 x 和 y 为引用同一对象时返回 **true**。否则, 返回 **false**。
2. 若 x 为 **null** 且 y 为 **undefined**, 返回 **true**。
3. 若 x 为 **undefined** 且 y 为 **null**, 返回 **true**。
4. 若 $Type(x)$ 为 **Number** 且 $Type(y)$ 为 **String**, 返回 $comparison\ x == ToNumber(y)$ 的结果。
5. 若 $Type(x)$ 为 **String** 且 $Type(y)$ 为 **Number**,
6. 返回比较 $ToNumber(x) == y$ 的结果。
7. 若 $Type(x)$ 为 **Boolean**, 返回比较 $ToNumber(x) == y$ 的结果。
8. 若 $Type(y)$ 为 **Boolean**, 返回比较 $x == ToNumber(y)$ 的结果。
9. 若 $Type(x)$ 为 **String** 或 **Number**, 且 $Type(y)$ 为 **Object**, 返回比较 $x == ToPrimitive(y)$ 的结果。
10. 若 $Type(x)$ 为 **Object** 且 $Type(y)$ 为 **String** 或 **Number**, 返回比较 $ToPrimitive(x) == y$ 的结果。
11. 返回 **false**。

按以上相等之定义:

- 字符串比较可以按这种方式强制执行: $"" + a == "" + b$ 。
- 数值比较可以按这种方式强制执行: $+a == +b$ 。
- 布尔值比较可以按这种方式强制执行: $!a == !b$ 。

等值比较操作保证以下不变:

- $A != B$ 等价于 $!(A==B)$ 。
- $A == B$ 等价于 $B == A$, 除了 A 与 B 的执行顺序。

相等运算符不总是传递的。例如，两个不同的 **String** 对象，都表示相同的字符串值；`==`运算符认为每个 **String** 对象都与字符串值相等，但是两个字符串对象互不相等。例如：

- `new String("a") == "a"` 和 `"a" == new String("a")` 皆为 **true**。
- `new String("a") == new String("a")` 为 **false**。

字符串比较使用的方式是简单地检测字符编码单元序列是否相同。不会做更复杂的、基于语义的字符或者字符串相等的定义以及 **Unicode** 规范中定义的 **collating order**。所以 **Unicode** 标准中认为相等的 **String** 值可能被检测为不等。实际上这一算法认为两个字符串已经是经过规范化的形式。

严格等于运算符 (`===`)

产生式 **EqualityExpression** : **EqualityExpression** `===` **RelationalExpression** 按照下面的过程执行：

1. 令 **lref** 为解释执行 **EqualityExpression** 的结果。
2. 令 **lval** 为 `GetValue(lref)`。
3. 令 **rref** 为解释执行 **RelationalExpression** 的结果。
4. 令 **rval** 为 `GetValue(rref)`。
5. 返回做用严格相等比较算法于 `rval === lval` (参见 11.9.6) 的结果

The Strict Does-not-equal Operator (`!==`)

产生式 **EqualityExpression** : **EqualityExpression** `!==` **RelationalExpression** 按照下面的过程执行：

1. 令 **lref** 为解释执行 **EqualityExpression** 的结果。
2. 令 **lval** 为 `GetValue(lref)`。
3. 令 **rref** 为解释执行 **RelationalExpression** 的结果。
4. 令 **rval** 为 `GetValue(rref)`。
5. 令 **r** 为做用严格相等比较算法于 `rval === lval` (参见 11.9.6) 的结果
6. 如果 **r** 为 **true**，返回 **false**。否则，返回 **true**。

严格等于比较算法

比较 `x===y`，**x** 和 **y** 为值，需要产出 **true** 或 **false**。比较过程如下：

1. 如果 `Type(x)` 与 `Type(y)` 的结果不一致，返回 **false**，否则
2. 如果 `Type(x)` 结果为 **Undefined**，返回 **true**
3. 如果 `Type(x)` 结果为 **Null**，返回 **true**

4. 如果 `Type(x)` 结果为 `Number`, 则
 1. 如果 `x` 为 `NaN`, 返回 `false`
 2. 如果 `y` 为 `NaN`, 返回 `false`
 3. 如果 `x` 与 `y` 为同一个数字, 返回 `true`
 4. 如果 `x` 为 `+0`, `y` 为 `-0`, 返回 `true`
 5. 如果 `x` 为 `-0`, `y` 为 `+0`, 返回 `true`
 6. 返回 `false`
5. 如果 `Type(x)` 结果为 `String`, 如果 `x` 与 `y` 为完全相同的字符序列 (相同的长度和相同的字符对应相同的位置), 返回 `true`, 否则, 返回 `false`
6. 如果 `Type(x)` 结果为 `Boolean`, 如果 `x` 与 `y` 都为 `true` 或 `false`, 则返回 `true`, 否则, 返回 `false`
7. 如果 `x` 和 `y` 引用到同一个 `Object` 对象, 返回 `true`, 否则, 返回 `false`

此算法与 `SameValue` 算法在对待有符号的零和 `NaN` 上表现不同。

二进制位运算符

语法:

`BitwiseANDExpression` :

`EqualityExpression`

`BitwiseANDExpression` &

`EqualityExpressionBitwiseANDExpressionNoIn` :

`EqualityExpressionNoIn`

`BitwiseANDExpressionNoIn` &

`EqualityExpressionNoInBitwiseXORExpression` :

`BitwiseANDExpression`

`BitwiseXORExpression` ^

`BitwiseANDExpressionBitwiseXORExpressionNoIn` :

`BitwiseANDExpressionNoIn`

`BitwiseXORExpressionNoIn` ^

`BitwiseANDExpressionNoInBitwiseORExpression` :

`BitwiseXORExpression`

`BitwiseORExpression` |

`BitwiseXORExpressionBitwiseORExpressionNoIn` :

BitwiseXORExpressionNoIn

BitwiseORExpressionNoIn | BitwiseXORExpressionNoIn

语义：

产生式 $A : A' @ B$, 其中 $@$ 是上述产生式中的位运算符之一, 按照下面的过程执行：

1. 令 $lref$ 为解释执行 A 的结果 .
2. 令 $lval$ 为 $GetValue(lref)$.
3. 令 $rref$ 为解释执行 B 的结果 .
4. 令 $rval$ 为 $GetValue(rref)$.
5. 令 $lnum$ 为 $ToInt32(lval)$.
6. 令 $rnum$ 为 $ToInt32(rval)$.
7. 返回作用位运算符 $@$ 到 $lnum$ 和 $rnum$. 结果是 32 位有符号整数。

二元逻辑运算符

语法

LogicalANDExpression :

BitwiseORExpression

LogicalANDExpression &&

BitwiseORExpressionLogicalANDExpressionNoIn :

BitwiseORExpressionNoIn

LogicalANDExpressionNoIn &&

BitwiseORExpressionNoInLogicalORExpression :

LogicalANDExpression

LogicalORExpression ||

LogicalANDExpressionLogicalORExpressionNoIn :

LogicalANDExpressionNoIn

LogicalORExpressionNoIn || LogicalANDExpressionNoIn

语义

产生式 `LogicalANDExpression : LogicalANDExpression && BitwiseORExpression` 按照下面的过程执行：

1. 令 `lref` 为解释执行 `LogicalANDExpression` 的结果。
2. 令 `lval` 为 `GetValue(lref)`。
3. 如果 `ToBoolean(lval)` 为 `false`，返回 `lval`。
4. 令 `rref` 为解释执行 `BitwiseORExpression` 的结果。
5. 返回 `GetValue(rref)`。

产生式 `LogicalORExpression : LogicalORExpression || LogicalANDExpression` 按照下面的过程执行：

1. 令 `lref` 为解释执行 `LogicalORExpression` 的结果。
2. 令 `lval` 为 `GetValue(lref)`。
3. 如果 `ToBoolean(lval)` 为 `true`，返回 `lval`。
4. 令 `rref` 为解释执行 `LogicalANDExpression` 的结果。
5. 返回 `GetValue(rref)`。

`LogicalANDExpressionNoIn` 和 `LogicalORExpressionNoIn` 执行完全按照 `LogicalANDExpression` 和 `LogicalORExpression` 相同的方式，`BitwiseORExpressionNoIn` 和 `LogicalORExpressionNoIn` 替代了 `BitwiseORExpression` 和 `LogicalORExpression` 除外。

由 `&&` 或者 `||` 运算符产生的值不是必须为 `Boolean` 类型，产生的值始终为两个运算表达式的结果之一。

条件运算符

语法：

`ConditionalExpression :`

`LogicalORExpression`

`LogicalORExpression ? AssignmentExpression :`

`AssignmentExpressionConditionalExpressionNoIn :`

`LogicalORExpressionNoIn`

`LogicalORExpressionNoIn ? AssignmentExpressionNoIn :`

`AssignmentExpressionNoIn`

语义

产生式 ConditionalExpression : LogicalORExpression ?

AssignmentExpression : AssignmentExpression 按照下面的过程执行：

1. 令 lref 为解释执行 LogicalORExpression 的结果。
2. 如果 ToBoolean(GetValue(lref)) 为 true，那么：令 trueRef 为解释执行第一个 AssignmentExpression 的结果。返回 GetValue(trueRef)。
3. Else 令 falseRef 为解释执行第二个 AssignmentExpression 的结果。返回 GetValue(falseRef)。

ConditionalExpressionNoIn 执行完全按照 ConditionalExpression 相同的方式，除了 AssignmentExpression 和 AssignmentExpressionNoIn 替代了第一个 AssignmentExpression 和第二个 AssignmentExpression。

ECMAScript 中的 ConditionalExpression 跟 C 和 Java 有一点点不同，它允许第二个子表达式是个 Expression 但是限制第三个表达式必须是 ConditionalExpression。ECMAScript 中这个差别的依据是可以允许赋值表达式出现在条件的任意一侧同时避免逗号表达式作为中间的表达式时无用且易混淆的使用方式。

赋值运算符

语法

AssignmentExpression :

ConditionalExpression

LeftHandSideExpression = AssignmentExpression

LeftHandSideExpression AssignmentOperator

AssignmentExpressionAssignmentExpressionNoIn :

ConditionalExpressionNoIn

LeftHandSideExpression = AssignmentExpressionNoIn

LeftHandSideExpression AssignmentOperator

AssignmentExpressionNoInAssignmentOperator : one of

*= /= %= += -= <<= >>= >>>= &= ^= |=

语义

AssignmentExpressionNoIn 执行完全按照 AssignmentExpression 相同的方式，除了 ConditionalExpressionNoIn 替代了 ConditionalExpression。

简单赋值

产生式 $\text{AssignmentExpression} : \text{LeftHandSideExpression} = \text{AssignmentExpression}$ 按照下面的过程执行：

1. 令 lref 为解释执行 LeftH 和 SideExpression 的结果。
2. 令 rref 为解释执行 $\text{AssignmentExpression}$ 的结果。
3. 令 rval 为 $\text{GetValue}(\text{rref})$ 。
4. 抛出一个 SyntaxError 异常，当以下条件都成立：
 - $\text{Type}(\text{lref})$ 为 Reference
 - $\text{IsStrictReference}(\text{lref})$ 为 true
 - $\text{Type}(\text{GetBase}(\text{lref}))$ 为环境记录项
 - $\text{GetReferencedName}(\text{lref})$ 为 "eval" 或 "arguments"
5. 调用 $\text{PutValue}(\text{lref}, \text{rval})$ 。
6. 返回 rval 。

当赋值发生在严格模式中时其 LeftHandSide 执行结果不能是无法解引用的引用类型。假如这样它会在赋值时抛出 ReferenceError 异常。 LeftHandSide 也不能是到具有 $\{[[\text{Writable}]]:\text{false}\}$ 特性的数据属性的引用，到具有 $\{[[\text{Set}]]:\text{undefined}\}$ 特性的访问器属性的引用，或者 $[[\text{Extensible}]]$ 内部属性值为 false 的对象上不存在的属性。这些情况都会抛出 TypeError 异常。

组合赋值

产生式 $\text{AssignmentExpression} : \text{LeftHandSideExpression}'@ = \text{AssignmentExpression}$, where $@$ represents one of the operators indicated above, 按照下面的过程执行：

1. 令 lref 为解释执行 LeftH 和 SideExpression 的结果。
2. 令 lval 为 $\text{GetValue}(\text{lref})$ 。
3. 令 rref 为解释执行 $\text{AssignmentExpression}$ 的结果。
4. 令 rval 为 $\text{GetValue}(\text{rref})$ 。
5. 令 r 为作用运算符 $@$ 于 lval 和 rval 的结果。
6. 抛出一个 SyntaxError 异常，当以下条件全部成立：
 - $\text{Type}(\text{lref})$ 为 Reference
 - $\text{IsStrictReference}(\text{lref})$ 为 true
 - $\text{Type}(\text{GetBase}(\text{lref}))$ 为环境记录项
 - $\text{GetReferencedName}(\text{lref})$ 为 "eval" 或 "arguments"
7. 调用 $\text{PutValue}(\text{lref}, r)$ 。
8. 返回 r 。

见 注 11.13.1。

逗号运算符

语法:

Expression :

AssignmentExpression

Expression , AssignmentExpression ExpressionNoIn :

AssignmentExpressionNoIn

ExpressionNoIn , AssignmentExpressionNoIn

语义:

产生式 Expression : Expression , AssignmentExpression 按照下面的过程执行 :

1. 令 lref 为解释执行 Expression 的结果 .
2. Call GetValue(lref).
3. 令 rref 为解释执行 AssignmentExpression 的结果 .
4. 返回 GetValue(rref).

ExpressionNoIn 执行完全按照 Expression 相同的方式, 除了 AssignmentExpressionNoIn 替代了 AssignmentExpression。

GetValue 必须调用, 即使它的值没有用, 因为它可能有附加效果。

语句

语法:

Statement :

Block

VariableStatement

EmptyStatement

ExpressionStatement

IfStatement

IterationStatement

ContinueStatement

BreakStatement

ReturnStatement

WithStatement

LabelledStatement

SwitchStatement

ThrowStatement

TryStatement

DebuggerStatement

语义：

一个 Statement 可以是 LabelledStatement 的一部分，这个 LabelledStatement 自身也可以是 LabelledStatement 的一部分，以此类推。当描述个别语句时引入标签的这种方式统称为“当前标签组”。一个 LabelledStatement 介绍了一个标签到一个 标签组 ，此外没有其他语义。一个 IterationStatement 或 SwitchStatement 的标签组最初包含单个 空 元素。任何其他语句的标签组最初是空的。

The result of evaluating a Statement is always a Completion value.

已知几个广泛使用的 ECMAScript 实现支持 FunctionDeclaration 当作语句使用。然而，在实现之间这种 FunctionDeclarations 应用的语义也有严重且不兼容的差异。由于这些不兼容的差异，将 FunctionDeclaration 当作 Statement 使用的结果是代码在实现之间的可移植性不可靠。建议 ECMAScript 实现禁止这样运用 FunctionDeclaration，或遇到这样的运用是发出一个警告。ECMAScript 的未来版本可能定义替代的兼容方案以在 Statement 上下文中声明函数。

块

语法：

Block :

{ StatementListopt }StatementList :

Statement

StatementList Statement

语义:

产生式 `Block : { }` 按照下面的过程执行 :

1. 返回 (normal, empty, empty)。

产生式 `Block : { StatementList }` 按照下面的过程执行 :

1. 返回解释执行 `StatementList` 的结果。

产生式 `StatementList : Statement` 按照下面的过程执行 :

1. 令 `s` 为解释执行 `Statement` 的结果。
2. 如果有一个异常被抛出, 返回 (throw, `V`, empty), 这里的 `V` 是异常。(仿佛没有抛出异常一样继续运行。)
3. 返回 `s`。

产生式 `StatementList : StatementList Statement` 按照下面的过程执行 :

1. 令 `sl` 为解释执行 `StatementList` 的结果。
2. 如果 `sl` 是个非常规完结, 返回 `sl`。
3. 令 `s` 为解释执行 `Statement` 的结果。
4. 如果有一个异常被抛出, 返回 (throw, `V`, empty), 这里的 `V` 是异常。(仿佛没有抛出异常一样继续运行。)
5. 如果 `s.value` 是 empty, 令 `V = sl.value`, 否则令 `V = s.value`。
6. 返回 (s.type, `V`, s.target)。

以上算法中步骤 5 和步骤 6 确保了 `StatementList` 的值是 `StatementList` 中最后一个产生值的 `Statement` 的值。例如以下 `eval` 函数的调用全都返回 1

```
eval("1;;;") eval("1;{}") eval("1;var a;")
```

变量语句

语法:

`VariableStatement :`

`var VariableDeclarationList ;VariableDeclarationList :`

`VariableDeclaration`

```

VariableDeclarationList ,
VariableDeclarationVariableDeclarationListNoIn :

VariableDeclarationNoIn

VariableDeclarationListNoIn ,
VariableDeclarationNoInVariableDeclaration :

Identifier InitialiseroptVariableDeclarationNoIn :

Identifier InitialiserNoInoptInitialiser :

= AssignmentExpressionInitialiserNoIn :

= AssignmentExpressionNoIn

```

一个变量语句声明依 10.5 中定义创建的变量。当创建变量时初始化为 `undefined`。当 `VariableStatement` 被执行时变量关联的 `Initialiser` 会被分配 `AssignmentExpression` 的值，而不是在变量创建时。

语义：

产生式 `VariableStatement : var VariableDeclarationList ;` 按照下面的过程执行：

1. 解释执行 `VariableDeclarationList`.
2. 返回 (normal, empty, empty).

产生式 `VariableDeclarationList : VariableDeclaration` 按照下面的过程执行：

1. 解释执行 `VariableDeclaration`.

产生式 `VariableDeclarationList : VariableDeclarationList , VariableDeclaration` 按照下面的过程执行：

1. 解释执行 `VariableDeclarationList`.
2. 解释执行 `VariableDeclaration`.

产生式 `VariableDeclaration : Identifier` 按照下面的过程执行：

1. 返回一个包含跟 `Identifier`. 完全相同的字符序列的字符串值

产生式 `VariableDeclaration : Identifier Initialiser` 按照下面的过程执行：

1. 令 `lhs` 为解释执行 `Identifier` 的结果 as described in 11.1.2.
2. 令 `rhs` 为解释执行 `Initialiser` 的结果 .
3. 令 `value` 为 `GetValue(rhs)`.

4. Call PutValue(lhs, value).
5. 返回一个包含跟 Identifier. 完全相同的字符序列的字符串值

VariableDeclaration 的字符串值用在 for-in 语句 (12.6.4) 的解释执行。

如果 VariableDeclaration 嵌套在 with 语句里并且 VariableDeclaration 里的标识符与 with 语句的对象式环境记录项关联的绑定对象的一个属性名相同, 则第 4 步将给这个属性分配值, 而不是为 Identifier 的 VariableEnvironment 绑定分配值。

产生式 Initialiser := AssignmentExpression 按照下面的过程执行 :

1. 返回解释执行 AssignmentExpression 的结果 .

产生式 VariableDeclarationListNoIn, VariableDeclarationNoIn, InitialiserNoIn 解释执行的方式与产生式 VariableDeclarationList, VariableDeclaration, Initialiser 相同, 除了他们包含的 VariableDeclarationListNoIn, VariableDeclarationNoIn, InitialiserNoIn, AssignmentExpressionNoIn 会分别替代 VariableDeclarationList, VariableDeclaration, Initialiser, AssignmentExpression 来解释执行。

严格模式的限制

如果一个 VariableDeclaration 或 VariableDeclarationNoIn 出现在 严格模式 代码 里并且其 Identifier 是 "eval" 或 "arguments", 那么这是个 SyntaxError。

空语句

语法 :

EmptyStatement :

;

语义:

产生式 EmptyStatement : ; 按照下面的过程执行 :

1. 返回 (normal, empty, empty).

表达式语句

语法:

ExpressionStatement :

[lookahead \notin {[, function]}]Expression ;

一个 ExpressionStatement 不能用一个开大括号开始，因为这可能会使它和 Block 混淆。此外，ExpressionStatement 不能用 function 关键字开始，因为这可能会使它和 FunctionDeclaration 混淆。

语义：

产生式 ExpressionStatement : [lookahead \notin {[, 'function'}]Expression; 按照下面的过程执行：

1. 令 exprRef 为解释执行 Expression 的结果。
2. 返回 (normal, GetValue(exprRef), empty).

if 语句

语法：

IfStatement :

if (Expression) Statement else Statement

if (Expression) Statement

每个 else 选择与它相关联的 if 是不确定的，应与此 else 最近的并且原本没有与其对应的 else 的可能的 if 对应。

语义：

产生式 IfStatement : if (Expression) Statement else Statement 按照下面的过程执行：

1. 令 exprRef 为解释执行 Expression 的结果。
2. 如果 ToBoolean(GetValue(exprRef)) is true , then
 1. 返回解释执行 the 的结果 first Statement.
3. Else,
 1. 返回解释执行 the 的结果 second Statement.

产生式 IfStatement : if (Expression) Statement 按照下面的过程执行：

1. 令 exprRef 为解释执行 Expression 的结果。
2. 如果 ToBoolean(GetValue(exprRef)) is false , return (normal, empty, empty).

3. 返回解释执行 **Statement** 的结果。

迭代语句

语法:

```
IterationStatement :  
  
do Statement while ( Expression );  
  
while ( Expression ) Statement  
  
for ( ExpressionNoInopt; Expressionopt ; Expressionopt )  
Statement  
  
for ( var VariableDeclarationListNoIn; Expressionopt ;  
Expressionopt ) Statement  
  
for ( LeftHandSideExpression in Expression ) Statement  
  
for ( var VariableDeclarationNoIn in Expression ) Statement
```

do-while 语句

产生式 `do Statement while (Expression);` 按照下面的过程执行：

1. 令 `V = empty`。
2. 令 `iterating` 为 `true`。
3. 只要 `iterating` 为 `true`，就重复
 1. 令 `stmt` 为解释执行 **Statement** 的结果。
 2. 如果 `stmt.value` 不是 `empty`，令 `V = stmt.value`。
 3. 如果 `stmt.type` 不是 `continue` || `stmt.target` 不在当前标签组，则
 - i. 如果 `stmt.type` 是 `break` 并且 `stmt.target` 在当前标签组内，返回 `(normal, V, empty)`。
 - ii. 如果 `stmt` 是个 非常规完结，返回 `stmt`。
 4. 令 `exprRef` 为解释执行 **Expression** 的结果。
 5. 如果 `ToBoolean(GetValue(exprRef))` 是 `false`，设定 `iterating` 为 `false`。
 4. 返回 `(normal, V, empty)`;

while 语句

产生式 `IterationStatement : while (Expression) Statement` 按照下面的过程执行：

1. 令 $V = \text{empty}$.
2. 重复
 1. 令 exprRef 为解释执行 Expression 的结果 .
 2. 如果 $\text{ToBoolean}(\text{GetValue}(\text{exprRef}))$ 是 false , 返回 $(\text{normal}, V, \text{empty})$.
 3. 令 stmt 为解释执行 Statement 的结果 .
 4. 如果 stmt.value 不是 empty , 令 $V = \text{stmt.value}$.
 5. 如果 stmt.type 不是 continue || stmt.target 不在当前标签组内, 则
 - i. 如果 stmt.type 是 break 并且 stmt.target 在当前标签组内, 则
 1. 返回 $(\text{normal}, V, \text{empty})$.
 - ii. 如果 stmt 是一个非常规完结, 返回 stmt .

for 语句

产生式 $\text{IterationStatement} : \text{for} (\text{ExpressionNoIn}_{\text{opt}} ; \text{Expression}_{\text{opt}} ; \text{Expression}_{\text{opt}}) \text{Statement}$ 按照下面的过程执行 :

1. 如果 ExpressionNoIn 是 present , 则 .
 1. 令 exprRef 为解释执行 ExpressionNoIn 的结果 .
 2. 调用 $\text{GetValue}(\text{exprRef})$. (不会用到此值。)
 2. 令 $V = \text{empty}$.
 3. 重复
 1. 如果第一个 Expression 是 present , 则
 - i. 令 testExprRef 为解释执行第一个 Expression 的结果 .
 - ii. 如果 $\text{ToBoolean}(\text{GetValue}(\text{testExprRef}))$ 是 false , 返回 $(\text{normal}, V, \text{empty})$.
 2. 令 stmt 为解释执行 Statement 的结果 .
 3. 如果 stmt.value 不是 empty , 令 $V = \text{stmt.value}$
 4. 如果 stmt.type 是 break 并且 stmt.target 在当前标签组内, 返回 $(\text{normal}, V, \text{empty})$.
 5. 如果 stmt.type 不是 continue || stmt.target 不在当前标签组内, 则
 - i. 如果 stmt 是个非常规完结, 返回 stmt .
 6. 如果第二个 Expression 是 present , 则
 - i. 令 incExprRef 为解释执行第二个 Expression 的结果 .
 - ii. 调用 $\text{GetValue}(\text{incExprRef})$. (不会用到此值 .)

产生式 $\text{IterationStatement} : \text{for} (\text{var VariableDeclarationListNoIn} ; \text{Expression}_{\text{opt}} ; \text{Expression}_{\text{opt}}) \text{Statement}$ 按照下面的过程执行 :

1. 解释执行 $\text{VariableDeclarationListNoIn}$.
2. 令 $V = \text{empty}$.
3. 重复
 1. 如果第一个 Expression 是 present , 则
 - i. 令 testExprRef 为解释执行第一个 Expression 的结果 .

- ii. 如果 `ToBoolean(GetValue(testExprRef))` 是 `false`, 则返回 `(normal, V, empty)`.
- 2. 令 `stmt` 为解释执行 `Statement` 的结果 .
- 3. 如果 `stmt.value` 不是 `empty`, 令 `V = stmt.value`.
- 4. 如果 `stmt.type` 是 `break` 并且 `stmt.target` 在当前标签组内, 返回 `(normal, V, empty)`.
- 5. 如果 `stmt.type` 不是 `continue` || `stmt.target` 不在当前标签组内, 则
- i. 如果 `stmt` 是个非常规完结, 返回 `stmt`.
- 6. 如果第二个 `Expression` 是 `present`, 则 .
- i. 令 `incExprRef` 为解释执行第二个 `Expression` 的结果 .
- ii. 调用 `GetValue(incExprRef)`. (不会用到此值 .)

for-in 语句

产生式 `IterationStatement : for (LeftHandSideExpression in Expression) Statement` 按照下面的过程执行 :

- 1. 令 `exprRef` 为解释执行 `Expression` 的结果 .
- 2. 令 `experValue` 为 `GetValue(exprRef)`.
- 3. 如果 `experValue` 是 `null` 或 `undefined`, 返回 `(normal, empty, empty)`.
- 4. 令 `obj` 为 `ToObject(experValue)`.
- 5. 令 `V = empty`.
- 6. 重复
 - 1. 令 `P` 为 `obj` 的下一个 `[[Enumerable]]` 特性为 `true` 的属性的名。如果不存在这样的属性, 返回 `(normal, V, empty)`.
 - 2. 令 `lhsRef` 为解释执行 `LeftHandSideExpression` 的结果 (它可能解释执行多次) .
 - 3. 调用 `PutValue(lhsRef, P)`.
 - 4. 令 `stmt` 为解释执行 `Statement` 的结果 .
 - 5. 如果 `stmt.value` 不是 `empty`, 令 `V = stmt.value`.
 - 6. 如果 `stmt.type` 是 `break` 并且 `stmt.target` 在当前标签组内, 返回 `(normal, V, empty)`.
 - 7. 如果 `stmt.type` 不是 `continue` || `stmt.target` 不在当前标签组内, 则
 - i. 如果 `stmt` 是非常规完结, 返回 `stmt`.

产生式 `IterationStatement : for (var VariableDeclarationNoIn in Expression) Statement` 按照下面的过程执行 :

- 1. 令 `varName` 为解释执行 `VariableDeclarationNoIn` 的结果 .
- 2. 令 `exprRef` 为解释执行 `Expression` 的结果 .
- 3. 令 `experValue` 为 `GetValue(exprRef)`.
- 4. 如果 `experValue` 是 `null` 或 `undefined`, 返回 `(normal, empty, empty)`.
- 5. 令 `obj` 为 `ToObject(experValue)`.
- 6. 令 `V = empty`.

7. 重复
 1. 令 `P` 为 `obj` 的下一个 `[[Enumerable]]` 特性为 `true` 的属性的名。如果不存在这样的属性，返回 `(normal, V, empty)`。
 2. 令 `varRef` 为解释执行 `varName` 的结果，仿佛它是个标示符引用 (11.1.2)；它可能解释执行多次。
 3. 调用 `PutValue(varRef, P)`。
 4. 令 `stmt` 为解释执行 `Statement` 的结果。
 5. 如果 `stmt.value` 不是 `empty`，令 `V = stmt.value`。
 6. 如果 `stmt.type` 是 `break` 并且 `stmt.target` 在当前标签组内，返回 `(normal, V, empty)`。
 7. 如果 `stmt.type` 不是 `continue` || `stmt.target` 不在当前标签组内，则
 - i. 如果 `stmt` 是非常规完结，返回 `stmt`。

枚举的属性（第一个算法中的步骤 6.a，第二个算法中的步骤 7.a）的机制和顺序并没有指定。在枚举过程中枚举的对象属性可能被删除。如果在枚举过程中，删除了还没有被访问到的属性，那么它将不会被访问到。如果在枚举过程中添加新属性到枚举的对象，新增加的属性也无法保证被当前执行中的枚举访问到。在任何枚举中对同一个属性名称的访问不得超过一次。

枚举一个对象的属性包括枚举其原型的属性，还有原型的原型，等等，递归；但是如果原型的一个属性是“阴影里的”那么不会枚举这个属性，因为原型链中之前某个对象有同名的属性了。当一个原型对象的一个属性是在原型链中之前对象的阴影里的，那么在决定是否枚举时不需要考虑 `[[Enumerable]]` 特性的值。

见 注 11.13.1。

continue 语句

语法：

```
ContinueStatement :
    continue ;

continue [ 此处无换行 ] Identifier ;
```

语义：

如果以下任意一个为真，那么程序被认为是语法错误的：

- 程序包含一个不带可选的 `Identifier` 的 `continue` 语句，没有直接或间接（不跨越函数边界）的嵌套在 `IterationStatement` 里。
- 程序包含一个有可选的 `Identifier` 的 `continue` 语句，这个 `Identifier` 没有出现在 `IterationStatement` 中闭合标签组里（不跨越函数边界）。

一个没有 Identifier 的 ContinueStatement 按照下面的过程执行：

1. 返回 (continue, empty, empty).

一个有可选的 Identifier 的 ContinueStatement 按照下面的过程执行：

1. 返回 (continue, empty, Identifier).

break 语句

语法：

BreakStatement :

break ;

break [此处无换行] Identifier ;

语义：

如果以下任意一个为真，那么程序被认为是语法错误的：

- 程序包含一个不带可选的 Identifier 的 break 语句，没有直接或间接（不跨越函数边界）的嵌套在 IterationStatement 或 SwitchStatement 里。
- 程序包含一个有可选的 Identifier 的 break 语句，这个 Identifier 没有出现在 Statement 中闭合标签组里（不跨越函数边界）。

一个没有 Identifier 的 BreakStatement 按照下面的过程执行：

1. 返回 (continue, empty, empty).

一个有可选的 Identifier 的 BreakStatement 按照下面的过程执行：

1. 返回 (continue, empty, Identifier).

return 语句

语法：

ReturnStatement :

return ;

return [此处无换行] Expression ;

语义：

在一个 ECMAScript 程序中包含的 `return` 语句没有在 `FunctionBody` 里面，那么就是语法错误的。一个 `return` 语句导致函数停止执行，并返回一个值给调用者。如果省略 `Expression`，返回值是 `undefined`。否则，返回值是 `Expression` 的值。

产生式 `ReturnStatement` : 'return' [no LineTerminator here] `Expressionopt` ; 按照下面的过程执行：

1. 如果 `Expression` 不是 present，返回 (return, undefined, empty).
2. 令 `exprRef` 为解释执行 `Expression` 的结果。
3. 返回 (return, GetValue(exprRef), empty).

with 语句

语法：

`WithStatement` :

`with (Expression) Statement`

`with` 语句为计算对象给当前执行环境的 词法环境 添加一个对象 环境记录项。然后，用这个增强的 词法环境 执行一个语句。最后，恢复到原来的 词法环境。

语义：

产生式 `WithStatement` : `with (Expression) Statement` 按照下面的过程执行：

1. 令 `val` 为解释执行 `Expression` 的结果。
2. 令 `obj` 为 `ToObject(GetValue(val))`.
3. 令 `oldEnv` 为运行中的执行环境的 `LexicalEnvironment`.
4. 令 `newEnv` 为以 `obj` 和 `oldEnv` 为参数调用 `NewObjectEnvironment` 的结果。
5. 设定 `newEnv` 的 `provideThis` 标志为 `true`。
6. 设定运行中的执行环境的 `LexicalEnvironment` 为 `newEnv`.
7. 令 `C` 为解释执行 `Statement` 的结果，但如果解释执行是由异常抛出，则令 `C` 为 (throw, V, empty)，这里的 `V` 是异常。(现在继续执行，仿佛没有抛出异常。)
8. 设定运行中的执行环境的 `LexicalEnvironment` 为 `oldEnv`.
9. 返回 `C`.

无论控制是从嵌入的 `Statement` 怎样离开的，不论是正常离开还是以 非常规完结 或异常，`LexicalEnvironment` 总是恢复到它之前的状态。

严格模式的限制

严格模式代码中不能包含 `WithStatement`。出现 `WithStatement` 的上下文被当作一个 `SyntaxError`。

switch 语句

语法：

```
SwitchStatement :  
  
switch ( Expression ) CaseBlockCaseBlock :  
  
{ CaseClausesopt }  
  
{ CaseClausesoptDefaultClause  
CaseClausesopt }CaseClauses :  
  
CaseClause  
  
CaseClauses CaseClauseCaseClause :  
  
case Expression : StatementListoptDefaultClause :  
  
default : StatementListopt
```

语义：

产生式 `SwitchStatement : switch (Expression) CaseBlock` 按照下面的过程执行：

1. 令 `exprRef` 为解释执行 `Expression` 的结果。
2. 令 `R` 为以 `GetValue(exprRef)` 作为参数解释执行 `CaseBlock` 的结果。
3. 如果 `R.type` 是 `break` 并且 `R.target` 在当前标签组内，返回 `(normal, R.value, empty)`。
4. 返回 `R`。

产生式 `CaseBlock : { CaseClausesopt }` 以一个给定输入参数 `input`，按照下面的过程执行：

1. 令 `V = empty`。
2. 令 `A` 为以源代码中顺序排列的 `CaseClause` 列表。
3. 令 `searching` 为 `true`。
4. 只要 `searching` 为 `true`，就重复

1. 令 C 为 A 里的下一个 CaseClause。如果没有 CaseClause 了, 返回 (normal, V, empty).
2. 令 clauseSelector 为解释执行 C 的结果 .
3. 如果 input 和 clauseSelector 是 === 操作符定义的相等, 则
 - i. 设定 searching 为 false.
 - ii. 如果 C 有一个 StatementList, 则
 1. 令 R 为解释执行 C 的 StatementList 的结果。
 2. 如果 R 是个非常规完结 , 则返回 R。
 3. 令 V =R.value
 5. 重复
 1. 令 C 为 A 里的下一个 CaseClause。如果没有 CaseClause 了, 返回 (normal, V, empty).
 2. 如果 C 有一个 StatementList, 则
 - i. 令 R 为解释执行 C 的 StatementList 的结果。
 - ii. 如果 R.value 不是 empty, 则令 V =R.value.
 - iii. 如果 R 是个非常规完结 , 则返回 (R.type,V,R.target).

产生式 CaseBlock : { CaseClauses_{opt}DefaultClause CaseClauses_{opt} } 以一个给定输入参数 input, 按照下面的过程执行 :

1. 令 V = empty.
2. 令 A 为第一个 CaseClauses 中以源代码中顺序排列的 CaseClause 列表。
3. 令 B 为第二个 CaseClauses 中以源代码中顺序排列的 CaseClause 列表。
4. 令 found 为 false.
5. 重复, 使 C 为 A 中的依次每个 CaseClause。
 1. 如果 found 是 false, 则
 - i. 令 clauseSelector 为解释执行 C 的结果 .
 - ii. 如果 input 和 clauseSelector 是 === 操作符定义的相等, 则设定 found 为 true.
 2. 如果 found 是 true, 则
 - i. 如果 C 有一个 StatementList, 则
 1. 令 R 为解释执行 C 的 StatementList 的结果。
 2. 如果 R.value 不是 empty, 则令 V =R.value.
 3. R 是个非常规完结 , 则返回 (R.type,V,R.target).
6. 令 foundInB 为 false.
7. 如果 found 是 false, 则
 1. 只要 foundInB 为 false 并且所有 B 中的元素都没有被处理, 就重复
 - i. 令 C 为 B 里的下一个 CaseClause.
 - ii. 令 clauseSelector 为解释执行 C 的结果 .
 - iii. 如果 input 和 clauseSelector 是 === 操作符定义的相等, 则
 1. 设定 foundInB 为 true.
 2. 如果 C 有一个 StatementList, 则

1. 令 R 为解释执行 C 的 StatementList 的结果。
2. 如果 R.value 不是 empty, 则令 V = R.value.
3. R 是个非常规完结, 则返回 (R.type, V, R.target).
8. 如果 foundInB 是 false 并且 DefaultClause 有个 StatementList, 则
 1. 令 R 为解释执行 DefaultClause 的 StatementList 的结果.
 2. 如果 R.value 不是 empty, 则令 V = R.value.
 3. 如果 R 是个非常规完结, 则返回 (R.type, V, R.target).
9. 重复 (注: 如果已执行步骤 7.a.i, 此循环不从 B 的开头开始。)
1. 令 C 为 B 的下一个 CaseClause。如果没有 CaseClause 了, 返回 (normal, V, empty).
2. 如果 C 有个 StatementList, 则
 - i. 令 R 为解释执行 C 的 StatementList 的结果。
 - ii. 如果 R.value 不是 empty, 则令 V = R.value.
 - iii. 如果 R 是个非常规完结, 则返回 (R.type, V, R.target).

产生式 CaseClause : case Expression : StatementList_{opt} 按照下面的过程执行:

1. 令 exprRef 为解释执行 Expression 的结果.
2. 返回 GetValue(exprRef).

解释执行 CaseClause 不会运行相关的 StatementList。它只简单的解释执行 Expression 并返回值, 这里的 CaseBlock 算法用于确定 StatementList 开始执行。

标签语句

语法:

LabelledStatement :

Identifier : Statement

语义:

一个 Statement 可以由一个标签作为前缀。标签语句仅与标签化的 break 和 continue 语句一起使用。ECMAScript 没有 goto 语句。

如果一个 ECMAScript 程序包含有相同 Identifier 作为标签的 LabelledStatement 闭合的 LabelledStatement, 那么认为它是语法错误的。这不适用于直接或间接嵌套在标签语句里面的 FunctionDeclaration 的 body 里出现标签的情况。

产生式 `Identifier : Statement` 的解释执行方式是，先添加 `Identifier` 到 `Statement` 的标签组，再解释执行 `Statement`。如果 `LabelledStatement` 自身有一个非空标签组，这些标签还是会添加到解释执行前的 `Statement` 的标签组里。如果 `Statement` 的解释执行结果是 `(break, V, L)`，这里的 `L` 等于 `Identifier`，则产生式的结果是 `(normal, V, empty)`。

在解释执行 `LabelledStatement` 之前，认为包含的 `Statement` 拥有一个空标签组，除非它是 `IterationStatement` 或 `SwitchStatement`，这种情况下认为它拥有一个包含单个元素 `empty` 的标签组。

throw 语句

语法：

`ThrowStatement :`

`throw [no LineTerminator here] Expression ;`

语义：

产生式 `ThrowStatement : throw [no LineTerminator here] Expression ;` 按照下面的过程执行：

1. 令 `exprRef` 为解释执行 `Expression` 的结果。
2. 返回 `(throw, GetValue(exprRef), empty)`。

try 语句

语法：

`TryStatement :`

`try Block Catch`

`try Block Finally`

`try Block Catch FinallyCatch :`

`catch (Identifier) BlockFinally :`

`finally Block`

`try` 语句包裹一个可以出现特殊状况，如果运行时错误或 `throw` 语句的代码块。`catch` 子句提供了异常处理代码。如果 `catch` 子句捕获到一个异常，这个异常会绑定到它的 `Identifier` 上。

语义：

产生式 TryStatement : try Block Catch 按照下面的过程执行：

1. 令 B 为解释执行 Block 的结果。
2. 如果 B.type 不是 throw, 返回 B.
3. 返回一参数 B 解释执行 Catch 的结果。

产生式 TryStatement : try Block Finally 按照下面的过程执行：

1. 令 B 为解释执行 Block 的结果。
2. 令 F 为解释执行 Finally 的结果。
3. 如果 F.type 是 normal, 返回 B.
4. 返回 F.

产生式 TryStatement : try Block Catch Finally 按照下面的过程执行：

1. 令 B 为解释执行 Block 的结果。
2. 如果 B.type 是 throw, 则
 1. 令 C 为以参数 B 解释执行 Catch 的结果。
3. 否则, B.type 不是 throw,
 1. 令 C 为 B.
4. 令 F 为解释执行 Finally 的结果。
5. 如果 F.type 是 normal, 返回 C.
6. 返回 F.

产生式 Catch : catch (Identifier) Block 按照下面的过程执行：

1. 令 C 为传给这个产生式的参数。
2. 令 oldEnv 为运行中执行环境的 LexicalEnvironment.
3. 令 catchEnv 为以 oldEnv 为参数调用 NewDeclarativeEnvironment 的结果
4. 以 Identifier 字符串值为参数调用 catchEnv 的 CreateMutableBinding 具体方法。
5. 以 Identifier, C, false 为参数调用 catchEnv 的 SetMutableBinding 具体方法。注：这种情况下最后一个参数无关紧要。
6. 设定运行中执行环境的 LexicalEnvironment 为 catchEnv.
7. 令 B 为解释执行 Block 的结果。
8. 设定运行中执行环境的 LexicalEnvironment 为 oldEnv.
9. 返回 B.

不管控制是怎样退出 Block 的, LexicalEnvironment 总是会恢复到其之前的状态。

产生式 Finally : finally Block 按照下面的过程执行：

1. 返回解释执行 Block 的结果。

严格模式的限制

如果一个有 Catch 的 TryStatement 出现在 严格模式代码 里，并且 Catch 产生式的 Identifier 是 "eval" 或 "arguments"，那么这是个 SyntaxError

debugger 语句

语法：

DebuggerStatement :

debugger ;

语义：

解释执行 DebuggerStatement 产生式可允许让一个实现在调试器下运行时设置断点。如果调试器不存在或是非激活状态，这个语句没有可观测效果。

产生式 DebuggerStatement : debugger ; 按照下面的过程执行：

1. 如果一个实现定义了可用的调试工具并且是开启的，则
 1. 执行实现定义的调试动作。
 2. 令 result 为实现定义的 Completion 值。
 2. 否则
 1. 令 result 为 (normal, empty, empty).
 3. 返回 result.

函数定义

语法

FunctionDeclaration :

```
function Identifier ( FormalParameterListopt )  
{ FunctionBody }FunctionExpression :
```

```
function Identifieropt ( FormalParameterListopt )  
{ FunctionBody }FormalParameterList :
```

Identifier

FormalParameterList , IdentifierFunctionBody :

SourceElementsopt

语义

产生式 FunctionDeclaration : function Identifier (FormalParameterList_{opt})
{ FunctionBody } 依照定义绑定初始化 (10.5) 如下初始化:

1. 依照 13.2, 指定 FormalParameterList_{opt} 为参数, 指定 FunctionBody 为 body, 创建一个新函数对象, 返回结果。运行中的执行环境的 VariableEnvironment 传递为 Scope。如果 FunctionDeclaration 包含在严格模式代码 里或 FunctionBody 是 严格模式代码 , 那么传递 true 为 Strict 标志。

产生式 FunctionExpression : function (FormalParameterList_{opt})
{ FunctionBody } 的解释执行如下:

1. 依照 13.2, 指定 FormalParameterList_{opt} 为参数, 指定 FunctionBody 为 body, 创建一个新函数对象, 返回结果。运行中的执行环境的 LexicalEnvironment 传递为 Scope。如果 FunctionExpression 包含在 严格模式代码 里或 FunctionBody 是 严格模式代码 , 那么传递 true 为 Strict 标志。

产生式 FunctionExpression : function Identifier_{opt} (FormalParameterList_{opt})
{ FunctionBody } 的解释执行如下:

1. 令 funcEnv 为以运行中执行环境的 Lexical Environment 为参数调用 NewDeclarativeEnvironment 的结果。
2. 令 envRec 为 funcEnv 的环境记录项。
3. 以 Identifier 的字符串值为参数调用 envRec 的具体方法 CreateImmutableBinding(N)。
4. 令 closure 为依照 13.2, 指定 FormalParameterList_{opt} 为参数, 指定 FunctionBody 为 body, 创建一个新函数对象的结果。传递 funcEnv 为 Scope。如果 FunctionExpression 包含在严格模式代码 里或 FunctionBody 是 严格模式代码 , 那么传递 true 为 Strict 标志。
5. 以 Identifier 的字符串值和 closure 为参数调用 envRec 的具体方法 InitializeImmutableBinding(N,V)。
6. 返回 closure。

可以从 FunctionExpression 的 FunctionBody 里面引用 FunctionExpression 的 Identifier, 以允许函数递归调用自身。然而不像 FunctionDeclaration, FunctionExpression 的 Identifier 不能被范围封闭的 FunctionExpression 引用, 也不会影响它。

产生式 `FunctionBody : SourceElementsopt` 的解释执行如下：

1. 如果这个 `FunctionBody` 所在 `FunctionDeclaration` 或 `FunctionExpression` 包含在严格模式代码内，或其 `SourceElements` 的指令序言 (14.1) 包含一个 `use strict` 指令，或满足 10.1 的任何条件，那么其代码是严格模式代码。如果 `FunctionBody` 的代码是严格模式代码，`SourceElements` 的解释执行为以下的严格模式代码步骤。否则，`SourceElements` 的解释执行为以下的非严格模式步骤。
2. 如果 `SourceElements` 是当前的，则返回 `SourceElements` 的解释执行结果。
3. 否则返回 (normal, undefined, empty)。

严格的模式的限制

如果严格模式 `FunctionDeclaration` 或 `FunctionExpression` 的 `FormalParameterList` 里出现多个相同 `Identifier` 值，那么这是个 `SyntaxError`。

如果严格模式 `FunctionDeclaration` 或 `FunctionExpression` 的 `FormalParameterList` 里出现标识符 "eval" 或标识符 "arguments"，那么这是个 `SyntaxError`。

如果严格模式 `FunctionDeclaration` 或 `FunctionExpression` 的 `Identifier` 是标识符 "eval" 或标识符 "arguments"，那么这是个 `SyntaxError`。

创建函数对象

指定 `FormalParameterList` 为可选参数列表，指定 `FunctionBody` 为函数体，指定 `Scope` 为 词法环境，`Strict` 为布尔标记，按照如下步骤构建函数对象：

1. 创建一个新的 ECMAScript 原生对象，令 `F` 为此对象。
2. 依照 8.12 描述设定 `F` 的除 `[[Get]]` 以外的所有内部方法。
3. 设定 `F` 的 `[[Class]]` 内部属性为 "Function"。
4. 设定 `F` 的 `[[Prototype]]` 内部属性为 15.3.3.1 指定的标准内置 `Function` 对象的 `prototype` 属性。
5. 依照 15.3.5.4 描述，设定 `F` 的 `[[Get]]` 内部属性。
6. 依照 13.2.1 描述，设定 `F` 的 `[[Call]]` 内部属性。
7. 依照 13.2.2 描述，设定 `F` 的 `[[Construct]]` 内部属性。
8. 依照 15.3.5.3 描述，设定 `F` 的 `[[HasInstance]]` 内部属性。
9. 设定 `F` 的 `[[Scope]]` 内部属性为 `Scope` 的值。
10. 令 `names` 为一个列表容器，其中元素是以从左到右的文本顺序对应 `FormalParameterList` 的标识符的字符串。
11. 设定 `F` 的 `[[FormalParameters]]` 内部属性为 `names`。
12. 设定 `F` 的 `[[Code]]` 内部属性为 `FunctionBody`。

13. 设定 F 的 `[[Extensible]]` 内部属性为 `true`。
14. 令 `len` 为 `FormalParameterList` 指定的形式参数的个数。如果没有指定参数, 则令 `len` 为 0。
15. 以参数 "length", 属性描述符 `{[[Value]]: len, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}`, `false` 调用 F 的 `[[DefineOwnProperty]]` 内部方法。
16. 令 `proto` 为仿佛使用 `new Object()` 表达式创建新对象的结果, 其中 `Object` 是标准内置构造器名。
17. 以参数 "constructor", 属性描述符 `{[[Value]]: F, { [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}`, `false` 调用 `proto` 的 `[[DefineOwnProperty]]` 内部方法。
18. 以参数 "prototype", 属性描述符 `{[[Value]]: proto, { [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false}`, `false` 调用 F 的 `[[DefineOwnProperty]]` 内部方法。
19. 如果 `Strict` 是 `true`, 则
 1. 令 `thrower` 为 `[[ThrowTypeError]]` 函数对象 (13.2.3)。
 2. 以参数 "caller", 属性描述符 `{[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}`, `false` 调用 F 的 `[[DefineOwnProperty]]` 内部方法。
 3. 以参数 "caller", 属性描述符 `{[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}`, `false` 调用 F 的 `[[DefineOwnProperty]]` 内部方法。
20. 返回 F。

每个函数都会自动创建一个 `prototype` 属性, 以满足函数会被当作构造器的可能性。

`[[call]]`

当用一个 `this` 值, 一个参数列表调用函数对象 F 的 `[[Call]]` 内部方法, 采用以下步骤:

1. 用 F 的 `[[FormalParameters]]` 内部属性值, 参数列表 `args`, 10.4.3 描述的 `this` 值来建立 函数代码 的一个新执行环境, 令 `funcCtx` 为其结果。
2. 令 `result` 为 `FunctionBody` (也就是 F 的 `[[Code]]` 内部属性) 解释执行的结果。如果 F 没有 `[[Code]]` 内部属性或其值是空的 `FunctionBody`, 则 `result` 是 `(normal, undefined, empty)`。
3. 退出 `funcCtx` 执行环境, 恢复到之前的执行环境。
4. 如果 `result.type` 是 `throw` 则抛出 `result.value`。
5. 如果 `result.type` 是 `return` 则返回 `result.value`。
6. 否则 `result.type` 必定是 `normal`。返回 `undefined`。

`[[Construct]]`

当以一个可能的空的参数列表调用函数对象 F 的 `[[Construct]]` 内部方法, 采用以下步骤:

1. 令 `obj` 为新创建的 ECMAScript 原生对象。
2. 依照 8.12 设定 `obj` 的所有内部方法。
3. 设定 `obj` 的 `[[Class]]` 内部方法为 "Object"。
4. 设定 `obj` 的 `[[Extensible]]` 内部方法为 `true`。
5. 令 `proto` 为以参数 "prototype" 调用 F 的 `[[Get]]` 内部属性的值。
6. 如果 `Type(proto)` 是 `Object`, 设定 `obj` 的 `[[Prototype]]` 内部属性为 `proto`。
7. 如果 `Type(proto)` 不是 `Object`, 设定 `obj` 的 `[[Prototype]]` 内部属性为 15.2.4 描述的标准内置的 `Object` 的 `prototype` 对象。
8. 以 `obj` 为 `this` 值, 调用 `[[Construct]]` 的参数列表为 `args`, 调用 F 的 `[[Call]]` 内部属性, 令 `result` 为调用结果。
9. 如果 `Type(result)` 是 `Object`, 则返回 `result`。
10. 返回 `obj`

`[[ThrowTypeError]]` 函数对象

`[[ThrowTypeError]]` 对象是个唯一的函数对象, 如下只定义一次:

1. 创建一个新 ECMAScript 原生对象, 令 F 为此对象。
2. 依照 8.12 设定 F 的所有内部属性。
3. 设定 F 的 `[[Class]]` 内部属性为 "Function"。
4. 设定 F 的 `[[Prototype]]` 内部属性为 15.3.3.1 指定的标准内置 `Function` 的 `prototype` 对象。
5. 依照 13.2.1 描述设定 F 的 `[[Call]]` 内部属性。
6. 设定 F 的 `[[Scope]]` 内部属性为 全局环境 。
7. 设定 F 的 `[[FormalParameters]]` 内部属性为一个空 列表 。
8. 设定 F 的 `[[Code]]` 内部属性为一个 `FunctionBody`, 它无条件抛出一个 `TypeError` 异常, 不做其他事情。
9. 以参数 "length", 属性描述符 `{[[Value]]: 0, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}`, `false` 调用 F 的 `[[DefineOwnProperty]]` 内部方法。
10. 设定 F 的 `[[Extensible]]` 内部属性为 `false`。
11. 令 `[[ThrowTypeError]]` 为 F。

程序

语法:

Program :

SourceElements_{opt}SourceElements :

SourceElement

SourceElements SourceElementSourceElement :

Statement

FunctionDeclaration

语义:

产生式 Program : SourceElements_{opt} 依照下面的步骤来解释执行 :

1. 若 SourceElements 的指令序言 (参考 14.1 章) 中 , 包含严格模式指令 , 或者满足 10.1.1 章节所描述的任何一个条件 . 则 Program 的代码 . 就是一段严格模式代码 . 并对应性的 , 以严格模式或非严格模式 , 依照下面列出的步骤来解释执行代码 .
2. 若没有 SourceElements 部分 , 则返回 (normal, empty, empty).
3. 令 progCxt 为一个新的 , 如 10.4.1 章节所描述的 , 应用于全局代码的执行环境 .
4. 令 result 为解释执行 SourceElements 的结果 .
5. 退出 progCxt 这个执行环境 .
6. 返回 result.

本规范不会规定 , 具体如何解释执行一个 Program 以及如何处理其结果 . 其具体行为由 ECMAScript 实现 , 自行定义

产生式 SourceElements : SourceElements SourceElement 依照下面的步骤来解释执行 :

1. 令 headResult 为解释执行 SourceElements 的结果 .
2. 若 headResult 是非常规性完结的 , 返回 headResult.
3. 令 tailResult 为解释执行 SourceElement 的结果 .
4. 若 tailResult.value 为 empty, 令 V = headResult.value, 其他情况 , 另 V = tailResult.value.
5. 返回 (tailResult.type, V, tailResult.target).

产生式 : SourceElement : Statement 依照下面的步骤来解释执行 :

1. 返回解释执行 Statement 的结果 .

产生式 : SourceElement : FunctionDeclaration 依照下面的步骤来解释执行 :

1. 返回 (normal, empty, empty)

指令序言和严格模式指令 .

一个指令序言，是那些从 **Program** 或 **FunctionBody** 的首个 **SourceElement** 开始，到那些完全由一个字符串字面量后面跟一个分号，所构成的最长的。那一组 **ExpressionStatement** 序列中的每一个。字符串字面量后面的分号，可以显式的插入，或者借助分号自动插入机制来插入。一个指令序言，也可以是一个空的序列。

严格模式指令是一个 **"use strict"** 或 **'use strict'** 的字符串字面量。一个严格模式指令中，不应该包含 **EscapeSequence** 或 **LineContinuation**。

一个指令序言，可以不仅仅包含一个严格模式指令。然而，当这种情况出现的时候，**ECMAScript** 实现，可以发出一个相关警告。

指令序言包含的 **ExpressionStatement** 产生式们，会在解释执行包含他们的 **SourceElements** 产生式期间，被正常的解析执行。**ECMAScript** 实现，可以在一个指令序言中定义其他非严格模式指令。当一个指令序言中的某个 **ExpressionStatement** 并不是一个严格模式指令，也不是一个被 **ECMAScript** 实现所定义的指令。且存在某种通知机制的话。就要借助该机制，发出一个警告。

标准 ECMAScript 内置对象

ECMAScript 代码运行时会有一些可用的内置对象。一是作为执行程序词法环境的一部分的全局对象。其他的可通过全局对象的初始属性访问。

除非另外指明，如果内置对象拥有 **[[Call]]** 内部属性，那么它的 **[[Class]]** 内部属性是 **"Function"**，如果没有 **[[Call]]** 内部属性，那么它的 **[[Class]]** 内部属性是 **"Object"**。除非另外指明，内置对象的 **[[Extensible]]** 内部属性的初始值是 **true**。

许多内置对象是函数：它们可以通过参数调用。其中有些还作为构造器：这些函数可被 **new** 运算符调用。对于每个内置函数，本规范描述了这些函数的必须参数和 **Function** 对象的属性。对于每个内置构造器，本规范还描述了这些构造器的 **prototype** 对象的属性，还描述了用 **new** 表达式调用这个构造器后返回的具体实例对象的属性。

除非另外指明了某一特定函数的描述，如果在调用本章中描述的函数或构造器时传入的参数少于必须的参数个数，那么这些函数或构造器将表现为仿佛传入了足够的参数，而那些缺少的参数会设定为 **undefined** 值。

除非另外指明了某一特定函数的描述,如果在调用本章中描述的函数或构造器时传入了比函数指定允许的更多的参数时,额外的参数会被函数忽略。然而,一个实现可以为这样的参数列表定义依赖于实现的特别行为,只要这种行为在单纯添加额外参数时不抛出 `TypeError` 异常。

实现为了给内置函数集合增添一些额外功能而添加新函数是被鼓励的,而不是为现有函数增加新参数。

每个内置函数和每个内置构造器都有 `Function` 原型对象, `Function.prototype` (15.3.4) 表达式的初始值作为其 `[[Prototype]]` 内部属性的值。

除非另外指明,每个内置的原型对象都有 `Object` 原型对象, `Object.prototype` (15.2.4) 表达式的初始值作为其 `[[Prototype]]` 内部属性的值,除了 `Object` 的原型对象自身。

除非另外指明了特定函数的描述,否则本章描述的内置函数中不存在不是构造器而要实现 `[[Construct]]` 内部方法的内置函数。除非另外指明了特定函数的描述,否则本章描述的内置函数都没有 `prototype` 属性。

本章通常描述构造器的“作为函数调用”和“用 `new` 表达式调用”有不同行为。“作为函数调用”的行为对应于调用构造器的 `[[Call]]` 内部方法,“用 `new` 表达式调用”的行为对应于调用构造器的 `[[Construct]]` 内部方法。

本章描述的每个内置 `Function` 对象 -- 不管是构造器还是普通函数,或二者都是 -- 拥有一个 `length` 属性,其值是个整数。除非另外指明,此值等于显示在函数描述的子章节标题的形式参数的个数,包括可选参数。

例如描述 `String` 的 `prototype` 对象的 `slice` 属性初始值的函数对象的子章节标题是“`String.prototype.slice (start, end)`”,这说明有两个形参 `start` 和 `end`,所以这个函数对象的 `length` 属性值是 2。

任何情况下,本章描述的内置函数对象的 `length` 属性拥有特性 `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`。除非另外指明,本章描述的所有其他属性拥有特性 `{ [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true }`。

全局对象

唯一的全局对象建立在控制进入任何执行环境之前。

除非另外指明,全局对象的标准内置属性拥有特性 `{ [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true }`。

全局对象没有 `[[Construct]]` 内部属性；全局对象不可能当做构造器用 `new` 运算符调用。

全局对象没有 `[[Call]]` 内部属性，全局对象不可能当做函数来调用。

全局对象的 `[[Prototype]]` 和 `[[Class]]` 内部属性值是依赖于实现的。

除了本规范定义的属性之外，全局对象还可以拥有额外的宿主定义的属性。全局对象可包含一个值是全局对象自身的属性；例如，在 HTML 文档对象模型中全局对象的 `window` 属性是全局对象自身。

全局对象的值属性

NaN

NaN 的值是 NaN（见 8.5）。这个属性拥有特性 { `[[Writable]]`: false, `[[Enumerable]]`: false, `[[Configurable]]`: false }。

Infinity

Infinity 的值是 $+\infty$ （见 8.5）。这个属性拥有特性 { `[[Writable]]`: false, `[[Enumerable]]`: false, `[[Configurable]]`: false }。

undefined

undefined 的值是 undefined（见 8.1）。这个属性拥有特性 { `[[Writable]]`: false, `[[Enumerable]]`: false, `[[Configurable]]`: false }。

全局对象的函数属性

eval (x)

当用一个参数 `x` 调用 `eval` 函数，采用如下步骤：

1. 如果 `Type(x)` 不是 `String`，返回 `x`。
2. 令 `prog` 为 ECMAScript 代码，它是将 `x` 作为一个程序解析的结果。如果解析失败，抛出一个 `SyntaxError` 异常（见 16 章）。
3. 令 `evalCtx` 为给 `eval` 代码 `prog` 建立的新执行环境 (10.4.2)。
4. 令 `result` 为解释执行程序 `prog` 的结果。
5. 退出执行环境 `evalCtx`，恢复到之前的执行环境。

6. 如果 `result.type` 是 `normal` 并且其完结类型值是 `V`，则返回 `V` 值。
7. 如果 `result.type` 是 `normal` 并且其完结类型值是 `empty`，则返回 `undefined` 值。
8. 否则，`result.type` 必定是 `throw`。将 `result.value` 作为异常抛出。

直接调用 Eval

一个 `eval` 函数的直接调用是表示为符合以下两个条件的 `CallExpression`：

解释执行 `CallExpression` 中的 `MemberExpression` 的结果是个引用，这个引用拥有一个环境记录项作为其基值，并且这个引用的名称是 `"eval"`。

以这个引用作为参数调用 `GetValue` 抽象操作的结果是 15.1.2.1 定义的标准内置函数。

`parseInt (string , radix)`

`parseInt` 函数根据指定的参数 `radix`，和 `string` 参数的内容解释结果来决定，产生一个整数值。`string` 开头的空白会被忽略。如果 `radix` 是 `undefined` 或 `0`，假定它是 `10`，除非数字是以字符对 `0x` 或 `0X` 开头的，这时假定 `radix` 是 `16`。如果 `radix` 是 `16`，数字开头的字符对 `0x` 或 `0X` 是可选的。

当调用 `parseInt` 函数时，采用以下步骤：

1. 令 `inputString` 为 `Tostring(string)`。
2. 令 `S` 为一个新创建的子字符串，它由 `inputString` 的第一个非 `StrWhiteSpaceChar` 字符和它后面跟着的所有字符组成。（换句话说，删掉前面的空白。）如果 `inputString` 不包含任何这样的字符，则令 `S` 为空字符串。
3. 令 `sign` 为 `1`。
4. 如果 `S` 不是空并且 `S` 的第一个字符是减号 `-`，则令 `sign` 为 `-1`。
5. 如果 `S` 不是空并且 `S` 的第一个字符加号 `+` 或减号 `-`，则删除 `S` 的第一个字符。
6. 令 `R = ToInt32(radix)`。
7. 令 `stripPrefix` 为 `true`。
8. 如果 `R ≠ 0`，则
 1. 如果 `R < 2` 或 `R > 36`，则返回 `NaN`。
 2. 如果 `R ≠ 16`，令 `stripPrefix` 为 `false`。
9. 否则，`R = 0`
 1. 令 `R = 10`。
10. 如果 `stripPrefix` 是 `true`，则
 1. 如果 `S` 长度大于 `2` 并且 `S` 的头两个字符是 `"0x"` 或 `"0X"`，则删除 `S` 的头两个字符并且令 `R = 16`。

11. 如果 `S` 包含任何不是 `radix-R` 进制的字符，则令 `Z` 为 `S` 的这样的字符之前的所有字符组成的子字符串；否则令 `Z` 为 `S`。
12. 如果 `Z` 是空，返回 `NaN`。
13. 令 `mathInt` 为 `Z` 的 `radix-R` 进制表示的数学值，用字母 `A-Z` 和 `a-z` 来表示 10 到 35 之间的值。（但是，如果 `R` 是 10 并且 `Z` 包含多余 20 位的值，可以替换 20 位后的每个数字为 0，这是实现可选的功能；如果 `R` 不是 2, 4, 8, 10, 16, 32，则 `mathInt` 可以是 `Z` 的 `radix-R` 进制表示的依赖于实现的近似值。）
14. 令 `number` 为 `mathInt` 的数值。
15. 返回 `sign × number`。

`parseInt` 可以只把 `string` 的开头部分解释为整数值；它会忽略所有不能解释为整数记法的一部分的字符，并且没有指示会给出任何这些忽略的字符。

`parseFloat (string)`

`parseFloat` 函数根据 `string` 参数的内容解释为十进制字面量的结果来决定，产生一个数值。

当调用 `parseFloat` 函数，采用以下步骤：

1. 令 `inputString` 为 `ToString(string)`。
2. 令 `trimmedString` 为一个新创建的子字符串，它由 `inputString` 的非 `StrWhiteSpaceChar` 字符的最左边字符和它右边跟着的所有字符组成。（换句话说，删掉前面的空白。）如果 `inputString` 不包含任何这样的字符，则令 `trimmedString` 为空字符串。
3. 如果 `trimmedString` 或 `trimmedString` 的任何前缀都不满足 `StrDecimalLiteral`（见 9.3.1）的语法，返回 `NaN`。
4. 令 `numberString` 为满足 `StrDecimalLiteral` 语法的 `trimmedString` 的最长前缀，可能是 `numberString` 自身。
5. 返回 `numberString` 的 MV 的数值。

`parseFloat` 可以只把 `string` 的开头部分解释为数值；它会忽略所有不能解释为数值字面量记法的一部分的字符，并且没有指示会给出任何这些忽略的字符。

`isNaN (number)`

如果指定参数为 `NaN`，则返回 `true`，否则返回 `false`。

1. 如果 `ToNumber(number)` 是 `NaN`，返回 `true`。
2. 否则，返回 `false`。

一个用 ECMAScript 代码来测试值 X 是否是 NaN 的方式是用 $X !== X$ 表达式。当且仅当 X 是 NaN 时结果才是 true。

isFinite (number)

如果指定参数为 NaN 或 $+\infty$ 或 $-\infty$ ，则返回 false，否则返回 true。

1. 如果 ToNumber(number) 是 NaN 或 $+\infty$ 或 $-\infty$ ，返回 false.
2. 否则，返回 true.

处理 URI 的函数属性

统一资源标识符，或叫做 URI，是用来标识互联网上的资源（例如，网页或文件）和怎样访问这些资源的传输协议（例如，HTTP 或 FTP）的字符串。除了 15.1.3.1, 15.1.3.2, 15.1.3.3, 15.1.3.4 说明的用来编码和解码 URI 的函数之外 ECMAScript 语言自身不提供任何使用 URL 的支持。

许多 ECMAScript 实现提供额外的函数，方法来操作网页；这些函数超出了本标准的范围。

一个 URI 是由组件分隔符分割的组件序列组成。其一般形式是：

Scheme : First / Second ; Third ? Fourth

其中斜体的名字代表组件；“:”，“/”，“;”，“?”是当作分隔符的保留字符。encodeURIComponent 和 decodeURI 函数操作的是完整的 URI；这俩函数假定 URI 中的任何保留字符都有特殊意义，所有不会编码它们。encodeURIComponent 和 decodeURIComponent 函数操作的是组成 URI 的个别组件；这俩函数假定任何保留字符都代表普通文本，所以必须编码它们，所以它们出现在组成一个完整 URI 的组件里面时不会解释成保留字符了。

以下词法文法指定了编码后 URI 的形式。

语法

uri :::

uriCharactersopt**uriCharacters** :::

uriCharacter **uriCharacters**opt**uriCharacter** :::

uriReserved

uriUnescaped


```

uriEscaped uriReserved ::: one of

; / ? : @ & = + $ , uriUnescaped :::

uriAlpha

DecimalDigit

uriMarkuriEscaped :::

% HexDigit HexDigit uriAlpha ::: one of

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z uriMark :::
one of

- _ . ! ~ * ' ( )

```

以上语法是基于 [RFC 2396](#) 的，并且较新的 [RFC 3986](#) 引入的更改没有反应在这里。

当 URI 里包含一个没在上面列出的字符或有时不想让给定的保留字符有特殊意义，那么必须编码这个字符。字符被转换成 UTF-8 编码，首先从 UTF-16 转换成相应的代码点值的替代对。（注：在 [0,127] 范围的代码单元在单字节中具有相同返回值。）然后返回的字节序列转换为一个字符串，每个字节用一个“%xx”形式的转移序列表示。

描述编码和转义过程的抽象操作 `Encode` 需要两个字符串参数 `string` 和 `unescapedSet`。

1. 令 `strLen` 为 `string` 的字符个数。
2. 令 `R` 为空字符串。
3. 令 `k` 为 0.
4. 重复
 1. 如果 `k` 等于 `strLen`, 返回 `R`.
 2. 令 `C` 为 `string` 中位置为 `k` 的字符。
 3. 如果 `C` 在 `unescapedSet` 里，则
 - i. 令 `S` 为一个只包含字符 `C` 的字符串。
 - ii. 令 `R` 为之前 `R` 的值和 `S` 连接得到的一个新字符串值。
 4. 否则，`C` 不在 `unescapedSet` 里
 - i. 如果 `C` 的代码单元值不小于 `0xDC00` 并且不大于 `0xDFFF`，则抛出一个 `URIError` 异常。
 - ii. 如果 `C` 的代码单元值小于 `0xD800` 或大于 `0xDBFF`，则
 1. 令 `V` 为 `C` 的代码单元值。
- iii. 否则，

1. k 递增 1.
2. 如果 k 等于 strLen, 抛出一个 URIError 异常 .
3. 令 kChar 为 string 的 k 位置的字符的代码单元值 .
4. 如果 kChar 小于 0xDC00 或大于 0xDFFF, 则抛出一个 URIError 异常 .
5. 令 V 为 $((C \text{ 的代码单元值}) - 0xD800) * 0x400 + (kChar - 0xDC00) + 0x10000$.
- iv. 令 Octets 为 V 执行 UTF-8 转换的结果字节排列 , 令 L 为这个字节排列的长度 .
- v. 令 j 为 0.
- vi. 只要 $j < L$, 就重复
 1. 令 jOctet 为 Octets 的 j 位置的值 .
 2. 令 S 为一个包含三个字符"%XY"的字符串, 这里 XY 是编码 jOctet 值的两个大写 16 进制数字 .
 3. 令 R 为之前 R 的值和 S 连接得到的一个新字符串值 .
 4. j 递增 1.
 5. k 递增 1.

描述反转义和解码过程的抽象操作 Decode 需要两个字符串参数 string 和 reservedSet。

1. 令 strLen 为 string 的字符个数 .
2. 令 R 为空字符串 .
3. 令 k 为 0.
4. 重复
 1. 如果 k 等于 strLen, 返回 R.
 2. 令 C 为 string 的 k 位置的字符 .
 3. 如果 C 不是 '%', 则
 - i. 令 S 为只包含字符 C 的字符串 .
 4. 否则 , C 是 '%'
 - i. 令 start 为 k.
 - ii. 如果 $k + 2$ 大于或等于 strLen, 抛出一个 URIError 异常 .
 - iii. 如果 string 的 (k+1) 和 (k+2) 位置的字符没有表示为 16 进制数字, 则抛出一个 URIError 异常 .
 - iv. 令 B 为 (k+1) 和 (k+2) 位置的两个 16 进制数字表示的 8 位值 .
 - v. k 递增 2.
 - vi. 如果 B 的最高有效位是 0, 则
 1. 令 C 为代码单元值是 B 的字符 .
 2. 如果 C 不在 reservedSet 里 , 则
 1. 令 S 为只包含字符 C 的字符串 .
 3. 否则 , C 在 reservedSet 里
 1. 令 S 为 string 的从位置 start 到位置 k 的子字符串 .
 - vii. 否则 , B 的最高有效位是 1
 1. 令 n 为满足 $(B \ll n) \& 0x80$ 等于 0 的最小非负数 .

2. 如果 n 等于 1 或 n 大于 4, 抛出一个 `URIError` 异常 .
3. 令 `Octets` 为一个长度为 n 的 8 位整数排列 .
4. 将 `B` 放到 `Octets` 的 0 位置 .
5. 如果 $k + (3 * (n - 1))$ 大于或等于 `strLen`, 抛出一个 `URIError` 异常 .
6. 令 j 为 1.
7. 重复 , 直到 $j < n$
 1. k 递增 1.
 2. 如果 `string` 的 k 位置的字符不是 '%', 抛出一个 `URIError` 异常 .
 3. 如果 `string` 的 $(k+1)$ 和 $(k+2)$ 位置的字符没有表示为 16 进制数字 , 抛出一个 `URIError` 异常 .
 4. 令 `B` 为 `string` 的 $(k+1)$ 和 $(k+2)$ 位置的两个 16 进制数字表示的 8 位值 .
 5. 如果 `B` 的两个最高有效位不是 10, 抛出一个 `URIError` 异常 .
 6. k 递增 2.
 7. 将 `B` 放到 `Octets` 的 j 位置 .
 8. j 递增 1.
8. 令 `V` 为给 `Octets` 执行 UTF-8 转换得到的值, 这是从一个字节排列到一个 32 位值的过程。 如果 `Octets` 不包含有效的 UTF-8 编码的 Unicode 代码点, 则抛出一个 `URIError` 异常 .
9. 如果 `V` 小于 `0x10000`, 则
 1. 令 `C` 为代码单元值是 `V` 的字符 .
 2. 如果 `C` 不在 `reservedSet` 里 , 则
 - i. 令 `S` 为只包含字符 `C` 的字符串 .
 3. 否则 , `C` 在 `reservedSet` 里
 - i. 令 `S` 为 `string` 的从位置 `start` 到位置 k 的子字符串 .
10. 否则 , $V \geq 0x10000$
 1. 令 `L` 为 $((V - 0x10000) \& 0x3FF) + 0xDC00$.
 2. 令 `H` 为 $((V - 0x10000) \gg 10) \& 0x3FF + 0xD800$.
 3. 令 `S` 为代码单元值是 `H` 和 `L` 的两个字符组成的字符串 .
 5. 令 `R` 为之前的 `R` 和 `S` 连接成的新字符串 .
 6. k 递增 1.

统一资源标识符的语法由 [RFC 2396](#) 给出, 这里并没有反应更新的替换了 [RFC 2396](#) 的 [RFC 3986](#). [RFC 3629](#) 给出了实现 UTF-8 的正式描述。

在 UTF-8 中, 用 1 到 6 个字节的序列来编码字符。只有“序列”中高阶位设置为 0 的字节, 其余的 7 位才用于编码字符值。在一个 n 个字节的序列中, $n > 1$, 初始字节有 n 个设置为 1 的高阶位, 其后的位设置为 0。这个字节的其他位包含是用来编码字符的比特。后面跟着的其字节都包含设定为 1 的高阶位, 并且都跟着设定为 0 的位, 剩下的 6 位都用作编码字符。表 21 指定了 ECMAScript 字符可能的 UTF-8 编码。

UTF-8 编码

字符编码值	表示	第 1 十六进	第 2 六进制	第 3 十六	第 4 十六
-------	----	---------	---------	--------	--------

		制位	位	进制位	进制位
0x0000 - 0x007F	00000000 0zzzzzzz	0zzzzzzz			
0x0080 - 0x07FF	00000yyy yyzzzzzz	110yyyyy	10zzzzzz		
0x0800 - 0xD7FF	xxxxyyyy yyzzzzzz	1110xxxx	10yyyyyy	10zzzzzz	
0xD800 - 0xDBFF 后跟 0xDC00 - 0xDFFF	110110vv vwwwwxx 后跟 110111yy yyzzzzzz	11110uuu	10uuwww	10xyyyyy	10zzzzzz
0xD800 - 0xDBFF 后无 0xDC00 - 0xDFFF	导致 URIError				
0xDC00 - 0xDFFF	导致 URIError				
0xE000 - 0xFFFF	xxxxyyyy yyzzzzzz	1110xxxx	10yyyyyy	10zzzzzz	

在这里

$uuuuu = vvvv + 1$

来访问附加的作为代理项的 0x10000，在 Unicode 标准 3.7 章节。

0xD800-0xDFFF 范围的代码单元值用来编码代理对；如上将 UTF-16 代理对转换组合成一个 UTF-32 表示，并编码 UTF-8 值的 21 位结果。解码重建代理对。

RFC 3629 禁止对无效 UTF-8 字节序列的解码。例如，无效序列 C0 80 不能解码成字符 U+0000。当 Decode 算法的实现遇到这样的无效序列必须抛出一个 URIError 异常。

decodeURI (encodedURI)

decodeURI 函数计算出一个新版 URI，将 URI 中可能是 encodeURIComponent 函数引入的每个转义序列和 UTF-8 编码组替换为代表它们的字符。不是 encodeURIComponent 导入的转义序列不会被替换。

当以一个参数 encodedURI 调用 decodeURI 函数，采用如下步骤：

1. 令 uriString 为 ToString(encodedURI)。

2. 令 `reservedURISet` 为一个字符串, 包含 `uriReserved` 的每个有效字符加上 `"#"` 的实例。
3. 返回调用 `Decode(uriString, reservedURISet)` 的结果。

`"#"` 字符不会从转义序列中解码, 即使它不是 `URI` 保留字符。

decodeURIComponent (encodedURIComponent)

`decodeURIComponent` 函数计算出一个新版 `URI`, 将 `URI` 中可能是 `encodeURIComponent` 函数引入的每个转义序列和 `UTF-8` 编码组替换为代表它们的字符。

当以一个参数 `encodedURIComponent` 调用 `decodeURIComponent` 函数, 采用如下步骤:

1. 令 `componentString` 为 `ToString(encodedURIComponent)`.
2. 令 `reservedURIComponentSet` 为一个空字符串。
3. 返回调用 `Decode(componentString, reservedURIComponentSet)` 的结果。

encodeURI (uri)

`encodeURI` 函数计算出一个新版 `URI`, 将 `URI` 中某些字符的每个实例替换为代表这些字符 `UTF-8` 编码的一个, 两个或三个转义序列。

当以一个参数 `uri` 调用 `encodeURI` 函数, 采用如下步骤:

1. 令 `uriString` 为 `ToString(uri)`.
2. 令 `unescapedURISet` 为一个字符串, 包含 `uriReserved` 和 `uriUnescaped` 的每个有效字符加上 `"#"` 的实例。
3. 返回调用 `Encode(uriString, unescapedURISet)` 的结果。

字符 `"#"` 不会被编码为一个转义序列, 即使它不是 `URI` 保留字符或非转义字符。

encodeURIComponent (uriComponent)

`encodeURIComponent` 函数计算出一个新版 `URI`, 将 `URI` 中某些字符的每个实例替换为代表这些字符 `UTF-8` 编码的一个, 两个或三个转义序列。

当以一个参数 `uriComponent` 调用 `encodeURIComponent` 函数, 采用如下步骤:

1. 令 `componentString` 为 `ToString(uriComponent)`.

2. 令 `unescapeURIComponentSet` 为一个字符串，包含 `uriUnescaped` 的每个有效字符的实例。
3. 返回调用 `Encode(componentString, unescapeURIComponentSet)` 的结果。

全局对象的构造器属性

Object (...)

见 15.2.1 和 15.2.2.

Function (...)

见 15.3.1 和 15.3.2

Array (...)

见 15.4.1 和 15.4.2.

String (...)

见 15.5.1 和 15.5.2.

Boolean (...)

见 15.6.1 和 15.6.2.

Number (...)

见 15.7.1 和 15.7.2.

Date (...)

见 15.9.2.

RegExp (...)

见 15.10.3 和 15.10.4.

Error (...)

见 15.11.1 和 15.11.2.

EvalError (...)

见 15.11.6.1.

RangeError (...)

见 15.11.6.2.

ReferenceError (...)

见 15.11.6.3.

SyntaxError (...)

见 15.11.6.4.

TypeError (...)

见 15.11.6.5.

URIError (...)

见 15.11.6.6.

全局对象的其他属性

Math

见 15.8.

JSON

见 15.12.

Object 对象

作为函数调用 Object 构造器

当把 Object 当做一个函数来调用，而不是一个构造器，它会执行一个类型转换。

Object ([value])

当以一个参数 value 或者无参数调用 Object 函数，采用如下步骤：

1. 如果 value 是 null, undefined 或未指定，则创建并返回一个新 Object 对象，这个对象与仿佛用相同参数调用标准内置的 Object 构造器 (15.2.2.1) 的结果一样。
2. 返回 ToObject(value).

Object 构造器

当 Object 是 new 表达式调用的一部分时，它是一个构造器，可创建一个对象。

new Object ([value])

当以一个参数 value 或者无参数调用 Object 构造器，采用如下步骤：

1. 如果提供了 value, 则
 1. 如果 Type(value) 是 Object, 则
 - i. 如果 value 是个原生 ECMAScript 对象，不创建新对象，简单的返回 value.
 - ii. 如果 value 是宿主对象，则采取动作和返回依赖实现的结果的方式可以使依赖于宿主对象的。
 2. 如果 Type(value) 是 String, 返回 ToObject(value).
 3. 如果 Type(value) 是 Boolean, 返回 ToObject(value).
 4. 如果 Type(value) 是 Number, 返回 ToObject(value).
2. 断言：未提供参数 value 或其类型是 Null 或 Undefined.
3. 令 obj 为一个新创建的原生 ECMAScript 对象。

4. 设定 obj 的 `[[Prototype]]` 内部属性为标准内置的 Object 的 prototype 对象 (15.2.4).
5. 设定 obj 的 `[[Class]]` 内部属性为 "Object".
6. 设定 obj 的 `[[Extensible]]` 内部属性为 true.
7. 设定 obj 的 8.12 指定的所有内部方法
8. 返回 obj.

Object 构造器的属性

Object 构造器的 `[[Prototype]]` 内部属性值是标准内置 Function 的 prototype 对象。

除了内部属性和 `length` 属性（其值是 1）之外，Object 构造器拥有以下属性：

Object.prototype

Object.prototype 的初始值是标准内置 Object 的 prototype 对象（15.2.4）。

这个属性包含特性 `{[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}`

Object.getPrototypeOf (O)

当以参数 O 调用 `getPrototypeOf` 函数，采用如下步骤：

1. 如果 `Type(O)` 不是 Object，则抛出一个 `TypeError` 异常。
2. 返回 O 的 `[[Prototype]]` 内部属性的值。

Object.getOwnPropertyDescriptor (O, P)

当调用 `getOwnPropertyDescriptor` 函数，采用如下步骤：

1. 如果 `Type(O)` 不是 Object，则抛出一个 `TypeError` 异常。
2. 令 name 为 `ToString(P)`.
3. 令 desc 为以参数 name 调用 O 的 `[[GetOwnProperty]]` 内部方法的结果。
4. 返回调用 `FromPropertyDescriptor(desc)` 的结果 (8.10.4).

Object.getOwnPropertyNames (O)

当调用 `getOwnPropertyNames` 函数，采用如下步骤：

1. 如果 `Type(O)` 不是 `Object`，则抛出一个 `TypeError` 异常。
2. 令 `array` 为仿佛是用表达式 `new Array()` 创建新对象的结果，这里的 `Array` 是标准内置构造器名。
3. 令 `n` 为 0。
4. 对 `O` 的每个自身属性 `P`
 1. 令 `name` 为值是 `P` 的名称的字符串。
 2. 以 `ToString(n)` 和属性描述 `{[[Value]]: name, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}` 和 `false` 为参数调用 `array` 的 `[[DefineOwnProperty]]` 内部方法。
3. `n` 递增 1。
5. 返回 `array`。

如果 `O` 是一个字符串实例，第 4 步处理的自身属性集合包含 15.5.5.2 定义的隐藏属性，他们对应对象的 `[[PrimitiveValue]]` 字符串中相应位置的字符。

Object.create (O [, Properties])

`create` 函数按照指定的原型创建一个新对象。当调用 `create` 函数，采用如下步骤：

1. 如果 `Type(O)` 不是 `Object` 或 `Null`，则抛出一个 `TypeError` 异常。
2. 令 `obj` 为仿佛是用表达式 `new Object()` 创建新对象的结果，这里的 `Object` 是标准内置构造器名。
3. 设定 `obj` 的 `[[Prototype]]` 内部属性为 `O`。
4. 如果传入了 `Properties` 参数并且不是 `undefined`，则仿佛是用 `obj` 和 `Properties` 当作参数调用标准内置函数 `Object.defineProperty` 一样给 `obj` 添加自身属性。
5. 返回 `obj`。

Object.defineProperty (O, P, Attributes)

`defineProperty` 函数用于给一个对象添加一个自身属性 并 / 或 更新现有自身属性的特性。当调用 `defineProperty` 函数，采用如下步骤：

1. 如果 `Type(O)` 不是 `Object`，则抛出一个 `TypeError` 异常。
2. 令 `name` 为 `ToString(P)`。
3. 令 `desc` 为以 `Attributes` 作为参数调用 `ToPropertyDescriptor` 的结果。
4. 以 `name, desc, true` 作为参数调用 `O` 的 `[[DefineOwnProperty]]` 内部方法。
5. 返回 `O`。

Object.defineProperties (O, Properties)

defineProperties 函数用于给一个对象添加一些自身属性 并 / 或 更新现有的一些自身属性的特性。当调用 **defineProperties** 函数，采用如下步骤：

1. 如果 **Type(O)** 不是 **Object**，则抛出一个 **TypeError** 异常。
2. 令 **props** 为 **ToObject(Properties)**。
3. 令 **names** 为一个内部列表，它包含 **props** 的每个可遍历自身属性的名称。
4. 令 **descriptors** 为一个空的内部列表。
5. 对 **names** 的每个元素 **P**，按照列表顺序，
 1. 令 **descObj** 为以 **P** 作为参数调用 **props** 的 **[[Get]]** 内部方法的结果。
 2. 令 **desc** 为以 **descObj** 作为参数调用 **ToPropertyDescriptor** 的结果。
 3. 将 **desc** 插入 **descriptors** 的尾部。
6. 对 **descriptors** 的每个元素 **desc**，按照列表顺序，
 1. 以参数 **P, desc, true** 调用 **O** 的 **[[DefineOwnProperty]]** 内部方法。
7. 返回 **O**

如果一个实现为 **for-in** 语句的定义了特定的枚举顺序，那么在这个算法的第 3 步中的列表元素必须也用相同的顺序排列。

Object.seal (O)

当调用 **seal** 函数，采用如下步骤：

1. 如果 **Type(O)** 不是 **Object**，则抛出一个 **TypeError** 异常。
2. 对 **O** 的每个命名自身属性名 **P**，
 1. 令 **desc** 为以参数 **P** 调用 **O** 的 **[[GetOwnProperty]]** 内部方法的结果。
 2. 如果 **desc.[[Configurable]]** 是 **true**，设定 **desc.[[Configurable]]** 为 **false**。
 3. 以 **P, desc, true** 为参数调用 **O** 的 **[[DefineOwnProperty]]** 内部方法。
3. 设定 **O** 的 **[[Extensible]]** 内部属性为 **false**。
4. 返回 **O**。

Object.freeze (O)

当调用 **freeze** 函数，采用如下步骤：

1. 如果 **Type(O)** 不是 **Object**，则抛出一个 **TypeError** 异常。
2. 对 **O** 的每个命名自身属性名 **P**，
 1. 令 **desc** 为以参数 **P** 调用 **O** 的 **[[GetOwnProperty]]** 内部方法的结果。
 2. 如果 **IsDataDescriptor(desc)** 是 **true**，则
 - i. 如果 **desc.[[Writable]]** 是 **true**，设定 **desc.[[Writable]]** 为 **false**。
 3. 如果 **desc.[[Configurable]]** 是 **true**，设定 **desc.[[Configurable]]** 为 **false**。
4. 以 **P, desc, true** 作为参数调用 **O** 的 **[[DefineOwnProperty]]** 内部方法。
3. 设定 **O** 的 **[[Extensible]]** 内部属性为 **false**。

4. 返回 O.

Object.preventExtensions (O)

当调用 preventExtensions 函数，采用如下步骤：

1. 如果 Type(O) 不是 Object，则抛出一个 TypeError 异常 .
2. 设定 O 的 [[Extensible]] 内部属性为 false.
3. 返回 O.

Object.isSealed (O)

当以参数 O 调用 isSealed 函数，采用如下步骤：

1. 如果 Type(O) 不是 Object，则抛出一个 TypeError 异常 .
2. 对 O 的每个命名自身属性名 P，
 1. 令 desc 为以参数 P 调用 O 的 [[GetOwnProperty]] 内部方法的结果 .
 2. 如果 desc.[[Configurable]] 是 true，则返回 false.
 3. 如果 O 的 [[Extensible]] 内部属性是 false，则返回 true.
 4. 否则，返回 false.

Object.isFrozen (O)

当以参数 O 调用 isFrozen 函数，采用如下步骤：

1. 如果 Type(O) 不是 Object，则抛出一个 TypeError 异常 .
2. 对 O 的每个命名自身属性名 P，
 1. 令 desc 为以参数 P 调用 O 的 [[GetOwnProperty]] 内部方法的结果 .
 2. 如果 IsDataDescriptor(desc) 是 true，则
 - i. 如果 desc.[[Writable]] 是 true，则返回 false.
 3. 如果 desc.[[Configurable]] 是 true，则返回 false.
 3. 如果 O 的 [[Extensible]] 内部属性是 false，则返回 true.
 4. 否则，返回 false.

Object.isExtensible (O)

当以参数 O 调用 isExtensible 函数，采用如下步骤：

1. 如果 Type(O) 不是 Object，则抛出一个 TypeError 异常 .
2. 返回 O 的 [[Extensible]] 内部属性布尔值 .

Object.keys (O)

当以参数 O 调用 keys 函数，采用如下步骤：

1. 如果 Type(O) 不是 Object，则抛出一个 TypeError 异常。
2. 令 n 为 O 的可遍历自身属性的个数
3. 令 array 为仿佛是用表达式 new Array () 创建新对象的结果，这里的 Array 是标准内置构造器名。
4. 令 index 为 0.
5. 对 O 的每个可遍历自身属性名 P，
 1. 以 ToString(index)，属性描述 {[Value]: P, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}，和 false 作为参数调用 array 的 [[DefineOwnProperty]] 内部方法。
 2. index 递增 1.
6. 返回 array.

如果一个实现为 for-in 语句的定义了特定的枚举顺序，那么在这个算法的第 5 步中的必须使用相同的枚举顺序。

Object 的 prototype 对象的属性

Object 的 prototype 对象的 [[Prototype]] 内部属性的值是 null，[[Class]] 内部属性的值是 "Object"，[[Extensible]] 内部属性的初始值是 true。

Object.prototype.constructor

Object.prototype.constructor 的初始值是标准内置的 Object 构造器。

Object.prototype.toString ()

当调用 toString 方法，采用如下步骤：

1. 如果 this 的值是 undefined，返回 "[object Undefined]"。
2. 如果 this 的值是 null，返回 "[object Null]"。
3. 令 O 为以 this 作为参数调用 ToObject 的结果。
4. 令 class 为 O 的 [[Class]] 内部属性的值。
5. 返回三个字符串 "[object ", class, and "]" 连起来的字符串。

Object.prototype.toLocaleString ()

当调用 `toLocaleString` 方法，采用如下步骤：

1. 令 `O` 为以 `this` 作为参数调用 `ToObject` 的结果。
2. 令 `toString` 为以 `"toString"` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
3. 如果 `IsCallable(toString)` 是 `false`，抛出一个 `TypeError` 异常。
4. 返回以 `O` 作为 `this` 值，无参数调用 `toString` 的 `[[Call]]` 内部方法的结果。

这个函数给所有 `Object` 对象提供一个通用的 `toLocaleString` 接口，即使并不是所有的都使用它。目前，`Array`, `Number`, `Date` 提供了它们自身的语言环境敏感的 `toLocaleString` 方法。

这个函数的第一个参数可能会在此标准的未来版本中使用到；因此建议实现不要用这个位置参数来做其他事情。

`Object.prototype.valueOf ()`

当调用 `valueOf` 方法，采用如下步骤：

1. 令 `O` 为以 `this` 作为参数调用 `ToObject` 的结果。
2. 如果 `O` 是以一个宿主对象 (15.2.2.1) 为参数调用 `Object` 构造器的结果，则
 1. 返回 `O` 或传递给构造器的原来的宿主对象。返回的具体结果是由实现定义的。
 3. 返回 `O`。

`Object.prototype.hasOwnProperty (V)`

当以参数 `V` 调用 `hasOwnProperty` 方法，采用如下步骤：

1. 令 `P` 为 `ToString(V)`。
2. 令 `O` 为以 `this` 值作为参数调用 `ToObject` 的结果。
3. 令 `desc` 为以 `P` 为参数调用 `O` 的 `[[GetOwnProperty]]` 内部方法的结果。
4. 如果 `desc` 是 `undefined`，返回 `false`。
5. 返回 `true`。

不像 `[[HasProperty]]`(8.12.6)，这个方法不考虑原形链中的对象。

为步骤 1 和 2 的选择这样的顺序，是为了确保在本规范之前版本中会在这里的步骤 1 里抛出的任何异常，即使 `this` 值是 `undefined` 或 `null`，也会继续抛出。

Object.prototype.isPrototypeOf (V)

当以参数 V 调用 isPrototypeOf 方法，采用如下步骤：

1. 如果 V 不是个对象，返回 false.
2. 令 O 为以 this 作为参数调用 ToObject 的结果 .
3. 重复
 1. 令 V 为 V 的 [[Prototype]] 内部属性的值 .
 2. 如果 V 是 null, 返回 false
 3. 如果 O 和 V 指向同一个对象，返回 true.

为步骤 1 和 2 的选择这样的顺序，是为了当 V 不是对象并且 this 值是 undefined 或 null 时能够保持本规范之前版本指定的行为。

Object.prototype.propertyIsEnumerable (V)

当以参数 V 调用 propertyIsEnumerable 方法，采用如下步骤：

1. 令 P 为 ToString(V).
2. 令 O 为以 this 作为参数调用 ToObject 的结果 .
3. 令 desc 为以 P 作为参数调用 O 的 [[GetOwnProperty]] 内部方法的结果 .
4. 如果 desc 是 undefined, 返回 false.
5. 返回 desc.Enumerable 的值 .

这个方法不考虑原型链中的对象。

为步骤 1 和 2 的选择这样的顺序，是为了确保在本规范之前版本中会在这里的步骤 1 里抛出的任何异常，即使 this 值是 undefined 或 null，也会继续抛出。

Object 的实例的属性

Object 的实例除了拥从 Object 的 prototype 对象继承来的属性之外不包含特殊的属性。

Function 对象

作为函数调用 Function 构造器

当将 **Function** 作为函数来调用，而不是作为构造器，它会创建并初始化一个新函数对象。所以函数调用 **Function(...)** 与用相同参数的 **new Function(...)** 表达式创建的对象相同。

Function (p1, p2, ... , pn, body)

当以 **p1, p2, ... , pn, body** 作为参数调用 **Function** 函数（这里的 **n** 可以是 0，也就是说没有“**p**”参数，这时还可以不提供 **body**），采用如下步骤：

1. 创建并返回一个新函数对象，它仿佛是用相同参数给标准内置构造器 **Function (15.3.2.1)** 用一个 **new** 表达式创建的。

Function 构造器

当 **Function** 作为 **new** 表达式的一部分被调用时，它是一个构造器：它初始化新创建的对象。

new Function (p1, p2, ... , pn, body)

最后一个参数指定为函数的 **body**(可执行代码)；之前的任何参数都指定为形式参数。

当以 **p1, p2, ... , pn, body** 作为参数调用 **Function** 构造器（这里的 **n** 可以是 0，也就是说没有“**p**”参数，这时还可以不提供 **body**），采用如下步骤：

1. 令 **argCount** 为传给这个函数调用的参数总数 .
2. 令 **P** 为空字符串 .
3. 如果 **argCount = 0**, 令 **body** 为空字符串 .
4. 否则如果 **argCount = 1**, 令 **body** 为那个参数 .
5. 否则 , **argCount > 1**
 1. 令 **firstArg** 为第一个参数 .
 2. 令 **P** 为 **ToString(firstArg)**.
 3. 令 **k** 为 2.
 4. 只要 **k < argCount** 就重复
 - i. 令 **nextArg** 为第 **k** 个参数 .
 - ii. 令 **P** 为之前的 **P** 值，字符串 **","**（一个逗号），**ToString(nextArg)** 串联的结果。
 - iii. **k** 递增 1.
 5. 令 **body** 为第 **k** 个参数 .
 6. 令 **body** 为 **ToString(body)**.
 7. 如果 **P** 不可解析为一个 **FormalParameterListopt**，则抛出一个 **SyntaxError** 异常 .

8. 如果 `body` 不可解析为 `FunctionBody`, 则抛出一个 `SyntaxError` 异常 .
9. 如果 `body` 是严格模式代码 (见 10.1.1), 则令 `strict` 为 `true`, 否则令 `strict` 为 `false`.
10. 如果 `strict` 是 `true`, 适用 13.1 指定抛出的任何异常 .
11. 返回一个新创建的函数对象, 它是依照 13.2 指定 -- 专递 `P` 作为 `FormalParameterList`, `body` 作为 `FunctionBody`, 全局环境作为 `Scope` 参数, `strict` 作为严格模式标志 -- 创建的。

每个函数都会自动创建一个 `prototype` 属性, 用来支持函数被当做构造器使用的可能性。

为每个形参指定一个参数是允许的, 但没必要。例如以下三个表达式产生相同的结果: `new Function("a", "b", "c", "return a+b+c")` `new Function("a, b, c", "return a+b+c")` `new Function("a,b", "c", "return a+b+c")`

Function 构造器的属性

`Function` 构造器自身是个函数对象, 它的 `[[Class]]` 是 `"Function"`。`Function` 构造器的 `[[Prototype]]` 内部属性值是标准内置 `Function` 的 `prototype` 对象 (15.3.4)。

`Function` 构造器的 `[[Extensible]]` 内部属性值是 `true`。

`Function` 构造器有如下属性 :

Function.prototype

`Function.prototype` 的初始值是标准内置 `Function` 的 `prototype` 对象 (15.3.4)。

此属性拥有特性 { `[[Writable]]`: `false`, `[[Enumerable]]`: `false`, `[[Configurable]]`: `false` }。

Function.length

这是个值为 `1` 的数据属性。此属性拥有特性 { `[[Writable]]`: `false`, `[[Enumerable]]`: `false`, `[[Configurable]]`: `false` }。

Function 的 prototype 对象的属性

`Function` 的 `prototype` 对象自身是一个函数对象 (它的 `[[Class]]` 是 `"Function"`), 调用这个函数对象时, 接受任何参数并返回 `undefined`。

Function 的 prototype 对象的 `[[Prototype]]` 内部属性值是标准内置 Object 的 prototype 对象 (15.2.4)。Function 的 prototype 对象的 `[[Extensible]]` 内部属性的初始值是 `true`。

Function 的 prototype 对象自身没有 `valueOf` 属性；但是，它从 Object 的 prototype 对象继承了 `valueOf` 属性。

Function 的 prototype 对象的 `length` 属性是 0。

Function.prototype.constructor

Function.prototype.constructor 的初始值是内置 Function 构造器。

Function.prototype.toString ()

此函数的返回值的表示是依赖于实现的。这个表示包含 `FunctionDeclaration` 的语法。特别注意，怎样在这个字符串表示中使用和放置空白，行终结符，分号是依赖于实现的。

这个 `toString` 不是通用的；如果它的 `this` 值不是一个函数对象，它会抛出一个 `TypeError` 异常。因此，它不能当做方法来转移到其他类型的对象中。

Function.prototype.apply (thisArg, argArray)

当以 `thisArg` 和 `argArray` 为参数在一个 `func` 对象上调用 `apply` 方法，采用如下步骤：

1. 如果 `IsCallable(func)` 是 `false`，则抛出一个 `TypeError` 异常。
2. 如果 `argArray` 是 `null` 或 `undefined`，则
 1. 返回提供 `thisArg` 作为 `this` 值并以空参数列表调用 `func` 的 `[[Call]]` 内部方法的结果。
3. 如果 `Type(argArray)` 不是 `Object`，则抛出一个 `TypeError` 异常。
4. 令 `len` 为以 `"length"` 作为参数调用 `argArray` 的 `[[Get]]` 内部方法的结果。
5. 令 `n` 为 `ToUint32(len)`。
6. 令 `argList` 为一个空列表。
7. 令 `index` 为 0。
8. 只要 `index < n` 就重复
 1. 令 `indexName` 为 `ToString(index)`。
 2. 令 `nextArg` 为以 `indexName` 作为参数调用 `argArray` 的 `[[Get]]` 内部方法的结果。
 3. 将 `nextArg` 作为最后一个元素插入到 `argList` 里。
 4. 设定 `index` 为 `index + 1`。

9. 提供 `thisArg` 作为 `this` 值并以 `argList` 作为参数列表，调用 `func` 的 `[[Call]]` 内部方法，返回结果。

`apply` 方法的 `length` 属性是 2。

在外面传入的 `thisArg` 值会修改并成为 `this` 值。`thisArg` 是 `undefined` 或 `null` 时它会被替换成全局对象，所有其他值会被应用 `ToObject` 并将结果作为 `this` 值，这是第三版引入的更改。

Function.prototype.call (thisArg [, arg1 [, arg2, ...]])

当以 `thisArg` 和可选的 `arg1, arg2` 等等作为参数在一个 `func` 对象上调用 `call` 方法，采用如下步骤：

1. 如果 `IsCallable(func)` 是 `false`，则抛出一个 `TypeError` 异常。
2. 令 `argList` 为一个空列表。
3. 如果调用这个方法参数多余一个，则从 `arg1` 开始以从左到右的顺序将每个参数插入为 `argList` 的最后一个元素。
4. 提供 `thisArg` 作为 `this` 值并以 `argList` 作为参数列表，调用 `func` 的 `[[Call]]` 内部方法，返回结果。

`call` 方法的 `length` 属性是 1。

在外面传入的 `thisArg` 值会修改并成为 `this` 值。`thisArg` 是 `undefined` 或 `null` 时它会被替换成全局对象，所有其他值会被应用 `ToObject` 并将结果作为 `this` 值，这是第三版引入的更改。

Function.prototype.bind (thisArg [, arg1 [, arg2, ...]])

`bind` 方法需要一个或更多参数，`thisArg` 和（可选的）`arg1, arg2`，等等，执行如下步骤返回一个新函数对象：

1. 令 `Target` 为 `this` 值。
2. 如果 `IsCallable(Target)` 是 `false`，抛出一个 `TypeError` 异常。
3. 令 `A` 为一个（可能为空的）新内部列表，它包含按顺序的 `thisArg` 后面的所有参数（`arg1, arg2` 等等）。
4. 令 `F` 为一个新原生 `ECMAScript` 对象。
5. 依照 8.12 指定，设定 `F` 的除了 `[[Get]]` 之外的所有内部方法。
6. 依照 15.3.5.4 指定，设定 `F` 的 `[[Get]]` 内部属性。
7. 设定 `F` 的 `[[TargetFunction]]` 内部属性为 `Target`。
8. 设定 `F` 的 `[[BoundThis]]` 内部属性为 `thisArg` 的值。
9. 设定 `F` 的 `[[BoundArgs]]` 内部属性为 `A`。
10. 设定 `F` 的 `[[Class]]` 内部属性为 `"Function"`。

11. 设定 F 的 `[[Prototype]]` 内部属性为 15.3.3.1 指定的标准内置 Function 的 prototype 对象。
12. 依照 15.3.4.5.1 描述, 设定 F 的 `[[Call]]` 内置属性。
13. 依照 15.3.4.5.2 描述, 设定 F 的 `[[Construct]]` 内置属性。
14. 依照 15.3.4.5.3 描述, 设定 F 的 `[[HasInstance]]` 内置属性。
15. 如果 Target 的 `[[Class]]` 内部属性是 "Function", 则
 1. 令 L 为 Target 的 length 属性减 A 的长度。
 2. 设定 F 的 length 自身属性为 0 和 L 中更大的值。
16. 否则设定 F 的 length 自身属性为 0。
17. 设定 F 的 length 自身属性的特性为 15.3.5.1 指定的值。
18. 设定 F 的 `[[Extensible]]` 内部属性为 true。
19. 令 thrower 为 `[[ThrowTypeError]]` 函数对象 (13.2.3)。
20. 以 "caller", 属性描述符 `{[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}`, 和 false 作为参数调用 F 的 `[[DefineOwnProperty]]` 内部方法。
21. 以 "arguments", 属性描述符 `{[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}`, 和 false 作为参数调用 F 的 `[[DefineOwnProperty]]` 内部方法。
22. 返回 F。

bind 方法的 length 属性是 1。

`Function.prototype.bind` 创建的函数对象不包含 `prototype` 属性或 `[[Code]]`, `[[FormalParameters]]`, `[[Scope]]` 内部属性。

`[[Call]]`

当调用一个用 bind 函数创建的函数对象 F 的 `[[Call]]` 内部方法, 传入一个 this 值和一个参数列表 ExtraArgs, 采用如下步骤:

1. 令 boundArgs 为 F 的 `[[BoundArgs]]` 内部属性值。
2. 令 boundThis 为 F 的 `[[BoundThis]]` 内部属性值。
3. 令 target 为 F 的 `[[TargetFunction]]` 内部属性值。
4. 令 args 为一个新列表, 它包含与列表 boundArgs 相同顺序相同值, 后面跟着与 ExtraArgs 是相同顺序相同值。
5. 提供 boundThis 作为 this 值, 提供 args 为参数调用 target 的 `[[Call]]` 内部方法, 返回结果。

`[[Construct]]`

当调用一个用 bind 函数创建的函数对象 F 的 `[[Construct]]` 内部方法, 传入一个参数列表 ExtraArgs, 采用如下步骤:

1. 令 `target` 为 `F` 的 `[[TargetFunction]]` 内部属性值。
2. 如果 `target` 不包含 `[[Construct]]` 内部方法，抛出一个 `TypeError` 异常。
3. 令 `boundArgs` 为 `F` 的 `[[BoundArgs]]` 内部属性值。
4. 令 `args` 为一个新列表，它包含与列表 `boundArgs` 相同顺序相同值，后面跟着与 `ExtraArgs` 是相同顺序相同值。
5. 提供 `args` 为参数调用 `target` 的 `[[Construct]]` 内部方法，返回结果。

`[[HasInstance]]` (V)

当调用一个用 `bind` 函数创建的函数对象 `F` 的 `[[Construct]]` 内部方法，并以 `V` 作为参数，采用如下步骤：

1. 令 `target` 为 `F` 的 `[[TargetFunction]]` 内部属性值。
2. 如果 `target` 不包含 `[[HasInstance]]` 内部方法，抛出一个 `TypeError` 异常。
3. 提供 `V` 为参数调用 `target` 的 `[[HasInstance]]` 内部方法，返回结果。

Function 的实例的属性

除了必要的内部属性之外，每个函数实例还有一个 `[[Call]]` 内部属性并且在大多数情况下使用不同版本的 `[[Get]]` 内部属性。函数实例根据怎样创建的（见 8.6.2, 13.2, 15, 15.3.4.5）可能还有一个 `[[HasInstance]]` 内部属性，一个 `[[Scope]]` 内部属性，一个 `[[Construct]]` 内部属性，一个 `[[FormalParameters]]` 内部属性，一个 `[[Code]]` 内部属性，一个 `[[TargetFunction]]` 内部属性，一个 `[[BoundThis]]` 内部属性，一个 `[[BoundArgs]]` 内部属性。

`[[Class]]` 内部属性的值是 `"Function"`。

对应于严格模式函数 (13.2) 的函数实例和用 `Function.prototype.bind` 方法 (15.3.4.5) 创建的函数实例有名为 `"caller"` 和 `"arguments"` 的属性时，抛出一个 `TypeError` 异常。一个 ECMAScript 实现不得为在严格模式函数代码里访问这些属性关联任何依赖实现的特定行为。

length

`length` 属性值是个整数，它指出函数预期的“一般的”参数个数。然而，语言允许用其他数量的参数来调用函数。当以与函数的 `length` 属性指定的数量不同的参数个数调用函数时，它的行为依赖于函数自身。这个属性拥有特性 `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`。

prototype

`prototype` 属性的值用于初始化一个新创建对象的 `[[Prototype]]` 内部属性，为了这个新创建对象要先将函数对象作为构造器调用。这个属性拥有特性 `{ [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false }`。

用 `Function.prototype.bind` 创建的函数对象没有 `prototype` 属性。

`[[HasInstance]] (V)`

设 `F` 是个函数对象。

当以 `V` 作为参数调用 `F` 的 `[[HasInstance]]` 内部方法，采用如下步骤：

1. 如果 `V` 不是个对象，返回 `false`。
2. 令 `O` 为用属性名 `"prototype"` 调用 `F` 的 `[[Get]]` 内部方法的结果。
3. 如果 `Type(O)` 不是 `Object`，抛出一个 `TypeError` 异常。
4. 重复
 1. 令 `V` 为 `V` 的 `[[Prototype]]` 内部属性值。
 2. 如果 `V` 是 `null`，返回 `false`。
 3. 如果 `O` 和 `V` 指向相同对象，返回 `true`。

用 `Function.prototype.bind` 创建的函数对象拥有的不同的 `[[HasInstance]]` 实现，在 15.3.4.5.3 中定义。

`[[Get]] (P)`

函数对象与其他原生 ECMAScript 对象 (8.12.3) 用不同的 `[[Get]]` 内部方法。

设 `F` 是一个函数对象，当以属性名 `P` 调用 `F` 的 `[[Get]]` 内部方法，采用如下步骤：

1. 令 `v` 为传入 `P` 作为属性名参数调用 `F` 的默认 `[[Get]]` 内部方法 (8.12.3) 的结果。
2. 如果 `P` 是 `"caller"` 并且 `v` 是个严格模式函数对象，抛出一个 `TypeError` 异常。
3. 返回 `v`。

用 `Function.prototype.bind` 创建的函数对象使用默认的 `[[Get]]` 内部方法。

Array 对象

数组对象会给予一些种类的属性名特殊待遇。对一个属性名 `P`（字符串形式），当且仅当 `ToString(ToUint32(P))` 等于 `P` 并且 `ToUint32(P)` 不等于 $2^{32}-1$ 时，它是个 数组索引。一个属性名是数组索引的属性还叫做 元素。所有数组对象

都有一个 `length` 属性，其值始终是一个小于 2^{32} 的非负整数。`length` 属性值在数值上比任何名为数组索引的属性的名称还要大；每当创建或更改一个数组对象的属性，要调整其他的属性以保持上面那个条件不变的需要。具体来说，每当添加一个名为数组索引的属性时，如果需要就更改 `length` 属性为在数值上比这个数组索引大 1 的值；每当更改 `length` 属性，所有属性名是数组索引并且其值不小于新 `length` 的属性会被自动删除。这个限制只应用于数组对象的自身属性，并且从原型中继承的 `length` 或数组索引不影响这个限制。

对一个对象 `O`，如果以下算法返回 `true`，那么就叫这个对象为 稀疏 的：

1. 令 `len` 为以 `"length"` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
2. 对每个范围在 $0 \leq i < \text{ToUint32}(\text{len})$ 的整数 `i`
 1. 令 `elem` 为以 `ToString(i)` 作为参数调用 `O` 的 `[[GetOwnProperty]]` 内部方法的结果。
 2. 如果 `elem` 是 `undefined`，返回 `true`。
 3. 返回 `false`。

作为函数调用 Array 构造器

当将 `Array` 作为函数来调用，而不是作为构造器，它会创建并初始化一个新数组对象。所以函数调用 `Array(...)` 与用相同参数的 `new Array(...)` 表达式创建的对象相同。

Array ([item1 [, item2 [, ...]]])

当调用 `Array` 函数，采用如下步骤：

1. 创建并返回一个新函数对象，它仿佛是用相同参数给标准内置构造器 `Array` 用一个 `new` 表达式创建的 (15.4.2)。

Array 构造器

当 `Array` 作为 `new` 表达式的一部分被调用时，它是一个构造器：它初始化新创建的对象。

new Array ([item0 [, item1 [, ...]]])

当且仅当以无参数或至少两个参数调用 `Array` 构造器时，适用这里的描述。

新构造对象的 `[[Prototype]]` 内部属性要设定为原始的数组原型对象，他是 `Array.prototype`(15.4.3.1) 的初始值。

新构造对象的 `[[Class]]` 内部属性要设定为 `"Array"`。

新构造对象的 `[[Extensible]]` 内部属性要设定为 `true`。

新构造对象的 `length` 属性要设定为参数的个数。

新构造对象的 `0` 属性要设定为 `item0`(如果提供了); 新构造对象的 `1` 属性要设定为 `item1`(如果提供了); 更多的参数可应用普遍规律, 新构造对象的 `k` 属性要设定为第 `k` 个参数, 这里的 `k` 是从 `0` 开始的。所有这些属性都有特性 `{[[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`。

new Array (len)

新构造对象的 `[[Prototype]]` 内部属性要设定为原始的数组原型对象, 他是 `Array.prototype(15.4.3.1)` 的初始值。新构造对象的 `[[Class]]` 内部属性要设定为 `"Array"`。新构造对象的 `[[Extensible]]` 内部属性要设定为 `true`。

如果参数 `len` 是个数字值并且 `ToUint32(len)` 等于 `len`, 则新构造对象的 `length` 属性要设定为 `ToUint32(len)`。如果参数 `len` 是个数字值并且 `ToUint32(len)` 不等于 `len`, 则抛出一个 `RangeError` 异常。

如果参数 `len` 不是数字值, 则新构造对象的 `length` 属性要设定为 `0`, 并且新构造对象的 `0` 属性要设定为 `len`, 设定它的特性为 `{[[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`。

Array 构造器的属性

`Array` 构造器的 `[[Prototype]]` 内部属性值是函数原型对象 (15.3.4)。

`Array` 构造器除了有一些内部属性和 `length` 属性 (其值是 `1`) 之外, 还有如下属性:

Array.prototype

`Array.prototype` 的初始值是数组原型对象 (15.4.4)。

此属性拥有特性 `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`。

Array.isArray (arg)

`isArray` 函数需要一个参数 `arg`，如果参数是个对象并且 `class` 内部属性是 `"Array"`，返回布尔值 `true`；否则它返回 `false`。采用如下步骤：

1. 如果 `Type(arg)` 不是 `Object`，返回 `false`。
2. 如果 `arg` 的 `[[Class]]` 内部属性值是 `"Array"`，则返回 `true`。
3. 返回 `false`。

数组原型对象的属性

数组原型对象的 `[[Prototype]]` 内部属性值是标准内置 `Object` 原型对象 (15.2.4)。

数组原型对象自身是个数组；它的 `[[Class]]` 是 `"Array"`，它拥有一个 `length` 属性(初始值是 `+0`)和 15.4.5.1 描述的特殊的 `[[DefineOwnProperty]]` 内部方法。

在以下的对数组原型对象的属性函数的描述中，短语“`this` 对象”指的是调用这个函数时的 `this` 值对象。允许 `this` 是 `[[Class]]` 内部属性值不是 `"Array"` 的对象。

数组原型对象不包含它自身的 `valueOf` 属性；然而，它从标准内置 `Object` 原型对象继承 `valueOf` 属性。

`Array.prototype.constructor`

`Array.prototype.constructor` 的初始值是标准内置 `Array` 构造器。

`Array.prototype.toString()`

当调用 `toString` 方法，采用如下步骤：

1. 令 `array` 为用 `this` 值调用 `ToObject` 的结果。
2. 令 `func` 为以 `"join"` 作为参数调用 `array` 的 `[[Get]]` 内部方法的结果。
3. 如果 `IsCallable(func)` 是 `false`，则令 `func` 为标准内置方法 `Object.prototype.toString` (15.2.4.2)。
4. 提供 `array` 作为 `this` 值并以空参数列表调用 `func` 的 `[[Call]]` 内部方法，返回结果。

`toString` 函数被有意设计成通用的；它的 `this` 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `toString` 函数是依赖于实现的。

`Array.prototype.toLocaleString()`

先用数组元素的 `toLocaleString` 方法，将他们转换成字符串。然后将这些字符串串联，用一个分隔符分割，这里的分隔符字符串是与特定语言环境相关，由实现定义的方式得到的。调用这个函数的结果除了与特定语言环境关联之外，与 `toString` 的结果类似。

结果是按照以下方式计算的：

1. 令 `array` 为以 `this` 值作为参数调用 `ToObject` 的结果。
2. 令 `arrayLen` 为以 `"length"` 作为参数调用 `array` 的 `[[Get]]` 内部方法的结果。
3. 令 `len` 为 `ToUint32(arrayLen)`。
4. 令 `separator` 为宿主环境的当前语言环境对应的列表分隔符字符串（这是实现定义的方式得到的）。
5. 如果 `len` 是零，返回空字符串。
6. 令 `firstElement` 为以 `"0"` 作为参数调用 `array` 的 `[[Get]]` 内部方法的结果。
7. 如果 `firstElement` 是 `undefined` 或 `null`，则
 1. 令 `R` 为空字符串。
8. 否则
 1. 令 `elementObj` 为 `ToObject(firstElement)`。
 2. 令 `func` 为以 `"toLocaleString"` 作为参数调用 `elementObj` 的 `[[Get]]` 内部方法的结果。
 3. 如果 `IsCallable(func)` 是 `false`，抛出一个 `TypeError` 异常。
 4. 令 `R` 为提供 `elementObj` 作为 `this` 值并以空参数列表调用 `func` 的 `[[Call]]` 内部方法的结果。
9. 令 `k` 为 1。
10. 只要 `k < len` 就重复
 1. 令 `S` 为串联 `R` 和 `separator` 产生的字符串。
 2. 令 `nextElement` 为以 `Tostring(k)` 作为参数调用 `array` 的 `[[Get]]` 内部方法的结果。
 3. 如果 `nextElement` 是 `undefined` 或 `null`，则
 - i. 令 `R` 为空字符串。
 4. 否则
 - i. 令 `elementObj` 为 `ToObject(nextElement)`。
 - ii. 令 `func` 为以 `"toLocaleString"` 作为参数调用 `elementObj` 的 `[[Get]]` 内部方法的结果。
 - iii. 如果 `IsCallable(func)` 是 `false`，抛出一个 `TypeError` 异常。
 - iv. 令 `R` 为提供 `elementObj` 作为 `this` 值并以空参数列表调用 `func` 的 `[[Call]]` 内部方法的结果。
 5. 令 `R` 为串联 `S` 和 `R` 产生的字符串。
 6. `k` 递增 1。
11. 返回 `R`

此函数的第一个参数可能会在本标准的未来版本中用到；建议实现不要以任何其他用途使用这个参数位置。

`toLocaleString` 函数被有意设计成通用的；它的 `this` 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `toLocaleString` 函数是依赖于实现的。

Array.prototype.concat ([item1 [, item2 [, ...]]])

当以零或更多个参数 `item1`, `item2`, 等等，调用 `concat` 方法，返回一个数组，这个数组包含对象的数组元素和后面跟着的每个参数按照顺序组成的数组元素。

采用如下步骤：

1. 令 `O` 为以 `this` 值作为参数调用 `ToObject` 的结果。
2. 令 `A` 为仿佛是用表达式 `new Array()` 创建的新数组，这里的 `Array` 是标准内置构造器名。
3. 令 `n` 为 0。
4. 令 `items` 为一个内部列表，他的第一个元素是 `O`，其后的元素是传给这个函数调用的参数（以从左到右的顺序）。
5. 只要 `items` 不是空就重复
 1. 删除 `items` 的第一个元素，并令 `E` 为这个元素值。
 2. 如果 `E` 的 `[[Class]]` 内部属性是 "Array"，则
 - i. 令 `k` 为 0。
 - ii. 令 `len` 为以 "length" 为参数调用 `E` 的 `[[Get]]` 内部方法的结果。
 - iii. 只要 `k < len` 就重复
 1. 令 `P` 为 `ToString(k)`。
 2. 令 `exists` 为以 `P` 作为参数调用 `E` 的 `[[HasProperty]]` 内部方法的结果。
 3. 如果 `exists` 是 `true`，则
 1. 令 `subElement` 为以 `P` 作为参数调用 `E` 的 `[[Get]]` 内部方法的结果。
 2. 以 `ToString(n)`，属性描述符 `{[[Value]]: subElement, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`，和 `false` 作为参数调用 `A` 的 `[[DefineOwnProperty]]` 内部方法。
 4. `n` 递增 1。
 5. `k` 递增 1。
 3. 否则，`E` 不是数组
 - i. 以 `ToString(n)`，属性描述符 `{[[Value]]: E, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`，和 `false` 作为参数调用 `A` 的 `[[DefineOwnProperty]]` 内部方法。
 - ii. `n` 递增 1。
6. 返回 `A`。

`concat` 方法的 `length` 属性是 1。

`concat` 函数被有意设计成通用的；它的 `this` 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `concat` 函数是依赖于实现的。

Array.prototype.join (separator)

数组元素先被转换为字符串，再将这些字符串用 `separator` 分割连接在一起。如果没提供分隔符，将一个逗号用作分隔符。

`join` 方法需要一个参数 `separator`，执行以下步骤：

1. 令 `O` 为以 `this` 值作为参数调用 `ToObject` 的结果。
2. 令 `lenVal` 为以 `"length"` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
3. 令 `len` 为 `ToUint32(lenVal)`。
4. 如果 `separator` 是 `undefined`，令 `separator` 为单字符字符串 `","`。
5. 令 `sep` 为 `ToString(separator)`。
6. 如果 `len` 是零，返回空字符串。
7. 令 `element0` 为以 `"0"` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
8. 如果 `element0` 是 `undefined` 或 `null`，令 `R` 为空字符串；否则，令 `R` 为 `ToString(element0)`。
9. 令 `k` 为 1。
10. 只要 `k < len` 就重复
 1. 令 `S` 为串联 `R` 和 `sep` 产生的字符串值。
 2. 令 `element` 为以 `ToString(k)` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
 3. 如果 `element` 是 `undefined` 或 `null`，令 `next` 为空字符串；否则，令 `next` 为 `ToString(element)`。
 4. 令 `R` 为串联 `S` 和 `next` 产生的字符串值。
 5. `k` 递增 1。
11. 返回 `R`。

`join` 方法的 `length` 属性是 1。

`join` 函数被有意设计成通用的；它的 `this` 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `join` 函数是依赖于实现的。

Array.prototype.pop ()

删除并返回数组的最后一个元素。

1. 令 `O` 为以 `this` 值作为参数调用 `ToObject` 的结果。
2. 令 `lenVal` 为以 `"length"` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
3. 令 `len` 为 `ToUint32(lenVal)`。
4. 如果 `len` 是零，
 1. 以 `"length"`, 0, 和 `true` 作为参数调用 `O` 的 `[[Put]]` 内部方法。
 2. 返回 `undefined`。

5. 否则 , `len > 0`
1. 令 `indx` 为 `ToString(len-1)`.
2. 令 `element` 为 以 `indx` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果 .
3. 以 `indx` 和 `true` 作为参数调用 `O` 的 `[[Delete]]` 内部方法 .
4. 以 `"length"`, `indx`, 和 `true` 作为参数调用 `O` 的 `[[Put]]` 内部方法 .
5. 返回 `element`.

`pop` 函数被有意设计成通用的；它的 `this` 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `pop` 函数是依赖于实现的。

Array.prototype.push ([item1 [, item2 [, ...]]])

将参数以他们出现的顺序追加到数组末尾。数组的新 `length` 属性值会作为调用的结果返回。

当以零或更多个参数 `item1`, `item2`, 等等，调用 `push` 方法，采用以下步骤：

1. 令 `O` 为以 `this` 值作为参数调用 `ToObject` 的结果 .
2. 令 `lenVal` 为以 `"length"` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果 .
3. 令 `n` 为 `ToUint32(lenVal)`.
4. 令 `items` 为一个内部列表，它的元素是调用这个函数时传入的参数（从左到右的顺序）.
5. 只要 `items` 不是空就重复
 1. 删除 `items` 的第一个元素，并令 `E` 为这个元素的值 .
 2. 以 `ToString(n)`, `E`, 和 `true` 作为参数调用 `O` 的 `[[Put]]` 内部方法 .
 3. `n` 递增 1.
6. 以 `"length"`, `n`, 和 `true` 作为参数调用 `O` 的 `[[Put]]` 内部方法 .
7. 返回 `n`.

`push` 方法的 `length` 属性是 1。

`push` 函数被有意设计成通用的；它的 `this` 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `push` 函数是依赖于实现的。

Array.prototype.reverse ()

重新排列数组元素，以翻转它们的顺序。对象会被当做调用的结果返回。

1. 令 `O` 为以 `this` 值作为参数调用 `ToObject` 的结果 .
2. 令 `lenVal` 为以 `"length"` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果 .
3. 令 `len` 为 `ToUint32(lenVal)`.

4. 令 middle 为 floor(len/2).
5. 令 lower 为 0.
6. 只要 lower \neq middle 就重复
 1. 令 upper 为 len-lower-1.
 2. 令 upperP 为 ToString(upper).
 3. 令 lowerP 为 ToString(lower).
 4. 令 lowerValue 为以 lowerP 作为参数调用 O 的 [[Get]] 内部方法的结果 .
 5. 令 upperValue 为以 upperP 作为参数调用 O 的 [[Get]] 内部方法的结果 .
 6. 令 lowerExists 为以 lowerP 作为参数调用 O 的 [[HasProperty]] 内部方法的结果 .
 7. 令 upperExists 为以 upperP 作为参数调用 O 的 [[HasProperty]] 内部方法的结果 .
 8. 如果 lowerExists 是 true 并且 upperExists 是 true, 则
 - i. 以 lowerP, upperValue, 和 true 作为参数调用 O 的 [[Put]] 内部方法 .
 - ii. 以 upperP, lowerValue, 和 true 作为参数调用 O 的 [[Put]] 内部方法 .
 9. 否则如果 lowerExists 是 false 并且 upperExists 是 true, 则
 - i. 以 lowerP, upperValue, 和 true 作为参数调用 O 的 [[Put]] 内部方法 .
 - ii. 以 upperP 和 true 作为参数调用 O 的 [[Delete]] 内部方法 .
 10. 否则如果 lowerExists 是 true 并且 upperExists 是 false, 则
 - i. 以 lowerP 和 true 作为参数调用 O 的 [[Delete]] 内部方法 .
 - ii. 以 upperP, lowerValue, 和 true 作为参数调用 O 的 [[Put]] 内部方法 .
 11. 否则 , lowerExists 和 upperExists 都是 false
 - i. 不需要做任何事情 .
12. lower 递增 1.
7. 返回 O .

reverse 函数被有意设计成通用的；它的 **this** 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 **reverse** 函数是依赖于实现的。

Array.prototype.shift ()

删除并返回数组的第一个元素。

1. 令 O 为以 this 值作为参数调用 ToObject 的结果 .
2. 令 lenVal 为以 "length" 作为参数调用 O 的 [[Get]] 内部方法的结果 .
3. 令 len 为 ToUint32(lenVal).
4. 如果 len 是零 , 则
 1. 以 "length", 0, 和 true 作为参数调用 O 的 [[Put]] 内部方法 .
2. 返回 undefined.
5. 令 first 为以 "0" 作为参数调用 O 的 [[Get]] 内部方法的结果 .
6. 令 k 为 1.

7. 只要 $k < \text{len}$ 就重复
 1. 令 `from` 为 `ToString(k)`.
 2. 令 `to` 为 `ToString(k-1)`.
 3. 令 `fromPresent` 为以 `from` 作为参数调用 `O` 的 `[[HasProperty]]` 内部方法的结果 .
 4. 如果 `fromPresent` 是 `true`, 则
 - i. 令 `fromVal` 为以 `from` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果 .
 - ii. 以 `to`, `fromVal`, 和 `true` 作为参数调用 `O` 的 `[[Put]]` 内部方法 .
 5. 否则 , `fromPresent` 是 `false`
 - i. 以 `to` 和 `ture` 作为参数调用 `O` 的 `[[Delete]]` 内部方法 .
6. k 递增 1.
8. 以 `ToString(len-1)` 和 `true` 作为参数调用 `O` 的 `[[Delete]]` 内部方法 .
9. 以 `"length"`, `(len-1)`, 和 `true` 作为参数调用 `O` 的 `[[Put]]` 内部方法 .
10. 返回 `first`.

`shift` 函数被有意设计成通用的；它的 `this` 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `shift` 函数是依赖于实现的。

Array.prototype.slice (start, end)

`slice` 方法需要 `start` 和 `end` 两个参数, 返回一个数组, 这个数组包含从第 `start` 个元素到 -- 但不包括 -- 第 `end` 个元素 (或如果 `end` 是 `undefined` 就到数组末尾)。如果 `start` 为负, 它会被当做是 `length+start`, 这里的 `length` 是数组长度。如果 `end` 为负, 它会被当做是 `length+end`, 这里的 `length` 是数组长度。采用如下步骤:

1. 令 `O` 为以 `this` 值作为参数调用 `ToObject` 的结果 .
2. 令 `A` 为仿佛用表达式 `new Array()` 创建的新数组, 这里的 `Array` 是标准内置构造器名 .
3. 令 `lenVal` 为以 `"length"` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果 .
4. 令 `len` 为 `ToUint32(lenVal)`.
5. 令 `relativeStart` 为 `ToInteger(start)`.
6. 如果 `relativeStart` 为负, 令 `k` 为 `max((len + relativeStart), 0)`; 否则令 `k` 为 `min(relativeStart, len)`.
7. 如果 `end` 是 `undefined`, 令 `relativeEnd` 为 `len`; 否则令 `relativeEnd` 为 `ToInteger(end)`.
8. 如果 `relativeEnd` 为负, 令 `final` 为 `max((len + relativeEnd), 0)`; 否则令 `final` 为 `min(relativeEnd, len)`.
9. 令 `n` 为 0.
10. 只要 $k < \text{final}$ 就重复
 1. 令 `Pk` 为 `ToString(k)`.
 2. 令 `kPresent` 为 `Pk` 作为参数调用 `O` 的 `[[HasProperty]]` 内部方法的结果 .

3. 如果 `kPresent` 是 `true`, 则
 - i. 令 `kValue` 为以 `Pk` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果 .
 - ii. 以 `ToString(n)`, 属性描述符 `{[[Value]]: kValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, 和 `false` 作为参数调用 `A` 的 `[[DefineOwnProperty]]` 内部方法 .
4. `k` 递增 1.
5. `n` 递增 1.
11. 返回 `A`.

`slice` 方法的 `length` 属性是 2。

`slice` 函数被有意设计成通用的；它的 `this` 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `slice` 函数是依赖于实现的。

Array.prototype.sort (comparefn)

给 `this` 数组的元素 排序。排序不一定是稳定的（相等的元素们不一定按照他们原来的顺序排列）。如果 `comparefn` 不是 `undefined`，它就必须是个函数，这个函数接受两个参数 `x` 和 `y`，如果 `x < y` 返回一个负值，如果 `x = y` 返回零，如果 `x > y` 返回一个正值。

令 `obj` 为以 `this` 值作为参数调用 `ToObject` 的结果。

以 `"length"` 作为参数调用 `obj` 的 `[[Get]]` 内部方法，将结果作为参数调用 `Uint32`，令 `len` 为返回的结果。

如果 `comparefn` 不是 `undefined` 并且不是对 `this` 数组的元素 保持一致的比较函数（见下面），那么这种情况下 `sort` 的行为是实现定义的。

令 `proto` 为 `obj` 的 `[[Prototype]]` 内部属性。如果 `proto` 不是 `null` 并且存在一个整数 `j` 满足下面列出的全部条件，那么这种情况下 `sort` 的行为是实现定义的：

- `obj` 是 稀疏的 (15.4)
- $0 \leq j < \text{len}$
- 以 `ToString(j)` 作为参数调用 `proto` 的 `[[HasProperty]]` 内部方法的结果是 `true`

如果 `obj` 是 稀疏的 并且以下任何条件为真，那么这种情况下 `sort` 的行为是实现定义的：

- `obj` 的 `[[Extensible]]` 内部属性是 `false`.
- 任何名为小于 `len` 的非负整数的数组索引属性中，有 `[[Configurable]]` 特性是 `false` 的数据属性。

任何名为小于 `len` 的非负整数的数组索引属性中，有访问器属性，或有 `[[Writable]]` 特性是 `false` 的数据属性，那么这种情况下 `sort` 的行为是实现定义的。

否则，采用如下步骤。

1. 对 `obj` 的 `[[Get]]`, `[[Put]]`, `[[Delete]]` 内部方法和 `SortCompare` (下面描述) 执行一个依赖于实现的调用序列，这里对每个 `[[Get]]`, `[[Put]]`, 或 `[[Delete]]` 调用的第一个参数是小于 `len` 的非负整数，`SortCompare` 调用的参数是前面调用 `[[Get]]` 内部方法的结果。调用 `[[Put]]` 和 `[[Delete]]` 内部方法时，`throw` 参数是 `true`。如果 `obj` 不是稀疏的，则必须不调用 `[[Delete]]`。
2. 返回 `obj`。

返回的对象必须拥有下面两个性质。

- 必须有这样的数学排列 π ，它是由比 `len` 小的非负整数组成，对于每个比 `len` 小的非负整数 j ，如果属性 `old[j]` 存在，则 `new[$\pi(j)$]` 有与 `old[j]` 相同的值，如果属性 `old[j]` 不存在，则 `new[$\pi(j)$]` 也不存在。
- 对于都比 `len` 小的所有非负整数 j 和 k ，如果 `SortCompare(j,k) < 0` (见下面的 `SortCompare`)，则 $\pi(j) < \pi(k)$ 。

这里的符号 `old[j]` 用来指：假定在执行这个函数之前以 j 作为参数调用 `obj` 的 `[[Get]]` 内部方法的结果，符号 `new[j]` 用来指：假定在执行这个函数后以 j 作为参数调用 `obj` 的 `[[Get]]` 内部方法的结果。

如果对于集合 S 里的任何值 a, b, c (可以是相同值)，都满足以下所有条件，那么函数 `comparefn` 是在集合 S 上保持一致的比较函数 (以下，符号 $a <_{CF} b$ 表示 `comparefn(a,b) < 0`; 符号 $a =_{CF} b$ 表示 `comparefn(a,b) = 0` (不论正负); 符号 $a >_{CF} b$ 表示 `comparefn(a,b) > 0`) :

- 当用指定值 a 和 b 作为两个参数调用 `comparefn(a,b)`，总是返回相同值 v 。此外 `Type(v)` 是 `Number`，并且 v 不是 `NaN`。注意，这意味着对于给定的 a 和 b ， $a <_{CF} b$, $a =_{CF} b$, and $a >_{CF} b$ 中正好有一个是真。
- 调用 `comparefn(a,b)` 不改变 `this` 对象。
- $a =_{CF} a$ (自反性)
- 如果 $a =_{CF} b$ ，则 $b =_{CF} a$ (对称性)
- 如果 $a =_{CF} b$ 并且 $b =_{CF} c$ ，则 $a =_{CF} c$ ($=_{CF}$ 传递)
- 如果 $a <_{CF} b$ 并且 $b <_{CF} c$ ，则 $a <_{CF} c$ ($<_{CF}$ 传递)
- 如果 $a >_{CF} b$ 并且 $b >_{CF} c$ ，则 $a >_{CF} c$ ($>_{CF}$ 传递)

这些条件是确保 `comparefn` 划分集合 S 为等价类并且是完全排序等价类的充分必要条件。

当用两个参数 j 和 k 调用抽象操作 `SortCompare`，采用如下步骤：

1. 令 jString 为 ToString(j).
2. 令 kString 为 ToString(k).
3. 令 hasj 为 以 jString 作为参数调用 obj 的 [[HasProperty]] 内部方法的结果。
4. 令 hask 为 以 kString 作为参数调用 obj 的 [[HasProperty]] 内部方法的结果。
5. 如果 hasj 和 hask 都是 false, 则返回 +0.
6. 如果 hasj 是 false, 则返回 1.
7. 如果 hask 是 false, 则返回 -1.
8. 令 x 为 以 jString 作为参数调用 obj 的 [[Get]] 内部方法的结果。
9. 令 y 为 以 kString 作为参数调用 obj 的 [[Get]] 内部方法的结果。
10. 如果 x 和 y 都是 undefined, 返回 +0.
11. 如果 x 是 undefined, 返回 1.
12. 如果 y 是 undefined, 返回 -1.
13. 如果 参数 comparefn 不是 undefined, 则
 1. 如果 IsCallable(comparefn) 是 false, 抛出一个 TypeError 异常 .
 2. 传入 undefined 作为 this 值, 以 x 和 y 作为参数调用 comparefn 的 [[Call]] 内部方法, 返回结果。
14. 令 xString 为 ToString(x).
15. 令 yString 为 ToString(y).
16. 如果 xString < yString, 返回 -1.
17. 如果 xString > yString, 返回 1.
18. 返回 +0.

因为不存在的属性值总是比 undefined 属性值大, 并且 undefined 属性值总是比任何其他值大, 所以 undefined 属性值总是排在结果的末尾, 后面跟着不存在属性值。

sort 函数被有意设计成通用的; 它的 this 值并非必须是数组对象。因此, 它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 sort 函数是依赖于实现的。

Array.prototype.splice (start, deleteCount [, item1 [, item2 [, ...]]])

当以两个或更多参数 start, deleteCount 和 (可选的) item1, item2, 等等, 调用 splice 方法, 从数组索引 start 开始的 deleteCount 个数组元素会被替换为参数 item1, item2, 等等。返回一个包含参数元素 (如果有) 的数组。采用以下步骤:

1. 令 O 为 以 this 值作为参数调用 ToObject 的结果 .
2. 令 A 为 仿佛用表达式 new Array() 创建的新数组, 这里的 Array 是标准内置构造器名。

3. 令 lenVal 为 以 "length" 作为参数调用 O 的 `[[Get]]` 内部方法的结果 .
4. 令 len 为 `ToUint32(lenVal)`.
5. 令 relativeStart 为 `ToInteger(start)`.
6. 如果 relativeStart 为负 , 令 actualStart 为 `max((len + relativeStart),0)`;
否则令 actualStart 为 `min(relativeStart, len)`.
7. 令 actualDeleteCount 为 `min(max(ToInteger(deleteCount),0),len – actualStart)`.
8. 令 k 为 0.
9. 只要 $k < \text{actualDeleteCount}$ 就重复
 1. 令 from 为 `ToString(actualStart+k)`.
 2. 令 fromPresent 为 以 from 作为参数调用 O 的 `[[HasProperty]]` 内部方法的结果。
 3. 如果 fromPresent 是 true, 则
 - i. 令 fromValue 为 以 from 作为参数调用 O 的 `[[Get]]` 内部方法的结果。
 - ii. 以 `ToString(k)`, 属性描述符 `{[[Value]]: fromValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, 和 false 作为参数调用 A 的 `[[DefineOwnProperty]]` 内部方法。
 4. k 递增 1.
10. 令 items 为一个内部列表, 它的元素是实际参数列表中 item1 开始的参数 (从左到右的顺序)。如果没传入这些项目, 则列表是空的。
11. 令 itemCount 为 items 的元素个数 .
12. 如果 $\text{itemCount} < \text{actualDeleteCount}$, 则
 1. 令 k 为 actualStart.
 2. 只要 $k < (\text{len} - \text{actualDeleteCount})$ 就重复
 - i. 令 from 为 `ToString(k+actualDeleteCount)`.
 - ii. 令 to 为 `ToString(k+itemCount)`.
 - iii. 令 fromPresent 为 以 from 作为参数调用 O 的 `[[HasProperty]]` 内部方法的结果 .
 - iv. 如果 fromPresent 是 true, 则
 1. 令 fromValue 为 以 from 作为参数调用 O 的 `[[Get]]` 内部方法的结果 .
 2. 以 to, fromValue, 和 true 作为参数调用 O 的 `[[Put]]` 内部方法 .
 - v. 否则 , fromPresent 是 false
 1. 以 to 和 true 作为参数调用 O 的 `[[Delete]]` 内部方法 .
 - vi. k 递增 1.
 3. 令 k 为 len.
 4. 只要 $k > (\text{len} - \text{actualDeleteCount} + \text{itemCount})$ 就重复
 - i. 以 `ToString(k-1)` 和 true 作为参数调用 O 的 `[[Delete]]` 内部方法 .
 - ii. k 递减 1.
13. 否则如果 $\text{itemCount} > \text{actualDeleteCount}$, 则
 1. 令 k 为 `(len – actualDeleteCount)`.
 2. 只要 $k > \text{actualStart}$ 就重复
 - i. 令 from 为 `ToString(k + actualDeleteCount – 1)`.
 - ii. 令 to 为 `ToString(k + itemCount – 1)`

- iii. 令 fromPresent 为 以 from 作为参数调用 O 的 `[[HasProperty]]` 内部方法的结果 .
- iv. 如果 fromPresent 是 true, 则
 - 1. 令 fromValue 为 以 from 作为参数调用 O 的 `[[Get]]` 内部方法的结果 .
 - 2. 以 to, fromValue, 和 true 作为参数调用 O 的 `[[Put]]` 内部方法 .
- v. 否则 , fromPresent 是 false
 - 1. 以 to 和 true 作为参数调用 O 的 `[[Delete]]` 内部方法 .
- vi. k 递减 1.
- 14. 令 k 为 actualStart.
- 15. 只要 items 不是空 就重复
 - 1. 删除 items 的第一个元素 , 并令 E 为这个元素值 .
 - 2. 以 ToString(k), E, 和 true 作为参数调用 O 的 `[[Put]]` 内部方法 .
 - 3. k 递增 1.
- 16. 以 "length", (len – actualDeleteCount + itemCount), 和 true 作为参数调用 O 的 `[[Put]]` 内部方法 .
- 17. 返回 A.

splice 方法的 length 属性是 2。

splice 函数被有意设计成通用的；它的 this 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 splice 函数是依赖于实现的。

Array.prototype.unshift ([item1 [, item2 [, ...]]])

将参数们插入到数组的开始位置，它们在数组中的顺序与它们出现在参数列表中的顺序相同。

当以零或更多个参数 item1,item2, 等等，调用 unshift 方法，采用如下步骤：

- 1. 令 O 为 以 this 值作为参数调用 ToObject 的结果 .
- 2. 令 lenVal 为 以 "length" 作为参数调用 O 的 `[[Get]]` 内部方法的结果 .
- 3. 令 len 为 ToUint32(lenVal).
- 4. 令 argCount 为 实际参数的个数 .
- 5. 令 k 为 len.
- 6. 只要 k > 0, 就重复
 - 1. 令 from 为 ToString(k-1).
 - 2. 令 to 为 ToString(k+argCount -1).
 - 3. 令 fromPresent 为 以 from 作为参数调用 O 的 `[[HasProperty]]` 内部方法的结果 .
 - 4. 如果 fromPresent 是 true, 则
 - i. 令 fromValue 为 以 from 作为参数调用 O 的 `[[Get]]` 内部方法的结果 .
 - ii. 以 to, fromValue, 和 true 作为参数调用 O 的 `[[Put]]` 内部方法 .
 - 5. 否则 , fromPresent 是 false

- i. 以 `to` 和 `true` 作为参数调用 `O` 的 `[[Delete]]` 内部方法。
6. `k` 递减 1.
7. 令 `j` 为 0.
8. 令 `items` 为一个内部列表, 它的元素是调用这个函数时传入的实际参数(从左到右的顺序)。
9. 只要 `items` 不是空, 就重复
 1. 删除 `items` 的第一个元素, 并令 `E` 为这个元素值。
 2. 以 `ToString(j)`, `E`, 和 `true` 作为参数调用 `O` 的 `[[Put]]` 内部方法。
 3. `j` 递增 1.
10. 以 `"length"`, `len+argCount`, 和 `true` 作为参数调用 `O` 的 `[[Put]]` 内部方法。
11. 返回 `len+argCount`.

`unshift` 方法的 `length` 属性是 1。

`unshift` 函数被有意设计成通用的; 它的 `this` 值并非必须是数组对象。因此, 它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `unshift` 函数是依赖于实现的。

Array.prototype.indexOf (searchElement [, fromIndex])

`indexOf` 按照索引的升序比较 `searchElement` 和数组里的元素们, 它使用内部的严格相等比较算法 (11.9.6), 如果找到一个或更多这样的位置, 返回这些位置中第一个索引; 否则返回 -1。

可选的第二个参数 `fromIndex` 默认是 0 (即搜索整个数组)。如果它大于或等于数组长度, 返回 -1, 即不会搜索数组。如果它是负的, 就把它当作从数组末尾到计算后的 `fromIndex` 的偏移量。如果计算后的索引小于 0, 就搜索整个数组。

当用一个或两个参数调用 `indexOf` 方法, 采用以下步骤:

1. 令 `O` 为以 `this` 值作为参数调用 `ToObject` 的结果。
2. 令 `lenValue` 为以 `"length"` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
3. 令 `len` 为 `ToUint32(lenValue)`.
4. 如果 `len` 是 0, 返回 -1.
5. 如果传入了参数 `fromIndex`, 则令 `n` 为 `ToInteger(fromIndex)`; 否则令 `n` 为 0.
6. 如果 `n ≥ len`, 返回 -1.
7. 如果 `n ≥ 0`, 则
 1. 令 `k` 为 `n`.
8. 否则, `n < 0`
 1. 令 `k` 为 `len - abs(n)`.

2. 如果 k 小于 0, 则令 k 为 0.
9. 只要 $k < \text{len}$, 就重复
 1. 令 $k\text{Present}$ 为 以 $\text{ToString}(k)$ 为参数调用 O 的 $[[\text{HasProperty}]]$ 内部方法的结果 .
 2. 如果 $k\text{Present}$ 是 `true`, 则
 - i. 令 elementK 为 以 $\text{ToString}(k)$ 为参数调用 O 的 $[[\text{Get}]]$ 内部方法的结果 .
 - ii. 令 same 为 对 searchElement 和 elementK 执行严格相等比较算法的结果 .
 - iii. 如果 same 是 `true`, 返回 k .
 3. k 递增 1.
10. 返回 -1.

`indexOf` 方法的 `length` 属性是 1。

`indexOf` 函数被有意设计成通用的；它的 `this` 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `indexOf` 函数是依赖于实现的。

Array.prototype.lastIndexOf (searchElement [, fromIndex])

`lastIndexOf` 按照索引的降序比较 `searchElement` 和数组里的元素们，它使用内部的严格相等比较算法 (11.9.6)，如果找到一个或更多这样的位置，返回这些位置中最后一个索引；否则返回 -1。

可选的第二个参数 `fromIndex` 默认是数组的长度减一（即搜索整个数组）。如果它大于或等于数组长度，将会搜索整个数组。如果它是负的，就把它当作从数组末尾到计算后的 `fromIndex` 的偏移量。如果计算后的索引小于 0，返回 -1。

当用一个或两个参数调用 `lastIndexOf` 方法，采用如下步骤：

1. 令 O 为 以 `this` 值作为参数调用 `ToObject` 的结果 .
2. 令 lenValue 为 以 `"length"` 作为参数调用 O 的 $[[\text{Get}]]$ 内部方法的结果 .
3. 令 len 为 `ToUint32(lenValue)`.
4. 如果 len 是 0, 返回 -1.
5. 如果 传入了参数 `fromIndex`, 则令 n 为 `ToInteger(fromIndex)`; 否则令 n 为 len .
6. 如果 $n \geq 0$, 则令 k 为 $\min(n, \text{len} - 1)$.
7. 否则, $n < 0$
 1. 令 k 为 $\text{len} - \text{abs}(n)$.
8. 只要 $k \geq 0$ 就重复
 1. 令 $k\text{Present}$ 为 以 $\text{ToString}(k)$ 作为参数调用 O 的 $[[\text{HasProperty}]]$ 内部方法的结果 .

2. 如果 `kPresent` 是 `true`, 则
 - i. 令 `elementK` 为 以 `ToString(k)` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
 - ii. 令 `same` 为 对 `searchElement` 和 `elementK` 执行严格相等比较算法的结果。
 - iii. 如果 `same` 是 `true`, 返回 `k`.
3. `k` 递减 1.
9. 返回 -1.

`lastIndexOf` 方法的 `length` 属性是 1。

`lastIndexOf` 函数被有意设计成通用的；它的 `this` 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `lastIndexOf` 函数是依赖于实现的。

Array.prototype.every (callbackfn [, thisArg])

`callbackfn` 应该是个函数，它接受三个参数并返回一个可转换为布尔值 `true` 和 `false` 的值。`every` 按照索引的升序，对数组里存在的每个元素调用一次 `callbackfn`，直到他找到一个使 `callbackfn` 返回 `false` 的元素。如果找到这样的元素，`every` 马上返回 `false`，否则如果对所有元素 `callbackfn` 都返回 `true`，`every` 将返回 `true`。`callbackfn` 只被数组里实际存在的元素调用；它不会被缺少的元素调用。

如果提供了一个 `thisArg` 参数，它会被当作 `this` 值传给每个 `callbackfn` 调用。如果没提供它，用 `undefined` 替代。

调用 `callbackfn` 时将传入三个参数：元素的值，元素的索引，和遍历的对象。

对 `every` 的调用不直接更改对象，但是对 `callbackfn` 的调用可能更改对象。

`every` 处理的元素范围是在首次调用 `callbackfn` 之前设定的。在 `every` 调用开始后追加到数组里的元素们不会被 `callbackfn` 访问。如果更改以存在数组元素，`every` 访问这些元素时的值会传给 `callbackfn`；在 `every` 调用开始后删除的和之前被访问过的元素们是不访问的。`every` 的行为就像数学量词“所有（for all）”。特别的，对一个空数组，它返回 `true`。

当以一个或两个参数调用 `every` 方法，采用以下步骤：

1. 令 `O` 为 以 `this` 值作为参数调用 `ToObject` 的结果。
2. 令 `lenValue` 为 以 `"length"` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
3. 令 `len` 为 `ToUint32(lenValue)`.
4. 如果 `IsCallable(callbackfn)` 是 `false`，抛出一个 `TypeError` 异常。

5. 如果提供了 `thisArg`, 令 `T` 为 `thisArg`; 否则令 `T` 为 `undefined`.
6. 令 `k` 为 0.
7. 只要 `k < len` , 就重复
 1. 令 `Pk` 为 `ToString(k)`.
 2. 令 `kPresent` 为 以 `Pk` 作为参数调用 `O` 的 `[[HasProperty]]` 内部方法的结果 .
 3. 如果 `kPresent` 是 `true`, 则
 - i. 令 `kValue` 为 以 `Pk` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果 .
 - ii. 令 `testResult` 为 以 `T` 作为 `this` 值以包含 `kValue`, `k`, 和 `O` 的参数列表调用 `callbackfn` 的 `[[Call]]` 内部方法的结果 .
 - iii. 如果 `ToBoolean(testResult)` 是 `false`, 返回 `false`.
 4. `k` 递增 1.
8. 返回 `true`.

`every` 方法的 `length` 属性是 1。

`every` 函数被有意设计成通用的；它的 `this` 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `every` 函数是依赖于实现的。

Array.prototype.some (callbackfn [, thisArg])

`callbackfn` 应该是个函数，它接受三个参数并返回一个可转换为布尔值 `true` 和 `false` 的值。`some` 按照索引的升序，对数组里存在的每个元素调用一次 `callbackfn`，直到他找到一个使 `callbackfn` 返回 `true` 的元素。如果找到这样的元素，`some` 马上返回 `true`，否则，`some` 返回 `false`。`callbackfn` 只被实际存在的数组元素调用；它不会被缺少的数组元素调用。

如果提供了一个 `thisArg` 参数，它会被当作 `this` 值传给每个 `callbackfn` 调用。如果没提供它，用 `undefined` 替代。

调用 `callbackfn` 时将传入三个参数：元素的值，元素的索引，和遍历的对象。

对 `some` 的调用不直接更改对象，但是对 `callbackfn` 的调用可能更改对象。

`some` 处理的元素范围是在首次调用 `callbackfn` 之前设定的。在 `some` 调用开始后追加到数组里的元素们不会被 `callbackfn` 访问。如果更改以存在数组元素，`some` 访问这些元素时的值会传给 `callbackfn`；在 `some` 调用开始后删除的和之前被访问过的元素们是不访问的。`some` 的行为就像数学量词“存在 (exists)”。特别的，对一个空数组，它返回 `false`。

当以一个或两个参数调用 `some` 方法，采用以下步骤：

1. 令 `O` 为 以 `this` 值作为参数调用 `ToObject` 的结果 .

2. 令 lenValue 为 以 "length" 作为参数调用 O 的 `[[Get]]` 内部方法的结果。
3. 令 len 为 `ToUint32(lenValue)`.
4. 如果 `IsCallable(callbackfn)` 是 `false`, 抛出一个 `TypeError` 异常。
5. 如果提供了 `thisArg`, 令 T 为 `thisArg`; 否则令 T 为 `undefined`.
6. 令 k 为 0.
7. 只要 `k < len`, 就重复
 1. 令 Pk 为 `Tostring(k)`.
 2. 令 kPresent 为 以 Pk 作为参数调用 O 的 `[[HasProperty]]` 内部方法的结果。
 3. 如果 kPresent 是 `true`, 则
 - i. 令 kValue 为 以 Pk 作为参数调用 O 的 `[[Get]]` 内部方法的结果。
 - ii. 令 testResult 为 以 T 作为 `this` 值以包含 kValue, k, 和 O 的参数列表调用 `callbackfn` 的 `[[Call]]` 内部方法的结果。
 - iii. 如果 `ToBoolean(testResult)` 是 `true`, 返回 `true`.
 4. k 递增 1.
8. 返回 `false`.

`some` 方法的 `length` 属性是 1。

`some` 函数被有意设计成通用的；它的 `this` 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `some` 函数是依赖于实现的。

Array.prototype.forEach (callbackfn [, thisArg])

`callbackfn` 应该是个函数，它接受三个参数。`forEach` 按照索引的升序，对数组里存在的每个元素调用一次 `callbackfn`。`callbackfn` 只被实际存在的数组元素调用；它不会被缺少的数组元素调用。

如果提供了一个 `thisArg` 参数，它会被当作 `this` 值传给每个 `callbackfn` 调用。如果没提供它，用 `undefined` 替代。

调用 `callbackfn` 时将传入三个参数：元素的值，元素的索引，和遍历的对象。

对 `forEach` 的调用不直接更改对象，但是对 `callbackfn` 的调用可能更改对象。

`forEach` 处理的元素范围是在首次调用 `callbackfn` 之前设定的。在 `forEach` 调用开始后追加到数组里的元素们不会被 `callbackfn` 访问。如果更改以存在数组元素，`forEach` 访问这些元素时的值会传给 `callbackfn`；在 `forEach` 调用开始后删除的和之前被访问过的元素们是不访问的。

当以一个或两个参数调用 `forEach` 方法，采用以下步骤：

1. 令 `O` 为 以 `this` 值作为参数调用 `ToObject` 的结果。
2. 令 `lenValue` 为 以 `"length"` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
3. 令 `len` 为 `ToUint32(lenValue)`。
4. 如果 `IsCallable(callbackfn)` 是 `false`, 抛出一个 `TypeError` 异常。
5. 如果提供了 `thisArg`, 令 `T` 为 `thisArg`; 否则令 `T` 为 `undefined`。
6. 令 `k` 为 0。
7. 只要 `k < len`, 就重复
 1. 令 `Pk` 为 `ToString(k)`。
 2. 令 `kPresent` 为 以 `Pk` 作为参数调用 `O` 的 `[[HasProperty]]` 内部方法的结果。
 3. 如果 `kPresent` 是 `true`, 则
 - i. 令 `kValue` 为 以 `Pk` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
 - ii. 以 `T` 作为 `this` 值以包含 `kValue`, `k`, 和 `O` 的参数列表调用 `callbackfn` 的 `[[Call]]` 内部方法。
 4. `k` 递增 1。
8. 返回 `undefined`。

`forEach` 方法的 `length` 属性是 1。

`forEach` 函数被有意设计成通用的；它的 `this` 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `forEach` 函数是依赖于实现的。

Array.prototype.map (callbackfn [, thisArg])

`callbackfn` 应该是个函数，它接受三个参数。`map` 按照索引的升序，对数组里存在的每个元素调用一次 `callbackfn`，并用结果构造一个新数组。`callbackfn` 只被实际存在的数组元素调用；它不会被缺少的数组元素调用。

如果提供了一个 `thisArg` 参数，它会被当作 `this` 值传给每个 `callbackfn` 调用。如果没提供它，用 `undefined` 替代。

调用 `callbackfn` 时将传入三个参数：元素的值，元素的索引，和遍历的对象。

对 `map` 的调用不直接更改对象，但是对 `callbackfn` 的调用可能更改对象。

`map` 处理的元素范围是在首次调用 `callbackfn` 之前设定的。在 `map` 调用开始后追加到数组里的元素们不会被 `callbackfn` 访问。如果更改以存在数组元素，`map` 访问这些元素时的值会传给 `callbackfn`；在 `map` 调用开始后删除的和之前被访问过的元素们是不访问的。

当以一个或两个参数调用 `map` 方法，采用以下步骤：

1. 令 `O` 为 以 `this` 值作为参数调用 `ToObject` 的结果。
2. 令 `lenValue` 为 以 `"length"` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
3. 令 `len` 为 `ToUint32(lenValue)`。
4. 如果 `IsCallable(callbackfn)` 是 `false`, 抛出一个 `TypeError` 异常。
5. 如果提供了 `thisArg`, 令 `T` 为 `thisArg`; 否则令 `T` 为 `undefined`。
6. 令 `A` 为 仿佛用 `new Array(len)` 创建的新数组, 这里的 `Array` 是标准内置构造器名, `len` 是 `len` 的值。
7. 令 `k` 为 0。
8. 只要 `k < len`, 就重复
 1. 令 `Pk` 为 `ToString(k)`。
 2. 令 `kPresent` 为 以 `Pk` 作为参数调用 `O` 的 `[[HasProperty]]` 内部方法的结果。
 3. 如果 `kPresent` 是 `true`, 则
 - i. 令 `kValue` 为 以 `Pk` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
 - ii. 令 `mappedValue` 为 以 `T` 作为 `this` 值以包含 `kValue`, `k`, 和 `O` 的参数列表调用 `callbackfn` 的 `[[Call]]` 内部方法的结果。
 - iii. 以 `Pk`, 属性描述符 `{[[Value]]: mappedValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, 和 `false` 作为参数调用 `A` 的 `[[DefineOwnProperty]]` 内部方法。
 4. `k` 递增 1。
9. 返回 `A`。

`map` 方法的 `length` 属性是 1。

`map` 函数被有意设计成通用的; 它的 `this` 值并非必须是数组对象。因此, 它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `map` 函数是依赖于实现的。

Array.prototype.filter (callbackfn [, thisArg])

`callbackfn` 应该是个函数, 它接受三个参数并返回一个可转换为布尔值 `true` 和 `false` 的值。`filter` 按照索引的升序, 对数组里存在的每个元素调用一次 `callbackfn`, 并用使 `callbackfn` 返回 `true` 的所有值构造一个新数组。`callbackfn` 只被实际存在的数组元素调用; 它不会被缺少的数组元素调用。

如果提供了一个 `thisArg` 参数, 它会被当作 `this` 值传给每个 `callbackfn` 调用。如果没提供它, 用 `undefined` 替代。

调用 `callbackfn` 时将传入三个参数: 元素的值, 元素的索引, 和遍历的对象。

对 `filter` 的调用不直接更改对象, 但是对 `callbackfn` 的调用可能更改对象。

filter 处理的元素范围是在首次调用 **callbackfn** 之前设定的。在 **filter** 调用开始后追加到数组里的元素们不会被 **callbackfn** 访问。如果更改以存在数组元素，**filter** 访问这些元素时的值会传给 **callbackfn**；在 **filter** 调用开始后删除的和之前被访问过的元素们是不访问的。

当以一个或两个参数调用 **filter** 方法，采用以下步骤：

1. 令 **O** 为 以 **this** 值作为参数调用 **ToObject** 的结果。
2. 令 **lenValue** 为 以 "length" 作为参数调用 **O** 的 **[[Get]]** 内部方法的结果。
3. 令 **len** 为 **ToUint32(lenValue)**。
4. 如果 **IsCallable(callbackfn)** 是 **false**，抛出一个 **TypeError** 异常。
5. 如果提供了 **thisArg**，令 **T** 为 **thisArg**；否则令 **T** 为 **undefined**。
6. 令 **A** 为 仿佛用 **new Array(len)** 创建的新数组，这里的 **Array** 是标准内置构造器名，**len** 是 **len** 的值。
7. 令 **k** 为 0。
8. 令 **to** 为 0。
9. 只要 **k < len**，就重复
 1. 令 **Pk** 为 **ToString(k)**。
 2. 令 **kPresent** 为 以 **Pk** 作为参数调用 **O** 的 **[[HasProperty]]** 内部方法的结果。
 3. 如果 **kPresent** 是 **true**，则
 - i. 令 **kValue** 为 以 **Pk** 作为参数调用 **O** 的 **[[Get]]** 内部方法的结果。
 - ii. 令 **selected** 为 以 **T** 作为 **this** 值以包含 **kValue**, **k**, 和 **O** 的参数列表调用 **callbackfn** 的 **[[Call]]** 内部方法的结果。
 - iii. 如果 **ToBoolean(selected)** 是 **true**，则
 1. 以 **ToString(to)**，属性描述符 **{[[Value]]: kValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}**，和 **false** 作为参数调用 **A** 的 **[[DefineOwnProperty]]** 内部方法。
 2. **to** 递增 1。
 4. **k** 递增 1。
10. 返回 **A**。

filter 方法的 **length** 属性是 1。

filter 函数被有意设计成通用的；它的 **this** 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 **filter** 函数是依赖于实现的。

Array.prototype.reduce (callbackfn [, initialValue])

callbackfn 应该是个函数，它需要四个参数。**reduce** 按照索引的升序，对数组里存在的每个元素，将 **callbackfn** 作为回调函数调用一次。

调用 `callbackfn` 时将传入四个参数：`previousValue` (`initialValue` 的值或上次调用 `callbackfn` 的返回值)，`currentValue` (当前元素值)，`currentIndex`，和遍历的对象。第一次调用回调函数时，`previousValue` 和 `currentValue` 的取值可以是下面两种情况之一。如果为 `reduce` 调用提供了一个 `initialValue`，则 `previousValue` 将等于 `initialValue` 并且 `currentValue` 将等于数组的首个元素值。如果没提供 `initialValue`，则 `previousValue` 将等于数组的首个元素值并且 `currentValue` 将等于数组的第二个元素值。如果数组里没有元素并且没有提供 `initialValue`，则抛出一个 `TypeError` 异常。

对 `reduce` 的调用不直接更改对象，但是对 `callbackfn` 的调用可能更改对象。

`reduce` 处理的元素范围是在首次调用 `callbackfn` 之前设定的。在 `reduce` 调用开始后追加到数组里的元素们不会被 `callbackfn` 访问。如果更改以存在数组元素，`reduce` 访问这些元素时的值会传给 `callbackfn`；在 `reduce` 调用开始后删除的和之前被访问过的元素们是不访问的。

当以一个或两个参数调用 `reduce` 方法，采用以下步骤：

1. 令 `O` 为以 `this` 值作为参数调用 `ToObject` 的结果。
2. 令 `lenValue` 为以 `"length"` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
3. 令 `len` 为 `ToUint32(lenValue)`。
4. 如果 `IsCallable(callbackfn)` 是 `false`，抛出一个 `TypeError` 异常。
5. 如果 `len` 是 0 并且 `initialValue` 不是 `present`，抛出一个 `TypeError` 异常。
6. 令 `k` 为 0。
7. 如果 `initialValue` 参数有传入值，则
 1. 设定 `accumulator` 为 `initialValue`。
8. 否则，`initialValue` 参数没有传入值
 1. 令 `kPresent` 为 `false`。
 2. 只要 `kPresent` 是 `false` 并且 `k < len`，就重复
 - i. 令 `Pk` 为 `Tostring(k)`。
 - ii. 令 `kPresent` 为以 `Pk` 作为参数调用 `O` 的 `[[HasProperty]]` 内部方法的结果。
 - iii. 如果 `kPresent` 是 `true`，则
 1. 令 `accumulator` 为以 `Pk` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
 - iv. `k` 递增 1。
 3. 如果 `kPresent` 是 `false`，抛出一个 `TypeError` 异常。
 9. 只要 `k < len`，就重复
 1. 令 `Pk` 为 `Tostring(k)`。
 2. 令 `kPresent` 为以 `Pk` 作为参数调用 `O` 的 `[[HasProperty]]` 内部方法的结果。
 3. 如果 `kPresent` 是 `true`，则
 - i. 令 `kValue` 为以 `Pk` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。

- ii. 令 accumulator 为以 undefined 作为 this 值并以包含 accumulator, kValue, k, 和 O 的参数列表调用 callbackfn 的 `[[Call]]` 内部方法的结果。
4. k 递增 1.
10. 返回 accumulator.

reduce 方法的 length 属性是 1。

reduce 函数被有意设计成通用的；它的 this 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 reduce 函数是依赖于实现的。

Array.prototype.reduceRight (callbackfn [, initialValue])

callbackfn 应该是个函数，它需要四个参数。reduceRight 按照索引的升序，对数组里存在的每个元素，将 callbackfn 作为回调函数调用一次。

调用 callbackfn 时将传入四个参数：previousValue (initialValue 的值或上次调用 callbackfn 的返回值)，currentValue (当前元素值)，currentIndex，和遍历的对象。第一次调用回调函数时，previousValue 和 currentValue 的取值可以是下面两种情况之一。如果为 reduceRight 调用提供了一个 initialValue，则 previousValue 将等于 initialValue 并且 currentValue 将等于数组的最后一个元素值。如果没提供 initialValue，则 previousValue 将等于数组的最后一个元素值并且 currentValue 将等于数组的倒数第二个元素值。如果数组里没有元素并且没有提供 initialValue，则抛出一个 TypeError 异常。

对 reduceRight 的调用不直接更改对象，但是对 callbackfn 的调用可能更改对象。

reduceRight 处理的元素范围是在首次调用 callbackfn 之前设定的。在 reduceRight 调用开始后追加到数组里的元素们不会被 callbackfn 访问。如果更改以存在数组元素，reduceRight 访问这些元素时的值会传给 callbackfn；在 reduceRight 调用开始后删除的和之前被访问过的元素们是不访问的。

当以一个或两个参数调用 reduceRight 方法，采用以下步骤：

1. 令 O 为以 this 值作为参数调用 ToObject 的结果。
2. 令 lenValue 为以 "length" 作为参数调用 O 的 `[[Get]]` 内部方法的结果。
3. 令 len 为 ToUint32(lenValue)。
4. 如果 IsCallable(callbackfn) 是 false，抛出一个 TypeError 异常。
5. 如果 len 是 0 并且 initialValue 不是 present，抛出一个 TypeError 异常。
6. 令 k 为 0。

7. 如果 `initialValue` 参数有传入值，则
 1. 设定 `accumulator` 为 `initialValue`.
8. 否则，`initialValue` 参数没有传入值
 1. 令 `kPresent` 为 `false`.
 2. 只要 `kPresent` 是 `false` 并且 $k \geq 0$ ，就重复
 - i. 令 `Pk` 为 `ToString(k)`.
 - ii. 令 `kPresent` 为以 `Pk` 作为参数调用 `O` 的 `[[HasProperty]]` 内部方法的结果。
 - iii. 如果 `kPresent` 是 `true`，则
 1. 令 `accumulator` 为以 `Pk` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
 - iv. `k` 递减 1.
 3. 如果 `kPresent` 是 `false`，抛出一个 `TypeError` 异常。
 9. 只要 $k \geq 0$ ，就重复
 1. 令 `Pk` 为 `ToString(k)`.
 2. 令 `kPresent` 为以 `Pk` 作为参数调用 `O` 的 `[[HasProperty]]` 内部方法的结果。
 3. 如果 `kPresent` 是 `true`，则
 - i. 令 `kValue` 为以 `Pk` 作为参数调用 `O` 的 `[[Get]]` 内部方法的结果。
 - ii. 令 `accumulator` 为以 `undefined` 作为 `this` 值并以包含 `accumulator`, `kValue`, `k`, 和 `O` 的参数列表调用 `callbackfn` 的 `[[Call]]` 内部方法的结果。
 4. `k` 递减 1.
10. 返回 `accumulator`.

`reduceRight` 方法的 `length` 属性是 1。

`reduceRight` 函数被有意设计成通用的；它的 `this` 值并非必须是数组对象。因此，它可以作为方法转移到其他类型的对象中。一个宿主对象是否可以正确应用这个 `reduceRight` 函数是依赖于实现的。

Array 实例的属性

`Array` 实例从数组原型对象继承属性，`Array` 实例的 `[[Class]]` 内部属性是 `"Array"`。`Array` 实例还有以下属性。

`[[DefineOwnProperty]] (P, Desc, Throw)`

数组对象使用一个，用在其他原生 ECMAScript 对象的 `[[DefineOwnProperty]]` 内部方法 (8.12.9) 的变化版。

设 `A` 为一个数组对象，`Desc` 为一个属性描述符，`Throw` 为一个布尔标示。

在以下算法中, 术语“拒绝”指代“如果 Throw 是 true, 则抛出 TypeError 异常, 否则返回 false。”

当用属性名 P, 属性描述 Desc, 布尔值 Throw 调用 A 的 [[DefineOwnProperty]] 内部方法, 采用以下步骤:

1. 令 oldLenDesc 为 以 "length" 作为参数调用 A 的 [[GetOwnProperty]] 内部方法的结果。结果绝不会是 undefined 或一个访问器描述符, 因为在创建数组时的 length 是一个不可删除或重新配置的数据属性。
2. 令 oldLen 为 oldLenDesc.[[Value]]。
3. 如果 P 是 "length", 则
 1. 如果 Desc 的 [[Value]] 字段不存在, 则
 - i. 以 "length", Desc, 和 Throw 作为参数在 A 上调用默认的 [[DefineOwnProperty]] 内部方法 (8.12.9), 返回结果。
 2. 令 newLenDesc 为 Desc 的一个拷贝。
 3. 令 newLen 为 ToUint32(Desc.[[Value]])。
 4. 如果 newLen 不等于 ToNumber(Desc.[[Value]]), 抛出一个 RangeError 异常。
 5. 设定 newLenDesc.[[Value]] 为 newLen。
 6. 如果 newLen ≥ oldLen, 则
 - i. 以 "length", newLenDesc, 和 Throw 作为参数在 A 上调用默认的 [[DefineOwnProperty]] 内部方法 (8.12.9), 返回结果。
 7. 如果 oldLenDesc.[[Writable]] 是 false, 拒绝
 8. 如果 newLenDesc.[[Writable]] 不存在或值是 true, 令 newWritable 为 true。
 9. 否则 ,
 - i. 因为它将使得无法删除任何元素, 所以需要延后设定 [[Writable]] 特性为 false。
 - ii. 令 newWritable 为 false。
 - iii. 设定 newLenDesc.[[Writable]] 为 true。
10. 令 succeeded 为 以 "length", newLenDesc, 和 Throw 作为参数在 A 上调用默认的 [[DefineOwnProperty]] 内部方法 (8.12.9) 的结果
11. 如果 succeeded 是 false, 返回 false..
12. 只要 newLen < oldLen, 就重复 ,
 - i. 设定 oldLen 为 oldLen - 1。
 - ii. 令 deleteSucceeded 为 以 ToString(oldLen) 和 false 作为参数调用 A 的 [[Delete]] 内部方法的结果。
- iii. 如果 deleteSucceeded 是 false, 则
 1. 设定 newLenDesc.[[Value]] 为 oldLen+1。
 2. 如果 newWritable 是 false, 设定 newLenDesc.[[Writable]] 为 false。
 3. 以 "length", newLenDesc, 和 false 为参数在 A 上调用默认的 [[DefineOwnProperty]] 内部方法 (8.12.9)。
 4. 拒绝。
13. 如果 newWritable 是 false, 则

- i. 以 "length", 属性描述符 `{[[Writable]]: false}`, 和 `false` 作为参数在 `A` 上调用 `[[DefineOwnProperty]]` 内部方法 (8.12.9). 这个调用始终返回 `true`.
14. 返回 `true`.
4. 否则如果 `P` 是一个数组索引 (15.4), 则
 1. 令 `index` 为 `ToUint32(P)`.
 2. 如果 `index ≥ oldLen` 并且 `oldLenDesc. [[Writable]]` 是 `false`, 拒绝 .
 3. 令 `succeeded` 为 以 `P`, `Desc`, 和 `false` 作为参数在 `A` 上调用默认的 `[[DefineOwnProperty]]` 内部方法 (8.12.9) 的结果 .
 4. 如果 `succeeded` 是 `false`, 拒绝 .
 5. 如果 `index ≥ oldLen`
- i. 设定 `oldLenDesc. [[Value]]` 为 `index + 1`.
- ii. 以 "length", `oldLenDesc`, 和 `false` 作为参数在在 `A` 上调用默认的 `[[DefineOwnProperty]]` 内部方法 (8.12.9). 这个调用始终返回 `true`.
6. 返回 `true`.
5. 以 `P`, `Desc`, 和 `Throw` 作为参数在在 `A` 上调用默认的 `[[DefineOwnProperty]]` 内部方法 (8.12.9), 返回结果 .

length

数组对象的 `length` 属性是个数据属性, 其值总是在数值上大于任何属性名是数组索引的可删除属性的属性名。

`length` 属性拥有的初始特性是 `{ [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false }`.

试图将一个数组对象的 `length` 属性设定为在数值上比 -- 数组中存在数组索引并且是不可参数属性中的最大数字属性名 -- 小或相等时, `length` 将设定为比那个最大数字属性名大一的数字子。见 15.4.5.1。

String 对象

作为函数调用 String 构造器

当将 `String` 作为函数调用, 而不是作为构造器, 它执行一个类型转换。

String ([value])

返回一个由 `ToString(value)` 计算出的字符串值 (不是 `String` 对象)。如果没有提供 `value`, 返回空字符串 `""`。

String 构造器

当 `String` 作为一个 `new` 表达式的一部分被调用，它是个构造器：它初始化新创建的对象。

`new String ([value])`

新构造对象的 `[[Prototype]]` 内部属性设定为标准内置的字符串原型对象，它是 `String.prototype` 的初始值 (15.5.3.1)。

新构造对象的 `[[Class]]` 内部属性设定为 `"String"`。

新构造对象的 `[[Extensible]]` 内部属性设定为 `true`。

新构造对象的 `[[PrimitiveValue]]` 内部属性设定为 `ToString(value)`，或如果没提供 `value` 则设定为空字符串。

`String` 构造器的属性

`String` 构造器的 `[[Prototype]]` 内部属性的值是标准内置的函数原型对象 (15.3.4)。

除了内部属性和 `length` 属性（值为 1）之外，`String` 构造器还有以下属性：

`String.prototype`

`String.prototype` 的初始值是标准内置的字符串原型对象 (15.5.4)。

这个属性有特性 `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`。

`String.fromCharCode ([char0 [, char1 [, ...]]])`

返回一个字符串值，包含的字符数与参数数目相同。每个参数指定返回字符串中的一个字符，也就是说第一个参数第一个字符，以此类推（从左到右）。一个参数转换为一个字符，通过先应用 `ToUint16` (9.7) 操作，再将返回的 16 位整数看作字符的代码单元值。如果没提供参数，返回空字符串。

`fromCharCode` 函数的 `length` 属性是 1。

字符串原型对象的属性

字符串原型对象本身是一个值为空字符串的 `String` 对象（它的 `[[Class]]` 是 `"String"`）。

字符串原型对象的 `[[Prototype]]` 内部属性值是标准内置的 `Object` 原型对象 (15.2.4)。

`String.prototype.constructor`

`String.prototype.constructor` 的初始值是内置 `String` 构造器。

`String.prototype.toString ()`

返回 `this` 字符串值。（注，对于一个 `String` 对象，`toString` 方法和 `valueOf` 方法返回相同值。）

`toString` 函数是非通用的；如果它的 `this` 值不是一个字符串或字符串对象，则抛出一个 `TypeError` 异常。因此它不能作为方法转移到其他类型对象上。

`String.prototype.valueOf ()`

返回 `this` 字符串值。

`valueOf` 函数是非通用的；如果它的 `this` 值不是一个字符串或字符串对象，则抛出一个 `TypeError` 异常。因此它不能作为方法转移到其他类型对象上。

`String.prototype.charAt (pos)`

将 `this` 对象转换为一个字符串，返回包含了这个字符串 `pos` 位置的字符的字符串。如果那个位置没有字符，返回空字符串。返回结果是个字符串值，不是字符串对象。

如果 `pos` 是一个数字类型的整数值，则 `x.charAt(pos)` 与 `x.substring(pos, pos+1)` 的结果相等。

当用一个参数 `pos` 调用 `charAt` 方法，采用以下步骤：

1. 以 `this` 值作为参数调用 `CheckObjectCoercible`。
2. 令 `S` 为以 `this` 值作为参数调用 `ToString` 的结果。
3. 令 `position` 为 `ToInteger(pos)`。
4. 令 `size` 为 `S` 的字符数。
5. 如果 `position < 0` 或 `position ≥ size`，返回空字符串。

6. 返回一个长度为 1 的字符串，它包含 S 中 position 位置的一个字符，在这里 S 中的第一个（最左边）字符被当作是在位置 0，下一个字符被当作是在位置 1，等等。

charAt 函数被有意设计成通用的；它不要求它的 this 值是字符串对象。因此，他可以当做方法转移到其他类型对象。

String.prototype.charCodeAt (pos)

将 this 对象转换为一个字符串，返回一个代表这个字符串 pos 位置字符的代码单元值的数字（小于 2^{16} 的非负整数）。如果那个位置没有字符，返回 NaN。

当用一个参数 pos 调用 charCodeAt 方法，采用以下步骤：

1. 以 this 值作为参数调用 CheckObjectCoercible。
2. 令 S 为以 this 值作为参数调用 ToString 的结果。
3. 令 position 为 ToInteger(pos)。
4. 令 size 为 S 的字符数。
5. 如果 position < 0 或 position ≥ size, 返回 NaN。
6. 返回一个数字类型值，值是字符串 S 中 position 位置字符的代码单元值。在这里 S 中的第一个（最左边）字符被当作是在位置 0，下一个字符被当作是在位置 1，等等。

charCodeAt 函数被有意设计成通用的；它不要求它的 this 值是字符串对象。因此，他可以当做方法转移到其他类型对象。

String.prototype.concat ([string1 [, string2 [, ...]]])

当用一个或更多参数 string1, string2, 等等，调用 concat 方法，它返回一个包含了 --this 对象（转换为一个字符串）中的字符们和后面跟着的每个 string1, string2, 等等，（每个参数都转换为字符串）里的字符们 -- 的字符串。返回结果是一个字符串值，不是一个字符串对象。采用以下步骤：

1. 以 this 值作为参数调用 CheckObjectCoercible。
2. 令 S 为以 this 值作为参数调用 ToString 的结果。
3. 令 args 为一个内部列表，它是传给这个函数的参数列表的拷贝。
4. 令 R 为 S。
5. 只要 args 不是空，就重复
 1. 删除 args 的第一个元素，并令 next 为这个元素。
 2. 令 R 为包含了 -- 之前的 R 值中的字符们和后面跟着的 ToString(next) 结果的字符们 -- 的字符串值。
6. 返回 R。

`concat` 方法的 `length` 属性是 1。

`concat` 函数被有意设计成通用的；它不要求它的 `this` 值是字符串对象。因此，他可以当做方法转移到其他类型对象。

String.prototype.indexOf (searchString, position)

将 `this` 对象转换为一个字符串，如果 `searchString` 在这个字符串里大于或等于 `position` 的位置中的一个或多个位置使它呈现为字符串的子串，那么返回这些位置中最小的索引；否则返回 -1。如果 `position` 是 `undefined`，就认为它是 0，以搜索整个字符串。

`indexOf` 需要两个参数 `searchString` 和 `position`，执行以下步骤：

1. 以 `this` 值作为参数调用 `CheckObjectCoercible`。
2. 令 `S` 为以 `this` 值作为参数调用 `ToString` 的结果。
3. 令 `searchStr` 为 `ToString(searchString)`。
4. 令 `pos` 为 `ToInteger(position)`。（如果 `position` 是 `undefined`，此步骤产生 0）。
5. 令 `len` 为 `S` 的字符数。
6. 令 `start` 为 `min(max(pos, 0), len)`。
7. 令 `searchLen` 为 `searchStr` 的字符数。
8. 返回一个不小于 `start` 的可能的最小值整数 `k`，使得 `k+searchLen` 不大于 `len`，并且对所有小于 `searchLen` 的非负数整数 `j`，`S` 的 `k+j` 位置字符和 `searchStr` 的 `j` 位置字符相同；但如果没有这样的整数 `k`，则返回 -1。

`indexOf` 的 `length` 属性是 1。

`indexOf` 函数被有意设计成通用的；它不要求它的 `this` 值是字符串对象。因此，他可以当做方法转移到其他类型对象。

String.prototype.lastIndexOf (searchString, position)

将 `this` 对象转换为一个字符串，如果 `searchString` 在这个字符串里小于或等于 `position` 的位置中的一个或多个位置使它呈现为字符串的子串，那么返回这些位置中最大的索引；否则返回 -1。如果 `position` 是 `undefined`，就认为它是字符串值的长度，以搜索整个字符串。

`lastIndexOf` 需要两个参数 `searchString` 和 `position`，执行以下步骤：

1. 以 `this` 值作为参数调用 `CheckObjectCoercible`。
2. 令 `S` 为以 `this` 值作为参数调用 `ToString` 的结果。
3. 令 `searchStr` 为 `ToString(searchString)`。

4. 令 numPos 为 `Tonumber(position)`. (如果 position 是 `undefined`, 此步骤产生 `NaN`).
5. 如果 numPos 是 `NaN`, 令 pos 为 $+\infty$; 否则, 令 pos 为 `ToInteger(numPos)`.
6. 令 len 为 S 的字符数.
7. 令 start 为 `min(max(pos, 0), len)`.
8. 令 searchLen 为 `SearchStr` 的字符数.
9. 返回 一个不大于 start 的可能的最大值整数 k, 使得 `k+searchLen` 不大于 len, 并且对所有小于 searchLen 的非负数整数 j, S 的 `k+j` 位置字符和 `searchStr` 的 j 位置字符相同; 但如果没有这样的整数 k, 则返回 -1.

`lastIndexOf` 的 `length` 属性是 1。

`lastIndexOf` 函数被有意设计成通用的; 它不要求它的 `this` 值是字符串对象。因此, 他可以当做方法转移到其他类型对象。

String.prototype.localeCompare (that)

当以一个参数 `that` 来调用 `localeCompare` 方法, 它返回一个非 `NaN` 数字值, 这个数字值反应了对 `this` 值 (转换为字符串) 和 `that` 值 (转换为字符串) 进行语言环境敏感的字符串比较的结果。两个字符串 S 和 That 用实现定义的一种方式进行比较。比较结果是为了按照系统默认语言环境指定的排列顺序来排列字符串, 根据按照排列顺序 S 是在 That 前面, 相同, 还是 S 在 That 后面, 结果分别是负数, 零, 正数。

在执行比较之前执行以下步骤以预备好字符串:

1. 以 `this` 值作为参数调用 `CheckObjectCoercible`。
2. 令 S 为以 `this` 值作为参数调用 `ToString` 的结果。
3. 令 That 为 `ToString(that)`。

如果将 `localeCompare` 方法看做是将 `this` 和 `that` 作为两个参数的函数, 那么它是在所有字符串集合上的保持一致的比较函数 (在 15.4.4.11 定义)。

实际返回值是实现定义的, 允许实现者们在返回值里编码附加信息。但是函数需要定义一个在所有字符串上的总的顺序, 并且, 当比较的字符串们被认为是 `Unicode` 标准定义的标准等价, 则返回 0。

如果宿主环境没有在所有字符串上语言敏感的比较, 此函数可执行按位比较。

`localeCompare` 方法自身不适合直接作为 `Array.prototype.sort` 的参数, 因为后者需要的是两个参数的函数。

这个函数的目的是在宿主环境中任何依靠语言敏感的比较方式都可用于 ECMAScript 环境，并根据宿主环境当前语言环境设置的规则进行比较。强烈建议这个函数将 -- 根据 Unicode 标准的标准等价的 -- 字符串当做是相同的也就是说，要比较的字符串仿佛是都先被转换为正常化形式 C 或正常化形式 D 了）。还建议这个函数不履行 Unicode 相容等价或分解。

本标准的未来版本可能会使用这个函数的第二个参数；建议实现不将这个参数位用作其他用途。

localeCompare 函数被有意设计成通用的；它不要求它的 this 值是字符串对象。因此，他可以当做方法转移到其他类型对象。

String.prototype.match (regexp)

当以 regexp 作为参数调用 match 方法，采用以下步骤：

1. 以 this 值作为参数调用 CheckObjectCoercible。
2. 令 S 为以 this 值作为参数调用 ToString 的结果。
3. 如果 Type(regexp) 是 Object 并且 regexp 的 [[Class]] 内部属性的值是 "RegExp", 则令 rx 为 regexp;
4. 否则，令 rx 为 仿佛是用表达式 new RegExp(regexp) 创建的新正则对象，这里的 RegExp 是标准内置构造器名。
5. 令 global 为以 "global" 为参数调用 rx 的 [[Get]] 内部方法的结果。
6. 令 exec 为 标准内置函数 RegExp.prototype.exec (见 15.10.6.2)
7. 如果 global 不是 true, 则
 1. 以 rx 作为 this 值，用包含 S 的参数列表调用 exec 的 [[Call]] 内部方法，返回结果。
8. 否则，global 是 true
 1. 以 "lastIndex" 和 0 作为参数调用 rx 的 [[Put]] 内部方法。
 2. 令 A 为 仿佛是用表达式 new Array() 创建的新数组，这里的 Array 是标准内置构造器名。
 3. 令 previousLastIndex 为 0.
 4. 令 n 为 0.
 5. 令 lastMatch 为 true.
 6. 只要 lastMatch 是 true, 就重复
 - i. 令 result 为以 rx 作为 this 值，用包含 S 的参数列表调用 exec 的 [[Call]] 内部方法的结果。
 - ii. 如果 result 是 null, 则设定 lastMatch 为 false.
 - iii. 否则，result 不是 null
 1. 令 thisIndex 为以 "lastIndex" 为参数调用 rx 的 [[Get]] 内部方法的结果。
 2. 如果 thisIndex = previousLastIndex 则
 1. 以 "lastIndex" 和 thisIndex+1 为参数调用 rx 的 [[Put]] 内部方法。
 2. 设定 previousLastIndex 为 thisIndex+1.

3. 否则，设定 `previousLastIndex` 为 `thisIndex`。
4. 令 `matchStr` 为以 0 为参数调用 `result` 的 `[[Get]]` 内部方法的结果。
5. 以 `ToString(n)`，属性描述符 `{[[Value]]: matchStr, [[Writable]]: true, [[Enumerable]]: true, [[configurable]]: true}`，和 `false` 作为参数调用 `A` 的 `[[DefineOwnProperty]]` 内部方法。
6. `n` 递增 1。
7. 如果 `n = 0`，则返回 `null`。
8. 返回 `A`。

`match` 函数被有意设计成通用的；它不要求它的 `this` 值是字符串对象。因此，他可以当做方法转移到其他类型对象。

String.prototype.replace (searchValue, replaceValue)

首先根据以下步骤设定 `string`：

1. 以 `this` 值作为参数调用 `CheckObjectCoercible`。
2. 令 `string` 为以 `this` 值作为为参数调用 `ToString` 的结果。

如果 `searchValue` 是一个正则表达式（`[[Class]]` 内部属性是 `"RegExp"` 的对象），按照如下执行：如果 `searchValue.global` 是 `false`，则搜索 `string`，找出匹配正则表达式 `searchValue` 的第一个子字符串。如果 `searchValue.global` 是 `true`，则搜索 `string`，找出匹配正则表达式 `searchValue` 的所有子字符串。搜索的做法与 `String.prototype.match` 相同，包括对 `searchValue.lastIndex` 的更新。令 `m` 为 `searchValue` 的左捕获括号的个数（使用 15.10.2.1 指定的 `NcapturingParens`）。

如果 `searchValue` 不是正则表达式，令 `searchString` 为 `ToString(searchValue)`，并搜索 `string`，找出第一个出现的 `searchString` 的子字符串。令 `m` 为 0。

如果 `replaceValue` 是函数，则对每个匹配的子字符串，以 `m + 3` 个参数调用这个函数。第一个参数是匹配的子字符串。如果 `searchValue` 是正则表达式，接下来 `m` 个参数是 `MatchResult`（见 15.10.2.1）里的所有捕获值。第 `m + 2` 个参数是发生的匹配在 `string` 里的偏移量，第 `m + 3` 个参数是 `string`。结果是将输入的原字符串里的每个匹配子字符串替换为相应函数调用的返回值（必要的情况下转换为字符串）得到的字符串。

否则，令 `newstring` 表示 `replaceValue` 转换为字符串的结果。结果是将输入的原字符串里的每个匹配子字符串替换为 -- 将 `newstring` 里的字符替换为表 22 指定的替代文本得到的字符串 -- 得到的字符串。替换这些 `$` 是由左到右进行的，并且一旦执行了这样的替换，新替换的文本不受进一步替换。例如，`"$1,$2".replace(/(\\$\\d)/g, "$$1-$1$2")` 返回 `"$1-$11,$1-$22"`。`newstring` 里的一个 `$`，如果不符合以下任何格式，就保持原状。

替代文本符号替换

字符编码值	表示
字符	替代文本
\$\$	\$
\$&	匹配到的子字符串
\$(译注: '\u0060')	string 中匹配到的子字符串之前部分。
\$(译注: '\u0027')	string 中匹配到的子字符串之后部分。
\$n	第 n 个捕获结果, n 是范围在 1 到 9 的单个数字, 并且紧接着 \$n 后面的不是十进制数字。如果 $n \leq m$ 且第 n 个捕获结果是 undefined, 就用空字符串代替。如果 $n > m$, 结果是实现定义的。
\$nn	第 nn 个捕获结果, nn 是范围在 01 到 99 的十进制两位数。如果 $nn \leq m$ 且第 nn 个捕获结果是 undefined, 就用空字符串代替。如果 $nn > m$, 结果是实现定义的。

replace 函数被有意设计成通用的; 它不要求它的 **this** 值是字符串对象。因此, 他可以当做方法转移到其他类型对象。

String.prototype.search (regexp)

当用参数 **regexp** 调用 **search** 方法, 采用以下步骤:

1. 以 **this** 值作为参数调用 **CheckObjectCoercible**。
2. 令 **string** 为以 **this** 值作为参数调用 **ToString** 的结果。
3. 如果 **Type(regexp)** 是 **Object** 且 **regexp** 的 **[[Class]]** 内部属性的值是 **"RegExp"**, 则令 **rx** 为 **regexp**;
4. 否则, 令 **rx** 为仿佛是用表达式 **new RegExp(regexp)** 创建的新正则对象, 这里的 **RegExp** 是标准内置构造器名。
5. 从 **string** 开始位置搜索正则表达式模式 **rx** 的匹配。如果找到匹配, 令 **result** 为匹配在 **string** 里的偏移量; 如果没有找到匹配, 令 **result** 为 **-1**。执行搜索时 **regexp** 的 **lastIndex** 和 **global** 属性是被忽略的。**regexp** 的 **lastIndex** 属性保持不变。
6. 返回 **result**。

search 函数被有意设计成通用的; 它不要求它的 **this** 值是字符串对象。因此, 他可以当做方法转移到其他类型对象。

String.prototype.slice (start, end)

slice 方法需要两个参数 **start** 和 **end**, 将 **this** 对象转换为一个字符串, 返回这个字符串中从 **start** 位置的字符到 (但不包含) **end** 位置的字符的一个子字

字符串（或如果 `end` 是 `undefined`，就直接到字符串尾部）。用 `sourceLength` 表示字符串长度，如果 `start` 是负数，就把它看做是 `sourceLength+start`；如果 `end` 是负数，就把它看做是 `sourceLength+end`。返回结果是一个字符串值，不是字符串对象。采用以下步骤：

1. 以 `this` 值作为参数调用 `CheckObjectCoercible`。
2. 令 `S` 为以 `this` 值作为参数调用 `ToString` 的结果。
3. 令 `len` 为 `S` 的字符数。
4. 令 `intStart` 为 `ToInteger(start)`。
5. 如果 `end` 是 `undefined`，令 `intEnd` 为 `len`；否则令 `intEnd` 为 `ToInteger(end)`。
6. 如果 `intStart` 是 `negative`，令 `from` 为 `max(len + intStart, 0)`；否则令 `from` 为 `min(intStart, len)`。
7. 如果 `intEnd` 是 `negative`，令 `to` 为 `max(len + intEnd, 0)`；否则令 `to` 为 `min(intEnd, len)`。
8. 令 `span` 为 `max(to - from, 0)`。
9. 返回一个包含 `--S` 中从 `form` 位置的字符开始的 `span` 个连续字符 `--` 的字符串。

`slice` 方法的 `length` 属性是 2。

`slice` 函数被有意设计成通用的；它不要求它的 `this` 值是字符串对象。因此，他可以当做方法转移到其他类型对象。

String.prototype.split (separator, limit)

将 `this` 字符串转换为一个字符串，返回一个数组对象，里面存储了这个字符串的子字符串。子字符串是从左到右搜索 `separator` 的匹配来确定的；这些匹配结果不成为返回数组的任何子字符串元素，但被用来分割字符串。`separator` 的值可以是一个任意长度的字符串，也可以是一个正则对象（即，一个 `[[Class]]` 内部属性为 `"RegExp"` 的对象；见 15.10）。

`separator` 值可以是一个空字符串，一个空正则表达式，或一个可匹配空字符串的正则表达式。这种情况下，`separator` 不匹配输入字符串开头和末尾的空子串，也不匹配分隔符的之前匹配结果末尾的空子串。（例如，如果 `separator` 是空字符串，要将字符串分割为单个字符；结果数组的长度等于字符串长度，且每个子串都包含一个字符。）如果 `separator` 是正则表达式，在 `this` 字符串的给定位置中只考虑首次匹配结果，即使如果在这个位置上回溯可产生一个非空的子串。（例如，`"ab".split(/a*?/)` 的执行结果是数组 `["a","b"]`，而 `"ab".split(/a*/)` 的执行结果是数组 `["","b"]`。）

如果 `this` 对象是（或转换成）空字符串，返回的结果取决于 `separator` 是否可匹配空字符串。如果可以，结果是不包含任何元素的数组。否则，结果是包含一个空字符串元素的数组。

如果 `separator` 是包含捕获括号的正则表达式，则对 `separator` 的每次匹配，捕获括号的结果（包括 `undefined`）都拼接为输出数组。例如，

```
"Aboldandcoded".split(/<(\//)?([<>]+)>/)
```

执行结果是数组：

```
["A", undefined, "B", "bold", "/", "B", "and", undefined,
"CODE", "coded", "/", "CODE", ""]
```

如果 `separator` 是 `undefined`，则返回结果是只包含 `this` 值（转换为字符串）一个字符串元素的数组。如果 `limit` 不是 `undefined`，则输出数组被切断为包含不大于 `limit` 个元素。

当调用 `split` 方法，采用以下步骤：

1. 以 `this` 值作为参数调用 `CheckObjectCoercible`。
2. 令 `S` 为以 `this` 值作为参数调用 `ToString` 的结果。
3. 令 `A` 为 仿佛使用表达式 `new Array()` 创建的新对象，这里的 `Array` 是标准内置构造器名。
4. 令 `lengthA` 为 0。
5. 如果 `limit` 是 `undefined`，令 `lim = 232-1`；否则 令 `lim = ToUint32(limit)`。
6. 令 `s` 为 `S` 的字符数。
7. 令 `p = 0`。
8. 如果 `separator` 是正则对象（它的 `[[Class]]` 是 `"RegExp"`），令 `R = separator`；否则，令 `R = ToString(separator)`。
9. 如果 `lim = 0`，返回 `A`。
10. 如果 `separator` 是 `undefined`，则
 1. 以 `"0"`，属性描述符 `{[[Value]]: S, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`，和 `false` 作为参数调用 `A` 的 `[[DefineOwnProperty]]` 内部方法。
 2. 返回 `A`。
11. 如果 `s = 0`，则
 1. 调用 `SplitMatch(S, 0, R)` 并 令 `z` 为 它的 `MatchResult` 结果。
 2. 如果 `z` 不是 `failure`，返回 `A`。
12. 以 `"0"`，属性描述符 `{[[Value]]: S, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`，和 `false` 作为参数调用 `A` 的 `[[DefineOwnProperty]]` 内部方法。
13. 只要 `q ≠ s`，就重复
 1. 调用 `SplitMatch(S, q, R)` 并 令 `z` 为 它的 `MatchResult` 结果。
 2. 如果 `z` 是 `failure`，则 令 `q = q+1`。
 3. 否则，`z` 不是 `failure`

- i. `z` 必定是一个 `State`. 令 `e` 为 `z` 的 `endIndex` 并令 `cap` 为 `z` 的 `captures` 数组 .
- ii. 如果 `e = p`, 则 令 `q = q+1`.
- iii. 否则 , `e ≠ p`
 1. 令 `T` 为一个字符串, 它的值等于包含 -- 在 `S` 中从 `p` (包括它) 位置到 `q` (不包括) 位置的字符 -- 的子字符串的值。
 2. 以 `ToString(lengthA)`, 属性描述符 `{[[Value]]: T, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, 和 `false` 作为参数调用 `A` 的 `[[DefineOwnProperty]]` 内部方法 .
 3. `lengthA` 递增 1.
 4. 如果 `lengthA = lim`, 返回 `A`.
 5. 令 `p = e`.
 6. 令 `i = 0`.
 7. 只要 `i` 不等于 `cap` 中的元素个数, 就重复 .
 1. 令 `i = i+1`.
 2. 以 `ToString(lengthA)`, 属性描述符 `{[[Value]]: cap[i], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, 和 `false` 作为参数调用 `A` 的 `[[DefineOwnProperty]]` 内部方法。
 3. `lengthA` 递增 1.
 4. 如果 `lengthA = lim`, 返回 `A`.
 8. 令 `q = p`.
 14. 令 `T` 为 为一个字符串, 它的值等于包含 -- 在 `S` 中从 `p` (包括它) 位置到 `q` (不包括) 位置的字符 -- 的子字符串的值。
 15. 以 `ToString(lengthA)`, 属性描述符 `{[[Value]]: T, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, 和 `false` 作为参数调用 `A` 的 `[[DefineOwnProperty]]` 内部方法。
 16. 返回 `A`.

`SplitMatch` 抽象操作需要三个参数, 字符串 `S`, 整数 `q`, 字符串或正则对象 `R`, 按照以下顺序执行并返回一个 `MatchResult` (见 15.10.2.1) :

1. 如果 `R` 是个正则对象 (它的 `[[Class]]` 是 `"RegExp"`), 则
 1. 以 `S` 和 `q` 作为参数调用 `R` 的 `[[Match]]` 内部方法, 并返回 `MatchResult` 的结果。
2. 否则, `Type(R)` 必定是 `String`. 令 `r` 为 `R` 的字符数 .
3. 令 `s` 为 `S` 的字符数 .
4. 如果 `q+r > s` 则返回 `MatchResult failure`.
5. 如果存在一个在 0 (包括) 到 `r` (不包括) 之间的整数 `i`, 使得 `S` 的 `q+i` 位置上的字符和 `R` 的 `i` 位置上的字符不同, 则返回 `failure`.
6. 令 `cap` 为 `captures` 的空数组 (见 15.10.2.1).
7. 返回 `State` 数据结构 (`q+r, cap`). (见 15.10.2.1)

`split` 方法的 `length` 属性是 2.

分隔符是正则对象时, `split` 方法忽略 `separator.global` 的值。

`split` 函数被有意设计成通用的；它不要求它的 `this` 值是字符串对象。因此，他可以当做方法转移到其他类型对象。

String.prototype.substring (start, end)

`substring` 方法需要两个参数 `start` 和 `end`，将 `this` 对象转换为一个字符串，返回包含 -- 在转换结果字符串中从 `start` 位置字符一直到（但不包括）`end` 位置的字符（或如果 `end` 是 `undefined`，就到字符串末尾）-- 的一个子串。返回结果是字符串值，不是字符串对象。

如果任一参数是 `NaN` 或负数，它被零取代；如果任一参数大于字符串长度，它被字符串长度取代。

如果 `start` 大于 `end`，交换它们的值。

采用以下步骤：

1. 以 `this` 值作为参数调用 `CheckObjectCoercible`。
2. 令 `S` 为以 `this` 值作为参数调用 `ToString` 的结果。
3. 令 `len` 为 `S` 的字符数。
4. 令 `intStart` 为 `ToInteger(start)`。
5. 如果 `end` 是 `undefined`，令 `intEnd` 为 `len`；否则 令 `intEnd` 为 `ToInteger(end)`。
6. 令 `finalStart` 为 `min(max(intStart, 0), len)`。
7. 令 `finalEnd` 为 `min(max(intEnd, 0), len)`。
8. 令 `from` 为 `min(finalStart, finalEnd)`。
9. 令 `to` 为 `max(finalStart, finalEnd)`。
10. 返回 一个长度是 `to - from` 的字符串，它包含 `S` 中从索引值 `from` 到 `to-1`（按照索引升序）的所有字符。

`substring` 方法的 `length` 属性是 2。

`substring` 函数被有意设计成通用的；它不要求它的 `this` 值是字符串对象。因此，他可以当做方法转移到其他类型对象。

String.prototype.toLowerCase ()

采用以下步骤：

1. 以 `this` 值作为参数调用 `CheckObjectCoercible`。
2. 令 `S` 为以 `this` 值作为参数调用 `ToString` 的结果。
3. 令 `L` 为一个字符串，`L` 的每个字符是 `S` 中相应字符的 `Unicode` 小写等量，或者（如果没有 `Unicode` 小写等量存在）是实际的 `S` 中相应字符值。

4. 返回 L.

为了此操作，字符串的 16 位代码单元被看作是 Unicode 基本多文种平面（Basic Multilingual Plane）中的代码点。代理代码点直接从 S 转移到 L，不做任何映射。

返回结果必须是根据 Unicode 字符数据库里的大小写映射得到的（对此数据库明确规定，不仅包括 UnicodeData.txt 文件，而且还包括 Unicode 2.1.8 和更高版本里附带的 SpecialCasings.txt 文件）。

某些字符的大小写映射可产生多个字符。这种情况下结果字符串与原字符串的长度未必相等。因为 toUpperCase 和 toLowerCase 都有上下文敏感的行为，所以这俩函数不是对称的。也就是说，s.toUpperCase().toLowerCase() 不一定等于 s.toLowerCase()。

toLowerCase 函数被有意设计成通用的；它不要求它的 this 值是字符串对象。因此，他可以当做方法转移到其他类型对象。

String.prototype.toLocaleLowerCase ()

此函数产生依照 -- 宿主环境的当前语言设置 -- 更正的结果，而不是独立于语言环境的结果，除此之外它的运作方式与 toLowerCase 完全一样。只有在少数情况下有一个区别（如，土耳其语），就是那个语言和正规 Unicode 大小写映射有冲突时的规则。

此函数的第一个参数可能会用于本标准的未来版本；建议实现不以任何用途使用这个参数位置。

toLocaleLowerCase 函数被有意设计成通用的；它不要求它的 this 值是字符串对象。因此，他可以当做方法转移到其他类型对象。

String.prototype.toUpperCase ()

此函数的将字符映射到在 Unicode 字符数据库中与其等值的大写字符，除此之外此函数的行为采用与 String.prototype.toLowerCase 完全相同的方式。

toUpperCase 函数被有意设计成通用的；它不要求它的 this 值是字符串对象。因此，他可以当做方法转移到其他类型对象。

String.prototype.toLocaleUpperCase ()

此函数产生依照 -- 宿主环境的当前语言设置 -- 更正的结果，而不是独立于语言环境的结果，除此之外它的运作方式与 toUpperCase 完全一样。只有在少数

情况下有一个区别（如，土耳其语），就是那个语言和正规 Unicode 大小写映射有冲突时的规则。

此函数的第一个参数可能会用于本标准的未来版本；建议实现不以任何用途使用这个参数位置。

`toLocaleUpperCase` 函数被有意设计成通用的；它不要求它的 `this` 值是字符串对象。因此，他可以当做方法转移到其他类型对象。

String.prototype.trim ()

采用以下步骤：

1. 以 `this` 值作为参数调用 `CheckObjectCoercible`。
2. 令 `S` 为以 `this` 值作为参数调用 `ToString` 的结果。
3. 令 `T` 为一个字符串值，它是 `S` 的一个拷贝，并删除了开头和结尾中空白的。空白的定义是 `WhiteSpace` 和 `LineTerminator` 的并集。
4. 返回 `T`。

`trim` 函数被有意设计成通用的；它不要求它的 `this` 值是字符串对象。因此，他可以当做方法转移到其他类型对象。

String 实例的属性

字符串实例从字符串原型对象继承属性，字符串实例的 `[[Class]]` 内部属性值是 `"String"`。字符串实例还有 `[[PrimitiveValue]]` 内部属性，`length` 属性，和一组属性名是数组索引的可遍历属性。

`[[PrimitiveValue]]` 内部属性是代表这个字符串对象的字符串值。以数组索引命名的属性对应字符串值里的单字符。一个特殊的 `[[GetOwnProperty]]` 内部方法用来为数组索引命名的属性指定数字，值，和特性。

length

在代表这个字符串对象的字符串值里的字符数。

一旦创建了一个字符串对象，这个属性是不可变的。它有特性 `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`。

[[GetOwnProperty]] (P)

数组对象使用一个，用在其他原生 ECMAScript 对象的 `[[GetOwnProperty]]` 内部方法 (8.12.1) 的变化版。这个特殊内部方法用来给命名属性添加访问器，对应到字符串对象的单字符。

设 `S` 为一个字符串对象，`P` 为一个字符串。

当以属性名 `P` 调用 `S` 的 `[[GetOwnProperty]]` 内部方法，采用以下步骤：

1. 令 `desc` 为以 `P` 为参数调用 `S` 的默认 `[[GetOwnProperty]]` 内部方法 (8.12.1) 的结果。
2. 如果 `desc` 不是 `undefined`，返回 `desc`。
3. 如果 `ToString(abs(ToInteger(P)))` 与 `P` 的值不同，返回 `undefined`。
4. 令 `str` 为 `S` 的 `[[PrimitiveValue]]` 内部属性字符串值。
5. 令 `index` 为 `ToInteger(P)`。
6. 令 `len` 为 `str` 里的字符数。
7. 如果 `len ≤ index`，返回 `undefined`。
8. 令 `resultStr` 为一个长度为 1 的字符串，里面包含 `str` 中 `index` 位置的一个字符，在这里 `str` 中的第一个（最左边）字符被认为是在位置 0，下一个字符在位置 1，依此类推。
9. 返回一个属性描述符 `{ [[Value]]: resultStr, [[Enumerable]]: true, [[Writable]]: false, [[Configurable]]: false }`

布尔对象

作为函数调用布尔构造器

当把 `Boolean` 作为函数来调用，而不是作为构造器，它执行一个类型转换。

`Boolean (value)`

返回由 `ToBoolean(value)` 计算出的布尔值（非布尔对象）。

布尔构造器

当 `Boolean` 作为 `new` 表达式的一部分来调用，那么它是一个构造器：它初始化新创建的对象。

`new Boolean (value)`

新构造对象的 `[[Prototype]]` 内部属性设定为原始布尔原型对象，它是 `Boolean.prototype` (15.6.3.1) 的初始值。

新构造对象的 `[[Class]]` 内部属性设定为 `"Boolean"`。

新构造对象的 `[[PrimitiveValue]]` 内部属性设定为 `ToBoolean(value)`。

新构造对象的 `[[Extensible]]` 内部属性设定为 `true`。

布尔构造器的属性

布尔构造器的 `[[Prototype]]` 内部属性的值是函数原型对象 (15.3.4)。

除了内部属性和 `length` 属性（值为 1）外，布尔构造器还有以下属性：

`Boolean.prototype`

`Boolean.prototype` 的初始值是布尔原型对象 (15.6.4)。

这个属性有特性 { `[[Writable]]`: `false`, `[[Enumerable]]`: `false`, `[[Configurable]]`: `false` }。

布尔原型对象的属性

布尔原型对象自身是一个值为 `false` 的布尔对象（它的 `[[Class]]` 是 `"Boolean"`）。

布尔原型对象的 `[[Prototype]]` 内部属性值是标准内置的对象原型对象 (15.2.4)。

`Boolean.prototype.constructor`

`Boolean.prototype.constructor` 的初始值是内置的 `Boolean` 构造器。

`Boolean.prototype.toString ()`

采用以下步骤：

1. 令 `B` 为 `this` 值。
2. 如果 `Type(B)` 是 `Boolean`，则令 `b` 为 `B`。
3. 否则如果 `Type(B)` 是 `Object` 且 `B` 的 `[[Class]]` 内部属性值是 `"Boolean"`，则令 `b` 为 `B` 的 `[[PrimitiveValue]]` 内部属性值。
4. 否则抛出一个 `TypeError` 异常。

5. 如果 `b` 是 `true`, 则返回 `"true"`; 否则返回 `"false"`.

Boolean.prototype.valueOf ()

采用以下步骤:

1. 令 `B` 为 `this` 值 .
2. 如果 `Type(B)` 是 `Boolean`, 则令 `b` 为 `B`.
3. 否则如果 `Type(B)` 是 `Object` 且 `B` 的 `[[Class]]` 内部属性值是 `"Boolean"`, 则令 `b` 为 `B` 的 `[[PrimitiveValue]]` 内部属性值。
4. 否则抛出一个 `TypeError` 异常 .
5. 返回 `b`.

布尔实例的属性

布尔实例从布尔原型对象继承属性, 且布尔实例的 `[[Class]]` 内部属性值是 `"Boolean"`。布尔实例还有一个 `[[PrimitiveValue]]` 内部属性。

`[[PrimitiveValue]]` 内部属性是代表这个布尔对象的布尔值。

Number 对象

作为函数调用的 Number 构造器

当把 `Number` 当作一个函数来调用, 而不是作为构造器, 它执行一个类型转换。

Number ([value])

如果提供了 `value`, 返回 `ToNumber(value)` 计算出的数字值 (非 `Number` 对象), 否则返回 `+0`。

Number 构造器

当把 `Number` 作为 `new` 表达式的一部分来调用, 它是构造器: 它初始化新创建的对象。

new Number ([value])

新构造对象的 `[[Prototype]]` 内部属性设定为原始数字原型对象，它是 `Number.prototype` 的初始值（15.7.3.1）。

新构造对象的 `[[Class]]` 内部属性设定为 `"Number"`。

新构造对象的 `[[PrimitiveValue]]` 内部属性在提供了 `value` 时设定为 `ToNumber(value)`，否则设定为 `+0`。

新构造对象的 `[[Extensible]]` 内部属性设定为 `true`。

Number 构造器的属性

`Number` 构造器的 `[[Prototype]]` 内部属性值是函数原型对象（15.3.4）。

除了内部属性和 `length` 属性（值为 1）之外，`Number` 构造器还有以下属性：

Number.prototype

`Number.prototype` 的初始值是数字原型对象。

这个属性有特性 { `[[Writable]]`: `false`, `[[Enumerable]]`: `false`, `[[Configurable]]`: `false` }。

Number.MAX_VALUE

`Number.MAX_VALUE` 的值是数字类型的最大正有限值，约为 $1.7976931348623157 \times 10^{308}$ 。

这个属性有特性 { `[[Writable]]`: `false`, `[[Enumerable]]`: `false`, `[[Configurable]]`: `false` }。

Number.MIN_VALUE

`Number.MIN_VALUE` 的值是数字类型的最小正有限值，约为 5×10^{-324} 。

这个属性有特性 { `[[Writable]]`: `false`, `[[Enumerable]]`: `false`, `[[Configurable]]`: `false` }。

Number.NaN

`Number.NaN` 的值是 `NaN`。

这个属性有特性 { `[[Writable]]`: false, `[[Enumerable]]`: false, `[[Configurable]]`: false }。

Number.NEGATIVE_INFINITY

`Number.NEGATIVE_INFINITY` 的值是 $-\infty$ 。

这个属性有特性 { `[[Writable]]`: false, `[[Enumerable]]`: false, `[[Configurable]]`: false }。

Number.POSITIVE_INFINITY

`Number.POSITIVE_INFINITY` 的值是 $+\infty$ 。

这个属性有特性 { `[[Writable]]`: false, `[[Enumerable]]`: false, `[[Configurable]]`: false }。

数字原型对象的属性

数字原型对象其自身是 `Number` 对象(其 `[[Class]]` 是 "Number"),其值为 $+0$ 。

数字原型对象的 `[[Prototype]]` 内部属性值是标准内置 `Object` 原型对象 (15.2.4)。

除非另外明确声明,以下定义的数字原型对象的方法是非通用的,传递给它们的 `this` 值必须是数字值或 `[[Class]]` 内部属性值是 "Number" 的对象。

在以下对作为数字原型对象属性的函数的描述中,短语“`this Number` 对象”是指函数调用中的 `this` 值,或如果 `Type(this 值)` 是 `Number`,“`this Number` 对象”指仿佛是用表达式 `new Number(this value)` 创建的对象,这里 `Number` 是标准内置构造器名。此外,短语“`this 数字值`”是指代表 `this Number` 对象的数字值,也就是 `this Number` 对象的 `[[PrimitiveValue]]` 内部属性值;或如果 `this` 是数字类型,“`this 数字值`”指 `this` 值。如果 `this` 值不是 `[[Class]]` 内部属性为 "Number" 的对象,也不是数字类型的值,则抛出一个 `TypeError` 异常。

Number.prototype.constructor

`Number.prototype.constructor` 的初始值是内置 `Number` 构造器。

Number.prototype.toString ([radix])

可选参数 **radix** 应当是 2 到 36 闭区间上的整数。如果 **radix** 不存在或是 **undefined**，用数字 10 作为 **radix** 的值。如果 **ToInteger(radix)** 是数字 10，则将 **this Number** 对象作为一个参数传给 **ToString** 抽象操作；返回结果字符串值。

如果 **ToInteger(radix)** 不是在 2 到 36 闭区间上的整数，则抛出一个 **RangeError** 异常。如果 **ToInteger(radix)** 是 2 到 36 的整数，但不是 10，则结果是 **this** 数字值使用指定基数表示法的字符串。字母 **a-z** 用来指值为 10 到 35 的数字。基数不为 10 时的精确算法是依赖于实现的，然而算法应当是 9.8.1 指定算法的推广形式。

toString 函数不是通用的；如果 **this** 值不是数字或 **Number** 对象，抛出一个 **TypeError** 异常。因此它不能当作方法转移到其他类型对象上。

Number.prototype.toLocaleString()

根据宿主环境的当前语言环境惯例来格式化 **this** 数字值，生成代表这个值的字符串。此函数是依赖于实现的，允许但不鼓励它的返回值与 **toString** 相同。

此函数的第一个参数可能会用于本标准的未来版本；建议实现不以任何用途使用这个参数位置。

Number.prototype.valueOf ()

返回 **this** 数字值。

valueOf 函数不是通用的；如果 **this** 值不是数字或 **Number** 对象，抛出一个 **TypeError** 异常。因此它不能当作方法转移到其他类型对象上。

Number.prototype.toFixed (fractionDigits)

返回一个包含了 **--** 代表 **this** 数字值的留有小数点后 **fractionDigits** 个数字的十进制固定小数点记法 **--** 的字符串。如果 **fractionDigits** 是 **undefined**，就认为是 0。具体来说，执行以下步骤：

1. 令 **f** 为 **ToInteger(fractionDigits)**. (如果 **fractionDigits** 是 **undefined**, 此步骤产生 0 值).
2. 如果 **f < 0** 或 **f > 20**, 抛出一个 **RangeError** 异常 .
3. 令 **x** 为 **this** 数字值 .
4. 如果 **x** 是 **NaN**, 返回字符串 "NaN".
5. 令 **s** 为空字符串 .
6. 如果 **x < 0**, 则

1. 令 s 为 "-".
2. 令 $x = -x$.
7. 如果 $x \geq 10^{21}$, 则
 1. 令 $m = \text{ToString}(x)$.
 8. 否则, $x < 10^{21}$
 1. 令 n 为一个整数, 让 $n \div 10^f - x$ 准确的数学值尽可能接近零。如果有两个这样 n 值, 选择较大的 n 。
 2. 如果 $n = 0$, 令 m 为字符串 "0". 否则, 令 m 为由 n 的十进制表示里的数组成的字符串 (为了没有前导零)。
 3. 如果 $f \neq 0$, 则
 - i. 令 k 为 m 里的字符数目。
 - ii. 如果 $k \leq f$, 则
 1. 令 z 为 $f+1-k$ 个 '0' 组成的字符串。
 2. 令 m 为串联字符串 z 的 m 的结果。
 3. 令 $k = f + 1$.
 - iii. 令 a 为 m 的前 $k-f$ 个字符, 令 b 为其余 f 个字符。
 - iv. 令 m 为串联三个字符串 a , ".", 和 b 的结果。
 9. 返回串联字符串 s 和 m 的结果。

`toFixed` 方法的 `length` 属性是 1。

如果以多个参数调用 `toFixed` 方法, 则行为是不确定的 (见 15 章)。

实现是被允许在 `fractionDigits` 小于 0 或大于 20 时扩展 `toFixed` 的行为。在这种情况下, 对这样的 `fractionDigits` 值 `toFixed` 将未必抛出 `RangeError`。

对于某些值, `toFixed` 的输出可比 `toString` 的更精确, 因为 `toString` 只打印区分相邻数字值的足够的有效数字。例如,

`(1000000000000000128).toString()` 返回 "1000000000000000100",

而 `(1000000000000000128).toFixed(0)` 返回 "1000000000000000128".

Number.prototype.toExponential (fractionDigits)

返回一个代表 `this` 数字值的科学计数法的字符串, 它的有效数字的小数点前有一个数字, 有效数字的小数点后有 `fractionDigits` 个数字。如果 `fractionDigits` 是 `undefined`, 包括指定唯一数字值需要的尽可能多的有效数字 (就像 `ToString`, 但在这里总是以科学计数法输出)。具体来说执行以下步骤:

1. 令 x 为 `this` 数字值。
2. 令 f 为 `ToInteger(fractionDigits)`.
3. 如果 x 是 `NaN`, 返回字符串 "NaN".
4. 令 s 为空字符串。

5. 如果 $x < 0$, 则
 1. 令 s 为 "-".
 2. 令 $x = -x$.
6. 如果 $x = +\infty$, 则
 1. 返回串联字符串 s 和 "Infinity" 的结果 .
7. 如果 `fractionDigits` 不是 `undefined` 且 ($f < 0$ 或 $f > 20$), 抛出一个 `RangeError` 异常 .
8. 如果 $x = 0$, 则
 1. 令 $f = 0$.
 2. 令 m 为包含 $f+1$ 个 '0' 的字符串。
 3. 令 $e = 0$.
9. 否则, $x \neq 0$
 1. 如果 `fractionDigits` 不是 `undefined`, 则
 - i. 令 e 和 n 为整数, 使得满足 $10^f \leq n < 10^{f+1}$ 且 $n \times 10^{e-f} - x$ 的准确数学值尽可能接近零。如果 e 和 n 有两个这样的组合, 选择使 $n \times 10^{e-f}$ 更大的组合。
 2. 否则, `fractionDigits` 是 `undefined`
 - i. 令 e, n , 和 f 为整数, 使得满足 $f \geq 0, 10^f \leq n < 10^{f+1}, n \times 10^{e-f}$ 的数字值是 x , 且 f 的值尽可能小。注: n 的十进制表示有 $f+1$ 个数字, n 不能被 10 整除, 并且 n 的最少有效位数不一定唯一由这些条件确定。
 3. 令 m 为由 n 的十进制表示里的数 组成的字符串 (没有前导零)。
 10. 如果 $f \neq 0$, 则
 1. 令 a 为 m 中的第一个字符, 令 b 为 m 中的其余字符 .
 2. 令 m 为串联三个字符串 $a, ".",$ 的 b 的结果 .
 11. 如果 $e = 0$, 则
 1. 令 $c = "+"$.
 2. 令 $d = "0"$.
 12. 否则
 1. 如果 $e > 0$, 则 令 $c = "+"$.
 2. 否则, $e \leq 0$
 - i. 令 $c = "-"$.
 - ii. 令 $e = -e$.
 3. 令 d 为有 e 的十进制表示里的数 组成的字符串 (没有前导零)。
 13. 令 m 为串联四个字符串 $m, "e", c,$ 和 d 的结果 .
 14. 返回串联字符串 s 和 m 的结果 .

`toExponential` 方法的 `length` 属性是 1。

如果用多于一个参数调用 `toExponential` 方法, 则行为是未定义的 (见 15 章)。

一个实现可以扩展 `fractionDigits` 的值小于 0 或大于 20 时 `toExponential` 的行为。这种情况下对这样的 `fractionDigits` 值, `toExponential` 不一定抛出 `RangeError` 异常 .

对于需要提供比上述规则更准确转换的实现, 建议用以下算法作为指引替代步骤 9.b.i :

1. 令 e, n , 和 f 为整数, 使得满足 $f \geq 0, 10^f \leq n < 10^{f+1}, n \times 10^{e-f}$ 的数字值是 x , 且 f 的值尽可能小。如果这样的 n 值可能多个, 选择使 $n \times 10^{e-f}$ 的值尽可能接近 x 的 n 值。如果有两个这样的 n 值, 选择偶数。

Number.prototype.toPrecision (precision)

返回一个字符串, 它代表 **this** 数字值的科学计数法 (有效数字的小数点前有一个数字, 有效数字的小数点后有 **precision-1** 个数字) 或十进制固定计数法 (**precision** 个有效数字)。如果 **precision** 是 **undefined**, 用 **ToString** (9.8.1) 调用代替。具体来说执行以下步骤:

1. 令 x 为 **this** 数字值 .
2. 如果 **precision** 是 **undefined**, 返回 **ToString**(x).
3. 令 p 为 **ToInteger**(**precision**).
4. 如果 x 是 **NaN**, 返回字符串 "NaN".
5. 令 s 为空字符串 .
6. 如果 $x < 0$, 则
 1. 令 s 为 "-".
 2. 令 $x = -x$.
7. 如果 $x = +\infty$, 则
 1. 返回串联字符串 s 和 "Infinity" 的结果 .
8. 如果 $p < 1$ 或 $p > 21$, 抛出一个 **RangeError** 异常 .
9. 如果 $x = 0$, 则
 1. 令 m 为 p 个 '0' 组成的字符串 .
 2. 令 $e = 0$.
10. 否则 $x \neq 0$,
 1. 令 e 和 n 为整数, 使得满足 $10^{p-1} \leq n < 10^p$ 且 $n \times 10^{e-p+1} - x$ 的准确数值值尽可能接近零。如果 e 和 n 有两个这样的组合, 选择使 $n \times 10^{e-p+1}$ 更大的组合。
 2. 令 m 为由 n 的十进制表示里的数 组成的字符串 (没有前导零)。
 3. 如果 $e < -6$ 或 $e \geq p$, 则
 - i. 令 a 为 n 的第一个字符, 令 b 为 m 的其余 $p-1$ 个字符 .
 - ii. 令 m 为串联三个字符串 $a, ".",$ 和 b 的结果 .
 - iii. 如果 $e = 0$, 则
 1. 令 $c = "+"$, 令 $d = "0"$.
 - iv. 否则 $e \neq 0$,
 1. 如果 $e > 0$, 则
 1. 令 $c = "+"$.
 2. 否则 $e < 0$,
 1. 令 $c = "-"$.
 2. 令 $e = -e$.

3. 令 d 为由 e 的十进制表示里的数 组成的字符串（没有前导零）。
- v. 令 m 为串联五个字符串 $s, m, "e", c,$ 和 d 的结果。
11. 如果 $e = p-1$, 则返回串联字符串 s 和 m 的结果。
12. 如果 $e \geq 0$, 则
 1. 令 m 为 m 的前 $e+1$ 个字符, 字符 '.', m 的其余 $p-(e+1)$ 个字符 串联的结果。
13. 否则 $e < 0$,
 1. 令 m 为 字符串 "0.", $-(e+1)$ 个字符 '0', 字符串 m 串联的结果。
14. 返回字符串 s 和 m 串联的结果。

`toPrecision` 方法的 `length` 属性是 1。

如果用多于一个参数调用 `toPrecision` 方法, 则行为是未定义的 (见 15 章)。

一个实现可以扩展 `precision` 的值小于 1 或大于 21 时 `toPrecision` 的行为。这种情况下对这样的 `precision` 值, `toPrecision` 不一定抛出 `RangeError` 异常。

数字实例的属性

数字实例从数字原型对象继承属性, 数字实例的 `[[Class]]` 内部属性是 "Number"。数字实例还有一个 `[[PrimitiveValue]]` 内部属性。

`[[PrimitiveValue]]` 内部属性是代表 `this Number` 对象的数字值。

Math 对象

`Math` 对象是拥有一些命名属性的单一对象, 其中一些属性值是函数。

`Math` 对象的 `[[Prototype]]` 内部属性值是标准内置 `Object` 原型对象 (15.2.4)。
`Math` 对象的 `[[Class]]` 内部属性值是 "Math"。

`Math` 对象没有 `[[Construct]]` 内部属性 ; `Math` 对象不能作为构造器被 `new` 运算符调用。

`Math` 对象没有 `[[Call]]` 内部属性; `Math` 对象不能作为函数被调用。

本规范中, 短语“ x 的数字值”的技术含义定义在 8.5。

Math 对象的值属性

E

自然对数的底数 e 的数字值，约为 2.7182818284590452354。

此属性有特性 { `[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false` }。

LN10

10 的自然对数的数字值，约为 2.302585092994046。

此属性有特性 { `[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false` }。

LN2

2 的自然对数的数字值，约为 0.6931471805599453。

此属性有特性 { `[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false` }。

LOG2E

自然对数的底数 e 的以 2 为底数的对数的数字值；约为 1.4426950408889634。

此属性有特性 { `[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false` }。

Math.LOG2E 的值约为 **Math.LN2** 值的倒数。

LOG10E

自然对数的底数 e 的以 10 为底数的对数的数字值；约为 0.4342944819032518。

此属性有特性 { `[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false` }。

Math.LOG10E 的值约为 **Math.LN10** 值的倒数。

PI

圆的周长与直径之比 π 的数字值，约为 3.1415926535897932。

此属性有特性 { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false } 。

SQRT1_2

$\frac{1}{2}$ 的平方根的数字值，约为 0.7071067811865476。

此属性有特性 { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false } 。

`Math.SQRT1_2` 的值约为 `Math.SQRT2` 值的倒数。

SQRT2

2 的平方根的数字值，约为 1.4142135623730951。

此属性有特性 { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false } 。

Math 对象的函数属性

对以下每个 `Math` 对象函数的每个参数（如果有多个，以左到右的顺序）应用 `ToNumber` 抽象操作，然后对结果数字值执行计算。

下面对函数的描述中，符号 `NaN`, `-0`, `+0`, `$-\infty$` , `$+\infty$` 指 8.5 描述的数字值。

这里没有精确规定函数 `acos`, `asin`, `atan`, `atan2`, `cos`, `exp`, `log`, `pow`, `sin`, `sqrt` 的行为，除了需要特别说明对边界情况某些参数值的结果之外。对其他参数值，这些函数旨在计算计算常见数学函数的结果，但选择的近似算法中的某些范围是被允许的。The general intent is that an implementer should be able to use the same mathematical library for ECMAScript on a given hardware platform that is available to C programmers on that platform.

Although the choice of algorithms is left to the implementation, it is recommended (but not specified by this standard) that implementations use the approximation algorithms for IEEE 754 arithmetic contained in `fdlibm`, the freely distributable mathematical library from Sun Microsystems (<http://www.netlib.org/fdlibm>).

abs(x)

返回 x 的绝对值。

- 若 x 是 NaN, 返回结果是 NaN.
- 若 x 是 -0 , 返回结果是 $+0$.
- 若 x 是 $-\infty$, 返回结果是 $+\infty$.

acos (x)

返回 x 的反余弦的依赖实现的近似值。结果以弧度形式表示, 范围是 $+0$ 到 $+\pi$ 。

- 若 x 是 NaN, 返回结果是 NaN.
- 若 x 大于 1, 返回结果是 NaN.
- 若 x 小于 -1 , 返回结果是 NaN.
- 若 x 正好是 1, 返回结果是 $+0$.

asin (x)

返回 x 的正弦的依赖实现的近似值。结果以弧度形式表示, 范围是 $-\pi/2$ 到 $+\pi/2$ 。

- 若 x 是 NaN, 返回结果是 NaN.
- 若 x 大于 1, 返回结果是 NaN.
- 若 x 小于 -1 , 返回结果是 NaN.
- 若 x 是 $+0$, 返回结果是 $+0$.
- 若 x 是 -0 , 返回结果是 -0 .

atan (x)

返回 x 的正切的依赖实现的近似值。结果以弧度形式表示, 范围是 $-\pi/2$ 到 $+\pi/2$ 。

- 若 x 是 NaN, 返回结果是 NaN.
- 若 x 是 $+0$, 返回结果是 $+0$.
- 若 x 是 -0 , 返回结果是 -0 .
- 若 x 是 $+\infty$, 返回结果是一个依赖于实现的近似值 $+\pi/2$.
- 若 x 是 $-\infty$, 返回结果是一个依赖于实现的近似值 $-\pi/2$.

atan2 (y, x)

返回 -- 参数 y 和 x 的商 y/x -- 的正切的依赖实现的近似值, y 和 x 的符号用于确定返回值的象限。注: 命名为 y 的参数为第一个, 命名为 x 的参数为

第二个，这是有意，是反正切函数俩参数的惯例。结果以弧度形式表示，范围是 $-\pi$ 到 $+\pi$ 。

- 若 x 和 y 至少一个是 NaN，返回结果是 NaN.
- 若 $y > 0$ 且 x 是 $+0$ ，返回结果是一个依赖于实现的近似值 $+\pi/2$.
- 若 $y > 0$ 且 x 是 -0 ，返回结果是一个依赖于实现的近似值 $+\pi/2$.
- 若 y 是 $+0$ 且 $x > 0$ ，返回结果是 $+0$.
- 若 y 是 $+0$ 且 x 是 $+0$ ，返回结果是 $+0$.
- 若 y 是 $+0$ 且 x 是 -0 ，返回结果是一个依赖于实现的近似值 $+\pi$.
- 若 y 是 $+0$ 且 $x < 0$ ，返回结果是一个依赖于实现的近似值 $+\pi$.
- 若 y 是 -0 且 $x > 0$ ，返回结果是 -0 .
- 若 y 是 -0 且 x 是 $+0$ ，返回结果是 -0 .
- 若 y 是 -0 且 x 是 -0 ，返回结果是一个依赖于实现的近似值 $-\pi$.
- 若 y 是 -0 且 $x < 0$ ，返回结果是一个依赖于实现的近似值 $-\pi$.
- 若 $y < 0$ 且 x 是 $+0$ ，返回结果是一个依赖于实现的近似值 $-\pi/2$.
- 若 $y < 0$ 且 x 是 -0 ，返回结果是一个依赖于实现的近似值 $-\pi/2$.
- 若 $y > 0$ 且 y 是有限的 且 x 是 $+\infty$ ，返回结果是 $+0$.
- 若 $y > 0$ 且 y 是有限的 且 x 是 $-\infty$ ，返回结果是一个依赖于实现的近似值 $+\pi$.
- 若 $y < 0$ 且 y 是有限的 且 x 是 $+\infty$ ，返回结果是 -0 .
- 若 $y < 0$ 且 y 是有限的 且 x 是 $-\infty$ ，返回结果是一个依赖于实现的近似值 $-\pi$.
- 若 y 是 $+\infty$ 且 x 是有限的，返回结果是 返回结果是一个依赖于实现的近似值 $+\pi/2$.
- 若 y 是 $-\infty$ 且 x 是有限的，返回结果是 返回结果是一个依赖于实现的近似值 $-\pi/2$.
- 若 y 是 $+\infty$ 且 x 是 $+\infty$ ，返回结果是一个依赖于实现的近似值 $+\pi/4$.
- 若 y 是 $+\infty$ 且 x 是 $-\infty$ ，返回结果是一个依赖于实现的近似值 $+3\pi/4$.
- 若 y 是 $-\infty$ 且 x 是 $+\infty$ ，返回结果是一个依赖于实现的近似值 $-\pi/4$.
- 若 y 是 $-\infty$ 且 x 是 $-\infty$ ，返回结果是一个依赖于实现的近似值 $-3\pi/4$.

ceil (x)

返回不小于 x 的且为数学整数的最小（接近 $-\infty$ ）数字值。如果 x 已是整数，则返回 x 。

- 若 x 是 NaN，返回结果是 NaN.
- 若 x 是 $+0$ ，返回结果是 $+0$.
- 若 x 是 -0 ，返回结果是 -0 .
- 若 x 是 $+\infty$ ，返回结果是 $+\infty$.
- 若 x 是 $-\infty$ ，返回结果是 $-\infty$.
- 若 x 小于 0 但大于 -1，返回结果是 -0 .

Math.ceil(x) 的值与 -Math.floor(-x) 的值相同。

cos (x)

返回 x 的余弦的依赖实现的近似值。参数被当做是弧度值。

- 若 x 是 NaN, 返回结果是 NaN.
- 若 x 是 +0, 返回结果是 1.
- 若 x 是 -0, 返回结果是 1.
- 若 x 是 $+\infty$, 返回结果是 NaN.
- 若 x 是 $-\infty$, 返回结果是 NaN.

exp (x)

返回 x 的指数的依赖实现的近似值 (e 为 x 次方, e 为自然对数的底)。

- 若 x 是 NaN, 返回结果是 NaN.
- 若 x 是 +0, 返回结果是 1.
- 若 x 是 -0, 返回结果是 1.
- 若 x 是 $+\infty$, 返回结果是 $+\infty$.
- 若 x 是 $-\infty$, 返回结果是 +0.

floor (x)

返回不大于 x 的且为数学整数的最大 (接近 $+\infty$) 数字值。如果 x 已是整数, 则返回 x 。

- 若 x 是 NaN, 返回结果是 NaN.
- 若 x 是 +0, 返回结果是 +0.
- 若 x 是 -0, 返回结果是 -0.
- 若 x 是 $+\infty$, 返回结果是 $+\infty$.
- 若 x 是 $-\infty$, 返回结果是 $-\infty$.
- 若 x 大于 0 但小于 1, 返回结果是 +0.

$\text{Math.floor}(x)$ 的值与 $-\text{Math.ceil}(-x)$ 的值相同。

log (x)

返回 x 的自然对数的依赖于实现的近似值。

- 若 x 是 NaN, 返回结果是 NaN.
- 若 x 小于 0, 返回结果是 NaN.
- 若 x 是 +0 或 -0, 返回结果是 $-\infty$.

- 若 x 是 1, 返回结果是 $+0$.
- 若 x 是 $+\infty$, 返回结果是 $+\infty$.

max ([value1 [, value2 [, ...]]])

给定零或多个参数，对每个参数调用 `ToNumber` 并返回调用结果里的最大值。

- 若 没有给定参数，返回结果是 $-\infty$.
- 若 任何值是 NaN, 返回结果是 NaN.
- 按照 11.8.5 指定方式进行值比较，确定最大值，与 11.8.5 指定方式的一个不同点是在这里 $+0$ 被看作大于 -0 .

`max` 方法的 `length` 属性是 2。

min ([value1 [, value2 [, ...]]])

给定零或多个参数，对每个参数调用 `ToNumber` 并返回调用结果里的最小值。

- 若 没有给定参数，返回结果是 $+\infty$.
- 若 任何值是 NaN, 返回结果是 NaN.
- 按照 11.8.5 指定方式进行值比较，确定最小值，与 11.8.5 指定方式的一个不同点是在这里 $+0$ 被看作大于 -0 .

`min` 方法的 `length` 属性是 2。

pow (x, y)

返回 x 的 y 次方的依赖于实现的近似值。

- 若 y 是 NaN, 返回结果是 NaN.
- 若 y 是 $+0$, 返回结果是 1, 即使 x 是 NaN.
- 若 y 是 -0 , 返回结果是 1, 即使 x 是 NaN.
- 若 x 是 NaN 且 y 是非零, 返回结果是 NaN.
- 若 $\text{abs}(x) > 1$ 且 y 是 $+\infty$, 返回结果是 $+\infty$.
- 若 $\text{abs}(x) > 1$ 且 y 是 $-\infty$, 返回结果是 $+0$.
- 若 $\text{abs}(x) == 1$ 且 y 是 $+\infty$, 返回结果是 NaN.
- 若 $\text{abs}(x) == 1$ 且 y 是 $-\infty$, 返回结果是 NaN.
- 若 $\text{abs}(x) < 1$ 且 y 是 $+\infty$, 返回结果是 $+0$.
- 若 $\text{abs}(x) < 1$ 且 y 是 $-\infty$, 返回结果是 $+\infty$.
- 若 x 是 $+\infty$ 且 $y > 0$, 返回结果是 $+\infty$.
- 若 x 是 $+\infty$ 且 $y < 0$, 返回结果是 $+0$.
- 若 x 是 $-\infty$ 且 $y > 0$ 且 y 是一个奇数, 返回结果是 $-\infty$.

- 若 x 是 $-\infty$ 且 $y > 0$ 且 y 不是一个奇数，返回结果是 $+\infty$.
- 若 x 是 $-\infty$ 且 $y < 0$ 且 y 是一个奇数，返回结果是 -0 .
- 若 x 是 $-\infty$ 且 $y < 0$ 且 y 不是一个奇数，返回结果是 $+0$.
- 若 x 是 $+0$ 且 $y > 0$ ，返回结果是 $+0$.
- 若 x 是 $+0$ 且 $y < 0$ ，返回结果是 $+\infty$.
- 若 x 是 -0 且 $y > 0$ 且 y 是一个奇数，返回结果是 -0 .
- 若 x 是 -0 且 $y > 0$ 且 y 不是一个奇数，返回结果是 $+0$.
- 若 x 是 -0 且 $y < 0$ 且 y 是一个奇数，返回结果是 $-\infty$.
- 若 x 是 -0 且 $y < 0$ 且 y 不是一个奇数，返回结果是 $+\infty$.
- 若 $x < 0$ 且 x 是有限的 且 y 是有限的 and y 不是整数，返回结果是 NaN.

random ()

返回一个大于或等于 0 但小于 1 的符号为正的数值，选择随机或在该范围内近似均匀分布的伪随机，用一个依赖与实现的算法或策略。此函数不需要参数。

round (x)

返回最接近 x 且为数学整数的数值。如果两个整数同等接近 x ，则结果是接近 $+\infty$ 的数值。如果 x 已是整数，则返回 x 。

- 若 x 是 NaN，返回结果是 NaN.
- 若 x 是 $+0$ ，返回结果是 $+0$.
- 若 x 是 -0 ，返回结果是 -0 .
- 若 x 是 $+\infty$ ，返回结果是 $+\infty$.
- 若 x 是 $-\infty$ ，返回结果是 $-\infty$.
- 若 x 大于 0 但小于 0.5，返回结果是 $+0$.
- 若 x 小于 0 但大于或等于 -0.5，返回结果是 -0 .

Math.round(3.5) 返回 4，但 Math.round(-3.5) 返回 -3.

当 x 为 -0 或 x 小于 0 当大于或等于 -0.5 时，Math.round(x) 返回 -0 ，但 Math.floor(x+0.5) 返回 $+0$ ，除了这种情况之外 Math.round(x) 的返回值与 Math.floor(x+0.5) 的返回值相同。

sin (x)

返回 x 的正弦的依赖实现的近似值。参数被当做是弧度值。

- 若 x 是 NaN，返回结果是 NaN.
- 若 x 是 $+0$ ，返回结果是 $+0$.

- 若 x 是 -0 , 返回结果是 -0 .
- 若 x 是 $+\infty$ 或 $-\infty$, 返回结果是 NaN .

sqrt (x)

返回 x 的平方根的依赖实现的近似值。

- 若 x 是 NaN , 返回结果是 NaN .
- 若 x 小于 0 , 返回结果是 NaN .
- 若 x 是 $+0$, 返回结果是 $+0$.
- 若 x 是 -0 , 返回结果是 -0 .
- 若 x 是 $+\infty$, 返回结果是 $+\infty$.

tan (x)

返回 x 的正切的依赖实现的近似值。参数被当做是弧度值。

- 若 x 是 NaN , 返回结果是 NaN .
- 若 x 是 $+0$, 返回结果是 $+0$.
- 若 x 是 -0 , 返回结果是 -0 .
- 若 x 是 $+\infty$ 或 $-\infty$, 返回结果是 NaN .

Date 对象

Date 对象的概述和抽象操作的定义

下面的抽象操作函数用来操作时间值（15.9.1.1 定义）。注：任何情况下，如果这些函数之一的任意参数是 NaN ，则结果将是 NaN 。

时间值和时间范围

一个 **Date** 对象包含一个表示特定时间瞬间的毫秒的数字值。这样的数字值叫做 **时间值**。一个时间值也可以是 NaN ，说明这个 **Date** 对象不表示特定时间瞬间。

ECMAScript 中测量的时间是从协调世界时 1970 年 1 月 1 日开始的毫秒数。在时间值中闰秒是被忽略的，假设每天正好有 86,400,000 毫秒。ECMAScript 数字值可表示的所有从 $-9,007,199,254,740,991$ 到 $9,007,199,254,740,991$ 的整数；这个范围足以衡量协调世界时 1970 年 1 月 1 日前后约 285,616 年内任何时间瞬间的精确毫秒。

ECMAScript Date 对象支持的实际时间范围是略小一些的：相对协调世界时 1970 年 1 月 1 日午夜 0 点的精确的-100,000,000 天到 100,000,000 天。这给出了协调世界时 1970 年 1 月 1 日前后 8,640,000,000,000,000 毫秒的范围。

精确的协调世界时 1970 年 1 月 1 日午夜 0 点用 +0 表示。

天数和天内时间

一个给定时间值 t 所属的天数是

$$\text{Day}(t) = \text{floor}(t / \text{msPerDay})$$

其中每天的毫秒数是

$$\text{msPerDay} = 86400000$$

余数叫做天内时间

$$\text{TimeWithinDay}(t) = t \bmod \text{msPerDay}$$

年数

ECMAScript 使用一个推算公历系统，来将一个天数映射到一个年数，并确定在那年的月份的日期。在这个系统中，闰年是且仅是（可被 4 整除）且（（不可被 100 整除）或（可被 400 整除））的年份。因此， y 年的天的数目定义为

$$\begin{aligned} \text{DaysInYear}(y) = & 365 \{ \text{如果 } (y \bmod 4) \neq 0 \} = 366 \{ \text{如果 } (y \bmod 4) = 0 \text{ 且 } (y \bmod 100) \neq 0 \} \\ & = 365 \{ \text{如果 } (y \bmod 100) = 0 \text{ 且 } (y \bmod 400) \neq 0 \} = 366 \{ \text{如果 } (y \bmod 400) = 0 \} \end{aligned}$$

所有非闰年有 365 天，其中每月的天的数目是常规的。闰年的二月里有个多出来的一天。 y 年第一天的天数是：

$$\text{DayFromYear}(y) = 365 \times (y-1970) + \text{floor}((y-1969)/4) - \text{floor}((y-1901)/100) + \text{floor}((y-1601)/400)$$

y 年的起始时间值是：

$$\text{TimeFromYear}(y) = \text{msPerDay} \times \text{DayFromYear}(y)$$

一个时间值决定的年数是：

$\text{YearFromTime}(t)$ = 满足条件 $\text{TimeFromYear}(y) \leq t$ 的最大整数 y (接近正无穷)

若时间值在闰年内, 闰年函数返回 1, 否则返回 0:

$\text{InLeapYear}(t) = 0$ { 如果 $\text{DaysInYear}(\text{YearFromTime}(t)) = 365$ } $= 1$ { 如果 $\text{DaysInYear}(\text{YearFromTime}(t)) = 366$ }

月数

月份是由闭区间 0 到 11 内的一个整数确定。一个时间值 t 到一个月数的映射 $\text{MonthFromTime}(t)$ 的定义为:

$\text{MonthFromTime}(t) = 0$ if $0 \leq \text{DayWithinYear}(t) < 31 = 1$ if $31 \leq \text{DayWithinYear}(t) < 59 + \text{InLeapYear}(t) = 2$ if $59 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 90 + \text{InLeapYear}(t) = 3$ if $90 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 120 + \text{InLeapYear}(t) = 4$ if $120 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 151 + \text{InLeapYear}(t) = 5$ if $151 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 181 + \text{InLeapYear}(t) = 6$ if $181 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 212 + \text{InLeapYear}(t) = 7$ if $212 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 243 + \text{InLeapYear}(t) = 8$ if $243 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 273 + \text{InLeapYear}(t) = 9$ if $273 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 304 + \text{InLeapYear}(t) = 10$ if $304 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 334 + \text{InLeapYear}(t) = 11$ if $334 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 365 + \text{InLeapYear}(t)$

其中

$\text{DayWithinYear}(t) = \text{Day}(t) - \text{DayFromYear}(\text{YearFromTime}(t))$

月数值 0 指一月; 1 指二月; 2 指三月; 3 指四月; 4 指五月; 5 指六月; 6 指七月; 7 指八月; 8 指九月; 9 指十月; 10 指十一月; 11 指十二月。注: $\text{MonthFromTime}(0) = 0$, 对应 1970 年 1 月 1 日, 星期四。

日期数

一个日期数用闭区间 1 到 31 内的一个整数标识。从一个时间值 t 到一个日期数的映射 $\text{DateFromTime}(t)$ 的定义为:

$\text{DateFromTime}(t) = \text{DayWithinYear}(t) + 1$ if $\text{MonthFromTime}(t) = 0 = \text{DayWithinYear}(t) - 30$ if $\text{MonthFromTime}(t) = 1 = \text{DayWithinYear}(t) - 58 - \text{InLeapYear}(t)$ if $\text{MonthFromTime}(t) = 2 = \text{DayWithinYear}(t) - 89 - \text{InLeapYear}(t)$ if $\text{MonthFromTime}(t) = 3 = \text{DayWithinYear}(t) - 119 - \text{InLeapYear}(t)$ if

```

MonthFromTime(t)=4 = DayWithinYear(t)-150-InLeapYear(t) if
MonthFromTime(t)=5 = DayWithinYear(t)-180-InLeapYear(t) if
MonthFromTime(t)=6 = DayWithinYear(t)-211-InLeapYear(t) if
MonthFromTime(t)=7 = DayWithinYear(t)-242-InLeapYear(t) if
MonthFromTime(t)=8 = DayWithinYear(t)-272-InLeapYear(t) if
MonthFromTime(t)=9 = DayWithinYear(t)-303-InLeapYear(t) if
MonthFromTime(t)=10 = DayWithinYear(t)-333-InLeapYear(t) if
MonthFromTime(t)=11

```

星期数

特定时间值 t 对应的星期数的定义为：

$$\text{WeekDay}(t) = (\text{Day}(t) + 4) \text{ modulo } 7$$

星期数的值 0 指星期日；1 指星期一；2 指星期二；3 指星期三；4 指星期四；5 指星期五；6 指星期六。注：WeekDay(0) = 4，对应 1970 年 1 月 01 日 星期四。

本地时区校准

期望一个 ECMAScript 的实现确定本地时区校准。本地时区校准是一个毫秒为单位的值 LocalTZA，它加上 UTC 代表本地标准时间。LocalTZA 不体现夏令时。LocalTZA 值不随时间改变，但只取决于地理位置。

夏令时校准

期望一个 ECMAScript 的实现确定夏令时算法。确定夏令时校准的算法 DaylightSavingTA(t)，以毫秒为单位，必须只依赖下面四个项目：

(1) 自本年开始以来的时间

$$t - \text{TimeFromYear}(\text{YearFromTime}(t))$$

(2) t 是否在闰年内

$$\text{InLeapYear}(t)$$

(3) 本年第一天的星期数

$$\text{WeekDay}(\text{TimeFromYear}(\text{YearFromTime}(t)))$$

(4) 地理位置。

The implementation of ECMAScript should not try to determine whether the exact time was subject to daylight saving time, but just whether daylight saving time would have been in effect if the current daylight saving time algorithm had been used at the time. This avoids complications such as taking into account the years that the locale observed daylight saving time year round.

If the host environment provides functionality for determining daylight saving time, the implementation of ECMAScript is free to map the year in question to an equivalent year (same leap-year-ness and same starting week day for the year) for which the host environment provides daylight saving time information. The only restriction is that all equivalent years should produce the same result.

本地时间

从协调世界时到本地时间的转换，定义为

$$\text{LocalTime}(t) = t + \text{LocalTZA} + \text{DaylightSavingTA}(t)$$

从本地时间到协调世界时的转换，定义为

$$\text{UTC}(t) = t - \text{LocalTZA} - \text{DaylightSavingTA}(t - \text{LocalTZA})$$

UTC(LocalTime(t)) 不一定总是等于 **t**。

小时，分钟，秒，毫秒

以下函数用于分解时间值：

$$\begin{aligned} \text{HourFromTime}(t) &= \text{floor}(t / \text{msPerHour}) \bmod \text{HoursPerDay} \\ \text{MinFromTime}(t) &= \text{floor}(t / \text{msPerMinute}) \bmod \text{MinutesPerHour} \\ \text{SecFromTime}(t) &= \text{floor}(t / \text{msPerSecond}) \bmod \text{SecondsPerMinute} \\ \text{msFromTime}(t) &= t \bmod \text{msPerSecond} \end{aligned}$$

其中

$$\begin{aligned} \text{HoursPerDay} &= 24 \\ \text{MinutesPerHour} &= 60 \\ \text{SecondsPerMinute} &= 60 \\ \text{msPerSecond} &= 1000 \\ \text{msPerMinute} &= 60000 = \text{msPerSecond} \times \text{SecondsPerMinute} \\ \text{msPerHour} &= 3600000 = \text{msPerMinute} \times \text{MinutesPerHour} \end{aligned}$$

MakeTime (hour, min, sec, ms)

MakeTime 抽象操作使用它的四个参数算出一个毫秒数，参数必须是 ECMAScript 数字值。此抽象操作运行如下：

1. 如果 hour 不是有限的或 min 不是有限的或 sec 不是有限的或 ms 不是有限的，返回 NaN。
2. 令 h 为 `ToInteger(hour)`。
3. 令 m 为 `ToInteger(min)`。
4. 令 s 为 `ToInteger(sec)`。
5. 令 milli 为 `ToInteger(ms)`。
6. 令 t 为 $h * \text{msPerHour} + m * \text{msPerMinute} + s * \text{msPerSecond} + \text{milli}$ ，执行的四则运算根据 IEEE 754 规则（这就像使用 ECMAScript 运算符 * 和 + 一样）。
7. 返回 t。

MakeDay (year, month, date)

MakeDay 抽象操作它的三个参数算出一个天数，参数必须是 ECMAScript 数字值。此抽象操作运行如下：

1. 如果 year 不是有限的或 month 不是有限的或 date 不是有限的，返回 NaN。
2. 令 y 为 `ToInteger(year)`。
3. 令 m 为 `ToInteger(month)`。
4. 令 dt 为 `ToInteger(date)`。
5. 令 ym 为 $y + \text{floor}(m / 12)$ 。
6. 令 mn 为 $m \bmod 12$ 。
7. 找一个满足 `YearFromTime(t) == ym` 且 `MonthFromTime(t) == mn` 且 `DateFromTime(t) == 1` 的 t 值；但如果这些条件是不可能的（因为有些参数超出了范围），返回 NaN。
8. 返回 `Day(t) + dt - 1`。

MakeDate (day, time)

MakeDate 抽象操作它的两个参数算出一个毫秒数，参数必须是 ECMAScript 数字值。此抽象操作运行如下：

1. 如果 day 不是有限的或 time 不是有限的，返回 NaN。
2. 返回 $\text{day} \times \text{msPerDay} + \text{time}$ 。

TimeClip (time)

TimeClip 抽象操作它的参数算出一个毫秒数，参数必须是 ECMAScript 数字值。此抽象操作运行如下：

1. 如果 time 不是有限的，返回 NaN。

2. 如果 `abs(time) > 8.64 x 1015`, 返回 NaN.
3. 返回 `ToInteger(time)` 和 `ToInteger(time) + (+0)` 之一, 这依赖于实现 (加正一是为了将 -0 转换成 +0)。

第 3 步的重点是说允许实现自行选择时间值的内部表示形式, 如 64 位有符号整数或 64 位浮点数。根据不同的实现, 这个内部表示可能区分也可能无法区分 -0 和 +0。

日期时间字符串格式

ECMAScript 定义了一个基于简化的 ISO 8601 扩展格式的日期时间的字符串互换格式, 格式为: `YYYY-MM-DDTHH:mm:ss.sssZ`

其中个字段为:

YYYY

是公历中年的十进制数字。

-

在字符串中直接以“-” (破折号) 出现两次。

MM

是一年中的月份, 从 01 (一月) 到 12 (十二月)。

DD

是月份中的日期, 从 01 到 30。

T

在字符串中直接以“T”出现, 用来表明时间元素的开始。

HH

是用两个十进制数字表示的, 自午夜 0 点以来的小时数。

:

在字符串中直接以“:” (冒号) 出现两次。

mm

是用两个十进制数字表示的, 自小时开始以来的分钟数。

ss

是用两个十进制数字表示的, 自开始以来的秒数。

.

在字符串中直接以“.” (点) 出现。

SSS

是用三个十进制数字表示的，自秒开始以来的毫秒数。

Z

是时区偏移量，由（“Z”（指 UTC）或“+”或“-”）和后面跟着的时间表达式 hh:mm 组成。

这个格式包括只表示日期的形式：

YYYY YYYY-MM YYYY-MM-DD

这个格式还包括“日期时间”形式，它由上面的只表示日期的形式之一和紧跟在后面的“T”和以下时间形式之一和可选的时区偏移量组成：

THH:mm THH:mm:ss THH:mm:ss.sss

所有数字必须是 10 进制的。如果缺少 MM 或 DD 字段 用“01”作为它们的值。如果缺少 mm 或 ss 字段，用“00”作为它们的值，对于缺少的 sss 用“000”作为它的值。对于缺少的时区偏移量用“Z”。

一个格式字符串里有非法值（越界以及语法错误），意味着这个格式字符串不是有效的本节描述格式的实例。

由于每天的开始和结束都在午夜，俩符号 00:00 和 24:00 可区分这样的可以是同一时间的两个午夜。这意味着两个符号 1995-02-04T24:00 和 1995-02-05T00:00 精准的指向同一时刻。

不存在用来规范像 CET, EST 这样的民间时区缩写的国际标准。有时相同的缩写甚至使用不同的时区。出于这个原因，ISO 8601 和这里的格式指定数字来表示时区。

扩展的年

ECMAScript 需要能表示 6 位数年份（扩展的年份）的能力；协调世界时 1970 年 1 月 1 日前后分别约 285,616 年。对于表示 0 年之前或 9999 年之后的年份，ISO 8601 允许对年的表示法进行扩展，但只能在发送和接受信息的双方有事先共同约定的情况下才能扩展。在已经简化的 ECMAScript 的格式中这样扩展的年份表示法有 2 个额外的数字和始终存在的前缀符号 + 或 - 。0 年被认为是正的，因此用 + 符号作为前缀。

扩展年的示例

-283457-03-21T15:00:59.008Z 283458 B. C. -000001-01-01T00:00:00Z 2 B. C.
+000000-01-01T00:00:00Z 1 B. C. +000001-01-01T00:00:00Z 1 A. D.

+001970-01-01T00:00:00Z 1970 A.D. +002009-12-15T00:00:00Z 2009 A.D.
+287396-10-12T08:59:00.992Z 287396 A.D.

作为函数调用 Date 构造器

当把 **Date** 作为函数来调用，而不作为构造器，它返回一个表示当前时间（协调世界时）的字符串。

函数调用 **Date(...)** 的结果和用相同参数调用表达式 **new Date(...)** 创建的对象是不同的。

Date ([year [, month [, date [, hours [, minutes [, seconds [,
ms]]]]]]])

所有参数都是可选的；接受提供的任何参数，但被完全忽略。返回一个仿佛是用表达式 **(new Date()).toString()** 创建的字符串，这里的 **Date** 是标准内置构造器，**toString** 是标准内置方法 **Date.prototype.toString**。

Date 构造器

当把 **Date** 作为 **new** 表达式的一部分来调用，它是个构造器：它初始化新创建的对象。

new Date (year, month [, date [, hours [, minutes [, seconds [,
ms]]]]])

当用二到七个参数调用 **Date** 构造器，它用 **year**, **month**, 还有 (可选的) **date**, **hours**, **minutes**, **seconds**, **ms** 来计算时间。

新构造对象的 **[[Prototype]]** 内部属性设定为原始的时间原型对象，它是 **Date.prototype(15.9.4.1)** 的初始值。

新构造对象的 **[[Class]]** 内部属性设定为 **"Date"**。

新构造对象的 **[[Extensible]]** 内部属性设定为 **true**。

新构造对象的 **[[PrimitiveValue]]** 内部属性按照以下步骤设定：

1. 令 **y** 为 **ToNumber(year)**。
2. 令 **m** 为 **ToNumber(month)**。

3. 如果提供了 `date` , 则令 `dt` 为 `ToNumber(date)`; 否则令 `dt` 为 1.
4. 如果提供了 `hours` , 则令 `h` 为 `ToNumber(hours)`; 否则令 `h` 为 0.
5. 如果提供了 `minutes` , 则令 `min` 为 `ToNumber(minutes)`; 否则令 `min` 为 0.
6. 如果提供了 `seconds` , 则令 `s` 为 `ToNumber(seconds)`; 否则令 `s` 为 0.
7. 如果提供了 `ms` , 则令 `milli` 为 `ToNumber(ms)`; 否则令 `milli` 为 0.
8. 如果 `y` 不是 NaN 且 $0 \leq \text{ToInteger}(y) \leq 99$, 则令 `yr` 为 `1900+ToInteger(y)`; 否则令 `yr` 为 `y`.
9. 令 `finalDate` 为 `MakeDate(MakeDay(yr, m, dt), MakeTime(h, min, s, milli))`.
10. 设定新构造对象的 `[[PrimitiveValue]]` 内部属性为 `TimeClip(UTC(finalDate))`.

new Date (value)

新构造对象的 `[[Prototype]]` 内部属性设定为原始的时间原型对象, 它是 `Date.prototype(15.9.4.1)` 的初始值。

新构造对象的 `[[Class]]` 内部属性设定为 "Date"。

新构造对象的 `[[Extensible]]` 内部属性设定为 `true`。

新构造对象的 `[[PrimitiveValue]]` 内部属性按照以下步骤设定:

1. 令 `v` 为 `ToPrimitive(value)`.
2. 如果 `Type(v)` 是 `String`, 则
 1. 用与 `parse` 方法 (15.9.4.2) 完全相同的方式将 `v` 解析为一个日期时间; 令 `V` 为这个日期时间的时间值。
3. 否则, 令 `V` 为 `ToNumber(v)`.
4. 设定新构造对象的 `[[PrimitiveValue]]` 内部属性为 `TimeClip(V)`, 并返回这个值。

new Date ()

新构造对象的 `[[Prototype]]` 内部属性设定为原始的时间原型对象, 它是 `Date.prototype(15.9.4.1)` 的初始值。

新构造对象的 `[[Class]]` 内部属性设定为 "Date"。

新构造对象的 `[[Extensible]]` 内部属性设定为 `true`。

新构造对象的 `[[PrimitiveValue]]` 内部属性设定为表示当前时间的时间值 (协调世界时)。

Date 构造器的属性

Date 构造器的 `[[Prototype]]` 内部属性的值是函数原型对象 (15.3.4)。

除了内部属性和 `length` 属性 (值为 7) 之外, Date 构造器还有以下属性:

Date.prototype

Date.prototype 的初始值是内置的 Date 原型对象 (15.9.5)。

此属性有特性 { `[[Writable]]`: false, `[[Enumerable]]`: false, `[[Configurable]]`: false }。

Date.parse (string)

`parse` 函数对它的参数应用 `ToString` 操作并将结果字符串解释为一个日期和时间; 返回一个数字值, 是对应这个日期时间的 UTC 时间值。字符串可解释为本地时间, UTC 时间, 或某个其他时区的时间, 这取决于字符串里的内容。此函数首先尝试根据日期时间字符串格式 (15.9.1.15) 里的规则来解析字符串的格式。如果字符串不符合这个格式此函数可回退, 用任意实现定义的试探方式或日期格式。无法识别的字符串或日期时间包含非法元素值, 将导致 `Date.parse` 返回 NaN。

在所有属性都指向它们的初始值的情况下, 如果 `x` 是一个在特定 ECMAScript 的实现里的毫秒数为零的任意 Date 对象, 则在这个实现中以下所有表达式应产生相同数字值:

```
x.valueOf()Date.parse(x.toString())Date.parse(x.toUTCString())
Date.parse(x.toISOString())
```

然而, 表达式

Date.parse(x.toLocaleString())

是不需要产生与前面三个表达参数相同的数字值。通常, 在给定的字符串不符合日期时间字符串格式 (15.9.1.15) 时, `Date.parse` 的产生值是依赖于实现, 并且在同一实现中 `toString` 或 `toUTCString` 方法不能产生不符合日期时间字符串格式的字符串。

Date.UTC (year, month [, date [, hours [, minutes [, seconds [, ms]]]]])

当用少于两个的参数调用 UTC 函数时，它的行为是依赖于实现的。当用二到七个参数调用 UTC 函数，它从 year, month 和 (可选的) date, hours, minutes, seconds, ms 计算出日期时间。采用以下步骤：

1. 令 y 为 ToNumber(year).
2. 令 m 为 ToNumber(month).
3. 如果提供了 date ，则令 dt 为 ToNumber(date); 否则令 dt 为 1.
4. 如果提供了 hours ，则令 h 为 ToNumber(hours); 否则令 h 为 0.
5. 如果提供了 minutes ，则令 min 为 ToNumber(minutes); 否则令 min 为 0.
6. 如果提供了 seconds ，则令 s 为 ToNumber(seconds); 否则令 s 为 0.
7. 如果提供了 ms ，则令 milli 为 ToNumber(ms); 否则令 milli 为 0.
8. 如果 y 不是 NaN 且 $0 \leq \text{ToInteger}(y) \leq 99$ ，则令 yr 为 $1900 + \text{ToInteger}(y)$; 否则令 yr 为 y.
9. 返回 TimeClip(MakeDate(MakeDay(yr, m, dt), MakeTime(h, min, s, milli))).

UTC 函数的 length 属性是 7。

UTC 函数与 Date 构造器的不同点有：它返回一个时间值，而不是创建 Date 对象，还有它将参数解释为 UTC，而不是本地时间。

Date.now ()

now 函数返回一个数字值，它表示调用 now 时的 UTC 日期时间的时间值。

Date 原型对象的属性

Date 原型对象自身是一个 Date 对象（其 [[Class]] 是 "Date"），其 [[PrimitiveValue]] 是 NaN。

Date 原型对象的 [[Prototype]] 内部属性的值是标准内置 Object 原型对象 (15.2.4)。

在以下对 Date 原型对象的函数属性的描述中，短语“this Date 对象”指调用函数时的 this 值对象。除非另外说明，这些函数不是通用的；如果 this 值不是 [[Class]] 内部属性为 "Date" 的对象，则抛出一个 TypeError 异常。短语“this

时间值”指代表 `this Date` 对象的时间值的数字值，它是 `this Date` 对象的 `[[PrimitiveValue]]` 内部属性的值。

Date.prototype.constructor

`Date.prototype.constructor` 的初始值是内置 `Date` 构造器。

Date.prototype.toString ()

此函数返回一个字符串值。字符串中内容是依赖于实现的，但目的是用一种方便，人类可读的形式表示当前时区的时间。

对毫秒数为零的任意 `Date` 值 `d`，`Date.parse(d.toString())` 和 `d.valueOf()` 的结果相同。见 15.9.4.2

Date.prototype.toDateString ()

此函数返回一个字符串值。字符串中内容是依赖于实现的，但目的是用一种方便，人类可读的形式表示当前时区时间的“日期”部分。

Date.prototype.toTimeString ()

此函数返回一个字符串值。字符串中内容是依赖于实现的，但目的是用一种方便，人类可读的形式表示当前时区时间的“时间”部分。

Date.prototype.toLocaleString ()

此函数返回一个字符串值。字符串中内容是依赖于实现的，但目的是用一种 -- 对应宿主环境的当前语言环境设定的 -- 方便，人类可读的形式表示当前时区的时间。

这个函数的第一个参数可能会在此标准的未来版本中使用到；因此建议实现不要以任何目的使用这个位置参数。

Date.prototype.toLocaleDateString ()

此函数返回一个字符串值。字符串中内容是依赖于实现的，但目的是用一种 -- 对应宿主环境的当前语言环境设定的 -- 方便，人类可读的形式表示当前时区时间的“日期”部分。

这个函数的第一个参数可能会在此标准的未来版本中使用到; 因此建议实现不要以任何目的使用这个位置参数。

Date.prototype.toLocaleTimeString ()

此函数返回一个字符串值。字符串中内容是依赖于实现的, 但目的是用一种 -- 对应宿主环境的当前语言环境设定的 -- 方便, 人类可读的形式表示当前时区时间的“时间”部分。

这个函数的第一个参数可能会在此标准的未来版本中使用到; 因此建议实现不要以任何目的使用这个位置参数。

Date.prototype.valueOf ()

valueOf 函数返回一个数字值, 它是 this 时间值。

Date.prototype.getTime ()

1. 返回 this 时间值。

Date.prototype.getFullYear ()

1. 令 t 为 this 时间值 .
2. 如果 t 是 NaN, 返回 NaN.
3. 返回 YearFromTime(LocalTime(t)).

Date.prototype.getUTCFullYear ()

1. 令 t 为 this 时间值 .
2. 如果 t 是 NaN, 返回 NaN.
3. 返回 YearFromTime(t).

Date.prototype.getMonth ()

1. 令 t 为 this 时间值 .
2. 如果 t 是 NaN, 返回 NaN.
3. 返回 MonthFromTime(LocalTime(t)).

Date.prototype.getUTCMonth ()

1. 令 t 为 this 时间值 .
2. 如果 t 是 NaN, 返回 NaN.
3. 返回 MonthFromTime(t).

Date.prototype.getDate ()

1. 令 t 为 this 时间值 .
2. 如果 t 是 NaN, 返回 NaN.
3. 返回 DateFromTime(LocalTime(t)).

Date.prototype.getUTCDate ()

1. 令 t 为 this 时间值 .
2. 如果 t 是 NaN, 返回 NaN.
3. 返回 DateFromTime(t).

Date.prototype.getDay ()

1. 令 t 为 this 时间值 .
2. 如果 t 是 NaN, 返回 NaN.
3. 返回 WeekDay(LocalTime(t)).

Date.prototype.getUTCDay ()

1. 令 t 为 this 时间值 .
2. 如果 t 是 NaN, 返回 NaN.
3. 返回 WeekDay(t).

Date.prototype.getHours ()

1. 令 t 为 this 时间值 .
2. 如果 t 是 NaN, 返回 NaN.
3. 返回 HourFromTime(LocalTime(t)).

Date.prototype.getUTCHours ()

1. 令 t 为 this 时间值 .
2. 如果 t 是 NaN, 返回 NaN.
3. 返回 HourFromTime(t).

Date.prototype.getMinutes ()

1. 令 *t* 为 *this* 时间值 .
2. 如果 *t* 是 NaN, 返回 NaN.
3. 返回 `MinFromTime(LocalTime(t))`.

Date.prototype.getUTCMinutes ()

1. 令 *t* 为 *this* 时间值 .
2. 如果 *t* 是 NaN, 返回 NaN.
3. 返回 `MinFromTime(t)`.

Date.prototype.getSeconds ()

1. 令 *t* 为 *this* 时间值 .
2. 如果 *t* 是 NaN, 返回 NaN.
3. 返回 `SecFromTime(LocalTime(t))`.

Date.prototype.getUTCSeconds ()

1. 令 *t* 为 *this* 时间值 .
2. 如果 *t* 是 NaN, 返回 NaN.
3. 返回 `SecFromTime(t)`.

Date.prototype.getMilliseconds ()

1. 令 *t* 为 *this* 时间值 .
2. 如果 *t* 是 NaN, 返回 NaN.
3. 返回 `msFromTime(LocalTime(t))`.

Date.prototype.getUTCMilliseconds ()

1. 令 *t* 为 *this* 时间值 .
2. 如果 *t* 是 NaN, 返回 NaN.
3. 返回 `msFromTime(t)`.

Date.prototype.getTimezoneOffset ()

返回本地时间和 UTC 时间之间相差的分钟数。

1. 令 `t` 为 `this` 时间值 .
2. 如果 `t` 是 `NaN`, 返回 `NaN`.
3. 返回 $(t - \text{LocalTime}(t)) / \text{msPerMinute}$.

Date.prototype.setTime (time)

1. 令 `v` 为 `TimeClip(ToNumber(time))`.
2. 设定 `this Date` 对象的 `[[PrimitiveValue]]` 内部属性为 `v`.
3. 返回 `v`.

Date.prototype.setMilliseconds (ms)

1. 令 `t` 为 `LocalTime(this 时间值)` 的结果 .
2. 令 `time` 为 `MakeTime(HourFromTime(t), MinFromTime(t), SecFromTime(t), ToNumber(ms))`.
3. 令 `u` 为 `TimeClip(UTC(MakeDate(Day(t), time)))`.
4. 设定 `this Date` 对象的 `[[PrimitiveValue]]` 内部属性为 `u`.
5. 返回 `u`.

Date.prototype.setUTCMilliseconds (ms)

1. 令 `t` 为 `this` 时间值 .
2. 令 `time` 为 `MakeTime(HourFromTime(t), MinFromTime(t), SecFromTime(t), ToNumber(ms))`.
3. 令 `v` 为 `TimeClip(MakeDate(Day(t), time))`.
4. 设定 `this Date` 对象的 `[[PrimitiveValue]]` 内部属性为 `v`.
5. 返回 `v`.

Date.prototype.setSeconds (sec [, ms])

没指定 `ms` 参数时的行为是, 仿佛 `ms` 被指定为调用 `getMilliseconds()` 的结果一样。

1. 令 `t` 为 `LocalTime(this 时间值)` 的结果 .
2. 令 `s` 为 `ToNumber(sec)`.
3. 如果没指定 `ms`, 则令 `milli` 为 `msFromTime(t)`; 否则, 令 `milli` 为 `ToNumber(ms)`.
4. 令 `date` 为 `MakeDate(Day(t), MakeTime(HourFromTime(t), MinFromTime(t), s, milli))`.
5. 令 `u` 为 `TimeClip(UTC(date))`.
6. 设定 `this Date` 对象的 `[[PrimitiveValue]]` 内部属性为 `u`.

7. 返回 u.

setSeconds 方法的 length 属性是 2。

Date.prototype.setUTCSeconds (sec [, ms])

没指定 ms 参数时的行为是，仿佛 ms 被指定为调用 getUTCMilliseconds() 的结果一样。

1. 令 t 为 this 时间值 .
2. 令 s 为 ToNumber(sec).
3. 如果没指定 ms , 则令 milli 为 msFromTime(t); 否则, 令 milli 为 ToNumber(ms).
4. 令 date 为 MakeDate(Day(t), MakeTime(HourFromTime(t), MinFromTime(t), s, milli)).
5. 令 v 为 TimeClip(date).
6. 设定 this Date 对象的 [[PrimitiveValue]] 内部属性为 v.
7. 返回 v.

setUTCSeconds 方法的 length 属性是 2。

Date.prototype.setMinutes (min [, sec [, ms]])

没指定 sec 参数时的行为是，仿佛 ms 被指定为调用 getSeconds() 的结果一样。

没指定 ms 参数时的行为是，仿佛 ms 被指定为调用 getMilliseconds() 的结果一样。

1. 令 t 为 LocalTime(this 时间值) 的结果 .
2. 令 m 为 ToNumber(min).
3. 如果没指定 sec , 则令 s 为 SecFromTime(t); 否则 , 令 s 为 ToNumber(sec).
4. 如果没指定 ms , 则令 milli 为 msFromTime(t); 否则 , 令 milli 为 ToNumber(ms).
5. 令 date 为 MakeDate(Day(t), MakeTime(HourFromTime(t), m, s, milli)).
6. 令 u 为 TimeClip(UTC(date)).
7. 设定 this Date 对象的 [[PrimitiveValue]] 内部属性为 u.
8. 返回 u.

setMinutes 方法的 length 属性是 3。

Date.prototype.setUTCMinutes (min [, sec [, ms]])

没指定 `sec` 参数时的行为是，仿佛 `ms` 被指定为调用 `getUTCSeconds()` 的结果一样。

没指定 `ms` 参数时的行为是，仿佛 `ms` 被指定为调用 `getUTCMilliseconds()` 的结果一样。

1. 令 `t` 为 `this` 时间值。
2. 令 `m` 为 `ToNumber(min)`。
3. 如果没指定 `sec`，则令 `s` 为 `SecFromTime(t)`；否则，令 `s` 为 `ToNumber(sec)`。
4. 如果没指定 `ms`，则令 `milli` 为 `msFromTime(t)`；否则，令 `milli` 为 `ToNumber(ms)`。
5. 令 `date` 为 `MakeDate(Day(t), MakeTime(HourFromTime(t), m, s, milli))`。
6. 令 `v` 为 `TimeClip(date)`。
7. 设定 `this Date` 对象的 `[[PrimitiveValue]]` 内部属性为 `v`。
8. 返回 `v`。

`setUTCMinutes` 方法的 `length` 属性是 3。

Date.prototype.setHours (hour [, min [, sec [, ms]]])

没指定 `min` 参数时的行为是，仿佛 `min` 被指定为调用 `getMinutes()` 的结果一样。

没指定 `sec` 参数时的行为是，仿佛 `ms` 被指定为调用 `getSeconds()` 的结果一样。

没指定 `ms` 参数时的行为是，仿佛 `ms` 被指定为调用 `getMilliseconds()` 的结果一样。

1. 令 `t` 为 `LocalTime(this 时间值)` 的结果。
2. 令 `h` 为 `ToNumber(hour)`。
3. 如果没指定 `min`，则令 `m` 为 `MinFromTime(t)`；否则，令 `m` 为 `ToNumber(min)`。
4. 如果没指定 `sec`，则令 `s` 为 `SecFromTime(t)`；否则，令 `s` 为 `ToNumber(sec)`。
5. 如果没指定 `ms`，则令 `milli` 为 `msFromTime(t)`；否则，令 `milli` 为 `ToNumber(ms)`。
6. 令 `date` 为 `MakeDate(Day(t), MakeTime(h, m, s, milli))`。
7. 令 `u` 为 `TimeClip(UTC(date))`。
8. 设定 `this Date` 对象的 `[[PrimitiveValue]]` 内部属性为 `u`。
9. 返回 `u`。

`setHours` 方法的 `length` 属性是 4。

Date.prototype.setUTCHours (hour [, min [, sec [, ms]]])

没指定 `min` 参数时的行为是，仿佛 `min` 被指定为调用 `getUTCMinutes()` 的结果一样。

没指定 `sec` 参数时的行为是，仿佛 `ms` 被指定为调用 `getUTCSeconds()` 的结果一样。

没指定 `ms` 参数时的行为是，仿佛 `ms` 被指定为调用 `getUTCMilliseconds()` 的结果一样。

1. 令 `t` 为 `this` 时间值。
2. 令 `h` 为 `ToNumber(hour)`。
3. 如果没指定 `min`，则令 `m` 为 `MinFromTime(t)`；否则，令 `m` 为 `ToNumber(min)`。
4. 如果没指定 `sec`，则令 `s` 为 `SecFromTime(t)`；否则，令 `s` 为 `ToNumber(sec)`。
5. 如果没指定 `ms`，则令 `milli` 为 `msFromTime(t)`；否则，令 `milli` 为 `ToNumber(ms)`。
6. 令 `date` 为 `MakeDate(Day(t), MakeTime(h, m, s, milli))`。
7. 令 `v` 为 `TimeClip(date)`。
8. 设定 `this Date` 对象的 `[[PrimitiveValue]]` 内部属性为 `v`。
9. 返回 `v`。

`setUTCHours` 方法的 `length` 属性是 4。

Date.prototype.setDate (date)

1. 令 `t` 为 `LocalTime(this 时间值)` 的结果。
2. 令 `dt` 为 `ToNumber(date)`。
3. 令 `newDate` 为 `MakeDate(MakeDay(YearFromTime(t), MonthFromTime(t), dt), TimeWithinDay(t))`。
4. 令 `u` 为 `TimeClip(UTC(newDate))`。
5. 设定 `this Date` 对象的 `[[PrimitiveValue]]` 内部属性为 `u`。
6. 返回 `u`。

Date.prototype.setUTCDate (date)

1. 令 `t` 为 `this` 时间值。
2. 令 `dt` 为 `ToNumber(date)`。
3. 令 `newDate` 为 `MakeDate(MakeDay(YearFromTime(t), MonthFromTime(t), dt), TimeWithinDay(t))`。

4. 令 `v` 为 `TimeClip(newDate)`.
5. 设定 `this Date` 对象的 `[[PrimitiveValue]]` 内部属性为 `v`.
6. 返回 `v`.

Date.prototype.setMonth (month [, date])

没指定 `date` 参数时的行为是, 仿佛 `ms` 被指定为调用 `getDate()` 的结果一样。

1. 令 `t` 为 `LocalTime(this 时间值)` 的结果。
2. 令 `m` 为 `ToNumber(month)`.
3. 如果没指定 `date`, 则令 `dt` 为 `DateFromTime(t)`; 否则, 令 `dt` 为 `ToNumber(date)`.
4. 令 `newDate` 为 `MakeDate(MakeDay(YearFromTime(t), m, dt), TimeWithinDay(t))`.
5. 令 `u` 为 `TimeClip(UTC(newDate))`.
6. 设定 `this Date` 对象的 `[[PrimitiveValue]]` 内部属性为 `u`.
7. 返回 `u`.

`setMonth` 方法的 `length` 属性是 2。

Date.prototype.setUTCMonth (month [, date])

没指定 `date` 参数时的行为是, 仿佛 `ms` 被指定为调用 `getUTCDate()` 的结果一样。

1. 令 `t` 为 `this` 时间值。
2. 令 `m` 为 `ToNumber(month)`.
3. 如果没指定 `date`, 则令 `dt` 为 `DateFromTime(t)`; 否则, 令 `dt` 为 `ToNumber(date)`.
4. 令 `newDate` 为 `MakeDate(MakeDay(YearFromTime(t), m, dt), TimeWithinDay(t))`.
5. 令 `v` 为 `TimeClip(newDate)`.
6. 设定 `this Date` 对象的 `[[PrimitiveValue]]` 内部属性为 `v`.
7. 返回 `v`.

`setUTCMonth` 方法的 `length` 属性是 2。

Date.prototype.setFullYear (year [, month [, date]])

没指定 `month` 参数时的行为是, 仿佛 `ms` 被指定为调用 `getMonth()` 的结果一样。

没指定 `date` 参数时的行为是，仿佛 `ms` 被指定为调用 `getDate()` 的结果一样。

1. 令 `t` 为 `LocalTime(this 时间值)` 的结果；但如果 `this` 时间值是 `NaN`，则令 `t` 为 `+0`。
2. 令 `y` 为 `ToNumber(year)`。
3. 如果没指定 `month`，则令 `m` 为 `MonthFromTime(t)`；否则，令 `m` 为 `ToNumber(month)`。
4. 如果没指定 `date`，则令 `dt` 为 `DateFromTime(t)`；否则，令 `dt` 为 `ToNumber(date)`。
5. 令 `newDate` 为 `MakeDate(MakeDay(y, m, dt), TimeWithinDay(t))`。
6. 令 `u` 为 `TimeClip(UTC(newDate))`。
7. 设定 `this Date` 对象的 `[[PrimitiveValue]]` 内部属性为 `u`。
8. 返回 `u`。

`setFullYear` 方法的 `length` 属性是 3。

Date.prototype.setUTCFullYear (year [, month [, date]])

没指定 `month` 参数时的行为是，仿佛 `ms` 被指定为调用 `getUTCMonth()` 的结果一样。

没指定 `date` 参数时的行为是，仿佛 `ms` 被指定为调用 `getUTCDate()` 的结果一样。

1. 令 `t` 为 `this` 时间值；但如果 `this` 时间值是 `NaN`，则令 `t` 为 `+0`。
2. 令 `y` 为 `ToNumber(year)`。
3. 如果没指定 `month`，则令 `m` 为 `MonthFromTime(t)`；否则，令 `m` 为 `ToNumber(month)`。
4. 如果没指定 `date`，则令 `dt` 为 `DateFromTime(t)`；否则，令 `dt` 为 `ToNumber(date)`。
5. 令 `newDate` 为 `MakeDate(MakeDay(y, m, dt), TimeWithinDay(t))`。
6. 令 `v` 为 `TimeClip(newDate)`。
7. 设定 `this Date` 对象的 `[[PrimitiveValue]]` 内部属性为 `v`。
8. 返回 `v`。

`setUTCFullYear` 方法的 `length` 属性是 3。

Date.prototype.toUTCString ()

此函数返回一个字符串值。字符串中内容是依赖于实现的，但目的是用一种方便，人类可读的形式表示 UTC 时间。

此函数的目的是为日期时间产生一个比 15.9.1.15 指定的格式更易读的字符串表示。没必要选择明确的或易于机器解析的格式。如果一个实现没有一个首选的人类可读格式，建议使用 15.9.1.15 定义的格式，但用空格而不是“T”分割日期和时间元素。

Date.prototype.toISOString ()

此函数返回一个代表 --this Date 对象表示的时间的实例 -- 的字符串。字符串的格式是 15.9.1.15 定义的日期时间字符串格式。字符串中包含所有的字段。字符串表示的时区总是 UTC，用后缀 Z 标记。如果 this 对象的时间值不是有限的数字值，抛出一个 RangeError 异常。

Date.prototype.toJSON (key)

此函数为 JSON.stringify (15.12.3) 提供 Date 对象的一个字符串表示。

当用参数 key 调用 toJSON 方法，采用以下步骤：

1. 令 O 为 以 this 值为参数调用 toObject 的结果。
2. 令 tv 为 ToPrimitive(O, hint Number)。
3. 如果 tv 是一个数字值且不是有限的，返回 null。
4. 令 toISO 为以 "toISOString" 为参数调用 O 的 [[Get]] 内部方法的结果。
5. 如果 IsCallable(toISO) 是 false，抛出一个 TypeError 异常。
6. O 作为以 this 值并用空参数列表调用 toISO 的 [[Call]] 内部方法，返回结果。

参数是被忽略的。

toJSON 函数是故意设计成通用的；它不需要其 this 值必须是一个 Date 对象。因此，它可以作为方法转移到其他类型的对象上。但转移到的对象必须有 toISOString 方法。对象可自由使用参数 key 来过滤字符串化的方式。

Date 实例的属性

Date 实例从 Date 原型对象继承属性，Date 实例的 [[Class]] 内部属性值是 "Date"。Date 实例还有一个 [[PrimitiveValue]] 内部属性。

[[PrimitiveValue]] 内部属性是代表 this Date 对象的时间值。

RegExp (正则表达式) 对象

一个 RegExp 对象包含一个正则表达式和关联的标志。

正则表达式的格式和功能是以 **Perl 5** 程序语言的正则表达式设施为蓝本的。

模式

RegExp 构造器对输入模式字符串应用以下文法。如果文法无法将字符串解释为 **Pattern** 的一个展开形式，则发生错误。

语法：

```
Pattern ::  
DisjunctionDisjunction ::  
Alternative  
Alternative | DisjunctionAlternative ::  
[empty]  
Alternative TermTerm ::  
Assertion  
Atom  
Atom QuantifierAssertion ::  
^  
$  
\ b  
\ B  
( ? = Disjunction )  
( ? ! Disjunction )Quantifier ::  
QuantifierPrefix  
QuantifierPrefix ?QuantifierPrefix ::  
*  
+ ?  
{ DecimalDigits }
```

```

{ DecimalDigits , }
{ DecimalDigits , DecimalDigits }Atom ::
PatternCharacter
.
\ AtomEscape
CharacterClass
( Disjunction )
( ? : Disjunction )PatternCharacter :: SourceCharacter but
not any of:
^ $ \ . * + ? ( ) [ ] { } |AtomEscape ::
DecimalEscape
CharacterEscape
CharacterClassEscapeCharacterEscape ::
ControlEscape
c ControlLetter
HexEscapeSequence
UnicodeEscapeSequence
IdentityEscape ControlEscape :: one of
f n r t v ControlLetter :: one of
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y ZIdentityEscape ::
SourceCharacter but not IdentifierPart

DecimalEscape ::
DecimalIntegerLiteral [lookahead ∉ DecimalDigit]
CharacterClassEscape :: one of

```



```

d D s S w WCharacterClass ::
[ [lookahead ∉ {^}] ClassRanges ]
[ ^ ClassRanges ]ClassRanges ::
[empty]
NonemptyClassRangesNonemptyClassRanges ::
ClassAtom
ClassAtom NonemptyClassRangesNoDash
ClassAtom - ClassAtom
ClassRangesNonemptyClassRangesNoDash ::
ClassAtom
ClassAtomNoDash NonemptyClassRangesNoDash
ClassAtomNoDash - ClassAtom ClassRangesClassAtom ::
-
ClassAtomNoDashClassAtomNoDash ::
SourceCharacter but not one of \ or ] or -
\ ClassEscapeClassEscape ::
DecimalEscape
b
CharacterEscape
CharacterClassEscape

```

模式语义

使用下面描述的过程来将一个正则表达式模式转换为一个内部程序。实现使用比下面列出的算法跟高效的算法是被鼓励的，只要结果是相同的。内部程序用作 `RegExp` 对象的 `[[Match]]` 内部属性的值。

表示法

后面的描述用到以下变量：

- **input**，是正则表达式模式要匹配的字符串。符号 **input[n]** 表示 **input** 的第 **n** 个字符，这里的 **n** 可以是 0(包括) 和 **InputLength**(不包括) 之间的。
- **InputLength**，是 **input** 字符串里的字符数目。
- **NcapturingParens**，是在模式中左捕获括号的总数 (即，**Atom :: (Disjunction)** 产生式被展开的总次数)。一个左捕获括号是匹配产生式 **Atom :: (Disjunction)** 中的 终结符 (的任意 (模式字符。
- **IgnoreCase**，是 **RegExp** 对象的 **ignoreCase** 属性的设定值。
- **Multiline**，是 **RegExp** 对象的 **multiline** 属性的设定值。

此外，后面的描述用到以下内部数据结构：

- **CharSet**，是字符的一个数学上的集合。
- **State**，是一个有序对 (**endIndex**, **captures**)，这里 **endIndex** 是一个整数，**captures** 是有 **NcapturingParens** 个值的内部数组。**States** 用来表示正则表达式匹配算法里的局部匹配状态。**endIndex** 是到目前为止模式匹配的最后一个输入字符的索引值加上一，而 **captures** 持有捕获括号的捕获结果。**captures** 的第 **n** 个元素是一个代表第 **n** 个捕获括号对捕获值的字符串，或如果第 **n** 个捕获括号对未能达到目的，**captures** 的第 **n** 个元素是 **undefined**。由于回溯，很多 **States** 可能在匹配过程中的任何时候被使用。
- **MatchResult**，值为 **State** 或表示匹配失败特殊 **token--failure**。
- **Continuation** 程序，是一个内部闭包 (即，一些参数已经绑定了值的内部程序)，它用一个 **State** 参数返回一个 **MatchResult** 结果。如果一个内部闭包引用的变量是绑定在创建这个闭包的函数里，则闭包使用在创建闭包时的这些变量值。**Continuation** 尝试从其 **State** 参数给定的中间状态开始用模式的其余部分 (由闭包的已绑定参数指定) 匹配输入字符串。如果匹配成功，**Continuation** 返回最终的 **State**；如果匹配失败，**Continuation** 返回 **failure**。
- **Matcher** 程序，是一个需要两个参数 -- 一个 **State** 和一个 **Continuation** -- 的内部闭包，它返回一个 **MatchResult** 结果。**Matcher** 尝试从其 **State** 参数给定的中间状态开始用模式的一个中间子模式 (由闭包的已绑定参数指定) 匹配输入字符串。**Continuation** 参数是去匹配模式中剩余部分的闭包。用模式的子模式匹配之后获得一个新 **State**，之后 **Matcher** 用新 **State** 去调用 **Continuation** 来测试模式的剩余部分是否能匹配成功。如果匹配成功，**matcher** 返回 **Continuation** 返回的 **State**；如果匹配失败，**Matcher** 尝试用不同的可选位置重复调用 **Continuation**，直到 **Continuation** 匹配成功或用尽所有的可选位置。
- **AssertionTester** 程序，是需要一个 **State** 参数并返回一个布尔结果的内部闭包。**AssertionTester** 测试输入字符串的当前位置是否满足一个特定条件 (由闭包的已绑定参数指定)，如果匹配了条件，返回 **true**；如果不匹配，返回 **false**。
- **EscapeValue**，是一个字符或一个整数。**EscapeValue** 用来表示 **DecimalEscape** 转移序列的解释结果：一个字符 **ch** 在转义序列里时，它

被解释为字符 `ch`；而一个整数 `n` 在转义序列里时，它被解释为对第 `n` 个捕获括号组的反响引用。

模式 (Pattern)

产生式 `Pattern :: Disjunction` 按照以下方式解释执行：

1. 解释执行 `Disjunction`，获得一个 `Matcher m`。
2. 返回一个需要两个参数的内部闭包，一个字符串 `str` 和一个整数 `index`，执行方式如下：
 1. 令 `Input` 为给定的字符串 `str`。15.10.2 中的算法都将用到此变量。
 2. 令 `InputLength` 为 `Input` 的长度。15.10.2 中的算法都将用到此变量。
 3. 令 `c` 为一个 `Continuation`，它始终对它的任何 `State` 参数都返回成功匹配的 `MatchResult`。
 4. 令 `cap` 为一个有 `NcapturingParens` 个 `undefined` 值的内部数组，索引是从 1 到 `NcapturingParens`。
 5. 令 `x` 为 `State (index, cap)`。
 6. 调用 `m(x, c)`，并返回结果。

一个模式解释执行（“编译”）为一个内部程序值。`RegExp.prototype.exec` 可将这个内部程序应用于一个字符串和字符串的一个偏移位，来确定从这个偏移位开始，模式是否能够匹配，如果能匹配，将返回捕获括号的值。15.10.2 中的算法被设计为只在编译一个模式时可抛出一个 `SyntaxError` 异常；反过来说，一旦模式编译成功，应用编译生成的内部程序在字符串中寻找匹配结果时不可抛出异常（除非是宿主定义的可在任何时候出现的异常，如内存不足）。

析取 (Disjunction)

产生式 `Disjunction :: Alternative` 的解释执行，是解释执行 `Alternative` 来获得 `Matcher` 并返回这个 `Matcher`。

产生式 `Disjunction :: Alternative | Disjunction` 按照以下方式解释执行：

1. 解释执行 `Alternative` 来获得一个 `Matcher m1`。
2. 解释执行 `Disjunction` 来获得一个 `Matcher m2`。
3. 返回一个需要两个参数的内部闭包 `Matcher`，参数分别是一个 `State x` 和一个 `Continuation c`，此内部闭包的执行方式如下：
 1. 调用 `m1(x, c)` 并令 `r` 为其结果。
 2. 如果 `r` 不是 `failure`，返回 `r`。
 3. 调用 `m2(x, c)` 并返回其结果。

正则表达式运算符 `|` 用来分隔两个选择项。模式首先尝试去匹配左侧的 `Alternative`（紧跟着是正则表达式的后续匹配结果）；如果失败，尝试匹配右侧

的 **Disjunction** (紧跟着是正则表达式的后续匹配结果)。如果左侧的 **Alternative**, 右侧的 **Disjunction**, 还有后续匹配结果, 全都有可选的匹配位置, 则后续匹配结果的所有可选位置是在左侧的 **Alternative** 移动到下一个可选位置之前确定的。如果左侧 **Alternative** 的可选位置被用尽了, 右侧 **Disjunction** 试图替代左侧 **Alternative**。一个模式中任何被 | 跳过的捕获括号参数 **undefined** 值还代替字符串。因此, 如:

```
/a|ab/.exec("abc")
```

返回结果是 "a", 而不是 "ab"。此外

```
/((a)|(ab))((c)|(bc))/.exec("abc")
```

返回的数组是

```
["abc", "a", "a", undefined, "bc", undefined, "bc"]
```

而不是

```
["abc", "ab", undefined, "ab", "c", "c", undefined]
```

选择项 (Alternative)

产生式 **Alternative :: [empty]** 解释执行返回一个 **Matcher**, 它需要两个参数, 一个 **State x** 和 一个 **Continuation c**, 并返回调用 **c(x)** 的结果。

产生式 **Alternative :: Alternative Term** 按照如下方式解释执行:

1. 解释执行 **Alternative** 来获得一个 **Matcher m1**.
2. 解释执行 **Term** 来获得一个 **Matcher m2**.
3. 返回一个内部闭包 **Matcher**, 它需要两个参数, 一个 **State x** 和一个 **Continuation c**, 执行方式如下:
 1. 创建一个 **Continuation d**, 它需要一个 **State** 参数 **y**, 返回调用 **m2(y, c)** 的结果.
 2. 调用 **m1(x, d)** 并返回结果.

连续的 **Term** 试着同时去匹配连续输入字符串的连续部分。如果左侧的 **Alternative**, 右侧的 **Term**, 还有后续匹配结果, 全都有可选的匹配位置, 则后续匹配结果的所有可选位置是在右侧的 **Term** 移动到下一个可选位置之前确定的, 并且则右侧的 **Term** 的所有可选位置是在左侧的 **Alternative** 移动到下一个可选位置之前确定的。

匹配项 (Term)

产生式 `Term :: Assertion` 解释执行，返回一个需要两个参数 `State x` 和 `Continuation c` 的内部闭包 `Matcher`，它的执行方式如下：

1. 解释执行 `Assertion` 来获得一个 `AssertionTester t`.
2. 调用 `t(x)` 并令 `r` 为调用结果布尔值 .
3. 如果 `r` 是 `false`, 返回 `failure`.
4. 调用 `c(x)` 并返回结果 .

产生式 `Term :: Atom` 的解释执行方式是，解释执行 `Atom` 来获得一个 `Matcher` 并返回这个 `Matcher`。

产生式 `Term :: Atom Quantifier` 的解释执行方式如下：

1. 解释执行 `Atom` 来获得一个 `Matcher m`.
2. 解释执行 `Quantifier` 来获得三个结果值：一个整数 `min`，一个整数（或 ∞ ）`max`，和一个布尔值 `greedy`.
3. 如果 `max` 是有限的且小于 `min`，则抛出一个 `SyntaxError` 异常 .
4. 令 `parenIndex` 为整个正则表达式中在此产生式 `Term` 展开形式左侧出现的左匹配括号的数目。这是此产生式 `Term` 前面展开的 `Atom :: (Disjunction)` 产生式总数与此 `Term` 里面的 `Atom :: (Disjunction)` 产生式总数之和。
5. 令 `parenCount` 为在展开的 `Atom` 产生式里的左捕获括号数目。这是 `Atom` 产生式里面 `Atom :: (Disjunction)` 产生式的总数。
6. 返回一个需要两个参数 `State x` 和 `Continuation c` 的内部闭包 `Matcher`，执行方式如下：
 1. 调用 `RepeatMatcher(m, min, max, greedy, x, c, parenIndex, parenCount)`，并返回结果 .

抽象操作 `RepeatMatcher` 需要八个参数，一个 `Matcher m`，一个整数 `min`，一个整数（或 ∞ ）`max`，一个布尔值 `greedy`，一个 `State x`，一个 `Continuation c`，一个整数 `parenIndex`，一个整数 `parenCount`，执行方式如下：

1. 如果 `max` 是零，则调用 `c(x)`，并返回结果 .
2. 创建需要一个 `State` 参数 `y` 的内部 `Continuation` 闭包 `d`，执行方式如下：
 1. 如果 `min` 是零且 `y` 的 `endIndex` 等于 `x` 的 `endIndex`，则返回 `failure`.
 2. 如果 `min` 是零，则令 `min2` 为零；否则令 `min2` 为 `min-1`.
 3. 如果 `max` 是 ∞ ，则令 `max2` 为 ∞ ；否则令 `max2` 为 `max-1`.
 4. 调用 `RepeatMatcher(m, min2, max2, greedy, y, c, parenIndex, parenCount)`，并返回结果 .
3. 令 `cap` 为 `x` 的捕获内部数组的一个拷贝。
4. 对所有满足条件 `parenIndex < k` 且 `k ≤ parenIndex+parenCount` 的整数 `k`，设定 `cap[k]` 为 `undefined`。
5. 令 `e` 为 `x` 的 `endIndex`。
6. 令 `xr` 为 `State` 值 `(e, cap)`。

7. 如果 `min` 不是零，则调用 `m(xr, d)`，并返回结果。
8. 如果 `greedy` 是 `false`，则
 1. 令 `z` 为调用 `c(x)` 的结果。
 2. 如果 `z` 不是 `failure`，返回 `z`。
 3. 调用 `m(xr, d)`，并返回结果。
9. 令 `z` 为调用 `m(xr, d)` 的结果。
10. 如果 `z` 不是 `failure`，返回 `z`。
11. 调用 `c(x)`，并返回结果。

一个 **Atom** 后跟 **Quantifier** 是用 **Quantifier** 指定重复的次数。**Quantifier** 可以是非贪婪的，这种情况下 **Atom** 模式在能够匹配序列的情况下尽可能重复少的次数，或者它可以使贪婪的，这种情况下 **Atom** 模式在能够匹配序列的情况下尽可能重复多的次数，**Atom** 模式重复的是他自己而不是它匹配的字符串，所以不同次的重复中 **Atom** 可以匹配不同的子串。

假如 **Atom** 和后续的正则表达式都有选择余地，**Atom** 首先尽量多匹配（或者尽量少，假如是非贪婪模式）在最后一次 **Atom** 的重复中移动到下一个选择前，所有的后续中的选项都应该被尝试。在倒数第二次（第 $n-1$ 次）**Atom** 的重复中移动到下一个选择前，所有 **Atom** 的选项在最后一次（第 n 次）重复中应该被尝试。这样可以得出更多或者更少的重复次数可行。这些情况在开始匹配下一个选项的第 $(n-1)$ 次重复时已经被穷举，以此类推。

比较

```
/a[a-z]{2,4}/.exec("abcdefghi")
```

它返回"abcde"而

```
/a[a-z]{2,4}?/.exec("abcdefghi")
```

它返回"abc".

再考虑

```
/(aa|aabaac|ba|b|c)*/.exec("aabaac")
```

按照上面要求的选择数，它返回

```
["aaba", "ba"]
```

而非以下：

```
["aabaac", "aabaac"] ["aabaac", "c"]
```

上面要求的选择数可以用来编写一个计算两个数最大公约数的正则表达式(用单一字符重复数表示)。以下实例用来计算 10 和 15 的最大公约数：

```
"aaaaaaaaa,aaaaaaaaaaaaaaaa".replace(/^(a+)\1*,\1+$/, "$1")
```

它返回最大公约数的单一字符重复数表示"aaaaa".

RepeatMatcher 的步骤 4 每重复一次就清除 **Atom** 的捕获。我们可以看到它在正则表达式中的行为:

```
/(z)((a+)?(b+)?(c))*/.exec("zaacbbbcac")
```

它返回数组

```
["zaacbbbcac", "z", "ac", "a", undefined, "c"]
```

而非

```
["zaacbbbcac", "z", "ac", "a", "bbb", "c"]
```

因为最外面的*每次迭代都会清除所有括起来的 **Atom** 中所含的捕获字符串, 在这个例子中就是包含编号为 2,3,4,5 的捕获字符串。

RepeatMatcher 的 d 闭包状态步骤 1, 一旦重复的最小次数达到, 任何 **Atom** 匹配空 **String** 的扩展不再会匹配到重复中。这可以避免正则引擎在匹配类似下面的模式时掉进无限循环:

```
/(a*)*/.exec("b")
```

或者稍微复杂一点:

```
/(a*)b\1+/.exec("baaaac")
```

它返回数组:

```
["b", ""]
```

Assertion

产生式 **Assertion** :: \wedge 解释执行返回一个 **AssertionTester**, 它需要 1 个参数 **State x**, 并按如下算法执行:

1. 使 **e** 为 **x** 的 **endIndex**
2. 若 **e** = 0, 返回 **true**
3. 若 **Multiline** 为 **false**, 返回 **false**
4. 若 **Input[e - 1]** 的字符为 **LineTerminator**, 返回 **true**
5. 返回 **false**

产生式 `Assertion :: $` 解释执行返回一个 `AssertionTester`，它需要 1 个参数 `State x`，并按如下算法执行：

1. 使 `e` 为 `x` 的 `endIndex`
2. 若 `e = InputLength`，返回 `true`
3. 若 `Multiline` 为 `false`，返回 `false`
4. 若 `Input[e - 1]` 的字符为 `LineTerminator`，返回 `true`
5. 返回 `false`

产生式 `Assertion :: \b` 解释执行返回一个 `AssertionTester`，它需要 1 个参数 `State x`，并按如下算法执行：

1. 使 `e` 为 `x` 的 `endIndex`
2. 调用 `IsWordChar(e-1)`，返回 `Boolean` 值赋给 `a`
3. 调用 `IsWordChar(e)`，返回 `Boolean` 值赋给 `b`
4. 若 `a` 为 `true`，`b` 为 `false`，返回 `true`
5. 若 `b` 为 `false`，`b` 为 `true`，返回 `true`
6. 若 `Input[e - 1]` 的字符为 `LineTerminator`，返回 `true`
7. 返回 `false`

产生式 `Assertion :: \B` 解释执行返回一个 `AssertionTester`，它需要 1 个参数 `State x`，并按如下算法执行：

1. 使 `e` 为 `x` 的 `endIndex`
2. 调用 `IsWordChar(e-1)`，返回 `Boolean` 值赋给 `a`
3. 调用 `IsWordChar(e)`，返回 `Boolean` 值赋给 `b`
4. 若 `a` 为 `true`，`b` 为 `false`，返回 `false`
5. 若 `b` 为 `false`，`b` 为 `true`，返回 `false`
6. 若 `Input[e - 1]` 的字符为 `LineTerminator`，返回 `true`
7. 返回 `true`

产生式 `Assertion :: (? = Disjunction)` 按如下算法执行：

1. 执行 `Disjunction`，得到 `Matcher m`
2. 返回一个需要两个参数的内部闭包 `Matcher`，参数分别是一个 `State x` 和一个 `Continuation c`，此内部闭包的执行方式如下：
 1. 使 `d` 为一个 `Continuation`，它始终对它的任何 `State` 参数都返回成功匹配的 `MatchResult`
 2. 调用 `m(x, d)`，令 `r` 为其结果
 3. 若 `r` 为 `failure`，返回 `failure`
 4. 使 `y` 为 `r` 的 `State`
 5. 使 `cap` 为 `r` 的 `captures`
 6. 使 `xe` 为 `r` 的 `endIndex`
 7. 使 `z` 为 `State (xe, cap)`
 8. 调用 `c(z)`，返回结果

产生式 `Assertion :: (? ! Disjunction)` 按如下算法执行：

1. 执行 `Disjunction`，得到 `Matcher m`
2. 返回一个需要两个参数的内部闭包 `Matcher`，参数分别是一个 `State x` 和一个 `Continuation c`，此内部闭包的执行方式如下：
 1. 使 `d` 为一个 `Continuation`，它始终对它的任何 `State` 参数都返回成功匹配的 `MatchResult`
 2. 调用 `m(x, d)`，令 `r` 为其结果
 3. 若 `r` 为 `failure`，返回 `failure`
 4. 调用 `c(z)`，返回结果

抽象操作 `IsWordChar`，拥有一个 `integer` 类型的参数 `e`，按如下方式执行：

1. 若 `e == -1` 或 `e == InputLength`，返回 `false`
2. 令 `c` 为 `Input[e]`
3. 若 `c` 为 以下 63 个字符，返回 `true`

`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_`

1. 返回 `false`

Quantifier

产生式 `Quantifier :: QuantifierPrefix` 按如下方式执行：

1. 执行 `QuantifierPrefix` 得到 2 个数 `min` 和 `max`（或 ∞ ）
2. 返回 `min, max, true`

产生式 `Quantifier :: QuantifierPrefix ?` 按如下方式执行：

1. 执行 `QuantifierPrefix` 得到 2 个数 `min` 和 `max`（或 ∞ ）
2. 返回 `min, max, true`

产生式 `Quantifier :: *` 返回 0 和 ∞

产生式 `Quantifier :: +` 返回 1 和 ∞

产生式 `Quantifier :: ?` 返回 0 和 1

产生式 `Quantifier :: { DecimalDigits }` 按如下方式执行：

1. 令 `i` 为 `DecimalDigits` 的 MV
2. 返回 2 个结果 `i, i`

产生式 `Quantifier :: { DecimalDigits, }` 按如下方式执行:

1. 令 `i` 为 `DecimalDigits` 的 MV
2. 返回 2 个结果 `i`, ∞

产生式 `Quantifier :: { DecimalDigits, DecimalDigits }` 按如下方式执行:

1. 令 `i` 为 `DecimalDigits` 的 MV
2. 令 `j` 为 `DecimalDigits` 的 MV
3. 返回 2 个结果 `i`, `j`

Atom

产生式 `Atom :: PatternCharacter` 执行方式如下:

1. 令 `ch` 为 `PatternCharacter` 表示的字符
2. 令 `A` 为单元素 `CharSet`, 包含 `ch`
3. 调用 `CharacterSetMatcher(A, false)`, 返回 `Matcher`

产生式 `Atom :: .` 执行方式如下:

1. 令 `A` 为 除去 `LineTerminator` 外的所有字符
2. 调用 `CharacterSetMatcher(A, false)`, 返回 `Matcher`

产生式 `Atom :: \ AtomEscape` 通过执行 `AtomEscape` 返回 `Matcher`。

产生式 `Atom :: CharacterClass` 执行方式如下:

1. 执行 `CharacterClass` 得到 `CharSet A` 和 `Boolean invert`
2. 调用 `CharacterSetMatcher(A, false)`, 返回 `Matcher`

产生式 `Atom :: (Disjunction)` 执行方式如下:

1. 执行 `Disjunction` 得到 `Matcher`
2. 令 `parenIndex` 为 在整个正则表达式中从产生式展开初始化左括号时, 当前展开左捕获括号的索引。`parenIndex` 为在产生式的 `Atom` 被展开之前, `Atom :: (Disjunction)` 产生式被展开的次数, 加上 `Atom :: (Disjunction)` 闭合 这个 `Atom` 的次数
3. 返回一个内部闭包 `Matcher`, 拥有 2 个参数: 一个 `State x` 和 `Continuation c`, 执行方式如下:
 1. 创建内容闭包 `Continuation d`, 参数为 `State y`, 并按如下方式执行:
 - i. 令 `cap` 为 `y` 的 `capture` 数组的一个拷贝
 - ii. 令 `xe` 为 `x` 的 `endIndex`
 - iii. 令 `ye` 为 `y` 的 `endIndex`
 - iv. 令 `s` 为 `Input` 从索引 `xe` (包括) 至 `ye` (不包括) 范围的新创建的字符串

- v. 令 s 为 cap[parenIndex+1]
 - vi. 令 z 为 State (ye, cap)
 - vii. 调用 c(z), 返回其结果
2. 执行 m(x, d), 返回其结果

产生式 Atom :: (? : Disjunction) 通过执行 Disjunction 得到并返回一个 Matcher。

抽象操作 CharacterSetMatcher , 拥有 2 个参数: 一个 CharSet A 和 Boolean invert 标志, 按如下方式执行:

1. 返回一个内部闭包 Matcher, 拥有 2 个参数: 一个 State x 和 Continuation c, 执行方式如下:
 1. 令 e 为 x 的 endIndex
 2. 若 e == InputLength, 返回 failure
 3. 令 ch 为字符 Input[e]
 4. 令 cc 为 Canonicalize(ch)的结果
 5. 若 invert 为 false, 如果 A 中不存在 a 使得 Canonicalize(a) == cc, 返回 failure
 6. 若 invert 为 true, 如果 A 中存在 a 使得 Canonicalize(a) == cc, 返回 failure
 7. 令 cap 为 x 的内部 captures 数组
 8. 令 y 为 State (e+1, cap)
 9. 调用 c(y), 返回结果

抽象操作 Canonicalize, 拥有一个字符参数 ch, 按如下方式执行:

1. 若 IgnoreCase 为 false, 返回 ch
2. 令 u 为 ch 转换为大写后的结果, 仿佛通过调用标准内置方法 String.prototype.toUpperCase
3. 若 u 不含单个字符, 返回 ch
4. 令 cu 为 u 的字符
5. 若 ch 的 code unit value >= 128 且 cu 的 code unit value <= 128, 返回 ch
6. 返回 cu

(Disjunction) 的括号 用来组合 Disjunction 模式, 并保存匹配结果。该结果可以通过后向引用 (一个非零数, 前置\), 在一个替换字符串中的引用, 或者作为正则表达式内部匹配过程的部分结果。使用(?: Disjunction)来避免括号的捕获行为。

(? = Disjunction)指定一个零宽正向预查。为了保证匹配成功, 其 Disjunction 必须首先能够匹配成功, 但在匹配后续字符前, 其当前位置会不变。如果 Disjunction 能在当前位置以多种方式匹配, 那么只会取第一次匹配的结果。不像其他正则表达式运算符, (?= 内部不会回溯 (这个特殊的行为是从 Perl 继承过来的)。在 Disjunction 含有捕获括号, 模式的后续字符包括后向引用时匹配结果会有影响。

例如，

```
/(?=(a+))/ .exec("baaabac")
```

会匹配第一个 **b** 后的空白字符串，得到：

```
["", "aaa"]
```

为了说明预查不会回溯，

```
/(?=(a+))a*b\1/.exec("baaabac")
```

得到：

```
["aba", "a"]
```

而不是：

```
["aaaba", "a"]
```

(?! Disjunction) 指定一个零宽正向否定预查。为了保证匹配成功，其 **Disjunction** 必须首先能够匹配失败，但在匹配后续字符前，其当前位置会不变。**Disjunction** 能含有捕获括号，但是对这些捕获分组的后向引用只在 **Disjunction** 中有效。在当前模式的其他位置后向引用捕获分组都会返回 **undefined**。因为否定预查必须满足预查失败来保证模式成功匹配。例如，

```
/(.*)a(?! (a+)b\2c)\2(.*)/.exec("baaabaac")
```

搜索 **a**，其后有 **n** 个 **a**，一个 **b**，**n** 个 **a** (**\2** 指定) 和一个 **c**。第二个 **\2** 位于负向预查模式的外部，因此它匹配 **undefined**，且总是成功的。整个表达式返回一个数组：

```
["baaabaac", "ba", undefined, "abaac"]
```

在发生比较前，一次不区分大小写的匹配中所有的字符都会隐式转换为大写。然而，如果某些单个字符在转换为大写时扩展为多个字符，那么该字符会保持原样。当某些非 **ASCII** 字符在转换为大写时变成 **ASCII** 字符，该字符也会保持原样。这样会阻止 **Unicode** 字符（例如 **\u0131** 和 **\u017F**）匹配正则表达式（例如仅匹配 **ASCII** 字符的正则表达式 **/[a z]/i**）。而且，如果转换允许，**/[\W]/i** 会匹配除去 **i** 或 **s** 外的每一个 **a**，**b**，.....，**h**。

AtomEscape

产生式 **AtomEscape :: DecimalEscape** 执行方式如下：

1. 执行 `DecimalEscape` 得到 `EscapeValue E`
2. 如果 `E` 为一个字符,
 1. 令 `ch` 为 `E` 的字符
 2. 令 `A` 为包含 `ch` 字符的单元字符集 `CharSet`
 3. 调用 `CharacterSetMatcher(A, false)` 返回 `Matcher` 结果
3. `E` 必须是一个数。令 `n` 为该数。
4. 如果 `n=0` 或 `n>NCapturingParens`, 抛出 `SyntaxError` 异常
5. 返回一个内部闭包 `Matcher`, 拥有 2 个参数: 一个 `State x` 和 `Continuation c`, 执行方式如下:
 1. 令 `cap` 为 `x` 的 `captures` 内部数组
 2. 令 `s` 为 `cap[n]`
 3. 如果 `s` 为 `undefined`, 调用 `c(x)`, 返回结果
 4. 令 `e` 为 `x` 的 `endIndex`
 5. 令 `len` 为 `s` 的 `length`
 6. 令 `f` 为 `e+len`
 7. 如果 `f>InputLength`, 返回 `failure`
 8. 如果存在位于 0 (包括) 到 `len` (不包括) 的整数 `i` 使得 `Canonicalize(s[i])` 等于 `Canonicalize(Input[e+i])`, 那么返回 `failure`
 9. 令 `y` 为 `State(f, cap)`
 10. 调用 `c(y)`, 返回结果

产生式 `AtomEscape :: CharacterEscape` 执行方式如下:

1. 执行 `CharacterEscape` 得到一个 `ch` 字符
2. 令 `A` 为包含 `ch` 字符的单元字符集 `CharSet`
3. 调用 `CharacterSetMatcher(A, false)` 返回 `Matcher` 结果

产生式 `AtomEscape :: CharacterClassEscape` 执行方式如下:

1. 执行 `CharacterClassEscape` 得到一个 `CharSet A`
2. 调用 `CharacterSetMatcher(A, false)` 返回 `Matcher` 结果

格式 `\n` 后为非零数 `n` 的转义序列匹配捕获分组的第 `n` 次匹配结果。如果正则表达式少于 `n` 个捕获括号, 会报错。如果正则表达式大于等于 `n` 个捕获括号, 由于没有捕获到任何东西, 导致第 `n` 个捕获分组结果为 `undefined`, 那么后向引用总是成功的。

CharacterEscape

产生式 `CharacterEscape :: ControlEscape` 执行返回一个根据表 23 定义的字符:

ControlEscape 字符转义值

ControlEscape	字符编码值	名称	符号
---------------	-------	----	----

t	\u0009	水平制表符	<HT>
n	\u000A	进行（新行）	<LF>
v	\u000B	竖直制表符	<VT>
f	\u000C	进纸	<FF>
r	\u000D	回车	<CR>

产生式 `CharacterEscape :: ch ControlLetter` 执行过程如下：

1. 令 `ch` 为通过 `ControlLetter` 表示的字符
2. 令 `i` 为 `ch` 的 code unit value
3. 令 `j` 为 `i/32` 的余数
4. 返回 `j`

产生式 `CharacterEscape :: HexEscapeSequence` 执行 `HexEscapeSequence` 的 CV，返回其字符结果。

产生式 `CharacterEscape :: UnicodeEscapeSequence` 执行 `UnicodeEscapeSequence` 的 CV，返回其字符结果。

产生式 `CharacterEscape :: IdentityEscape` 执行返回由 `IdentityEscape` 表示的字符。

DecimalEscape

产生式 `DecimalEscape :: DecimalIntegerLiteral [lookahead ∉ DecimalDigit]` 按如下方式执行：

1. 令 `i` 为 `DecimalIntegerLiteral` 的 CV 值
2. 如果 `i` 为 0，返回包含一个 <NUL> 字符（Unicode 值为 0000）的 `EscapeValue`
3. 返回包含整数 `i` 的 `EscapeValue`

“the MV of `DecimalIntegerLiteral`”在 7.8.3 节定义。

如果 \ 后面是一个数字，且首位为 0，那么，该转义序列被认为是一个后向引用。如果 `n` 比在整个正则表达式左捕获括号个数大，那么会出错。 `\0` 表示 <NUL> 字符，其后不能再有数字。

CharacterClassEscape

产生式 `CharacterClassEscape :: d` 执行返回包含 0 到 9 之间的十元素字符集。

产生式 `CharacterClassEscape :: D` 执行返回不包括 `CharacterClassEscape :: d` 的字符集。

产生式 `CharacterClassEscape :: s` 执行返回包含 `WhiteSpace` 或 `LineTerminator` 产生式右部分字符的字符集。

产生式 `CharacterClassEscape :: S` 执行返回不包括 `CharacterClassEscape :: s` 的字符集。

产生式 `CharacterClassEscape :: w` 执行返回包含如下 63 个字符的字符集：

`a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 _`

产生式 `CharacterClassEscape :: W` 执行返回不包括 `CharacterClassEscape :: w` 的字符集。

CharacterClass

产生式 `CharacterClass :: [[lookahead ∉ {^}] ClassRanges]` 通过执行 `ClassRanges` 获得并返回这个 `CharSet` 和 `Boolean false`。

产生式 `CharacterClass :: [^ ClassRanges]` 通过执行 `ClassRanges` 获得并返回这个 `CharSet` 和 `Boolean true`。

ClassRanges

产生式 `ClassRanges :: [empty]` 执行返回一个空的 `CharSet`。

产生式 `ClassRanges :: NonemptyClassRanges` 通过执行 `NonemptyClassRanges` 获得并返回这个 `CharSet`。

NonemptyClassRanges

产生式 `NonemptyClassRanges :: ClassAtom` 通过执行 `ClassAtom` 获得一个 `CharSet` 并返回这个 `CharSet`。

产生式 `NonemptyClassRanges :: ClassAtom NonemptyClassRangesNoDash` 按如下方式执行：

1. 执行 `ClassAtom` 得到一个 `CharSet A`
2. 执行 `NonemptyClassRangesNoDash` 得到一个 `CharSet B`
3. 返回 `A` 与 `B` 的并集

产生式 `NonemptyClassRanges :: ClassAtom - ClassAtom ClassRanges` 按如下方式执行：

1. 执行第一个 `ClassAtom` 得到一个 `CharSet A`
2. 执行第二个 `ClassAtom` 得到一个 `CharSet B`
3. 执行 `ClassRanges` 得到一个 `CharSet C`
4. 调用 `CharacterRange(A, B)`, 令 `D` 为其结果 `CharSet`
5. 返回 `D` 与 `C` 的并集

抽象操作 `CharacterRange`, 拥有 2 个 `CharSet` 参数 `A` 和 `B`, 执行方式如下:

1. 如果 `A` 或 `B` 为空, 抛出 `SyntaxError` 异常
2. 令 `a` 为 `CharSet A` 的一个字符
3. 令 `b` 为 `CharSet B` 的一个字符
4. 令 `i` 为 `a` 的 code unit value
5. 令 `j` 为 `b` 的 code unit value
6. 如果 `i > j`, 抛出 `SyntaxError` 异常
7. 返回位于在 `i` 到 `j` (包括边界) 之间的所有字符的字符集

NonemptyClassRangesNoDash

产生式 `NonemptyClassRangesNoDash :: ClassAtom` 执行过程是执行 `ClassAtom` 产生一个 `CharSet` 并且返回这个 `CharSet`。

产生式 `NonemptyClassRangesNoDash :: ClassAtomNoDash`
`NonemptyClassRangesNoDash` 按以下方式执行:

1. 执行 `ClassAtomNoDash` 产生 `CharSet A`
2. 执行 `NonemptyClassRangesNoDash` 产生 `CharSet B`
3. 返回 `CharSet A` 和 `B` 的并集

产生式 `NonemptyClassRangesNoDash :: ClassAtomNoDash - ClassAtom`
`ClassRanges` 按以下方式执行:

1. 执行 `ClassAtomNoDash` 产生 `CharSet A`
2. 执行 `ClassAtom` 产生 `CharSet B`
3. 执行 `ClassRanges` 产生 `CharSet C`
4. 调用 `CharacterRange(A, B)` 并设 `CharSet D` 为结果。
5. 返回 `CharSet D` 和 `C` 的并集

`ClassRanges` 可以拆分成单独的 `ClassAtom` 且/或两个用减号分隔的 `ClassAtom`。在后面的情况下 `ClassAtom` 包含第一个到第二个 `ClassAtom` 间的所有字符。如果两个 `ClassAtom` 之一不是表示一个单独字符 (例如其中一个是 `\w`) 或者第一个 `ClassAtom` 的字符编码值比第二个 `ClassAtom` 的字符编码值大则发生错误。

即使匹配忽略大小写，区间两端的大小写在区分哪些字符属于区间时仍然有效。这意味着，例如模式/[E-F]/仅仅匹配 E, F, e, 和 f。/[E-f]/i 则匹配所有大写和小写的 ASCII 字母以及[, \,], ^, _ 和 `

-字符可能被当做字面意思或者表示一个区间，它作为 **ClassRange** 的开头或者结尾、在区间指定开头或者结尾，或者紧跟一个区间指定的时候被当做字面意思。

ClassAtom

产生式 **ClassAtom** :: - 执行返回包含单个字符 - 的字符集。

产生式 **ClassAtom** :: **ClassAtomNoDash** 通过执行 **ClassAtomNoDash** 获得并返回这个 **CharSet**。

ClassAtomNoDash

产生式 **ClassAtomNoDash** :: **SourceCharacter** 不包括\,], - 执行返回包含由 **SourceCharacter** 表示的字符的单元字符集。

产生式 **ClassAtomNoDash** :: \ **ClassEscape** 通过执行 **ClassEscape** 得到并返回这个 **CharSet**。

ClassEscape

产生式 **ClassEscape** :: **DecimalEscape** 按如下方式执行：

1. 执行 **DecimalEscape** 得到 **EscapeValue E**
2. 如果 E 不是一个字符，抛出 **SyntaxError** 异常
3. 令 ch 为 E 的字符
4. 返回包含字符 ch 的单元字符集 **CharSet**

产生式 **ClassEscape** :: b 执行返回包含一个<BS>字符（Unicode 值 0008）的字符集。

产生式 **ClassEscape** :: **CharacterEscape** 通过执行 **CharacterEscape** 获得一个字符 **CharSet** 并返回包含该字符的单元字符集 **CharSet**。

产生式 **ClassEscape** :: **CharacterClassEscape** 通过执行 **CharacterClassEscape** 获得并返回这个 **CharSet**。

ClassAtom 可以使用除\b, \B, 后向引用外的转义序列。在 **CharacterClass** 中，\b 表示退格符。然而，\B 和后向引用会报错。同样，在一个 **ClassAtom** 中使用后向引用会报错。

The RegExp Constructor Called as a Function

RegExp(pattern, flags)

如果 `pattern` 是一个对象 `R`，其内部属性 `[[Class]]` 为 `RegExp` 且 `flags` 为 `undefined`，返回 `R`。否则，调用内置 `RegExp` 构造器，通过表达式 `new RegExp (pattern, flags)` 返回由该构造器构造的对象。

The RegExp Constructor

当 `RegExp` 作为 `new` 表达式一部分调用时，它是一个构造器，用来初始化一个新创建的对象。

new RegExp(pattern, flags)

如果 `pattern` 是一个对象 `R`，其内部 `[[CLASS]]` 属性为 `RegExp`，且 `flags` 为 `undefined`，那么，令 `P` 为 `pattern` 和令 `F` 为 `flags` 用来构造 `R`。如果 `pattern` 是一个对象 `R`，其内部 `[[CLASS]]` 属性为 `RegExp`，且 `flags` 为 `undefined`，那么，抛出 `TypeError` 异常。否则，如果 `pattern` 为 `undefined` 且 `ToString(pattern)`，令 `P` 为空的字符串；如果 `flags` 为 `undefined` 且 `ToString(flags)` 令 `F` 为空字符串。

如果字符 `P` 不满足 `Pattern` 语义，那么抛出 `SyntaxError` 异常。否则，令新构造的对象拥有内部 `[[Match]]` 属性，该属性通过执行（编译）字符 `P` 作为在 15.10.2 节描述的 `Pattern`。

如果 `F` 含有除“`g`”，“`i`”，“`m`”外的任意字符，或者 `F` 中包括出现多次的字符，那么，抛出 `SyntaxError` 异常。

如果 `SyntaxError` 异常未抛出，那么：

令 `S` 为一个字符串，其等价于 `P` 表示的 `Pattern`，`S` 中的字符按如下描述进行转义。这样，`S` 可能或者不会与 `P` 或者 `pattern` 相同；然而，由执行 `S` 作为一个 `Pattern` 的内部处理程序必须和通过构造对象的内部 `[[Match]]` 属性的内部处理程序完全相同。

如果 `pattern` 里存在字符/或者\，那么这些字符应该被转义，以确保由“/”，`S`，“/”构成的字符串的 `S` 值有效，而且 `F` 能被解析（在适当的词法上下文中）为一个与构造的正则表达式行为完全相同的 `RegularExpressionLiteral`。例如，如果 `P` 是“/”，那么 `S` 应该为“\”或“\u002F”，而不是“/”，因为 `F` 后的 `///` 会被解析

为一个 `SingleLineComment`，而不是一个 `RegularExpressionLiteral`。如果 `P` 为空字符串，那么该规范定义为令 `S` 为“(?:)”。

这个新构造对象的如下属性为数据属性，其特性在 15.10.7 中定义。各属性的 `[[Value]]` 值按如下方式设置：

其 `source` 属性置为 `S`。

其 `global` 属性置为一个 `Boolean` 值。当 `F` 含有字符 `g` 时，为 `true`，否则为 `false`。

其 `ignoreCase` 属性置为一个 `Boolean` 值。当 `F` 含有字符 `i` 时，为 `true`，否则，为 `false`。

其 `multiline` 属性置为一个 `Boolean` 值。当 `F` 含有字符 `m` 时，为 `true`，否则，为 `false`。

其 `lastIndex` 属性置为 0。

其内部 `[[Prototype]]` 属性置为 15.10.6 中定义的内置 `RegExp` 原型对象。

其内部 `[[Class]]` 属性置为“`RegExp`”。

如果 `pattern` 为 `StringLiteral`，一般的转义字符替换发生在被 `RegExp` 处理前。如果 `pattern` 必须含有 `RegExp` 识别的转义字符，那么当构成 `StringLiteral` 的内容时，为了防止被移除被移除，在 `StringLiteral` 中的任何 `\` 必须被转义

Properties of the RegExp Constructor

`RegExp` 构造器的 `[[Prototype]]` 值为内置 `Function` 的原型（15.3.4）。

除了内部的一些属性和 `length` 属性（其值为 2），`RegExp` 构造器还有如下属性：

RegExp.prototype

`RegExp.prototype` 的初始值为 `RegExp` 的原型（15.10.6）。

该属性有这些特性：{ `[[Writable]]`: `false`, `[[Enumerable]]`: `false`, `[[Configurable]]`: `false` }。

Properties of the RegExp Prototype Object

`RegExp` 的原型的内部 `[[Prototype]]` 属性为 `Object` 的原型（15.2.4）。`RegExp` 的原型为其本身的一个普通的正则表达式对象；它的 `[[Class]]` 为“`RegExp`”。

RegExp 的原型对象的数据式属性的初始值被设置为仿佛由内置 RegExp 构造器深生成的表达式 `new RegExp()` 创建的对象。

RegExp 的原型本身没有 `valueOf` 属性；然而，该 `valueOf` 属性是继承至 `Object` 的原型。

在作为 RegExp 原型对象的属性的如下函数描述中，“this RegExp object”是指函数激活时 this 对象；如果 this 值不是一个对象，或者一个其内部 `[[Class]]` 属性值不是“RegExp”的对象，那么一个 `TypeError` 会抛出。

RegExp.prototype.constructor

RegExp.prototype.constructor 的初始值为内置 RegExp 构造器。

RegExp.prototype.exec(string)

Performs a regular expression match of string against the regular expression and returns an Array object containing the results of the match, or null if string did not match.

The String ToString(string) is searched for an occurrence of the regular expression pattern as follows:

1. 令 R 为这一 RegExp 对象.
2. 令 S 为 ToString(string) 的值.
3. 令 length 为 S 的长度.
4. 令 lastIndex 为以参数 "lastIndex" 调用 R 的内部方法 `[[Get]]` 的结果
5. 令 i 为 ToInteger(lastIndex) 的值.
6. 令 global 为以参数 "global" 调用 R 的内部方法 `[[Get]]` 的结果
7. 若 global 为 false, 则令 i = 0.
8. 令 matchSucceeded 为 false.
9. 到 matchSucceeded 为 false 前重复以下
 1. 若 i < 0 或者 i > length, 则
 - i. 以参数 "lastIndex", 0, and true 调用 R 的内部方法 `[[Put]]`
 - ii. Return null.
 2. 以参数 S 和 i 调用 R 的内部方法 `[[Match]]`
 3. 若 `[[Match]]` 返回失败, 则
 - i. 令 i = i + 1.
 4. 否则
 - i. 令 r 为调用 `[[Match]]` 的结果 State.
 - ii. 设 matchSucceeded 为 true.
10. 令 e 为 r 的 endIndex 值.
11. 若 global 为 true,

12. 以参数"lastIndex", e, 和 true 调用 R 的内部方法[[Put]]
13. 令 n 为 r 的捕获数组的长度. (这跟 15.10.2.1's NCapturingParens 是同一个值)
14. 令 A 为如同以表达式 new Array 创建的新数组, 其中 Array 是这个名字的内置构造器.
15. 令 matchIndex 为匹配到的子串在整个字符串 S 中的位置.
16. 以参数"index", 属性描述{[[Value]]: matchIndex, [[Writable]: true, [[Enumerable]]: true, [[Configurable]]: true}, 和 true 调用 A 的内部方法 [[DefineOwnProperty]]
17. 以参数"input", 属性描述{[[Value]]: S, [[Writable]: true, #[[Enumerable]]: true, [[Configurable]]: true}, 和 true 调用 A 的内部方法 [[DefineOwnProperty]]
18. 以参数"length", 属性描述{[[Value]]: n + 1}, 和 true 调用 A 的内部方法 [[DefineOwnProperty]]
19. 令 matchedSubstr 为匹配到的子串(i.e. the portion of S between offset i inclusive and offset e exclusive).
20. 以参数"0", 属性描述{[[Value]]: matchedSubstr, [[Writable]: true, [[Enumerable]]: true, [[Configurable]]: true}, 和 true 调用 A 的内部方法 [[DefineOwnProperty]]
21. 对每一满足 $i > 0$ 且 $i \leq n$ 的整数 i
 1. 令 captureI 为第 i 个捕获数组中的元素.
 2. 以参数 ToString(i), 属性描述{[[Value]]: captureI, [[Writable]: true, [[Enumerable]]: true, [[Configurable]]: true}, 和 true 调用 A 的内部方法 [[DefineOwnProperty]]
22. 返回 A.

RegExp.prototype.test(string)

采用如下步骤:

1. 令 match 为在这个 RegExp 对象上使用 string 作为参数执行 RegExp.prototype.exec (15.10.6.2) 的结果。
2. 如果 match 不为 null, 返回 true; 否则返回 false。

RegExp.prototype.toString()

返回一个 String, 由"/", RegExp 对象的 source 属性值, "/"与"g" (如果 global 属性为 true), "i" (如果 ignoreCase 为 true), "m" (如果 multiline 为 true) 通过连接组成。

如果返回的字符串包含一个 **RegularExpressionLiteral**, 那么该 **RegularExpressionLiteral** 用同样的方式解释执行。

Properties of RegExp Instances

RegExp 实例继承至 RegExp 原型对象，其[[CLASS]]内部属性值为“RegExp”。RegExp 实例也拥有一个[[Match]]内部属性和一个 length 属性。

内部属性[[Match]]的值是正则表达式对象的 Pattern 的依赖实现的表示形式。

RegExp 实例还有如下属性。

source

source 属性为构成正则表达式 Pattern 的字符串。该属性拥有这些特性 { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }。

global

global 属性是一 Boolean 值，表示正则表达式 flags 是否有“g”。该属性拥有这些特性 { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }。

ignoreCase

ignoreCase 属性是一 Boolean 值，表示正则表达式 flags 是否有“i”。该属性拥有这些特性 { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }。

multiline

multiline 属性是一 Boolean 值，表示正则表达式 flags 是否有“m”。该属性拥有这些特性 { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }。

lastIndex

lastIndex 属性指定从何处开始下次匹配的一个字符串类型的位置索引。当需要时该值会转换为一个整型数。该属性拥有这些特性 { [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false }。

不同于其他 RegExp 实例内置属性，lastIndex 是可写的。

Error Objects

Error 对象的实例在运行时遇到错误的情况下会被当做异常抛出。**Error** 对象也可以作为用户自定义异常类的基对象。

The Error Constructor Called as a Function

当 **Error** 被作为函数而不是构造器调用时，它创建并初始化一个新的 **Error** 对象。这样函数调用 **Error(...)** 与同样参数的对象创建表达式 **new Error(...)** 是等效的。

Error (message)

新构造的对象内部属性 **Prototype** 会被设为原本的 **Error** 原型对象，也就是 **Error.prototype** 的初始值。(15.11.3.1)

新构造的对象内部属性 **Class** 会被设为 "Error"。

新构造的对象内部属性 **Extensible** 会被设为 **true**。

如果形参 **message** 不是 **undefined**，新构造的对象本身属性 **message** 则被设为 **ToString(message)**。

The Error Constructor

当 **Error** 作为 **new** 表达式的一部分被调用时，它是一个构造器：它初始化新创建的对象。

new Error (message)

新构造的对象内部属性 **Prototype** 会被设为原本的 **Error** 原型对象，也就是 **Error.prototype** 的初始值。(15.11.3.1)

新构造的对象内部属性 **Class** 会被设为 "Error"。

新构造的对象内部属性 **Extensible** 会被设为 **true**。

如果形参 **message** 不是 **undefined**，新构造的对象本身属性 **message** 则被设为 **ToString(message)**。

Properties of the Error Constructor

Error 构造器的内部属性 **Prototype** 值为 **Function** 原型对象(15.3.4)。

除内部属性和 `length` 属性（其值为 1）以外，`Error` 构造器还有以下属性：

Error.prototype

`Error.prototype` 的初始值为 `Error` 原型对象(15.11.4)。

此属性有以下特性： { Writable: false, Enumerable: false, Configurable: false }。

Properties of the Error Prototype Object

`Error` 原型对象本身是一个 `Error` 对象（其 `Class` 为"`Error`"）。

`Error` 原型对象的内部属性 `Prototype` 为标准内置的 `Object` 原型对象(15.2.4)。

Error.prototype.constructor

`Error.prototype.constructor` 初始值为内置的 `Error` 构造器。

Error.prototype.name

`Error.prototype.name` 初始值为"`Error`"。

Error.prototype.message

`Error.prototype.message` 初始值为空字符串。

Error.prototype.toString ()

执行以下步骤

1. 令 `o` 为 `this` 值
2. 如果 `Type(O)`不是对象，抛出一个 `TypeError` 异常。
3. 令 `name` 为以"`name`"为参数调用 `O` 的 `Get` 内置方法的结果。
4. 如果 `name` 为 `undefined`，令 `name` 为"`Error`";否则令 `name` 为 `ToString(name)`。
5. 令 `msg` 为以"`message`"为参数调用 `O` 的 `Get` 内置方法的结果。
6. 如果 `msg` 为 `undefined`，令 `msg` 为空字符串;否则令 `msg` 为 `ToString(msg)`。
7. 如果 `name` 与 `msg` 都是空字符串，返回"`Error`"。
8. 如果 `name` 为空字符串，返回 `msg`。
9. 如果 `msg` 为空字符串，返回 `name`。

10. 返回拼接 `name,":",` 一个空格字符, 以及 `msg` 的结果。

Error 实例的属性

Error 实例从 Error 原型对象继承属性, 且它们的内部属性 `class` 值为 "Error"。Error 实例没有特殊属性。

Native Error Types Used in This Standard

以下原生 Error 对象之一会在运行时错误发生时被抛出。所有这些对象共享同样的结构, 如 15.11.7 所述。

EvalError

本规范现在已经不再使用这个异常, 这个对象保留用于跟规范之前版本的兼容性。

RangeError

表示一个数值超出了允许的范围, 见 15.4.2.2, 15.4.5.1, 15.7.4.2, 15.7.4.5, 15.7.4.6, 以及 15.7.4.7, 15.9.5.43.

ReferenceError

表示一个不正确的引用值被检测到。见 8.7.1, 8.7.2, 10.2.1, 10.2.1.1.4, 10.2.1.2.4, 以及 11.13.1

SyntaxError

表示一个解析错误发生。见 11.1.5, 11.3.1, 11.3.2, 11.4.1, 11.4.4, 11.4.5, 11.13.1, 11.13.2, 12.2.1, 12.10.1, 12.14.1, 13.1, 15.1.2.1, 15.3.2.1, 15.10.2.2, 15.10.2.5, 15.10.2.9, 15.10.2.15, 15.10.2.19, 15.10.4.1, 以及 15.12.2

TypeError

表示一个操作数的真实类型与期望类型不符。见 8.6.2, 8.7.2, 8.10.5, 8.12.5, 8.12.7, 8.12.8, 8.12.9, 9.9, 9.10, 10.2.1, 10.2.1.1.3, 10.6, 11.2.2, 11.2.3, 11.4.1, 11.8.6, 11.8.7, 11.3.1, 13.2, 13.2.3, 15, 15.2.3.2, 15.2.3.3, 15.2.3.4, 15.2.3.5, 15.2.3.6, 15.2.3.7, 15.2.3.8, 15.2.3.9, 15.2.3.10, 15.2.3.11, 15.2.3.12, 15.2.3.13, 15.2.3.14, 15.2.4.3, 15.3.4.2, 15.3.4.3, 15.3.4.4, 15.3.4.5, 15.3.4.5.2,

15.3.4.5.3, 15.3.5, 15.3.5.3, 15.3.5.4, 15.4.4.3, 15.4.4.11, 15.4.4.16, 15.4.4.17, 15.4.4.18, 15.4.4.19, 15.4.4.20, 15.4.4.21, 15.4.4.22, 15.4.5.1, 15.5.4.2, 15.5.4.3, 15.6.4.2, 15.6.4.3, 15.7.4, 15.7.4.2, 15.7.4.4, 15.7.4.8 [?], 15.9.5, 15.9.5.44, 15.10.4.1, 15.10.6, 15.11.4.4 以及 15.12.3

URIError

表示全局 URI 处理函数被以不符合其定义的方式使用。见 15.1.3。

NativeError 对象结构

当 ECMAScript 实现探测到一个运行时错误时，它抛出一个 15.11.6 所定义的 **NativeError** 对象的实例。每个这些对象都有如下所述结构，不同仅仅是在 **name** 属性中以构造器名称替换掉 **NativeError**，以及原型对象由实现自定义的 **message** 属性。

对于每个错误对象，定义中到 **NativeError** 的引用应当用 15.11.6 中具体的对象名替换。

NativeError Constructors Called as Functions

当 **NativeError** 被作为函数而不是构造器调用时，它创建并初始化一个新的 **NativeError** 对象。这样函数调用 **NativeError(...)** 与同样参数的对象创建表达式 **new NativeError(...)** 是等效的。

NativeError (message)

新构造的对象内部属性 **Prototype** 会被设为这一错误构造器附带的原型对象。新构造的对象内部属性 **Class** 会被设为 "Error"。新构造的对象内部属性 **Extensible** 会被设为 **true**。

如果形参 **message** 不是 **undefined**，新构造的对象本身属性 **message** 则被设为 **ToString(message)**。

The NativeError Constructors

当 **NativeError** 作为 **new** 表达式的一部分被调用时，它是一个构造器：它初始化新创建的对象。

New NativeError (message)

新构造的对象内部属性 **Prototype** 会被设为这一错误构造器附带的原型对象。新构造的对象内部属性 **Class** 会被设为"Error"。新构造的对象内部属性 **Extensible** 会被设为 true。

如果形参 **message** 不是 **undefined**，新构造的对象本身属性 **message** 则被设为 **ToString(message)**。

Properties of the NativeError Constructors

NativeError 构造器的内部属性 **Prototype** 值为 **Function** 原型对象(15.3.4)。

除内部属性和 **length** 属性（其值为 1）以外，**Error** 构造器还有以下属性：

NativeError.prototype

NativeError.prototype 的初始值为一个 **Error**(15.11.4)。

此属性有以下特性： { Writable: false, Enumerable: false, Configurable: false }。

Properties of the NativeError Prototype Objects

每个 **NativeError** 的 **prototype** 的初始值为一个 **Error**（其 **Class** 为"Error"）。

NativeError 原型对象的内部属性 **Prototype** 为标准内置的 **Error** 对象(15.2.4)。

NativeError.prototype.constructor

对于特定的 **NativeError**，**Error.prototype.constructor** 初始值为 **NativeError** 构造器本身。

NativeError.prototype.name

对于特定的 **NativeError**，**Error.prototype.name** 初始值为构造器的名字。

NativeError.prototype.message

对于特定的 **NativeError**，**NativeError.prototype.message** 初始值为空字符串。

NativeError 构造器的原型他们自身并不提供 **toString** 函数，但是错误的实例可以从 **Error** 原型对象继承到它。

NativeError 实例的属性

NativeError 实例从 NativeError 原型对象继承属性，且它们的内部属性 class 值为"Error"。Error 实例没有特殊属性。

JSON 对象

JSON 对象是一个单一的对象，它包含两个函数，parse 和 stringify，是用于解析和构造 JSON 文本的。JSON 数据的交换格式在 RFC4627 里进行了描述。 <<http://www.ietf.org/rfc/rfc4627.txt>>。本规范里面的 JSON 交换格式会使用 RFC4627 里所描述的，以下两点除外：

- ECMAScript JSON 文法中的顶级 JSONText 产生式是由 JSONValue 构成，而不是 RFC4627 中限制成的 JSONObject 或者 JSONArray。
- 确认 JSON.parse 和 JSON.stringify 的实现，它们必须准确的支持本规范描述的交换格式，而不允许对格式进行删除或扩展。这一点要区别于 RFC4627，它允许 JSON 解析器接受 non-JSON 的格式和扩展。

JSON 对象内部属性 [[Prototype]] 的值是标准内建的 Object 原型对象（15.2.4）。内部属性 [[Class]] 的值是"JSON"。内部属性 [[Extensible]] 的值设置为 true。

JSON 对象没有内部属性 [[Construct]]；不能把 JSON 对象当作构造器来使用 new 操作符。

JSON 对象没有内部属性 [[Call]]；不能把 JSON 对象当作函数来调用。

JSON 语法

JSON.stringify 会产生一个符合 JSON 语法的字符串。JSON.parse 接受的是一个符合 JSON 语法的字符串。

JSON 词法

类似于 ECMAScript 源文本，JSON 是由一系列符合 SourceCharacter 规则的字符构成的。JSON 词法定义的 tokens 使得 JSON 文本类似于 ECMAScript 词法定义的 tokens 得到的 ECMAScript 源文本。JSON 词法仅能识别由 JSONWhiteSpace 产生式得到的空白字符。在语法上，所有非终止符均不是由"JSON"字符开始，而是由 ECMAScript 词法产生式定义的。

语法

```

JSONWhiteSpace ::
    <tab>
    <cr>
    <lf>
    <sp>

JSONString ::
    " JSONStringCharactersopt "JSONStringCharacters ::

JSONStringCharacter
JSONStringCharactersoptJSONStringCharacter ::

SourceCharacter 但非 双引号 " 或反斜杠 \ 或 U+0000 抑或是
U+001F

\ JSONEscapeSequenceJSONEscapeSequence ::

JSONEscapeCharacter

UnicodeEscapeSequence JSONEscapeCharacter :: 以下之一

" / \ b f n r tJSONNumber ::

-optDecimalIntegerLiteral JSONFractionopt
ExponentPartoptJSONFraction ::

. DecimalDigitsJSONNullLiteral ::

NullLiteralJSONBooleanLiteral ::

BooleanLiteral

```

JSON 语法

根据 JSON 词法定义的 `tokens`, JSON 语法定义了一个合法的 JSON 文本。
语法的目标符号是 `JSONText`。

语法

```

JSONText :
    JSONValueJSONValue :

```

```

JSONNullLiteral

JSONBooleanLiteral

JSONObject

JSONArray

JSONString

JSONNumberJSONObject :

{ }

{ JSONMemberList }JSONMember :

JSONString : JSONValueJSONMemberList :

JSONMember

JSONMemberList , JSONMemberJSONArray :

[ ]

[ JSONElementList ]JSONElementList :

JSONValue

JSONElementList , JSONValue

```

parse (text [, reviver])

parse 函数解析一段 JSON 文本（JSON 格式字符串），生成一个 ECMAScript 值。JSON 格式是 ECMAScript 直接量 的受限模式。JSON 对象可以被理解为 ECMAScript 对象。JSON 数组可以被理解为 ECMAScript 数组。JSON 字符串、数字、布尔值以及 null 可以被认为是 ECMAScript 字符串、数字、布尔值以及 null。JSON 使用受限更多的空白字符集合，并且允许 Unicode 码点 U+2028 和 U+2029 直接出现在 JSONString 直接量当中而无需使用转义序列。解析流程与 11.1.4 和 11.1.5 一样，但是由 JSON 语法限定。

可选参数 **reviver** 是一个接受两个参数的函数（**key** 和 **value**）。它可以过滤和转换结果。它在每个 **key/value** 对产生时被调用，它的返回值可以用于替代原本的值。如果它原样返回接收到的，那么结构不会被改变。如果它返回 **undefined**，那么属性会被从结果中删除。

1. 令 JText 为 ToString(text)

2. 以 15.12.1 所述语法解析 JText。如果 JText 不能以 JSON grammar 解析成 JSONText，则抛出 SyntaxError 异常。
3. 令 unfiltered 为按 ECMAScript 程序(但是用 JSONString 替换 StringLiteral)解析和执行 JText 的结果。注因 JText 符合 JSON 语法，这个结果要么是原始值类型要么是 ArrayLiteral 或者 ObjectLiteral 所定义的对象。
4. 若 IsCallable(reviver)为 true 则
 1. 令 root 为由表达式 new Object()创建的新对象，其中 Object 是以 Object 为名的标准内置的构造器。
 2. 以空字符串和属性描述{Value: unfiltered, Writable: true, Enumerable: true, Configurable: true}以及 false 为参数调用 root 的 DefineOwnProperty 内置方法。
 3. 返回传入 root 和空字符串为参数调用抽象操作 Walk 的结果，抽象操作 Walk 如下文所定义。
5. 否则，返回 unfiltered。

抽象操作 Walk 是一个递归的抽象操作，它接受两个参数：一个持有对象和表示一个该对象的属性名的 String。Walk 使用最开始被传入 parse 函数的 reviver 的值。

1. 令 val 为以参数 name 调用 hold 的 get 内部方法的结果。
2. 若 val 为对象，则
 1. 若 val 的 Class 内部属性为"Array"
 - i. 设 l 为 0
 - ii. 令 len 为以参数"length"调用 val 的 get 内部方法的结果
 - iii. 当 l < len 时重复
 1. 令 newElement 为调用抽象操作 Walk 的结果，传入 val 和 ToString(l)为参数。
 2. 若 newElement 为 undefined，则
 1. 以 ToString(l)和 false 做参数，调用 val 的内部方法 Delete
 2. 否则，以 ToString(l)，属性描述{Value: newElement, Writable: true, Enumerable: true, Configurable: true}以及 false 做参数调用调用 val 的内部方法 DefineOwnProperty
 3. 对 l 加 1
 2. 否则
 - i. 令 keys 为包含 val 所有的具有 Enumerable 特性的属性名 String 值的内部类型 List。字符串的顺序应当与内置函数 Object.keys 一致。
 - ii. 对于 keys 中的每个字符串 P 做一下
 1. 令 newElement 为调用抽象操作 Walk 的结果，传入 val 和 P 为参数。
 2. 若 newElement 为 undefined，则
 1. 以 P 和 false 做参数，调用 val 的内部方法 Delete
 2. 否则，以 P，属性描述{Value: newElement, Writable: true, Enumerable: true, Configurable: true}以及 false 做参数调用调用 val 的内部方法 DefineOwnProperty

3. 返回传入 **holder** 作为 **this** 值以及以 **name** 和 **val** 构成的参数列表调用 **reviver** 的 **Call** 内部属性的结果。

实现不允许更改 **JSON.parse** 的实现以扩展 **JSON** 语法。如果一个实现想要支持更改或者扩展过的 **JSON** 交换格式它必须以定义一个不同的 **parse** 函数的方式做这件事。

在对象中存在同名字符串的情况下，同一 **key** 的值会被按照文本顺序覆盖掉。

stringify (value [, replacer [, space]])

stringify 函数返回一个 **JSON** 格式的字符串，用以表示一个 **ECMAScript** 值。它可以接受三个参数。第一个参数是必选的。**value** 参数是一个 **ECMAScript** 值，它通常是对象或者数组，尽管它也可以是 **String**, **Boolean**, **Number** 或者是 **null**。可选的 **replacer** 参数要么是个可以修改对象和数组字符串化的方式的函数，要么是个扮演选择对象字符串化的属性的白名单这样的角色的 **String** 和 **Number** 组成的数组。可选的 **space** 参数是一个 **String** 或者 **Number**，可以允许结果中插入空白符以改善人类可读性。

以下为字符串化一对象的步骤：

1. 令 **stack** 为空 **List**
2. 令 **indent** 为空 **String**
3. 令 **PropertyList** 和 **ReplacerFunction** 为 **undefined**
4. 若 **Type(replacer)** 为 **Object**, 则
 1. 若 **IsCallable(replacer)** 为 **true**, 则
 - i. 令 **ReplacerFunction** 为 **replacer**
 2. 否则若 **replacer** 的内部属性 **Class** 为 **"Array"**, 则
 - i. 令 **PropertyList** 为一空内部类型 **List**
 - ii. 对于所有名是数组下标的 **replacer** 的属性 **v**. 以数组下标递增顺序枚举属性
 1. 令 **item** 为 **undefined**
 2. 若 **Type(v)** 为 **String** 则 令 **item** 为 **v**.
 3. 否则若 **Type(v)** 为 **Number** 则 令 **item** 为 **ToString(v)**.
 4. 否则若 **Type(v)** 为 **Object** 则,
 1. 若 **v** 的 **Class** 内部属性为 **"String"** or **"Number"** 则 令 **item** 为 **ToString(v)**.
 5. 若 **item** is not **undefined** and **item** 为 not currently an element of **PropertyList** 则,
 1. **Append item** to the end of **PropertyList**.
 5. 若 **Type(space)** 为 **Object** 则,
 1. 若 **space** 的 **Class** 内部属性为 **"Number"** 则,
 - i. 令 **space** 为 **ToNumber(space)**
 2. 否则若 **space** 的 **Class** 内部属性为 **"String"** 则,
 - i. 令 **space** 为 **ToString(space)**

6. 若 `Type(space)` 为 `Number` 令 `space` 为 `min(10, ToInteger(space))` 设 `gap` 为一包含 `space` 个空格的 `String`. 这将会是空 `String` 加入 `space` 小于 1.
7. 否则若 `Type(space)` 为 `String`
 1. 若 `space` 中字符个数为 10 或者更小, 设 `gap` 为 `space`, 否则设 `gap` 为包含前 10 个 `space` 中字符的字符串
 8. 否则 设 `gap` 为空 `String`.
9. 令 `wrapper` 为一个如同以表达式 `new Object()` 创建的新对象, 其中 `Object` 是这个名字的标准内置构造器。
10. 以参数空 `String`, 属性描述 `{Value: value, Writable: true, Enumerable: true, Configurable: true}`, 和 `false` 调用 `wrapper` 的 `DefineOwnProperty` 内部方法。
11. 返回以空 `String` 和 `wrapper` 调用抽象方法 `Str` 的结果。

抽象操作 `Str(key, holder)` 可以访问调用它的 `stringify` 方法中的 `ReplacerFunction`。其算法如下：

1. 令 `value` 为以 `key` 为参数调用 `holder` 的内部方法 `Get`
2. 若 `Type(value)` 为 `Object`, 则
 1. 令 `toJSON` 为以 `"toJSON"` 为参数调用 `value` 的内部方法 `Get`
 2. If `IsCallable(toJSON)` is true
 - i. 令 `value` 为以调用 `toJSON` 的内部方法 `Call` 的结果, 传入 `value` 为 `this` 值以及由 `key` 构成的参数列表
3. 若 `ReplacerFunction` 为 `not undefined`, 则
 1. 令 `value` 为以调用 `ReplacerFunction` 的内部方法 `Call` 的结果, 传入 `holder` 为 `this` 值以及由 `key` and `value` 构成的参数列表
4. 若 `Type(value)` 为 `Object` 则,
 1. 若 `value` 的 `Class` 内部属性为 `"Number"` 则,
 - i. 令 `value` 为 `ToNumber(value)`
 2. 否则若 `value` 的 `Class` 内部属性为 `"String"` 则,
 - i. 令 `value` 为 `ToString(value)`
 3. 否则若 `value` 的 `Class` 内部属性为 `"Boolean"` 则,
 - i. 令 `value` 为 `value` 的 `value of the PrimitiveValue` 内部属性
5. 若 `value` 为 `null` 则 返回 `"null"`.
6. 若 `value` 为 `true` 则 返回 `"true"`.
7. 若 `value` 为 `false` 则 返回 `"false"`.
8. 若 `Type(value)` 为 `String`, 则返回以 `value` 调用 `Quote` 抽象操作的结果。
9. 若 `Type(value)` 为 `Number`
 1. 若 `value` 为 `finite` 则 返回 `ToString(value)`.
 2. 否则, 返回 `"null"`.
10. 若 `Type(value)` 为 `Object`, 且 `IsCallable(value)` 为 `false`
 1. 若 `value` 的 `Class` 内部属性为 `"Array"` 则
 - i. 返回以 `value` 为参数调用抽象操作 `JA` 的结果
 2. 否则, 返回以 `value` 为参数调用抽象操作 `value` 的结果
11. 返回 `undefined`.

抽象操作 **Quote(value)** 将一个 **String** 值封装在双引号中，并且对其中的字符转义。

1. 令 **product** 为双引号字符
2. 对 **value** 中的每一个字符 **C**
 1. 若 **C** 为双引号字符或者反斜杠字符
 - i. 令 **product** 为 **product** 和反斜杠连接的结果
 - ii. 令 **product** 为 **product** 与 **C** 的连接
 2. 否则若 **C** 为退格, **formfeed** 换行回车或者 **tab**
 - i. 令 **product** 为 **product** 与反斜杠字符的连接
 - ii. 令 **abbrev** 为如下表所示 **C** 对应的字符: **backspace** "b" **formfeed** "f" **newline** "n" **carriage return** "r" **tab** "t"
 - iii. 令 **product** 为 **product** 与 **abbrev** 的连接
3. 否则若 **C** 为代码值小于 **space** 的控制字符
 - i. 令 **product** 为 **product** 与反斜杠字符的连接
 - ii. 令 **product** 为 **product** 与 "u" 的连接
 - iii. 令 **hex** 为转换 **C** 代码值按十六进制转换到四位字符串的结果
 - iv. 令 **product** 为 **product** 与 **hex** 的连接
4. 否则
 - i. 令 **product** 为 **product** 与 **C** 的连接
3. 令 **product** 为 **product** 与双引号字符的连接
4. 返回 **product**.

抽象操作 **JO(value)** 序列化一个对象，它可以访问调用它的方法中的 **stack**, **indent**, **gap**, **PropertyList**, **ReplacerFunction** 以及 **space**

1. 若 **stack** 包含 **value**，则抛出一个 **TypeError**，因为对象结构中存在循环。
2. 将 **value** 添加到 **stack**.
3. 令 **stepback** 为 **indent**
4. 令 **indent** 为 **indent** 与 **gap** 的连接
5. 若 **PropertyList** 没有被定义, 则
 1. 令 **K** 为 **PropertyList**
6. 否则
 1. 令 **K** 为以由所有 **Enumerable** 特性为 **true** 的自身属性名构成的内部 **String** 列表类型.
7. 令 **partial** 为空 **List**
8. 对于 **K** 的每一个元素 **P**
 1. 令 **strP** 为以 **P** 和 **value** 为参数调用抽象操作 **Str** 的结果
 2. 若 **strP** 没有被定义
 - i. 令 **member** 为以 **P** 为参数调用抽象操作 **P** 的结果
 - ii. 令 **member** 为 **member** 与冒号字符的连接
 - iii. 若 **gap** 不为空 **String**
 - iv. 令 **member** 为 **member** 与空格字符的连接
 - v. 令 **member** 为 **member** 与 **strP** 的连接
 - vi. 将 **member** 添加到 **partial**.

9. 若 **partial** 为 **empty**,则
 1. 令 **final** 为 "{}"
10. 否则
 1. 若 **gap** 为空 **String**
 - i. 令 **properties** 为一个连接所有 **partial** 中的字符串而成的字符串，键值对之间用逗号分隔。第一个字符串之前和最后一个字符串之后没有逗号。
 - ii. 令 **final** 为连接 "{", **properties**, 和 "}" 的结果
 2. 否则 **gap** 不是空 **String**
 - i. 令 **separator** 为连接 逗号字符，换行字符以及 **indent** 而成的字符串。
 - ii. 令 **properties** 为一个连接所有 **partial** 中的字符串而成的字符串，键值对之间用 **separator** 分隔。第一个字符串之前和最后一个字符串之后没有 **separator**。
 - iii. 令 **final** 为连接 "{", 换行符, **indent**, **properties**, 换行符, **stepback**, 和 "}" 的结果。
11. 移除 **stack** 中的最后一个元素。
12. 令 **indent** 为 **stepback**
13. 返回 **final**.

抽象操作 **JA(value)** 序列化一个数组。它可以访问调用它的 **stringify** 方法中的 **stack**, **indent**, **gap**, **PropertyList**, **ReplacerFunction** 以及 **space**。数组的表示中仅包扩零到 **array.length-1** 的区间。命名的属性将会被从字符串化操作中排除。数组字符串化成开头的左方括号，逗号分隔的元素，以及结束的右方括号。

1. 若 **stack** 包含 **value**，则抛出一个 **TypeError**，因为对象结构中存在循环。
2. 将 **value** 添加到 **stack**.
3. 令 **stepback** 为 **indent**
4. 令 **indent** 为 **indent** 与 **gap** 的连接
5. 令 **partial** 为空 **List**
6. 令 **len** 为以 "length" 为参数调用 **value** 的内部方法 **Get**
7. 令 **index** 为 0
8. 当 **index < len** 时重复以下
 1. 令 **strP** 为 the result of calling the abstract operation **Str** with arguments **ToString(index)** and **value**
 2. 若 **strP** 没有被定义
 - i. 添加 **null** 到 **partial**.
 3. 否则
 - i. 添加 **strP** 到 **partial**.
 4. 使 **index** 增加 1.
9. 若 **partial** 为空,则
 1. 令 **final** 为 "[]"
10. 否则
 1. 若 **gap** 为空 **String**
 - i. 令 **properties** 为为一个连接所有 **partial** 中的字符串而成的字符串，键值对之间用逗号分隔。第一个字符串之前和最后一个字符串之后没有逗号。

- ii. 令 **final** 为连接 "[", **properties**, 和 "]" 的结果
2. 否则
 - i. 令 **separator** 为逗号字符, 换行字符以及 **indent** 而成的字符串。
 - ii. 令 **properties** 为一个连接所有 **partial** 中的字符串而成的字符串, 键值对之间用 **separator** 分隔。第一个字符串之前和最后一个字符串之后没有 **separator**。
 - iii. 令 **final** 为连接 "[", 换行符, **indent**, **properties**, 换行符, **stepback**, 和 "]" 的结果。
11. 移除 **stack** 中的最后一个元素。
12. 令 **indent** 为 **stepback**
13. 返回 **final**.

JSON 结构允许任何深度的嵌套, 但是不能够循环引用。若 **value** 是或者包含了一个循环结构, 则 **stringify** 函数必须抛出一个 **TypeError** 异常。以下是一个不能够被字符串化的值的例子:

```
a = []; a[0] = a; my_text = JSON.stringify(a); // This must throw an
TypeError.
```

符号式简单值按以下方式表示:

1. **null** 值在 JSON 文本中表示为字符串 **null**
2. **undefined** 值不出现
3. **true** 值在 JSON 文本中表示为字符串 **true**
4. **false** 值在 JSON 文本中表示为字符串 **false**

字符串值用双引号括起。字符"和\会被转义成带\前缀的。控制字符用转义序列 \uHHHH 替换, 或者使用简略形式 \b(backspace), \f(formfeed), \n(newline), \r(carriage return), \t(tab)

有穷的数字按照调用 **ToString(number)** 字符串化。**NaN** 和不论正负的 **Infinity** 都表示为字符串 **null**

没有 JSON 表示的值(如 **undefined** 和函数)不会产生字符串。而是会产生 **undefined** 值。在数组中这些值表示为字符串 **null**。在对象中不能表示的值会导致属性被排除在字符串化过程之外。

对象表示为开头的左大括号跟着零个或者多个属性, 以逗号分隔, 以右大括号结束。属性是用用来表示 **key** 或者属性名的引号引起的字符串, 冒号然后是字符串化的属性值。数组表示为开头的左方括号, 后跟零个或者多个值, 以逗号分隔, 以右方括号结束。

错误

在 ECMAScript 相关语言构造被求值之时，实现报告大部分错误。早期的错误是一种可以检测和优先报告程序中所有错误内任何构造的求值问题。具体实现一定要在一个程序首次执行评估时报告早期错误。早期错误在 `eval` 被调用时报告 `eval` 错误代码，但是在 `eval` 代码内之优先评估任意构造。

一个实现要处理任意实例中的以下几种误差作为早期错误：

1. 任意语法错误
2. 试图定义一个有多个相同名字的 `get` 属性设置或多个相同名字的 `set` 属性设置的 对象字面量
3. 试图定义一个数据属性设置并且 `get` 或 `set` 具有相同的名称属性设置的 对象字面量
4. 错误在正则表达式字面量中没有实现语法扩展
5. 试图在严格代码模式下定义一个有多个相同名称属性设置数据的 对象字面量
6. `with` 语句在严格代码模式下出现
7. 在严格模式下的函数定义或函数表达的参数列表内不止一次出现标识符值的情况
8. 使用 `return`, `break` 和 `continue` 不当
9. 试图在早期已经确定为非引用的任意值上调用 `PutValue`（例如，执行赋值语句 `3 = 4`）

一个实现不应过早处理其他种类的错误，即使编译器可以证实某一构造会在任何情况下产生执行错误。

一个实现应报告所有的指定错误，但以下情况除外：

1. 实现可以扩展程序语法和正则表达式或标志语法。使用此功能，当它们遇到一个实现程序语法定义扩展或正则表达式或标记语法时，所有操作（如调用 `eval`，使用正则表达式字面，或使用 `Function` 或 `RegExp` 构造）被获准展现实现定义扩展的行为，而非抛出 `SyntaxError`。
2. 一个实现可以提供超出本规范中所描述的功能范围外的类型，值，对象，属性。这可能会导致构造（如寻找一个在全局作用域内的变量）实现定义的行为而非抛出一个错误（如 `ReferenceError`）。
3. 当在 `fractionDigits` 或 `precision` 参数是在指定的范围之外，一个实现可以为 `toFixed`, `toExponential`, 和 `toPrecision` 定义 `RangeError` 以外的其他行为。

文法摘要

词法

```

SourceCharacter :: 任意 Unicode 编码单元 InputElementDiv :: WhiteSpace
LineTerminator Comment Token DivPunctuator InputElementRegExp ::
WhiteSpace LineTerminator Comment Token RegularExpressionLiteral
WhiteSpace :: <TAB> <VT> <FF> <SP> <#x0a> <BOM> <USP> LineTerminator ::
<LF> <CR> <LS> <PS> LineTerminatorSequence :: <LF> <CR> [lookahead ≠ ]
<LS> <PS> <CR> <LF> Comment :: MultiLineComment SingleLineComment
MultiLineComment :: /* MultiLineCommentCharsopt */
MultiLineCommentChars :: MultiLineNotAsteriskChar
MultiLineCommentCharsopt * PostAsteriskCommentCharsopt
PostAsteriskCommentChars :: MultiLineNotForwardSlashorAsteriskChar
MultiLineCommentCharsopt * PostAsteriskCommentCharsopt
MultiLineNotAsteriskChar :: SourceCharacter 但非 星号 *
MultiLineNotForwardSlashorAsteriskChar :: SourceCharacter 但非 正斜杠
/ 或 星号 * SingleLineComment :: // SingleLineCommentCharsopt
SingleLineCommentChars :: SingleLineCommentChar
SingleLineCommentCharsopt SingleLineCommentChar :: SourceCharacter 但
非 LineTerminator Token :: IdentifierName Punctuator NumericLiteral
StringLiteral Identifier :: IdentifierName 但非 ReservedWord
IdentifierName :: IdentifierStart IdentifierName IdentifierPart
IdentifierStart :: UnicodeLetter $ _ \ UnicodeEscapeSequence
IdentifierPart :: IdentifierStart UnicodeCombiningMark UnicodeDigit
UnicodeConnectorPunctuation <ZWNJ> <ZWJ> UnicodeLetter 在以下 Unicode
分类中的字符: “Uppercase letter (Lu)”, “Lowercase letter (Ll)”,
“Titlecase letter (Lt)”, “Modifier letter (Lm)”, “Other letter
(Lo)”, 或 “Letter number (Nl)”. UnicodeCombiningMark 在以下 Unicode
分类中的字符: “Non-spacing mark (Mn)” 或 “Combining spacing mark (Mc)”
UnicodeDigit 在以下 Unicode 分类中的字符: “Decimal number (Nd)”
UnicodeConnectorPunctuation 在以下 Unicode 分类中的字符: “Connector
punctuation (Pc)” ReservedWord :: Keyword FutureReservedWord
NullLiteral BooleanLiteral Keyword :: 以下之一 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
FutureReservedWord :: 以下之一 class enum extends super const export
import 或在严格模式下以下之一 implements let private public interface
package protected static yield Punctuator :: 以下之一 { } ( ) [ ] . ; ,
< > <= >= == != === !== + - * % ++ -- << >> >>> & | ^ ! ~ && || ? : = +=
-= *= %= <<= >>= >>>= &= |= ^= DivPunctuator :: 以下之一 / /= Literal ::
NullLiteral BooleanLiteral NumericLiteral StringLiteral
RegularExpressionLiteral NullLiteral :: null BooleanLiteral :: true
false NumericLiteral :: DecimalLiteral HexIntegerLiteral
DecimalLiteral :: DecimalIntegerLiteral . DecimalDigitsopt
ExponentPartopt . DecimalDigits ExponentPartopt DecimalIntegerLiteral
ExponentPartopt DecimalIntegerLiteral :: 0 NonZeroDigit
DecimalDigitsopt DecimalDigits :: DecimalDigit DecimalDigits

```

```

DecimalDigit DecimalDigit :: 以下之一 0 1 2 3 4 5 6 7 8 9 NonZeroDigit::
以下之一 1 2 3 4 5 6 7 8 9 ExponentPart:: ExponentIndicator SignedInteger
ExponentIndicator :: 以下之一 e E SignedInteger :: DecimalDigits +
DecimalDigits - DecimalDigits HexIntegerLiteral :: 0x HexDigit 0X
HexDigit HexIntegerLiteral HexDigit HexDigit :: 以下之一 0 1 2 3 4 5 6
7 8 9 a b c d e f A B C D E F StringLiteral :: "DoubleStringCharactersopt
" 'SingleStringCharactersopt' DoubleStringCharacters ::
DoubleStringCharacter DoubleStringCharactersopt
SingleStringCharacters :: SingleStringCharacter
SingleStringCharactersopt DoubleStringCharacter :: SourceCharacter 但
非 double-quote " 或 backslash \ 或 LineTerminator \ EscapeSequence
LineContinuation SingleStringCharacter :: SourceCharacter 但非
single-quote ' 或 backslash \ 或 LineTerminator \ EscapeSequence
LineContinuation LineContinuation :: \ LineTerminatorSequence
EscapeSequence :: CharacterEscapeSequence 0 [lookahead & DecimalDigit]
HexEscapeSequence UnicodeEscapeSequence CharacterEscapeSequence ::
SingleEscapeCharacter NonEscapeCharacter SingleEscapeCharacter :: 以下
之一 ' " \ b f n r t v NonEscapeCharacter :: SourceCharacter 但非
EscapeCharacter 或 LineTerminator EscapeCharacter ::
SingleEscapeCharacter DecimalDigit x u HexEscapeSequence :: x HexDigit
HexDigit UnicodeEscapeSequence :: u HexDigit HexDigit HexDigit HexDigit
RegularExpressionLiteral :: / RegularExpressionBody /
RegularExpressionFlags RegularExpressionBody ::
RegularExpressionFirstChar RegularExpressionChars
RegularExpressionChars :: [empty] RegularExpressionChars
RegularExpressionChar RegularExpressionFirstChar ::
RegularExpressionNonTerminator 但非 * 或 \ 或 / 或
[ RegularExpressionBackslashSequence RegularExpressionClass
RegularExpressionChar :: RegularExpressionNonTerminator 但非 \ 或 / 或
[ RegularExpressionBackslashSequence RegularExpressionClass
RegularExpressionBackslashSequence :: \ RegularExpressionNonTerminator
RegularExpressionNonTerminator :: SourceCharacter 但非 LineTerminator
RegularExpressionClass :: [ RegularExpressionClassChars ]
RegularExpressionClassChars :: [ 空 ] RegularExpressionClassChars
RegularExpressionClassChar RegularExpressionClassChar ::
RegularExpressionNonTerminator 但非 ] 或 \
RegularExpressionBackslashSequence RegularExpressionFlags :: [ 空 ]
RegularExpressionFlags IdentifierPart

```

数字转换

```

StringNumericLiteral ::: StrWhiteSpaceopt
StrWhiteSpaceoptStrNumericLiteral StrWhiteSpaceopt StrWhiteSpace :::

```

```

StrWhiteSpaceChar StrWhiteSpaceopt StrWhiteSpaceChar ::: WhiteSpace
LineTerminator StrNumericLiteral ::: StrDecimalLiteral
HexIntegerLiteral StrDecimalLiteral ::: StrUnsignedDecimalLiteral +
StrUnsignedDecimalLiteral - StrUnsignedDecimalLiteral
StrUnsignedDecimalLiteral ::: Infinity DecimalDigits . DecimalDigitsopt
ExponentPartopt . DecimalDigits ExponentPartopt DecimalDigits
ExponentPartopt DecimalDigits ::: DecimalDigit DecimalDigits
DecimalDigit DecimalDigit ::: 以下之一 0 1 2 3 4 5 6 7 8 9 ExponentPart :::
ExponentIndicator SignedInteger ExponentIndicator ::: 以下之一 e E
SignedInteger ::: DecimalDigits + DecimalDigits - DecimalDigits
HexIntegerLiteral ::: 0x HexDigit 0X HexDigit HexIntegerLiteral HexDigit
HexDigit ::: 以下之一 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

```

表达式

```

PrimaryExpression : this Identifier Literal ArrayLiteral ObjectLiteral
(Expression) ArrayLiteral : [ Elisionopt ] [ ElementList ] [ ElementList ,
Elisionopt ] ElementList : Elisionopt AssignmentExpression ElementList ,
Elisionopt AssignmentExpression Elision : , Elision , ObjectLiteral : { }
{ PropertyNameAndValueList } { PropertyNameAndValueList , }
PropertyNameAndValueList : PropertyAssignment PropertyNameAndValueList ,
PropertyAssignment PropertyAssignment : PropertyName :
AssignmentExpression get PropertyName() { FunctionBody } set
PropertyName( PropertySetParameterList ) { FunctionBody } PropertyName :
IdentifierName StringLiteral NumericLiteral PropertySetParameterList :
Identifier MemberExpression : PrimaryExpression FunctionExpression
MemberExpression [ Expression ] MemberExpression . IdentifierName new
MemberExpression Arguments NewExpression : MemberExpression new
NewExpression CallExpression : MemberExpression Arguments
CallExpression Arguments CallExpression [ Expression ] CallExpression .
IdentifierName Arguments : ( ) ( ArgumentList ) ArgumentList :
AssignmentExpression ArgumentList , AssignmentExpression
LeftHandSideExpression : NewExpression CallExpression
PostfixExpression : LeftHandSideExpression LeftHandSideExpression [ 此
处无换行 ] ++ LeftHandSideExpression [ 此处无换行 ] -- UnaryExpression :
PostfixExpression delete UnaryExpression void UnaryExpression typeof
UnaryExpression ++ UnaryExpression -- UnaryExpression + UnaryExpression
- UnaryExpression ~ UnaryExpression ! UnaryExpression
MultiplicativeExpression : UnaryExpression MultiplicativeExpression *
UnaryExpression MultiplicativeExpression / UnaryExpression
MultiplicativeExpression % UnaryExpression AdditiveExpression :
MultiplicativeExpression AdditiveExpression + MultiplicativeExpression
AdditiveExpression - MultiplicativeExpression ShiftExpression :

```



```

AdditiveExpression ShiftExpression << AdditiveExpression
ShiftExpression >> AdditiveExpression ShiftExpression >>>
AdditiveExpression RelationalExpression : ShiftExpression
RelationalExpression < ShiftExpression RelationalExpression >
ShiftExpression RelationalExpression <= ShiftExpression
RelationalExpression >= ShiftExpression RelationalExpression instanceof
ShiftExpression RelationalExpression in ShiftExpression
RelationalExpressionNoIn : ShiftExpression RelationalExpressionNoIn <
ShiftExpression RelationalExpressionNoIn > ShiftExpression
RelationalExpressionNoIn <= ShiftExpression RelationalExpressionNoIn >=
ShiftExpression RelationalExpressionNoIn instanceof ShiftExpression
EqualityExpression : RelationalExpression EqualityExpression ==
RelationalExpression EqualityExpression != RelationalExpression
EqualityExpression === RelationalExpression EqualityExpression !==
RelationalExpression EqualityExpressionNoIn : RelationalExpressionNoIn
EqualityExpressionNoIn == RelationalExpressionNoIn
EqualityExpressionNoIn != RelationalExpressionNoIn
EqualityExpressionNoIn === RelationalExpressionNoIn
EqualityExpressionNoIn !== RelationalExpressionNoIn
BitwiseANDExpression : EqualityExpression BitwiseANDExpression &
EqualityExpression BitwiseANDExpressionNoIn : EqualityExpressionNoIn
BitwiseANDExpressionNoIn & EqualityExpressionNoIn BitwiseXORExpression :
BitwiseANDExpression BitwiseXORExpression ^ BitwiseANDExpression
BitwiseXORExpressionNoIn : BitwiseANDExpressionNoIn
BitwiseXORExpressionNoIn ^ BitwiseANDExpressionNoIn
BitwiseORExpression : BitwiseXORExpression BitwiseORExpression |
BitwiseXORExpression BitwiseORExpressionNoIn :
BitwiseXORExpressionNoIn BitwiseORExpressionNoIn |
BitwiseXORExpressionNoIn LogicalANDExpression : BitwiseORExpression
LogicalANDExpression && BitwiseORExpression LogicalANDExpressionNoIn :
BitwiseORExpressionNoIn LogicalANDExpressionNoIn &&
BitwiseORExpressionNoIn LogicalORExpression : LogicalANDExpression
LogicalORExpression || LogicalANDExpression LogicalORExpressionNoIn :
LogicalANDExpressionNoIn LogicalORExpressionNoIn ||
LogicalANDExpressionNoIn ConditionalExpression : LogicalORExpression
LogicalORExpression ? AssignmentExpression : AssignmentExpression
ConditionalExpressionNoIn : LogicalORExpressionNoIn
LogicalORExpressionNoIn ? AssignmentExpressionNoIn :
AssignmentExpressionNoIn AssignmentExpression : ConditionalExpression
LeftHandSideExpression AssignmentOperator AssignmentExpression
AssignmentExpressionNoIn : ConditionalExpressionNoIn
LeftHandSideExpression AssignmentOperator AssignmentExpressionNoIn
AssignmentOperator : 以下之一 = *= /= %= += -= <<= >>= >>>= &= ^= |=
Expression : AssignmentExpression Expression , AssignmentExpression

```

ExpressionNoIn : AssignmentExpressionNoIn ExpressionNoIn ,
AssignmentExpressionNoIn

语句

Statement : Block VariableStatement EmptyStatement ExpressionStatement
IfStatement IterationStatement ContinueStatement BreakStatement
ReturnStatement WithStatement LabelledStatement SwitchStatement
ThrowStatement TryStatement DebuggerStatement Block :
{ StatementListopt } StatementList : Statement StatementList Statement
VariableStatement : var VariableDeclarationList ;
VariableDeclarationList : VariableDeclaration VariableDeclarationList ,
VariableDeclaration VariableDeclarationListNoIn :
VariableDeclarationNoIn VariableDeclarationListNoIn ,
VariableDeclarationNoIn VariableDeclaration : Identifier Initialiseropt
VariableDeclarationNoIn : Identifier InitialiserNoInopt Initialiser : =
AssignmentExpression InitialiserNoIn : = AssignmentExpressionNoIn
EmptyStatement : ; ExpressionStatement : [lookahead \notin { {,
function} }] Expression ; IfStatement : if (Expression) Statement else
Statement if (Expression) Statement IterationStatement : do Statement
while (Expression) ; while (Expression) Statement for
(ExpressionNoInopt ; Expressionopt ; Expressionopt) Statement for (var
VariableDeclarationListNoIn ; Expressionopt ; Expressionopt) Statement
for (LeftHandSideExpression in Expression) Statement for (var
VariableDeclarationNoIn in Expression) Statement ContinueStatement :
continue [此处无换行] Identifieropt ; BreakStatement : break [此处无
换行] Identifieropt ; ReturnStatement : return [此处无换
行] Expressionopt ; WithStatement : with (Expression) Statement
SwitchStatement : switch (Expression) CaseBlock CaseBlock :
{ CaseClausesopt } { CaseClausesopt DefaultClause CaseClausesopt }
CaseClauses : CaseClause CaseClauses CaseClause CaseClause : case
Expression : StatementListopt DefaultClause : default : StatementListopt
LabelledStatement : Identifier : Statement ThrowStatement : throw
[noLineTerminator here] Expression ; TryStatement : try Block Catch try
Block Finally try Block Catch Finally Catch : catch (Identifier) Block
Finally : finally Block DebuggerStatement : debugger ;

函数和程序

FunctionDeclaration : function Identifier (FormalParameterListopt)
{ FunctionBody } FunctionExpression : function Identifieropt
(FormalParameterListopt) { FunctionBody } FormalParameterList :
Identifier FormalParameterList , Identifier FunctionBody :

```
SourceElementsopt Program : SourceElementsopt SourceElements :
SourceElement SourceElements SourceElement SourceElement : Statement
FunctionDeclaration
```

统一资源定位符字符分类

```
uri ::: uriCharactersopt uriCharacters ::: uriCharacter uriCharactersopt
uriCharacter ::: uriReserved uriUnescaped uriEscaped uriReserved ::: 以下之一 ; / ? : @ & = + $ , uriUnescaped ::: uriAlpha DecimalDigit uriMark
uriEscaped ::: % HexDigit HexDigit uriAlpha ::: 以下之一 a b c d e f g
h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q
R S T U V W X Y Z uriMark ::: 以下之一 - _ . ! ~ * ' ( )
```

正则表达式

```
Pattern :: Disjunction Disjunction :: Alternative Alternative |
Disjunction Alternative :: [ 空 ] Alternative Term Term :: Assertion Atom
Atom Quantifier Assertion :: ^ $ \ b \ B ( ? = Disjunction ) ( ? !
Disjunction ) Quantifier :: QuantifierPrefix QuantifierPrefix ?
QuantifierPrefix :: * + ? { DecimalDigits } { DecimalDigits , }
{ DecimalDigits , DecimalDigits } Atom :: PatternCharacter . \ AtomEscape
CharacterClass ( Disjunction ) ( ? : Disjunction ) PatternCharacter ::
SourceCharacter 但非以下之一 : ^ $ \ . * + ? ( ) [ ] { } | AtomEscape ::
DecimalEscape CharacterEscape CharacterClassEscape CharacterEscape ::
ControlEscape c ControlLetter HexEscapeSequence UnicodeEscapeSequence
IdentityEscape ControlEscape :: 以下之一 f n r t v ControlLetter :: 以下之一 a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F
G H I J K L M N O P Q R S T U V W X Y Z IdentityEscape :: SourceCharacter
but not IdentifierPart DecimalEscape :: DecimalIntegerLiteral [lookahead
⊥ DecimalDigit] CharacterClassEscape :: 以下之一 d D s S w W
CharacterClass :: [ [lookahead ⊥ {^}] ClassRanges ] [ ^ ClassRanges ]
ClassRanges :: [empty] NonemptyClassRanges NonemptyClassRanges ::
ClassAtom ClassAtom NonemptyClassRangesNoDash ClassAtom - ClassAtom
ClassRanges NonemptyClassRangesNoDash :: ClassAtom ClassAtomNoDash
NonemptyClassRangesNoDash ClassAtomNoDash - ClassAtom ClassRanges
ClassAtom :: - ClassAtomNoDash ClassAtomNoDash :: SourceCharacter 但非
以下之一 \ 或 ] 或 - \ ClassEscape ClassEscape :: DecimalEscape b
CharacterEscape CharacterClassEscape
```

JSON

JSON 词法

```
JSONWhiteSpace :: <TAB> <CR> <LF> <SP> JSONString :: "  
JSONStringCharactersopt " JSONStringCharacters :: JSONStringCharacter  
JSONStringCharactersopt JSONStringCharacter :: SourceCharacter 但非 双  
引号 " 或反斜杠 \ 或 U+0000 抑或是 U+001F \ JSONEscapeSequence  
JSONEscapeSequence :: JSONEscapeCharacter UnicodeEscapeSequence  
JSONEscapeCharacter :: 以下之一 " / \ b f n r t JSONNumber ::  
-optDecimalIntegerLiteral JSONFractionopt ExponentPartopt  
JSONFraction :: . DecimalDigits JSONNullLiteral :: NullLiteral  
JSONBooleanLiteral :: BooleanLiteral
```

JSON 语法

```
JSONText : JSONValue JSONValue : JSONNullLiteral JSONBooleanLiteral  
JSONObject JSONArray JSONString JSONNumber JSONObject : { }  
{ JSONMemberList } JSONMember : JSONString : JSONValue JSONMemberList :  
JSONMember JSONMemberList , JSONMember JSONArray : [ ] [ JSONElementList ]  
JSONElementList : JSONValue JSONElementList , JSONValue
```

兼容性

附加语法

ECMAScript 的过去版本中还包含了说明八进制直接量和八进制转义序列的额外语法、语义。在此版本中已将这些附加语法、语义移除。这个非规范性的附录给出与八进制直接量和八进制转义序列一致的语法、语义，以兼容某些较老的 **ECMAScript** 程序。

数字直接量

7.8.3 中的语法、语义可以做如下扩展，但在严格模式代码里不允许做这样的扩展。

语法

```
NumericLiteral :: DecimalLiteral HexIntegerLiteral OctalIntegerLiteral  
OctalIntegerLiteral :: 0 OctalDigit OctalIntegerLiteral OctalDigit  
OctalDigit :: 以下之一 0 1 2 3 4 5 6 7
```

语义

- `NumericLiteral :: OctalIntegerLiteral` 的 MV 是 `OctalIntegerLiteral` 的 MV。
- `OctalDigit :: 0` 的 MV 是 0。
- `OctalDigit :: 1` 的 MV 是 1。
- `OctalDigit :: 2` 的 MV 是 2。
- `OctalDigit :: 3` 的 MV 是 3。
- `OctalDigit :: 4` 的 MV 是 4。
- `OctalDigit :: 5` 的 MV 是 5。
- `OctalDigit :: 6` 的 MV 是 6。
- `OctalDigit :: 7` 的 MV 是 7。
- `OctalIntegerLiteral :: 0 OctalDigit` 的 MV 是 `OctalDigit` 的 MV。
- `OctalIntegerLiteral :: OctalIntegerLiteral OctalDigit` 的 MV 是 `OctalIntegerLiteral` 的 MV 乘以 8 再加上 `OctalDigit` 的 MV。

字符串直接量

7.8.4 中的语法、语义可以做如下扩展，但在严格模式代码里不允许做这样的扩展。

语法

```
EscapeSequence :: CharacterEscapeSequence OctalEscapeSequence  
HexEscapeSequence UnicodeEscapeSequence OctalEscapeSequence ::  
OctalDigit [lookahead &#x2194; DecimalDigit] ZeroToThree OctalDigit [lookahead  
&#x2194; DecimalDigit] FourToSeven OctalDigit ZeroToThree OctalDigit OctalDigit  
ZeroToThree :: 以下之一 0 1 2 3 FourToSeven :: 以下之一 4 5 6 7
```

语义

- `EscapeSequence :: OctalEscapeSequence` 的 CV 是 `OctalEscapeSequence` 的 CV。
- `OctalEscapeSequence :: OctalDigit [lookahead ↔ DecimalDigit]` 的 CV 是个字符，它的 unicode 代码单元值是 `OctalDigit` 的 MV。
- `OctalEscapeSequence :: ZeroToThree OctalDigit [lookahead ↔ DecimalDigit]` 的 CV 是个字符，它的 unicode 代码单元值是 `ZeroToThree` 的 MV 乘以 8 再加上 `OctalDigit` 的 MV。
- `OctalEscapeSequence :: FourToSeven OctalDigit` 的 CV 是个字符，它的 unicode 代码单元值是 `FourToSeven` 的 MV 乘以 8 再加上 `OctalDigit` 的 MV。
- `OctalEscapeSequence :: ZeroToThree OctalDigit OctalDigit` 的 CV 是个字符，它的 unicode 代码单元值是 (`ZeroToThree` 的 MV 乘以 $64 (8^2)$) 加上 (第一个 `OctalDigit` 的 MV 乘以 8) 加上 `OctalDigit` 的 MV。

- ZeroToThree :: 0 的 MV 是 0。
- ZeroToThree :: 1 的 MV 是 1。
- ZeroToThree :: 2 的 MV 是 2。
- ZeroToThree :: 3 的 MV 是 3。
- FourToSeven :: 4 的 MV 是 4。
- FourToSeven :: 5 的 MV 是 5。
- FourToSeven :: 6 的 MV 是 6。
- FourToSeven :: 7 的 MV 是 7。

附加属性

一些 ECMAScript 的实现可能会包含一些标准原生对象上的附加属性。这一非标准附录提示了一些这样的属性常见的语义，但是这并不意味着他们和其语义成为标准的一部分。

escape(string)

escape 函数是全局对象的一个属性。它通过将一些字符替换成十六进制转义序列，计算出一个新字符串值。

对于代码单元小于等于 0xFF 的被替换字符，使用 %xx 格式的两位数转义序列。对于代码单元大于 0xFF 的被替换字符，使用 %uxxxx 格式的四位数转义序列。

用一个参数 **string** 调用 **escape** 函数，采用以下步骤：

1. 调用 ToString(string)。
2. 计算 Result(1) 的字符数。
3. 令 R 为空字符串。
4. 令 k 为 0。
5. 如果 k 等于 Result(2)，返回 R。
6. 获得 Result(1) 中 k 位置的字符（表示为 16 位无符号整数）。
7. 如果 Result(6) 是 69 个非空字符 "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789@*_+-./" 之一，则转到步骤 13。
8. 如果 Result(6) 小于 256，则转到步骤 11。
9. 令 S 为包含六个字符 "%u wxyz" 的字符串，其中 wxyz 是用四个十六进制数字编码的 Result(6) 值。
10. 转到步骤 14。
11. 令 S 为包含三个字符 "% xy" 的字符串，其中 xy 是用两个十六进制数字编码的 Result(6) 值。
12. 转到步骤 14。
13. 令 S 为包含单个字符 Result(6) 的字符串。
14. 令 R 为将之前的 R 和 S 值连起来组成的新字符串。

- 15.k 递增 1。
- 16.转到步骤 5。

这里的编码方式有部分是基于 [RFC 1738](#) 描述的编码方式，但本标准规定的完整编码方式只有上面描述的这些，不考虑 [RFC 1738](#) 中的内容。此编码方式并没有反映出从 [RFC 1738](#) 到 [RFC 3986](#) 的变化。

unescape(string)

`unescape` 函数是全局对象的一个属性。它通过将每个可能是 `escape` 函数导入的转义序列，分别替换成代表这些转义序列的字符，计算出一个新字符串值。

用一个参数 `string` 调用 `unescape` 函数，采用以下步骤：

1. 调用 `ToString(string)`。
2. 计算 `Result(1)` 的字符数。
3. 令 `R` 为空字符串。
4. 令 `k` 为 0。
5. 如果 `k` 等于 `Result(2)`，返回 `R`。
6. 令 `c` 为 `Result(1)` 中 `k` 位置的字符。
7. 如果 `c` 不是 `%`，转到步骤 18。
8. 如果 `k` 大于 `Result(2)-6`，转到步骤 14。
9. 如果 `Result(1)` 中 `k+1` 位置的字符不是 `u`，转到步骤 14。
10. 如果 `Result(1)` 中分别在 `k+2`，`k+3`，`k+4`，`k+5` 位置的四个字符不全是十六进制数字，转到步骤 14。
11. 令 `c` 为一个字符，它的代码单元值是 `Result(1)` 中 `k+2`，`k+3`，`k+4`，`k+5` 位置的四个十六进制数字代表的整数。
- 12.k 递增 5。
- 13.转到步骤 18。
- 14.如果 `k` 大于 `Result(2)-3`，转到步骤 18。
- 15.如果 `Result(1)` 中分别在 `k+1`，`k+2` 位置的两个字符不都是十六进制数字，转到步骤 18。
- 16.令 `c` 为一个字符，它的代码单元值是两个零加上 `Result(1)` 中 `k+1`，`k+2` 位置的两个十六进制数字代表的整数。
- 17.k 递增 2。
- 18.令 `R` 为将之前的 `R` 和 `c` 值连起来组成的新字符串。
- 19.k 递增 1。
- 20.转到步骤 5。

String.prototype.substr(start, length)

`substr` 方法有两个参数 `start` 和 `length`，将 `this` 对象转换为一个字符串，并返回这个字符串中从 `start` 位置一直到 `length` 位置（或如果 `length` 是 `undefined`，就一直到字符串结束位置）的字符组成的子串。如果 `start` 是负数，那么它就被当作是 `(sourceLength+start)`，这里的 `sourceLength` 是字符串的长度。返回结果是一个字符串值，不是 `String` 对象。采用以下步骤：

1. 将 `this` 值作为参数调用 `ToString`。
2. 调用 `ToInteger(start)`。
3. 如果 `length` 是 `undefined`，就用 `+∞`；否则调用 `ToInteger(length)`。
4. 计算 `Result(1)` 的字符数。
5. 如果 `Result(2)` 是正数或零，就用 `Result(2)`；否则使用 `max(Result(4)+Result(2),0)`。
6. 计算 `min(max(Result(3),0), Result(4)−Result(5))`。
7. 如果 `Result(6) ≤ 0`，返回空字符串 `""`。
8. 返回一个由 `Result(1)` 中的 `Result(5)` 位置的字符开始的连续的 `Result(6)` 个字符组成的字符串。

`substr` 方法的 `length` 属性是 2。

`substr` 函数被刻意设计成通用的；它并不要求其 `this` 值为字符串对象。因此它可以作为方法转移到其他种类的对象中。

Date.prototype.getYear()

对于近乎所有用途，`getFullYear` 方法都是首选的，因为它避免了“2000 年问题”。

无参数方式调用 `getYear` 方法，采用以下步骤：

1. 令 `t` 为 `this` 时间值。
2. 如果 `t` 是 `NaN`，返回 `NaN`。
3. 返回 `YearFromTime(LocalTime(t)) − 1900`。

Date.prototype.setYear(year)

对于近乎所有用途，`setFullYear` 方法都是首选的，因为它避免了“2000 年问题”。

用一个参数 `year` 调用 `setYear` 方法，采用以下步骤：

1. 令 `t` 为 `LocalTime(this 时间值)` 的结果；但如果 `this` 时间值是 `NaN`，那么令 `t` 为 `+0`。
2. 调用 `ToNumber(year)`。
3. 如果 `Result(2)` 是 `NaN`，将 `this` 值的 `[[PrimitiveValue]]` 内部属性设为 `NaN`，并返回 `NaN`。

4. 如果 `Result(2)` 不是 NaN 并且 $0 \leq \text{ToInteger}(\text{Result}(2)) \leq 99$, 则 `Result(4)` 是 `ToInteger(Result(2)) + 1900`。否则, `Result(4)` 是 `Result(2)`。
5. 计算 `MakeDay(Result(4), MonthFromTime(t), DateFromTime(t))`。
6. 计算 `UTC(MakeDate(Result(5), TimeWithinDay(t)))`。
7. 将 `this` 值的 `[[PrimitiveValue]]` 内部属性设为 `TimeClip(Result(6))`。
8. 返回 `this` 值的 `[[PrimitiveValue]]` 内部属性值。

Date.prototype.toGMTString()

`toUTCString` 属性是首选的, `toGMTString` 属性是为了兼容较老的代码才提供的。建议在新的 ECMAScript 代码中使用 `toUTCString` 属性。

`Date.prototype.toGMTString` 的初始值是与 `Date.prototype.toUTCString` 的初始值相同的函数对象。

ECMAScript 的严格模式

严格模式下的限制说明

- 在严格模式下的代码中, "implements", "interface", "let", "package", "private", "protected", "public", "static", 和 "yield" 都被作为未来可能会使用到的保留字 (7.6.12)。
- 符合规范的实现中, 当处理严格模式下的代码时, 不应该像 B.1.1 中描述地那样将 `OctalIntegerLiteral` 扩展到 `NumericLiteral` (7.8.3) 的语法中。
- 符合规范的实现中, 当处理严格模式下的代码时, 不应该像 B.1.2 中描述地那样将 `OctalEscapeSequence` 扩展到 `EscapeSequence` 的语法中。
- 无法注册一个未定义的标识符或者其他无法解析的引用到全局对象下。当在严格模式下进行一个简单注册时, 它左部不能解析为一个无法解析的引用。如果是无法解析的, 那么就会抛出一个 `ReferenceError` 异常。左部也不能是一个数据属性的引用,
- `eval` 或者 `arguments` 不能出现在一个注册操作 (11.13) 或者一个 `Postfix` 表达式的左部, 也不能作为 `Prefix Increment` (11.4.4) 或者 `prefix decrement` 操作 (11.4.5) 上面的一元表达式操作。
- 严格模式下的 `arguments` 对象定义了不可配置的存取属性, 包括“caller”和“callee”, 如果访问这两个对象则会抛出一个类型错误。
- 严格模式下的 `Arguments` 对象不会动态共享它们的数组索引值, 这些索引值包含了函数绑定时对应格式的参数。
- 严格模式下的函数中, 如果一个参数对象绑定了作用域内的 `arguments` 标识符来获取参数对象, 那么这个参数对象是不可变的, 并在之后也不能进行注册操作。
- 在严格模式下, 如果代码包含了一个含有一个以上任意数据属性的定义, 那么这就是一个语法错误。

- 在严格模式下，如果 `eval` 或者 `argument` 出现在属性参数列表中，那么这就是一个语法错误

第 5 版中可能会对第 3 版产生兼容性影响的更正及澄清

全体：在第 3 版规范中像“就像用表达式 `new Array()` 一样”这样的短语的意思受到了误解。第 5 版规范中，对标准内置对象、属性的所有内部引用和内部调用相关文本描述，都做了澄清：应使用实际的内置对象，而不是对应命名属性的当前动态值。

11.8.2, 11.8.3, 11.8.5: ECMAScript 总体上是以从左到右的顺序解释执行，但是第 3 版规范中 `>` 和 `<=` 运算符的描述语言导致了局部从右到左的顺序。本规范已经更正了这些运算符，现在完全是从左到右的顺序解释执行。然而，这个对顺序的修改，如果在解释执行过程期间产生副作用，就有可能被观察到。

11.1.4: 第 5 版澄清了 `Array Initialiser` 结束位置的尾端逗号不计入数组长度。这不是对第 3 版语义的修改，但有些实现在之前可能对此有误解。

11.2.3: 第 5 版调换了算法步骤 2 和 3 的顺序。第 1 版一直到第 3 版规定的顺序是错误的，原来的顺序在解释执行 `Arguments` 时有副作用，可能影响到 `MemberExpression` 的解释执行结果。

12.14: 在第 3 版中，对于传给 `try` 语句的 `catch` 子句的异常参数的名称解析，用与 `new Object()` 一样的方式创建一个对象来作为解析这个名称的作用域。如果实际的异常对象是个函数并且在 `catch` 子句中调用了它，那么作用域对象将会作为 `this` 值传给这个调用。在函数体里可以给它的 `this` 值定义新属性，并且这些属性名将在函数返回之后在 `catch` 子句的作用域内变成可见的标识符绑定。在第 5 版中，如果把异常参数作为函数来调用，传入的 `this` 值是 `undefined`。

13: 在第 3 版中，有 `Identifier` 的 `FunctionExpression` 产生式的算法，用与 `new Object()` 一样的方式创建一个对象并加入到作用域链中，用来提供函数名查找的作用域。标识符解析规则（第 3 版里的 10.1.4）会作用在这样的对象上，如果需要，还会用对象的原型链来尝试解析标识符。这种方式使得 `Object.prototype` 的所有属性以标识符的形式在这个作用域里可见。实践中，大多数第 3 版的实现都没有实行这个语义。第 5 版更改了这里的语义，用一个声明式环境记录项来绑定了函数名。

14: 在第 3 版中，产生式 `SourceElements : SourceElements SourceElement` 的算法不像相同形式的 `Block`，对 `statement` 的结果值做正确的传递。这可导

致 `eval` 函数解释执行一个 `Program` 文本时产生错误的结果。实践中，大多数第 3 版的实现都做了正确的传递，而不关心第 5 版规定了什么。

15.10.6: `RegExp.prototype` 现在是一个 `RegExp` 对象，而不是 `Object` 的一个实例。用 `Object.prototype.toString` 可看到它的 `[[Class]]` 内部属性值现在是 `"RegExp"`，不是 `"Object"`。

第 5 版内容的增加与变化，介绍第 3 版不兼容问题

7.1: `Unicode` 格式控制字符在受到处理之前不再从 `ECMAScript` 源文本中剥离。在第五版中，如果这样一个字符在字符串字面量或者正则表达式字面量中出现，这个字符会被合并到字面量中，而在第三版里，这个字符不会被合并。

7.2: `Unicode` 字符 `<BOM>` 现在是作为空格使用，如果它出现在本该是一个标识符的位置的中间，则会产生一个语法错误，而在第三版里不会。

7.3: 换行符以前是作为转义字符处理，而现在允许换行符被包含在字符串字面量标记中。这在第三版中会产生一个语法错误。

7.8.5: 现在的正则表达式字面量在字面量解析执行的时候都会返回一个唯一的对象。这个改变可以被任意测试字面量值的对象 `ID` 或者一些敏感的副作用的程序检测到。

7.8.5: 第五版要求提前抛出任意可能的正则表达式结构错误，这些结构错误会在将正则表达式字面量转换成正则表达式对象的时候产生。在第五版之前的实现允许延迟抛出 `[TypeError]`，直到真正执行到这个对象。

7.8.5: 在第五版中，未转义的 `"/"` 字符可以作为 `CharacterClass` 存在于正则表达式字面量中。在第三版里，这样的字符是作为字面量的最后一个字符存在。

10.4.2: 在第五版中，间接调用 `eval` 函数会将全局对象作为 [执行代码](#) 的变量环境和 [词法环境](#)。在第三版中，`[eval]` 函数的间接调用者的变量和 [词法环境](#) 是作为 [执行代码](#) 的环境使用。

15.4.4: 在第五版中, 所有 [Array.prototype](#) 下的方法都是通用的。在第三版中, `toString` 和 `toLocaleString` 方法不是通用的, 如果被非 `Array` 实例调用时会抛出一个 `TypeError` 的异常。

10.6: 在第五版中, `argument` 对象与实际的参数符合, 它的数组索引属性是可枚举的。在第三版中, 这些属性是不可枚举的。

10.6: 在第五版中, 一个 `arguments` 对象的 [Class](#) 内置属性值是“Arguments”。在第三版中, 它是“Object”。当对 `argument` 对象调用`toString` 的时候

12.6.4: 当 `in` 表达式执行一个 `null` 或者 `undefined` 时, `for-in` 语句不再抛出 `TypeError`。取而代之的是将其作为不包含可枚举属性的对象执行。

15: 在第五版中, 下面的新属性都是在第三种中已存在的内建对象中定义, `Object.getPrototypeOf`, `Object.getOwnPropertyDescriptor`, `Object.getOwnPropertyNames`, `Object.create`, `Object.defineProperty`, `Object.defineProperties`, `Object.seal`, `Object.freeze`, `Object.preventExtensions`, `Object.isSealed`, `Object.isFrozen`, `Object.isExtensible`, `Object.keys`, `Function.prototype.bind`, `Array.prototype.indexOf`, `Array.prototype.lastIndexOf`, `Array.prototype.every`, `Array.prototype.some`, `Array.prototype.forEach`, `Array.prototype.map`, `Array.prototype.filter`, `Array.prototype.reduce`, `Array.prototype.reduceRight`, `String.prototype.trim`, `Date.now`, `Date.prototype.toISOString`, `Date.prototype.toJSON`。

15: 实现现在要求忽略内建方法中的额外参数, 除非明确指定。在第三版中, 并没有规定额外参数的处理方式, 实现中明确允许抛出一个 `TypeError` 错误。

15.1.1: 全局对象的值属性 `NaN`, `Infinity` 和 `Undefined` 改为只读属性。

15.1.2.1: 实现不再允许约束非直接调用 `eval` 的方式。另外间接调用 `eval` 会使用全局对象作为变量环境, 而不是使用调用者的变量环境作为变量环境。

15.1.2.2: `parseInt` 的规范不再允许实现将 0 开头的字符串作为 8 进制值。

15.3.4.3: 在第三版中, 如果传入 [Function.prototype.apply](#) 的第二个参数不是一个数组对象或者一个 `arguments` 对象, 就会抛出一个 `TypeError`。在第五版中, 参数也可以是任意类型的含有 `length` 属性的类数组对象。

15.3.4.3, 15.3.4.4: 在第三版中, 在 [Function.prototype.apply](#) 或

者 [Function.prototype.call](#) 中传入 `undefined` 或者 `null` 作为第一个参数会

导致 [全局对象](#) 被作为一个个参数传入，间接导致目标函数的 `[this]` 会指向全局变量环境。如果第一个参数是一个 [原始值](#)，在 [原始值](#) 上调用 [ToObject](#) 的结果会作为 `this` 的值。在第五版中，这些转换不会出现，目标函数的 `this` 会指向真实传入的参数。这个不同点一般情况下对已存在的遵循 **ECMAScript** 第三版的代码来说不太明显，因为相应转换会在目标函数生效之前执行。然而，基于不同的实现，如果使用 `apply` 或者 `call` 调用函数时，这个不同点就会很明显。另外，用这个方法调用一个标准的内建函数，并使用 `null` 或者 `undefined` 作为参数时，很可能会导致第五版标准下的实现与第三版标准下的实现不同。特别是第五版中代表性地规定了需要将实际调用的传入的 `this` 值作为对象的内建函数，在传入 `null` 或者 `undefined` 作为 `this` 值时，会抛出一个 `TypeError` 异常。

15.3.5.2: 在第五版中，函数实例的 `prototype` 属性是不可枚举的。在第三版中，是可以枚举的。

15.5.5.2: 在第五版中，一个字符串对象的 [primitiveValue](#) 的单个字符可以作为字符串对象的数组索引属性访问。这些属性是不可泄也不可配置的，并会影响任意名字相同的继承属性。在第三版中，这些属性不会存在，**ECMAScript** 代码可以通过这些名字动态添加和移除可写的属性并访问以这些名字继承的属性。

15.9.4.2: [Date.parse](#) 方法现在不要求第一个参数首先作为 **ISO** 格式字符串解析。使用这个格式但是基于特定行为实现（包括未来的一些行为）或许会表现的不太一样。

15.10.2.12: 在第五版中，`\s` 现在可以匹配 `<BOM>` 了

15.10.4.1: 在第三版中，由 **RegExp** 构造器创建的对对象的 `source` 字符串的精确形式由实现定义。在第五版中，字符串必须符合确定的指定条件，因此会和第三版标准的实现的结果不一样。

15.10.6.4: 在第三版中，[RegExp.prototype.toString](#) 的规则不需要由 **RegExp** 对象的 `source` 属性决定。在第五版中，结果必须由 `source` 属性经由一个指定的规则，因此会和第三版实现的结果不一样。

15.11.2.1, 15.11.4.3: 在第五版中，如果一个错误对象的 `message` 属性原始值没有通过 **Error** 构造器指定，那么这个原始值就是一个空的字符串。在第三版中，这个原始值由实现决定。

15.11.4.4: 在第三版中，`Error.prototype.toString` 的结果是由实现定义的。在第五版中，有完整的规范指定，因此可能会和第三版的实现不同。

15.12: 在第五版中，JSON 是在全局环境中定义的。第三版中，测试这个名词的存在会发现它是 `undefined`，除非这个程序或者实现定义了这个名词。

5.1 版中技术上的重大更正和阐明

7.8.4: CV 定义追加了 `DoubleStringCharacter :: LineContinuation` 与 `SingleStringCharacter :: LineContinuation`。

10.2.1.1.3: 参数 `S` 是不能被忽略的。它控制着试图设置一个不可改变的绑定时是否抛出异常。

10.2.1.2.2: 在算法的第 5 步，真被传递后最后一个参数为 `[[DefineOwnProperty]]`。

10.5: 当重定义全局函数使，原算法步骤 5.E 调整为现在的 5.F，并加入一个新的步骤 5.E 用来还原与第三版的兼容性。

11.5.3: 在最后符号项，指定使用 `IEEE754` 舍入到最接近模式。

12.6.3: 在步骤 3.a.ii 的两种算法中修复缺失的 `ToBoolean`。

12.6.4: 在最后两段的额外最后一句中，阐明某些属性枚举的规定。

12.7, 12.8, 12.9: BNF 的修改为阐明 `continue` 或 `break` 语句没有一个 `Identifier` 或一个 `return` 语句没有一个 `Expression` 时，在分号之前可以有一个 `LineTerminator`。

12.14: 算法 1 的步骤 3 算法 3 的步骤 2.a 中，纠正这样的值域 `B` 是作为参数传递而不是 `B` 本身。

15.1.2.2: 在算法的步骤 2 中阐明 `S` 可能是空字符串。

15.1.2.3: 在算法的步骤 2 中阐明 `trimmedString` 可以是空字符串。

15.1.3: 添加注释阐明 ECMAScript 中的 URI 语法基于 [RFC 2396](#) 和较新的 [RFC 3986](#)。

15.2.3.7: 在算法步骤 5 和 6 中更正使用变量 `P`。

15.2.4.2: 第五版处理 `undefined` 和 `null` 值导致现有代码失败。规范修改为保持这样的代码的兼容性。在算法中加入新的步骤 1 和 2。

15.3.4.3: 步骤 5 和 7 版 5 算法已被删除，因为它们规定要求 `argArray` 参数与泛数组状对象的其它用法不一致。

15.4.4.12: 在步骤 9.A，用 `actualStart` 替换不正确 `relativeStart` 引用。

15.4.4.15: 阐明 `fromIndex` 的默认值是数组的长度减去 1。

15.4.4.18: 在算法的第 9 步，`undefined` 是现在指定的返回值。

15.4.4.22: 在步骤 9.c.ii 中，第一个参数的 `[[Call]]` 内部方法已经改变为 `undefined`，保持与 `Array.prototype.reduce` 定义的一致性。

15.4.5.1: 在算法步骤 3.l.ii 和 3.l.iii 中，变量的名字是相反的，导致一个不正确的相反测试。

15.5.4.9: 规范要求每有关规范等效字符串删除，算法从每一个段落都承接，因为它在注 2 中被列为建议的。

15.5.4.14: 在 `split` 算法步骤 11.A 和 13.a，`SplitMatch` 参数的位置顺序已修正为匹配 `SplitMatch` 的实际参数特征。在步 13.a.iii.7.d，`lengthA` 取代 `A.length`。

15.5.5.2: 在首段中，删除的单个字符属性访问“array index”语义的含义。改进算法步骤 3 和 5，这样它们不执行“array index”的要求。

15.9.1.15: 为缺失字段指定了合法值范围。淘汰“time-only”格式。所有可选字段指定默认值。

15.10.2.2: 算法步骤编号为第二步所产生的内部闭包被错误的编号，它们是额外的算法步骤。

15.10.2.6: 在步骤 3 中的列表中抽象运算符 `IsWordChar` 的第一个字符是“a”而不是“A”。

15.10.2.8: 在闭包算法返回抽象运算符 `CharacterSetMatcher` 中，为了避免与一个闭包的形参名称冲突，步骤 3 中定义的变量作为参数传递在第 4 步更名为 `ch`。

15.10.6.2: 步骤 9.e 被删除，因为它执行了 `l` 的额外增量。

15.11.1.1: 当 `message` 参数是 `undefined` 时，撤销 `message` 自身属性设置为空字符串的要求。

15.11.1.2: 当 `message` 参数是 `undefined` 时，撤销 `message` 自身属性设置为空字符串的要求。

15.11.4.4: 步骤 6-10 修改 / 添加正确处理缺少或空的 `message` 属性值。

15.11.1.2: 移除了当 `message` 参数为 `undefined` 时将 `message` 自身属性设为空字符串的要求。

15.12.3: 在 JA 的内部操作的第一步 10.b.iii, 串联的最后一个元素是 “`]`”。

B.2.1: 追加注释, 说明编码是基于 [RFC 1738](#) 而不是新的 [RFC 3986](#)。

附录 C: 增加了 `FutureReservedWords` 在标准模式下的相应内容到 7.6.12 节。

参考书目

1. ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic. Institute of Electrical and Electronic Engineers, New York (1985)
2. The Unicode Consortium. The Unicode Standard, Version 3.0, defined by: The Unicode Standard, Version 3.0 (Boston, MA, Addison-Wesley, 2000. [ISBN 0-201-61635-5](#))
3. Unicode Inc. (1998), Unicode Technical Report #15: Unicode Normalization Forms
4. ISO 8601:2004(E) Data elements and interchange formats – Information interchange -- Representation of dates and times
5. [RFC 1738](#) “Uniform Resource Locators (URL)”, available at [<http://tools.ietf.org/html/rfc1738>](http://tools.ietf.org/html/rfc1738)
6. [RFC 2396](#) “Uniform Resource Identifiers (URI): Generic Syntax”, available at [<http://tools.ietf.org/html/rfc2396>](http://tools.ietf.org/html/rfc2396)
7. [RFC 3629](#) “UTF-8, a transformation format of ISO 10646”, available at [<http://tools.ietf.org/html/rfc3629>](http://tools.ietf.org/html/rfc3629)
8. [RFC 4627](#) “The application/json Media Type for JavaScript Object Notation (JSON)”, available at [<http://tools.ietf.org/html/rfc4627>](http://tools.ietf.org/html/rfc4627)