第6章

对象

对象是JavaScript的基本数据类型。对象是一种复合值：它将很多值（原始值或者其他对象）聚合在一起，可通过名字访问这些值。对象也可看做是属性的无序集合，每个属性都是一个名/值对。属性名是字符串，因此我们可以把对象看成是从字符串到值的映射。这种基本数据结构还有很多种叫法，有些我们已然非常熟悉，比如“散列”（hash）、“散列表”（hashtable）、“字典”（dictionary）、“关联数组”（associative array）。然而对象不仅仅是字符串到值的映射，除了可以保持自有的属性，JavaScript对象还可以从一个称为原型的对象继承属性。对象的方法通常是继承的属性。这种“原型式继承”（prototypal inheritance）是JavaScript的核心特征。

JavaScript对象是动态的——可以新增属性也可以删除属性——但它们常用来模拟静态对象以及静态类型语言中的“结构体”（struct）。有时它们也用做字符串的集合（忽略名/值对中的值）。

除了字符串、数字、true、false、null和undefined之外，JavaScript中的值都是对象。尽管字符串、数字和布尔值不是对象，但它们的行为和不可变对象（参照3.6节）非常类似。

3.7节已经讲到，对象是可变的，我们通过引用而非值来操作对象。如果变量x是指向一个对象的引用，那么执行代码`var y = x;` 变量y也是指向同一个对象的引用，而非这个对象的副本。通过变量y修改这个对象亦会对变量x造成影响。

对象最常见的用法是创建（create）、设置（set）、查找（query）、删除（delete）、检测（test）和枚举（enumerate ）它的属性。我们会在开始的几节讲述这些基础操作。后续的几节讲述高级主题，其中相当一部分内容来自于ECMAScript 5。

属性包括名字和值。属性名可以是包含空字符串在内的任意字符串，但对象中不能存在两个同名的属性。值可以是任意JavaScript值，或者（在ECMAScript 5中）可以是一个getter或setter函数（或两者都有）。6.6节会有关于getter和setter函数的讲解。除了名字和值之外，每个属性还有一些与之相关的值，称为“属性特性”（property attribute）[译注1]：

- 可写（writable attribute），表明是否可以设置该属性的值。
- 可枚举（enumerable attribute），表明是否可以通过for/in循环返回该属性。
- 可配置（configurable attribute），表明是否可以删除或修改该属性。

在ECMAScript 5之前，通过代码给对象创建的所有属性都是可写的、可枚举的和可配置的。在ECMAScript 5中则可以对这些特性加以配置。6.7节讲述如何操作。

除了包含属性之外，每个对象还拥有三个相关的对象特性（object attribute）：

- 对象的原型（prototype）指向另外一个对象，本对象的属性继承自它的原型对象。
- 对象的类（class）是一个标识对象类型的字符串。
- 对象的扩展标记（extensible flag）指明了（在ECMAScript 5中）是否可以向该对象添加新属性。

6.1.3节和6.2.2节会有关于原型和属性继承的讲述，6.8节会进一步详细讲述这三个特性。

最后，我们用下面这些术语来对三类JavaScript对象和两类属性作区分：

- 内置对象（native object）是由ECMAScript规范定义的对象或类。例如，数组、函数、日期和正则表达式都是内置对象。
- 宿主对象（host object）是由JavaScript解释器所嵌入的宿主环境（比如Web浏览器）定义的。客户端JavaScript中表示网页结构的HTMLElement对象均是宿主对象。既然宿主环境定义的方法可以当成普通的JavaScript函数对象，那么宿主对象也可以当成内置对象。
- 自定义对象（user-defined object）是由运行中的JavaScript代码创建的对象。
- 自有属性（own property）是直接在对象中定义的属性。
- 继承属性（inherited property）是在对象的原型对象中定义的属性。

译注1：property和attribute都可以单独理解为“属性”，在这里property attribute是一个词组，意思是“属性的特性”，随后提到的“可写”、“可枚举”和“可配置”即是属性的三种特性。

6.1 创建对象

可以通过对象直接量、关键字new和（ECMAScript 5中的）Object.create()函数来创建对象。接下来几节将对这些技术一一讲述。

6.1.1 对象直接量

创建对象最简单的方式就是在JavaScript代码中使用对象直接量。对象直接量是由若干名/值对组成的映射表，名/值对中间用冒号分隔，名/值对之间用逗号分隔，整个映射表用花括号括起来。属性名可以是JavaScript标识符也可以是字符串直接量（包括空字符串）。属性的值可以是任意类型的JavaScript表达式，表达式的值（可以是原始值也可以是对象值）就是这个属性的值。下面有一些例子：

```
var empty = {};                                  // 没有任何属性的对象
var point = { x:0, y:0 };                        // 两个属性
var point2 = { x:point.x, y:point.y+1 };         // 更复杂的值
var book = {
    "main title": "JavaScript",                  // 属性名字里有空格,必须用字符串表示
    'sub-title': "The Definitive Guide",         // 属性名字里有连字符，必须用字符串表示
    "for": "all audiences",                      //  "for"是保留字，因此必须用引号
    author: {                                    // 这个属性的值是一个对象
        firstname: "David",                      // 注意，这里的属性名都没有引号
        surname: "Flanagan"
    }
};
```

在ECMAScript 5（以及ECMAScript 3的一些实现）中，保留字可以用做不带引号的属性名。然而对于ECMAScript 3来说，使用保留字作为属性名必须使用引号引起来。在ECMAScript 5中，对象直接量中的最后一个属性后的逗号将忽略，且在ECMAScript 3的大部分实现中也可以忽略这个逗号，但在IE中则报错。

对象直接量是一个表达式，这个表达式的每次运算都创建并初始化一个新的对象。每次计算对象直接量的时候，也都会计算它的每个属性的值。也就是说，如果在一个重复调用的函数中的循环体内使用了对象直接量，它将创建很多新对象，并且每次创建的对象的属性值也有可能不同。

6.1.2 通过new创建对象

new运算符创建并初始化一个新对象。关键字new后跟随一个函数调用。这里的函数称做构造函数（constructor），构造函数用以初始化一个新创建的对象。JavaScript语言核心中的原始类型都包含内置构造函数。例如：

```
var o = new Object(); // 创建一个空对象，和{}一样
```

```
var a = new Array();          // 创建一个空数组，和[]一样
var d = new Date();           // 创建一个表示当前时间的Date对象
var r = new RegExp("js");     //创建一个可以进行模式匹配的EegExp对象
```

除了这些内置构造函数，用自定义构造函数来初始化新对象也是非常常见的。第9章将详细讲述其中的细节。

6.1.3 原型

在讲述第三种对象创建技术之前，我们应当首先解释一下原型。每一个JavaScript对象（null除外）都和另一个对象相关联。“另一个”对象就是我们熟知的原型，每一个对象都从原型继承属性。

所有通过对象直接量创建的对象都具有同一个原型对象，并可以通过JavaScript代码Object.prototype获得对原型对象的引用。通过关键字new和构造函数调用创建的对象的原型就是构造函数的prototype属性的值。因此，同使用{}创建对象一样，通过new Object()创建的对象也继承自Object.prototype。同样，通过new Array()创建的对象的原型就是Array.prototype，通过new Date()创建的对象的原型就是Date.prototype。

没有原型的对象为数不多，Object.prototype就是其中之一。它不继承任何属性。其他原型对象都是普通对象，普通对象都具有原型。所有的内置构造函数（以及大部分自定义的构造函数）都具有一个继承自Object.prototype的原型。例如，Date.prototype的属性继承自Object.prototype，因此由new Date()创建的Date对象的属性同时继承自Date.prototype和Object.prototype。这一系列链接的原型对象就是所谓的“原型链”（prototype chain）。

6.2.2节讲述属性继承的工作机制。6.8.1节将会讲到如何获取对象的原型。第9章将会更详细地讨论原型和构造函数，包括如何通过编写构造函数定义对象的“类”，以及给构造函数的prototype属性赋值可以让其“实例”直接使用这个原型上的属性和方法。

6.1.4 Object.create()

ECMAScript 5定义了一个名为Object.create()的方法，它创建一个新对象，其中第一个参数是这个对象的原型。Object.create()提供第二个可选参数，用以对对象的属性进行进一步描述。6.7节会详细讲述第二个参数。

Object.create()是一个静态函数，而不是提供给某个对象调用的方法。使用它的方法很简单，只须传入所需的原型对象即可：

```
var o1 = Object.create({x:1, y:2});    // o1继承了属性x和y
```

可以通过传入参数null来创建一个没有原型的新对象，但通过这种方式创建的对象不会继承任何东西，甚至不包括基础方法，比如toString()，也就是说，它将不能和“+”运算符一起正常工作：

```
var o2 = Object.create(null); //o2不继承任何属性和方法
```

如果想创建一个普通的空对象（比如通过{}或new Object()创建的对象），需要传入Object.prototype：

```
var o3 = Object.create(Object.prototype);       //o3和{}和new Object()一样
```

可以通过任意原型创建新对象（换句话说，可以使任意对象可继承），这是一个强大的特性。在ECMAScript 3中可以用类似例6-1中的代码来模拟原型继承[注1]：

例6-1：通过原型继承创建一个新对象

```
// inherit() 返回了一个继承自原型对象p的属性的新对象
// 这里使用ECMAScript 5中的Object.create()函数（如果存在的话）
// 如果不存在Object.create()，则退化使用其他方法
function inherit(p) {
    if (p == null) throw TypeError();      // p是一个对象，但不能是null
    if (Object.create)                     // 如果Object.create()存在
        return Object.create(p);           // 直接使用它
    var t = typeof p;                      // 否则进行进一步检测
    if (t !== "object" && t !== "function") throw TypeError();
    function f() {};                       // 定义一个空构造函数
    f.prototype = p;                       //将其原型属性设置为p
    return new f();                        //使用f()创建p的继承对象
}
```

在看完第9章关于构造函数的内容后，例6-1中的inherit()函数会更容易理解。现在只要知道它返回的新对象继承了参数对象的属性就可以了。注意，inherit()并不能完全代替Object.create()，它不能通过传入null原型来创建对象，而且不能接收可选的第二个参数。不过我们仍会在本章和第9章的示例代码中多次用到inherit()。

inherit()函数的其中一个用途就是防止库函数无意间（非恶意地）修改那些不受你控制的对象。不是将对象直接作为参数传入函数，而是将它的继承对象传入函数。当函数读取继承对象的属性时，实际上读取的是继承来的值。如果给继承对象的属性赋值，则这些属性只会影响这个继承对象自身，而不是原始对象：

```
var o = { x: "don't change this value" };
library_function(inherit(o)); // 防止对o的意外修改
```

注1：一致认为Douglas Crockford是最早提出用这种方法实现对象继承函数的人，参照：*http://javascript.crockford.com/prototypal.html*。

了解其工作原理，需要首先了解JavaScript中属性的查询和设置机制。接下来会讲到。

6.2 属性的查询和设置

4.4节已经提到，可以通过点(.)或方括号([])运算符来获取属性的值。运算符左侧应当是一个表达式，它返回一个对象。对于点(.)来说，右侧必须是一个以属性名称命名的简单标识符。对于方括号来说([])，方括号内必须是一个计算结果为字符串的表达式，这个字符串就是属性的名字：

```
var author = book.author;      //得到book的"author"属性
var name = author.surname      //得到获得author的"surname"属性
var title = book["main title"]//得到book的"main title"属性
```

和查询属性值的写法一样，通过点和方括号也可以创建属性或给属性赋值，但需要将它们放在赋值表达式的左侧：

```
book.edition = 6;                      //给book创建一个名为"edition"的属性
book["main title"] = "ECMAScript";     //给"main title"属性赋值
```

在ECMAScript 3中，点运算符后的标识符不能是保留字，比如，o.for或o.class是非法的，因为for是JavaScript的关键字，class是保留字。如果一个对象的属性名是保留字，则必须使用方括号的形式访问它们，比如o["for"]和o["class"]。ECMAScript 5对此放宽了限制（包括ECMAScript 3的某些实现），可以在点运算符后直接使用保留字。

当使用方括号时，我们说方括号内的表达式必须返回字符串。其实更严格地讲，表达式必须返回字符串或返回一个可以转换为字符串的值。在第7章里有一些例子中的方括号内使用了数字，这情况象是非常常见的。

6.2.1 作为关联数组的对象

上文提到，下面两个JavaScript表达式的值相同：

```
object.property
object["property"]
```

第一种语法使用点运算符和一个标识符，这和C和Java中访问一个结构体或对象的静态字段非常类似。第二种语法使用方括号和一个字符串，看起来更像数组，只是这个数组元素是通过字符串索引而不是数字索引。这种数组就是我们所说的关联数组（associative array），也称做散列、映射或字典（dictionary）。JavaScript对象都是关联数组，本节将讨论它的重要性。

在C、C++和Java和一些强类型(strong typed)[译注2]语言中，对象只能拥有固定数目的属性，并且这些属性名称必须提前定义好。由于JavaScript是弱类型语言，因此不必遵循这条规定，在任何对象中程序都可以创建任意数量的属性[译注3]。但当通过点运算符(.)访问对象的属性时，属性名用一个标识符来表示。标识符必须直接出现在JavaScript程序中，它们不是数据类型，因此程序无法修改它们[译注4]。

反过来讲，当通过[]来访问对象的属性时，属性名通过字符串来表示。字符串是JavaScript的数据类型，在程序运行时可以修改和创建它们。因此，可以在JavaScript中使用下面这种代码：

```
var addr = "";
for(i = 0; i < 4; i++) {
    addr += customer["address" + i] + '\n';}
```

这段代码读取customer对象的address0、address1、address2和address3属性，并将它们连接起来。

这个例子主要说明了使用数组写法和用字符串表达式来访问对象属性的灵活性。这段代码也可以通过点运算符来重写，但是很多场景只能使用数组写法来完成。假设你正在写一个程序，这个程序利用网络资源计算当前用户股票市场投资的金额。程序允许用户输入每只股票的名称和购股份额。该程序使用名为portfolio的对象来存储这些信息。每只股票在这个对象中都有对应的属性，属性名称就是股票名称，属性值就是购股数量，例如，如果用户持有IBM的50股，那么portfolio.ibm属性的值就为50。

下面是程序的部分代码，这个函数用来给portifolio添加新的股票：

```
function addstock(portfolio, stockname, shares) {
    portfolio[stockname] = shares;
}
```

由于用户是在程序运行时输入股票名称，因此在之前无法得知这些股票的名称是什么。而由于在写程序的时候不知道属性名称，因此无法通过点运算符(.)来访问对象portfolio的属性。但可以使用[ ]运算符，因为它使用字符串值（字符串值是动态的，可

译注2：强类型，为所有变量指定数据类型称为“强类型”。强/弱类型是指类型检查的严格程度。语言有无类型、弱类型和强类型三种。无类型的不检查，甚至不区分指令和数据。弱类型的检查很弱，仅能严格地区分指令和数据。强类型的则严格的在编译期间进行检查。

译注3：这里的意思是可以动态地给对象添加属性。严格讲，JavaScript对象的属性个数是有上限的。

译注4：“程序不能修改标识符”的意思是说，在程序运行时无法动态指定一个标识符，当然eval除外。

以在运行时更改）而不是标识符（标识符是静态的，必须写死在程序中）作为索引对属性进行访问。

第5章介绍了for/in循环（6.5节还会进一步介绍）。当使用for/in循环遍历关联数组时，就可以清晰地体会到for/in的强大之处。下面的例子就是利用for/in计算portfolio的总计值：

```
function getvalue(portfolio) {
    var total = 0.0;
    for(stock in portfolio) {          // 遍历portfolio中的每只股票
        var shares = portfolio[stock]; // 得到每只股票的份额
        var price = getquote(stock);   // 查找股票价格
        total += shares * price;       // 将结果累加至total中
    }
    return total;                      // 返回total的值
}
```

6.2.2 继承

JavaScript对象具有“自有属性”（own property），也有一些属性是从原型对象继承而来的。为了更好地理解这种继承，必须更深入地了解属性访问的细节。本节中的许多示例代码借用了例6-1中的inherit()函数，通过给它传入指定原型对象来创建实例。

假设要查询对象o的属性x，如果o中不存在x，那么将会继续在o的原型对象中查询属性x。如果原型对象中也没有x，但这个原型对象也有原型，那么继续在这个原型对象的原型上执行查询，直到找到x或者查找到一个原型是null的对象为止。可以看到，对象的原型属性构成了一个“链”，通过这个“链”可以实现属性的继承。

```
var o = {}   // o 从 Object.prototype 继承对象的方法
o.x = 1;     // 给o定义一个属性x
var p = inherit(o); // p继承o和Object.prototype
p.y = 2;     // 给p定义一个属性y
var q = inherit(p); // q继承p、o和Object.prototype
q.z = 3;     // 给q定义一个属性z
var s = q.toString(); // toString继承自Object.prototype
q.x + q.y    // => 3: x和y分别继承自o和p
```

现在假设给对象o的属性x赋值，如果o中已经有属性x（这个属性不是继承来的），那么这个赋值操作只改变这个已有属性x的值。如果o中不存在属性x，那么赋值操作给o添加一个新属性x。如果之前o继承自属性x，那么这个继承的属性就被新创建的同名属性覆盖了。

属性赋值操作首先检查原型链，以此判定是否允许赋值操作。例如，如果o继承自一个只读属性x，那么赋值操作是不允许的（6.2.3节将对此进行详细讨论）。如果允许属性赋

值操作，它也总是在原始对象上创建属性或对已有的属性赋值，而不会去修改原型链。在JavaScript中，只有在查询属性时才会体会到继承的存在，而设置属性则和继承无关，这是JavaScript的一个重要特性，该特性让程序员可以有选择地覆盖（override）继承的属性。

```
var unitcircle = { r:1 };      // 一个用来继承的对象
var c = inherit(unitcircle);   // c继承属性r
c.x = 1; c.y = 1;              // c定义两个属性
c.r = 2;                       // c覆盖继承来的属性
unitcircle.r;                  // => 1，原型对象没有修改
```

属性赋值要么失败，要么创建一个属性，要么在原始对象中设置属性，但有一个例外，如果o继承自属性x，而这个属性是一个具有setter方法的accessor属性（参照6.6节），那么这时将调用setter方法而不是给o创建一个属性x。需要注意的是，setter方法是由对象o调用的，而不是定义这个属性的原型对象调用的。因此如果setter方法定义任意属性，这个操作只是针对o本身，并不会修改原型链。

6.2.3 属性访问错误

属性访问并不总是返回或设置一个值。本节讲述查询或设置属性时的一些出错情况。

查询一个不存在的属性并不会报错，如果在对象o自身的属性或继承的属性中均未找到属性x，属性访问表达式o.x返回undefined。回想一下我们的book对象有属性“subtitle”，而没有属性“subtitle”：

```
book.subtitle; // => undefined:属性不存在
```

但是，如果对象不存在，那么试图查询这个不存在的对象的属性就会报错。null和undefined值都没有属性，因此查询这些值的属性会报错，接上例：

```
// 抛出一个类型错误异常，undefined没有length属性
var len = book.subtitle.length;
```

除非确定book和book.subtitle都是（或在行为上）对象，否则不能这样写表达式book.subtitle.length，因为这样会报错，下面提供了两种避免出错的方法：

```
// 一种冗余但很易懂的方法
var len = undefined;
if (book) {
    if (book.subtitle) len = book.subtitle.length;
}

// 一种更简练的常用方法，获取subtitle的length属性或undefined
var len = book && book.subtitle && book.subtitle.length;
```

为了理解为什么这里的第二种方法可以避免类型错误异常，可以参照4.10.1节中关于 && 运算符的短路行为。

当然，给null和undefined设置属性也会报类型错误。给其他值设置属性也不总是成功，有一些属性是只读的，不能重新赋值，有一些对象不允许新增属性，但让人颇感意外的是，这些设置属性的失败操作不会报错：

```
// 内置构造函数的原型是只读的
Object.prototype = 0; // 赋值失败，但没报错，Object.prototype没有修改
```

这是一个历史遗留问题，这个bug在ECMAScript 5的严格模式中已经修复。在严格模式中，任何失败的属性设置操作都会抛出一个类型错误异常。

尽管属性赋值成功或失败的规律看起来很简单，但要描述清楚并不容易。在这些场景下给对象o设置属性p会失败：

- o中的属性p是只读的：不能给只读属性重新赋值（defineProperty()方法中有一个例外，可以对可配置的只读属性重新赋值）。
- o中的属性p是继承属性，且它是只读的：不能通过同名自有属性覆盖只读的继承属性。
- o中不存在自有属性p：o没有使用setter方法继承属性p，并且o的可扩展性（extensible attribute）是false（参照6.8.3节）。如果o中不存在p，而且没有setter方法可供调用，则p一定会添加至o中。但如果o不是可扩展的，那么在o中不能定义新属性。

6.3 删除属性

delete运算符（见4.13.3节）可以删除对象的属性。它的操作数应当是一个属性访问表达式。让人感到意外的是，delete只是断开属性和宿主对象的联系，而不会去操作属性中的属性[译注5]：

```
delete book.author;           // book不再有属性author
delete book["main title"];    // book也不再有属性"main title"
```

delete运算符只能删除自有属性，不能删除继承属性（要删除继承属性必须从定义这个属性的原型对象上删除它，而且这会影响到所有继承自这个原型的对象）。

译注5：a = {p:{x:1}}; b = a.p; delete a.p; 执行这段代码之后 b.x 的值依然是1。由于已经删除的属性的引用依然存在，因此在JavaScript的某些实现中，可能因为这种不严谨的代码而造成内存泄漏。所以在销毁对象的时候，要遍历属性中的属性，依次删除。

当delete表达式删除成功或没有任何副作用（比如删除不存在的属性）时，它返回true。如果delete后不是一个属性访问表达式，delete同样返回true：

```
o = {x:1};            // o有一个属性x，并继承属性toString
delete o.x;           // 删除x，返回true
delete o.x;           // 什么都没做（x已经不存在了），返回true
delete o.toString;    // 什么也没做（toString是继承来的），返回true
delete 1;             // 无意义，返回true
```

delete不能删除那些可配置性为false的属性（尽管可以删除不可扩展对象的可配置属性）。某些内置对象的属性是不可配置的，比如通过变量声明和函数声明创建的全局对象的属性。在严格模式中，删除一个不可配置属性会报一个类型错误。在非严格模式中（以及ECMAScript 3中），在这些情况下的delete操作会返回false：

```
delete Object.prototype;      // 不能删除，属性是不可配置的
var x = 1;                    // 声明一个全局变量
delete this.x;                // 不能删除这个属性
function f() {}               // 声明一个全局函数
delete this.f;                // 也不能删除全局函数
```

当在非严格模式中删除全局对象的可配值属性时，可以省略对全局对象的引用，直接在delete操作符后跟随要删除的属性名即可：

```
this.x = 1;   // 创建一个可配置的全局属性（没有用var）
delete x;     // 将它删除
```

然而在严格模式中，delete后跟随一个非法的操作数（比如x），则会报一个语法错误，因此必须显式指定对象及其属性：

```
delete x;           //在严格模式下报语法错误
delete this.x;      //正常工作
```

6.4 检测属性

JavaScript对象可以看做属性的集合，我们经常会检测集合中成员的所属关系——判断某个属性是否存在于某个对象中。可以通过in运算符、hasOwnPreperty()和propertyIsEnumerable()方法来完成这个工作，甚至仅通过属性查询也可以做到这一点。

in运算符的左侧是属性名（字符串），右侧是对象。如果对象的自有属性或继承属性中包含这个属性则返回true：

```
var o = { x: 1 }
"x" in o;           // true：“x”是o的属性
"y" in o;           // false：“y”不是o的属性
"toString" in o;    // true：o继承toString 属性
```

对象的hasOwnProperty()方法用来检测给定的名字是否是对象的自有属性。对于继承属性它将返回false：

```
var o = { x: 1 }
o.hasOwnProperty("x");                  // true：o有一个自有属性x
o.hasOwnProperty("y");                  // false：o中不存在属性y
o.hasOwnProperty("toString");           // false：toString是继承属性
```

propertyIsEnumerable()是hasOwnProperty()的增强版，只有检测到是自有属性且这个属性的可枚举性（enumerable attribute）为true时它才返回true。某些内置属性是不可枚举的。通常由JavaScript代码创建的属性都是可枚举的，除非在ECMAScript 5中使用一个特殊的方法来改变属性的可枚举性，随后会提到：

```
var o = inherit({ y: 2 });
o.x = 1;
o.propertyIsEnumerable("x"); // true: o有一个可枚举的自有属性x
o.propertyIsEnumerable("y"); // false: y是继承来的
Object.prototype.propertyIsEnumerable("toString"); // false: 不可枚举
```

除了使用in运算符之外，另一种更简便的方法是使用“!==”判断一个属性是否是undefined：

```
var o = { x: 1 }
o.x !== undefined;              //true: o中有属性x
o.y !== undefined;              // false: o中没有属性y
o.toString !== undefined;       //true: o继承了toString属性
```

然而有一种场景只能使用in运算符而不能使用上述属性访问的方式。in可以区分不存在的属性和存在但值为undefined的属性。例如下面的代码：

```
var o = { x: undefined }        // 属性被显式赋值为undefined
o.x !== undefined               // false：属性存在，但值为undefined
o.y !== undefined               // false：属性不存在
"x" in o                        // true：属性存在
"y" in o                        // false：属性不存在
delete o.x;                     // 删除了属性x
"x" in o                        // false：属性不再存在
```

注意，上述代码中使用的是“!==”运算符，而不是“!=”。“!==”可以区分undefined和null。有时则不必作这种区分：

```
// 如果o中含有属性x，且x的值不是null或undefined，o.x乘以2.
if (o.x != null) o.x *= 2;
// 如果o中含有属性x，且x的值不能转换为false，o.x乘以2.
// 如果x是undefined、null、false、" "、0或NaN，则它保持不变
if (o.x) o.x *= 2;
```

6.5 枚举属性

除了检测对象的属性是否存在，我们还会经常遍历对象的属性。通常使用for/in循环遍历，ECMAScript 5提供了两个更好用的替代方案。

5.5.4节讨论过for/in循环，for/in循环可以在循环体中遍历对象中所有可枚举的属性（包括自有属性和继承的属性），把属性名称赋值给循环变量。对象继承的内置方法不可枚举的，但在代码中给对象添加的属性都是可枚举的（除非用下文中提到的一个方法将它们转换为不可枚举的）。例如：

```
var o = {x:1, y:2, z:3};                    // 三个可枚举的自有属性
o.propertyIsEnumerable("toString")          // =>false，不可枚举
for(p in o)                                 // 遍历属性
console.log(p);                             // 输出x、y和z，不会输出toString
```

有许多实用工具库给Object.prototype添加了新的方法或属性，这些方法和属性可以被所有对象继承并使用。然而在ECMAScript 5标准之前，这些新添加的方法是不能定义为不可枚举的，因此它们都可以在for/in循环中枚举出来。为了避免这种情况，需要过滤for/in循环返回的属性，下面两种方式是最常见的：

```
for(p in o) {
   if (!o.hasOwnProperty(p)) continue;          // 跳过继承的属性
}
for(p in o) {
   if (typeof o[p] === "function") continue;    // 跳过方法
}
```

例6-2定义了一些有用的工具函数来操控对象的属性，这些函数用到了for/in循环。实际上extend()函数经常出现在JavaScript实用工具库中[注2]。

例6-2：用来枚举属性的对象工具函数

```
/*
 * 把p中的可枚举属性复制到o中，并返回o
 * 如果o和p中含有同名属性，则覆盖o中的属性
 * 这个函数并不处理getter和setter以及复制属性
 */
function extend(o, p) {
        for (prop in p) {           // 遍历p中的所有属性
            o[prop] = p[prop];      // 将属性添加至o中
        }
        return o;
}
```

注2：这里所实现的extend()逻辑虽然正确，但并不能弥补IE中有一些众所周知的bug，在例8-3中会有更健壮的extend()实现。

```
/*
 * 将p中的可枚举属性复制至o中，并返回o
 * 如果o和p中有同名的属性，o中的属性将不受影响
 * 这个函数并不处理getter和setter以及复制属性
 */
function merge(o, p) {
    for (prop in p) {                                   // 遍历p中的所有属性
        if (o.hasOwnProperty[prop]) continue;           // 过滤掉已经在o中存在的属性
        o[prop] = p[prop];                              // 将属性添加至o中
    }
    return o;
}

/*
 * 如果o中的属性在p中没有同名属性，则从o中删除这个属性
 * 返回o
 */
function restrict(o, p) {
    for (prop in o) {                           // 遍历o中的所有属性
        if (! (prop in p)) delete o[prop]; // 如果在p中不存在，则删除之
    }
    return o;
}

/*
 * 如果o中的属性在p中存在同名属性，则从o中删除这个属性
 * 返回o
 */
function subtract(o, p) {
    for (prop in p) {     // 遍历p中的所有属性
         delete o[prop];  // 从o中删除（删除一个不存在的属性不会报错）

    }
    return o;
}

/*
 * 返回一个新对象，这个对象同时拥有o的属性和p的属性
 * 如果o和p中有重名属性，使用p中的属性值
 */
function union(o, p) { return extend(extend({},o), p);}

/*
 * 返回一个新对象，这个对象拥有同时在o和p中出现的属性
 * 很像求o和p的交集，但p中属性的值被忽略
 */
function intersection(o, p) { return restrict(extend({},o), p);}

/*
 * 返回一个数组，这个数组包含的是o中可枚举的自有属性的名字
 */
function keys(o) {
    if (typeof o !== "object") throw TypeError();  // 参数必须是对象
    var result = [];                               // 将要返回的数组
    for (var prop in o) {                          // 遍历所有可枚举的属性
         if (o.hasOwnProperty(prop))               // 判断是否是自有属性
```

```
            result.push(prop);              // 将属性名添加至数组中
    }
    return result;                          // 返回这个数组
}
```

除了for/in循环之外，ECMAScript 5定义了两个用以枚举属性名称的函数。第一个是Object.keys()，它返回一个数组，这个数组由对象中可枚举的自有属性的名称组成，它的工作原理和例6-2中的工具函数keys()类似。

ECMAScript 5中第二个枚举属性的函数是Object.getOwnPropertyNames()，它和Ojbect.keys()类似，只是它返回对象的所有自有属性的名称，而不仅仅是可枚举的属性。在ECMAScript 3中是无法实现的类似的函数的，因为ECMAScript 3中没有提供任何方法来获取对象不可枚举的属性。

6.6 属性getter和setter

我们知道，对象属性是由名字、值和一组特性（attribute）构成的。在ECMAScript 5[注3]中，属性值可以用一个或两个方法替代，这两个方法就是getter和setter。由getter和setter定义的属性称做“存取器属性”（accessor property），它不同于“数据属性”（data property），数据属性只有一个简单的值。

当程序查询存取器属性的值时，JavaScript调用getter方法（无参数）。这个方法的返回值就是属性存取表达式的值。当程序设置一个存取器属性的值时，JavaScript调用setter方法，将赋值表达式右侧的值当做参数传入setter。从某种意义上讲，这个方法负责“设置”属性值。可以忽略setter方法的返回值。

和数据属性不同，存取器属性不具有可写性（writable attribute）。如果属性同时具有getter和setter方法，那么它是一个读/写属性。如果它只有getter方法，那么它是一个只读属性。如果它只有setter方法，那么它是一个只写属性（数据属性中有一些例外），读取只写属性总是返回undefined。

定义存取器属性最简单的方法是使用对象直接量语法的一种扩展写法：

```
var o = {
    // 普通的数据属性
    data_prop: value,

    // 存取器属性都是成对定义的函数
    get accessor_prop() { /*这里是函数体 */ },
    set accessor_prop(value) { /* 这里是函数体*/ }
```

注3：　包括除了IE之外的最新主流浏览器的ECMAScript 3的实现。

```
};
```

存取器属性定义为一个或两个和属性同名的函数，这个函数定义没有使用function关键字，而是使用get和（或）set。注意，这里没有使用冒号将属性名和函数体分隔开，但在函数体的结束和下一个方法或数据属性之间有逗号分隔。例如，思考下面这个表示2D笛卡尔点坐标[译注6]的对象。它有两个普通的属性x和y分别表示对应点的X坐标和Y坐标，它还有两个等价的存取器属性用来表示点的极坐标：

```
var p = {
    //x和y是普通的可读写的数据属性
    x: 1.0,
    y: 1.0,

    // r是可读写的存取器属性，它有getter和setter.
    // 函数体结束后不要忘记带上逗号
    get r() { return Math.sqrt(this.x*this.x + this.y*this.y); },
    set r(newvalue) {
      var oldvalue = Math.sqrt(this.x*this.x + this.y*this.y);
      var ratio = newvalue/oldvalue;
      this.x *= ratio;
      this.y *= ratio;
    },
    //theta是只读存取器属性，它只有getter方法
    get theta() { return Math.atan2(this.y, this.x); }
};
```

注意在这段代码中getter和setter里this关键字的用法。JavaScript把这些函数当做对象的方法来调用，也就是说，在函数体内的this指向表示这个点的对象，因此，r属性的getter方法可以通过this.x和this.y引用x和y属性。8.2.2节会对方法和this关键字做更详尽的讲述。

和数据属性一样，存取器属性是可以继承的，因此可以将上述代码中的对象p当做另一个“点”的原型。可以给新对象定义它的x和y属性，但r和theta属性是继承来的：

```
var q = inherit(p);   // 创建一个继承getter和setter的新对象
q.x = 1, q.y = 1;     // 给q添加两个属性
console.log(q.r);     // 可以使用继承的存取器属性
console.log(q.theta);
```

这段代码使用存取器属性定义API，API提供了表示同一组数据的两种方法（笛卡尔坐标系表示法和极坐标系表示法）。还有很多场景可以用到存取器属性，比如智能检测属性的写入值以及在每次属性读取时返回不同值：

```
// 这个对象产生严格自增的序列号
```

译注6：笛卡尔坐标系就是直角坐标系和斜角坐标系的统称。相交于原点的两条数轴，构成了平面放射坐标系。

```
var serialnum = {
    // 这个数据属性包含下一个序列号
    // $符号暗示这个属性是一个私有属性
    $n: 0,

    // 返回当前值，然后自增
    get next() { return this.$n++; },

    // 给n设置新的值，但只有当它比当前值大时才设置成功
    set next(n) {
        if (n >= this.$n) this.$n = n;
        else throw "序列号的值不能比当前值小";
    }
};
```

最后我们再来看一个例子，这个例子使用getter方法实现一种“神奇”的属性：

```
// 这个对象有一个可以返回随机数的存取器属性
// 例如，表达式"random.octet"产生一个随机数
// 每次产生的随机数都在0～255之间
var random = {
    get octet() { return Math.floor(Math.random()*256); },
    get uint16() { return Math.floor(Math.random()*65536); },
    get int16() { return Math.floor(Math.random()*65536)-32768; }
};
```

本节介绍了如何给对象直接量定义存取器属性。下一节会介绍如何给一个已经存在的对象添加一个存取器属性。

6.7 属性的特性

除了包含名字和值之外，属性还包含一些标识它们可写、可枚举和可配置的特性。在ECMAScript 3中无法设置这些特性，所有通过ECMAScript 3的程序创建的属性都是可写的、可枚举的和可配置的，且无法对这些特性做修改。本节将讲述ECMAScript 5 中查询和设置这些属性特性的API。这些API对于库的开发者来说非常重要，因为：

- 可以通过这些API给原型对象添加方法，并将它们设置成不可枚举的，这让它们看起来更像内置方法。
- 可以通过这些API给对象定义不能修改或删除的属性，借此“锁定”这个对象。

在本节里，我们将存取器属性的getter和setter方法看成是属性的特性。按照这个逻辑，我们也可以把数据属性的值同样看做属性的特性。因此，可以认为一个属性包含一个名字和4个特性。数据属性的4个特性分别是它的值（value）、可写性（writable）、可枚举性（enumerable）和可配置性（configurable）。存取器属性不具有值（value）特性和

可写性，它们的可写性是由setter方法存在与否决定的。因此存取器属性的4个特性是读取（get）、写入（set）、可枚举性和可配置性。

为了实现属性特性的查询和设置操作，ECMAScript 5中定义了一个名为“属性描述符”（property descriptor）的对象，这个对象代表那4个特性。描述符对象的属性和它们所描述的属性特性是同名的。因此，数据属性的描述符对象的属性有value、writable、enumerable和configurable。存取器属性的描述符对象则用get属性和set属性代替value和writable。其中writable、enumerable和configurable都是布尔值，当然，get属性和set属性是函数值。

通过调用Object.getOwnPropertyDescriptor()可以获得某个对象特定属性的属性描述符：

```
// 返回 {value: 1, writable:true, enumerable:true, configurable:true}
Object.getOwnPropertyDescriptor({x:1}, "x");

// 查询上文中定义的randam对象的octet属性
// 返回 { get: /*func*/, set:undefined, enumerable:true, configurable:true}
Object.getOwnPropertyDescriptor(random, "octet");

// 对于继承属性和不存在的属性，返回undefined
Object.getOwnPropertyDescriptor({}, "x");              // undefined，没有这个属性
Object.getOwnPropertyDescriptor({}, "toString");       // undefined，  继承属性
```

从函数名字就可以看出，Object.getOwnPropertyDescriptor()只能得到自有属性的描述符。要想获得继承属性的特性，需要遍历原型链（参照6.8.1节的Object.getPrototypeOf()）。

要想设置属性的特性，或者想让新建属性具有某种特性，则需要调用Object.definePeoperty()，传入要修改的对象、要创建或修改的属性的名称以及属性描述符对象：

```
var o = {}; // 创建一个空对象
// 添加一个不可枚举的数据属性x，并赋值为1
Object.defineProperty(o, "x", { value : 1,
                                writable: true,
                                enumerable: false,
                                configurable: true});

// 属性是存在的，但不可枚举
o.x;            // => 1
Object.keys(o) // => []

// 现在对属性x做修改，让它变为只读
Object.defineProperty(o, "x", { writable: false });

// 试图更改这个属性的值
o.x = 2; // 操作失败但不报错，而在严格模式中抛出类型错误异常
```

```
o.x // => 1

// 属性依然是可配置的，因此可以通过这种方式对它进行修改：
Object.defineProperty(o, "x", { value: 2 });
o.x // => 2

// 现在将x从数据属性修改为存取器属性
Object.defineProperty(o, "x", { get: function() { return 0; } });
o.x // => 0
```

传入Object.defineProperty()的属性描述符对象不必包含所有4个特性。对于新创建的属性来说，默认的特性值是false或undefined。对于修改的已有属性来说，默认的特性值没有做任何修改。注意，这个方法要么修改已有属性要么新建自有属性，但不能修改继承属性。

如果要同时修改或创建多个属性，则需要使用Object.defineProperties()。第一个参数是要修改的对象，第二个参数是一个映射表，它包含要新建或修改的属性的名称，以及它们的属性描述符，例如：

```
var p = Object.defineProperties({}, {
    x: { value: 1, writable: true, enumerable:true, configurable:true },
    y: { value: 1, writable: true, enumerable:true, configurable:true },
    r: {
        get: function() { return Math.sqrt(this.x*this.x + this.y*this.y) },
        enumerable:true,
        configurable:true
    }
});
```

这段代码从一个空对象开始，然后给它添加两个数据属性和一个只读存取器属性。最终Object.defineProperties()返回修改后的对象（和Object.defineProperty()一样）。

对于那些不允许创建或修改的属性来说，如果用Object.defineProperty() 和 Object.defineProperties()对其操作（新建或修改）就会抛出类型错误异常，比如，给一个不可扩展的对象（参照6.8.3节）新增属性就会抛出类型错误异常。造成这些方法抛出类型错误异常的其他原因则和特性本身相关。可写性控制着对值特性的修改。可配置性控制着对其他特性（包括属性是否可以删除）的修改。然而规则远不止这么简单，例如，如果属性是可配置的话，则可以修改不可写属性的值。同样，如果属性是不可配置的，仍然可以将可写属性修改为不可写属性。下面是完整的规则，任何对 Object.defineProperty()或Object.defineProperties()违反规则的使用都会抛出类型错误异常：

- 如果对象是不可扩展的，则可以编辑已有的自有属性，但不能给它添加新属性。
- 如果属性是不可配置的，则不能修改它的可配置性和可枚举性。

- 如果存取器属性是不可配置的，则不能修改其getter和setter方法，也不能将它转换为数据属性。
- 如果数据属性是不可配置的，则不能将它转换为存取器属性。
- 如果数据属性是不可配置的，则不能将它的可写性从false修改为true，但可以从true修改为false。
- 如果数据属性是不可配置且不可写的，则不能修改它的值。然而可配置但不可写属性的值是可以修改的（实际上是先将它标记为可写的，然后修改它的值，最后转换为不可写的）。

例6-2中实现了extend()函数，这个函数把一个对象的属性复制到另一个对象中。这个函数只是简单地复制属性名和值，没有复制属性的特性，而且也没有复制存取器属性的getter和setter方法，只是将它们简单地转换为静态的数据属性。例6-3给出了改进的extend()，它使用Object.getOwnPropertyDescriptor()和Object.defineProperty()对属性的所有特性进行复制。新的extend()作为不可枚举属性添加到Object.prototype中，因此它是Object上定义的新方法，而不是一个独立的函数。

例6-3：复制属性的特性

```
/*
 * 给Object.prototype添加一个不可枚举的extend()方法
 * 这个方法继承自调用它的对象，将作为参数传入的对象的属性一一复制
 * 除了值之外，也复制属性的所有特性，除非在目标对象中存在同名的属性，
 * 参数对象的所有自有对象（包括不可枚举的属性）也会一一复制。
 *
 */
Object.defineProperty(Object.prototype,
    "extend",                       // 定义 Object.prototype.extend
    {
        writable: true,
        enumerable: false,          // 将其定义为不可枚举的
        configurable: true,
        value: function(o) {        // 值就是这个函数
            // 得到所有的自有属性，包括不可枚举属性
            var names = Object.getOwnPropertyNames(o);
            // 遍历它们
            for(var i = 0; i < names.length; i++) {
                // 如果属性已经存在，则跳过
                if (names[i] in this) continue;
                // 获得o中的属性的描述符
                var desc = Object.getOwnPropertyDescriptor(o,names[i]);
                // 用它给this创建一个属性
                Object.defineProperty(this, names[i], desc);
            }
        }
    });
```

getter和setter的老式API

可以通过6.6节描述的对象直接量语法给新对象定义存取器属性，但不能查询属性的getter和setter方法或给已有的对象添加新的存取器属性。在ECMAScript 5中，可以通过Object.getOwnPropertyDescriptor()和Object.defineProperty()来完成这些工作。

在ECMAScript 5标准被采纳之前，大多数JavaScript的实现（IE浏览器除外）已经可以支持对象直接量语法中的get和set写法。这些实现提供了非标准的老式API用来查询和设置getter和setter。这些API由4个方法组成，所有对象都拥有这些方法。__lookupGetter__()和__lookupSetter__()用以返回一个命名属性的getter和setter方法。__defineGetter__() 和 __defineSetter__()用以定义getter和setter，这两个函数的第一个参数是属性名字，第二个参数是getter和setter方法。这4个方法都是以两条下划线作前缀，两条下划线作后缀，以表明它们是非标准的方法。本书第三部分没有对非标准的方法做介绍。

6.8 对象的三个属性

每一个对象都有与之相关的原型（prototype）、类（class）和可扩展性（extensible attribute）。下面几节将会展开讲述这些属性有什么作用，以及如何查询和设置它们。

6.8.1 原型属性

对象的原型属性是用来继承属性的（关于原型和原型继承的更多内容请参照6.1.3节和6.2.2节），这个属性如此重要，以至于我们经常把“o的原型属性”直接叫做“o的原型”。

原型属性是在实例对象创建之初就设置好的，回想一下6.1.3节提到的，通过对象直接量创建的对象使用Object.prototype作为它们的原型。通过new创建的对象使用构造函数的prototype属性作为它们的原型。通过Object.create()创建的对象使用第一个参数（也可以是null）作为它们的原型。

在ECMAScript 5中，将对象作为参数传入Object.getPrototypeOf()可以查询它的原型。在ECMAScript 3中，则没有与之等价的函数，但经常使用表达式o.constructor.prototype来检测一个对象的原型。通过new表达式创建的对象，通常继承一个constructor属性，这个属性指代创建这个对象的构造函数。更多细节将会放在9.2节进一步讨论，9.2节还解释了使用这种方法来检测对象原型的方式并不可靠的原因。注意，通过对象直接量或Object.create()创建的对象包含一个名为constructor的属性，这个属性指代Object()构造函数。因此，constructor.prototype才是对象直接量的真正的原型，但对于通过Object.create()创建的对象则往往不是这样。

要想检测一个对象是否是另一个对象的原型（或处于原型链中），请使用isPrototypeOf()方法。例如，可以通过p.isPrototypeOf(o)来检测p是否是o的原型：

```
var p = {x:1};                          // 定义一个原型对象
var o = Object.create(p);               // 使用这个原型创建一个对象
p.isPrototypeOf(o)                      // => true:o继承自p
Object.prototype.isPrototypeOf(o)       //=> true:p继承自Object.prototype
```

需要注意的是，isPrototypeOf()函数实现的功能和instanceof运算符非常类似（参照4.9.4节）。

Mozilla实现的JavaScript（包括早些年的Netscape）对外暴露了一个专门命名为__proto__的属性，用以直接查询/设置对象的原型。但并不推荐使用__proto__，因为尽管Safari和Chrome的当前版本都支持它，但IE和Opera还未实现它（可能以后也不会实现）。实现了ECMAScript 5的Firefox版本依然支持__proto__，但对修改不可扩展对象的原型做了限制。

6.8.2 类属性

对象的类属性（class attribute）是一个字符串，用以表示对象的类型信息。ECMAScript 3和ECMAScript 5都未提供设置这个属性的方法，并只有一种间接的方法可以查询它。默认的toString()方法（继承自Object.prototype）返回了如下这种格式的字符串：

```
[object class]
```

因此，要想获得对象的类，可以调用对象的toString()方法，然后提取已返回字符串的第8个到倒数第二个位置之间的字符。不过让人感觉棘手的是，很多对象继承的toString()方法重写了，为了能调用正确的toString()版本，必须间接地调用Function.call()方法（参照8.7.3节）。例6-4中的classof()函数可以返回传递给它的任意对象的类：

例6-4：classof()函数

```
function classof(o) {
    if (o === null) return "Null";
    if (o === undefined) return "Undefined";
    return Object.prototype.toString.call(o).slice(8,-1);
}
```

classof()函数可以传入任何类型的参数。数字、字符串和布尔值可以直接调用toString()方法，就和对象调用toString()方法一样[译注7]，并且这个函数包含了对null

译注7：实际上是这些类型的变量调用toString()方法，而不是通过它们的直接量调用toString()，比如1.toString()是不对的，而是要先声明变量var a = 1；然后调用a.toString()。

和undefined的特殊处理（在ECMAScript 5中不需要对这些特殊情况做处理）。通过内置构造函数（比如Array和Date）创建的对象包含“类属性”（class attribute），它与构造函数名称相匹配。宿主对象也包含有意义的“类属性”，但这和具体的JavaScript实现有关。通过对象直接量和Object.create创建的对象的类属性是“Object”，那些自定义构造函数创建的对象也是一样，类属性也是“Object”，因此对于自定义的类来说，没办法通过类属性来区分对象的类：

```
classof(null)         // => "Null"
classof(1)            // => "Number"
classof("")           // => "String"
classof(false)        // => "Boolean"
classof({})           // => "Object"
classof([])           // => "Array"
classof(/./)          // => "Regexp"
classof(new Date())   // => "Date"
classof(window)       // => "Window"(这是客户端宿主对象)
function f() {};      // 定义一个自定义构造函数
classof(new f());     // => "Object"
```

6.8.3可扩展性

对象的可扩展性用以表示是否可以给对象添加新属性。所有内置对象和自定义对象都是显式可扩展的，宿主对象的可扩展性是由JavaScript引擎定义的。在ECMAScript 5中，所有的内置对象和自定义对象都是可扩展的，除非将它们转换为不可扩展的，同样，宿主对象的可扩展性也是由实现ECMAScript 5的JavaScript引擎定义的。

ECMAScript 5定义了用来查询和设置对象可扩展性的函数。通过将对象传入Object.esExtensible()，来判断该对象是否是可扩展的。如果想将对象转换为不可扩展的，需要调用Object.preventExtensions()，将待转换的对象作为参数传进去。注意，一旦将对象转换为不可扩展的，就无法再将其转换回可扩展的了。同样需要注意的是，preventExtensions()只影响到对象本身的可扩展性。如果给一个不可扩展的对象的原型添加属性，这个不可扩展的对象同样会继承这些新属性。

可扩展属性的目的是将对象“锁定”，以避免外界的干扰。对象的可扩展性通常和属性的可配值性与可写性配合使用，ECMAScript 5定义的一些函数可以更方便地设置多种属性。

Object.seal()和Object.preventExtensions()类似，除了能够将对象设置为不可扩展的，还可以将对象的所有自有属性都设置为不可配置的。也就是说，不能给这个对象添加新属性，而且它已有的属性也不能删除或配置，不过它已有的可写属性依然可以设置。对于那些已经封闭（sealed）起来的对象是不能解封的。可以使用Object.isSealed()来检测对象是否封闭。

Object.freeze()将更严格地锁定对象——“冻结”（frozen）。除了将对象设置为不可扩展的和将其属性设置为不可配置的之外，还可以将它自有的所有数据属性设置为只读（如果对象的存取器属性具有setter方法，存取器属性将不受影响，仍可以通过给属性赋值调用它们）。使用Object.isFrozen()来检测对象是否冻结。

Object.preventExtensions()、Object.seal()和Object.freeze()都返回传入的对象，也就是说，可以通过函数嵌套的方式调用它们：

```
// 创建一个封闭对象，包括一个冻结的原型和一个不可枚举的属性
var o = Object.seal(Object.create(Object.freeze({x:1}),
                                  {y: {value: 2, writable: true}}));
```

6.9 序列化对象

对象序列化（serialization）是指将对象的状态转换为字符串，也可将字符串还原为对象。ECMAScript 5提供了内置函数JSON.stringify()和JSON.parse()用来序列化和还原JavaScript对象。这些方法都使用JSON作为数据交换格式，JSON的全称是“JavaScript Object Notation”——JavaScript对象表示法，它的语法和JavaScript对象与数组直接量的语法非常相近：

```
o = {x:1, y:{z:[false,null,""]}};       // 定义一个测试对象
s = JSON.stringify(o);                  // s是 '{"x":1,"y":{"z":[false,null,""]}}'
p = JSON.parse(s);                      // p是o的深拷贝
```

ECMAScript 5中的这些函数的本地实现和*http://json.org/json2.js*中的公共域ECMAScript 3版本的实现非常类似，或者说完全一样，因此可以通过引入*json2.js*模块在ECMAScript 3的环境中使用ECMAScript 5中的这些函数。

JSON的语法是JavaScript语法的子集，它并不能表示JavaScript里的所有值。支持对象、数组、字符串、无穷大数字、true、false和null，并且它们可以序列化和还原。NaN、Infinity和-Infinity序列化的结果是null，日期对象序列化的结果是ISO格式的日期字符串（参照Date.toJSON()函数），但JSON.parse()依然保留它们的字符串形态，而不会将它们还原为原始日期对象。函数、RegExp、Error对象和undefined值不能序列化和还原。JSON.stringify()只能序列化对象可枚举的自有属性。对于一个不能序列化的属性来说，在序列化后的输出字符串中会将这个属性省略掉。JSON.stringify()和JSON.parse()都可以接收第二个可选参数，通过传入需要序列化或还原的属性列表来定制自定义的序列化或还原操作。第三部分有关于这些函数的详细文档。

6.10 对象方法

上文已经讨论过，所有的JavaScript对象都从Object.prototype继承属性（除了那些不通过原型显式创建的对象）。这些继承属性主要是方法，因为JavaScript程序员普遍对继承方法更感兴趣。我们已经讨论过 hasOwnProperty()、propertyIsEnumerable()和 isPrototypeOf()这三个方法，以及在Object构造函数里定义的静态函数Object.create() 和 Object.getPrototypeOf()等。本节将对定义在Object.prototype里的对象方法展开讲解，这些方法非常好用而且使用广泛，但一些特定的类会重写这些方法。

6.10.1 toString()方法

toString()方法没有参数，它将返回一个表示调用这个方法的对象值的字符串。在需要将对象转换为字符串的时候，JavaScript都会调用这个方法。比如，当使用“+”运算符连接一个字符串和一个对象时或者在希望使用字符串的方法中使用了对象时都会调用toString()。

默认的toString()方法的返回值带有的信息量很少（尽管它在检测对象的类型时非常有用，参照6.8.2节），例如，下面这行代码的计算结果为字符串“[object Object]”：

```
var s = { x:1, y:1 }.toString();
```

由于默认的toString()方法并不会输出很多有用的信息，因此很多类都带有自定义的toString()。例如，当数组转换为字符串的时候，结果是一个数组元素列表，只是每个元素都转换成了字符串，再比如，当函数转换为字符串的时候，得到函数的源代码。第三部分有关于toString()的详细文档说明，比如Array.toString()、Date.toString()以及Function.toString()。

9.6.3节介绍如何给自定义类重写toString()方法。

6.10.2 toLocaleString()方法

除了基本的toString()方法之外，对象都包含toLocaleString()方法，这个方法返回一个表示这个对象的本地化字符串。Object中默认的toLocaleString()方法并不做任何本地化自身的操作，它仅调用toString()方法并返回对应值。Date和Number类对toLocaleString()方法做了定制，可以用它对数字、日期和时间做本地化的转换。Array类的toLocaleString()方法和toString()方法很像，唯一的不同是每个数组元素会调用toLocaleString()方法转换为字符串，而不是调用各自的toString()方法。

6.10.3 toJSON()方法

Object.prototype实际上没有定义toJSON()方法，但对于需要执行序列化的对象来说，JSON.stringify()方法会调用toJSON()方法。如果在待序列化的对象中存在这个方法，则调用它，返回值即是序列化的结果，而不是原始的对象。具体示例参见Date.toJSON()。

6.10.4 valueOf() 方法

valueOf()方法和toString()方法非常类似，但往往当JavaScript需要将对象转换为某种原始值而非字符串的时候才会调用它，尤其是转换为数字的时候。如果在需要使用原始值的上下文中使用了对象，JavaScript就会自动调用这个方法。默认的valueOf()方法不足为奇，但有些内置类自定义了valueOf()方法（比如Date.valueOf()），9.6.3节讨论如何给自定义对象类型定义valueOf()方法。