**目录**

第1章 作用范围……3

第2章 遵从性……3

第3章 引用标准……4

第4章 概述……4

4.1节 Web脚本……5

4.2节 语言概述……5

4.3节 术语和定义……7

第5章 记号表示的约定……11

5.1节 语法和词法文法……11

5.2节 算法约定……16

第6章 源文本……17

第7章 词法约定……18

7.1节 Unicode格式控制字符……18

7.2节 空白……19

7.3节 行终止符……20

7.4节 注释……21

7.5节 词法单元……22

7.6节 标识符名字和标识符……22

7.7节 标点符号……24

7.8节 字面值……25

7.9节 自动分号插入……33

第8章 类型……35

8.1节 Undefined类型……35

8.2节 Null类型……35

8.3节 Boolean类型……35

8.4节 String类型……35

8.5节 Number类型……36

8.6节 Object类型……37

8.7节 Reference规范类型……42

8.8节 List规范类型……44

8.9节 Completion规范类型……44

8.10节 Property Descriptor和Property Identifier规范类型……44

8.11节 Lexical Environment和Environment Record规范类型……47

8.12节 用于Object内部方法的算法……47

第9章 类型转换和测试……52

9.1节 ToPrimitive……52

9.2节 ToBoolean……53

9.3节 ToNumber……53

9.4节 ToInteger……57

9.5节 ToInt32：（有符号32位整数）……57

9.6节 ToUint32：（无符号32位整数）……57

9.7节 ToUint16：（无符号16位整数）……58

9.8节 ToString……58

9.9节 ToObject……59

9.10节 CheckObjectCoercible……60

9.11节 IsCallable……60

9.12节 SameValue算法……61

第10章 可执行代码和执行上下文……61

10.1节 可执行代码类型……61

10.2节 词法环境……62

10.3节 执行上下文……68

10.4节 建立一个执行上下文……69

10.5节 声明绑定实例化……70

10.6节 实参对象……72

第11章 表达式……75

11.1节 主表达式……75

11.2节 左手边表达式……80

11.3节 后缀表达式……83

11.4节 一元运算符……84

11.5节 乘除运算符……87

11.6节 加减运算符……89

11.7节 位移运算符……90

11.8节 关系运算符……91

11.9节 对等运算符……95

11.10节 二元位运算符……98

11.11节 二元逻辑运算符……99

11.12节 条件运算符（**? :**）……100

11.13节 赋值运算符……100

11.14节 逗号运算符（**,**）……102

第12章 语句……102

12.1节 块……103

12.2节 变量语句……104

12.3节 空语句……106

12.4节 表达式语句……106

12.5节 **if**语句……106

12.6节 迭代语句……107

12.7节 **continue**语句……110

12.8节 **break**语句……110

12.9节 **return**语句……111

12.10节 **with**语句……111

12.11节 **switch**语句……112

12.12节 带标号的语句……114

12.13节 **throw**语句……115

12.14节 **try**语句……115

12.15节 **debugger**语句……116

第13章 函数定义……117

13.1节 严格模式限制……118

13.2节 创建Function对象……118

第14章 程序……120

14.1节 指令序言和Use Strict指令……121

【SP11】

序言

这份Ecma标准是建立于多个始发技术之上，最著名的就是JavaScript（Netscape）和JScript（Microsoft）。语言是由Netscape公司的Brendan Eich发明，并且在该公司的Navigator 2.0浏览器中首次出现。它已经出现在Netscape公司所有后继的浏览器中，以及Microsoft公司从IE3.0开始的所有浏览器中。

这份标准的制订始于1996年11月，第一版被Ecma General Assembly于1997年6月采纳。

那份标准被提交给ISO/IEC JTC1用于在fast-track过程下采纳，并于1998年4月作为国际标准ISO/IEC 16262获得批准。1998年6月Ecma General Assembly批准了ECMA-262第二版，为了完全对齐ISO/IEC 16262。第一版与第二版本质上只在编辑方面【用词、排版等】有所变化。

标准的第三版引入了功能强大的正则表达式，更好的字符串处理，新的控制语句，try/catch异常处理，更紧凑的错误定义，格式化数字输出，以及在展望即将来临的国际化能力和未来语言成长方面的少量改变。第三版标准被Ecma General Assembly于1999年12月采纳，并于2002年6月作为ISO/IEC 16262:2002颁布。

从第三版的颁布开始，ECMAScript和万维网一起获得了大量的采纳，在万维网里它已经成为一门实质上得到所有Web浏览器支持的编程语言。为了开发ECMAScript第四版，大量的工作已经进行。尽管那些工作未能完成并且也没有作为ECMAScript第四版的标准颁布，但这仍然预示着语言的继续发展。ECMAScript第五版（作为ECMA-262 5th版颁布）把语言规范在浏览器实现里已经成为共同的事实上的理解编入标准，并加入了对一些从第三版颁布起就已经出现的新功能的支持。这些功能包括存取器属性，运用反射创建和洞察对象，程序控制属性特性，附加的数组操纵函数，支持JSON对象编码格式，和一个提供了强化错误检查及程序安全性的严格模式。

这份ECMAScript标准5.1版完全对齐于国际标准ISO/IEC 16262:2011第三版。

ECMAScript是一门鲜活的语言，而且语言的发展还没有完成。重大的技术增强将会继续伴随着这份规范的未来版本。

这份Ecma标准已经被Ecma General Assembly于2011年6月采纳。

【SP13】

ECMAScript语言规范

第1章 作用范围

这份标准定义了ECMAScript脚本语言。

第2章 遵从性

一个ECMAScript的遵从实现必须提供并支持这份规范里描述的所有类型，值，对象，属性，函数，和程序语法及语义。

一个这份标准的遵从实现应按照Unicode标准，版本3.0或后续版本，和ISO/IEC 10646-1，使用UCS-2或UTF-16作为采用的编码形式，实现等级3，来解释字符。如果采用的ISO/IEC 10646-1子集没有专门指定，假定是BMP子集，集合300。如果采用的编码形式没有专门指定，假定是UTF-16编码形式。

允许一个ECMAScript的遵从实现提供比在这份规范里描述的更多的类型，值，对象，属性，和函数。特别的，允许一个ECMAScript的遵从实现为在这份规范里描述的对象提供在这份规范里没有描述的属性，及那些属性的值。

允许一个ECMAScript的遵从实现支持在这份规范里没有描述的程序和正则表达式语法。特别的，允许一个ECMAScript的遵从实现支持利用了“将来保留字”（列于这份规范7.6.1.2节）的程序语法。

第3章 引用标准

下面的参考文档对于这份文档的应用是不可或缺的。对于标明日期的参考，只有引用的版本适用；对于未标明日期的参考，最新版本（包括任何修正）适用。

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

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

第4章 概述

这一章包含了一个对ECMAScript语言非规范的概述。

ECMAScript是一门面向对象编程语言，用于在一个宿主环境里执行计算和操纵参与计算的对象。这里定义的ECMAScript并不表示在计算上自给自足，的确，这份规范里没有条款用于外部数据的输入或计算结果的输出；反之，规范期望一个ECMAScript程序的计算环境不仅会提供这份规范里描述的对象和其它设施，还会提供某些特定于环境的宿主对象，除了指出它们可以提供某些能从ECMAScript程序访问的属性和调用的函数外，它们的描述和行为超出了这份规范的作用范围。

【SP14】

脚本语言是一种编程语言，被用于操纵，定制，和自动化一个现有系统。在这种系统里，有用的功能已经可通过用户界面使用，而脚本语言是一种机制，用于把那些功能暴露给程序控制。以这种方式，现有系统被称为提供了对象和设施的宿主环境，这个环境使得脚本语言的功能完整。一门脚本语言的目标是即可被专业程序员使用，也可被非专业程序员使用。

ECMAScript最初被设计为成为一门Web脚本语言，提供一种机制为浏览器里的Web页面带来活力，并作为基于Web的客户端-服务器体系结构的一部分执行服务器计算。ECMAScript能够为各种宿主环境提供核心脚本功能，因此在这份文档里规定了核心脚本语言，无关于任何特殊的宿主环境。

ECMAScript的一些设施类似于那些用在其它编程语言里的设施；特别是如下面文献所描述的Java，Self和Scheme：

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.

4.1节 Web脚本

一个Web浏览器为客户端计算提供了一个ECMAScript宿主环境，包括一些对象用于表示诸如窗口，菜单，弹窗，对话框，文本框，锚，框架，历史，Cookies，和输入输出。进一步，宿主环境还提供了一种方法可将脚本代码连接到事件，例如焦点改变，页面和图像加载，卸载，错误和中止，选取，表单提交，和鼠标动作。脚本代码出现在HTML里，而显示的页面是由用户界面元素，固定的及计算得到的文本，和图像联合而成的。脚本代码响应用户交互，因此不需要主程序。

Web服务器为服务端计算提供了一个不同的宿主环境，包括一些对象用于表示诸如请求，客户端，和文件；以及一些机制用于锁定和共享数据。通过一起使用浏览器端和服务器端脚本，就有可能在客户端和服务器之间分配计算，同时为一个基于Web的应用程序提供一个定制的用户界面。

每个支持ECMAScript的Web浏览器和服务器都提供了它自己的宿主环境，使得ECMAScript执行环境变得完整。

4.2节 语言概述

下面是对ECMAScript非正式的概述——并没有描述语言的所有部分。这个概述不是标准的一部分。

ECMAScript是基于对象的：基本语言和宿主设施是以对象的形式提供的，而ECMAScript程序就是一群相互通信的对象。一个ECMAScript对象是一个属性的集合，每个属性带有零个或多个决定了属性可以被如何使用的特性——例如，当一个属性的Writable特性被设为false时，任何通过执行ECMAScript代码来改变属性值的尝试都将失败。属性是持有其他对象，基元值，或函数的容器。一个基元值是如下内建类型之一的成员：Undefined，Null，Boolean，Number，和String；一个对象是剩余内建类型Object的成员；一个函数是一个可调用的对象。通过属性关联到一个对象上的函数叫做方法。

ECMAScript定义了一个内建对象的集合，完善了ECMAScript实体的定义。这些内建对象包括Global对象，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，

【SP15】

JSON对象，以及一些错误对象如Error，EvalError，RangeError，ReferenceError，SyntaxError，TypeError，和URIError。

ECMAScript同样定义了一组内建运算符。ECMAScript运算符包含各种一元运算符，乘除运算符，加减运算符，移位运算符，关系运算符，对等运算符，二元位运算符，二元逻辑运算符，赋值运算符，和逗号运算符。

ECMAScript语法有意模仿Java语法。ECMAScript语法是很宽松的，为了使它能够作为一种易于使用的脚本语言。例如，定义变量时不需要声明类型，定义属性时也不需要；函数的定义位置可以出现在调用位置后，在调用位置前不需要给出声明。

4.2.1段 对象

ECMAScript没有像C++，Smalltalk，或Java中的类那样的类。相反，对象可以用多种方式创建，包括通过字面值表示法，或通过构造函数（它创建对象然后执行为全部或部分属性赋予初始值的代码以初始化对象）。每个构造函数是一个具有名为“prototype”属性的函数，该属性可被用于实现基于原型的继承和共享属性。通过在**new**表达式中使用构造函数可以创建对象；例如，**new** Date(2009, 11)创建了一个新的Date对象。如果调用一个构造函数而没有使用**new**，其结果依赖于构造函数。例如，Date()产生一个当前日期和时间的字符串表示而不是一个对象。

由构造函数创建的每个对象具有一个对其构造函数的“prototype”属性值的隐式引用（叫做对象的原型）。进一步，一个原型也可以具有一个对其原型的非空隐式引用，如此等等，这被称为原型链。当引用一个对象的某一属性时，引用指向了——原型链里第一个包含同名属性的对象里的——那一属性。换句话说，首先查看直接提到的那个对象是否有那一属性，是的话，那就是引用所指；否则，下一步查看那个对象的原型，如此等等。

在一个基于类的面向对象语言里，一般来说，状态由实例携带，方法由类携带，而继承的只是结构和行为。在ECMAScript里，状态和方法由对象携带，而结构，行为，和状态都被继承。

【SP16】

对于所有——未直接包含某个特定属性、而是由它们共同的原型包含的——对象来说，它们共享了那个属性和属性值，图1示范了这一点：

↑

┆ ┌─────┐ ↑ …………→ 隐式原型链

└……│CF │ ┌─────┐ ┆ ────→ 显示原型属性

│prototype ┼──→│CFp │……┘

│P1 │ │CFP1 │

│P2 │ │ │

└─────┘ └─────┘

↑ ↑ ↑ ↑ ↑

┆ ┆ ┆ ┆ ┆

┌…………………┘ ┆ ┆ ┆ └…………………┐

┆ ┌………┘ ┆ └………┐ ┆

┆ ┆ ┆ ┆ ┆

┌─┴─┐ ┌─┴─┐ ┌─┴─┐ ┌─┴─┐ ┌─┴─┐

│cf1 │ │cf2 │ │cf3 │ │cf4 │ │cf5 │

│q1 │ │q1 │ │q1 │ │q1 │ │q1 │

│q2 │ │q2 │ │q2 │ │q2 │ │q2 │

└───┘ └───┘ └───┘ └───┘ └───┘

图1 —— 对象∕原型关系

CF是一个构造函数（也是一个对象）。有五个对象已经使用**new**表达式创建出来：cf1，cf2，cf3，cf4，和cf5。每个对象都包含名为q1和q2的属性。虚线表示隐式原型关系；因此，例如，cf3的原型是CFp。构造函数CF自身有两个属性，叫做P1和P2，它们俩对于CFp，cf1，cf2，cf3，cf4，和cf5都是不可见的。在CFp里名为CFP1的属性被cf1，cf2，cf3，cf4，和cf5共享（但不被CF），在CFp的隐式原型链里的——任何名字不是q1，q2，或CFP1的——属性都会被那五个对象共享。注意在CF和CFp之间没有隐式原型连接。

不像基于类的对象语言，通过向属性赋值就可以把属性动态地添加到对象上。也就是说，构造函数不必为——被构造对象的全部或任何属性——命名或赋值。在上面的图里，我们可以通过向CFp里的属性赋值而为cf1，cf2，cf3，cf4，和cf5添加一个新的共享属性。

4.2.2段 ECMAScript的严格变体

ECMAScript语言认识到存在这样一种可能性，即语言的一些用户可能想限制对语言里某些功能的使用。他们或许是出于安全考虑，避免使用他们认为易于出错的功能，以获得强化的错误检查，或许是因为其他原因。为了支持这种可能性，ECMAScript定义了一个语言的严格变体。语言的严格变体排除了一些——普通ECMAScript语言具有的——特殊的语法和语义功能，并修改了一些功能的细节语义。有些——没有被语言的非严格形式规定为错误的——情况，严格变体却将它们规定为错误情况，此时必须通过抛出错误异常来报告。

ECMAScript的严格变体通常被称为语言的严格模式。严格模式的选择，及ECMAScript严格模式语法和语义的使用，是在各个ECMAScript代码单元的级别上进行的。因为严格模式是在语法代码单元的级别上选择的，所以严格模式只会施加——在这个代码单元里具有局部效果的——限制。严格模式不会限制或修改任何——跨多个代码单元必须一致操作的——ECMAScript语义方面。一个完整的ECMAScript程序可以由严格模式代码单元和非严格模式代码单元一起组成。在这种情况下，仅当实际执行定义在严格模式代码单元里的代码时才会应用严格模式。

为了遵从这份规范，一个ECMAScript实现必须同时实现完全无限制的ECMAScript语言和这份规范里定义的ECMAScript语言严格模式变体。此外，一个实现必须支持把无限制的代码单元和严格模式代码单元合并成一个复合的程序。

4.3节 术语和定义

下面的术语和定义将应用于这份文档。

4.3.1段 类型

数据值的集合，如这份规范第8章所定义。

4.3.2段 基元值

以下类型之一的成员：Undefined，Null，Boolean，Number，或String，如第8章所定义。

注意：一个基元值是一个在语言实现的最底层被直接表示的数据。

4.3.3段 对象

Object类型的成员。

注意：一个对象是一个属性的集合，并有一个原型对象。原型值可以是null。

【SP17】

4.3.4段 构造函数

创建并初始化对象的函数对象。

注意：构造函数的“prototype”属性的值是一个原型对象，被用于实现继承和共享属性。

4.3.5段 原型

为其它对象提供了共享属性的对象。

注意：当一个构造函数创建了一个对象时，那个对象隐式引用了构造函数的“prototype”属性，用于解析属性引用。构造函数的“prototype”属性可通过程序表达式constructor.prototype引用，而且添加到一个对象的原型上的属性，通过继承，被全部共享了那个原型的对象所共享。作为选择，一个新对象也可以通过使用Object.create内建函数并显示指定原型来创建。

4.3.6段 原生对象

在一个ECMAScript实现里，语义完全由这份规范定义而不是由宿主环境定义的对象。【“原生”可以理解为“这门语言本身的，和宿主无关的”，如String】

注意：标准原生对象在这份规范里定义。一些原生对象是内建的；其它的可以在ECMAScript程序执行过程里被构造。

4.3.7段 内建对象

由一个ECMAScript实现提供、无关于宿主环境、且在ECMAScript程序执行开始时就已存在的对象。【如Array】

注意：标准内建对象在这份规范里定义，一个ECMAScript实现还可以规定并定义其它的。每个内建对象都是一个原生对象。一个内建构造函数既是一个内建对象，也是一个构造函数。

4.3.8段 宿主对象

为使ECMAScript执行环境完整而由宿主环境提供的对象。【如window】

注意：任何不是原生对象的对象，是一个宿主对象。

【关系图：

┌───┐

│ 对象 │

└─┬─┘

│

┌─────────┴─────────┐

│ │

┌─┴─┐ ┌─┴─┐

│ 原生 │ │ 宿主 │

└─┬─┘ └───┘

│

┌─────┴─────┐

│ │

┌──┴──┐ ┌──┴──┐

│ 内建 │ │ 非内建 │

│（程序开始│ │（程序执行│

│ 时就有）│ │ 时创建）│

└─────┘ └─────┘

】

4.3.9段 undefined值

基元值，当一个变量还没有被赋值时使用。

4.3.10段 Undefined类型

类型，它唯一的值是undefined。

4.3.11段 null值

基元值，表示任何对象值的有意缺失。

4.3.12段 Null类型

类型，它唯一的值是null。

4.3.13段 Boolean值

Boolean类型的成员。

注意：只有两个Boolean值，true和false。

【SP18】

4.3.14段 Boolean类型

类型，由基元值true和false构成。

4.3.15段 Boolean对象

Object类型的成员，是由标准内建Boolean构造函数创建的实例。

注意：一个Boolean对象是在**new**表达式中使用Boolean构造函数创建的，提供一个Boolean值作为实参，产生的对象具有一个内部属性，其值就是那个Boolean值。一个Boolean对象可被强制转换成一个Boolean值。

4.3.16段 String值

基元值，零个或多个16位无符号整数的有限有序序列。

注意：一个String值是String类型的成员。序列中每个整数值通常表示UTF-16文本的一个16位单元。然而，ECMAScript没有在数值上施加任何限制或要求，除了它们必须是16位无符号整数。

4.3.17段 String类型

所有可能的String值的集合。

4.3.18段 String对象

Object类型的成员，是由标准内建String构造函数创建的实例。

注意：一个String对象是在**new**表达式中使用String构造函数创建的，提供一个String值作为实参，产生的对象具有一个内部属性，其值就是那个String值。通过把String构造函数作为普通函数调用，一个String对象可被强制转换成一个String值（15.5.1段）。

4.3.19段 Number值

基元值，对应到一个双精度64位二进制格式IEEE 754的值。

注意：一个Number值是Number类型的成员，也是一个数字的直接表示。

4.3.20段 Number类型

所有可能的Number值的集合，包括特殊的“不是一个数字”（NaN）值，正无穷，和负无穷。

4.3.21段 Number对象

Object类型的成员，是由标准内建Number构造函数创建的实例。

注意：一个Number对象是在**new**表达式中使用Number构造函数创建的，提供一个Number值作为实参，产生的对象具有一个内部属性，其值就是那个Number值。通过把Number构造函数作为普通函数调用，一个Number对象可被强制转换成一个Number值（15.7.1段）。

4.3.22段 Infinity

数字值，表示正无穷的Number值。

4.3.23段 NaN

数字值，表示IEEE 754“不是一个数字”的Number值。

【SP19】

4.3.24段 function

Object类型的成员，是由标准内建Function构造函数创建的实例，并可以作为一个子例程而被调用。

注意：除了它的具名属性，一个function还包含了当调用时决定其行为的可执行代码和状态。一个function的代码可以用ECMAScript编写，也可以不用。

4.3.25段 内建function

内建对象，且是一个function。

注意：内建function的例子包括parseInt和Math.exp。一个实现可以提供没有在这份规范里描述、依赖于实现的内建function。

4.3.26段 属性【property】

一个名字和一个值之间的关联，且是一个对象的一部分。

注意：依赖于属性的形态，属性值或者可被直接表示成一个数据值（一个基元值，一个对象，或者一个函数对象），或者可被间接表示成一对存取器函数。

4.3.27段 方法

一个函数，且是一个属性的值。

注意：当一个函数作为一个对象的方法被调用时，这个对象被传入那个函数，作为它的this值。

4.3.28段 内建方法

一个方法，且是一个内建函数。

注意：标准内建方法在这份规范里定义，而一个ECMAScript实现还可以规定并提供其它附加的内建方法。

4.3.29段 特性【attribute】

一个内部值，定义了一个属性的一些特征。

4.3.30段 自有属性【own property】

一个属性，且被其对象直接包含。【不在原型链里】

4.3.31段 继承属性【inherited property】

一个属性，却不是自有属性，而是其对象的原型的（自有或继承）属性。【在原型链里】

第5章 记号表示的约定

5.1节 语法和词法文法

5.1.1段 上下文无关文法

一个上下文无关文法是由许多产生式组成的。每个产生式有一个称为非终结符的抽象符号作为它的左部，还有一个由零个或多个非终结符和终结符组成的序列作为它的右部。对于每个文法，终结符从一个指定的字母表里抽取。

【SP20】

从一个仅由单一独特非终结符构成的句子开始（该非终结符称为目标符号），一个给定的上下文无关文法规定了一个语言，也就是由终结符组成的所有可能序列的集合（可能是无限集合），这个集合可以如下获得：对于任一个由零个或多个非终结符和终结符组成的序列，不断地把序列中的非终结符替换成一个以该非终结符作为左部的产生式的右部。

5.1.2段 词法和正则表达式文法

ECMAScript的词法文法在第7章给出。这个文法的终结符是由——遵从第6章定义的SourceCharacter规则的——字符（Unicode编码单位）构成的。它定义了一组产生式，从目标符号InputElementDiv或InputElementRegExp开始，描述了这些字符组成的序列是如何被翻译成一个输入元素序列的。

空白和注释以外的输入元素为ECMAScript语法文法构成了终结符，并被称为ECMAScript词法单元【token】。这些词法单元有保留字，标识符，字面值，和ECMAScript语言的标点符号。此外，行终止符，尽管不被认为是词法单元，也成为输入元素流的一部分，并且指引着自动分号插入过程（7.9节）。简单的空白和单行注释被丢弃，不会出现在语法文法的输入元素流里。多行注释（由“/\*...\*/”形成的注释，无论是否跨越了多行）【块注释】如果没有包含行终止符则也被类似的丢弃；但如果包含一个或多个行终止符，那么整个注释就被替换成一个行终止符，并成为语法文法输入元素流的一部分。

ECMAScript的正则表达式文法在15.10节给出。这个文法的终结符也是由遵从SourceCharacter规则的字符构成的。它定义了一组产生式，从目标符号Pattern开始，描述了这些字符组成的序列是如何被翻译成正则表达式模式的。

词法和正则表达式文法的产生式根据使用两个冒号“::”作为分割标点来识别。词法和正则表达式文法共享了一些产生式。

5.1.3段 数字字符串文法

这个文法被用来把字符串翻译成数字值，它类似于词法文法处理数字字面值的那一部分，它也把SourceCharacter作为它的终结符。这个文法在9.3.1段给出。

5.1.4段 语法文法

ECMAScript语法文法在11、12、13和14章给出。这个文法使用由词法文法定义的ECMAScript词法单元作为它的终结符（5.1.2段）。它定义了一组产生式，从目标符号Program开始，描述了词法单元序列如何才能构成语法正确的ECMAScript程序。

当一个字符流将作为ECMAScript程序解析时，通过反复应用词法文法，这个字符流首先被转换成一个输入元素流；之后再应用一次语法文法来解析这个输入元素流。如果输入元素流里的词法单元不能作为目标非终结符Program的单一实例而被解析，且没有剩余词法单元，那么程序在语法上就有错误。

语法文法的产生式根据只使用一个冒号“:”作为分割标点来识别。

呈现在11、12、13和14章的语法文法实际上不是一份完整的、关于哪些词法单元序列将作为正确的ECMAScript程序而被接受的报告。某些附加的词法单元序列也被接受，就是那些将在文法里描述的、只在某些位置（如行终止符之前）添加了分号的序列。进一步，文法描述的某些词法单元序列不被认为是可接受的，如果行终止符出现在某些“不协调的”位置。

【SP21】

5.1.5段 JSON文法

JSON文法被用于把一组由字符串描述的ECMAScript对象翻译成实际的对象。这个文法在15.12.1段给出。

JSON文法是由JSON词法文法和JSON语法文法构成的。JSON词法文法被用于把字符序列翻译成词法单元，且和ECMAScript词法文法的一些地方类似。JSON语法文法描述了来自JSON词法文法的词法单元序列如何才能构成语法正确的JSON对象描述。

JSON词法文法的产生式根据使用两个冒号“::”作为分割标点来识别。JSON词法文法使用了一些来自ECMAScript词法文法的产生式。JSON语法文法和ECMAScript语法文法的一些地方类似。JSON语法文法的产生式根据只使用一个冒号“:”作为分割标点来识别。

5.1.6段 语法记号表示

词法，正则表达式，和数字字符串文法的终结符，以及其它文法的一些终结符，用定宽字体显示，既在文法的产生式里，也在这份规范的文本里每当直接引用这样一个终结符时。这些终结符和写在程序代码里的完全一致。所有以这种方式指定的终结符字符将作为来自ASCII范围的相应Unicode字符理解，而不是任何具有类似外观但来自其它Unicode范围的字符。

非终结符以斜体显示。通过在非终结符的名字后面跟上一个或多个冒号来引入非终结符的定义（冒号数量表示这个产生式属于哪个文法），非终结符的一个或多个可选右部跟在下面连续的几行里。例如，语法定义：

*WhileStatement* :

**while (** *Expression* **)** *Statement*

说得是非终结符*WhileStatement*表示词法单元**while**，后跟一个词法单元左圆括号，后跟一个*Expression*，后跟一个词法单元右圆括号，后跟一个*Statement*。出现的*Expression*和*Statement*本身也是非终结符。作为另一个例子，语法定义：

*ArgumentList* :

*AssignmentExpression*

*ArgumentList* , *AssignmentExpression*

说得是一个*ArgumentList*可以表示单独一个*AssignmentExpression*，或者一个*ArgumentList*，后跟一个逗号，后跟一个*AssignmentExpression*。对*ArgumentList*的定义是递归的，也就是说，它是从它自身的角度定义的。结果是一个*ArgumentList*可以包含任意数量的实参，以逗号分隔，而每个实参是一个*AssignmentExpression*。非终结符的这种递归定义很常见。

下标形式的后缀“opt”可以出现在一个终结符或非终结符之后，表示这是一个可选的符号。用来表示可选符号的另一办法是指定两个右部，一个省略了可选元素而另一个包含了它，这意味着：

*VariableDeclaration* :

*Identifier Initializer* opt

是以下写法的简写：

*VariableDeclaration* :

*Identifier*

*Identifier Initializer*

【SP22】

而

*IterationStatement* :

**for (** *ExpressionNoIn* opt ; *Expression* opt ; Expression opt **)** *Statement*

是以下写法的简写：

*IterationStatement* :

**for (** ; *Expression* opt ; Expression opt **)** *Statement*

**for (** *ExpressionNoIn* opt ; *Expression* opt ; Expression opt **)** *Statement*

而它又是以下写法的简写：

*IterationStatement* :

**for (** ; ; Expression opt **)** *Statement*

**for (** ; *Expression* ; Expression opt **)** *Statement*

**for (** *ExpressionNoIn* ; ; Expression opt **)** *Statement*

**for (** *ExpressionNoIn* ; *Expression* ; Expression opt **)** *Statement*

而它又是以下写法的简写：

*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*

所以非终结符*IterationStatement*实际上有个8个可选的右部。

当词语“**其中一个**”跟在一个文法定义的冒号后时，表示在下面一行或几行里的每个终结符都是一个可供选择的定义。例如，ECMScript词法文法包含产生式：

*NonZeroDigit* :: **其中一个**

1. **2 3 4 5 6 7 8 9**

这只不过是以下写法的简写：

*NonZeroDigit* ::

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

如果短语“[empty]”出现在一个产生式的右部，表示产生式的右部既没有包含终结符，也没有包含非终结符。

如果短语“[lookahead ∉ *set*]”出现在一个产生式的右部，表示若紧接着的下一个输入词法单元是给定的*set*的成员时不能使用该产生式。*set*可被写作由一对花括号括起的终结符列表。为了简便，*set*也可被写作一个非终结符，

【SP23】

此时它表示可由那个非终结符扩展成的全部终结符的集合。例如，给出下列定义：

*DecimalDigit* :: **其中一个**

**0 1 2 3 4 5 6 7 8 9**

*DecimalDigits* ::

*DecimalDigit*

*DecimalDigits DecimalDigit*

下面的定义：

*LookaheadExample* ::

**n** [lookahead ∉ {1,3,5,7,9}] *DecimalDigits*

*DecimalDigit* [lookahead ∉ *DecimalDigit*]

匹配：①字母n后跟一个或多个十进制数位，且第一个数位是偶数；②一个十进制数位，且后面没有另一个十进制数位。

如果短语“[no *LineTerminator* here]”出现在一个语法文法的产生式右部，表示该产生式是一个受限产生式：如果*LineTerminator*位于输入流的指定位置则该产生式不可用。例如，产生式：

*ThrowStatement* :

**throw** [no *LineTerminator* here] *Expression* **;**

表示如果一个*LineTerminator*在程序里位于词法单元**throw**和非终结符*Expression*之间则该产生式不可用。

除非*LineTerminator*的存在被受限产生式禁止，否则输入元素流里任何两个连续的词法单元之间可以出现任意数量的*LineTerminator*，而不会影响程序的语法接受性。

在词法文法或数字字符串文法的产生式里，当一个供选符号是以多字符词法单元的形式出现时【如**0x**】，它表示构成这样一个词法单元的字符序列。

一个产生式的右部可以通过使用词语“**但不是**”指定某些扩展是不允许的，继而表示那些扩展将被排除。例如，产生式：

*Identifier* ::

*IdentifierName* **但不是** *ReservedWord*

意思是非终结符*Identifier*可以被替换成任何可以替换*IdentifierName*的字符序列，倘若那个字符序列不能替换*ReservedWord*。

最后，少数几个非终结符使用一个描述性的短句描述，在列出所有供选是不切实际的情况下：

*SourceCharacter* ::

任何Unicode编码单位。

5.2节 算法约定

规范通常使用一个带编号的清单指定一个算法的各个步骤。这些算法用来精确地指定ECMAScript语言构造必要的语义。算法并不有意暗示使用任何特殊的实现技术。在实践中，可能会有更有效的算法可用于实现给定的功能。

【SP24】

为了使某些算法可在这份规范的多个地方使用，那些算法称作抽象操作，被命名并被写成参数化的函数形式，以便于可从其它算法里根据名字引用它们。

当一个算法将产生一个值作为结果时，指令“return x”用来表示算法的结果就是x的值，并且算法应该终止。记号Result(n)用作“步骤n的结果”的简写。

为了表述清晰，算法步骤可以被划分成顺次的子步骤。子步骤被缩进，且其自身也可以被进一步划分成缩进的子步骤。大纲编号约定被用于标识子步骤，第一级子步骤使用小写字母作为标号，第二级子步骤使用小写罗马数字。如果需要三级以上的子步骤，这些规则将从使用数字标号的第四级步骤开始重复。例如：

1. 顶级步骤

a. 第一级子步骤

b. 第一级子步骤

i. 第二级子步骤

ii. 第二级子步骤

1. 第三级子步骤，即第四级步骤

a. 第四级子步骤

一个步骤或子步骤可被写作一个“if”判定，用来制约它的子步骤。此时，只有当判定为真时才会应用那些子步骤。如果一个步骤或子步骤以单词“else”开头，它就作为与处于同一级里前面的“if”判定相反的判定。

一个步骤可以指定反复地应用它的子步骤。

一个步骤可以断言它的算法的不变条件。这些断言被用于显式地表明算法不变式，如不使用断言，那些不变式将被暗含。这些断言没有增加额外的语义要求，因此一个实现不需要检查这些断言，它们仅仅用来澄清算法。

数学操作，例如加，减，取负，乘，除，和本章稍后定义的数学函数，应该总是以在数学实数上计算准确的数学结果来理解，数学实数不包括正负无穷，也不包括与正零不同的负零。这份标准里的算法模塑了浮点算术，在需要的地方包含了一些显式步骤，用以处理正负无穷和有符号零并进行舍入。如果一个数学操作或数学函数被应用到一个浮点数，这应该作为应用到了由那个浮点数表示的准确的数学值来理解；这样一个浮点数必须是有限的，如果它是+0或-0，那么相应的数学值就是0。

数学函数abs(x)产生x的绝对值——如果x是负数（小于零）则结果是-x，否则是x本身。

数学函数sign(x)产生——如果x是正数则为1，如果x是负数则为-1，当x是零时在这份标准里不会使用该函数。

记号“x modulo y”（y必须有限且非零）计算一个与y正负号相同的值k（或0），使得对某个整数q，有abs(k)＜abs(y)，且x－k＝q×y。

数学函数floor(x)产生不大于x的最大整数（朝着正无穷方向）。

注意：floor(x)＝x－(x modulo 1)

如果一个算法按照定义要“抛出一个异常”，则算法的执行被终止且没有结果返回。调用算法也被终止，直到抵达了一个算法步骤，它使用术语例如“如果一个异常被抛出……”来显式地处理那个异常。一旦遇到了这样一个算法步骤，那么那个异常就不再被认为已经发生过。

【SP25】

第6章 源文本

ECMAScript源文本被表示成一个以Unicode字符编码的字符序列，版本3.0或后续版本。文本被寄望于已经标准化成Unicode归一化形式C（典型组成），如Unicode技术报告#15所述。遵从ECMAScript实现不需要进行任何文本标准化，也不需要如同它们自己进行文本标准化那般运作。这份规范假定ECMAScript源文本是一个16位编码单位序列。这样的源文本可以包含非有效UTF-16字符编码的16位编码单位序列。如果一个实际的源文本是以非16位编码单位编码的，它必须按照就好像它首先被转换成UTF-16来进行处理。

语法

*SourceCharacter* ::

any Unicode code unit

在这份文档剩余部分里，词组“编码单位【code unit】”和单词“字符【character】”用来表示一个16位无符号数值，该数值用于描述一个16位文本单位。词组“Unicode字符【Unicode character】”用来表示由一个Unicode标量值（长度可以超过16位，因此可能使用多个编码单位）表示的抽象语言或印刷单位。词组“编码点数【code point】”用来表示这样一个Unicode标量值。“Unicode字符”只表示由一个Unicode标量值表示的实体：一个联合字符序列的各个成分仍然是单独的“Unicode字符”，即使用户可能把整个序列看作一个字符。

在字符串字面值，正则表达式字面值，和标识符里，任何字符（编码单位）也可以被表示成一个Unicode转义序列，其由六个字符构成，也就是\u后跟四个十六进制数字。在注释里，这样的转义序列作为注释的一部分而被忽略。在字符串字面值或正则表达式字面值里，Unicode转义序列为字面值贡献一个值。在标识符里，转义序列为标识符贡献一个字符。

注意：尽管这份文档有时提到——在一个“字符串”里的一个“字符”和那个字符的编码单位的16位无符号整数之间的—— “变换”，但实际上它们之间没有变换，因为一个“字符串”里的一个“字符”实际上就是用那个16位无符号值表示的。

在Unicode转义序列的行为方面，ECMAScript不同于Java编程语言。在Java程序里，例如，如果Unicode转义序列\u000A存在于一个单行注释里，它被当作一个行终止符解释（Unicode字符000A就是换行符【line feed】），因此下一个字符不是注释的一部分。类似的，如果Unicode转义序列\u000A存在于一个字符串字面值里，它也被同样地解释成一个行终止符，这在字符串字面值里是不允许的——为使换行符成为字符串字面值的一部分，我们必须写\n而不是\u000A。但在ECMAScript程序里，存在于注释里的Unicode转义序列永远不会被解释，因此也不可能终止注释。类似的，存在于字符串字面值里的Unicode转义序列总是为字符串字面值贡献一个字符，也永远不会被解释成一个行终止符或可能终止字符串字面值的引号。

第7章 词法约定

一个ECMAScript程序的源文本首先被转换成一个输入元素的序列，这些输入元素是词法单元，行终止符，注释，或空白。源文本被从左向右扫描，重复地提取最长可能的字符序列作为下一个输入元素。

词法文法有两个目标符号。*InputElementDiv*符号用在允许一个前导除法（/）或除法赋值（/=）运算符的语法文法上下文里；*InputElementRegExp*用在其它语法文法上下文里。

注意：没有语法文法上下文既允许前导除法或除法赋值，也允许前导*RegularExpressionLiteral*。这不会受分号插入（参阅7.9节）的影响；在像下面这样的例子里：

【SP26】

a = b

/hi/g.exec(c).map(d);

在*LineTerminator*后的第一个非空白、非注释字符是斜杠（/），而且语法文法允许除法或除法赋值，在*LineTerminator*处没有分号被插入。也就是说，上面的例子和下面的以相同方式解释：

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

语法

*InputElementDiv* ::

*WhiteSpace*

*LineTerminator*

*Comment*

*Token*

*DivPunctuator*

*InputElementRegExp* ::

*WhiteSpace*

*LineTerminator*

*Comment*

*Token*

*RegularExpressionLiteral*

7.1节 Unicode格式控制字符

Unicode格式控制字符（也就是在Unicode字符数据库如LEFT-TO-RIGHT MARK或RIGHT-TO-LEFT MARK里的“Cf”类别下的字符）是一些控制码，用于在缺失高层协议的情况下控制一系列文本（例如标记语言）的格式化。

允许在源文本里使用格式控制字符以帮助编辑和显示是有益的。所有格式控制字符也可以用在注释，字符串字面值，和正则表达式字面值里。

<ZWNJ>和<ZWJ>是两个格式控制字符，用于当在某些语言里构成单词或短语时作出必要的区分。在ECMAScript源文本里，<ZWNJ>和<ZWJ>也可被用在标识符里的第一个字符之后。

<BOM>是一个格式控制字符，主要用在文本的开头，把这份文本标记为Unicode并允许检测文本的编码和字节顺序。用作这一目的的<BOM>字符有时也可以出现在文本的开头之后，例如作为拼接文件的结果。<BOM>字符被当作空白字符处理（参阅7.2节）。

在注释，字符串字面值，和正则表达式字面值之外特殊处理的某些格式控制字符总结在表格1里。

表格1 —— 格式控制字符用法

┌───────────────────────────────────┐

│编码单位值 名称 形式名 用法 │

├───────────────────────────────────┤

│\u200C 零宽非连结符【？】 <ZWNJ> *IdentifierPart* │

│\u200D 零宽连结符 <ZWJ> *IdentifierPart* │

│\uFEFF 字节顺序标记 <BOM> *WhiteSpace* │

└───────────────────────────────────┘

7.2节 空白

空白字符用于改进源文本可读性和分离词法单元（不可分割的词法单位），除此之外别无它用。空白字符可以出现在任何两个词法单元之间，以及输入的开头和结尾。空白字符也可以出现在*StringLiteral*或

【SP27】

*RegularExpressionLiteral*（在这两种情况下它们被认为是构成字面值一部分的重要字符）或*Comment*里，但不能出现在任何其它种类的词法单元里。

ECMAScript空白字符罗列在表格2里。

表格2 —— 空白字符

┌───────────────────────────────────┐

│编码单位值 名称 形式名 │

├───────────────────────────────────┤

│\u0009 水平制表 <TAB> │

│\u000B 垂直制表 <VT> │

│\u000C 换页 <FF> │

│\u0020 空格 <SP> │

│\u00A0 不间断空格 <NBSP> │

│\uFEFF 字节顺序标记 <BOM> │

│其它“Zs”类别 任何其它Unicode“空格分隔符” <USP> │

└───────────────────────────────────┘

ECMAScript实现必须识别定义在Unicode 3.0里的所有空白字符。Unicode标准后续版本可能定义其它空白字符，ECMAScript实现可以识别来自Unicode标准后续版本的空白字符。

语法

*WhiteSpace* ::

<TAB>

<VT>

<FF>

<SP>

<NBSP>

<BOM>

<USP>

7.3节 行终止符

类似空白字符，行终止符也被用于改进源文本可读性和分离词法单元（不可分割的词法单位）。然而，不像空白字符，行终止符对语法文法的行为具有一些影响。一般来说，行终止符可以出现在任何两个词法单元之间，但是在少数几个地方语法文法禁止它们出现。行终止符同样影响自动分号插入过程（7.9节）。行终止符不能出现在除*StringLiteral*以外的任何词法单元内部。行终止符只能作为一个*LineContinuation*出现在一个词法单元*StringLiteral*内部。

行终止符可以出现在*MultiLineComment*里（7.4节），但不能出现在*SingleLineComment*里。

行终止符被包含在能够由正则表达式的\s类匹配的空白字符集合里。

ECMAScript行终止符罗列在表格3里。

表格3 —— 行终止字符

┌───────────────────────────────────┐

│编码单位值 名称 形式名 │

├───────────────────────────────────┤

│\u000A 换行 <LF> │

│\u000D 回车 <CR> │

│\u2028 行分隔符 <LS> │

│\u2029 段分隔符 <PS> │

└───────────────────────────────────┘

【SP28】

只有表格3里的字符被当作行终止符处理。其它新行字符或行间断字符被当作空白而不是行终止符处理。字符序列<CR><LF>通常被作为一个行终止符来使用，为了报告行号这个序列应该算作一个字符。

语法

*LineTerminator* ::

<LF>

<CR>

<LS>

<PS>

*LineTerminatorSequence* ::

<LF>

<CR> [lookahead ∉ <LF>]

<LS>

<PS>

<CR><LF>

7.4节 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*LineTerminator*以外的任何字符，又因为一般规则是一个词法单元总是尽可能的长，所以一个单行注释总是由从**//**标记开始到行末为止的所有字符构成。但是，在行末的*LineTerminator*不被认为是单行注释的一部分；它由词法文法单独识别并成为语法文法输入元素流的一部分。这一点非常重要，因为这暗示着单行注释的存在与否并不影响自动分号插入过程（7.9节）。

注释的行为和空白类似并也被丢弃，但有个例外，如果一个*MultiLineComment*含有行终止符，那么为了由语法文法解析，整个多行注释被作为一个*LineTerminator*。

语法

*Comment* ::

*MultiLineComment*

*SingleLineComment*

*MultiLineComment* ::

**/\*** *MultiLineCommentChars* opt **\*/**

*MultiLineCommentChars* ::

*MultiLineNotAsteriskChar MultiLineCommentChars* opt

**\*** *PostAsteriskCommentChars* opt

*PostAsteriskCommentChars* ::

*MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentChars* opt

**\*** *PostAsteriskCommentChars* opt

*MultiLineNotAsteriskChar* ::

*SourceCharacter* **但不是 \***

*MultiLineNotForwardSlashOrAsteriskChar* ::

*SourceCharacter* **但不是 /**或**\***中的任何一个

【SP29】

*SingleLineComment* ::

**//** *SingleLineCommentChars* opt

*SingleLineCommentChars* ::

*SingleLineCommentChar SingleLineCommentChars* opt

*SingleLineCommentChar* ::

*SourceCharacter* **但不是** *LineTerminator*

7.5节 词法单元

语法

*Token* ::

*IdentifierName*

*Punctuator*

*NumericLiteral*

*StringLiteral*

注意：*DivPunctuator*和*RegularExpressionLiteral*产生式也定义了词法单元，但没有被包含在*Token*产生式里。

7.6节 标识符名字和标识符

标识符名字是一些将依照Unicode标准第5章“Identifiers”一节给出的、并带有少量修改的文法进行解释的词法单元。一个*Identifier*就是一个非*ReservedWord*（参阅7.6.1段）的*IdentifierName*。Unicode标识符文法是基于由Unicode标准规定的normative和informative字符类别。所有遵从ECMAScript实现必须把在Unicode标准3.0版中规定的类别里的字符作为那两个字符类别里的字符处理。

这份标准还增加了两个特殊字符：美元符号（$）和下划线（\_）可以出现在一个*IdentifierName*里的任何地方。

Unicode转义序列同样可以出现在一个*IdentifierName*里，此时它们为*IdentifierName*贡献一个字符，计算这个字符的规则如计算*UnicodeEscapeSequence*的CV那样（参阅7.8.4节）。放在*UnicodeEscapeSequence*前面的反斜杠“**\**”并不算作*IdentifierName*的字符。不能使用*UnicodeEscapeSequence*把一个非法字符放入*IdentifierName*里，换句话说，如果一个**\** *UnicodeEscapeSequence*被替换成*UnicodeEscapeSequence*的CV，结果必须仍然是一个有效的、具有和原始*IdentifierName*完全相同的字符序列的*IdentifierName*。在这份规范里所有对标识符的理解都是基于它们实际的字符而无论是否使用了转义序列来表示任何特殊字符。

两个依照Unicode标准在标准上等价的*IdentifierName*并不相等，除非它们是使用完全相同的编码单位序列表示的（换句话说，遵从ECMAScript实现只需要在*IdentifierName*值上进行逐位比较）。含义是输入源文本在到达编译器之前就已被转换成归一化形式C了。

ECMAScript实现可以识别定义在Unicode标准后续版本里的标识符字符。如果关心可移植性，程序员应该只使用定义在Unicode 3.0版里的标识符字符。

语法

*Identifier* ::

*IdentifierName* **但不是** *ReservedWord*

【SP30】

*IdentifierName* ::

*IdentifierStart*

*IdentifierName IdentifierPart*

*IdentifierStart* ::

*UnicodeLetter*

**$**

**\_**

**\** *UnicdoeEscapeSequence*

*IdentifierPart* ::

*IdentifierStart*

*UnicodeCombiningMark*

*UnicodeDigit*

*UnicodeConnectorPunctuation*

<ZWNJ>

<ZWJ>

*UnicodeLetter* ::

Unicode类别“Uppercase letter (Lu)”，“Lowercase letter (Ll)”，“Titlecase letter (Lt)”，“Modifier letter (Lm)”，“Other letter (Lo)”，或“Letter number (Nl)”里的任何字符。

*UnicodeCombiningMark* **::**

Unicode类别“Non-spacing mark (Mn)”或“Combining spacing mark (Mc)”里的任何字符。

*UnicodeDigit* **::**

Unicode类别“Decimal number (Nd)”里的任何字符。

*UnicodeConnectorPunctuation* **::**

Unicode类别“Connector punctuation (Pc)”里的任何字符。

非终结符*UnicodeEscapeSequence*在7.8.4段给出。

7.6.1段 保留字

一个保留字是一个不能被用作*Identifier*的*IdentifierName*。

语法

*ReservedWord* ::

*Keyword*

*FutureReservedWord*

*NullLiteral*

*BooleanLiteral*

7.6.1.1则 关键字

以下词法单元就是ECMAScript关键字，它们在ECMAScript程序里不能用作*Identifier*。

语法

*Keyword* :: **其中一个**

**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**

7.6.1.2则 将来保留字

以下单词在提议的扩展里用作关键字，因此被保留以考虑将来采纳那些扩展的可能。

语法

*FutureReservedWord* :: **其中一个**

**class enum extends super**

**const export import**

以下词法单元当出现在严格模式代码里时（参阅10.1.1段）也被认为是*FutureReservedWord*。这些词法单元中的任何一个若出现在——严格模式代码里的任何因为*FutureReservedWord*的存在而产生错误的——上下文里时，必须同样产生一个等价的错误：

**implements let private public yield**

**interface package protected static**

7.7节 标点符号

语法

*Punctuator* :: **其中一个**

**{ } ( ) [ ]**

**. ; , < > <=**

**>= == != === !==**

**+ - \* % ++ --**

**<< >> >>> & | ^**

**! ~ && || ? :**

**= += -= \*= %= <<=**

**>>= >>>= &= |= ^=**

*DivPunctuator* :: **其中一个**

**/ /=**

7.8节 字面值

语法

*Literal* ::

*NullLiteral*

*BooleanLiteral*

*NumericLiteral*

*StringLiteral*

*RegularExpressionLiteral*

7.8.1段 空值字面值

语法

*NullLiteral* ::

**null**

【SP32】

语义

空值字面值的值**null**是Null类型的唯一值。

7.8.2段 布尔字面值

语法

*BooleanLiteral* ::

**true**

**false**

语义

布尔字面值的值**true**是一个Boolean类型的值。

布尔字面值的值**false**是一个Boolean类型的值。

7.8.3段 数字字面值

语法

*NumericLiteral* ::

*DecimalLiteral*

*HexIntegerLiteral*

*DecimalLiteral* ::

*DecimalIntegerLiteral* **.** *DecimalDigits* opt *ExponentPart* opt

**.** *DecimalDigits ExponentPart* opt

*DecimalIntegerLiteral ExponentPart* opt

*DecimalIntegerLiteral* ::

**0**

*NonZeroDigit DecimalDigits* opt

*DecimalDigits* ::

*DecimalDigit*

*DecimalDigits DecimalDigit*

*DecimalDigit* :: **其中一个**

**0 1 2 3 4 5 6 7 8 9**

*NonZeroDigit* :: **其中一个**

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

*ExponentPart* ::

*ExponentIndicator SignedInteger*

*ExponentIndicator* :: **其中一个**

**e E**

*SignedInteger* ::

*DecimalDigits*

**+** *DecimalDigits*

**-** *DecimalDigits*

*HexIntegerLiteral* ::

**0x** *HexDigit*

**0X** *HexDigit*

*HexIntegerLiteral HexDigit*

【SP33】

*HexDigit* :: **其中一个**

**0 1 2 3 4 5 6 7 8 9**

**a b c d e f**

**A B C D E F**

立即跟在一个*NumericLiteral*之后的源字符绝不能是*IdentifierStart*或*DecimalDigit*。

注意：例如

**3in**

是一个错误而不是两个输入元素**3**和**in**。

语义

一个数字字面值代表一个Number类型的值。这个值分两步确定：首先，一个数学值（MV）从字面值推导而来；然后，这个数学值按下面的描述进行舍入。

·*NumericLiteral* :: *DecimalLiteral*的MV就是*DecimalLiteral*的MV。

·*NumericLiteral* :: *HexIntegerLiteral*的MV就是*HexIntegerLiteral*的MV。

·*DecimalLiteral* :: *DecimalIntegerLiteral* **.**的MV就是*DecimalIntegerLiteral*的MV。

·*DecimalLiteral* :: *DecimalIntegerLiteral* **.** *DecimalDigits*的MV就是*DecimalIntegerLiteral*的MV加上(*DecimalDigits*的MV乘以10^-n)，其中n是*DecimalDigits*中的字符数【圆括号用于控制运算优先级】。

·*DecimalLiteral* :: *DecimalIntegerLiteral* **.** *ExponentPart*的MV就是*DecimalIntegerLiteral*的MV乘以10^e，其中e是*ExponentPart*的MV。

·*DecimalLiteral* :: *DecimalIntegerLiteral* **.** *DecimalDigits ExponentPart*的MV就是(*DecimalIntegerLiteral*的MV加上(*DecimalDigits*的MV乘以10^-n))乘以10^e，其中n是*DecimalDigits*中的字符数，e是*ExponentPart*的MV。

·*DecimalLiteral* ::**.** *DecimalDigits*的MV就是*DecimalDigits*的MV乘以10^-n，其中n是*DecimalDigits*中的字符数。

·*DecimalLiteral* ::**.** *DecimalDigits ExponentPart*的MV就是*DecimalDigits*的MV乘以10^(e－n)，其中n是*DecimalDigits*中的字符数，e是*ExponentPart*的MV。

·*DecimalLiteral* :: *DecimalIntegerLiteral*的MV就是*DecimalIntegerLiteral*的MV。

·*DecimalLiteral* :: *DecimalIntegerLiteral ExponentPart*的MV就是*DecimalIntegerLiteral*的MV乘以10^e，其中e是*ExponentPart*的MV。

·*DecimalIntegerLiteral* :: **0**的MV就是0。

·*DecimalIntegerLiteral* :: *NonZeroDigit*的MV就是*NonZeroDigit*的MV。

·*DecimalIntegerLiteral* :: *NonZeroDigit DecimalDigits*的MV就是(*NonZeroDigit*的MV乘以10^n)加上*DecimalDigits*的MV，其中n是*DecimalDigits*中的字符数。

·*DecimalDigits* :: *DecimalDigit*的MV就是*DecimalDigit*的MV。

·*DecimalDigits* :: *DecimalDigits DecimalDigit*的MV就是(*DecimalDigits*的MV乘以10)加上*DecimalDigit*的MV。

·*ExponentPart* :: *ExponentIndicator SignedInteger*的MV就是*SignedInteger*的MV。

·*SignedInteger* :: *DecimalDigits*的MV就是*DecimalDigits*的MV。

·*SignedInteger* :: **+** *DecimalDigits*的MV就是*DecimalDigits*的MV。

·*SignedInteger* :: **-** *DecimalDigits*的MV就是*DecimalDigits*的MV的相反数。

·*DecimalDigit* :: **0**或*HexDigit* :: **0**的MV就是0。

·*DecimalDigit* :: **1**或*NonZeroDigit* :: **1**或*HexDigit* :: **1**的MV就是1。

·*DecimalDigit* :: **2**或*NonZeroDigit* :: **2**或*HexDigit* :: **2**的MV就是2。

·*DecimalDigit* :: **3**或*NonZeroDigit* :: **3**或*HexDigit* :: **3**的MV就是3。

·*DecimalDigit* :: **4**或*NonZeroDigit* :: **4**或*HexDigit* :: **4**的MV就是4。

·*DecimalDigit* :: **5**或*NonZeroDigit* :: **5**或*HexDigit* :: **5**的MV就是5。

·*DecimalDigit* :: **6**或*NonZeroDigit* :: **6**或*HexDigit* :: **6**的MV就是6。

·*DecimalDigit* :: **7**或*NonZeroDigit* :: **7**或*HexDigit* :: **7**的MV就是7。

·*DecimalDigit* :: **8**或*NonZeroDigit* :: **8**或*HexDigit* :: **8**的MV就是8。

·*DecimalDigit* :: **9**或*NonZeroDigit* :: **9**或*HexDigit* :: **9**的MV就是9。

·*HexDigit* :: **a**或*HexDigit* :: **A**的MV就是10。

【SP34】

·*HexDigit* :: **b**或*HexDigit* :: **B**的MV就是11。

·*HexDigit* :: **c**或*HexDigit* :: **C**的MV就是12。

·*HexDigit* :: **d**或*HexDigit* :: **D**的MV就是13。

·*HexDigit* :: **e**或*HexDigit* :: **E**的MV就是14。

·*HexDigit* :: **f**或*HexDigit* :: **F**的MV就是15。

·*HexIntegerLiteral* :: **0x** *HexDigit*的MV就是*HexDigit*的MV。

·*HexIntegerLiteral* :: **0X** *HexDigit*的MV就是*HexDigit*的MV。

·*HexIntegerLiteral* :: *HexIntegerLiteral* *HexDigit*的MV就是(*HexIntegerLiteral*的MV乘以16)加上*HexDigit*的MV。

一个数字字面值准确的MV一经被确定，它就被舍入成一个Number类型的值。如果MV是0，则舍入值是**+0**；否则，舍入值一定是MV的Number值（依8.5节规定），除非字面值是一个*DecimalLiteral*且字面值具有多于20个有效数位，此时Number值可以是——使用数位0替换第20位之后的每个有效数位产生的——字面值的MV的Number值，或是——使用数位0替换第20位之后的每个有效数位，然后把位于第20个有效数位上的字面值加1产生的——字面值的MV的Number值。一个数位是有效的，如果它不是*ExponentPart*的一部分，且

·它不是0；或

·既有一个非零数位在它的左边，又有一个非零数位在它的右边且不在*ExponentPart*里。

一个遵从实现，当处理严格模式代码（参阅10.1.1段）时，绝不能扩展*NumericLiteral*的语法以包含*OctalIntegerLiteral*，如附录B.1.1节所述。

7.8.4段 字符串字面值

一个字符串字面值就是包含在单引号或双引号里的零个或多个字符。每个字符可以用一个转义序列表示。所有字符可以在字面上出现在一个字符串字面值里，除了闭合引号字符，反斜杠符，回车符，行分隔符，段分隔符，和换行符。任何字符都能以转义序列的形式出现。

语法

*StringLiteral* ::

**“** *DoubleStringCharacters* opt **”**

**‘** *SingleStringCharacters*opt **’**

*DoubleStringCharacters* ::

*DoubleStringCharacter DoubleStringCharacters* opt

*SingleStringCharacters* ::

*SingleStringCharacter SingleStringCharacters* opt

*DoubleStringCharacter* ::

*SourceCharacter* **但不是 ”**或**\**或*LineTerminator*中的任何一个

**\** *EscapeSequence*

*LineContinuation*

*SingleStringCharacter* ::

*SourceCharacter* **但不是 ’**或**\**或*LineTerminator*中的任何一个

**\** *EscapeSequence*

*LineContinuation*

*LineContinuation* ::

**\** *LineTerminatorSequence*

【SP35】

*EscapeSequence* ::

*CharacterEscapeSequence*

**0** [lookahead ∉ *DecimalDigit*]

*HexEscapeSequence*

*UnicodeEscapeSequence*

*CharacterEscapeSequence* ::

*SingleEscapeCharacter*

*NonEscapeCharacter*

*SingleEscapeCharacter* :: **其中一个**

**’ ” \ b f n r t v**

*NonEscapeCharacter* ::

*SourceCharacter* **但不是** *EscapeCharacter*或*LineTerminator*中的任何一个

*EscapeCharacter* ::

*SingleEscapeCharacter*

*DecimalDigit*

**x**

**u**

*HexEscapeSequence* ::

**x** *HexDigit HexDigit*

*UnicodeEscapeSequence* ::

**u** *HexDigit HexDigit HexDigit HexDigit*

非终结符*HexDigit*的定义在7.8.3段给出，*SourceCharacter*的定义在第6章。

语义

一个字符串字面值代表一个String类型的值。字面值的String值（SV）是从字符串字面值各个部分贡献的字符值（CV）的角度描述的。作为这个过程的一部分，字符串字面值里的一些字符被解释成具有一个数学值（MV），如下面或7.8.3段所述：

·*StringLiteral* :: **“”**的SV就是空字符序列。

·*StringLiteral* :: **‘’**的SV就是空字符序列。

·*StringLiteral* :: **“** *DoubleStringCharacters* **”**的SV就是*DoubleStringCharacters*的SV。

·*StringLiteral* :: **‘** *SingleStringCharacters* **’**的SV就是*SingleStringCharacters*的SV。

·*DoubleStringCharacters* :: *DoubleStringCharacter*的SV就是*DoubleStringCharacter*的CV组成的单字符序列。

·*DoubleStringCharacters* :: *DoubleStringCharacter DoubleStringCharacters*的SV就是*DoubleStringCharacter*的CV后面依次跟着*DoubleStringCharacters*的SV里的所有字符组成的序列。

·*SingleStringCharacters* :: *SingleStringCharacter*的SV就是*SingleStringCharacter*的CV组成的单字符序列。

·*SingleStringCharacters* :: *SingleStringCharacter SingleStringCharacters*的SV就是*SingleStringCharacter*的CV后面依次跟着*SingleStringCharacters*的SV里的所有字符组成的序列。

·*LineContinuation* :: **\** *LineTerminatorSequence*的SV就是空字符序列。

·*DoubleStringCharacter* :: *SourceCharacter* **但不是 ”**或**\**或*LineTerminator*中的任何一个 的CV就是*SourceCharacter*字符本身。

·*DoubleStringCharacter* :: **\** *EscapeSequence*的CV就是*EscapeSequence*的CV。

·*DoubleStringCharacter* :: *LineContinuation*的CV就是空字符序列。

·*SingleStringCharacter* :: *SourceCharacter* **但不是 ’**或**\**或*LineTerminator*中的任何一个 的CV就是*SourceCharacter*字符本身。

·*SingleStringCharacter* :: **\** *EscapeSequence*的CV就是*EscapeSequence*的CV。

【SP36】

·*SingleStringCharacter* :: *LineContinuation*的CV就是空字符序列。

·*EscapeSequence* :: *CharacterEscapeSequence*的CV就是*CharacterEscapeSequence*的CV。

·*EscapeSequence* :: **0** [lookahead ∉ *DecimalDigit*]的CV就是一个<NUL>字符（Unicode值0000）。

·*EscapeSequence* :: *HexEscapeSequence*的CV就是*HexEscapeSequence*的CV。

·*EscapeSequence* :: *UnicodeEscapeSequence*的CV就是*UnicodeEscapeSequence*的CV。

·*CharacterEscapeSequence* :: *SingleEscapeCharacter*的CV就是一个字符，其编码单位值由*SingleEscapeCharacter*依照表格4确定。

表格4 —— 字符串单字符转义序列

┌───────────────────────────────────┐

│转义序列 编码单位值 名称 符号 │

├───────────────────────────────────┤

│\b \u0008 退格 <BS> │

│\t \u0009 水平制表 <HT> │

│\n \u000A 换行（新行） <LF> │

│\v \u000B 垂直制表 <VT> │

│\f \u000C 换页 <FF> │

│\r \u000D 回车 <CR> │

│\” \u0022 双引号 ” │

│\’ \u0027 单引号 ’ │

│\\ \u005C 反斜杠 \ │

└───────────────────────────────────┘

·*CharacterEscapeSequence* :: *NonEscapeCharacter* 的CV就是*NonEscapeCharacter*的CV。

·*NonEscapeCharacter* :: *SourceCharacter* **但不是** *EscapeCharacter*或*LineTerminator*中的任何一个 的CV就是*SourceCharacter*字符本身。

·*HexEscapeSequence* :: **x** *HexDigit HexDigit*的CV就是一个字符，其编码单位值为(16乘以第一个*HexDigit*的MV)加上第二个*HexDigit*的MV。

·*UnicodeEscapeSequence* :: **u** *HexDigit HexDigit HexDigit HexDigit*的CV就是一个字符，其编码单位值为(4096乘以第一个*HexDigit*的MV)加上(256乘以第二个*HexDigit*的MV)加上(16乘以第三个*HexDigit*的MV)加上第四个*HexDigit*的MV。

一个遵从实现，当处理严格模式代码（参阅10.1.1段）时，不可以扩展*EscapeSequence*的语法以包含*OctalEscapeSequence*，如附录B.1.2节所述。

注意：行终止符不能出现在字符串字面值里，除非作为*LineContinuation*的一部分以产生空字符序列。为使行终止符成为一个字符串字面值的String值的一部分，正确方式是使用转义序列，例如**\n**或**\u000A**。

7.8.5段 正则表达式字面值

一个正则表达式字面值是一个输入元素，每当该字面值被求值时就被转换成一个RegExp对象（参阅15.10节）。永远不要使用===比较程序里的两个正则表达式字面值求值得到的正则表达式对象，即使字面值的内容完全一致。一个RegExp对象也可以在运行时创建，通过**new** RegExp(参阅15.10.4段)或把RegExp构造函数作为普通函数调用。

下面的产生式描述了正则表达式字面值语法，并被输入元素扫描器用于找出正则表达式字面值的结束位置。构成*RegularExpressionBody*和*RegularExpressionFlags*的字符串不经解释的传给正则表达式构造函数，正则表达式构造函数根据它自己的、更为严格的文法予以解释。一个实现可以扩展正则表达式构造函数的文法，但绝不能扩展*RegularExpressionBody*和*RegularExpressionFlags*产生式或由这些产生式使用的产生式。

【SP37】

语法

*RegularExpressionLiteral* ::

**/** *RegularExpressionBody* **/** *RegularExpressionFlags*

*RegularExpressionBody* ::

*RegularExpressionFirstChar RegularExpressionChars*

*RegularExpressionChars* ::

[empty]

*RegularExpressionChars RegularExpressionChar*

*RegularExpressionFirstChar* ::

*RegularExpressionNonTerminator* **但不是 \***或**\**或**/**或**[**中的任何一个

*RegularExpressionBackslashSequence*

*RegularExpressionClass*

*RegularExpressionChar* ::

*RegularExpressionNonTerminator* **但不是 \**或**/**或**[**中的任何一个

*RegularExpressionBackslashSequence*

*RegularExpressionClass*

*RegularExpressionBackslashSequence* ::

**\** *RegularExpressionNonTerminator*

*RegularExpressionNonTerminator* ::

*SourceCharacter* **但不是** *LineTerminator*

*RegularExpressionClass* ::

**[** *RegularExpressionClassChars* **]**

*RegularExpressionClassChars* ::

[empty]

*RegularExpressionClassChars RegularExpressionClassChar*

*RegularExpressionClassChar* ::

*RegularExpressionNonTerminator* **但不是 ]或\中的任何一个**

*RegularExpressionBackslashSequence*

*RegularExpressionFlags* ::

[empty]

*RegularExpressionFlags IdentifierPart*

注意：正则表达式字面值不能为空；双斜杠**//**开启了一个单行注释而不是表示一个空正则表达式。要指定一个空正则表达式，使用**/(?:)/**。

语义

一个正则表达式字面值求值为一个Object类型的值，它是由标准内建RegExp构造函数创建的实例。这个值分两步确定：首先，构成正则表达式的*RegularExpressionBody*和*RegularExpressionFlags*产生式展开式的字符不加解释的分别收集进两个字符串Pattern和Flags里。然后每当该字面值被求值时，一个新对象被创建出来，就好像使用表达式**new** RegExp(Pattern, Flags)，其中RegExp是标准内建构造函数。新构造的对象成为*RegularExpressionLiteral*的值。如果对**new** RegExp的调用按照15.10.4.1条规定将产生一个错误，这个错误必须被作为一个早期错误（第16章）。

【SP38】

7.9节 自动分号插入

某些ECMAScript语句（空语句，变量语句，表达式语句，**do-while**语句，**continue**语句，**break**语句，**return**语句，和**throw**语句）必须以分号结束。这些分号也许总是显式出现在源文本里。然而为了方便，这些分号在某些情况下也可以从源文本里省略。可以如此描述这些情况：我们称在那些情况下分号被自动插入到源代码词法单元流里。

7.9.1段 自动分号插入规则

分号插入有三个基本规则：

1. 随着程序从左向右被解析，当遇到一个不被文法的任何产生式允许的词法单元（称为违规词法单元【offending token】）时，那么一个分号被自动插入到违规词法单元之前，如果下述条件至少一个成立：

·违规词法单元与前一个词法单元至少相隔一个*LineTerminator*。

·违规词法单元是**}**。

2. 随着程序从左向右被解析，当到达词法单元输入流的末尾，且解析器不能把输入词法单元流解析成一个完整的ECMAScript *Program*时，那么一个分号被自动插入到输入流的末尾。

3. 随着程序从左向右被解析，当遇到一个被文法的某个产生式允许的词法单元，但那个产生式是个受限产生式，且该词法单元是——用于立即跟在受限产生式里的注解“[no *LineTerminator* here]”后的终结符或非终结符的——第一个词法单元（因此这样的词法单元被称为受限词法单元），且受限词法单元与前一个词法单元至少相隔一个*LineTerminator*时，那么一个分号被自动插入到受限词法单元之前。

然而，有一个附加的条件覆盖在前述规则之上：一个分号永远不会被自动插入，如果该分号插入后将被解析成一条空语句，或将成为**for**语句（参阅12.6.3段）头部的两个分号之一。

注意：在文法里只有下列产生式才是受限产生式：

*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践效果如下所述：

在解析器会将**++**或**--**作为后缀运算符处理的地方，当遇到一个**++**或**--**词法单元，且在前一个词法单元和**++**或**--**词法单元之间至少存在一个*LineTerminator*时，那么一个分号被自动插入到那个**++**或**--**词法单元之前。

【SP39】

当遇到一个**continue**，**break**，**return**，或**throw**词法单元，且在下一个词法单元前遇到一个*LineTerminator*，一个分号被自动插入到**continue**，**break**，**return**，或**throw**词法单元之后。

由此，给ECMAScript程序员的实践忠告是：

一个后缀**++**或**--**运算符应该和它的运算对象出现在同一行上。

一个在**return**或**throw**语句里的*Expression*应该始发自和**return**或**throw**词法单元相同的一行。

一个在**break**或**continue**语句里的*Identifier*应该和**break**或**continue**词法单元在同一行上。

7.9.2段 自动分号插入示例

源代码

**{ 1 2 } 3**

在ECMAScript文法里不是一个有效的句子，即使借助自动分号插入规则。作为对比，源代码

**{ 1**

**2 } 3**

也不是一个有效的ECMAScript句子，但被自动分号插入变换成下面的样子：

**{ 1**

**;2 ;} 3;**

这是一个有效的ECMAScript句子。

源代码

**for (a; b**

**)**

不是一个有效的ECMAScript句子，也不会被自动分号插入所改动，因为分号对于**for**语句的头部是必须的。自动分号插入永远不会在**for**语句的头部里插入两个分号中的任何一个。

源代码

**return**

**a + b**

被自动分号插入变换成下面的样子：

**return;**

**a + b;**

注意：表达式**a + b**不会作为**return**语句将要返回的值处理，因为有一个*LineTerminator*把它和词法单元**return**分隔开。

源代码

**a = b**

**++c**

被自动分号插入变换成下面的样子：

**a = b;**

**++c;**

注意：词法单元**++**不会被当作应用到变量**b**的后缀运算符处理，因为在**b**和**++**之间存在一个*LineTerminator*。

源代码

**if (a > b)**

**else c = d**

不是一个有效的ECMAScript句子，也不会在**else**词法单元之前被自动分号插入所改动，即使在那一位置没有文法的产生式可以应用，因为一个自动插入的分号将被解析成一条空语句。

源代码

**a = b + c**

**(d + e).print()**

不会被自动分号插入变换，因为在第二行开始的圆括号表达式可被解释成一个函数调用的实参列表：

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

在赋值表达式必须以一个左圆括号开始的场合里，对于程序员来说，在前一语句的末尾提供一个显式的分号而不是依赖自动分号插入是个好主意。

第8章 类型

每一个由这份规范里的算法所操纵的值都有一个关联的类型。可能的值类型正是那些定义在本章里的类型。类型被进一步细分成ECMAScript语言类型和规范类型。

一个ECMAScript语言类型对应到由ECMAScript程序员使用ECMAScript语言直接操纵的值。ECMAScript语言类型有Undefined，Null，Boolean，String，Number，和Object。

一个规范类型对应到在算法里使用的元值【meta-values】，元值用来描述ECMAScript语言构造和ECMAScript语言类型的语义。规范类型有Reference，List，Completion，Property Descriptor，Property Identifier，Lexical Environment，和Environment Record。规范类型值都是规范自造品，不需要对应到在ECMAScript实现里的任何特殊实体。规范类型值可以被用来描述ECMAScript表达式求值的中间结果，但这些值不能作为对象的属性或ECMAScript语言变量的值存储。

在这份规范里，记号“Type(x)”用作“x的类型”的简写，“类型”表示在本章定义的ECMAScript语言类型和规范类型。

8.1节 Undefined类型

Undefined类型只有一个值，叫做**undefined**。任何还没有被赋值的变量具有值**undefined**。

8.2节 Null类型

Null类型只有一个值，叫做**null**。

8.3节 Boolean类型

Boolean类型表示一个逻辑学的实体，具有两个值，分别叫做**true**和**false**。

8.4节 String类型

String类型是——由零个或多个16位无符号整数值（“元素”）组成的全部有限有序序列的——集合。在一个运行中的ECMAScript程序里，String类型一般用来表示文本数据，在这种情况下String里的每个元素被当作一个编码单位值（参阅第6章）处理。每个元素被认为在序列中占据一个位置，这些位置使用非负整数进行索引。第一个元素（如果有）在位置0，下一个元素（如果有）在位置1，等等。

【SP41】

String的长度就是它包含的元素（即16位值）的数量。空String长度为0，因此没有包含元素。

当一个String包含了实际的文本数据，每个元素被认为是单独一个UTF-16编码单位。无论这是否是一个String的实际存储格式，String所含的字符根据它们最初的编码单位元素位置进行编号，就如同它们是使用UTF-16表示的一样。String上的所有操作（除非另附说明）把String当作和16位无符号整数序列无差异的序列处理；这些操作并不保证得到的结果String处于归一化形式，也不保证得到语言敏感的【language-sensitive】结果。

注意：这种设计背后的理论基础是为了保持String的实现尽可能的简单和高效。含义在于文本数据从外面（例如用户输入，从文件读取或网络接收的文本）进入到执行环境里，在运行中的程序看到它之前就已被转换成Unicode归一化形式C。通常这与入境文本从它的原始字符编码转换成Unicode同时发生（且不会强加额外的开销）。因为推荐ECMAScript源代码处于归一化形式C，所以字符串字面值被保证将被归一化（如果源文本被保证将被归一化），只要它们没有包含任何Unicode转义序列。

8.5节 Number类型

Number类型恰有18437736874454810627（即2^64－2^53＋3）个值，表示双精度64位格式IEEE 754值，如用于二进制浮点算术的IEEE标准所规定，除了IEEE标准的9007199254740990（即2^53－2）个不同的“非数字”值在ECMAScript里被表示成单个特殊的非数字值（注意该非数字值是由程序表达式**NaN**产生的）。在一些实现里，外部代码可能会探测到在各种非数字值之间的差别，但这种行为是实现相关的；对于ECMAScript代码来说，所有非数字值彼此是不可区分的。

还有其它两个特殊值，分别叫做正无穷和负无穷。为了简明，这两个值也分别使用符号**+∞**和**-∞**表示（注意这两个无穷Number值是由程序表达式**+Infinity**（或**Infinity**）和**- Infinity**产生的）。

其它的18437736874454810624（即2^64－2^53）个值叫做有限数。其中一半是正数而另一半是负数；对于每个有限正Number值，有一个对应的具有相同模【绝对值】的负值。

注意既有一个正零也有一个负零。为了简明，这两个值也分别使用符号+**0**和-**0**表示（注意这两个不同的零Number值是由程序表达式**+0**（或**0**）和**-0**产生的）。

18437736874454810622（即2^64―2^53―2）个有限非零值分为两类：它们中的18428729675200069632（即2^64―2^54）个值是标准化的，具有形式s×m×2^e，其中s是+1或-1，m是一个正整数，2^52≤m＜2^53，e是个整数，e∈[-1074, 971]。余下的9007199254740990（即2^53－2）个值是未标准化的，具有形式s×m×2^e，其中s是+1或-1，m是一个正整数，m＜2^52，e是-1074。

注意所有模不大于2^53的正整数和负整数在Number类型里是可以表示的（实际上整数0具有两种表示，**+0**和**-0**）。

【SP42】

一个有限数具有一个奇有效数，如果它非零且用来（在上面展示的两种形式之一里）表示它的整数m是奇数；否则，它具有一个偶有效数。

在这份规范里，短句“x的Number值”，其中x表示一个严格非零实数学量（甚至可以是一个无理数如π），意思是一个按下述方式选择的Number值：考虑Number类型的全部有限值的集合，从中移除-0并加入两个在Number类型里不可表示的值，也就是2^1024（即+1×2^53×2^971）和-2^1024（即-1×2^53×2^971）。选择这个集合里在数值上与x最接近的成员，如果集合里有两个值接近程度相同，那么选择带有偶有效数的那个；因为这一点，两个附加值2^1024和-2^1024都被认为具有偶有效数。最后，如果选择了2^1024，把它替换成**+∞**；如果选择了-2^1024，把它替换成**-∞**；如果选择了**+0**，把它替换成**-0**当且仅当x＜0；选择的任何其它值保持不变。结果就是x的Number值。（这一过程严格对应到IEEE 754“舍入到最近”模式的行为。）

一些ECMAScript运算符只处理位于区间[-2^31, 2^31－1] 或[0, 2^32－1]内的整数。这些运算符接受Number类型的任何值但首先把每个这样的值转换成2^32个整数值中的一个。参阅ToInt32和ToUint32运算符的描述，分别位于9.5节和9.6节。

8.6节 Object类型

一个Object是一个属性的集合。每个属性要么是个具名数据属性，要么是个具名存取器属性，要么是个内部属性：

·一个具名数据属性把一个名字关联到一个ECMAScript语言值和一组布尔特性。

·一个具名存取器属性把一个名字关联到一个或两个存取器函数，和一组布尔特性。存取器函数被用于存入或取回一个和该属性关联的ECMAScript语言值。

·一个内部属性没有名字且不是通过ECMAScript语言运算符可以直接访问到的。内部属性纯粹是为了这份规范而存在。【在这份规范的一些讲解过程中需要用到内部属性】

对于具名（非内部）属性有两种访问方式：获取【get】和设置【put】，分别对应取回和赋值。

8.6.1段 属性特性

在这份规范里使用特性来定义和解释具名属性的状态。一个具名数据属性把一个名字和罗列在表格5中的特性相关联。

表格5 —— 一个具名数据属性的特性

┌─────────────────────────────────────┐

│特性名 值定义域 描述 │

├─────────────────────────────────────┤

│[[Value]] 任何ECMAScript语言类型 #1 │

│[[Writable]] Boolean #2 │

│[[Enumerable]] Boolean #3 │

│[[Configurable]] Boolean #4 │

└─────────────────────────────────────┘

#1 通过读取属性取回的值。

#2 若为**false**，则当ECMAScript代码试图使用[[Put]]改变属性的[[Value]]时将不会成功。

#3 若为**true**，该属性将被for-in列举列出（参阅12.6.4段）；否则，称该属性是不可列举的。

#4 若为**false**，试图删除该属性，或把该属性改变成存取器属性，或改变它的特性（[[Value]] 除外）将会失败。

一个具名存取器属性把一个名字和罗列在表格6中的特性相关联。

【SP43】

表格6 —— 一个具名存取器属性的特性

┌─────────────────────────────────────┐

│特性名 值定义域 描述 │

├─────────────────────────────────────┤

│[[Get]] Object或Undefined #1 │

│[[Set]] Object或Undefined #2 │

│[[Enumerable]] Boolean #3 │

│[[Configurable]] Boolean #4 │

└─────────────────────────────────────┘

#1 如果值是一个Object则必须是一个函数Object。每次执行属性的获取访问都会调用该函数的[[Call]]内部方法（8.6.2段）并传入空实参列表，以返回属性值。

#2 如果值是一个Object则必须是一个函数Object。每次执行属性的设置访问都会调用该函数的[[Call]]内部方法（8.6.2段）并传入实参列表，其中含有被赋予的值，作为函数的唯一实参。一个属性的[[Set]]内部方法的效果可以，但不必，在——后续对属性的[[Get]]内部方法调用返回的——值上产生影响。

#3 若为**true**，该属性将被for-in列举列出（参阅12.6.4段）；否则，称该属性是不可列举的。

#4 若为**false**，试图删除该属性，或把该属性改变成数据属性，或改变它的特性将会失败。

如果这份规范并没有为一个具名属性显式规定一个特性的值，则使用定义在表格7里的默认值：

表格7 —— 默认特性值

┌───────────────────────┐

│特性名 默认值 │

├───────────────────────┤

│[[Value]] **undefined** │

│[[Get]] **undefined** │

│[[Set]] **undefined** │

│[[Writable]] **false** │

│[[Enumerable]] **false** │

│[[Configurable]] **false** │

└───────────────────────┘

8.6.2段 Object内部属性和方法

这份规范使用多种内部属性来定义Object值的语义。这些内部属性不是ECMAScript语言的一部分。这份规范定义它们纯粹是为了讲解目的。一个ECMAScript实现必须如同它以这里描述的方式产生和操作内部属性那般运作。内部属性的名字被包含在双方括号[[ ]]里。当一个算法用到了一个对象的一个内部属性，且那个对象没有实现指定的内部属性，则一个TypeError异常被抛出。

表格8总结了由这份规范使用的、适用于所有ECMAScript对象的内部属性。表格9总结了由这份规范使用的、仅适用于某些ECMAScript对象的内部属性。这些表格里的描述指定了它们对于原生ECMAScript对象的行为，除非在这份文档里为特殊几种原生ECMAScript对象另附说明。宿主对象可以借助任何实现相关的行为来支持这些内部属性，只要它符合在这份文档里陈述的、明确的宿主对象限制。

下面表格的“值类型定义域”列定义了和内部属性关联的值的类型，类型名称指向定义在本章里的类型，并使用以下附加的名称扩充：“any”表示值可以是任何ECMAScript语言类型；“primitive”表示Undefined，Null，Boolean，String，或Number；“SpecOp”表示内部属性是一个内部方法，即一个由抽象操作规范定义的、实现提供的过程。“SpecOp”后跟一个描述性形参名称的列表，如果一个形参名称和一个类型名称相同，那么该名称就描述了该形参的类型。

【SP44】

如果一个“SpecOp”返回了一个值，它的形参列表就会后跟符号“→”及返回值的类型。

表格8 —— 所有对象共同的内部属性

PN：propertyName，属性名称【为方便排版】

PD：PropertyDescriptor，属性描述符【为方便排版】

┌─────────────────────────────────────┐

│内部属性 值类型定义域 描述 │

├─────────────────────────────────────┤

│[[Prototype]] Object或Null #1 │

│[[Class]] String #2 │

│[[Extensible]] Boolean #3 │

│[[Get]] SpecOp(PN)→any #4 │

│[[GetOwnProperty]] SpecOp(PN)→Undefined或PD #5 │

│[[GetProperty]] SpecOp(PN)→Undefined或PD #6 │

│[[Put]] SpecOp(PN, any, Boolean) #7 │

│[[CanPut]] SpecOp(PN)→Boolean #8 │

│[[HasProperty]] SpecOp(PN)→Boolean #9 │

│[[Delete]] SpecOp(PN, Boolean)→Boolean #10 │

│[[DefaultValue]] SpecOp(Hint)→primitive #11 │

│[[DefineOwnProperty]] SpecOp(PN, PD, Boolean)→Boolean #12 │

└─────────────────────────────────────┘

#1 该对象的原型。

#2 一个String值，指示着规范定义的对象类别。

#3 若为**true**，自有属性可被添加到该对象上。

#4 返回具名属性的值。

#5 返回该对象的具名自有属性的属性描述符，或**undefined**若不存在。

#6 返回该对象的具名属性的完全装配的属性描述符，或**undefined**若不存在。

#7 将指定的具名属性设置为第二个形参值。标志控制失败处理。

#8 返回一个Boolean值指示着一个传入PN的[[Put]]操作能否被执行。

#9 返回一个Boolean值指示着该对象是否已经具有一个给定名字的属性。

#10 从该对象移除指定的具名自有属性。标志控制失败处理。

#11 Hint是一个String，为该对象返回一个默认值。

#12 创建或更改具名自有属性，使其具有PD所描述的状态。标志控制失败处理。

每个对象（包括宿主对象）必须实现罗列在表格8里的全部内部属性。然而，对于某些对象，[[DefaultValue]]内部方法可以只是抛出一个TypeError异常。

所有对象都有一个内部属性叫做[[Prototype]]，该属性的值要么是**null**要么是一个对象，该属性被用来实现继承。一个原生对象是否能够使用一个宿主对象作为它的[[Prototype]]依赖于实现。每个[[Prototype]]链长度必须有限（也就是说，从任何对象开始，递归地访问[[Prototype]]内部属性必须最终得到一个**null**值）。[[Prototype]]对象的具名数据属性被继承（以子对象的属性形式出现），是为了获取访问而不是设置访问。具名存取器属性被继承，既是为了获取访问也是为了设置访问。

每个ECMAScript对象有一个Boolean值的[[Extensible]]内部属性，控制具名属性是否可以被添加到该对象上。如果[[Extensible]]内部属性的值是**false**，那么附加的具名属性不能被添加到该对象上。此外，如果[[Extensible]]是**false**，则该对象的[[Class]]和[[Prototype]]内部属性的值不能再被修改。一旦一个[[Extensible]]内部属性的值已被设为**false**，就无法在后来改回**true**。

注意：这份规范没有定义ECMAScript语言运算符或内建函数，以允许一个程序修改一个对象的[[Class]]或[[Prototype]]内部属性，或把[[Extersible]]的值从**false**改回**true**。允许修改[[Class]]，[[Prototype]]或[[Extensible]]的实现特定的扩展，绝不能违反定义在上一段里的不变式。

【SP45】

这份规范为每一种内建对象定义了[[Class]]内部属性的值。宿主对象的[[Class]]内部属性的值可以是任何String值，除了“Arguments”，“Array”，“Boolean”，“Date”，“Error”，“Function”，“JSON”，“Math”，“Number”，“Object”，“RegExp”，和“String”中的任何一个。[[Class]]内部属性的值被内部用于辨别不同种类的对象。注意这份规范没有为一个程序提供任何方法来访问该属性值，除了凭借Object.prototype.toString（参阅15.2.4.2条）。

除非另附说明，原生ECMAScript对象的公共内部方法按照8.12节描述的那般运作。Array对象对于[[DefineOwnProperty]]内部方法（参阅15.4.5.1条）具有一个稍微不同的实现，String对象对于[[GetOwnProperty]]内部方法（参阅15.5.5.2条）具有一个稍微不同的实现。Arguments对象（10.6节）对于[[Get]]，[[GetOwnProperty]]，[[DefineOwnProperty]]，和[[Delete]]分别具有不同的实现。Function对象（15.3节）对于[[Get]]具有一个不同的实现。

宿主对象可以用任何方式实现这些内部方法，除非另附说明；例如，一种可能是用于一个特殊的宿主对象的[[Get]]和[[Put]]确实取回和存入属性值，但[[HasProperty]]总是产生**false**。然而，如果一个宿主对象的内部属性的任何规定的操纵不被一个实现支持，当尝试那一操纵时它必须抛出一个TypeError异常。

一个宿主对象的[[GetOwnProperty]]内部方法对于该宿主对象的每一属性必须遵从下面的不变式：

·如果一个属性被描述为一个数据属性，且它可能随时间变化而返回不同的值，那么[[Writable]]和/或[[Configurable]]特性必须是**true**，即使没有改变这些值的机制经由其它内部方法被暴露出来。

·如果一个属性被描述为一个数据属性，且它的[[Writable]]和[[Configurable]]都是**false**，那么对[[GetOwnProperty]]【并以该属性名称为实参】的所有调用【在返回的属性描述符里】必须为该属性的[[Value]]特性返回 SameValue（依照9.12节）。

·如果 [[Writable]] 以外的特性可以随时间而变，或如果该属性【指特性关联的宿主对象属性】可以消失，那么[[Configurable]]特性必须是**true**。

·如果 [[Writable]] 特性可以从**false**变成**true**，那么[[Configurable]]特性必须是**true**。

·如果宿主对象的[[Extensible]]内部属性的值已被ECMAScript代码观察到为**false**，那么如果对[[GetOwnProperty]]的一次调用把一个属性描述为不存在，则所有后续调用也必须把那个属性描述为不存在。

宿主对象的[[DefineOwnProperty]]内部方法绝不能允许添加一个新属性到一个宿主对象上，如果那个宿主对象的[[Extensible]]内部属性已被ECMAScript代码观察到为**false**。

如果那个宿主对象的[[Extensible]]内部属性已被ECMAScript代码观察到为**false**，那么它绝不能后来又变回**true**。

【SP46】

表格9 —— 只为某些对象定义的内部属性

AL：List of any，any列表【为方便排版】

┌─────────────────────────────────────┐

│内部属性 值类型定义域 描述 │

├─────────────────────────────────────┤

│[[PrimitiveValue]] primitive #1 │

│[[Construct]] SpecOp(AL)→Object #2 │

│[[Call]] SpecOp(any, AL)→any或Reference #3 │

│[[HasInstance]] SpecOp(any)→Boolean #4 │

│[[Scope]] Lexical Environment #5 │

│[[FormalParameters]] String列表 #6 │

│[[Code]] ECMAScript代码 #7 │

│[[TargetFunction]] Object #8 │

│[[BoundThis]] any #9 │

│[[BoundArguments]] AL #10 │

│[[Match]] SpecOp(String, index)→MatchResult #11 │

│[[ParameterMap]] Object #12 │

└─────────────────────────────────────┘

#1 和该对象关联的内部状态信息。对于标准内建ECMAScript对象，只有Boolean，Date，Number，和String对象实现了该属性。

#2 创建一个对象，经由**new**运算符引发【Invoke】。传给SpecOp的实参就是传给**new**运算符的实参。实现了这个内部方法的对象叫做构造函数。

#3 执行和该对象关联的代码，经由函数调用表达式引发。传给SpecOp的实参是**this**对象和一个列表，它包含了传给函数调用表达式的实参。实现了这个内部方法的对象是可调用的。只有可调用的宿主对象才可以返回Reference值。

#4 返回一个Boolean值指示着实参是否有可能是一个由该对象构造出来的对象。对于标准内建ECMAScript对象，只有Function对象实现了该属性。

#5 一个词法环境，定义了一个Function对象执行所在的环境。对于标准内建ECMAScript对象，只有Function对象实现了该属性。

#6 一个可能为空的列表，包含着一个Function的*FormalParameterList*的标识符字符串。对于标准内建ECMAScript对象，只有Function对象实现了该属性。

#7 一个函数的ECMAScript代码。对于标准内建ECMAScript对象，只有Function对象实现了该属性。

#8 一个使用标准内建Function.prototype.bind方法创建的Function对象的目标函数，只有使用这个方法创建的ECMAScript对象才有该属性。

#9 一个使用标准内建Function.prototype.bind方法创建的Function对象的预绑定的**this**值，只有使用这个方法创建的ECMAScript对象才有该属性。

#10 一个使用标准内建Function.prototype.bind方法创建的Function对象的预绑定的实参值，只有使用这个方法创建的ECMAScript对象才有该属性。

#11 测试一个正则表达式是否匹配，并返回一个MatchResult值（参阅15.10.2.1条）。对于标准内建ECMAScript对象，只有RegExp对象实现了该属性。

#12 在arguments对象（参阅10.6节）的属性和关联的函数的形参之间提供一个映射。只有arguments对象才有该属性。

【SP47】

8.7节 Reference规范类型

Reference类型被用于解释这样一些运算符的行为，如**delete**，**typeof**，和赋值运算符。例如，一个赋值的左运算数被期望产生一个引用。赋值的行为除了借助引用，还可以根据对赋值运算符的左运算数的语法形式进行案例分析来完全解释，要不是有一个困难：函数调用可以返回引用。允许这种可能性纯粹是为了宿主对象。没有这份规范定义的内建ECMAScript函数返回一个引用，对于用户定义函数也没有规定返回一个引用。（不使用语法案例分析的另一个原因是，它将会十分漫长且难以处理，影响着规范的许多地方。）

一个Reference是一个已解析的名称绑定，它是由三个分量构成的：基【base】值，被引用的名称【referenced name】，和严格引用【strict reference】（Boolean值的标志）。基值可以是**undefined**，一个Object，一个Boolean，一个String，一个Number，或一个环境记录（10.2.1段）。**undefined**基值表明该引用不能被解析成一个绑定。被引用的名称是一个String。

这份规范使用下面的抽象操作来访问引用的各个分量：

·GetBase(V)，返回引用V的基值分量。

·GetReferenceName(V)，返回引用V的被引用的名称分量。

·IsStrictReference(V)，返回引用V的严格引用分量。

·HasPrimitiveBase(V)，如果基值是一个Boolean，String，或Number，则返回**true**。

·IsPropertyReference(V)，如果基值是一个Object或HasPrimitiveBase(V)返回**true**，则返回**true**，否则返回**false**。

·IsUnresolvableReference(V)，如果基值是**undefined**，则返回**true**，否则返回**false**。

这份规范使用下面的抽象操作来操作引用：

8.7.1段 GetValue(V)

1. 如果Type(V)不是Reference，返回V。

2. 令base是调用GetBase(V)的结果。

3. 如果IsUnresolvableReference(V)为**true**，抛出一个ReferenceError异常。

4. 如果IsPropertyReference(V)为**true**，那么

a. 如果HasPrimitiveBase(V)为**false**，那么令get是base的[[Get]]内部方法，否 则令get是定义在下面的特殊的[[Get]]内部方法。

b. 以base作为**this**值，并传入GetReferencedName(V)作为实参，返回调用get内 部方法的结果。

5. 否则，base一定是一个环境记录。

a. 传入GetReferencedName(V)和IsStrictReference(V)作为实参，返回调用base 的GetBindingValue（参阅10.2.1段）具体方法的结果。

当V是一个带有基元基值的属性引用时，下面的[[Get]]内部方法由GetValue使用，以base作为它的**this**值，属性名称P作为它的实参进行调用。采取以下步骤：

1. 令O是ToObject(base)。

2. 以属性名称P作为实参，令desc是调用O的[[GetProperty]]内部方法的结果。

3. 如果desc是**undefined**，返回**undefined**。

4. 如果IsDataDescriptor(desc)为**true**，返回desc.[[Value]]。

5. 否则，IsAccessorDescriptor(desc)一定是**true**，所以，令getter是desc.[[Get]]（参阅8.10节）。

6. 如果getter是**undefined**，返回**undefined**。

7. 以base作为**this**值，不传任何实参，返回调用getter的[[Call]]内部方法的结果。

注意：可能在第1步里被创建的那个对象，在上面的方法之外是不可访问的。一个实现可以选择避免实际创建那个对象。一个用到该内部方法的实际属性访问可能具有可见效果的唯一情形就是当引发一个存取器函数时。

【SP48】

8.7.2段 PutValue(V, W)

1. 如果Type(V)不是Reference，抛出一个ReferenceError异常。

2. 令base是调用GetBase(V)的结果。

3. 如果IsUnresolvableReference(V)为**true**，那么

a. 如果IsStrictReference(V)为**true**，那么

i. 抛出一个ReferenceError异常。

b. 为属性名称传入GetReferencedName(V)，为值传入W，为Throw标志传入**false**， 调用Global对象的[[Put]]内部方法。

4. 否则如果IsPropertyReference(V)为**true**，那么

a. 如果HasPrimitiveBase(V)为**false**，那么令put是base的[[Put]]内部方法，否 则令put是定义在下面的特殊的[[Put]]内部方法。

b. 以base作为**this**值，并为属性名称传入GetReferencedName(V)，为值传入W，为 Throw标志传入IsStrictReference(V)，调用put内部方法。

5. 否则base一定是一个基值为环境记录的引用，所以，

a. 传入GetReferencedName(V)，W，和IsStrictReference(V)作为实参，调用base 的SetMutableBinding（10.2.1段）具体方法。

6. 返回。

当V是一个带有基元基值的属性引用时，下面的[[Put]]内部方法由PutValue使用，以base作为它的**this**值，属性名称P，值W，和Boolean标志Throw作为它的实参进行调用。采取以下步骤：

1. 令O是ToObject(base)。

2. 传入P作为实参，如果调用O的[[CanPut]]内部方法的结果为**false**，那么

a. 如果Throw为**true**，那么抛出一个TypeError异常。

b. 否则返回。

3. 传入P作为实参，令ownDesc是调用O的[[GetOwnProperty]]内部方法的结果。

4. 如果IsDataDescriptor(ownDesc)为**true**，那么

a. 如果Throw为**true**，那么抛出一个TypeError异常。

b. 否则返回。

5. 传入P作为实参，令desc是调用O的[[GetProperty]]内部方法的结果，这要么是一个自有或继承的存取器属性描述符，要么是一个继承的数据属性描述符。

6. 如果IsAccessorDescriptor(desc)为**true**，那么

a. 令setter是desc.[[Set]]（参阅8.10节），它不可能是**undefined**。

b. 以base作为**this**值，传入W作为唯一实参，调用setter的[[Call]]内部方法。

7. 否则，这是一个在临时【transient】对象O上创建一个自有属性的请求，

a. 如果Throw为**true**，那么抛出一个TypeError异常。

8. 返回。

注意：可能在第1步里被创建的那个对象，在上面的方法之外是不可访问的。一个实现可以选择避免实际创建那个临时对象。一个用到该内部方法的实际属性赋值可能具有可见效果的唯一情形就是当引发一个存取器函数，或违反了以Throw判定的错误检查时。当Throw为**true**时，任何将要在临时对象上创建一个新属性的属性赋值都会抛出一个错误。

8.8节 List规范类型

List类型被用来解释实参列表（参阅11.2.4段）的求值，实参列表可以在**new**表达式里，函数调用里，和其它需要一个简单值列表的算法里。List类型的值仅仅是值的有序序列，这些序列可以具有任意长度。

8.9节 Completion规范类型

Completion类型被用来解释一些语句的行为，那些语句执行控制的非局部转移（**break**，**continue**，**return**，和**throw**）。Completion类型的值是三元组形式(type, value, target)，其中type是**normal**，**break**，**continue**，**return**，或**throw**中的一个；value是任何ECMAScript语言值或**empty**；target是任何ECMAScript标识符或**empty**。如果cv是一个Completion值，那么cv.type，cv.value，和cv.target可被用来直接引用它的构成值。

【SP49】

术语“abrupt completion”表示任何type不是**normal**的Completion值。

8.10节 Property Descriptor和Property Identifier规范类型

Property Descriptor类型被用来解释具名属性特性的操纵和物化。Property Descriptor类型的值是一个由具名字段构成的记录，其中每个字段的名称是一个特性名，值是一个按8.6.1段规定的对应特性值。此外，任何字段可以存在也可以缺失。

Property Descriptor值可以被进一步分类成数据属性描述符和存取器属性描述符，根据某些字段的存在性或对某些字段的使用。一个数据属性描述符就是一个含有任何名为[[Value]]或[[Writable]]的字段的属性描述符。一个存取器属性描述符就是一个含有任何名为[[Get]]或[[Set]]的字段的属性描述符。任何属性描述符可以具有名为[[Enumerable]]和[[Configurable]]的字段。一个Property Descriptor值不能既是一个数据属性描述符，又是一个存取器属性描述符；然而，它可以两者都不是。一个泛型【generic】属性描述符是一个既不是数据属性描述符、也不是存取器属性描述符的Property Descriptor值。一个完全装配的【fully populated】属性描述符是这样一个属性描述符：一个存取器属性描述符或数据属性描述符，并具有全部——对应到定义在8.6.1段表格5或表格6里的属性特性的——字段。

为了在这份规范里便于表示，一个类似于对象字面值表示法的语法可被用于定义一个Property Descriptor值。例如，Property Descriptor { [[Value]]:42, [[Writable]]:**false**, [[Configurable]]:**true** }定义了一个数据属性描述符。字段名顺序是不重要的。任何没有显式列出的字段被认为是缺失的。

在这份规范的原文和算法里，点记号可被用于引用一个Property Descriptor的特定字段。例如，如果D是一个属性描述符，那么D.[[Value]]就是“D的名为[[Value]]的字段”的简写。

Property Identifier类型被用来把一个属性名称和一个Property Descriptor相关联。Property Identifier类型的值是二元组形式(name, descriptor)，其中name是一个String，descriptor是一个Property Descriptor值。

在这份规范里使用下面的抽象操作来操作Property Descriptor值：

8.10.1段 IsAccessorDescriptor(Desc)

传入属性描述符Desc作为实参，当调用抽象操作IsAccessorDescriptor时，采取以下步骤：

1. 如果Desc是**undefined**，那么返回**false**。

2. 如果Desc.[[Get]]和Desc.[[Set]]都缺失，那么返回**false**。

3. 返回**true**。

8.10.2段 IsDataDescriptor(Desc)

传入属性描述符Desc作为实参，当调用抽象操作IsDataDescriptor时，采取以下步骤：

1. 如果Desc是**undefined**，那么返回**false**。

2. 如果Desc.[[Value]]和Desc.[[Writable]]都缺失，那么返回**false**。

3. 返回**true**。

8.10.3段 IsGenericDescriptor(Desc)

传入属性描述符Desc作为实参，当调用抽象操作IsGenericDescriptor时，采取以下步骤：

1. 如果Desc是**undefined**，那么返回**false**。

2. 如果IsAccessorDescriptor(Desc)和IsDataDescriptor(Desc)都为**false**，那么返回**true**。

【SP50】

3. 返回**false**。

8.10.4段 FromPropertyDescriptor(Desc)

传入属性描述符Desc作为实参，当调用抽象操作FromPropertyDescriptor时，采取以下步骤：

以下算法假定Desc是一个完全装配的Property Descriptor，例如从[[GetOwnProperty]]（参阅8.12.1段）返回的。

1. 如果Desc是**undefined**，那么返回**undefined**。

2. 令obj是如同利用表达式**new** Object()创建一个新对象的结果，其中Object是标准内建构造函数。

3. 如果IsDataDescriptor(Desc)为**true**，那么

a. 传入"value"，Property Descriptor { [[Value]]:Desc.[[Value]],[[Writable]]:**true**, [[Enumerable]]:**true**, [[Configurable]]:**true** }，和**false** 作为实参， 调用obj的 [[DefineOwnProperty]]内部方法。

b. 传入"writable"，Property Descriptor { [[Value]]:Desc.[[ Writable]],[[Writable]]:**true**, [[Enumerable]]:**true**, [[Configurable]]:**true** }，和**false** 作为实参， 调用obj的 [[DefineOwnProperty]]内部方法。

4. 否则，IsAccessorDescriptor(Desc)一定为**true**，所以

a. 传入"get"，Property Descriptor { [[Value]]:Desc.[[Get]],[[Writable]]:**true**, [[Enumerable]]:**true**, [[Configurable]]:**true** }，和**false** 作为实参， 调用obj的 [[DefineOwnProperty]]内部方法。

b. 传入"set"，Property Descriptor { [[Value]]:Desc.[[Set]],[[Writable]]:**true**, [[Enumerable]]:**true**, [[Configurable]]:**true** }，和**false** 作为实参， 调用obj的 [[DefineOwnProperty]]内部方法。

5. 传入"enumerable"，Property Descriptor { [[Value]]:Desc.[[ Enumerable]],[[Writable]]:**true**, [[Enumerable]]:**true**, [[Configurable]]:**true** }，和**false** 作为实参， 调用obj的[[DefineOwnProperty]]内部方法。

6. 传入"configurable"，Property Descriptor { [[Value]]:Desc.[[ Configurable]],[[Writable]]:**true**, [[Enumerable]]:**true**, [[Configurable]]:**true** }，和**false** 作为实参， 调用obj的[[DefineOwnProperty]]内部方法。

7. 返回obj。

8.10.5段 ToPropertyDescriptor(Obj)

传入对象Obj作为实参，当调用抽象操作ToPropertyDescriptor时，采取以下步骤：

1. 如果Type(Obj)不是Object，则抛出一个TypeError异常。

2. 令desc是创建一个初始不含任何字段的新Property Descriptor的结果。

3. 传入"enumerable"作为实参，如果调用Obj的[[HasProperty]]内部方法的结果为**true**，那么

a. 传入"enumerable"作为实参，令enum是调用Obj的[[Get]]内部方法的结果。

b. 把desc的[[Enumerable]]字段设为ToBoolean(enum)。

4. 传入"configurable"作为实参，如果调用Obj的[[HasProperty]]内部方法的结果为**true**，那么

a. 传入"configurable"作为实参，令conf是调用Obj的[[Get]]内部方法的结果。

b. 把desc的[[Configurable]]字段设为ToBoolean(conf)。

5. 传入"value"作为实参，如果调用Obj的[[HasProperty]]内部方法的结果为**true**，那么

a. 传入"value"作为实参，令value是调用Obj的[[Get]]内部方法的结果。

b. 把desc的[[Value]]字段设为value。

6. 传入"writable"作为实参，如果调用Obj的[[HasProperty]]内部方法的结果为**true**，那么

a. 传入"writable"作为实参，令writable是调用Obj的[[Get]]内部方法的结果。

b. 把desc的[[Writable]]字段设为ToBoolean(writable)。

7. 传入"get"作为实参，如果调用Obj的[[HasProperty]]内部方法的结果为**true**，那么

a. 传入"get"作为实参，令getter是调用Obj的[[Get]]内部方法的结果。

b. 如果IsCallable(getter)为**false**，且getter不是**undefined**，那么抛出一个 TypeError异常。

c. 把desc的[[Get]]字段设为getter。

8. 传入"set"作为实参，如果调用Obj的[[HasProperty]]内部方法的结果为**true**，那么

a. 传入"set"作为实参，令setter是调用Obj的[[Get]]内部方法的结果。

b. 如果IsCallable(setter)为**false**，且setter不是**undefined**，那么抛出一个 TypeError异常。

c. 把desc的[[Set]]字段设为setter。

9. 如果desc.[[Get]]或desc.[[Set]]存在，那么

a. 如果desc.[[Value]]或desc.[[Writable]]存在，那么抛出一个TypeError异常。

【SP51】

10. 返回desc。

8.11节 Lexical Environment和Environment Record规范类型

Lexical Environment和Environment Record类型被用来解释在嵌套的函数和代码块里名称解析的行为。这些类型和它们之上的操作在第10章定义。

8.12节 用于Object内部方法的算法

在下面的算法描述里，假定O是一个原生的ECMAScript对象，P是一个String，Desc是一个Property Descriptor记录，Throw是一个Boolean标志。

8.12.1段 [[GetOwnProperty]](P)

传入属性名称P作为实参，当调用O的[[GetOwnProperty]]内部方法时，采取以下步骤：

1. 如果O没有名为P的自有属性，返回**undefined**。

2. 令D是一个新创建的Property Descriptor，没有字段。

3. 令X是O的名为P的自有属性。

4. 如果X是一个数据属性，那么

a. 把D.[[Value]]设为X的[[Value]]特性的值。

b. 把D.[[Writable]]设为X的[[Writable]]特性的值。

5. 否则，X是一个存取器属性，所以

a. 把D.[[Get]]设为X的[[Get]]特性的值。

b. 把D.[[Set]]设为X的[[Set]]特性的值。

6. 把D.[[Enumerable]]设为X的[[Enumerable]]特性的值。

7. 把D.[[Configurable]]设为X的[[Configurable]]特性的值。

8. 返回D。

然而，如果O是一个String对象，它有一个更加复杂的[[GetOwnProperty]]内部方法，定义在15.5.5.2条。

8.12.2段 [[GetProperty]](P)

传入属性名称P作为实参，当调用O的[[GetProperty]]内部方法时，采取以下步骤：

1. 传入P作为实参，令prop是调用O的[[GetOwnProperty]]内部方法的结果。

2. 如果prop不是**undefined**，返回prop。

3. 令proto是O的[[Prototype]]内部属性的值。

4. 如果proto是**null**，返回**undefined**。

5. 传入P作为实参，返回调用proto的[[GetProperty]]内部方法的结果。

8.12.3段 [[Get]](P)

传入属性名称P作为实参，当调用O的[[Get]]内部方法时，采取以下步骤：

1. 传入P作为实参，令desc是调用O的[[GetProperty]]内部方法的结果。

2. 如果desc是**undefined**，返回**undefined**。

3. 如果IsDataDescriptor(desc)为**true**，返回desc.[[Value]]。

4. 否则，IsAccessorDescriptor(desc)一定是**true**，所以，令getter是desc.[[Get]]。

5. 如果getter是**undefined**，返回**undefined**。

6. 以O作为**this**值，不传任何实参，返回调用getter的[[Call]]内部方法的结果。

8.12.4段 [[CanPut]](P)

传入属性名称P作为实参，当调用O的[[CanPut]]内部方法时，采取以下步骤：

1. 传入P作为实参，令desc是调用O的[[GetOwnProperty]]内部方法的结果。

【SP52】

2. 如果desc不是**undefined**，那么

a. 如果IsAccessorDescriptor(desc)为**true**，那么

i. 如果desc.[[Set]]是**undefined**，那么返回**false**。

ii. 否则返回**true**。

b. 否则，desc一定是一个数据描述符，所以，返回desc.[[Writable]]的值。

3. 令proto是O的[[Prototype]]内部属性的值。

4. 如果proto是**null**，那么返回O的[[Extensible]]内部属性的值。

5. 传入P作为实参，令inherited是调用proto的[[GetProperty]]内部方法的结果。

6. 如果inherited为**undefined**，那么返回O的[[Extensible]]内部属性的值。

7. 如果IsAccessorDescriptor(inherited)为**true**，那么

a. 如果inherited.[[Set]]为**undefined**，那么返回**false**。

b. 否则返回**true**。

8. 否则，inherited一定是一个数据描述符。

a. 如果O的[[Extensible]]内部属性的值为**false**，那么返回**false**。

b. 否则返回inherited.[[Writable]]的值。

宿主对象可以在[[Put]]操作上定义附加的约束。如果可能，在[CanPut]]的这一定义返回**false**的情况下，宿主对象不应该允许[[Put]]操作。

8.12.5段 [[Put]](P, V, Throw)

传入属性名称P，值V，和Boolean标志Throw作为实参，当调用O的[[Put]]内部方法时，采取以下步骤：

1. 传入P作为实参，如果调用O的[[CanPut]]内部方法的结果为**false**，那么

a. 如果Throw为**true**，那么抛出一个TypeError异常。

b. 否则返回。

2. 传入P作为实参，令ownDesc是调用O的[[GetOwnProperty]]内部方法的结果。

3. 如果IsDataDescriptor(ownDesc)为**true**，那么

a. 令valueDesc是Property Descriptor { [[Value]]:V }。

b. 传入P，valueDesc，和Throw作为实参，调用O的[[DefineOwnProperty]]内部方 法。

c. 返回。

4. 传入P作为实参，令desc是调用O的[[GetProperty]]内部方法的结果，这要么是一个自有或继承的存取器属性描述符，要么是一个继承的数据属性描述符。

5. 如果IsAccessorDescriptor(desc)为**true**，那么

a. 令setter是desc.[[Set]]，它不可能是**undefined**。

b. 以O作为**this**值，并传入V作为唯一实参，调用setter的[[Call]]内部方法。

6. 否则，按如下步骤在对象O上创建一个名为P的具名数据属性。

a. 令newDesc是Property Descriptor { [[Value]]:V, [[Writable]]:**true**, [[Enumerable]]:**true**, [[Configurable]]:**true** }。

b. 传入P，newDesc，和Throw作为实参，调用O的[[DefineOwnProperty]]内部方法。

7. 返回。

8.12.6段 [[HasProperty]](P)

传入属性名称P作为实参，当调用O的[[HasProperty]]内部方法时，采取以下步骤：

1. 传入P作为实参，令desc是调用O的[[GetProperty]]内部方法的结果。

2. 如果desc为**undefined**，那么返回**false**。

3. 否则返回**true**。

8.12.7段 [[Delete]](P, Throw)

传入属性名称P，和Boolean标志Throw作为实参，当调用O的[[Delete]]内部方法时，采取以下步骤：

1. 传入P作为实参，令desc是调用O的[[GetOwnProperty]]内部方法的结果。

2. 如果desc为**undefined**，那么返回**true**。

【SP53】

3. 如果desc.[[Configurable]]为**true**，那么

a. 从O移除名为P的自有属性。

b. 返回**true**。

4. 否则，如果Throw为**ture**，那么抛出一个TypeError异常。

5. 返回**false**。

8.12.8段 [[DefaultValue]](hint)

为hint指定String作为实参，当调用O的[[DefaultValue]]内部方法时，采取以下步骤：

1. 传入"toString"作为实参，令toString是调用O的[[Get]]内部方法的结果。

2. 如果IsCallable(toString)为**true**，那么，

a. 以O作为**this**值，不传任何实参，令str是调用toString的[[Call]]内部方法的 结果。

b. 如果str是一个基元值，返回str。

3. 传入"valueOf"作为实参，令valueOf是调用O的[[Get]]内部方法的结果。

4. 如果IsCallable(valueOf)为**true**，那么，

a. 以O作为**this**值，不传任何实参，令val是调用valueOf的[[Call]]内部方法的 结果。

b. 如果val是一个基元值，返回val。

5. 抛出一个TypeError异常。

为hint指定Number作为实参，当调用O的[[DefaultValue]]内部方法时，采取以下步骤：

1. 传入"valueOf"作为实参，令valueOf是调用O的[[Get]]内部方法的结果。

2. 如果IsCallable(valueOf)为**true**，那么，

a. 以O作为**this**值，不传任何实参，令val是调用valueOf的[[Call]]内部方法的 结果。

b. 如果val是一个基元值，返回val。

3. 传入"toString"作为实参，令toString是调用O的[[Get]]内部方法的结果。

4. 如果IsCallable(toString)为**true**，那么，

a. 以O作为**this**值，不传任何实参，令str是调用toString的[[Call]]内部方法的 结果。

b. 如果str是一个基元值，返回str。

5. 抛出一个TypeError异常。

不传hint实参，当调用O的[[DefaultValue]]内部方法时，其如同hint是Number般运作，除非O是一个Date对象（参阅15.9.6段），此时其如同hint是String般运作。

上述用于原生对象的[[DefaultValue]]规范只能返回基元值。如果一个宿主对象实现了它自己的[[DefaultValue]]内部方法，它必须确保它的[[DefaultValue]]内部方法只能返回基元值。

8.12.9段 [[DefineOwnProperty]](P, Desc, Throw)

在下面的算法里，术语“Reject”意思是“如果Throw为**true**，那么抛出一个TypeError异常，否则返回**false**”。算法包含了一些测试Property Descriptor值Desc的各个字段是否取特定值的步骤，以这种方式测试的字段不需要实际存在于Desc中。如果一个字段缺失，那么它的值被认为是**false**。

传入属性名称P，Property Descriptor值Desc，和Boolean标志Throw作为实参，当调用O的[[DefineOwnProperty]]内部方法时，采取以下步骤：

1. 传入P作为实参，令current是调用O的[[GetOwnProperty]]内部方法的结果。

2. 令extensible是O的[[Extensible]]内部属性的值。

3. 如果current是**undefined**，且extensible为**false**，那么Reject。

4. 如果current是**undefined**，且extensible为**true**，那么

a. 如果IsGenericDescriptor(Desc)或IsDataDescriptor(Desc)为**true**，那么

i. 创建一个属于对象O的名为P的自有数据属性，它的[[Value]]，[[Writable]]， [[Enumerable]]，和[[Configurable]]特性值由Desc描述，

【SP54】

如果Desc的一个特性字段值缺失，把新创建的属性的相应特性设为它的默认值。

b. 否则，Desc一定是一个存取器属性描述符，所以，

i. 创建一个属于对象O的名为P的自有存取器属性，它的[[Get]]，[[Set]]， [[Enumerable]]，和[[Configurable]]特性值由Desc描述，如果Desc的一个特性 字段值缺失，把新创建的属性的相应特性设为它的默认值。

c. 返回**true**。

5. 如果Desc里的每个字段都缺失，返回**true**。

6. 如果Desc里的每个字段也存在于current里，且当使用SameValue算法（9.12节）比较时Desc里的每个字段的值和current里的对应字段的值相同，返回**true**。

7. 如果current的[[Configurable]]字段为**false**，那么

a. 如果Desc的[[Configurable]]字段为**true**，Reject。

b. 如果Desc的[[Enumerable]]字段存在，且current的[[Enumerable]]字段和Desc 的恰好是彼此的布尔相反值【一个为**true**，另一个为**false**】，Reject。

8. 如果IsGenericDescriptor(Desc)为**true**，那么不需要进一步验证。

9. 否则，如果IsDataDescriptor(current)和IsDataDescriptor(Desc)具有不同的结果，那么

a. 如果current的[[Configurable]]字段为**false**，Reject。

b. 如果IsDataDescriptor(current)为**true**，那么

i. 把O的名为P的属性从一个数据属性转换成一个存取器属性，保持被转换属性 的[[Configurable]]和[[Enumerable]]特性的现有值，并把属性的其余特性设 为它们的默认值。

c. 否则，

i. 把O的名为P的属性从一个存取器属性转换成一个数据属性，保持被转换属性 的[[Configurable]]和[[Enumerable]]特性的现有值，并把属性的其余特性设 为它们的默认值。

10. 否则，如果IsDataDescriptor(current)和IsDataDescriptor(Desc)都为**true**，那么

a. 如果current的[[Configurable]]字段为**false**，那么

i. 如果current的[[Writable]]字段为**false**，且Desc的[[Writable]]字段为 **true**，Reject。

ii. 如果current的[[Writable]]字段为**false**，那么

1. 如果Desc的[[Value]]字段存在，且SameValue(Desc.[[Value]], current.[[Value]])为**false**，Reject。

b. 否则，current的[[Configurable]]字段为**true**，所以任何改变都是可接受的。

11. 否则，IsAccessorDescriptor(current)和IsAccessorDescriptor(Desc)都为**true**，所以

a. 如果current的[[Configurable]]字段为**false**，那么

i. 如果Desc的[[Set]]字段存在，且SameValue(Desc.[[Set]], current.[[Set]])为**false**，Reject。

ii. 如果Desc的[[Get]]字段存在，且SameValue(Desc.[[Get]], current.[[Get]])为**false**，Reject。

12. 对于Desc的每个存在的特性字段，把O的名为P的属性的相应名字的特性设为该字段的值。

13. 返回**true**。

然而，如果O是一个Array对象，它有一个更为复杂的[[DefineOwnProperty]]内部方法，定义在15.4.5.1条里。

注意：步骤10.b允许Desc的任何字段不同于current的对应字段，只要current的[[Configurable]]字段为**true**。这甚至允许改变一个[[Writable]]特性为**false**的属性的[[Value]]。之所以允许是因为一个值为**true**的[[Configurable]]特性将会允许这样一个等价的调用序列，即[[Writable]]首先被设为**true**，然后一个新的[[Value]]被设置，最后再把[[Writable]]设为**false**。

第9章 类型转换和测试

ECMAScript运行时系统按需执行自动类型转换。为了澄清某些构造的语义，定义一组转换抽象操作是有用的。这些抽象操作不是语言的一部分；在这里定义它们是为了辅助语言语义的说明。转换抽象操作是多态的，也就是说，它们可以接受任何ECMAScript语言类型的值，但不接受规范类型的值。

【SP55】

9.1节 ToPrimitive

抽象操作ToPrimitive接受一个实参input和一个可选的实参PreferredType。抽象操作ToPrimitive把input转换成一个非Object类型。如果一个对象能够转换成多种基元类型，它可以使用可选的PreferredType提示首选类型。转换依照表格10进行：

表格10 —— ToPrimitive转换

┌─────────────────────────────────────┐

│input类型 结果 │

├─────────────────────────────────────┤

│Undefined 结果等于input实参（没有转换）。 │

│Null 结果等于input实参（没有转换）。 │

│Boolean 结果等于input实参（没有转换）。 │

│Number 结果等于input实参（没有转换）。 │

│String 结果等于input实参（没有转换）。 │

│Object #1 │

└─────────────────────────────────────┘

#1 为那个对象返回默认值。一个对象的默认值通过调用那个对象的[[DefaultValue]]内部方法获得，传入可选提示PreferredType作为实参。这份规范已在8.12.8段为所有原生ECMAScript对象定义了[[DefaultValue]]内部方法的行为。

9.2节 ToBoolean

抽象操作ToBoolean依照表格11把input转换成一个Boolean类型的值：

表格11 —— ToBoolean转换

┌─────────────────────────────────────┐

│input类型 结果 │

├─────────────────────────────────────┤

│Undefined **false**。 │

│Null **false**。 │

│Boolean 结果等于input实参（没有转换）。 │

│Number #1 │

│String #2 │

│Object **true** │

└─────────────────────────────────────┘

#1 如果input是**+0**，**-0**，或**NaN**，结果为**false**；否则为**true**。

#2 如果input是空String（长度为0），结果为**false**；否则为**true**。

9.3节 ToNumber

抽象操作ToNumber依照表格12把input转换成一个Number类型的值：

【SP56】

表格12 —— ToNumber转换

┌─────────────────────────────────────┐

│input类型 结果 │

├─────────────────────────────────────┤

│Undefined **NaN**。 │

│Null **+0**。 │

│Boolean #1 │

│Number 结果等于input实参（没有转换）。 │

│String 参阅下面的文法和注意。 │

│Object #2 │

└─────────────────────────────────────┘

#1 如果input为**true**，结果是**1**；如果input为**false**，结果是**+0**。

#2 应用下面的步骤：

1. 令primValue是ToPrimitive(input, Number)。【提示首选类型是Number】

2. 返回ToNumber(primValue)。

9.3.1段 应用到String类型的ToNumber

应用到String类型的ToNumber将下面的文法应用到String值input，如果文法不能把input解释成一个*StringNumericLiteral*的展开式，那么ToNumber的结果是**NaN**。

语法

*StringNumericLiteral* :::

*StrWhiteSpace* opt

*StrWhiteSpace* opt *StrNumericLiteral StrWhiteSpace* opt

*StrWhiteSpace* :::

*StrWhiteSpaceChar StrWhiteSpace* opt

*StrWhiteSpaceChar* :::

*WhiteSpace*

*LineTerminator*

*StrNumericLiteral* :::

*StrDecimalLiteral*

*HexIntegerLiteral*

*StrDecimalLiteral* :::

*StrUnsignedDecimalLiteral*

**+** *StrUnsignedDecimalLiteral*

**-** *StrUnsignedDecimalLiteral*

*StrUnsignedDecimalLiteral* :::

**Infinity**

*DecimalDigits* **.** *DecimalDigits* opt *ExponentPart* opt

**.** *DecimalDigits ExponentPart* opt

*DecimalDigits ExponentPart* opt

*DecimalDigits* :::

*DecimalDigit*

*DecimalDigits DecimalDigit*

*DecimalDigit* ::: **其中一个**

**0 1 2 3 4 5 6 7 8 9**

*ExponentPart* :::

*ExponentIndicator SignedInteger*

【SP57】

*ExponentIndicator* ::: **其中一个**

**e E**

*SignedInteger* :::

*DecimalDigits*

**+** *DecimalDigits*

**-** *DecimalDigits*

*HexIntegerLiteral* :::

**0x** *HexDigit*

**0X** *HexDigit*

*HexIntegerLiteral HexDigit*

*HexDigit* ::: **其中一个**

**0 1 2 3 4 5 6 7 8 9**

**a b c d e f**

**A B C D E F**

应该注意到在*StringNumericLiteral*和*NumericLiteral*（参阅7.8.3段）之间的一些差异：

·一个*StringNumericLiteral*可以前带和/或后跟空白和/或行终止符。

·一个为十进制数的*StringNumericLiteral*可以具有任意数量的前导数位**0**。

·一个为十进制数的*StringNumericLiteral*可以前带**+**或**-**以指定它的符号。

·一个为空或只包含空白的*StringNumericLiteral*被转换成**+0**。

String到Number的转换总体上类似于为一个数字字面值确定Number值（参阅7.8.3段），但一些细节有所不同，所以把一个String数字字面值转换成一个Number值的过程在这里完整给出。这个值分两步确定：首先，一个数学值（MV）从String数字字面值推导而来；然后，这个数学值按下面的描述进行舍入：

·*StringNumericLiteral* ::: [empty]的MV就是**0**。

·*StringNumericLiteral* ::: *StrWhiteSpace*的MV就是**0**。

·*StringNumericLiteral* ::: *StrWhiteSpace* opt *StrNumericLiteral StrWhiteSpace* opt的MV就是*StrNumericLiteral*的MV，无论空白是否存在。

·*StrNumericLiteral* ::: *StrDecimalLiteral*的MV就是*StrDecimalLiteral*的MV。

·*StrNumericLiteral* ::: *HexIntegerLiteral*的MV就是*HexIntegerLiteral*的MV。

·*StrDecimalLiteral* ::: *StrUnsignedDecimalLiteral*的MV就是*StrUnsignedDecimalLiteral*的MV。

·*StrDecimalLiteral* ::: **+** *StrUnsignedDecimalLiteral*的MV就是*StrUnsignedDecimalLiteral*的MV。

·*StrDecimalLiteral* ::: **-** *StrUnsignedDecimalLiteral*的MV就是*StrUnsignedDecimalLiteral*的MV的相反数。（注意如果*StrUnsignedDecimalLiteral*的MV是**0**，其相反数还是**0**。下面描述的舍入规则把这种无符号数学零根据情况适当地转换成一个浮点**+0**或**-0**。）

·*StrUnsignedDecimalLiteral* ::: **Infinity**的MV就是10^10000（如此大的一个值将被舍入到**+∞**）。

·*StrUnsignedDecimalLiteral* ::: *DecimalDigits* **.** 的MV就是*DecimalDigits*的MV。

·*StrUnsignedDecimalLiteral* ::: *DecimalDigits* **.** *DecimalDigits*的MV就是第一个*DecimalDigits*的MV加上(第二个*DecimalDigits*的MV乘以10^-n)，其中n是第二个*DecimalDigits*中的字符数。

·*StrUnsignedDecimalLiteral* ::: *DecimalDigits* **.** *ExponentPart* 的MV就是*DecimalDigits*的MV乘以10^e，其中e是*ExponentPart*的MV。

·*StrUnsignedDecimalLiteral* ::: *DecimalDigits* **.** *DecimalDigits ExponentPart*的MV就是(第一个*DecimalDigits*的MV加上(第二个*DecimalDigits*的MV乘以10^-n)) 乘以10^e，其中n是第二个*DecimalDigits*中的字符数，e是*ExponentPart*的MV。

·*StrUnsignedDecimalLiteral* :::**.** *DecimalDigits*的MV就是*DecimalDigits*的MV乘以10^-n，其中n是*DecimalDigits*中的字符数。

·*StrUnsignedDecimalLiteral* :::**.** *DecimalDigits ExponentPart* 的MV就是*DecimalDigits*的MV乘以10^(e－n)，其中n是*DecimalDigits*中的字符数，e是*ExponentPart*的MV。

·*StrUnsignedDecimalLiteral* ::: *DecimalDigits*的MV就是*DecimalDigits*的MV。

【SP58】

·*StrUnsignedDecimalLiteral* ::: *DecimalDigits ExponentPart*的MV就是*DecimalDigits*的MV乘以10^e，其中e是*ExponentPart*的MV。

·*DecimalDigits* ::: *DecimalDigit*的MV就是*DecimalDigit*的MV。

·*DecimalDigits* ::: *DecimalDigits DecimalDigit*的MV就是(*DecimalDigits*的MV乘以10)加上*DecimalDigit*的MV。

·*ExponentPart* ::: *ExponentIndicator SignedInteger*的MV就是*SignedInteger*的MV。

·*SignedInteger* ::: *DecimalDigits*的MV就是*DecimalDigits*的MV。

·*SignedInteger* ::: **+** *DecimalDigits*的MV就是*DecimalDigits*的MV。

·*SignedInteger* ::: **-** *DecimalDigits*的MV就是*DecimalDigits*的MV的相反数。

·*DecimalDigit* ::: **0**或*HexDigit* ::: **0**的MV就是0。

·*DecimalDigit* ::: **1**或*HexDigit* ::: **1**的MV就是1。

·*DecimalDigit* ::: **2**或*HexDigit* ::: **2**的MV就是2。

·*DecimalDigit* ::: **3**或*HexDigit* ::: **3**的MV就是3。

·*DecimalDigit* ::: **4**或*HexDigit* ::: **4**的MV就是4。

·*DecimalDigit* ::: **5**或*HexDigit* ::: **5**的MV就是5。

·*DecimalDigit* ::: **6**或*HexDigit* ::: **6**的MV就是6。

·*DecimalDigit* ::: **7**或*HexDigit* ::: **7**的MV就是7。

·*DecimalDigit* ::: **8**或*HexDigit* ::: **8**的MV就是8。

·*DecimalDigit* ::: **9**或*HexDigit* ::: **9**的MV就是9。

·*HexDigit* ::: **a**或*HexDigit* ::: **A**的MV就是10。

·*HexDigit* ::: **b**或*HexDigit* ::: **B**的MV就是11。

·*HexDigit* ::: **c**或*HexDigit* ::: **C**的MV就是12。

·*HexDigit* ::: **d**或*HexDigit* ::: **D**的MV就是13。

·*HexDigit* ::: **e**或*HexDigit* ::: **E**的MV就是14。

·*HexDigit* ::: **f**或*HexDigit* ::: **F**的MV就是15。

·*HexIntegerLiteral* ::: **0x** *HexDigit*的MV就是*HexDigit*的MV。

·*HexIntegerLiteral* ::: **0X** *HexDigit*的MV就是*HexDigit*的MV。

·*HexIntegerLiteral* ::: *HexIntegerLiteral HexDigit*的MV就是(*HexIntegerLiteral*的MV乘以16)加上*HexDigit*的MV。

一个String数字字面值准确的MV一经被确定，它就被舍入成一个Number类型的值。如果MV是0，则舍入值是**+0**，除非String数字字面值里的第一个非空白字符是“**-**”，此时舍入值是**-0**；否则，舍入值一定是MV的Number值（依8.5节规定），除非字面值包含一个*StrUnsignedDecimalLiteral*且字面值具有多于20个有效数位，此时Number值可以是——使用数位0替换第20位之后的每个有效数位产生的——字面值的MV的Number值，或是——使用数位0替换第20位之后的每个有效数位，然后把位于第20个有效数位上的字面值加1产生的——字面值的MV的Number值。一个数位是有效的，如果它不是*ExponentPart*的一部分，且

·它不是**0**；或

·既有一个非零数位在它的左边，又有一个非零数位在它的右边且不在*ExponentPart*里。

9.4节 ToInteger

抽象操作ToInteger依照如下方式把input转换成一个整数值：

1. 令number是调用ToNumber(input)的结果。

2. 如果number是**NaN**，返回**+0**。

3. 如果number是**+0**，**-0**，**+∞**，或**-∞**，返回number。

4. 返回计算sign(number)×floor(abs(number))的结果。

【SP59】

9.5节 ToInt32：（有符号32位整数）

抽象操作ToInt32依照如下方式把input转换成位于闭区间[-2^31, 2^31－1]里的2^32个整数值之一：

1. 令number是调用ToNumber(input)的结果。

2. 如果number是**NaN**，**+0**，**-0**，**+∞**，或**-∞**，返回**+0**。

3. 令posInt是sign(number)×floor(abs(number))。

4. 令int32bit是posInt模2^32；也就是说，一个Number类型的有限整数k，带有正号并小于2^32，使得posInt和k之差是2^32的整数倍。

5. 如果int32bit大于等于2^31，返回int32bit－2^32，否则返回int32bit。

注意：给定上述ToInt32的定义，

·ToInt32抽象操作是幂等的，即若对它产生的结果再次应用，第二次应用返回相同结果。

·ToInt32(ToUint32(x))等于ToInt32(x)，对所有x值。（正是为了保持这一性质才把**+∞**和**-∞**映射成**+0**。）

·ToInt32把**-0**映射成**+0**。

9.6节 ToUint32：（无符号32位整数）

抽象操作ToUint32依照如下方式把input转换成位于闭区间[0, 2^32－1]里的2^32个整数值之一：

1. 令number是调用ToNumber(input)的结果。

2. 如果number是**NaN**，**+0**，**-0**，**+∞**，或**-∞**，返回**+0**。

3. 令posInt是sign(number)×floor(abs(number))。

4. 令int32bit是posInt模2^32；也就是说，一个Number类型的有限整数k，带有正号并小于2^32，使得posInt和k之差是2^32的整数倍。

5. 返回int32bit。

注意：给定上述ToUint32的定义，

·第5步是ToUint32和ToInt32之间唯一的区别。

·ToUint32抽象操作是幂等的，即若对它产生的结果再次应用，第二次应用返回相同结果。

·ToUint32(ToInt32(x))等于ToUint32(x)，对所有x值。（正是为了保持这一性质才把**+∞**和**-∞**映射成**+0**。）

·ToUint32把**-0**映射成**+0**。

9.7节 ToUint16：（无符号16位整数）

抽象操作ToUint16依照如下方式把input转换成位于闭区间[0, 2^16－1]里的2^16个整数值之一：

1. 令number是调用ToNumber(input)的结果。

2. 如果number是**NaN**，**+0**，**-0**，**+∞**，或**-∞**，返回**+0**。

3. 令posInt是sign(number)×floor(abs(number))。

4. 令int16bit是posInt模2^16；也就是说，一个Number类型的有限整数k，带有正号并小于2^16，使得posInt和k之差是2^16的整数倍。

5. 返回int16bit。

注意：给定上述ToUint16的定义，

·第4步里用2^16替换2^32是ToUint16和ToUint32之间唯一的区别。

·ToUint16把**-0**映射成**+0**。

【SP60】

9.8节 ToString

抽象操作ToString依照表格13把input转换成一个String类型的值：

表格13 —— ToString转换

┌─────────────────────────────────────┐

│input类型 结果 │

├─────────────────────────────────────┤

│Undefined "undefined" │

│Null "null" │

│Boolean #1 │

│Number 参阅9.8.1段 │

│String 结果等于input实参（没有转换）。 │

│Object #2 │

└─────────────────────────────────────┘

#1 如果input为**true**，结果是"true"；如果input为**false**，结果是"false"。

#2 应用下面的步骤：

1. 令primValue是ToPrimitive(input, String)。【提示首选类型是String】

2. 返回ToString(primValue)。

9.8.1段 应用到Number类型的ToString

抽象操作ToString依照如下方式把一个Number值m转换成String格式。

1. 如果m是**NaN**，返回String值"NaN"。

2. 如果m是**+0**或**-0**，返回String值"0"。

3. 如果m小于0，返回String值"-"和ToString(**-**m)的拼接。

4. 如果m是无穷大，返回String值"Infinity"。

5. 否则，令n，k，和s是一些整数，使得k≥1,10^(k－1)≤s＜10^k，s×10^(n－k)的Number值是m，且k尽可能的小。注意k是s的十进制表示的数位数量，s不可被10除尽，s的最低有效数位不一定能被这些必须条件唯一确定。

6. 如果k≤n≤21，返回由s的十进制表示的k个数位（按序，没有前导零），后跟n－k个字符“0”构成的String。

7. 如果0＜n≤21，返回由s的十进制表示的n个最高有效数位，后跟一个小数点“.”，后跟s的十进制表示的剩余的k－n个数位构成的String。

8. 如果-6＜n≤0，返回由字符“0”，后跟一个小数点“.”，后跟-n个字符“0”，后跟s的十进制表示的k个数位构成的String。

9. 否则，如果k＝1，返回由s的单一数位，后跟小写字符“e”，后跟正号“+”或负号“-”，依据n－1是正是负，后跟整数abs(n－1)的十进制表示构成的String（没有前导零）。

10. 返回由s的十进制表示的最高有效数位，后跟一个小数点“.”，后跟s的十进制表示的剩余的k－1个数位，后跟小写字符“e”，后跟正号“+”或负号“-”，依据n－1是正是负，后跟整数abs(n－1)的十进制表示构成的String（没有前导零）。

注意1：对于实现把下面的观察用作指导方针可能是有用的，但它们不是这份规范的标准化要求。

·如果x是**-0**以外的任何Number值，则ToNumber(ToString(x))和x是完全相同的Number值。

·s的最低有效数位并不总是能被列在步骤5里的必须条件唯一确定。

注意2：对于——提供了比上述规则要求的更为精确的转换的——实现，推荐把下面步骤5的另一供选版本作为指导方针。

否则，令n，k，和s是一些整数，使得k≥1,10^(k－1)≤s＜10^k，s×10^(n－k)的Number值是m，且k尽可能的小。如果s有多个可能的取值，选择s×10^(n－k)在数值上最接近于m的那个s值；如果有两个这样的s值，选择为偶数的那个。注意k是s的十进制表示的数位数量，s不可被10除尽。

【SP61】

注意3：ECMAScript的实现者可能发现由David M. Gay为浮点数的二进制到十进制转换撰写的论文和代码会帮上忙，

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。在如下位置

<http://cm.bell-labs.com/cm/cs/doc/90/4-10.ps.gz>

相关的代码在如下位置

<http://cm.bell-labs.com/netlib/fp/dtoa.c.gz>

和

<http://cm.bell-labs.com/netlib/fp/g_fmt.c.gz>

而且在各种netlib镜像站点也可能找到。

9.9节 ToObject

抽象操作ToObject依照表格14把input转换成一个Object类型的值：

表格14 —— ToObject转换

┌─────────────────────────────────────┐

│input类型 结果 │

├─────────────────────────────────────┤

│Undefined 抛出一个TypeError异常。 │

│Null 抛出一个TypeError异常。 │

│Boolean #1 │

│Number #2 │

│String #3 │

│Object 结果等于input实参（没有转换）。 │

└─────────────────────────────────────┘

#1 创建一个新的Boolean对象，它的[[PrimitiveValue]]内部属性被设为input的值，参阅15.6节关于Boolean对象的描述。

#2 创建一个新的Number对象，它的[[PrimitiveValue]]内部属性被设为input的值，参阅15.7节关于Number对象的描述。

#3 创建一个新的String对象，它的[[PrimitiveValue]]内部属性被设为input的值，参阅15.5节关于String对象的描述。

9.10节 CheckObjectCoercible

如果input是一个使用ToObject不能转换成一个Object的值，抽象操作CheckObjectCoercible就会抛出一个异常，它由表格15定义：

表格15 —— CheckObjectCoercible结果

┌─────────────────────────────────────┐

│input类型 结果 │

├─────────────────────────────────────┤

│Undefined 抛出一个TypeError异常。 │

│Null 抛出一个TypeError异常。 │

│Boolean 返回。 │

│Number 返回。 │

│String 返回。 │

│Object 返回。 │

└─────────────────────────────────────┘

9.11节 IsCallable

抽象操作IsCallable依照表格16判定input，其必须是一个ECMAScript语言值，是否是一个可调用的函数Object。

【SP62】

表格16 —— IsCallable结果

┌─────────────────────────────────────┐

│input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 #1 │

└─────────────────────────────────────┘

#1 如果input有一个[[Call]]内部方法，则返回**true**，否则返回**false**。

9.12节 SameValue算法

内部比较抽象操作SameValue(x, y)，其中x和y是ECMAScript语言值，产生**true**或**false**。这样一个比较按如下方式进行：

1. 如果Type(x)不同于Type(y)，返回**false**。

2. 如果Type(x)是Undefined，返回**true**。

3. 如果Type(x)是Null，返回**true**。

4. 如果Type(x)是Number，那么

a. 如果x是**NaN**，且y是**NaN**，返回**true**。

b. 如果x是**+0**，且y是**-0**，返回**false**。

c. 如果x是**-0**，且y是**+0**，返回**false**。

d. 如果x和y具有相同的Number值，返回**true**。

e. 返回**false**。

5. 如果Type(x)是String，则当x和y是完全相同的字符序列时（长度相同，且在对应位置字符相同）返回**true**，否则返回**false**。

6. 如果Type(x)是Boolean，则当x和y都是**true**或都是**false**时返回**true**，否则返回**false**。

7. 如果x和y引用同一个对象，则返回**true**，否则返回**false**。

第10章 可执行代码和执行上下文

10.1节 可执行代码类型

有三种类型的ECMAScript可执行代码：

·全局代码，是当作一个ECMAScript *Program*处理的源文本。一个特定*Program*的全局代码不包括任何作为一个*FunctionBody*的一部分被解析的源文本。

·求值代码，是提供给内建**eval**函数的源文本。更准确地讲，如果内建**eval**函数的实参是一个String，它就被当作一个ECMAScript *Program*处理。用于**eval**一次特定调用的求值代码就是那个*Program*的全局代码部分【还可能有其它部分，如求值代码】。

·函数代码，是作为一个*FunctionBody*的一部分被解析的源文本。一个特定*FunctionBody*的函数代码不包括任何当作一个嵌套*FunctionBody*的一部分被解析的源文本。函数代码同样表示当把内建Function对象作为构造函数使用时所提供的源文本。更准确地讲，提供给Function构造函数的最后一个实参被转换成一个String，并作为*FunctionBody*处理。如果有多个实参提供给Function构造函数，除了最后一个之外的所有实参被转换成String并拼接到一起，以逗号分隔；产生的String作为——用于由最后一个实参定义的*FunctionBody*的——*FormalParameterList*来解释。用于Function的一次特定实例化的函数代码不包括任何作为一个嵌套*FunctionBody*的一部分被解析的源文本。

【SP63】

10.1.1段 严格模式代码

一个ECMAScript *Program*语法单元可以使用无限制的或严格模式语法和语义进行处理。当使用严格模式处理时，三种类型的ECMAScript代码分别被称作严格全局代码，严格求值代码，和严格函数代码。代码在以下情况里作为严格模式代码解释：

·全局代码是严格全局代码，如果它以一个包含了Use Strict指令的指令序言开始（参阅14.1节）。

·求值代码是严格求值代码，如果它以一个包含了Use Strict指令的指令序言开始，或对**eval**的调用是一个对包含在严格模式代码里的**eval**函数的直接调用（参阅15.1.2.1.1则）。

·作为*FunctionDeclaration*，*FunctionExpression*，或存取器*PropertyAssignment*一部分的函数代码是严格函数代码，如果它所在的*FunctionDeclaration*，*FunctionExpression*，或*PropertyAssignment*包含在严格模式代码里，或它以一个包含了Use Strict指令的指令序言开始。

·作为内建Function构造函数最后一个实参提供的函数代码是严格函数代码，如果该实参是一个String，且当作为一个*FunctionBody*处理时，它以一个包含了Use Strict指令的指令序言开始。

10.2节 词法环境

一个Lexical Environment是一个规范类型，用于根据ECMAScript代码的词法嵌套结构定义标识符到特定变量和函数的联系。一个Lexical Environment是由一个Environment Record和一个可能为空的、对一个外部Lexical Environment的引用构成的。通常一个Lexical Environment被关联到一些ECMAScript代码的特定语法结构，例如一个*FunctionDeclaration*，一个*WithStatement*，或一个*TryStatement*的*Catch*子句，并且这些代码每次被求值时都会创建一个新的Lexical Environment。

一个Environment Record记录了——在它关联的Lexical Environment的作用域里创建的——标识符绑定。

外部环境引用被用来模塑多个Lexical Environment值的逻辑嵌套。一个（内部）Lexical Environment的外部引用是一个——对在逻辑上包围内部Lexical Environment的Lexical Environment的——引用。一个外部Lexical Environment当然可以具有它自己的外部Lexical Environment。一个Lexical Environment可以作为多个内部Lexical Environment的外部环境。例如，如果一个*FunctionDeclaration*内含两个*FunctionDeclaration*，那么每个内层函数的Lexical Environment将把外层函数当前执行使用的Lexical Environment作为它们的外部Lexical Environment。

Lexical Environment和Environment Record值是纯粹的规范机制，不需要对应到一个ECMAScript实现的任何具体产物。一个ECMAScript程序是不可能直接访问或操纵这些值的。

10.2.1段 环境记录

在这份规范里用到了两种Environment Record值：声明环境记录和对象环境记录。声明环境记录被用来定义——直接把标识符绑定和ECMAScript语言值相关联的——ECMAScript语言语法元素的功效，例如*FunctionDeclaration*，*VariableDeclaration*，和*Catch*子句。对象环境记录被用来定义——把标识符绑定和一些对象的属性相关联的——ECMAScript元素的功效，例如*Program*和*WithStatement*。

为了这份规范的目的，Environment Record值可以被想象成存在于一个简单的面向对象类层次结构里，其中Environment Record是一个抽象类，它有两个具体子类，分别是声明环境记录和对象环境记录。

【SP64】

抽象类含有定义在表格17里的抽象规范方法。这些抽象方法对于每一种具体子类都有不同的具体算法。

表格17 —— Environment Record的抽象方法

┌─────────────────────────────────────┐

│方法 用途 │

├─────────────────────────────────────┤

│HasBinding(N) #1 │

│CreateMutableBinding(N, D) #2 │

│SetMutableBinding(N, V, S) #3 │

│GetBindingValue(N, S) #4 │

│DeleteBinding(N) #5 │

│ImplicitThisValue() #6 │

└─────────────────────────────────────┘

#1 判定一个Environment Record是否具有关于一个标识符的绑定。如果有的话返回**true**，否则返回**false**。String值N是标识符的文本。

#2 在一个Environment Record里创建一个新的可变绑定。String值N是被绑定名称的文本。如果可选Boolean实参D为**true**，该绑定随后可以被删除。

#3 设置在一个Environment Record里已经存在的可变绑定的值。String值N是被绑定名称的文本。V是绑定的值，可以是任何ECMAScript语言类型的值。S是一个Boolean标志，如果S为**true**且绑定不能被设置，就会抛出一个TypeError异常。S被用来识别严格模式引用。

#4 返回在一个Environment Record里已经存在的绑定的值。String值N是被绑定名称的文本。S被用来识别严格模式引用。如果S为**true**且绑定不存在或未初始化，就会抛出一个ReferenceError异常。

#5 从一个Environment Record里删除一个绑定。String值N是被绑定名称的文本。如果关于N的绑定存在，移除那个绑定并返回**true**。如果绑定存在但不能被移除，返回**false**。如果绑定不存在，返回**true**。

#6 返回——当调用作为绑定值从该Environment Record得到的函数对象时——用作**this**值的值。

10.2.1.1则 声明环境记录

每个声明环境记录被关联到一个包含着变量和/或函数声明的ECMAScript程序作用域。一个声明环境记录绑定了一组——由包含在它关联作用域里的声明所定义的——标识符。

除了受到所有Environment Record支持的可变绑定外，声明环境记录还支持不可变绑定。一个不可变绑定是指——在一个标识符和一个值之间的联系一旦被建立后就不能再被修改的——绑定。不可变绑定的创建和初始化是两个不同的步骤，所以这种绑定存在于已初始化状态或未初始化状态都是可能的。声明环境记录除了支持Environment Record抽象规范方法外，还支持列在表格18里的方法：

表格18 —— 声明环境记录的附加方法

┌─────────────────────────────────────┐

│方法 用途 │

├─────────────────────────────────────┤

│CreateImmutableBinding(N) #1 │

│InitializeImmutableBinding(N, V) #2 │

└─────────────────────────────────────┘

#1 在一个Environment Record里创建一个新的、但未初始化的不可变绑定。String值N是被绑定名称的文本。

#2 设置在一个Environment Record里已经存在、但未初始化的不可变绑定的值。String值N是被绑定名称的文本。V是绑定的值，可以是任何ECMAScript语言类型的值。

【SP65】

用于声明环境记录的具体规范方法的行为由下面的算法定义。

10.2.1.1.1条 HasBinding(N)

用于声明环境记录的具体环境记录方法HasBinding只是判定实参标识符是否为被记录绑定的标识符之一：

1. 令envRec是这次方法调用所在的声明环境记录。

2. 如果envRec具有关于名称为N的值的绑定，返回**true**。

3. 如果它没有这样一个绑定，返回**false**。

10.2.1.1.2条 CreateMutableBinding(N, D)

用于声明环境记录的具体环境记录方法CreateMutableBinding创建一个新的、关于名称N的可变绑定，初始值为**undefined**。在这一Environment Record里绝不能已经存在关于N的绑定。如果提供了Boolean实参D且值为**true**，则新建的绑定被标记为将经历删除。

1. 令envRec是这次方法调用所在的声明环境记录。

2. 断言：envRec尚无关于N的绑定。

3. 在envRec里创建一个关于N的可变绑定，并把绑定值设为**undefined**。如果D为**true**，将新建的绑定标记为可被后来的DeleteBinding调用删除。

10.2.1.1.3条 SetMutableBinding(N, V, S)

用于声明环境记录的具体环境记录方法SetMutableBinding尝试把——名称为N的值的标识符的当前绑定的——绑定值改变为V的值。一个关于N的绑定必须已经存在。如果那个绑定是一个不可变绑定，且S为**true**，则抛出一个TypeError异常。

1. 令envRec是这次方法调用所在的声明环境记录。

2. 断言：envRec一定具有一个关于N的绑定。

3. 如果在envRec里关于N的绑定是一个可变绑定，把它的绑定值改成V。

4. 否则，这一定是尝试改变一个不可变绑定的值，所以如果S为**true**，抛出一个TypeError异常。

10.2.1.1.4条 GetBindValue(N, S)

用于声明环境记录的具体环境记录方法GetBindValue只是返回名称为N的值的标识符的绑定值。一个关于N的绑定必须已经存在。如果S为**true**，且那个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是这次方法调用所在的声明环境记录。

2. 断言：envRec一定具有一个关于N的绑定。

3. 如果在envRec里关于N的绑定是一个未初始化的不可变绑定，那么

a. 如果S为**false**，返回**undefined**，否则抛出一个ReferenceError异常。

4. 否则，返回当前绑定到在envRec里的N的值。

10.2.1.1.5条 DeleteBinding(N)

用于声明环境记录的具体环境记录方法DeleteBinding只能删除已被显式标记为将经历删除的绑定。

1. 令envRec是这次方法调用所在的声明环境记录。

2. 如果envRec没有关于名称为N的值的绑定，返回**true**。

3. 如果关于在envRec里的N的绑定不能被删除，返回**false**。

【SP66】

4. 从envRec里移除关于N的绑定。

5. 返回**true**。

10.2.1.1.6条 ImplicitThisValue()

声明环境记录总是返回**undefined**作为它们的ImplicitThisValue。

1. 返回**undefined**。

10.2.1.1.7条 CreateImmutableBinding(N)

用于声明环境记录的具体环境记录方法CreateImmutableBinding创建一个新的、关于名称N的不可变绑定，初始值为**undefined**。在这一Environment Record里绝不能已经存在关于N的绑定。

1. 令envRec是这次方法调用所在的声明环境记录。

2. 断言：envRec尚无关于N的绑定。

3. 在envRec里创建一个关于N的不可变绑定，并记录它尚为未初始化。

10.2.1.1.8条 InitializeImmutableBinding(N, V)

用于声明环境记录的具体环境记录方法InitializeImmutableBinding被用来把——名称为N的值的标识符的当前绑定的——绑定值设为V的值。一个关于N的未初始化不可变绑定必须已经存在。

1. 令envRec是这次方法调用所在的声明环境记录。

2. 断言：envRec一定具有一个关于N的未初始化不可变绑定。

3. 把在envRec里关于N的绑定值设为V。

4. 记录在envRec里关于N的不可变绑定已被初始化。

10.2.1.2则 对象环境记录

每个对象环境记录都被关联到一个对象，称为它的绑定对象。一个对象环境记录绑定了一组——直接对应到它的绑定对象的属性名称的——标识符名称。不是*IdentifierName*的属性名称没有包含在这个已绑定标识符的集合里。自有和继承属性都被包含在集合里，无论它们的[[Enumerable]]特性如何设置。因为属性可以动态地添加到对象或从对象删除，由一个对象环境记录绑定的标识符集合可能发生变化，作为任何添加或删除属性操作的副作用。任何由于这样一种副作用而创建的绑定被认为是一个可变绑定，即使对应属性的[[Writable]]特性值为**false**。对于对象环境记录不存在不可变绑定。

对象环境记录可被配置为，提供它们的绑定对象作为隐式**this**值，为在函数调用里使用。这一功能被用来指定With语句（12.10节）人工绑定的行为。该功能由一个关联到每个对象环境记录的Boolean值provideThis控制。默认情况下，对于任何对象环境记录，provideThis的值为**false**。

用于对象环境记录的具体规范方法的行为由下面的算法定义。

10.2.1.2.1条 HasBinding(N)

用于对象环境记录的具体环境记录方法HasBinding判定它关联的绑定对象是否具有一个名称为N的值的属性：

1. 令envRec是这次方法调用所在的对象环境记录。

2. 令bindings是envRec关联的绑定对象。

3. 传入N作为实参，返回调用bindings的[[HasProperty]]内部方法的结果。

【SP67】

10.2.1.2.2条 CreateMutableBinding(N, D)

用于对象环境记录的具体环境记录方法CreateMutableBinding在一个环境记录关联的绑定对象里创建一个属性，它的名称是一个String值并被初始化为**undefined**。在该绑定对象里绝不能已经存在名为N的属性。如果提供了Boolean实参D且值为**true**，则新建的属性的[[Configurable]]特性被设为**true**，否则被设为**false**。

1. 令envRec是这次方法调用所在的对象环境记录。

2. 令bindings是envRec关联的绑定对象。

3. 断言：传入N作为实参，调用bindings的[[HasProperty]]内部方法的结果是**false**。

4. 如果D为**true**，那么令configValue是**true**，否则令configValue是**false**。

5. 传入N，Property Descriptor { [[Value]]:**undefined**, [[Writable]]:**true**, [[Configurable]]:configValue }，和**true**作为实参，调用bindings的[[DefineOwnProperty]]内部方法。

10.2.1.2.3条 SetMutableBinding(N, V, S)

用于对象环境记录的具体环境记录方法SetMutableBinding尝试把——环境记录关联的绑定对象的名称为N的值的属性的——值设为V的值。一个名为N的属性应该已经存在，但如果不存在或当前不可写，错误处理由Boolean实参S的值决定。

1. 令envRec是这次方法调用所在的对象环境记录。

2. 令bindings是envRec关联的绑定对象。

3. 传入N，V，和S作为实参，调用bindings的[[Put]]内部方法。

10.2.1.2.4条 GetBindValue(N, S)

用于对象环境记录的具体环境记录方法GetBindValue返回它关联的绑定对象的名称为N的值的属性的值。一个名为N的属性应该已经存在，但如果不存在，结果取决于Boolean实参S的值。

1. 令envRec是这次方法调用所在的对象环境记录。

2. 令bindings是envRec关联的绑定对象。

3. 传入N作为实参，令value是调用bindings的[[HasProperty]]内部方法的结果。

3. 如果value为**false**，那么

a. 如果S为**false**，返回**undefined**，否则抛出一个ReferenceError异常。

5. 传入N作为实参，返回调用bindings的[[Get]]内部方法的结果。

10.2.1.2.5条 DeleteBinding(N)

用于对象环境记录的具体环境记录方法DeleteBinding只能删除——对应到环境记录关联的绑定对象的[[Configurable]]特性值为**true**的属性的——绑定。

1. 令envRec是这次方法调用所在的对象环境记录。

2. 令bindings是envRec关联的绑定对象。

3. 传入N和**false**作为实参，返回调用bindings的[[Delete]]内部方法的结果。

10.2.1.2.6条 ImplicitThisValue()

对象环境记录返回**undefined**作为它们的ImplicitThisValue，除非它们的provideThis标志为**true**。

1. 令envRec是这次方法调用所在的对象环境记录。

2. 如果envRec的provideThis标志为**true**，返回envRec关联的绑定对象。

3. 否则，返回**undefined**。

【SP68】

10.2.2段 Lexical Environment操作

在这份规范里使用下面的抽象操作来操作Lexical Environment。

10.2.2.1则 GetIdentifierReference(lex, name, strict)

抽象操作GetIdentifierReference的实参分别是Lexical Environment值lex，标识符String值name，和Boolean标志strict。lex的值可以是**null**。当调用时采取以下步骤：

1. 如果lex为**null**，那么

a. 返回一个Reference类型的值，它的基值是**undefined**，被引用的名称是name，严 格模式标志是strict。

2. 令envRec是lex的环境记录。

3. 传入name作为实参，令exists是调用envRec的HasBinding(N)具体方法的结果。

4. 如果exists为**true**，那么

a. 返回一个Reference类型的值，它的基值是envRec，被引用的名称是name，严 格模式标志是strict。

5. 否则

a. 令outer是lex的外部环境引用的值。

b. 传入outer，name，和strict作为实参，返回调用GetIdentifierReference的结 果。

10.2.2.2则 NewDeclarativeEnvironment(E)

传入一个Lexical Environment或**null**作为实参，当调用抽象操作NewDeclarativeEnvironment时采取以下步骤：

1. 令env是一个新建的Lexical Environment。

2. 令envRec是一个新建的声明环境记录，没有包含绑定。

3. 把env的环境记录设为envRec。

4. 把env的外部Lexical Environment引用设为E。

5. 返回env。

10.2.2.3则 NewObjectEnvironment(O, E)

传入一个Object值O和一个Lexical Environment值E（或**null**）作为实参，当调用抽象操作NewObjectEnvironment时采取以下步骤：

1. 令env是一个新建的Lexical Environment。

2. 令envRec是一个新建的对象环境记录，以O作为绑定对象。

3. 把env的环境记录设为envRec。

4. 把env的外部Lexical Environment引用设为E。

5. 返回env。

10.2.3段 Global Environment

Global Environment是一个独特的Lexical Environment，它是在任何ECMAScript代码执行前创建的。Global Environment的Environment Record是一个对象环境记录，它的绑定对象是Global对象（15.1节）。Global Environment的外部环境引用是**null**。

随着ECMAScript代码的执行,附加的属性可以被添加到Global对象上，初始属性也可以被修改。

10.3节 执行上下文

当控制被转移到ECMAScript可执行代码时，控制进入到一个执行上下文。活动的执行上下文在逻辑上形成了一个栈。这个逻辑栈的栈顶执行上下文就是当前运作中的执行上下文。每当控制从——与当前运作中的执行上下文关联的可执行代码——转移到——未与那个执行上下文关联的可执行代码——时，一个新的执行上下文就被创建出来。

【SP69】

新创建的执行上下文被压入到栈中，并成为当前运作中的执行上下文。

一个执行上下文包含了任何——对于追踪它关联的代码的执行进程所需的——状态。此外，每个执行上下文具有列在表格19里的状态组件。

表格19 —— 执行上下文状态组件

┌─────────────────────────────────────┐

│组件 用途 │

├─────────────────────────────────────┤

│LexicalEnvironment #1 │

│VariableEnvironment #2 │

│ThisBinding #3 │

└─────────────────────────────────────┘

#1 标识一个Lexical Environment，它被用来解析由该执行上下文里的代码发出的标识符引用。

#2 标识一个Lexical Environment，它的环境记录携带着由该执行上下文里的*VariableStatements*和*FunctionDeclaration*创建的绑定。

#3 与——和该执行上下文关联的ECMAScript代码里的——**this**关键字关联的值。

一个执行上下文的LexicalEnvironment和VariableEnvironment组件总是Lexical Environment类型的值。当一个执行上下文被创建时，它的LexicalEnvironment和VariableEnvironment组件初始具有相同的值。VariableEnvironment组件的值永远不会改变，然而LexicalEnvironment组件的值可以在一个执行上下文里的代码执行期间改变【catch、with语句等】。

在大多数情况下，只有当前运作中的执行上下文（执行上下文栈的栈顶）被这份规范里的算法直接操纵。因此当不加限定的使用术语“LexicalEnvironment”，“VariableEnvironment”和“ThisBinding”时，它们表示当前运作中的执行上下文的那些组件。

执行上下文纯粹是一种规范机制，不需要对应到一个ECMAScript实现的任何具体产物。一个ECMAScript程序是不可能访问到一个执行上下文的。

10.3.1段 标识符解析

标识符解析是指——使用当前运作中的执行上下文的LexicalEnvironment，确定一个*Identifier*的绑定的——过程。在ECMAScript代码的执行期间，语法产生式*PrimaryExpression* : *Identifier*使用以下算法求值：

1. 令env是当前运作中的执行上下文的LexicalEnvironment。

2. 如果正被求值的语法产生式包含在严格模式代码里，则令strict为**true**，否则令strict为**false**。

3. 传入env，*Identifier*，和strict作为实参，返回调用GetIdentifierReference函数的结果。

对一个标识符求值的结果总是一个Reference类型的值，其中被引用的名称分量等于String值*Identifier*。

10.4节 建立一个执行上下文

求值全局代码，或使用**eval**函数（15.1.2.1则）求值代码，就会建立并进入一个新的执行上下文。对一个ECMAScript代码函数（13.2.1段）的每次调用也会建立并进入一个新的执行上下文，即使一个函数递归调用它自身。每次返回就会退出一个执行上下文。一个被抛出的异常也会退出一个或多个执行上下文。

当控制进入一个执行上下文，执行上下文的ThisBinding被设置，它的VariableEnvironment和初始LexicalEnvironment被定义，并且声明绑定实例化（10.5节）也被执行。这些动作发生的确切方式取决于进入的代码类型。

【SP70】

10.4.1段 进入全局代码

当控制进入用于全局代码的执行上下文时，执行以下步骤：

1. 使用全局代码初始化执行上下文，如10.4.1.1则所述。

2. 使用全局代码执行声明绑定实例化，如10.5节所述。

10.4.1.1则 初始化全局执行上下文

要初始化一个用于ECMAScript 代码C的全局执行上下文，执行以下步骤：

1. 把VariableEnvironment设为Global Environment。

2. 把LexicalEnvironment设为Global Environment。

3. 把ThisBinding设为Global对象。

10.4.2段 进入求值代码

当控制进入用于求值代码的执行上下文时，执行以下步骤：

1. 如果没有调用上下文，或求值代码并不是通过对**eval**函数的直接调用（15.1.2.1.1条）被求值的，那么

a. 初始化执行上下文，如同它是一个以求值代码作为C的全局执行上下文，如 10.4.1.1则所述。

2. 否则，

a. 把ThisBinding设为和调用执行上下文的ThisBinding相同的值。

b. 把LexicalEnvironment设为和调用执行上下文的LexicalEnvironment相同的值。

c. 把VariableEnvironment设为和调用执行上下文的VariableEnvironment相同的值。

3. 如果求值代码是严格代码，那么

a. 传入LexicalEnvironment作为实参，令strictVarEnv是调用 NewDeclarativeEnvironment的结果。

b. 把LexicalEnvironment设为strictVarEnv。

c. 把VariableEnvironment设为strictVarEnv。

4. 使用求值代码执行声明绑定实例化，如10.5节所述。

10.4.2.1则 严格模式限制

求值代码不能在引发**eval**的调用上下文的VariableEnvironment里实例化变量或函数绑定，如果调用上下文的代码或求值代码两者中的任何一个是严格代码。取而代之，这些绑定在一个新建的、只对求值代码可访问的VariableEnvironment里实例化。

10.4.3段 进入函数代码

对于包含在函数对象F里，伴有一个调用者提供的thisArg，和一个调用者提供的argumentsList的函数代码，当控制进入用于该函数代码的执行上下文时，执行以下步骤：

1. 如果函数代码是严格代码，把ThisBinding设为thisArg。

2. 否则如果thisArg为**null**或**undefined**，把ThisBinding设为Global对象。

3. 否则如果Type(thisArg)不是Object，把ThisBinding设为ToObject(thisArg)。

4. 否则把ThisBinding设为thisArg。

5. 传入F的[[Scope]]内部属性的值作为实参，令localEnv是调用NewDeclarativeEnvironment的结果。

6. 把LexicalEnvironment设为localEnv。

7. 把VariableEnvironment设为localEnv。

8. 令code是F的[[Code]]内部属性的值。

9. 使用函数代码code和argumentsList执行声明绑定实例化，如10.5节所述。

【SP71】

10.5节 声明绑定实例化

每个执行上下文都有一个关联的VariableEnvironment。声明在ECMAScript代码里并在一个执行上下文里求值的变量和函数，作为绑定被添加到那个VariableEnvironment的Environment Record中。对于函数代码，参数同样作为绑定被添加到那个Environment Record中。

使用哪个和哪种Environment Record来绑定一个声明，取决于由执行上下文执行的ECMAScript代码类型，但其余行为是通用的。当进入一个执行上下文时，使用调用者提供的code和，如果它是函数代码，实参列表args，按照下述方式在VariableEnvironment里创建绑定：

1. 令env是当前运作中的执行上下文的VariableEnvironment的Environment Record组件。

2. 如果code为求值代码，那么令configurableBindings是**true**，否则令configurableBindings是**false**。

3. 如果code为严格模式代码，那么令strict是**true**，否则令strict是**false**。

4. 如果code为函数代码，那么

a. 令func是那个函数，该函数的[[Call]]内部方法引发code的执行。令names 是 func的[[FormalParameters]]内部属性的值。

b. 令argCount是args里的元素数量。

c. 令n是数字0。

d. 对于names里的每个String值argName，按列出顺序执行

i. 令n是n的当前值加1。

ii. 如果n大于argCount，令v是**undefined**，否则令v是args第n个元素的值。

iii. 传入argName作为实参，令argAlreadyDeclared是调用env的HasBinding 具体方法的结果。

iv. 如果argAlreadyDeclared为**false**，传入argName作为实参，调用env的 CreateMutableBinding具体方法。

v. 传入argName，v，和strict作为实参，调用env的SetMutableBinding具体 方法。

5. 对于code里的每个*FunctionDeclaration* f，按源文本顺序执行

a. 令fn是*FunctionDeclaration* f里的*Identifier*。

b. 令fo是实例化*FunctionDeclaration* f的结果，如第13章所述。

c. 传入fn作为实参，令funcAlreadyDeclared是调用env的HasBinding具体方法的 结果。

d. 如果funcAlreadyDeclared为**false**，传入fn和configurableBindings作为实参， 调用env的CreateMutableBinding具体方法。

e. 否则如果env是Global Environment的Environment Record组件，那么

i. 令go是Global对象。

ii. 传入fn作为实参，令existingProp是调用go的[[GetProperty]]内部方法的 结果。

iii. 如果existingProp.[[Configurable]]为**true**，那么

1. 传入fn，Property Descriptor { [[Value]]:**undefined**, [[Writable]]:**true**, [[Enumerable]]:**true**, [[Configurable]]:configurableBindings }，和**true**作为实参，调用go的 [[DefineOwnProperty]]内部方法。

iv. 否则如果IsAccessorDescriptor(existingProp)为**true**，或existingProp 没有特性值 { [[Writable]]:**true**, [[Enumerable]]:**true** }，那么

1. 抛出一个TypeError异常。

f. 传入fn，fo，和strict作为实参，调用env的SetMutableBinding具体方法。

6. 传入"**arguments**"作为实参，令argumentsAlreadyDeclared是调用env的HasBinding具体方法的结果。

7. 如果code为函数代码，且argumentsAlreadyDeclared为**false**，那么

a. 传入func，names，args，env和strict作为实参，令argsObj是调用抽象操作 CreateArgumentsObject（10.6节）的结果。

b. 如果strict为**true**，那么

i. 传入"**arguments**"作为实参，调用env的CreateImmutableBinding具体方法。

ii. 传入"**arguments**"和argsObj作为实参，调用env的 InitializeImmutableBinding具体方法。

c. 否则，

【SP72】

i. 传入"**arguments**"作为实参，调用env的CreateMutableBinding具体方法。

ii. 传入"**arguments**"，argsObj和**false**作为实参，调用env的SetMutableBinding 具体方法。

8. 对于code里的每个*VariableDeclaration*和*VariableDeclarationNoIn* d，按源文本顺序执行

a. 令dn是d里的*Identifier*。

b. 传入dn作为实参，令varAlreadyDeclared是调用env的HasBinding具体方法的 结果。

c. 如果varAlreadyDeclared为**false**，那么

i. 传入dn和configurableBindings作为实参，调用env的 CreateMutableBinding具体方法。

ii. 传入dn，**undefined**，和strict作为实参，调用env的SetMutableBinding 具体方法。

10.6节 实参对象

当控制进入一个用于函数代码的执行上下文时，一个**arguments**对象被创建出来，除非（如10.5节规定）标识符**arguments**作为一个*Identifier*出现在函数的*FormalParameterList*里，或作为一个*VariableDeclaration*或*FunctionDeclaration*的*Identifier*包含在函数代码里。

传入代码将被求值的函数对象func，包含函数形参名的List值names，传递给[[Call]]内部方法的实参args，用于函数代码的VariableEnvironment【的Environment Record】值env，和指示函数代码是否为严格代码的Boolean值strict作为实参，**arguments**对象通过调用抽象操作CreateArgumentsObject创建，当调用时执行以下步骤：

1. 令len是args里的元素数量。

2. 令obj是创建一个新ECMAScript对象的结果。

3. 依照8.12节规定，设置obj的全部内部方法。

4. 把obj的内部属性[[Class]]设为"Arguments"。

5. 令Object是标准内建Object构造函数（15.2.2段）。

6. 把obj的[[Prototype]]内部属性设为标准内建Object原型对象（15.2.4段）。

7. 传入"length"，Property Descriptor { [[Value]]:len, [[Writable]]**true**, [[Enumerable]]:**false**, [[Configurable]]:**true** }，和**false**作为实参，在obj上调用[[DefineOwnProperty]]内部方法。

8. 令map是创建一个新对象的结果，如同使用表达式**new** Object**()**。

9. 令mappedNames是一个空List。

10. 令indx＝len－1。

11. 当indx≥0时重复，

a. 令val是args在从0开始的List位置indx上的元素。

b. 传入ToString(indx)，Property Descriptor { [[Value]]:val, [[Writable]]:**true**, [[Enumerable]]:**true**, [[Configurable]]:**true** }，和**false**作 为实参，在obj上调用[[DefineOwnProperty]]内部方法。

c. 如果indx小于names里的元素数，那么

i. 令name是names在从0开始的List位置indx上的元素。

ii. 如果strict为**false**，且name不是mappedNames的元素，那么

1. 添加name作为列表mappedNames的一个元素。

2. 传入name和env作为实参，令g是调用MakeArgGetter抽象操作的结果。

3. 传入name和env作为实参，令p是调用MakeArgSetter抽象操作的结果。

4. 传入ToString(indx)，Property Descriptor { [[Set]]:p, [[Get]]:g, [[Configurable]]:**true** }，和**false**作为实参，调用map的 [[DefineOwnProperty]]内部方法。

d. 令indx＝indx－1。

12. 如果mappedNames非空，那么

a. 把obj的[[ParameterMap]]内部属性设为map。

b. 把obj的[[Get]]，[[GetOwnProperty]]，[[DefineOwnProperty]]，和[[Delete]] 内部方法设为提供在下面的定义。

【SP73】

13. 如果strict为**false**，那么

a. 传入"**callee**"，Property Descriptor { [[Value]]:func, [[Writable]]:**true**, [[Enumerable]]:**false**, [[Configurable]]:**true** }，和**false**作为实参，在obj上调 用[[DefineOwnProperty]]内部方法。

14. 否则，strict为**true**，所以

a. 令thrower是[[ThrowTypeError]]函数对象（13.2.3段）。

b. 传入"**caller**"，Property Descriptor { [[Get]]:thrower, [[Set]]thrower, [[Enumerable]]:**false**, [[Configurable]]:**false** }，和**false**作为实参，调用obj 的[[DefineOwnProperty]]内部方法。

c. 传入"**callee**"，Property Descriptor { [[Get]]:thrower, [[Set]]thrower, [[Enumerable]]:**false**, [[Configurable]]:**false** }，和**false**作为实参，调用obj 的[[DefineOwnProperty]]内部方法。

15. 返回obj。

抽象操作MakeArgGetter，接受实参String值name和Environment Record值env，创建一个函数对象，当执行时返回在env里为name绑定的值。它执行以下步骤：

1. 令body是拼接字符串"return"，name，和";"的结果。

2. 如13.2节所述，不指定*FormalParameterList*，为*FunctionBody*指定body，为*Scope*指定env，为*Strict*指定**true**，返回创建一个函数对象的结果。

抽象操作MakeArgSetter，接受实参String值name和Environment Record值env，创建一个函数对象，当执行时设置在env里为name绑定的值。它执行以下步骤：

1. 令param是拼接字符串name和String值"\_arg"的结果。

2. 令body是String值"<name>=<param>;"，并将<name>替换成name的值，<param>替换成param的值。

3. 如13.2节所述，使用一个仅包含String值param的列表作为*FormalParameterList*，为*FunctionBody*指定body，为*Scope*指定env，为*Strict*指定**true**，返回创建一个函数对象的结果。

传入属性名称P作为实参，当调用一个——用于带有形参的非严格模式函数的——**arguments**对象的[[Get]]内部方法时，执行以下步骤：

1. 令map是**arguments**对象的[[ParameterMap]]内部属性的值。

2. 传入P作为实参，令isMapped是调用map的[[GetOwnProperty]]内部方法的结果。

3. 如果isMapped为**undefined**，那么

a. 传入P作为实参，令v是在**arguments**对象上调用默认[[Get]]内部方法（8.12.3 段）的结果。

b. 如果P为"**caller**"，且v是一个严格模式Function对象，抛出一个TypeError异 常。

c. 返回v。

4. 否则，map包含一个关于P的形参映射，所以，

a. 传入P作为实参，返回调用map的[[Get]]内部方法的结果。

传入属性名称P作为实参，当调用一个——用于带有形参的非严格模式函数的——**arguments**对象的[[GetOwnProperty]]内部方法时，执行以下步骤：

1. 传入P作为实参，令desc是在**arguments**对象上调用默认[[GetOwnProperty]]内部方法（8.12.1段）的结果。

2. 如果desc为**undefined**，那么返回desc。

3. 令map是**arguments**对象的[[ParameterMap]]内部属性的值。

4. 传入P作为实参，令isMapped是调用map的[[GetOwnProperty]]内部方法的结果。

5. 如果isMapped非**undefined**，那么

a. 传入P作为实参，把desc.[[Value]]设为调用map的[[Get]]内部方法的结果。

6. 返回desc。

传入属性名称P，Property Descriptor值Desc，和Boolean标志Throw作为实参，当调用一个——用于带有形参的非严格模式函数的——**arguments**对象的[[DefineOwnProperty]]内部方法时，执行以下步骤：

【SP74】

1. 令map是**arguments**对象的[[ParameterMap]]内部属性的值。

2. 传入P作为实参，令isMapped是调用map的[[GetOwnProperty]]内部方法的结果。

3. 传入P，Desc，和**false**作为实参，令allowed是在**arguments**对象上调用默认[[DefineOwnProperty]]内部方法（8.12.9段）的结果。

4. 如果allowed为**false**，那么

a. 如果Throw为**true**，那么抛出一个TypeError异常，否则返回**false**。

5. 如果isMapped非**undefined**，那么

a. 如果IsAccessorDescriptor(Desc)为**true**，那么

i. 传入P和**false**作为实参，调用map的[[Delete]]内部方法。

b. 否则，

i. 如果Desc.[[Value]]存在，那么

1. 传入P，Desc.[[Value]]，和Throw作为实参，调用map的[[Put]]内部 方法。

ii. 如果Desc.[[Writable]]存在，且它的值为**false**，那么

1. 传入P和**false**作为实参，调用map的[[Delete]]内部方法。

6. 返回**true**。

传入属性名称P，和Boolean标志Throw作为实参，当调用一个——用于带有形参的非严格模式函数的——**arguments**对象的[[Delete]]内部方法时，执行以下步骤：

1. 令map是**arguments**对象的[[ParameterMap]]内部属性的值。

2. 传入P作为实参，令isMapped是调用map的[[GetOwnProperty]]内部方法的结果。

3. 传入P和Throw作为实参，令result是在**arguments**对象上调用默认[[Delete]]内部方法（8.12.7段）的结果。

4. 如果result为**true**，且isMapped非**undefined**，那么

a. 传入P和**false**作为实参，调用map的[[Delete]]内部方法。

5. 返回result。

注意1：对于非严格模式函数，一个**arguments**对象的——数字名【如**"**0**"**】之数值【如0】小于对应函数对象的形参数量的——数组索引（定义在15.4节）具名数据属性【如**"**0**"**】，初始与在函数的执行上下文里的对应实参绑定共享了它们的值。这意味着改变了属性也就改变了实参绑定的对应值，反之亦然。如果这样一个属性被删除然后重定义，或者该属性被改变成一个存取器属性，这种对应关系就会被破坏。对于严格模式函数，**arguments**对象的属性值只是传给函数的实参的拷贝，因此在属性值和形参值之间没有动态链接。

注意2：ParameterMap对象和它的属性值作为一种装置用于规定**arguments**对象和实参绑定的对应关系。ParameterMap对象和作为它属性值的对象不可从ECMAScript代码直接访问。一个ECMAScript实现不需要实际创建或使用这些对象来实现规定的语义。

注意3：用于严格模式函数的**arguments**对象定义了不可配置的存取器属性，名为“**caller**”和“**callee**”，当访问时会抛出一个TypeError异常。“**callee**”属性对于非严格模式函数具有一个更明确的含义，而“**caller**”属性在历史上被一些ECMAScript实现作为一个实现定义的扩展提供。这两个属性的严格模式定义之所以存在是为了确保它们中的任何一个都不会被遵从ECMAScript实现以任何其它方式定义。

【SP75】

第11章 表达式

11.1节 主【Primary】表达式

语法

*PrimaryExpression* :

**this**

*Identifier*

*Literal*

*ArrayLiteral*

*ObjectLiteral*

**(***Expression***)**

11.1.1段 **this**关键字

**this**关键字求值为当前执行上下文的ThisBinding的值。

11.1.2段 标识符引用

一个*Identifier*通过执行依照10.3.1段规定的标识符解析进行求值。求值一个*Identifier*的结果总是一个Reference类型的值。

11.1.3段 字面值引用

一个*Literal*依照7.8节所述进行求值。

11.1.4段 数组初始化器

一个数组初始化器是一个表达式，描述了一个Array对象的初始化，以字面值的形式编写。它是由零个或多个表达式构成的列表，其中每个表达式表示一个数组元素，整个列表被包含在一对方括号里。元素不必是字面值，数组初始化器每次被求值时元素也被求值。

数组元素在元素列表的开头，中间，或结尾处可以省略。每当元素列表里的一个逗号没有前带*AssignmentExpression*时（例如，一个逗号位于开头或另一个逗号之后），丢失的数组元素仍对数组的长度有贡献，并递增了后续元素的索引。省略的数组元素没有被定义【值为**undefined**】。如果一个元素在数组的结尾处省略，那个元素对数组的长度没有贡献。

语法

*ArrayLiteral* :

**[** *Elision* opt **]**

**[** *ElementList* **]**

**[** *ElementList, Elision* opt **]**

*ElementList* :

*Elision* opt *AssignmentExpression*

*ElementList* **,** *Elision* opt *AssignmentExpression*

*Elision* :

**,**

*Elision* **,**

语义

产生式*ArrayLiteral* : **[** *Elision* opt **]**按以下方式求值：

【SP76】

1. 令array是创建一个新对象的结果，如同使用表达式**new** Array**()**，其中Array是标准内建构造函数。

2. 令pad是求值*Elision*的结果，如果不存在，使用数字值0。

3. 传入"length"，pad，和**false**作为实参，调用array的[[Put]]内部方法。

4. 返回array。

产生式*ArrayLiteral* : **[** *ElementList* **]**按以下方式求值：

1. 返回求值*ElementList*的结果。

产生式*ArrayLiteral* : **[** *ElementList, Elision* opt **]**按以下方式求值：

1. 令array是求值*ElementList*的结果。

2. 令pad是求值*Elision*的结果，如果不存在，使用数字值0。

3. 传入"length"作为实参，令len是调用array的[[Get]]内部方法的结果。

4. 传入"length"，ToUint32(pad+len)，和**false**作为实参，调用array的[[Put]]内部方法。

5. 返回array。

产生式*ElementList* : *Elision* opt *AssignmentExpression*按以下方式求值：

1. 令array是创建一个新对象的结果，如同使用表达式**new** Array**()**，其中Array是标准内建构造函数。

2. 令firstIndex是求值*Elision*的结果，如果不存在，使用数字值0。

3. 令initResult是求值*AssignmentExpression*的结果。

4. 令initValue是GetValue(initResult)。

5. 传入ToString(firstIndex)，Property Descriptor { [[Value]]:initValue, [[Writable]]:**true**, [[Enumeralbe]]:**true**, [[Configurable]]:**true** }，和**false**作为实参，调用array的[[DefineOwnProperty]]内部方法。

6. 返回array。

产生式*ElementList* : *ElementList* **,** *Elision* opt *AssignmentExpression*按以下方式求值：

1. 令array是求值*ElementList*的结果。

2. 令pad是求值*Elision*的结果，如果不存在，使用数字值0。

3. 令initResult是求值*AssignmentExpression*的结果。

4. 令initValue是GetValue(initResult)。

5. 传入"length"作为实参，令len是调用array的[[Get]]内部方法的结果。

6. 传入ToString(ToUint32(pad+len))，Property Descriptor { [[Value]]:initValue, [[Writable]]:**true**, [[Enumeralbe]]:**true**, [[Configurable]]:**true** }，和**false**作为实参，调用array的[[DefineOwnProperty]]内部方法。

7. 返回array。

产生式*Elision* : **,** 按以下方式求值：

1. 返回数字值1。

产生式*Elision* : *Elision* **,** 按以下方式求值：

1. 令preceding是求值*Elision*的结果。

2. 返回preceding+1。

注意：[[DefineOwnProperty]]被用来确保已为数组定义自有属性，即使标准内建Array原型对象已按这样一种方式修改——将会阻止使用[[Put]]创建新的自有属性。

【SP77】

11.1.5段 对象初始化器

一个对象初始化器是一个表达式，描述了一个Object对象的初始化，以类似于字面值的形式编写。它是由零个或多个名值对构成的列表，其中每个名值对表示一个属性名称和关联的值，整个列表被包含在一对花括号里。值不必是字面值，对象初始化器每次被求值时值也被求值。

语法

*Object* :

**{ }**

**{** *PropertyNameAndValueList* **}**

**{** *PropertyNameAndValueList* **, }**

*PropertyNameAndValueList* :

*PropertyAssignment*

*PropertyNameAndValueList* **,** *PropertyAssignment*

*PropertyAssignment* :

*PropertyName* **:** *AssignmentExpression*

**get** *PropertyName* **( ) {** *FunctionBody* **}**

**set** *PropertyName* **(** *PropertySetParameterList* **) {** *FunctionBody* **}**

*PropertyName* :

*IdentifierName*

*StringLiteral*

*NumericLiteral*

*PropertySetParameterList* :

*Identifier*

语义

产生式*ObjectLiteral* : **{ }**按以下方式求值：

1. 返回一个新对象，如同使用表达式**new** Object**()**创建，其中Object是标准内建构造函数。

产生式*ObjectLiteral* : **{** *PropertyNameAndValueList* **}**和*ObjectLiteral* : **{** *PropertyNameAndValueList* **, }**按以下方式求值：

1. 返回求值*PropertyNameAndValueList*的结果。

产生式*PropertyNameAndValueList* : *PropertyAssignment*按以下方式求值：

1. 令obj是创建一个新对象的结果，如同使用表达式**new** Object**()**，其中Object是标准内建构造函数。

2. 令propId是求值*PropertyAssignment*的结果。

3. 传入propId.name，propId.descriptor，和**false**作为实参，调用obj的[[DefineOwnProperty]]内部方法。

4. 返回obj。

产生式*PropertyNameAndValueList* : *PropertyNameAndValueList* **,**  *PropertyAssignment*按以下方式求值：

1. 令obj是求值*PropertyNameAndValueList*的结果。

2. 令propId是求值*PropertyAssignment*的结果。

3. 传入propId.name作为实参，令previous是调用obj的[[GetOwnProperty]]内部方法的结果。

【SP78】

4. 如果previous非**undefined**，那么当以下条件中的任何一个成立时，抛出一个SyntaxError异常：

a. 该产生式包含在严格代码里，且IsDataDescriptor(previous)为**true**，且 IsDataDescriptor(propId.descriptor)为**true**。

b. IsDataDescriptor(previous)为**true**，且 IsAccessorDescriptor(propId.descriptor)为**true**。

c. IsAccessorDescriptor (previous)为**true**，且IsDataDescriptor (propId.descriptor)为**true**。

d. IsAccessorDescriptor (previous)为**true**，且IsAccessorDescriptor (propId.descriptor)为**true**，且要么previous和propId.descriptor都具有[[Get]] 字段，要么previous和propId.descriptor都具有[[Set]]字段。

5. 传入propId.name，propId.descriptor，和**false**作为实参，调用obj的[[DefineOwnProperty]]内部方法。

6. 返回obj。

如果以上步骤将会抛出一个SyntaxError，那么一个实现必须把该错误作为一个早期错误（第16章）。

产生式*PropertyAssignment* : *PropertyName* **:** *AssignmentExpression*按以下方式求值：

1. 令propName是求值*PropertyName*的结果。

2. 令exprValue是求值*AssignmentExpression*的结果。

3. 令propValue是GetValue(exprValue)。

4. 令desc是Property Descriptor { [[Value]]:propValue, [[Writable]]:**true**, [[Enumerable]]:**true**, [[Configurable]]:**true** }。

5. 返回Property Identifier(propName, desc)。

产生式*PropertyAssignment* : **get** *PropertyName* **( ) {** *FunctionBody* **}**按以下方式求值：

1. 令propName是求值*PropertyName*的结果。

2. 如13.2节所述，不指定*FormalParameterList*，为那里的*FunctionBody*指定这里的*FunctionBody*，为*Scope*指定当前运作中的执行上下文的LexicalEnvironment，如果*PropertyAssignment*包含在严格代码里或它的*FunctionBody*是严格代码则为*Strict*指定**true**，令closure是创建一个新Function对象的结果。

3. 令desc是Property Descriptor { [[Get]]:closure, [[Enumerable]]:**true**, [[Configurable]]:**true** }。

4. 返回Property Identifier(propName, desc)。

产生式*PropertyAssignment* : **set** *PropertyName* **(** *PropertySetParameterList* **) {** *FunctionBody* **}**按以下方式求值：

1. 令propName是求值*PropertyName*的结果。

2. 如13.2节所述，为*FormalParameterList*指定*PropertySetParameterList*，为那里的*FunctionBody*指定这里的*FunctionBody*，为*Scope*指定当前运作中的执行上下文的LexicalEnvironment，如果*PropertyAssignment*包含在严格代码里或它的*FunctionBody*是严格代码则为*Strict*指定**true**，令closure是创建一个新Function对象的结果。

3. 令desc是Property Descriptor { [[Set]]:closure, [[Enumerable]]:**true**, [[Configurable]]:**true** }。

4. 返回Property Identifier(propName, desc)。

对于一个包含在严格代码里或它的*FunctionBody*是严格代码的*PropertyAssignment*，如果标识符“**eval**”或“**arguments**”作为*Identifier*出现在它的*PropertySetParameterList*里，这是一个SyntaxError。

产生式*PropertyName* : *IdentifierName*按以下方式求值：

1. 返回和*IdentifierName*包含相同字符序列的String值。

产生式*PropertyName* : *StringLiteral*按以下方式求值：

1. 返回*StringLiteral*的SV。

产生式*PropertyName* : *NumericLiteral*按以下方式求值：

1. 令nbr是计算*NumericLiteral*的值的结果。

【SP79】

2. 返回ToString(nbr)。

11.1.6段 分组运算符

产生式*PrimaryExpression* : **(** *Expression* **)**按以下方式求值：

1. 返回求值*Expression*的结果，结果可能是Reference类型。

注意：该算法并没有对求值*Expression*的结果应用GetValue，最主要的动机是为了使像**delete**和**typeof**这样的运算符可以被应用到带括号的表达式。

11.2节 左手边表达式

语法

*MemberExpression* :

*PrimaryExpression*

*FunctionExpression*

*MemberExpression* **[** *Expression* **]**

*MemberExpression* **.** *IdentifierName*

**new** *MemberExpression Arguments*

*NewExpression* :

*MemberExpression*

**new** *NewExpression*

*CallExpressoin* :

*MemberExpression Arguments*

*CallExpression Arguments*

*CallExpression* **[** *Expression* **]**

*CallExpression* **.** *IdentifierName*

*Arguments* :

**( )**

**(** *ArgumentList* **)**

*ArgumentList* :

*AssignmentExpression*

*ArgumentList* **,** *AssignmentExpression*

*LeftHandSideExpression* :

*NewExpression*

*CallExpression*

11.2.1段 属性存取器

属性根据名称进行存取，使用点号：

*MemberExpression* **.** *IdentifierName*

*CallExpression* **.** *Identifier*

或方括号：

*MemberExpression* **[** *Expression* **]**

*CallExpression* **[** *Expression* **]**

点号根据以下语法转换解释：

*MemberExpression* **.** *IdentifierName*

在行为上与

【SP80】

*MemberExpression* **[** <标识符名称字符串>**]**

一致，类似的，

*CallExpression* **.** *Identifier*

在行为上与

*CallExpression* **[** <标识符名称字符串> **]**

一致，其中<标识符名称字符串>是一个字符串字面值，包含着在处理Unicode转义序列【\u####】之后和*IdentifierName*相同的字符序列。

产生式*MemberExpression* : *MemberExpression* **[** *Expression* **]**按以下方式求值：

1. 令baseReference是求值*MemberExpression*的结果。

2. 令baseValue是GetValue(baseReference)。

3. 令propertyNameReference是求值*Expression*的结果。

4. 令propertyNameValue是GetValue(propertyNameReference)。

5. 调用CheckObjectCoercible(baseValue)。

6. 令propertyNameString是ToString(propertyNameValue)。

7. 如果正被求值的语法产生式包含在严格模式代码里，令strict为**true**，否则令strict为**false**。

8. 返回一个Reference类型的值，它的基值是baseValue，被引用的名称是propertyNameString，严格模式标志是strict。

产生式*CallExpression* : *CallExpression* **[** *Expression* **]**以完全相同的方式求值，除了在第一步里求值的是所含的*CallExpression*。

11.2.2段 **new**运算符

产生式*NewExpression* : **new** *NewExpression*按以下方式求值：

1. 令ref是求值*NewExpression*的结果。

2. 令constructor是GetValue(ref)。

3. 如果Type(constructor)不是Object，抛出一个TypeError异常。

4. 如果constructor没有实现[[Construct]]内部方法，抛出一个TypeError异常。

5. 不传任何实参，返回在constructor上调用[[Construct]]内部方法的结果。

产生式*MemberExpression* : **new** *MemberExpressoin Arguments*按以下方式求值：

1. 令ref是求值*MemberExpressoin*的结果。

2. 令constructor是GetValue(ref)。

3. 令argList是求值Arguments的结果，产生一个实参值的内部列表（11.2.4段）。

4. 如果Type(constructor)不是Object，抛出一个TypeError异常。

5. 如果constructor没有实现[[Construct]]内部方法，抛出一个TypeError异常。

6. 传入argList作为实参，返回在constructor上调用[[Construct]]内部方法的结果。

11.2.3段 函数调用

产生式*CallExpression* : *MemberExpression Arguments*按以下方式求值：

1. 令ref是求值*MemberExpression*的结果。

2. 令func是GetValue(ref)。

3. 令argList是求值Arguments的结果，产生一个实参值的内部列表（11.2.4段）。

4. 如果Type(func)不是Object，抛出一个TypeError异常。

5. 如果IsCallable(func)为**false**，抛出一个TypeError异常。

6. 如果Type(ref)是Reference，那么

a. 如果IsPropertyReference(ref)为**true**，那么

【SP81】

i. 令thisValue是GetBase(ref)。

b. 否则，ref的基值是一个Environment Record，

i. 令thisValue是调用GetBase(ref)的ImplicitThisValue具体方法的结果。

7. 否则，Type(ref)不是Reference，

a. 令thisValue是**undefined**。

8. 以thisValue作为**this**值，传入argList作为实参，返回在func上调用[[Call]]内部方法的结果。

产生式*CallExpression* : *CallExpression Arguments*以完全相同的方式求值，除了在第一步里求值的是所含的*CallExpression*。

注意：如果func是一个原生ECMAScript对象，则返回的结果永远不会是Reference类型。调用一个宿主对象是否会返回一个Reference类型的值取决于实现。如果返回了一个Reference类型的值，它必须是一个非严格属性引用【strict reference分量为**false**】。

11.2.4段 实参列表

实参列表的求值产生一个值的列表（参阅8.8节）。

产生式*Arguments* : **( )**按以下方式求值：

1. 返回一个空列表。

产生式*Arguments* : **(** *ArgumentList* **)**按以下方式求值：

1. 返回求值*ArgumentList*的结果。

产生式*ArgumentList* :*AssignmentExpression*按以下方式求值：

1. 令ref是求值*AssignmentExpression*的结果。

2. 令arg是GetValue(ref)。

3. 返回一个列表，其唯一的元素就是arg。

产生式*ArgumentList* :*ArgumentList* **,** *AssignmentExpression*按以下方式求值：

1. 令precedingArgs是求值*ArgumentList*的结果。

2. 令ref是求值*AssignmentExpression*的结果。

3. 令arg是GetValue(ref)。

4. 返回一个列表，其长度比precedingArgs的长度大1，其元素是precedingArgs的元素，依序排列，后跟arg，它是该列表的最后一个元素。

11.2.5段 函数表达式

产生式*MemberExpression* : *FunctionExpression*按以下方式求值：

1. 返回求值*FunctionExpression*的结果。

11.3节 后缀表达式

语法

*PostfixExpression* :

*LeftHandSideExpression*

*LeftHandSideExpression* [no LineTerminator here] **++**

*LeftHandSideExpression* [no LineTerminator here] **--**

【SP82】

11.3.1段 后缀递增运算符

产生式*PostfixExpression* : *LeftHandSideExpression* [no LineTerminator here] **++**按以下方式求值：

1. 令lhs是求值*LeftHandSideExpression*的结果。

2. 如果下列条件全为真，抛出一个SyntaxError异常：

· Type(lhs)是Reference。

· IsStrictReference(lhs)

· Type(GetBase(lhs))是Environment Record

· GetReferencedName(lhs)是“**eval**”或“**arguments**”。

3. 令oldValue是ToNumber(GetValue(lhs))。

4. 令newValue是oldValue加1的结果，使用和**+**运算符（参阅11.6.3段）相同的规则。

5. 调用PutValue(lhs, newValue)。

6. 返回oldValue。

11.3.2段 后缀递减运算符

产生式*PostfixExpression* : *LeftHandSideExpression* [no LineTerminator here] **--**按以下方式求值：

1. 令lhs是求值*LeftHandSideExpression*的结果。

2. 如果下列条件全为真，抛出一个SyntaxError异常：

· Type(lhs)是Reference。

· IsStrictReference(lhs)

· Type(GetBase(lhs))是Environment Record

· GetReferencedName(lhs)是“**eval**”或“**arguments**”。

3. 令oldValue是ToNumber(GetValue(lhs))。

4. 令newValue是oldValue减1的结果，使用和**-**运算符（参阅11.6.3段）相同的规则。

5. 调用PutValue(lhs, newValue)。

6. 返回oldValue。

11.4节 一元运算符

语法

*UnaryExpression* :

*PostfixExpression*

**delete** *UnaryExpression*

**void** *UnaryExpression*

**typeof** *UnaryExpression*

**++** *UnaryExpression*

**--** *UnaryExpression*

**+** *UnaryExpression*

**-** *UnaryExpression*

**~** *UnaryExpression*

**!** *UnaryExpression*

11.4.1段 **delete**运算符

产生式*UnaryExpression* : **delete** *UnaryExpression*按以下方式求值：

1. 令ref是求值*UnaryExpression*的结果。

2. 如果Type(ref)不是Reference，返回**true**。

3. 如果IsUnresolvableReference(ref)为**true**，那么

a. 如果IsStrictReference(ref)为**true**，抛出一个SyntaxError异常。

b. 否则，返回**true**。

【SP83】

4. 如果IsPropertyReference(ref)为**true**，那么

a. 传入GetReferencedName(ref)和IsStrictReference(ref)作为实参，返回在 ToObject(GetBase(ref))上调用[[Delete]]内部方法的结果。

5. 否则，ref是对一个Environment Record绑定的引用，所以

a. 如果IsStrictReference(ref)为**true**，抛出一个SyntaxError异常。

b. 令bindings是GetBase(ref)。

c. 传入GetReferencedName(ref)作为实参，返回调用bindings的DeleteBinding具 体方法的结果。

注意：当一个**delete**运算符出现在严格模式代码里时，如果它的*UnaryExpression*是对一个变量，函数实参，或函数名的直接引用，则抛出SyntaxError异常。此外，如果一个**delete**运算符出现在严格模式代码里，且将被删除的属性具有特性{ [[Configurable]]:**false** }，则抛出TypeError异常。

11.4.2段 **void**运算符

产生式*UnaryExpression* : **void** *UnaryExpression*按以下方式求值：

1. 令expr是求值*UnaryExpression*的结果。

2. 调用GetValue(expr)。

3. 返回**undefined**。

注意：GetValue必须被调用，即使它的值没有被用到，因为它或许具有可观察的副作用。

11.4.3段 **typeof**运算符

产生式*UnaryExpression* : **typeof** *UnaryExpression*按以下方式求值：

1. 令val是求值*UnaryExpression*的结果。

2. 如果Type(val)是Reference，那么

a. 如果IsUnresolvableReference(val)为**true**，返回"**undefined**"。

b. 令val是GetValue(val)。

3. 返回一个由Type(val) 依照表格20确定的String：

表格20 —— **typeof**运算符结果

┌─────────────────────────────────────┐

│val的类型 结果 │

├─────────────────────────────────────┤

│Undefined "**undefined**" │

│Null "**object**" │

│Boolean "**boolean**" │

│Number "**number**" │

│String "**string**" │

│Object（原生，且未实现[[Call]]） "**object**" │

│Object（原生或宿主，且实现[[Call]]） "**function**" │

│Object（宿主，且未实现[[Call]]） #1 │

└─────────────────────────────────────┘

#1 由实现定义，除了不能是"**undefined**"，"**boolean**"，"**number**"，或"**string**"。

11.4.4段 前缀递增运算符

产生式*UnaryExpression* : **++** *UnaryExpression*按以下方式求值：

1. 令expr是求值*UnaryExpression*的结果。

2. 如果下列条件全为真，抛出一个SyntaxError异常：

【SP84】

· Type(expr)是Reference。

· IsStrictReference(expr)

· Type(GetBase(expr))是Environment Record

· GetReferencedName(expr)是“**eval**”或“**arguments**”。

3. 令oldValue是ToNumber(GetValue(expr))。

4. 令newValue是oldValue加1的结果，使用和**+**运算符（参阅11.6.3段）相同的规则。

5. 调用PutValue(expr, newValue)。

6. 返回newValue。

11.3.2段 后缀递减运算符

产生式*UnaryExpression* : **--** *UnaryExpression*按以下方式求值：

1. 令expr是求值*UnaryExpression*的结果。

2. 如果下列条件全为真，抛出一个SyntaxError异常：

· Type(expr)是Reference。

· IsStrictReference(expr)

· Type(GetBase(expr))是Environment Record

· GetReferencedName(expr)是“**eval**”或“**arguments**”。

3. 令oldValue是ToNumber(GetValue(lhs))。

4. 令newValue是oldValue减1的结果，使用和**-**运算符（参阅11.6.3段）相同的规则。

5. 调用PutValue(lhs, newValue)。

6. 返回newValue。

11.4.6段 一元**+**运算符

一元**+**运算符把它的运算数转换成Number类型。

产生式*UnaryExpression* : **+** *UnaryExpression*按以下方式求值：

1. 令expr是求值*UnaryExpression*的结果。

2. 返回ToNumber(GetValue(expr))。

11.4.7段 一元**-**运算符

一元**-**运算符把它的运算数转换成Number类型，然后取相反数。注意对**+0**取相反数产生**-0**，而对**-0**取相反数产生**+0**。

产生式*UnaryExpression* : **-** *UnaryExpression*按以下方式求值：

1. 令expr是求值*UnaryExpression*的结果。

2. 令oldValue是ToNumber(GetValue(expr))。

3. 如果oldValue为**NaN**，返回**NaN**。

4. 返回对oldValue取相反数的结果；也就是，计算一个Number，具有和oldValue相同的绝对值，但符号相反。

11.4.8段 逐位非运算符（**~**）

产生式*UnaryExpression* : **~** *UnaryExpression*按以下方式求值：

1. 令expr是求值*UnaryExpression*的结果。

2. 令oldValue是ToInt32(GetValue(expr))。

3. 返回对oldValue应用逐位反码的结果，该结果是一个有符号32位整数。

【SP85】

11.4.9段 逻辑非运算符（**！**）

产生式*UnaryExpression* : **!** *UnaryExpression*按以下方式求值：

1. 令expr是求值*UnaryExpression*的结果。

2. 令oldValue是ToBoolean(GetValue(expr))。

3. 如果oldValue为**true**，返回**false**。

4. 返回**true**。

11.5节 乘除运算符

语法

*MultiplicativeExpression* :

*UnaryExpression*

*MultiplicativeExpression* **\*** *UnaryExpression*

*MultiplicativeExpression* **/** *UnaryExpression*

*MultiplicativeExpression* **%** *UnaryExpression*

语义

产生式*MultiplicativeExpression* : *MultiplicativeExpression* **@** *UnaryExpression*，其中**@**表示上述定义里的一个运算符，按以下方式求值：

1. 令left是求值*MultiplicativeExpression*的结果。

2. 令leftValue是GetValue(left)。

3. 令right是求值*UnaryExpression*的结果。

4. 令rightValue是GetValue(right)。

5. 令leftNum是ToNumber(leftValue)。

6. 令rightNum是ToNumber(rightValue)。

7. 返回对leftNum和rightNum应用指定运算（**\***，**/**，或**%**）的结果，参阅11.5.1段，11.5.2段，11.5.3段后面的注意。

11.5.1段 应用**\***运算符

**\***运算符执行乘法，产生它的运算数的积。乘法满足交换律，在ECMAScript里乘法并不总是满足结合律，由于有限的精度。

浮点乘法的结果受IEEE 754二进制双精度算术规则的支配：

·如果任何一个运算数是**NaN**，结果是**NaN**。

·如果两个运算数同号，结果的符号是正号，如果两个运算数异号，结果的符号是负号。

·**Infinity**和**0**的积产生**NaN**。

·**Infinity**和**Infinity**的积产生**Infinity**，符号根据上面陈述的规则确定。

·**Infinity**和有限非**0**值的积产生有符号**Infinity**，符号根据上面陈述的规则确定。

·在剩余情况里，既不涉及**Infinity**也不涉及**NaN**，积被计算出来，并使用IEEE 754舍入到最近模式舍入到最近可表示的值。如果值的大小过大而不能表示，结果则是带有恰当符号的**Infinity**；如果值的大小过小而不能表示，结果则是带有恰当符号的**0**。ECMAScript语言要求支持逐级下溢【gradual underflow】，如IEEE 754所定义。

【SP86】

11.5.2段 应用**/**运算符

**/**运算符执行除法，产生它的运算数的商。左运算数是被除数，右运算数是除数。ECMAScript不执行整数除法。所有除法运算的两个运算数和结果都是双精度浮点数。除法的结果由IEEE 754算术规范确定：

·如果任何一个运算数是**NaN**，结果是**NaN**。

·如果两个运算数同号，结果的符号是正号，如果两个运算数异号，结果的符号是负号。

·**Infinity**除以**Infinity**产生**NaN**。

·**Infinity**除以**0**产生**Infinity**，符号根据上面陈述的规则确定。

·**Infinity**除以非**0**有限值产生有符号**Infinity**，符号根据上面陈述的规则确定。

·有限值除以**Infinity**产生**0**，符号根据上面陈述的规则确定。

·**0**除以**0**产生**NaN**；**0**除以任何其它有限值产生**0**，符号根据上面陈述的规则确定。

·非**0**有限值除以**0**产生有符号**Infinity**，符号根据上面陈述的规则确定。

·在剩余情况里，不涉及**Infinity**，**0**，和**NaN**，商被计算出来，并使用IEEE 754舍入到最近模式舍入到最近可表示的值。如果值的大小过大而不能表示，运算上溢出，结果则是带有恰当符号的**Infinity**；如果值的大小过小而不能表示，运算下溢出，结果则是带有恰当符号的**0**。ECMAScript语言要求支持逐级下溢【gradual underflow】，如IEEE 754所定义。

11.5.3段 应用**%**运算符

**%**运算符从一个隐含的除法产生它的运算数的余数；左运算数是被除数，右运算数是除数。

注意：在C和C++里，求余运算只接受整型运算数；在ECMAScript里，它也接受浮点型运算数。

由**%**运算符计算的浮点求余运算的结果不同于由IEEE 754定义的“求余”运算。IEEE 754“求余”运算从一个舍入除法计算余数，而不是截断除法，因此它的行为并不类似于通常的整型求余运算。相反，ECMAScript语言定义浮点运算上的**%**以一种类似于Java整型求余运算的方式运作；这可以与C库函数fmod相比。

ECMAScript浮点求余运算的结果由IEEE 754算术规则确定：

·如果任何一个运算数是**NaN**，结果是**NaN**。

·结果的符号与被除数的符号相同。

·如果被除数是**Infinity**，或除数是**0**，或两者同时成立，结果是**NaN**。

·如果被除数是有限值，且除数是**Infinity**，结果等于被除数。

·如果被除数是**0**，且除数是非**0**有限值，结果等于**0**。

·在剩余情况里，不涉及**Infinity**，**0**，和**NaN**，从被除数n和除数d产生的浮点余数r由数学关系式r＝n－(d×q)定义，其中q是一个整数，仅当n除以d为负时才为负，仅当n除以d为正时才为正，它的大小【magnitude，可以理解为绝对值】应该尽可能的大，只要不会超出n和d的商的大小。r被计算出来，并使用IEEE 754舍入到最近模式舍入到最近可表示的值。

【SP87】

11.6节 加减运算符

语法

*AdditiveExpression* :

*MultiplicativeExpression*

*AdditiveExpression* **+** *MultiplicativeExpression*

*AdditiveExpression* **-** *MultiplicativeExpression*

11.6.1段 加运算符（**+**）

加运算符要么执行字符串拼接，要么执行数字加法。

产生式*AdditiveExpression* : *AdditiveExpression* **+** *MultiplicativeExpression*按以下方式求值：

1. 令lref是求值*AdditiveExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*MultiplicativeExpression*的结果。

4. 令rval是GetValue(rref)。

5. 令lprim是ToPrimitive(lval)。

6. 令rprim是ToPrimitive(rval)。

7. 如果Type(lprim)是String或Type(rprim)是String，那么

a. 返回作为ToString(lprim)后跟ToString(rprim)拼接结果的String。

8. 返回对ToNumber(lprim)和ToNumber(rprim) 应用加运算符的结果。参阅11.6.3段下面的注意。

注意1：在第5步和第6步调用ToPrimitive时没有提供提示。除了Date对象，所有原生ECMAScript对象应对提示缺失如同给出的提示是Number；Date对象应对提示缺失如同给出的提示是String。宿主对象可以以其它方式应对提示缺失。

注意2：第7步和用于关系运算符的比较算法的第3步不同（11.8.5段），这里使用逻辑或运算，而不是逻辑与运算。

11.6.2段 减运算符（**-**）

产生式产生式*AdditiveExpression* : *AdditiveExpression* **-** *MultiplicativeExpression*按以下方式求值：

1. 令lref是求值*AdditiveExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*MultiplicativeExpression*的结果。

4. 令rval是GetValue(rref)。

5. 令lnum是ToNumber(lval)。

6. 令rnum是ToNumber(rval)。

7. 返回对lnum和rnum 应用减运算符的结果。参阅11.6.3段下面的注意。

11.6.3段 对数字应用加减运算符

当应用到两个数字类型的运算数时，**+**运算符执行加法，产生两个运算数的和；**-**运算符执行减法，产生两个运算数的差。

加法满足交换律，但并不总是满足结合律。

加法的结果使用IEEE 754二进制双精度算术规则确定：

·如果任何一个运算数是**NaN**，结果是**NaN**。

·两个异号**Infinity**之和是**NaN**。

·两个同号**Infinity**之和是**Infinity**,符号不变。

·**Infinity**与一个有限值之和是**Infinity**。

【SP88】

·两个**-0**之和是**-0**，两个**+0**之和是**+0**，两个异号**0**之和是**+0**。

·**0**与一个非**0**有限值之和是那个非**0**有限值。

·两个大小相同、符号相反的非**0**有限值之和是**+0**。

·在剩余情况里，不涉及**Infinity**，**0**，和**NaN**，且两个运算数具有相同的符号或具有不同的大小，和被计算出来，并使用IEEE 754舍入到最近模式舍入到最近可表示的值。如果值的大小过大而不能表示，运算上溢出，结果则是带有恰当符号的**Infinity**。ECMAScript语言要求支持逐级下溢【gradual underflow】，如IEEE 754所定义。

当应用到两个数字类型的运算数时，**-**运算符执行减法，产生两个运算数的差；左运算数是被减数，右运算数是减数。给定数字运算数a和b，a **-** b总是与a **+ (-**b**)**产生相同的结果。

11.7节 位移运算符

语法

*ShiftExpression* :

*AdditiveExpression*

*ShiftExpression* **<<** *AdditiveExpression*

*ShiftExpression* **>>** *AdditiveExpression*

*ShiftExpression* **>>>** *AdditiveExpression*

11.7.1段 左移运算符（**<<**）

在左运算数上执行逐位左移运算，数量由右运算数指定。

产生式*ShiftExpression* : *ShiftExpression* **<<** *AdditiveExpression*按以下方式求值：

1. 令lref是求值*ShiftExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*AdditiveExpression*的结果。

4. 令rval是GetValue(rref)。

5. 令lnum是ToInt32(lval)。

6. 令rnum是ToUint32(rval）。

7. 令shiftCount是掩盖掉rnum除最低5个有效数位外的所有数位的结果，也就是，计算rnum & 0x1F【shiftCount∈[0, 31]整数】。

8. 返回把lnum左移shiftCount个数位的结果。结果是一个有符号32位整数。

11.7.2段 有符号右移运算符（**>>**）

在左运算数上执行符号位填充逐位右移运算，数量由右运算数指定。

产生式*ShiftExpression* : *ShiftExpression* **>>** *AdditiveExpression*按以下方式求值：

1. 令lref是求值*ShiftExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*AdditiveExpression*的结果。

4. 令rval是GetValue(rref)。

5. 令lnum是ToInt32(lval)。

6. 令rnum是ToUint32(rval）。

7. 令shiftCount是掩盖掉rnum除最低5个有效数位外的所有数位的结果，也就是，计算rnum & 0x1F【shiftCount∈[0, 31]整数】。

【SP89】

8. 返回把lnum执行符号扩展右移shiftCount个数位的结果，最高有效数位向右传播。结果是一个有符号32位整数。

11.7.3段 无符号右移运算符（**>>>**）

在左运算数上执行0数位填充逐位右移运算，数量由右运算数指定。

产生式*ShiftExpression* : *ShiftExpression* **>>>** *AdditiveExpression*按以下方式求值：

1. 令lref是求值*ShiftExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*AdditiveExpression*的结果。

4. 令rval是GetValue(rref)。

5. 令lnum是ToInt32(lval)。

6. 令rnum是ToUint32(rval）。

7. 令shiftCount是掩盖掉rnum除最低5个有效数位外的所有数位的结果，也就是，计算rnum & 0x1F【shiftCount∈[0, 31]整数】。

【SP89】

8. 返回把lnum执行0数位填充右移shiftCount个数位的结果，腾出来的数位用0填充。结果是一个有符号32位整数。

11.8节 关系运算符

语法

*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*

注意：“NoIn”变体对于避免混淆关系表达式里的**in**运算符与**for**语句里的**in**运算符是需要的。

语义

求值一个关系运算符的结果总是Boolean类型，反映了在两个运算数之间由运算符命名的关系是否成立。

*RelationalExpressionNoIn*产生式和*RelationalExpression*产生式以相同的方式求值，除了所含的*RelationalExpressionNoIn*将被求值，而不是所含的*RelationalExpression*。

11.8.1段 小于运算符（**<**）

产生式*RelationalExpression* : *RelationalExpression* **<** *ShiftExpression*按以下方式求值：

【SP90】

1. 令lref是求值*RelationalExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*ShiftExpression*的结果。

4. 令rval是GetValue(rref)。

5. 令r是执行抽象关系比较lval<rval的结果（参阅11.8.5段）。

6. 如果r为**undefined**，返回**false**，否则返回r。

11.8.2段 大于运算符（**>**）

产生式*RelationalExpression* : *RelationalExpression* **>** *ShiftExpression*按以下方式求值：

1. 令lref是求值*RelationalExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*ShiftExpression*的结果。

4. 令rval是GetValue(rref)。

5. 为LeftFirst指定**false**，令r是执行抽象关系比较rval<lval的结果（参阅11.8.5段）。

6. 如果r为**undefined**，返回**false**，否则返回r。

11.8.3段 小于等于运算符（**<=**）

产生式*RelationalExpression* : *RelationalExpression* **<=** *ShiftExpression*按以下方式求值：

1. 令lref是求值*RelationalExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*ShiftExpression*的结果。

4. 令rval是GetValue(rref)。

5. 为LeftFirst指定**false**，令r是执行抽象关系比较rval<lval的结果（参阅11.8.5段）。

6. 如果r为**true**或**undefined**，返回**false**，否则返回**true**。

11.8.4段 小于等于运算符（**>=**）

产生式*RelationalExpression* : *RelationalExpression* **>=** *ShiftExpression*按以下方式求值：

1. 令lref是求值*RelationalExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*ShiftExpression*的结果。

4. 令rval是GetValue(rref)。

5. 令r是执行抽象关系比较lval<rval的结果（参阅11.8.5段）。

6. 如果r为**true**或**undefined**，返回**false**，否则返回**true**。

11.8.5段 抽象关系比较算法

比较式x<y，其中x和y是值，产生**true**，**false**或**undefined**（说明至少一个运算数为**NaN**）。除了x和y，算法还接收一个Boolean标志叫做LeftFirst作为参数，该标志被用来控制——带有潜在可见副作用的运算在x和y上执行的——顺序。这是必要的，因为ECMAScript规定表达式从左向右求值。LeftFirst的默认值是**true**，指示x参数对应一个——出现在y参数对应表达式左边的——表达式。如果LeftFirst为**false**，那么情形恰好相反，此时运算必须先在y上执行，而后x。这样的比较按以下方式执行：

1. 如果LeftFirst为**true**，那么

a. 令px是调用ToPrimitive(x, 提示Number)的结果。

b. 令py是调用ToPrimitive(y, 提示Number)的结果。

2. 否则，求值的顺序需要被反转，以保持从左向右求值，

a. 令py是调用ToPrimitive(y, 提示Number)的结果。

b. 令px是调用ToPrimitive(x, 提示Number)的结果。

【SP91】

3. 如果Type(px)与Type(py)不全是String，那么

a. 令nx是调用ToNumber(px)的结果。因为px和py是基元值，求值顺序不重要。

b. 令ny是调用ToNumber(py)的结果。

c. 如果nx是**NaN**，返回**undefined**。

d. 如果ny是**NaN**，返回**undefined**。

e. 如果nx和ny是相同的Number值，返回**false**。

f. 如果nx是**+0**，且ny是**-0**，返回**false**。

g. 如果nx是**-0**，且ny是**+0**，返回**false**。

h. 如果nx是**+∞**，返回**false**。

i. 如果ny是**+∞**，返回**true**。

j. 如果ny是**-∞**，返回**false**。

k. 如果nx是**-∞**，返回**true**。

l. 如果nx的数学值小于ny的数学值——注意这两个数学值都是有限的，且不全为零——返回**true**，否则返回**false**。

4. 否则，px和py都是String值，

a. 如果py是px的一个前缀，返回**false**。（一个String值p是String值q的前缀， 如果q可以是拼接p和其它String值r的结果。注意任何String值是它自身的前缀， 因为r可以是空String。）

b. 如果px是py的一个前缀，返回**true**。

c. 令k是最小非负整数，使得px里位置k处的字符不同于py里位置k处的字符。（对 于不互为前缀的两个String值，一定有这样的k。）

d. 令m是作为px里位置k处的字符的编码单位值的整数。

e. 令n是作为py里位置k处的字符的编码单位值的整数。

f. 如果m<n，返回**true**，否则返回**false**。

注意1：第3步和用于加运算符的算法的第7步不同（11.6.1段），这里使用逻辑与运算，而不是逻辑或运算。

注意2：String值的比较在编码单位值的序列上使用一个简单的字典顺序。这里不会尝试使用更复杂的，面向语义的字符或字符串相等定义，以及在Unicode规范里定义的排字顺序。因此依照Unicode标准正式相等的两个String值可能测试为不相等。事实上该算法假定两个String值已经是归一形式。同样，注意对于含有增补字符的字符串，在UTF-16编码单位值的序列上的字典顺序不同于编码点数值的序列上的字典顺序。

11.8.6段 **instanceof**运算符

产生式*RelationalExpression* : *RelationalExpression* **instanceof** *ShiftExpression*按以下方式求值：

1. 令lref是求值*RelationalExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*ShiftExpression*的结果。

4. 令rval是GetValue(rref)。

5. 如果Type(rval)非**Object**，抛出一个TypeError异常。

6. 如果rval没有[[HasInstance]]内部方法，抛出一个TypeError异常。

7. 传入lval作为实参，返回调用rval的[[HasInstance]]内部方法的结果。

11.8.7段 **in**运算符

产生式*RelationalExpression* : *RelationalExpression* **in** *ShiftExpression*按以下方式求值：

1. 令lref是求值*RelationalExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*ShiftExpression*的结果。

4. 令rval是GetValue(rref)。

5. 如果Type(rval)非**Object**，抛出一个TypeError异常。

6. 传入ToString(lval)作为实参，返回调用rval的[[HasProperty]]内部方法的结果。

【SP92】

11.9节 对等运算符

语法

*EqualityExpression* :

*RelationalExpression*

*EqualityExpression* **==** *RelationalExpression*

*EqualityExpression* **!=** *RelationalExpression*

*EqualityExpression* **===** *RelationalExpression*

*EqualityExpression* **!==** *RelationalExpression*

*EqualityExpressionNoIn* :

*RelationalExpressionNoIn*

*EqualityExpressionNoIn* **==** *RelationalExpressionNoIn*

*EqualityExpressionNoIn* **!=** *RelationalExpressionNoIn*

*EqualityExpressionNoIn* **===** *RelationalExpressionNoIn*

*EqualityExpressionNoIn* **!==** *RelationalExpressionNoIn*

语义

求值一个对等运算符的结果总是Boolean类型，反映了在两个运算数之间由运算符命名的关系是否成立。

*EqualityExpressionNoIn*产生式和*EqualityExpression*产生式以相同的方式求值，除了所含的*EqualityExpressionNoIn*和*RelationalExpressionNoIn*将分别被求值，而不是所含的*EqualityExpression*和*RelationalExpression*。

11.9.1段 相等运算符（**==**）

产生式*EqualityExpression* : *EqualityExpression* **==** *RelationalExpression*按以下方式求值：

1. 令lref是求值*EqualityExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*RelationalExpression*的结果。

4. 令rval是GetValue(rref)。

5. 返回执行抽象相等比较rval==lval的结果（参阅11.9.3段）。

11.9.2段 不等运算符（**!=**）

产生式*EqualityExpression* : *EqualityExpression* **!=** *RelationalExpression*按以下方式求值：

1. 令lref是求值*EqualityExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*RelationalExpression*的结果。

4. 令rval是GetValue(rref)。

5. 令r是执行抽象相等比较rval==lval的结果（参阅11.9.3段）。

6. 如果r为**true**，返回**false**，否则返回**true**。

11.9.3段 抽象相等比较算法

比较式x**==**y，其中x和y是值，产生**true**或**false**。这样一个比较按以下方式执行：

1. 如果Type(x)和Type(y)相同，那么

a. 如果Type(x)为Undefined，返回**true**。

b. 如果Type(x)为Null，返回**true**。

c. 如果Type(x)为Number，那么

i. 如果x为**NaN**，返回**false**。

【SP93】

ii. 如果y为**NaN**，返回**false**。

iii. 如果x和y具有相同的Number值，返回**true**。

iv. 如果x为**+0**，且y为**-0**，返回**true**。

v. 如果x为**-0**，且y为**+0**，返回**true**。

vi. 返回**false**。

d. 如果Type(x)为String，当x和y具有完全相同的字符序列时（长度相同，且在对 应位置的字符也相同）返回**true**，否则返回**false**。

e. 如果Type(x)为Boolean，当x和y都为**true**或都为**false**时返回**true**，否则返 回**false**。

f. 当x和y引用同一个对象时返回**true**，否则返回**false**。

2. 如果x为**null**，且y为**undefined**，返回**true**。

3. 如果x为**undefined**，且y为**null**，返回**true**。

4. 如果Type(x)为Number，且Type(y)为String，返回比较x**==**ToNumber(y)的结果。

5. 如果Type(x)为String，且Type(y)为Number，返回比较ToNumber(x)**==**y的结果。

6. 如果Type(x)为Boolean，返回比较ToNumber(x)**==**y的结果。

7. 如果Type(y)为Boolean，返回比较x**==**ToNumber(y)的结果。

8. 如果Type(x)为String或Number，且Type(y)为Object，返回比较x**==**ToPrimitive(y)的结果。

9. 如果Type(x)为Object，且Type(y)为String或Number，返回比较ToPrimitive(x)**==**y的结果。

10. 返回**false**。

注意1：给定上述相等定义：

·可以强制执行字符串比较：**"" +** a **== "" +** b。

·可以强制执行数字比较：**+**a **== +**b。

·可以强制执行布尔比较：**!**a **== !**b。

注意2：对等运算符保持如下不变式：

·A **!=** B 等价于 **! (**A**==**B**)**。

·A **==** B 等价于 B **==** A，除了对A和B求值的顺序不同。

注意3：对等运算符并不总是可传递的。例如，可能有两个不同的String对象，每个都表示相同的String值；若使用**==**运算符比较，每个String对象将被认为等于该String值，但是两个String对象彼此却不相等，如下：

·**new** String**("**a**")** **== "**a**"** 和 **"**a**" == new** String**("**a**")** 都是**true**。

·**new** String**("**a**") == new** String**("**a**")** 却是**false**。

注意4：String值的比较在编码单位值的序列上使用一个简单的相等测试。这里不会尝试使用更复杂的，面向语义的字符或字符串相等定义，以及在Unicode规范里定义的排字顺序。因此依照Unicode标准正式相等的两个String值可能测试为不相等。事实上该算法假定两个String值已经是归一形式。

11.9.4段 严格相等运算符（**===**）

产生式*EqualityExpression* : *EqualityExpression* **===** *RelationalExpression*按以下方式求值：

1. 令lref是求值*EqualityExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*RelationalExpression*的结果。

4. 令rval是GetValue(rref)。

5. 返回执行严格相等比较rval===lval的结果（参阅11.9.6段）。

11.9.5段 严格不等运算符（**!==**）

产生式*EqualityExpression* : *EqualityExpression* **!==** *RelationalExpression*按以下方式求值：

1. 令lref是求值*EqualityExpression*的结果。

【SP94】

2. 令lval是GetValue(lref)。

3. 令rref是求值*RelationalExpression*的结果。

4. 令rval是GetValue(rref)。

5. 令r是执行严格相等比较rval===lval的结果（参阅11.9.6段）。

6. 如果r为**true**，返回**false**，否则返回**true**。

11.9.6段 严格相等比较算法

比较式x**===**y，其中x和y是值，产生**true**或**false**。这样一个比较按以下方式执行：

1. 如果Type(x)和Type(y)不同，返回**false**。

2. 如果Type(x)为Undefined，返回**true**。

3. 如果Type(x)为Null，返回**true**。

4. 如果Type(x)为Number，那么

a. 如果x为**NaN**，返回**false**。

b. 如果y为**NaN**，返回**false**。

c. 如果x和y具有相同的Number值，返回**true**。

d. 如果x为**+0**，且y为**-0**，返回**true**。

e. 如果x为**-0**，且y为**+0**，返回**true**。

f. 返回**false**。

5. 如果Type(x)为String，当x和y具有完全相同的字符序列时（长度相同，且在对应位置的字符也相同）返回**true**，否则返回**false**。

6. 如果Type(x)为Boolean，当x和y都为**true**或都为**false**时返回**true**，否则返回**false**。

7. 当x和y引用同一个对象时返回**true**，否则返回**false**。

注意：该算法在处理有符号零和非数字方面与SameValue算法（9.12节）不同。

11.10节 二元位运算符

语法

*BitwiseANDExpression* :

*EqualityExpression*

*BitwiseANDExpression* **&** *EqualityExpression*

*BitwiseANDExpressionNoIn* :

*EqualityExpressionNoIn*

*BitwiseANDExpressionNoIn* **&** *EqualityExpressionNoIn*

*BitwiseXORExpression* :

*BitwiseANDExpression*

*BitwiseXORExpression* **^** *BitwiseANDExpression*

*BitwiseXORExpressionNoIn* :

*BitwiseANDExpressionNoIn*

*BitwiseXORExpressionNoIn* **^** *BitwiseANDExpressionNoIn*

*BitwiseORExpression* :

*BitwiseXORExpression*

*BitwiseORExpression* **|** *BitwiseXORExpression*

*BitwiseORExpressionNoIn* :

*BitwiseXORExpressionNoIn*

*BitwiseORExpressionNoIn* **|** *BitwiseXORExpressionNoIn*

语义

产生式 *A* : *A* **@** *B*，其中**@**是上述产生式里的位运算符之一，按以下方式求值：

【SP95】

1. 令lref是求值*A*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*B*的结果。

4. 令rval是GetValue(rref)。

5. 令lnum是ToInt32(lval)。

6. 令rnum是ToInt32(rval)。

7. 返回对lnum和rnum应用位运算符**@**的结果，结果是一个有符号32位整数。

11.11节 二元逻辑运算符

语法

*LogicalANDExpression* :

*BitwiseORExpression*

*LogicalANDExpression* **&&** *BitwiseORExpression*

*LogicalANDExpressionNoIn* :

*BitwiseORExpressionNoIn*

*LogicalANDExpressionNoIn* **&&** *BitwiseORExpressionNoIn*

*LogicalORExpression* :

*LogicalANDExpression*

*LogicalORExpressoin* **||** *LogicalANDExpression*

*LogicalORExpressionNoIn* :

*LogicalANDExpressionNoIn*

*LogicalORExpressoinNoIn* **||** *LogicalANDExpressionNoIn*

语义

产生式*LogicalANDExpression* : *LogicalANDExpression* **&&** *BitwiseORExpression*按以下方式求值：

1. 令lref是求值*LogicalANDExpression*的结果。

2. 令lval是GetValue(lref)。

3. 如果ToBoolean(lval)为**false**，返回lval。

4. 令rref是求值*BitwiseORExpression*的结果。

5. 返回GetValue(rref)。

产生式*LogicalORExpressoin* : *LogicalORExpressoin* **||** *LogicalANDExpression*按以下方式求值：

1. 令lref是求值*LogicalORExpressoin*的结果。

2. 令lval是GetValue(lref)。

3. 如果ToBoolean(lval)为**true**，返回lval。

4. 令rref是求值*LogicalANDExpressoin*的结果。

5. 返回GetValue(rref)。

*LogicalANDExpressionNoIn*与*LogicalORExpressionNoIn*产生式，和*LogicalANDExpression*与*LogicalORExpression*产生式，分别以相同的方式求值，除了所含的*LogicalANDExpressionNoIn*，*BitwiseORExpressionNoIn*，与*LogicalORExpressionNoIn*将分别被求值，而不是所含的*LogicalANDExpression*，*BitwiseORExpression*，与*LogicalORExpression*。

注意：由**&&**或**||**运算符产生的值并不一定是Boolean类型，产生的值总是两个运算数表达式之一的值。

【SP96】

11.12节 条件运算符（**? :**）

语法

*ConditionalExpression* :

*LogicalORExpression*

*LogicalORExpression* **?** *AssignmentExpression* **:** *AssignmentExpression*

*ConditionalExpressionNoIn* :

*LogicalORExpressionNoIn*

*LogicalORExpressionNoIn* **?** *AssignmentExpression* **:** *AssignmentExpressionNoIn*

语义

产生式*ConditionalExpression* : *LogicalORExpression* **?** *AssignmentExpression* **:** *AssignmentExpression*按以下方式求值：

1. 令lref是求值*LogicalORExpression*的结果。

2. 如果ToBoolean(GetValue(lref))为**true**，那么

a. 令trueRef是求值第一个*AssignmentExpression*的结果。

b. 返回GetValue(trueRef)。

3. 否则

a. 令falseRef是求值第二个*AssignmentExpression*的结果。

b. 返回GetValue(falseRef)。

*ConditionalExpressionNoIn*产生式和*ConditionalExpression*产生式以相同的方式求值，除了所含的*LogicalORExpressionNoIn*，*AssignmentExpression*，与*AssignmentExpressionNoIn*将分别被求值，而不是所含的*LogicalORExpression*，第一个*AssignmentExpression*，与第二个*AssignmentExpression*。

注意：在EMCAScript里，*ConditionalExpression*的文法与C和Java里的稍微有些不同，后两者都允许第二个子表达式是一个*Expression*，但限制第三个表达式必须是一个*ConditionalExpression*。在EMCAScript里的这一差异的动机是允许一个赋值表达式受到一个条件的任一分支的支配，并消除一个逗号表达式作为中间表达式所带来的令人困惑和完全无用的情形。

11.13节 赋值运算符

语法

*AssignmentExpression* :

*ConditionalExpression*

*LeftHandSideExpression* **=** *AssignmentExpression*

*LeftHandSideExpression AssignmentOperator AssignmentExpression*

*AssignmentExpressionNoIn* :

*ConditionalExpressionNoIn*

*LeftHandSideExpression* **=** *AssignmentExpressionNoIn*

*LeftHandSideExpression AssignmentOperator AssignmentExpressionNoIn*

*AssignmentOperator* : **其中之一**

**\*= /= %= += -=**

**<<= >>= >>>=**

**&= ^= |=**

语义

*AssignmentExpressionNoIn*产生式和*AssignmentExpression*产生式以相同的方式求值，除了所含的*ConditionalExpressionNoIn*，与*AssignmentExpressionNoIn*将分别被求值，而不是所含的*ConditionalExpression*，与*AssignmentExpression*。

【SP97】

11.13.1段 简单赋值（**=**）

产生式*AssignmentExpression* : *LeftHandSideExpression* **=** *AssignmentExpression*按以下方式求值：

1. 令lref是求值*LeftHandSideExpression*的结果。

2. 令rref是求值*AssignmentExpression*的结果。

3. 令rval是GetValue(rref)。

4. 如果下列条件全为真，抛出一个SyntaxError异常：

· Type(lref)是Reference。

· IsStrictReference(lref)

· Type(GetBase(lref))是Environment Record

· GetReferencedName(lref)是“**eval**”或“**arguments**”。

5. 调用PutValue(lref, rval)。

6. 返回rval。

注意：当一个赋值出现在严格模式代码里时，它的*LeftHandSide*绝不能求值为一个不可解析的引用，否则一个ReferenceError异常将在赋值期间被抛出。*LeftHandSide*也不能是一个——对带有特性值{ [[Writable]]:**false** }的数据属性的——引用，也不能是一个——对带有特性值{ [[Set]]:**undefined** }的存取器属性的——引用，也不能是一个——对[[Extensible]]内部属性为**false**的对象的尚不存在的属性的——引用；在这些情况下，一个TypeError异常被抛出。

11.13.2段 复合赋值（**op=**）

产生式*AssignmentExpression* : *LeftHandSideExpression AssignmentOperator AssignmentExpression*，其中*AssignmentOperator*是**@=**，而**@**表示上面指定的运算符之一，按以下方式求值：

1. 令lref是求值*LeftHandSideExpression*的结果。

2. 令lval是GetValue(lref)。

3. 令rref是求值*AssignmentExpression*的结果。

4. 令rval是GetValue(rref)。

5. 令r是对lval和rval应用运算符**@**的结果。

6. 如果下列条件全为真，抛出一个SyntaxError异常：

· Type(lref)是Reference。

· IsStrictReference(lref)

· Type(GetBase(lref))是Environment Record

· GetReferencedName(lref)是“**eval**”或“**arguments**”。

7. 调用PutValue(lref, r)。

8. 返回r。

注意：参阅11.13.1段的注意。

11.14节 逗号运算符（**,**）

语法

*Expression* :

*AssignmentExpression*

*Expression* **,** *AssignmentExpression*

*ExpressionNoIn* :

*AssignmentExpressionNoIn*

*ExpressionNoIn* **,** *AssignmentExpressionNoIn*

语义

产生式*Expression* : *Expression* **,** *AssignmentExpression*按以下方式求值：

1. 令lref是求值*Expression*的结果。

2. 调用GetValue(lref)。

【SP98】

3. 令rref是求值*AssignmentExpression*的结果。

4. 返回GetValue(rref)。

*ExpressionNoIn*产生式和*Expression*产生式以相同的方式求值，除了所含的*ExpressionNoIn*，与*AssignmentExpressionNoIn*将分别被求值，而不是所含的*Expression*，与*AssignmentExpression*。

注意：GetValue必须被调用，即使它的值没有被用到，因为它或许具有可观察的副作用。

第12章 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*可以是一个*LabelledStatement*的一部分，后者本身又可以是一个*LabelledStatement*的一部分，如此等等。当描述各个语句的语义时，以这种方式引入的标号全体被称作“当前标号集”。一个*LabelledStatement*除了把一个标号引入到标号集之外，没有语义含义。一个*IterationStatement*或*SwitchStatement*的标号集初始包含单一元素**empty**；任何其它语句的标号集初始为空。

求值一个*Statement*的结果总是一个Completion值。

注意：已知几个广泛使用的ECMAScript实现支持把*FunctionDeclaration*用作一个*Statement*。然而，那些实现在应用到这些*FunctionDeclaration*的语义方面有一些重要的、不可调和的差异。由于这些不可调和的差异，把*FunctionDeclaration*用作一个*Statement*会导致代码不能可靠的在实现之间移植。推荐ECMAScript实现要么禁止*FunctionDeclaration*的这种用法，要么当遇到这种用法时发出一个警告。ECMAScript的将来版本可能会为在一个*Statement*上下文里声明函数，定义供选的可移植方法。

12.1节 块

语法

*Block* :

**{** *StatementList* opt **}**

*StatementList* :

*Statement*

*StatementList Statement*

【SP99】

语义

产生式*Block* : **{ }**按以下方式求值：

1. 返回(**normal**, **empty**, **empty**)。

产生式*Block* : **{** *StatementList* **}**按以下方式求值：

1. 返回求值*StatementList*的结果。

产生式*StatementList* : *Statement*按以下方式求值：

1. 令s是求值*Statement*的结果。

2. 如果一个异常被抛出，返回(**throw**, V, **empty**)，其中V是那个异常。（执行现在可以继续，就好像没有异常被抛出。）

3. 返回s。

产生式*StatementList* : *StatementList Statement*按以下方式求值：

1. 令sl是求值*StatementList*的结果。

2. 如果sl是一个abrupt completion，返回sl。

3. 令s是求值*Statement*的结果。

4. 如果一个异常被抛出，返回(**throw**, V, **empty**)，其中V是那个异常。（执行现在可以继续，就好像没有异常被抛出。）

5. 如果s.value为**empty**，令V=sl.value，否则令V=s.value。

6. 返回(s.type, V, s.target)。

注意：上述算法的第5步和第6步确保一个*StatementList*的值是在该*StatementList*里最后一个产生值的*Statement*产生的那个值。例如，以下对**eval**函数的调用全都返回值1：

**eval("1;;;;;")**

**eval("1;{ }")**

**eval("1;var a;")**

12.2节 变量语句

语法

*VariableStatement* :

**var** *VariableDeclarationList* **;**

*VariableDeclarationList* :

*VariableDeclaration*

*VariableDeclarationList* **,** *VariableDeclaration*

*VariableDeclarationListNoIn* :

*VariableDeclarationNoIn*

*VariableDeclarationListNoIn* **,** *VariableDeclarationNoIn*

*VariableDeclaration* :

*Identifier Initialiser* opt

*VariableDeclarationNoIn* :

*Identifier InitialiserNoIn* opt

*Initialiser* :

**=** *AssignmentExpression*

*InitialiserNoIn* :

**=** *AssignmentExpressionNoIn*

【SP100】

一个变量语句声明了依照10.5节定义所创建的变量。当创建时变量被初始化为**undefined**。一个带有*Initialiser*的变量在当*VariableStatement*执行时被赋予它的*AssignmentExpression*的值，而不是当变量被创建时。

语义

产生式*VariableStatement* : **var** *VariableDeclarationList* **;**按以下方式求值：

1. 求值*VariableDeclarationList*。

2. 返回(**normal**, **empty**, **empty**)。

产生式*VariableDeclarationList* : *VariableDeclaration*按以下方式求值：

1. 求值*VariableDeclaration*。

产生式*VariableDeclarationList* : *VariableDeclarationList* **,**  *VariableDeclaration*按以下方式求值：

1. 求值*VariableDeclarationList*。

2. 求值*VariableDeclaration*。

产生式*VariableDeclaration* : *Identifier*按以下方式求值：

1. 返回一个和*Identifier*含有相同字符序列的String值。

产生式*VariableDeclaration* : *Identifier Initialiser*按以下方式求值：

1. 令lhs是求值*Identifier*的结果，如11.1.2段所述。

2. 令rhs是求值*Initialiser*的结果。

3. 令value是GetValue(rhs)。

4. 调用PutValue(lhs, value)。

5. 返回一个和*Identifier*含有相同字符序列的String值。

注意：一个*VariableDeclaration*的String值在for-in语句的求值里使用（12.6.4段）。

如果一个*VariableDeclaration*内嵌在一个**with**语句里，且该*VariableDeclaration*里的*Identifier*和**with**语句的对象环境记录的绑定对象的一个属性名相同，那么第4步将把值赋予那个属性，而不是赋予*VariableEnvironment*的*Identifier*绑定。

产生式*Initialiser* : **=** *AssignmentExpression*按以下方式求值：

1. 返回求值*AssignmentExpression*的结果。

*VariableDeclarationListNoIn*，*VariableDeclarationNoIn*，与*InitialiserNoIn*产生式和*VariableDeclarationList*，*VariableDeclaration*，与*Initialiser*产生式以相同的方式求值，除了所含的*VariableDeclarationListNoIn*，*VariableDeclarationNoIn*，*InitialiserNoIn*，与*AssignmentExpressionNoIn*将分别被求值，而不是所含的*VariableDeclarationList*，*VariableDeclaration*，*Initialiser*，与*AssignmentExpression*。

12.2.1段 严格模式限制

如果一个*VariableDeclaration*或*VariableDeclarationNoIn*出现在严格代码里，且它的*Identifier*要么是“**eval**”，要么是“**arguments**”，则是一个SyntaxError。

12.3节 空语句

语法

*EmptyStatement* :

**;**

【SP101】

语义

产生式*EmptyStatement* : **;**按以下方式求值：

1. 返回(**normal**, **empty**, **empty**)。

12.4节 表达式语句

语法

*ExpressionStatement* :

[lookahead ∉ { **{**, **function** }] *Expression* **;**

注意：*ExpressionStatement*不能以左花括号开头，因为那样可能与*Block*产生歧义；也不能以**function**开头，因为那样可能与*FunctionDeclaration*产生歧义。

语义

产生式*ExpressionStatement* : [lookahead ∉ { **{**, **function** }] *Expression* **;**按以下方式求值：

1. 令exprRef是求值*Expression*的结果。

2. 返回(**normal**, GetValue(exprRef), **empty**)。

12.5节 **if**语句

语法

*IfStatement* :

**if (** *Expression* **) else** *Statement*

**if (** *Expression* **)** *Statement*

对于每个在关联**if**的选择上带有歧义的**else**，应该与最近的、尚无对应**else**的**if**相关联。

语义

产生式*IfStatement* : **if (** *Expression* **) else** *Statement*按以下方式求值：

1. 令exprRef是求值*Expression*的结果。

2. 如果ToBoolean(GetValue(exprRef))为**true**，那么

a. 返回求值第一个*Statement*的结果。

3. 否则，

a. 返回求值第二个*Statement*的结果。

产生式*IfStatement* : **if (** *Expression* **)** *Statement*按以下方式求值：

1. 令exprRef是求值*Expression*的结果。

2. 如果ToBoolean(GetValue(exprRef))为**false**，返回(**normal**, **empty**, **empty**)。

3. 返回求值*Statement*的结果。

【SP102】

12.6节 迭代语句

语法

*Iteration* :

**do** *Statement* **while (** *Expression* **) ;**

**while (** *Expression* **)** *Statement*

**for (** *ExpressionNoIn* opt **;** *Expression* opt **;** *Expression* opt **)** *Statement*

**for ( var** *VariableDeclarationListNoIn* **;** *Expression* opt **;** *Expression* opt **)** *Statement*

**for (** *LeftHandSideExpression* **in** *Expressoin* **)** *Statement*

**for ( var** *VariableDeclarationNoIn* **in** *Expression* **)** *Statement*

12.6.1段 **do - while**语句

产生式*Iteration* : **do** *Statement* **while (** *Expression* **) ;**按以下方式求值：

1. 令V=**empty**。

2. 令iteration是**true**。

3. 重复，在iteration为**true**时

a. 令stmt是求值*Statement*的结果。

b. 如果stmt.value非**empty**，令V=stmt.value。

c. 如果stmt.type非**continue**，或stmt.target不在当前标号集里，那么

i. 如果stmt.type为**break**，且stmt.target在当前标号集里，返回(**normal**, V, **empty**)。

ii. 如果stmt是一个“abrupt completion”，返回stmt。

d. 令exprRef是求值*Expression*的结果。

e. 如果ToBoolean(GetValue(exprRef))为**false**，把iterating设为**false**。

4. 返回(**normal**, V, **empty**)。

12.6.2段 **while**语句

产生式*Iteration* : **while (** *Expression* **)** *Statement*按以下方式求值：

1. 令V=**empty**。

2. 重复，

a. 令exprRef是求值*Expression*的结果。

b. 如果ToBoolean(GetValue(exprRef))为**false**，返回(**normal**, V, **empty**)。

c. 令stmt是求值*Statement*的结果。

d. 如果stmt.value非**empty**，令V=stmt.value。

e. 如果stmt.type非**continue**，或stmt.target不在当前标号集里，那么

i. 如果stmt.type为**break**，且stmt.target在当前标号集里，那么

1. 返回(**normal**, V, **empty**))。

ii. 如果stmt是一个“abrupt completion”，返回stmt。

12.6.3段 **for**语句

产生式*IterationStatement* : **for (** *ExpressionNoIn* opt **;** *Expression* opt **;** *Expression* opt **)** *Statement*按以下方式求值：

1. 如果*ExpressionNoIn*存在，那么

a. 令exprRef是求值*ExpressionNoIn*的结果。

b. 调用GetValue(exprRef)。（这个值不会被用到，但该调用可能具有副作用。）

2. 令V=**empty**。

3. 重复，

a. 如果第一个*Expression*存在，那么

i. 令testExprRef是求值第一个*Expression*的结果。

ii. 如果ToBoolean(GetValue(testExprRef))为**false**，返回(**normal**, V, **empty**)。

b. 令stmt是求值*Statement*的结果。

【SP103】

c. 如果stmt.value非**empty**，令V=stmt.value。

d. 如果stmt.type为**break**，且stmt.target在当前标号集里，返回(**normal**, V, **empty**)。

e. 如果stmt.type非**continue**，或stmt.target不在当前标号集里，那么

i. 如果stmt是一个“abrupt completion”，返回stmt。

f. 如果第二个*Expression*存在，那么

i. 令incExprRef是求值第二个*Expression*的结果。

ii. 调用GetValue(incExprRef)。（这个值不会被用到。）

产生式*IterationStatement* : **for ( var** *VariableDeclarationListNoIn* **;** *Expression* opt **;** *Expression* opt **)** *Statement*按以下方式求值：

1. 求值*VariableDeclarationListNoIn*。

2. 令V=**empty**。

3. 重复，

a. 如果第一个*Expression*存在，那么

i. 令testExprRef是求值第一个*Expression*的结果。

ii. 如果ToBoolean(GetValue(testExprRef))为**false**，那么返回(**normal**, V, **empty**)。

b. 令stmt是求值*Statement*的结果。

c. 如果stmt.value非**empty**，令V=stmt.value。

d. 如果stmt.type为**break**，且stmt.target在当前标号集里，返回(**normal**, V, **empty**)。

e. 如果stmt.type非**continue**，或stmt.target不在当前标号集里，那么

i. 如果stmt是一个“abrupt completion”，返回stmt。

f. 如果第二个*Expression*存在，那么

i. 令incExprRef是求值第二个*Expression*的结果。

ii. 调用GetValue(incExprRef)。（这个值不会被用到。）

12.6.4段 **for - in**语句

产生式*IterationStatement* : **for (** *LeftHandSideExpression* **in** *Expressoin* **)** *Statement*按以下方式求值：

1. 令exprRef是求值*Expression*的结果。

2. 令experValue是GetValue(exprRef)。

3. 如果experValue为**null**或**undefined**，返回(**normal**, **empty**, **empty**)。

4. 令obj是ToObject(experValue)。

5. 令V=**empty**。

6. 重复，

a. 令P是obj的下一个[[Enumerable]]特性为**true**的属性的名称；如果没有这样的 属性，返回(**normal**, V, **empty**)。

b. 令lhsRef是求值*LeftHandSideExpression*的结果（它可能会被重复地求值）。

c. 调用PutValue(lhsRef, P)。

d. 令stmt是求值*Statement*的结果。

e. 如果stmt.value非**empty**，令V=stmt.value。

f. 如果stmt.type为**break**，且stmt.target在当前标号集里，返回(**normal**, V, **empty**)。

g. 如果stmt.type非**continue**，或stmt.target不在当前标号集里，那么

i. 如果stmt是一个“abrupt completion”，返回stmt。

产生式*IterationStatement* : **for ( var** *VariableDeclarationNoIn* **in** *Expression* **)** *Statement*按以下方式求值：

1. 令varName是求值*VariableDeclarationNoIn*的结果。

2. 令exprRef是求值*Expression*的结果。

3. 令experValue是GetValue(exprRef)。

4. 如果experValue为**null**或**undefined**，返回(**normal**, **empty**, **empty**)。

5. 令obj是ToObject(experValue)。

6. 令V=**empty**。

【SP104】

7. 重复，

a. 令P是obj的下一个[[Enumerable]]特性为**true**的属性的名称；如果没有这样的 属性，返回(**normal**, V, **empty**)。

b. 令varRef是求值*varName*的结果，就好像它是一个Identifier Reference（11.1.2 段）；它可能会被重复地求值。

c. 调用PutValue(varRef, P)。

d. 令stmt是求值*Statement*的结果。

e. 如果stmt.value非**empty**，令V=stmt.value。

f. 如果stmt.type为**break**，且stmt.target在当前标号集里，返回(**normal**, V, **empty**)。

g. 如果stmt.type非**continue**，或stmt.target不在当前标号集里，那么

i. 如果stmt是一个“abrupt completion”，返回stmt。

列举属性的机制和顺序（第一个算法里的步骤6.a，第二个算法里的步骤7.a）没有被规定。被列举的对象属性可以在列举期间删除。如果一个属性在列举期间还没有被访问时就被删除了，那么它将不会被访问。如果新的属性被添加到正被列举的对象上，新添加的属性不被保证能在进行中的列举中访问到。在任何列举中，一个属性名绝不能被访问多于一次。

列举一个对象的属性包括列举它的原型的属性，以及它的原型的原型的属性，如此等等；但是一个原型的属性不会被列举，如果它被“遮蔽”，即在原型链里某个更靠前的对象具有同名属性。当判定一个原型的属性是否被原型链里更靠前的对象遮蔽时，不会考虑[[Enumerable]]特性值。

注意：参阅11.13.1段的注意。

12.7节 **continue**语句

语法

*ContinueStatement* :

**continue ;**

**continue** [no *LineTerminator* here]*Identifier* **;**

语义

如果下列条件中的任何一个为真，程序就被认为在语法上是不正确的：

·程序包含一个不带可选的*Identifier*的**continue**语句，它没有内嵌于，直接或间接（但不会跨越函数边界），一个*IterationStatement*里。

·程序包含一个带有可选的*Identifier*的**continue**语句，但*Identifier*没有出现在一个封闭的（但不会跨越函数边界的）*IterationStatement*的标号集里。

一个不带*Identifier*的*ContinueStatement*按以下方式求值：

1. 返回(**continue**, **empty**, **empty**)。

一个带有可选的*Identifier*的*ContinueStatement*按以下方式求值：

1. 返回(**continue**, **empty**, *Identifier*)。

【SP105】

12.8节 **break**语句

语法

*BreakStatement* :

**break ;**

**break** [no *LineTerminator* here] *Identifier* **;**

语义

如果下列条件中的任何一个为真，程序就被认为在语法上是不正确的：

·程序包含一个不带可选的*Identifier*的**break**语句，它没有内嵌于，直接或间接（但不会跨越函数边界），一个*IterationStatement*或*SwitchStatement*里。

·程序包含一个带有可选的*Identifier*的**break**语句，但*Identifier*没有出现在一个封闭的（但不会跨越函数边界的）*Statement*的标号集里。

一个不带*Identifier*的*BreakStatement*按以下方式求值：

1. 返回(**break**, **empty**, **empty**)。

一个带有可选的*Identifier*的*BreakStatement*按以下方式求值：

1. 返回(**break**, **empty**, *Identifier*)。

12.9节 **return**语句

语法

*ReturnStatement* :

**return ;**

**return** [no *LineTerminator* here] *Expression* **;**

语义

如果一个ECMAScript程序含有一个不在*FunctionBody*里的**return**语句，就被认为在语法上是不正确的。一个**return**语句使得一个函数终止执行，并把一个值返回给调用者。如果*Expression*被省略，返回值是**undefined**；否则，返回值就是*Expression*的值。

*ReturnStatement*按以下方式求值：

1. 如果*Expression*不存在，返回(**return**, **undefined**, **empty**)。

2. 令exprRef是求值*Expression*的结果。

3. 返回(**return**, GetValue(exprRef), **empty**)。

12.10节 **with**语句

语法

*WithStatement* :

**with (** *Expression* **)** *Statement*

**with**语句把一个用于算出的对象【computed object】的对象环境记录添加到当前执行上下文的LexicalEnvironment里。然后它使用这个扩充的LexicalEnvironment执行语句。最后，它恢复了原先的LexicalEnvironment。

【SP106】

语义

产生式*WithStatement* : **with (** *Expression* **)** *Statement*按以下方式求值：

1. 令val是求值*Expression*的结果。

2. 令obj是ToObject(GetValue(val))。

3. 令oldEnv是当前运作中的执行上下文的LexicalEnvironment。

4. 传入obj和oldEnv作为实参，令newEnv是调用NewObjectEnvironment的结果。

5. 把newEnv的provideThis标志设为**true**。

6. 把当前运作中的执行上下文的LexicalEnvironment设为newEnv。

7. 令C是求值*Statement*的结果，但如果在求值期间一个异常被抛出，令C是(**throw**, V, **empty**)，其中V是那个异常。（执行现在可以继续，就好像没有异常被抛出。）

8. 把当前运作中的执行上下文的LexicalEnvironment设为oldEnv。

9. 返回C。

注意：无论控制是如何离开内嵌的*Statement*的，正常地，或某种形式的“abrupt completion”，或异常，LexicalEnvironment总是被恢复到它的先前状态。

12.10.1段 严格模式限制

严格模式代码不能包含*WithStatement*，在这种上下文里出现*WithStatement*将作为一个SyntaxError处理。

12.11节 **switch**语句

语法

*SwitchStatement* :

**switch (** *Expression* **)** *CaseBlock*

*CaseBlock* :

**{** *CaseClauses* opt **}**

**{** *CaseClauses* opt*DefaultClause CaseClauses* opt **}**

*CasClauses* :

*CaseClause*

*CaseClauses CaseClause*

*CaseClause* :

**case** *Expression* **:** *StatementList* opt

*DefaultClause* :

**default :** *StatementList* opt

语义

产生式*SwitchStatement* : **switch (** *Expression* **)** *CaseBlock*按以下方式求值：

1. 令exprRef是求值*Expression*的结果。

2. 为input指定GetValue(exprRef)，令R是求值*CaseBlock*的结果。

3. 如果R.type为**break**，且R.target在当前标号集里，返回(**normal**, R.value, **empty**)。

4. 返回R。

产生式*CaseBlock* : **{** *CaseClauses* opt **}**接受一个输入参数input，并按以下方式求值：

1. 令V=**empty**。

2. 令A是*CaseClause*项目的列表，以源文本顺序排列。

3. 令searching是**true**。

4. 重复，当searching为**true**时

a. 令C是A里的下一个*CaseClause*，如果没有这样的*CaseClause*，返回(**normal**, V, **empty**)。

【SP107】

b. 令clauseSelector是求值C的结果。

c. 如果根据**===**运算符，input等于clauseSelector，那么

i. 把searching设为**false**。

ii. 如果C有一个*StatementList*，那么

1. 令R是求值C的*StatementList*的结果。

2. 如果R是一个“abrupt completion”，那么返回R。

3. 令V=R.value。

5. 重复，

a. 令C是A里的下一个*CaseClause*，如果没有这样的*CaseClause*，返回(**normal**, V, **empty**)。

b. 如果C有一个*StatementList*，那么

i. 令R是求值C的*StatementList*的结果。

ii. 如果R.value非**empty**，那么令V=R.value。

iii. 如果R是一个“abrupt completion”，那么返回(R.type, V, R.target)。

产生式*CaseBlock* : **{** *CaseClauses* opt*DefaultClause CaseClauses* opt **}**接受一个输入参数input，并按以下方式求值：

1. 令V=**empty**。

2. 令A是在第一个*CaseClauses*里的*CaseClause*项目的列表，以源文本顺序排列。

3. 令B是在第二个*CaseClauses*里的*CaseClause*项目的列表，以源文本顺序排列。

4. 令found是**false**。

5. 反复令C依次取A里的每一个*CaseClause*，

a. 如果found为**false**，那么

i. 令clauseSelector是求值C的结果。

ii. 如果根据**===**运算符，input等于clauseSelector，那么把found设为**true**。

b. 如果found为**true**，那么

i. 如果C有一个*StatementList*，那么

1. 令R是求值C的*StatementList*的结果。

2. 如果R.value非**empty**，那么令V=R.value。

3. 如果R是一个“abrupt completion”，那么返回(R.type, V, R.target)。

6. 令foundInB是**false**。

7. 如果found为**false**，那么

a. 重复，当foundInB为**false**，且B的全部元素还没有被处理时，

i. 令C是B里的下一个*CaseClause*。

ii. 令clauseSelector是求值C的结果。

iii. 如果根据**===**运算符，input等于clauseSelector，那么

1. 把foundInB设为**true**。

2. 如果C有一个*StatementList*，那么

a. 令R是求值C的*StatementList*的结果。

b. 如果R.value非**empty**，那么令V=R.value。

c. 如果R是一个“abrupt completion”，那么返回(R.type, V, R.target)。

8. 如果foundInB为**false**，且*DefaultClause*有一个*StatementList*，那么

a. 令R是求值*DefaultClause*的*StatementList*的结果。

b. 如果R.value非**empty**，那么令V=R.value。

c. 如果R是一个“abrupt completion”，那么返回(R.type, V, R.target)。

9. 重复，（注意如果步骤7.a.i已被执行，则这个循环不从B的开头开始）

a. 令C是B里的下一个*CaseClause*，如果没有这样的*CaseClause*，返回(**normal**, V, **empty**)。

b. 如果C有一个*StatementList*，那么

i. 令R是求值*C*的*StatementList*的结果。

ii. 如果R.value非**empty**，那么令V=R.value。

iii. 如果R是一个“abrupt completion”，那么返回(R.type, V, R.target)。

产生式*CaseClause* : **case** *Expression* **:** *StatementList* opt按以下方式求值：

1. 令exprRef是求值*Expression*的结果。

2. 返回GetValue(exprRef)。

注意：求值*CaseClause*不会执行关联的*StatementList*，它只是求值*Expression*并返回那个值，该值由*CaseBlock*算法用于确定哪个*StatementList*开始执行。

【SP108】

12.12节 带标号的语句

语法

*LabelledStatement* :

*Identifier* **:** *Statement*

语义

一个*Statement*可以前附标号，带标号的*Statement*只与带标号的**break**和**continue**语句一起使用。ECMAScript没有**goto**语句。

一个ECMAScript程序被认为在语法上是不正确的，如果它含有一个——包含在另一个使用相同*Identifier*作为标号的*LabelledStatement*里的——*LabelledStatement*。这条规则不适用于出现在一个——直接或间接内嵌在一个带标号的*Statement*里的——*FunctionDeclaration*的函数体里的标号。

产生式*Identifier* : *Statement*求值方式为，把*Identifier*添加到*Statement*的标号集里，然后求值*Statement*。如果*LabelledStatement*本身具有一个非空标号集，这些标号在求值*Statement*之前也被添加到*Statement*的标号集里。如果求值*Statement*的结果是(**break**, V, L)，其中L等于*Identifier*，产生式的结果是(**normal**, V, **empty**)。

在求值*LabelledStatement*之前，所含的*Statement*被认为拥有一个空标号集，除非它是一个*IterationStatement*或*SwitchStatement*，此时它被认为拥有一个仅由单一元素**empty**构成的标号集。

12.13节 **throw**语句

语法

*ThrowStatement* :

**throw** [no *LineTerminator* here] *Expression* **;**

语义

产生式*ThrowStatement* : **throw** [no *LineTerminator* here] *Expression* **;**按以下方式求值：

1. 令exprRef是求值*Expression*的结果。

2. 返回(**throw**, GetValue(exprRef), **empty**)。

12.14节 **try**语句

语法

*TryStatement*

**try** *Block Catch*

**try** *Block Finally*

**try** *Block Catch Finally*

*Catch* :

**catch (** *Identifier* **)** *Block*

*Finally* :

**finally** *Block*

**try**语句包围一个其中可能会发生异常情况的代码块，例如一个运行期异常或一个**throw**语句。**catch**从句提供异常处理代码。当**catch**从句捕获了一个异常时，它的*Identifier*绑定到那个异常。

【109】

语义

产生式*TryStatement* : **try** *Block Catch*按以下方式求值：

1. 令B是求值*Block*的结果。

2. 如果B.type非**throw**，返回B。

3. 为参数指定B.value，返回求值*Catch*的结果。

产生式*TryStatement* : **try** *Block Finally*按以下方式求值：

1. 令B是求值*Block*的结果。

2. 令F是求值*Finally*的结果。

3. 如果F.type为**normal**，返回B。

4. 返回F。

产生式*TryStatement* : **try** *Block Catch Finally*按以下方式求值：

1. 令B是求值*Block*的结果。

2. 如果B.type为**throw**，那么

a. 为参数指定B.value，令C是求值*Catch*的结果。

3. 否则，B.type非**throw**，

a. 令C是B。

4. 令F是求值*Finally*的结果。

5. 如果F.type为**normal**，返回C。

6. 返回F。

产生式*Catch* : **catch (** *Identifier* **)** *Block*按以下方式求值：

1. 令C是传给该产生式的实参。

2. 令oldEnv是当前运作中的执行上下文的LexicalEnvironment。

3. 传入oldEnv作为实参，令catchEnv是调用NewDeclarativeEnvironment的结果。

4. 传入String值*Identifier*作为实参，调用catchEnv的CreateMutableBinding具体方法。

5. 传入*Identifier*，C，和**false**作为实参，调用catchEnv的SetMutableBinding具体方法。注意在这种情况下最后一个实参是无关紧要的。

6. 把当前运作中的执行上下文的LexicalEnvironment设为catchEnv。

7. 令B是求值*Block*的结果。

8. 把当前运作中的执行上下文的LexicalEnvironment设为oldEnv。

9. 返回B。

注意：无论控制是如何离开*Block*的，LexicalEnvironment总是被恢复为它的先前状态。

产生式*Finally* : **finally** *Block*按以下方式求值：

1. 返回求值*Block*的结果。

12.14.1段 严格模式限制：

如果一个带有*Catch*的*TryStatement*出现在严格代码里，且*Catch*产生式的*Identifier*要么是“**eval**”，要么是“**arguments**”，则是一个SyntaxError。

12.15节 **debugger**语句

语法

*DebuggerStatement* :

**debugger ;**

【SP110】

语义

当运行在一个调试器下时，求值*DebuggerStatement*产生式允许一个实现引发一个断点。如果调试器不存在或没有激活，该语句没有可观察的效果。

产生式*DebuggerStatement* : **debugger ;**按以下方式求值：

1. 如果一个实现定义的调试设施可用并已被启用，那么

a. 执行一个实现定义的调试动作。

b. 令result是一个实现定义的Completion值。

2. 否则

a. 令result是(**normal**, **empty**, **empty**)。

3. 返回result。

第13章 函数定义

语法

*FunctionDeclaration* :

**function** *Identifier* **(** *FormalParameterList* opt **) {** *FunctionBody* **}**

*FunctionExpression* :

**function** *Identifier* opt**(** *FormalParameterList* opt **) {** *FunctionBody* **}**

*FormalParameterList* :

*Identifier*

*FormalParameterList* **,** *Identifier*

*FunctionBody* :

*SourceElements* opt

语义

产生式*FunctionDeclaration* : **function** *Identifier* **(** *FormalParameterList* opt **) {** *FunctionBody* **}**在声明绑定实例化（10.5节）期间按以下方式实例化：

1. 如13.2节规定，为参数指定*FormalParameterList* opt，为函数体指定*FunctionBody*，为Scope指定当前运作中的执行上下文的VariableEnvironment【定义将被“提升”，不受词法环境扩增影响】，如果*FunctionDeclaration*包含在严格代码里或它的*FunctionBody*是严格代码则为Strict标志指定**true**，返回创建一个新Function对象的结果。

产生式*FunctionExpression* : **function (** *FormalParameterList* opt **) {** *FunctionBody* **}**按以下方式求值：

1. 如13.2节规定，为参数指定*FormalParameterList* opt，为函数体指定*FunctionBody*，为Scope指定当前运作中的执行上下文的LexicalEnvironment【表达式若出现在catch里可访问到其形参】，如果*FunctionExpression*包含在严格代码里或它的*FunctionBody*是严格代码则为Strict标志指定**true**，返回创建一个新Function对象的结果。

产生式*FunctionExpression* : **function** *Identifier* **(** *FormalParameterList* opt **) {** *FunctionBody* **}**按以下方式求值：

1. 传入当前运作中的执行上下文的LexicalEnvironment作为实参，令funcEnv是调用的NewDeclarativeEnvironment结果。

2. 令envRec是funcEnv的Environment Record。

3. 传入String值*Identifier*作为实参，调用envRec的CreateImmutableBinding具体方法。

【SP111】

4. 如13.2节规定，为参数指定*FormalParameterList* opt，为函数体指定*FunctionBody*，为Scope指定funcEnv，如果*FunctionExpression*包含在严格代码里或它的*FunctionBody*是严格代码则为Strict标志指定**true**，令closure是创建一个新Function对象的结果。

5. 传入String值*Identifier*和closure作为实参，调用envRec的InitializeImmutableBinding具体方法。

6. 返回closure。

注意：一个*FunctionExpression*的*Identifier*可以从该*FunctionExpression*的*FunctionBody*里引用，以允许函数递归地调用它自己。然而，不像*FunctionDeclaration*的*Identifier*，*FunctionExpression*的*Identifier*不能从包含该*FunctionExpression*的作用域里引用，也不会影响那个作用域。

产生式*FunctionBody* : *SourceElements* opt 按以下方式求值：

1. 该*FunctionBody*的代码是严格模式代码，如果它是一个包含在严格模式代码里的*FunctionDeclaration*或*FunctionExpression*的一部分，或如果它的*SourceElements*的指令序言【Directive Prologue】含有一个Use Strict指令，或如果10.1.1段里的任何情况适用。如果该*FunctionBody*的代码是严格模式代码，*SourceElements*作为严格模式代码按以下步骤求值，否则作为非严格模式代码按以下步骤求值。

2. 如果*SourceElements*存在，返回求值*SourceElements*的结果。

3. 否则返回(**normal**, **undefined**, **empty**)。

13.1节 严格模式限制

如果任何*Identifier*值在一个严格模式*FunctionDeclaration*或*FunctionExpression*的*FormalParameterList*里出现超过一次，则是一个SyntaxError。

如果*Identifier*值“**eval**”或“**arguments**”出现在一个严格模式*FunctionDeclaration*或*FunctionExpression*的*FormalParameterList*里，则是一个SyntaxError。

如果*Identifier*值“**eval**”或“**arguments**”作为一个严格模式*FunctionDeclaration*或*FunctionExpression*的*Identifier*出现，则是一个SyntaxError。

13.2节 创建Function对象

给定一个可选的参数列表*FormalParameterList*，一个函数体*FunctionBody*，一个Lexical Environment值Scope，和一个Boolean标志Strict，一个Function对象按以下方式构造：

1. 创建一个新的原生ECMAScript对象，并令F是那个对象。

2. 如8.12节所述，设置F的除[[Get]]外的全部内部方法。

3. 把F的[[Class]]内部属性设为**"**Function**"**。

4. 如15.3.3.1则规定，把F的[[Prototype]]内部属性设为标准内建Function的原型对象。

5. 如15.3.5.4则所述，设置F的[[Get]]内部属性。

6. 如13.2.1段所述，设置F的[[Call]]内部属性。

7. 如13.2.2段所述，设置F的[[Construct]]内部属性。

8. 如15.3.5.3则所述，设置F的[[HasInstance]]内部属性。

9. 把F的[[Scope]]内部属性设为Scope的值。

10. 令names是一个列表，以从左到右的原文顺序包含着对应到*FormalParameterList*所含的标识符的String值。如果没有指定参数，令names是一个空列表。

11. 把F的[[FormalParameters]]内部属性设为names。

12. 把F的[[Code]]内部属性设为*FunctionBody*。

13. 把F的[[Extensible]]内部属性设为**true**。

14. 令len是由*FormalParameterList*指定的形参数量。如果没有指定参数，令len是0。

15. 传入**"length"**， Property Descriptor { [[Value]]:len, [[Writable]]:**false**, [[Enumerable]]:**false**, [[Configurable]]:**false** }，和**false**作为实参，调用F的[[DefineOwnProperty]]内部方法。

16. 令proto是创建一个新对象的结果，如同使用表达式**new** Object**()**构造的一般。

【SP112】

17. 传入**"constructor"**，Property Descriptor { [[Value]]:F, [[Writable]]:**true**, [[Enumerable]]:**false**, [[Configurable]]:**true** }，和**false**作为实参，调用proto的[[DefineOwnProperty]]内部方法。

18. 传入**"prototype"**，Property Descriptor { [[Value]]:proto, [[Writable]]:**true**, [[Enumerable]]:**false**, [[Configurable]]:**false** }，和**false**作为实参，调用F的[[DefineOwnProperty]]内部方法。

19. 如果Strict为**true**，那么

a. 令thrower是[[ThrowTypeError]]函数对象（13.2.3段）。

b. 传入**"caller"**，Property Descriptor { [[Get]]:thrower, [[Set]]:thrower, [[Enumerable]]:**false**, [[Configurable]]:**false** }，和**false**作为实参，调用F的 [[DefineOwnProperty]]内部方法。

c. 传入**"arguments"**，Property Descriptor { [[Get]]:thrower, [[Set]]:thrower, [[Enumerable]]:**false**, [[Configurable]]:**false** }，和**false**作为实参，调用F的 [[DefineOwnProperty]]内部方法。

20. 返回F。

注意：一个名为“prototype”的属性自动为每个函数创建，以考虑那个函数将被用作构造函数的可能性。

13.2.1段 [[Call]]

指定一个**this**值和一个实参列表，当一个Function对象F的[[Call]]内部方法被调用时，采取以下步骤：

1. 如10.4.3段所述，使用F的[[FormalParameters]]内部属性值，传入的实参列表args，和**this**值作为参数，令funcCtx是为函数代码建立一个新执行上下文的结果。

2. 令result是求值作为F的[[Code]]内部属性值的*FunctionBody*的结果。如果F没有[[Code]]内部属性，或该属性的值是一个空*FunctionBody*，那么result是(**normal**, **undefined**, **empty**)。

3. 退出执行上下文funcCtx，恢复先前的执行上下文。

4. 如果result.type为**throw**，那么抛出result.value。

5. 如果result.type为**return**，那么返回result.value。

6. 否则result.type一定是**normal**，返回**undefined**。

13.2.2段 [[Construct]]

指定一个可能为空的实参列表，当一个Function对象F的[[Construct]]内部方法被调用时，采取以下步骤：

1. 令obj是一个新创建的原生ECMAScript对象。

2. 如8.12节规定，设置obj的所有内部方法。

3. 把obj的[[Class]]内部属性设为**"**Object**"**。

4. 把obj的[[Extensible]]内部属性设为**true**。

5. 传入**"**prototype**"**作为实参，令proto是调用F的[[Get]]内部方法的结果。

6. 如果Type(proto)为Object，把obj的[[Prototype]]内部属性设为proto。

7. 如果Type(proto)非Object，把obj的[[Prototype]]内部属性设为标准内建Object的原型对象，如15.2.4段所述。

8. 以obj作为**this**值，传入传给[[Construct]]的实参列表作为实参列表args，令result是调用F的[[Call]]内部方法的结果。

9. 如果Type(result)为Object，那么返回result。

10. 返回obj。

13.2.3段 [[ThrowTypeError]]函数对象

[[ThrowTypeError]] 对象是一个独一无二的函数对象，唯一的一次定义如下：

1. 创建一个新的原生ECMAScript对象，令F是那个对象。

2. 如果8.12节所述，设置F的所有内部方法。

3. 把F的[[Class]]内部属性设为**"**Function**"**。

4. 把F的[[Prototype]]内部属性设为标准内建Function的原型对象，如15.3.3.1则规定。

5. 如13.2.1段所述，设置F的[[Call]]内部属性。

6. 把F的[[Scope]]内部属性设为Global Environment。

【SP113】

7. 把F的[[FormalParameters]]内部属性设置成一个空列表。

8. 把F的[[Code]]内部属性设为一个——无条件抛出一个TypeError异常且不执行任何其它动作的——*FunctionBody*。

9. 传入**"**length**"**，Property Descriptor { [[Value]]:**0**, [[Writable]]:**false**, [[Enumerable]]:**false**, [[Configurable]]:**false** }，和**false**作为实参，调用F的[[DefineOwnProperty]]内部方法。

10. 把F的[[Extensible]]内部属性设为**false**。

11. 令[[ThrowTypeError]]是F。

第14章 程序

语法

*Program* :

*SourceElements* opt

*SourceElements* :

*SourceElement*

*SourceElements SourceElement*

*SourceElement* :

*Statement*

*FunctionDeclaration*

语义

产生式*Program* : *SourceElements* opt 按以下方式求值：

1. 该*Program*的代码是严格模式代码，如果它的*SourceElements*的指令序言（14.1节）

含有一个Use Strict指令，或如果10.1.1段里的任何情况适用。如果该*Program*的代码是严格模式代码，*SourceElements*作为严格模式代码按以下步骤求值，否则作为非严格模式代码按以下步骤求值。

2. 如果*SourceElements*不存在，返回(**normal**, **empty**, **empty**)。

3. 如10.4.1段所述，令progCxt是一个用于全局代码的新的执行上下文。

4. 令result是求值*SourceElements*的结果。

5. 退出执行上下文progCxt。

6. 返回result。

注意：用于开始一个*Program*的求值的过程，以及用于处理这样一个求值的结果的过程，是由ECMAScript实现定义，而不是由这份规范。

产生式*SourceElements* : *SourceElements SourceElement*按以下方式求值：

1. 令headResult是求值*SourceElements*的结果。

2. 如果headResult是一个“abrupt completion”，返回headResult。

3. 令tailResult是求值*SourceElement*的结果。

4. 如果tailResult.value为**empty**，令V=headResult.value，否则令V=tailResult.value。

5. 返回(tailResult.type, V, tailResult.target)。

产生式*SourceElement* : *Statement*按以下方式求值：

1. 返回求值*Statement*的结果。

产生式*SourceElement* : *FunctionDeclaration*按以下方式求值：

1. 返回(**normal**, **empty**, **empty**)。

【SP114】

14.1节 指令序言和Use Strict指令

一个指令序言是——作为一个*Program*或*FunctionBody*的初始*SourceElement*产生式出现的——最长的*ExpressionStatement*产生式的序列，且序列中的每个*ExpressionStatement*完全由一个*StringLiteral*词法单元后跟一个分号构成。分号可以显式出现也可以由自动分号插入过程插入。一个指令序言可以是一个空序列。

一个Use Strict指令是一个在指令序言里的*ExpressionStatement*，其*StringLiteral*是精确的字符序列**"use strict"**或**'use strict'**。一个Use Strict指令不能包含一个*EscapeSequence*或*LineContinuation*。

一个指令序言可以包含多个Use Strict指令，但一个实现可以针对这种情况发出警告。

注意：一个指令序言的*ExpressionStatement*产生式在求值包含它的*SourceElements*产生式期间如常被求值。实现可以为出现在一个指令序言里，但不是Use Strict指令的*ExpressionStatement*产生式定义实现特定的含义。当一个实现在指令序言里遇到一个非Use Strict指令或不具有实现定义的含义的*ExpressionStatement*时，如果一个恰当的通知机制存在，实现应该发出警告。