你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将`this`引用添加属性，并且使用`new`运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“()”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了Undefined，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象“constructor”是否有属性名“1”
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为false，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是true了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为Object(1.)意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上

的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

   NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的“值”是多少）。
   NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将`this`引用添加属性，并且使用`new`运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“()”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了`Undefined`，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象“constructor”是否有属性名“1”
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为`false`，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是`true`了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为`Object(1.)`意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

   NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的“值”是多少）。
   NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将this引用添加属性，并且使用new运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用new Car()来创建的“类的实例”（也就是对象this）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“()”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了Undefined，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象“constructor”是否有属性名“1”
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为false，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是true了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为Object(1.)意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

   NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的“值”是多少）。
   NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将`this`引用添加属性，并且使用`new`运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“()”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了`Undefined`，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象“constructor”是否有属性名“1”
> 1 in Object(1.0).constructor
false

# (同上)
```

```
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为`false`，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是`true`了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为`Object(1.)`意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

   NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的“值”是多少）。
   NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将`this`引用添加属性，并且使用`new`运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
```

```
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否**

能用for..in语句列举出来”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“()”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了`Undefined`，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），

而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象"constructor"是否有属性名"1"
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为`false`，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是`true`了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为`Object(1.)`意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的“值”是多少）。
NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将`this`引用添加属性，并且使用`new`运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
```

```
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
```

```
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或defineProperty()等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“( )”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了Undefined，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象“constructor”是否有属性名“1”
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为false，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是true了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为`Object(1.)`意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

   NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的“值”是多少）。
   NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将`this`引用添加属性，并且使用`new`运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在

JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“()”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了`Undefined`，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象“constructor”是否有属性名“1”
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为`false`，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是`true`了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为`Object(1.)`意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

   NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它

的“值”是多少）。

NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将`this`引用添加属性，并且使用`new`运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
```

```
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“()”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了Undefined，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象“constructor”是否有属性名“1”
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为false，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是true了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为`Object(1.)`意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

   NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的“值”是多少）。
   NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将`this`引用添加属性，并且使用`new`运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有

一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来

管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“( )”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了`Undefined`，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报

错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象“constructor”是否有属性名“1”
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为`false`，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是`true`了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为`Object(1.)`意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

   NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的“值”是多少）。
   NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将`this`引用添加属性，并且使用`new`运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静

态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代

码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“()”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了`Undefined`，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象"constructor"是否有属性名"1"
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为`false`，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是`true`了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为`Object(1.)`意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

   NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的“值”是多少）。
   NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将`this`引用添加属性，并且使用`new`运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“( )”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了`Undefined`，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样

的。

```
# 检查对象"constructor"是否有属性名"1"
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为`false`，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式"1 in 1..constructor"的值就会是`true`了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为`Object(1.)`意味着将数字"1.0"封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么"1..constructor"也就指向x.constructor。

```
x = new Number(1.0);
```

而"x.constructor"不是自有属性，并且，由于x是"Number()"这个类/构造器的子类实例，因此该属性实际继承自原型链上的"Number.prototype.construtcotr"这个属性。然后，在缺省情况下，"*aFunction*.prototype.construtcotr"指向这个函数自身。

也就是说，"Number.prototype.construtctor"与"1..constructor"相同，且都指向Number()自身。

所以上面的示例中，当我们添加了"Number[1]"这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是"1在某个1..n"的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号"`+-*/`"并确保结果可作为表达式求值。

   NOTE：题目1是一个空数组的"单值表达式"，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的"值"是多少）。
   NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将this引用添加属性，并且使用new运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用new Car()来创建的“类的实例”（也就是对象this）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“()”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了`Undefined`，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身

并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象"constructor"是否有属性名"1"
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为`false`，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是`true`了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为`Object(1.)`意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂

概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

   NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的“值”是多少）。
   NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将`this`引用添加属性，并且使用`new`运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。

这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；

2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“( )”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了`Undefined`，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象“constructor”是否有属性名“1”
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
```

```
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为`false`，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是`true`了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为`Object(1.)`意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

   NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的“值”是多少）。
   NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将`this`引用添加属性，并且使用`new`运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}
```

```
var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“()”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了`Undefined`，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制

并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象“constructor”是否有属性名“1”
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为`false`，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是`true`了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为`Object(1.)`意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的“值”是多少）。
NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将`this`引用添加属性，并且使用`new`运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
```

```
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
```

```
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或defineProperty()等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“( )”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了Undefined，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象“constructor”是否有属性名“1”
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为false，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是true了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为`Object(1.)`意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

   NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它的“值”是多少）。
   NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。欢迎你回到我的专栏。

如果你听过上一讲，那么你应该知道，接下来我要与你聊的是JavaScript的**面向对象系统**。

最早期的JavaScript只有一个非常弱的对象系统。我用过JavaScript 1.0，甚至可能还是最早尝试用它在浏览器中写代码的一批程序员，我也寻找和收集过早期的CEniv和ScriptEase，只为了探究它最早的语言特性与JavaScript之间的相似之处。

然而，不得不说的是，曾经的JavaScript在**面向对象**特性方面，在语法上更像Java，而在实现上却是谁也不像。

JavaScript 1.0~1.3中的对象

在JavaScript 1.0的时候，对象是不支持继承的。那时的JavaScript使用的是称为“**类抄写**”的技术来创建对象，就是“在一个函数中将`this`引用添加属性，并且使用`new`运算来创建对象实例”，例如：

```
function Car() {
  this.name = "Car";
  this.color = "Red";
}

var x = new Car();
```

关于类抄写以及与此相关的性质，我会在后续的内容中详细讲述。现在，你在这里需要留意的是：在“Car()”这个函数中，事实上该函数是以“类”的身份来声明了一系列的属性（Property）。正是因此，使用`new Car()`来创建的“类的实例”（也就是对象`this`）也就具有了这些属性。

这样的“类→对象”的模型其实是很简单和粗糙的。但JavaScript 1.0时代的**对象**就是如此，并且，重要的是，事实上直到现在

JavaScript的对象仍然如此。ECMAScript规范明确定义了这样的一个概念：

对象是零到多个的属性的集合。

> In ECMAScript, an *object* is a collection of zero or more *properties*.

你可能还注意到了，JavaScript 1.0的对象系统是有类的，并且在语义上也是“对象创建自类”。这使得它在表面上“看起来”还是有一些继承性的。例如，一个对象必然继承了它的类所声明的那些性质，也就是“属性”。但是因为这个1.0版存在的时间很短，所以后来大多数人都不记得JavaScript“**有类，而又不支持类的继承**”这件事情，从而将从JavaScript 1.1才开始具有的**原型继承**作为它最主要的面向对象特征。

在这个阶段，JavaScript中有关全局环境和全局变量的设计也已经成熟了，简单地来说，就是：

1. 向没有声明的变量名赋值，会隐式地创建一个全局变量；
2. 全局变量会被绑定为全局对象（global）的属性 。

这样一来，JavaScript的变量环境（或者全局环境）与对象系统就关联了起来。而接下来，由于JavaScript也实现了带有闭包性质的函数，因此“闭包”也成了环境的管理组件。也就是说，闭包与对象都具有实现变量环境的能力。

因此，在这个阶段，JavaScript提出了“**对象闭包**”与“**函数闭包**”两个概念，并把它们用来实现的环境称为“**域**（Scope）”。这些概念和语言特性，一直支持JavaScript走到1.3版本，并随着ECMAScript ed3确定了下来。

在这个时代，JavaScript语言的设计与发展还基本是以它的发明者布兰登·艾奇（Brendan Eich）为主导的，JavaScript的语言特性也处于一个较小的集合中，并且它的应用也主要是以浏览器客户端为主。这时代的JavaScript深得早期设计与语言定义的精髓。这些东西，你可以从后来布兰登·艾奇的一个开源项目中读到。这个项目称为Narcissus，是用JavaScript来实现的一个完整的JavaScript 1.3。在这个项目中，对象和函数所创建的闭包都统一由一个简单的对象表示，称为scope，它包括“object”和“parent”两个成员，分别表示本闭包的对象，以及父一级的作用域。例如：

```
scope = {
  object: <创建本闭包的对象或函数>,
  parent: <父级的scope>
}
```

因此，所谓“**使用with语句创建一个对象闭包**”就简单地被实现为“向既有的作用域链尾加入一个新的scope”。

```
// code from $(narcissus)/src/jsexec.js
...
// 向x所代表的scope-chain表尾加入一个新的scope
x.scope = {object: t, parent: x.scope};
try {
  // n.body是with语句中执行的语句块
  execute(n.body, x); // 指在该闭包（链）`x`中执行上述语句
}
finally {
  x.scope = x.scope.parent;  // 移除链尾的一个scope
}
```

可见JavaScript 1.3时代的执行环境，其实就是一个闭包链的管理。而且这种闭包既可以是对象的，也可以是函数的。尽管在静态语法说明或描述时，它们被称为**作用域**或**域**（Scope），或者在动态环境中它们被称为**上下文**（Context），但在本质上，它们是同样的一堆东西。

综合来看，JavaScript中的对象本质上是**属性集**，这可以视为一个**键值列表**，而对象继承是由这样的列表构成的、称为原型的链。另一方面，执行的上下文就是函数或全局的变量表，这同样可以表达为一个键值列表，而执行环境也可以视为一个由该键值列表构成的链。

于是，在JavaScript 1.3，以及ECMAScript ed3的整个时代，这门语言仅仅依赖**键值列表**和**基于它们的链**实现并完善了它最初的设计。

属性访问与可见性

但是从一开始，JavaScript就有一个东西没有说清楚，那就是属性名的可见性。

这种可见性在OOP（面向对象编程）中有专门的、明确的说法，但在早期的JavaScript中，它可以简单地理解为“**一个属性是否能用for..in语句列举出来**”。如果它可以被列举，那么就是可见的，否则就称为隐藏的。

你知道，任何对象都有“constructor”这个属性，缺省指向创建它的构造器函数，并且它应当是隐藏的属性。但是在早期的JavaScript中，这个属性如何隐藏，却是没有规范来约定的。例如在JScript中，它就是一个特殊名字，只要是这个名字，就隐藏；而在SpiderMonkey中，当用户重写这个属性后，它就变成了可见的。

后来ECMAScript就约定了所谓的“**属性的性质**（attributes）”这样的东西，也就是我们现在知道的**可写性**、**可列举性**（可见性）和**可配置性**。ECMAScript约定：

- “constructor”缺省是一个不可列举的属性；
- 使用赋值表达式添加属性时，属性的可列举性缺省为`true`。

这样一来，“constructor”在可见性（这里是指可列举性）上的行为就变得可预期了。

类似于此的，ECMAScript约定了读写属性的方法，以及在属性中访问、操作性质的全部规则，并统一使用所谓“属性描述符”来管理这些规则。于是，这使得ECMAScript规范进入了5.x时代。相较于早期的3.x，这个版本的ECMAScript规范并没有太多的改变，只是从语言概念层面上实现了“大一统”，所有浏览器厂商，以及引擎的开发者都遵循了这些规则，为后续的JavaScript大爆发——ECMAScript 6的发布铺平了道路。

到目前为止，JavaScript中的对象仍然是简单的、原始的、使用JavaScript 1.x时代的基础设计的原型继承。而每一个对象，仍然都只是简简单单的一个所谓的“**属性包**”。

从原型中继承来的属性

对于绝大多数对象来说，“constructor”是从它的原型继承来的一个属性，这有别于它“自有的（Own）”属性。在原型继承中，在子类实例重写属性时，实际发生的行为是“**在子类实例的自有属性表中添加一个新项**”。这并不改变原型中相同属性名的值，但子类实例中的**属性性质**以及**值**覆盖了原型中的。这是原型继承——几乎是公开的——所有的秘密所在。

在使用原型继承来的属性时，有两种可能的行为，这取决于属性的具体性质——属性描述符的类型。

1. 如果是**数据描述符**（d），那么`d.value`总是指向这个数据的值本身；
2. 如果是**存取描述符**，那么`d.get()`和`d.set()`将分别指向属性的存取方法。

并且，如果是存取描述符，那么存取方法（get/setter）并不一定关联到数据，也并不一定是数据的置值或取值。某些情况下，存取方法可能会用作特殊的用途，例如模拟在VBScript中常常出现的“无括号的方法调用”。

```
excel = Object.defineProperty(new Object, 'Exit', {
  get() {
    process.exit();
  }
});

// 类似JScript/VBScript中的ActiveObject组件的调用方法
excel.Exit;
```

当用户不使用属性赋值或`defineProperty()`等方法来添加自有的属性时，属性访问会（默认地）上溯原型链直到找到指定属性。这一定程度上成就了“包装类”这一特殊的语言特性。

所谓“**包装类**”是JavaScript从Java借鉴来的特性之一，它使得用户代码可以用标准的面向对象方法来访问普通的值类型数据。于是，所谓“一切都是对象”就在眨眼间变成了现实。例如，下面这个示例中使用的字符串常量x，它的值是"abc"：

```
x = "abc";
console.log(x.toString());
```

当在使用x.toString()时，JavaScript会自动将“值类型的字符串（"abc"）”通过包装类变成一个字符串对象。这类似于执行下面的代码，使用函数Object()来“将这个值显式地转换为对象”。

```
console.log(Object(x).toString());
```

这个包装的过程发生于**函数调用运算“()”**的处理过程中，或者将“x.toString”作为整体来处理的过程中（例如作为一个ECMAScript规范引用类型来处理的过程）。也就是说，仅仅是“对象属性存取”这个行为本身，并不会触发一个普通“值类型数据”向它的包装类型转换。

除了`Undefined`，基本类型中的所有值类型数据都有自己的包装类，包括符号，又或者布尔值。这使得这些值类型的数据也可以具有与之对应的包装类的原型属性或方法。这些属性与方法自己引用自原型，而不是自有数据。很显然的，值类型数据本身并不是对象，因此也不可能拥有自有的属性表。

字面量与标识符

通常情况下，开发人员会将标识符直接称为**名字**（在ECMAScript规范中，它的全称是“标识符名字（*IdentifierName*）”），而**字面量**是一个数据的文本表示。显然，通常标识符就用作后者的名字标识。对于这两种东西，在ECMAScript中的处理机制并不太一样，并且在文本解析阶段就会把二者区分开来。

```
// var x = 1;
1;
x;
```

比如在这个例子中，如果其中“1”是字面量值，JavaScript会直接处理它；而x是一个标识符（哪怕它只是一个值类型数据的变量名），就需要建立一个“引用”来处理了。但是接下来，如果是代码（假设下面的代码是成立的）：

```
1.toString
```

那么它作为“整体”就需要被创建为一个引用，以作为后续计算的操作数（取成员值，或仅是引用该成员）。是的，就它们同是“引用”这一事实而言，“1.toString”与“x”在引擎级别有些类似。

然而在数字字面量中，“1.xxxxx”这样的语法是有含义的。它是浮点数的表示法。所以“1.toString”这样的语法在JavaScript中会报错，这个错误来自于浮点数的字面量解析过程，而不是“.作为存取运算符”的处理过程。在JavaScript中，浮点数的小位数是可以为空的，因此“1.”和“1.0”将作为相同的浮点数被解析出来。

既然“1.”表示的是浮点数，那么“1..constructor”表示的就是该浮点数字面量的“.constructor”属性。

现在我想你已经看出来了，标题中的：

```
1 in 1..constructor
```

其实是一个表达式。在语义上，“1..constructor”与“Object(1.0).constructor”这样的表达式是等义的，且它们的使用效果也是一样的。

```
# 检查对象“constructor”是否有属性名“1”
> 1 in Object(1.0).constructor
false

# (同上)
> 1 in 1..constructor
fales
```

属性存取的不确定性

除了**存取器**（get/setter）带来的不确定性之外，JavaScript的属性存取结果还受到**原型继承（链）**的影响。上例中的表达式值并不恒为`false`，例如我们给Number加一个下标值为1的属性（我们不用管这个属性的值是什么），那么标题中的表达式“1 in 1..constructor”的值就会是`true`了。

```
# 修改原型链中的对象
> Number[1] = true; // or anything


# 影响到上例中表达式的结果
> 1 in 1..constructor
true
```

因为`Object(1.)`意味着将数字“1.0”封装成它对应的包装类的一个对象实例（x），我们假设这个对象是x，那么“1..constructor”也就指向x.constructor。

```
x = new Number(1.0);
```

而“x.constructor”不是自有属性，并且，由于x是“Number()”这个类/构造器的子类实例，因此该属性实际继承自原型链上的“Number.prototype.construtcotr”这个属性。然后，在缺省情况下，“*aFunction*.prototype.construtcotr”指向这个函数自身。

也就是说，“Number.prototype.construtctor”与“1..constructor”相同，且都指向Number()自身。

所以上面的示例中，当我们添加了“Number[1]”这个下标属性之后，标题中表达式的值就变了。

知识回顾

这一讲的标题看起来像是其他语言中的循环或迭代，又或者在代码文本上看起来像是一个范围检查（语义上看起来像是“1在某个1..n”的范围中）。但事实上，它不仅包含了JavaScript中从对象成员存取这样的基础话题，还一直延伸到了**包装类**这样的复杂概念的全部知识。

当然，重要的是，源于JavaScript中面向对象系统的独特设计，它的对象属性存取结果总是不确定的。

- 如果属性不是自有的，那么它的值就是原型决定的；
- 当属性是存取方法的，那么它的值就是求值决定的。

思考题

虽然这一讲没有太深入的内容，但是有两道练习题留给大家，非常烧脑：

1. 试述表达式`[]`的求值过程。
2. 在上述表达式中加上符号“`+-*/`”并确保结果可作为表达式求值。

   NOTE：题目1是一个空数组的“单值表达式”，当它作为表达式来处理时，请问它是如何求值的（你得先想想它

的“值”是多少）。

NOTE：题目2的意思，就是如何把这些字符组合在一起，仍然是一个可求值的表达式。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。