第9章

类和模块

第6章详细介绍了JavaScript对象，每个JavaScript对象都是一个属性集合，相互之间没有任何联系。在JavaScript中也可以定义对象的类，让每个对象都共享某些属性，这种“共享”的特性是非常有用的。类的成员或实例都包含一些属性，用以存放或定义它们的状态，其中有些属性定义了它们的行为（通常称为方法）。这些行为通常是由类定义的，而且为所有实例所共享。例如，假设有一个名为Complex的类用来表示复数，同时还定义了一些复数运算。一个Complex实例应当包含复数的实部和虚部（状态），同样Complex类还会定义复数的加法和乘法操作（行为）。

在JavaScript中，类的实现是基于其原型继承机制的。如果两个实例都从同一个原型对象上继承了属性，我们说它们是同一个类的实例。JavaScript原型和继承在6.1.3节和6.2.2节中有详细讨论，为了更好地理解本章的内容，请务必首先阅读这两个章节。本章将会在9.1节中对原型做进一步讨论。

如果两个对象继承自同一个原型，往往意味着（但不是绝对）它们是由同一个构造函数创建并初始化的。我们已经在4.6节、6.2节和8.2.3节中详细讲解了构造函数，9.2节会有进一步讨论。

如果你对诸如Java和C++这种强类型[译注1]的面向对象编程比较熟悉，你会发现JavaScript中的类和Java以及C++中的类有很大不同。尽管在写法上类似，而且在JavaScript中也能“模拟”出很多经典的类的特性[译注2]，但是最好要理解JavaScript的类和基于原型的继承

译注1：强/弱类型是指类型检查的严格程度，为所有变量指定数据类型称为“强类型”。

译注2：比如传统类的封装、继承和多态。

机制，以及和传统的Java（当然还有类似Java的语言）的类和基于类的继承机制的不同之处。9.3节展示了如何在JavaScript中实现经典的类。

JavaScript中类的一个重要特性是“动态可继承”（dynamically extendable），9.4节详细解释这一特性。我们可以将类看做是类型，9.5节讲解检测对象的类的几种方式，该节同样介绍一种编程哲学——“鸭式辩型”（duck-typing），它弱化了对象的类型，强化了对象的功能。

在讨论了JavaScript中所有基本的面向对象编程特性之后，我们将关注点从抽象的概念转向一些实例。9.6节介绍两种非常重要的实现类的方法，包括很多实现面向对象的技术，这些技术可以很大程度上增强类的功能。9.7节展示（包含很多示例代码）如何实现类的继承，包括如何在JavaScript中实现类的继承。9.8节讲解如何使用ECMAScript 5中的新特性来实现类以及面向对象编程。

定义类是模块开发和重用代码的有效方式之一，本章最后一节会集中讨论JavaScript中的模块。

9.1 类和原型

在JavaScript中，类的所有实例对象都从同一个原型对象上继承属性。因此，原型对象是类的核心。在例6-1中定义了`inherit()`函数，这个函数返回一个新创建的对象，后者继承自某个原型对象。如果定义一个原型对象，然后通过`inherit()`函数创建一个继承自它的对象，这样就定义了一个JavaScript类。通常，类的实例还需要进一步的初始化，通常是通过定义一个函数来创建并初始化这个新对象，参照例9-1。例9-1给一个表示“值的范围”的类定义了原型对象，还定义了一个“工厂”函数[译注3]用以创建并初始化类的实例。

例 9-1：一个简单的JavaScript 类

```
// range.js: 实现一个能表示值的范围的类

// 这个工厂方法返回一个新的"范围对象"
function range(from, to) {
    // 使用inherit()函数来创建对象，这个对象继承自在下面定义的原型对象
    // 原型对象作为函数的一个属性存储，并定义所有"范围对象"所共享的方法（行为）
    var r = inherit(range.methods);

    // 存储新的"范围对象"的起始位置和结束位置（状态）
    // 这两个属性是不可继承的，每个对象都拥有唯一的属性
    r.from = from;
    r.to = to;

    // 返回这个新创建的对象
```

译注3：参照：*http://zh.wikipedia.org/zh/工厂方法*。

```
    return r;
}

// 原型对象定义方法，这些方法为每个范围对象所继承
range.methods = {
    // 如果x在范围内，则返回true；否则返回false
    // 这个方法可以比较数字范围，也可以比较字符串和日期范围
    includes: function (x) {
    return this.from <= x && x <= this.to; },

    // 对于范围内的每个整数都调用一次f
    // 这个方法只可用做数字范围
    foreach: function (f) {
        for (var x = Math.ceil(this.from); x <= this.to; x++) f(x);
    },
    // 返回表示这个范围的字符串
    toString: function () {return "(" + this.from + "..." + this.to + ")";}
};

// 这里是使用"范围对象"的一些例子
var r = range(1, 3);      // 创建一个范围对象
r.includes(2);            // => true: 2 在这个范围内
r.foreach(console.log);   // 输出 1 2 3
console.log(r);           // 输出 (1...3)
```

在例9-1中有一些代码是没有用的。这段代码定义了一个工厂方法range()，用来创建新的范围对象。我们注意到，这里给range()函数定义了一个属性range.methods，用以快捷地存放定义类的原型对象。把原型对象挂在函数上没什么大不了，但也不是惯用做法。再者，注意range()函数给每个范围对象都定义了from和to属性，用以定义范围的起始位置和结束位置，这两个属性是非共享的，当然也是不可继承的。最后，注意在range.methods中定义的那些可共享、可继承的方法都用到了from和to属性，而且使用了this关键字，为了指代它们，二者使用this关键字来指代调用这个方法的对象。任何类的方法都可以通过this的这种基本用法来读取对象的属性。

9.2 类和构造函数

例9-1展示了在JavaScript中定义类的其中一种方法。但这种方法并不常用，毕竟它没有定义构造函数，构造函数是用来初始化新创建的对象的。8.2.3节已经讲到，使用关键字new来调用构造函数。使用new调用构造函数会自动创建一个新对象，因此构造函数本身只需初始化这个新对象的状态即可。调用构造函数的一个重要特征是，构造函数的prototye属性被用做新对象的原型。这意味着通过同一个构造函数创建的所有对象都继承自一个相同的对象，因此它们都是同一个类的成员。例9-2对例9-1中的“范围类”做了修改，使用构造函数代替工厂函数：

例9-2：使用构造函数来定义“范围类”

```
// range2.js: 表示值的范围的类的另一种实现

// 这是一个构造函数，用以初始化新创建的"范围对象"
// 注意，这里并没有创建并返回一个对象，仅仅是初始化
function Range(from, to) {
    // 存储"范围对象"的起始位置和结束位置（状态）
    // 这两个属性是不可继承的，每个对象都拥有唯一的属性
    this.from = from;
    this.to = to;
}

// 所有的"范围对象"都继承自这个对象
// 注意，属性的名字必须是"prototype"
Range.prototype = {
    // 如果x在范围内，则返回true；否则返回false
    // 这个方法可以比较数字范围，也可以比较字符串和日期范围
    includes: function (x) { return this.from <= x && x <= this.to; },
    //对于范围内的每个整数都调用一次f
    // 这个方法只可用于数字范围
    foreach: function (f) {
        for (var x = Math.ceil(this.from); x <= this.to; x++) f(x);
    },
    // 返回表示这个范围的字符串
    toString: function () {return "(" + this.from + "..." + this.to + ")";}
};

// 这里是使用"范围对象"的一些例子
var r = range(1, 3);      // 创建一个范围对象
r.includes(2);            // => true: 2 在这个范围内
r.foreach(console.log);   // 输出 1 2 3
console.log(r);           // 输出 (1...3)
```

将例9-1和例9-2中的代码做一个仔细的对比，可以发现两种定义类的技术的差别。首先，注意当工厂函数range()转化为构造函数时被重命名为Range()。这里遵循了一个常见的编程约定：从某种意义上讲，定义构造函数既是定义类，并且类名首字母要大写。而普通的函数和方法都是首字母小写。

再者，注意Range()构造函数是通过new关键字调用的（在示例代码的末尾），而range()工厂函数则不必使用new。例9-1通过调用普通函数（见8.2.1节）来创建新对象，例9-2则使用构造函数调用（见8.2.3节）来创建新对象。由于Range()构造函数是通过new关键字调用的，因此不必调用inherit()或其他什么逻辑来创建新对象。在调用构造函数之前就已经创建了新对象，通过this关键字可以获取这个新对象。Range()构造函数只不过是初始化this而已。构造函数甚至不必返回这个新创建的对象，构造函数会自动创建对象，然后将构造函数作为这个对象的方法来调用一次，最后返回这个新对象。事实上，构造函数的命名规则（首字母大写）和普通函数是如此不同还有另外一个原因，构造函数调用和普通函数调用是不尽相同的。构造函数就是用来“构造新对象”的，它必须通过关

键字new调用，如果将构造函数用做普通函数的话，往往不会正常工作。开发者可以通过命名约定来（构造函数首字母大写，普通方法首字母小写）判断是否应当在函数之前冠以关键字mew。

例9-1和例9-2之间还有一个非常重要的区别，就是原型对象的命名。在第一段示例代码中的原型是range.methods。这种命名方式很方便同时具有很好的语义，但又过于随意。在第二段示例代码中的原型是Range.prototype，这是一个强制的命名。对Range()构造函数的调用会自动使用Range.prototype作为新Range对象的原型。

最后，需要注意在例9-1和例9-2中两种类定义方式的相同之处，两者的范围方法定义和调用方式是完全一样的。

9.2.1 构造函数和类的标识

上文提到，原型对象是类的唯一标识：当且仅当两个对象继承自同一个原型对象时，它们才是属于同一个类的实例。而初始化对象的状态的构造函数则不能作为类的标识，两个构造函数的prototype属性可能指向同一个原型对象。那么这两个构造函数创建的实例是属于同一个类的。

尽管构造函数不像原型那样基础，但构造函数是类的“外在表现”。很明显的，构造函数的名字通常用做类名。比如，我们说Range()构造函数创建Range对象。然而，更根本地讲，当使用instanceof运算符来检测对象是否属于某个类时会用到构造函数。假设这里有一个对象r，我们想知道r是否是Range对象，我们这样写：

```
r instanceof Range // 如果r继承自Range.prototype，则返回true
```

实际上instanceof运算符并不会检查r是否是由Range()构造函数初始化而来，而会检查r是否继承自Range.prototype。不过，instanceof的语法则强化了“构造函数是类的公有标识”的概念。在本章的后面还会碰到对instanceof运算符的介绍。

9.2.2 constructor属性

在例9-2中，将Range.prototype定义为一个新对象，这个对象包含类所需要的方法。其实没有必要新创建一个对象，用单个对象直接量的属性就可以方便地定义原型上的方法。任何JavaScript函数都可以用做构造函数，并且调用构造函数是需要用到一个prototye属性的。因此，每个JavaScript函数（ECMAScript 5中的Function.bind()方法返回的函数除外）都自动拥有一个prototype属性。这个属性的值是一个对象，这个对象包含唯一一个不可枚举属性constructor。constructor属性的值是一个函数对象：

```
var F = function() {};          // 这是一个函数对象
```

```
var p = F.prototype;          // 这是F相关联的原型对象
var c = p.constructor;        // 这是与原型相关联的函数
c === F                       // => true: 对于任意函数F.prototype.constructor==F
```

可以看到构造函数的原型中存在预先定义好的constructor属性，这意味着对象通常继承的constructor均指代它们的构造函数。由于构造函数是类的“公共标识”，因此这个constructor属性为对象提供了类。

```
var o = new F();      //创建类F的一个对象
o.constructor === F   // => true, constructor属性指代这个类
```

如图9-1所示，图9-1展示了构造函数和原型对象之间的关系，包括原型到构造函数的反向引用以及构造函数创建的实例。

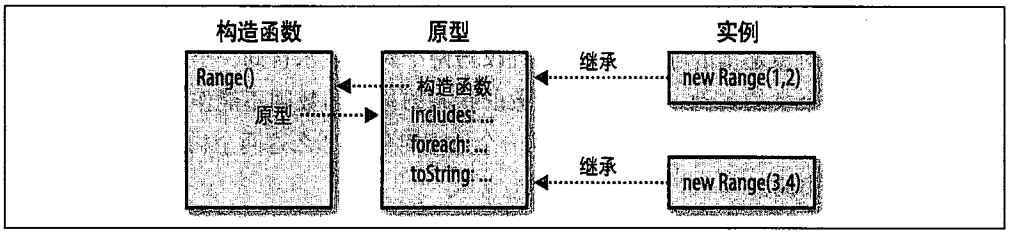

图9-1：构造函数及其原型和实例

需要注意的是，图9-1用Range()构造函数作为示例，但实际上，例9-2中定义的Range类使用它自身的一个新对象重写预定义的Range.prototype对象。这个新定义的原型对象不含有constructor属性。因此Range类的实例也不含有constructor属性。我们可以通过补救措施来修正这个问题，显式给原型添加一个构造函数：

```
Range.prototype = {
    constructor: Range, // 显式设置构造函数反向引用
    includes: function(x) { return this.from <= x && x <= this.to; },
    foreach: function(f) {
        for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
    },
    toString: function() { return "(" + this.from + "..." + this.to + ")"; }
};
```

另一种常见的解决办法是使用预定义的原型对象，预定义的原型对象包含constructor属性，然后依次给原型对象添加方法：

```
// 扩展预定义的Range.prototype对象，而不重写之
// 这样就自动创建Range.prototype.constructor属性
Range.prototype.includes = function (x) {return this.from <= x && x <= this.to;};
Range.prototype.foreach = function (f) {
    for (var x = Math.ceil(this.from); x <= this.to; x++) f(x);
```

```
};
Range.prototype.toString = function () {
    return "(" + this.from + "..." + this.to + ")";
};
```

9.3 JavaScript中Java式的类继承

如果你有过Java或其他类似强类型面向对象语言的开发经历的话，在你的脑海中，类成员的模样可能会是这个样子：

实例字段

它们是基于实例的属性或变量，用以保存独立对象的状态。

实例方法

它们是类的所有实例所共享的方法，由每个独立的实例调用。

类字段

这些属性或变量是属于类的，而不是属于类的某个实例的。

类方法

这些方法是属于类的，而不是属于类的某个实例的。

JavaScript和Java的一个不同之处在于，JavaScript中的函数都是以值的形式出现的，方法和字段之间并没有太大的区别。如果属性值是函数，那么这个属性就定义一个方法；否则，它只是一个普通的属性或“字段”。尽管存在诸多差异，我们还是可以用JavaScript模拟出Java中的这四种类成员类型。JavaScript中的类牵扯三种不同的对象（参照图9-1），三种对象的属性的行为和下面三种类成员非常相似：

构造函数对象

之前提到，构造函数（对象）为JavaScript的类定义了名字。任何添加到这个构造函数对象中的属性都是类字段和类方法（如果属性值是函数的话就是类方法）。

原型对象

原型对象的属性被类的所有实例所继承，如果原型对象的属性值是函数的话，这个函数就作为类的实例的方法来调用。

实例对象

类的每个实例都是一个独立的对象，直接给这个实例定义的属性是不会为所有实例对象所共享的。定义在实例上的非函数属性，实际上是实例的字段。

在JavaScript中定义类的步骤可以缩减为一个分三步的算法。第一步，先定义一个构造函数，并设置初始化新对象的实例属性。第二步，给构造函数的`prototype`对象定义实例

的方法。第三步，给构造函数定义类字段和类属性。我们可以将这三个步骤封装进一个简单的defineClass()函数中（这里用到了例6-2中的extend()函数和例8-3中的改进版）：

```
// 一个用以定义简单类的函数
function defineClass(constructor,      // 用以设置实例的属性的函数
                     methods,          // 实例的方法，复制至原型中
                     statics)          // 类属性，复制至构造函数中
{
    if (methods) extend(constructor.prototype, methods);
    if (statics) extend(constructor, statics);
    return constructor;
}

// 这是Range类的另一个实现
var SimpleRange =
    defineClass(function(f,t) { this.f = f; this.t = t; },
                {
                    includes: function(x) { return this.f <= x && x <= this.t;},
                    toString: function() { return this.f + "..." + this.t; }
                },
                { upto: function(t) { return new SimpleRange(0, t); } });
```

例9-3中定义类的代码更长一些。这里定义了一个表示复数的类，这段代码展示了如何使用JavaScript来模拟实现Java式的类成员。例9-3中的代码没有用到上面的defineClass()函数，而是“手动”来实现：

例 9-3：Complex.js：表示复数的类

```
/*
 * Complex.js:
 * 这个文件定义了Complex类，用来描述复数
 * 回忆一下，复数是实数和虚数的和，并且虚数i是－1的平方根
 */

/*
 * 这个构造函数为它所创建的每个实例定义了实例字段r和i
 * 这两个字段分别保存复数的实部和虚部
 * 它们是对象的状态
 */
function Complex(real, imaginary) {
    if (isNaN(real) || isNaN(imaginary))   // 确保两个实参都是数字
        throw new TypeError();             // 如果不都是数字则抛出错误
    this.r = real;                         // 复数的实部
    this.i = imaginary;                    // 复数的虚部
}

/*
 * 类的实例方法定义为原型对象的函数值属性
 * 这里定义的方法可以被所有实例继承，并为它们提供共享的行为
 * 需要注意的是，JavaScript的实例方法必须使用关键字this
 * 来存取实例的字段
 */
```

```
// 当前复数对象加上另外一个复数，并返回一个新的计算和值后的复数对象
Complex.prototype.add = function (that) {
    return new Complex(this.r + that.r, this.i + that.i);
};

// 当前复数乘以另外一个复数，并返回一个新的计算乘积之后的复数对象
Complex.prototype.mul = function (that) {
    return new Complex(this.r * that.r - this.i * that.i, this.r * that.i + this.i * that.r);
};

// 计算复数的模，复数的模定义为原点(0,0)到复平面的距离
Complex.prototype.mag = function () {
    return Math.sqrt(this.r * this.r + this.i * this.i);
};

// 复数的求负运算
Complex.prototype.neg = function () {
    return new Complex(-this.r, -this.i);
};

// 将复数对象转换为一个字符串
Complex.prototype.toString = function () {
    return "{" + this.r + "," + this.i + "}";
};

// 检测当前复数对象是否和另外一个复数值相等
Complex.prototype.equals = function (that) {
    return that != null &&                          // 必须有定义且不能是null
    that.constructor === Complex &&                 // 并且必须是Complex的实例
    this.r === that.r && this.i === that.i;         // 并且必须包含相同的值
};

/*
 * 类字段（比如常量）和类方法直接定义为构造函数的属性
 * 需要注意的是，类的方法通常不使用关键字this，
 * 它们只对其参数进行操作
 */

// 这里预定义了一些对复数运算有帮助的类字段
// 它们的命名全都是大写，用以表明它们是常量
// (在ECMAScript 5中，还能设置这些类字段的属性为只读)
Complex.ZERO = new Complex(0, 0);
Complex.ONE = new Complex(1, 0);
Complex.I = new Complex(0, 1);

// 这个类方法将由实例对象的toString方法返回的字符串格式解析为一个Complex对象
// 或者抛出一个类型错误异常
Complex.parse = function (s) {
    try {   // 假设解析成功
        var m = Complex._format.exec(s); // 利用正则表达式进行匹配
        return new Complex(parseFloat(m[1]), parseFloat(m[2]));
    } catch(x) { // 如果解析失败则抛出异常
        throw new TypeError("Can't parse '" + s + "' as a complex number.");
    }
};
```

```
// 定义类的"私有"字段，这个字段在Complex.parse()中用到了
// 下划线前缀表明它是类内部使用的，而不属于类的公有API的部分
Complex._format = /^\{([^,]+),([^}]+)\}$/;
```

从例9-3中所定义的Complex类可以看出，我们用到了构造函数、实例字段、实例方法、类字段和类方法，看一下这段示例代码：

```
var c = new Complex(2,3);      // 使用构造函数创建新的对象
var d = new Complex(c.i,c.r); // 用到了c的实例属性
c.add(d).toString();           // => "{5,5}": 使用了实例的方法
// 这个稍微复杂的表达式用到了类方法和类字段
Complex.parse(c.toString()).   // 将c转换为字符串
    add(c.neg()).              // 加上它的负数
    equals(Complex.ZERO)       // 结果应当永远是"零"
```

尽管JavaScript可以模拟出Java式的类成员，但Java中有很多重要的特性是无法在JavaScript类中模拟的。首先，对于Java类的实例方法来说，实例字段可以用做局部变量，而不需要使用关键字this来引用它们。JavaScript是没办法模拟这个特性的，但可以使用with语句来近似地实现这个功能（但这种做法并不推荐）：

```
Complex.prototype.toString = function() {
    with(this) {
        return "{" + r + "," + i + "}";
    }
};
```

在Java中可以使用final声明字段为常量，并且可以将字段和方法声明为private，用以表示它们是私有成员且在类的外面是不可见的。在JavaScript中没有这些关键字。例9-3中使用了一些命名写法上的约定来给出一些暗示，比如哪些成员是不能修改的（以大写字母命名的命名），哪些成员在类外部是不可见的（以下划线为前缀的命名）。关于这两个主题的讨论在本章后续还会碰到：私有属性可以使用闭包里的局部变量来模拟（参照9.6.6节），常量属性可以在ECMAScript 5中直接实现（参照9.8.2节）。

9.4 类的扩充

JavaScript中基于原型的继承机制是动态的：对象从其原型继承属性，如果创建对象之后原型的属性发生改变，也会影响到继承这个原型的所有实例对象。这意味着我们可以通过给原型对象添加新方法来扩充JavaScript类。这里我们给例9-3中的Complex类添加方法来计算复数的共轭复数[译注4]。

译注4：两个实部相等，虚部互为相反数的复数互为共轭复数。

```
// 返回当前复数的共轭复数
Complex.prototype.conj = function() { return new Complex(this.r, -this.i); };
```

JavaScript内置类的原型对象也是一样如此“开放”，也就是说可以给数字、字符串、数组、函数等数据类型添加方法。在例8-5中我们曾给ECMAScript 3中的函数类添加了bind()方法，这个方法原来是没有的：

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(o /*, args */) {
        // bind()方法的代码...
    };
}
```

这里有一些其他的例子：

```
// 多次调用这个函数f，传入一个迭代数
// 比如，要输出 "hello"三次:
// var n = 3;
// n.times(function(n) { console.log(n + " hello"); });
Number.prototype.times = function(f, context) {
    var n = Number(this);
    for(var i = 0; i < n; i++) f.call(context, i);
};

// 如果不存在ES5的String.trim()方法的话，就定义它
// 这个方法用以去除字符串开头和结尾的空格
String.prototype.trim = String.prototype.trim || function() {
    if (!this) return this;                 // 空字符串不做处理
    return this.replace(/^\s+|\s+$/g, "");  // 使用正则表达式进行空格替换
};

// 返回函数的名字，如果它有（非标准的）name属性，则直接使用name属性
// 否则，将函数转换为字符串然后从中提取名字
// 如果是没有名字的函数，则返回一个空字符串
Function.prototype.getName = function() {
    return this.name || this.toString().match(/function\s*([^()*]\(/)[1];
};
```

可以给Object.prototype添加方法，从而使所有的对象都可以调用这些方法。但这种做法并不推荐，因为在ECMAScript 5之前，无法将这些新增的方法设置为不可枚举的，如果给Object.prototype添加属性，这些属性是可以被for/in循环遍历到的。在9.8.1节中会给出ECMAScript 5中的一个例子，其中使用Object.defineProperty()方法可以安全地扩充Object.prototype。

然而并不是所有的宿主环境（比如Web浏览器）都可以使用Object.defineProperty()，这跟ECMAScript的具体实现有关。比如，在很多Web浏览器中，可以给HTMLElement.prototype添加方法，这样当前文档中表示HTML标记的所有对象就可以继承这些方法。但当前版本的IE则不支持这样做。这对客户端编程实用技术有着严重的限制。

9.5 类和类型

回想一下第3章的内容，JavaScript定义了少量的数据类型：null、undefined、布尔值、数字、字符串、函数和对象。typeof运算符（见4.13.2节）可以得出值的类型。然而，我们往往更希望将类作为类型来对待，这样就可以根据对象所属的类来区分它们。JavaScript语言核心中的内置对象（通常是指客户端JavaScript的宿主对象）可以根据它们的class属性（见6.8.2节）来区分彼此，比如在例6-4中用到了classof()函数。但当我们使用本章所提到的技术来定义类的话，实例对象的class属性都是“Object”，这时classof()函数也无用武之地。

接下来的几节介绍了三种用以检测任意对象的类的技术：instanceof运算符，constructor属性，以及构造函数的名字。但每种技术都不甚完美，本节总结讨论了鸭式辩型，这种编程哲学更加关注对象可以完成什么工作（它包含什么方法）而不是对象属于哪个类。

9.5.1 instanceof运算符

4.9.4节已经讨论过了instanceof运算符。左操作数是待检测其类的对象，右操作数是定义类的构造函数。如果o继承自c.prototype，则表达式o instanceof c值为true。这里的继承可以不是直接继承，如果o所继承的对象继承自另一个对象，后一个对象继承自c.prototype，这个表达式的运算结果也是true。

正如在本章前面所讲到的，构造函数是类的公共标识，但原型是唯一的标识。尽管instanceof运算符的右操作数是构造函数，但计算过程实际上是检测了对象的继承关系，而不是检测创建对象的构造函数。

如果你想检测对象的原型链上是否存在某个特定的原型对象，有没有不使用构造函数作为中介的方法呢？答案是肯定的，可以使用isPrototypeOf()方法。比如，可以通过如下代码来检测对象r是否是例9-1中定义的范围类的成员：

```
range.methods.isPrototypeOf(r);  //range.method 是原型对象
```

instanceof运算符和isPrototypeOf()方法的缺点是，我们无法通过对象来获得类名，只能检测对象是否属于指定的类名。在客户端JavaScript中还有一个比较严重的不足，就是在多窗口和多框架子页面的Web应用中兼容性不佳。每个窗口和框架子页面都具有单独的执行上下文，每个上下文都包含独有的全局变量和一组构造函数。在两个不同框架页面中创建的两个数组继承自两个相同但相互独立的原型对象，其中一个框架页面中的数组不是另一个框架页面的Array()构造函数的实例，instanceof运算结果是false。

9.5.2 constructor属性

另一种识别对象是否属于某个类的方法是使用constructor属性。因为构造函数是类的公共标识，所以最直接的方法就是使用constructor属性，比如：

```
function typeAndValue(x) {
    if (x == null) return ""; // Null 和 undefined 没有构造函数
    switch(x.constructor) {
    case Number: return "Number: " + x;         // 处理原始类型
    case String: return "String: '" + x + "'";
    case Date: return "Date: " + x;             // 处理内置类型
    case RegExp: return "Regexp: " + x;
    case Complex: return "Complex: " + x;       // 处理自定义类型
    }
}
```

需要注意的是，在代码中关键字case后的表达式都是函数，如果改用typeof运算符或获取到对象的class属性的话，它们应当改为字符串。

使用constructor属性检测对象属于某个类的技术的不足之处和instanceof一样。在多个执行上下文的场景中它是无法正常工作的（比如在浏览器窗口的多个框架子页面中）。在这种情况下，每个框架页面各自拥有独立的构造函数集合，一个框架页面中的Array构造函数和另一个框架页面的Array构造函数不是同一个构造函数。

同样，在JavaScript中也并非所有的对象都包含constructor属性。在每个新创建的函数原型上默认会有constructor属性，但我们常常会忽觉原型上的constructor属性。比如本章前面的示例代码中所定义的两个类（在例9-1和例9-2中），它们的实例都没有constructor属性。

9.5.3 构造函数的名称

使用instanceof运算符和constructor属性来检测对象所属的类有一个主要的问题，在多个执行上下文中存在构造函数的多个副本的时候，这两种方法的检测结果会出错。多个执行上下文中的函数看起来是一模一样的，但它们是相互独立的对象，因此彼此也不相等。

一种可能的解决方案是使用构造函数的名字而不是构造函数本身作为类标识符。一个窗口里的Array构造函数和另一个窗口的Array构造函数是不相等的，但是它们的名字是一样的。在一些JavaScript的实现中为函数对象提供了一个非标准的属性name，用来表示函数的名称。对于那些没有name属性的JavaScript实现来说，可以将函数转换为字符串，然后从中提取出函数名（在9.4节中的示例代码给Function类添加了getName()方法，就是使用这种方式来得到函数名）。

例9-4定义的type()函数以字符串的形式返回对象的类型。它用typeof运算符来处理原始值和函数。对于对象来说，它要么返回class属性的值要么返回构造函数的名字。type()函数用到了例6-4中的classof()函数和9.4节中的Function.getName()方法。为了简单起见，这里包含了函数和方法的代码。

例9-4：可以判断值的类型的type()函数

```
/**
 * 以字符串形式返回o的类型：
 *  -如果o是null，返回 "null"；如果o是 NaN，返回 "nan"
 *  -如果typeof所返回的值不是"object"，则返回这个值
 *  (注意，有一些JavaScript的实现将正则表达式识别为函数)
 *  -如果o的类不是"Object"，则返回这个值
 *  -如果o包含构造函数并且这个构造函数具有名称，则返回这个名称
 *  -否则，一律返回"Object"
 **/
function type(o) {
    var t, c, n; // type, class, name

    //处理null值的特殊情形
    if (o === null) return "null";

    // 另外一种特殊情形：NaN和它自身不相等
    if (o !== o) return "nan";

    // 如果typeof的值不是"object"，则使用这个值
    // 这可以识别出原始值的类型和函数
    if ((t = typeof o) !== "object") return t;

    // 返回对象的类名，除非值为"Object"
    // 这种方式可以识别出大多数的内置对象
    if ((c = classof(o)) !== "Object") return c;

    // 如果对象构造函数的名字存在的话，则返回它
    if (o.constructor && typeof o.constructor === "function" &&
        (n = o.constructor.getName())) return n;

    // 其他的类型都无法判别，一律返回"Object"
    return "Object";
}

// 返回对象的类
function classof(o) {
    return Object.prototype.toString.call(o).slice(8, -1);
};

// 返回函数的名字（可能是空字符串），不是函数的话返回null
Function.prototype.getName = function () {
    if ("name" in this) return this.name;
    return this.name = this.toString().match(/function\s*([^(]*)\(/)[1];
};
```

这种使用构造函数名字来识别对象的类的做法和使用constructor属性一样有一个问

题：并不是所有的对象都具有constructor属性。此外，并不是所有的函数都有名字。如果使用不带名字的函数定义表达式[译注5]定义一个构造函数，getName()方法则会返回空字符串：

```
// 这个构造函数没有名字
var Complex = function(x,y) { this.r = x; this.i = y; }
// 这个构造函数有名字
var Range = function Range(f,t) { this.from = f; this.to = t; }
```

9.5.4 鸭式辩型

上文所描述的检测对象的类的各种技术多少都会有些问题，至少在客户端JavaScript中是如此。解决办法就是规避掉这些问题：不要关注“对象的类是什么”，而是关注“对象能做什么”。这种思考问题的方式在Python和Ruby中非常普遍，称为“鸭式辩型”（这个表述是由作家James Whitcomb Riley 提出的）。

> 像鸭子一样走路、游泳并且嘎嘎叫的鸟就是鸭子。

对于JavaScript程序员来说，这句话可以理解为“如果一个对象可以像鸭子一样走路、游泳并且嘎嘎叫，就认为这个对象是鸭子，哪怕它并不是从鸭子类的原型对象继承而来的”。

我们拿例9-2中的Range类来举例好了。起初定义这个类用以描述数字的范围。但要注意，Range()构造函数并没有对实参进行类型检查以确保实参是数字类型。但却将参数使用“>”运算符进行比较运算，因为这里假定它们是可比较的。同样，includes()方法使用“<=”运算符进行比较，但没有对范围的结束点进行类似的假设。因为类并没有强制使用特定的类型，它的inlcudes()方法可以作用于任何结束点，只要结束点可以用关系运算符执行比较运算。

```
var lowercase = new Range("a", "z");
var thisYear = new Range(new Date(2009, 0, 1), new Date(2010, 0, 1));
```

Range类的foreach()方法中也没有显式地检测表示范围的结束点的类型，但Math.ceil()和“++”运算符表明它只能对数字结束点进行操作。

另外一个例子，回想一下在7.11节中所讨论的类数组对象。在很多场景下，我们并不知道一个对象是否真的是Array的实例，当然是可以通过判断是否包含非负的length属性来得知是否是Array的实例。我们说“包含一个值是非负整数的length”是数组的一个特征——“会走路”，任何具有“会走路”这个特征的对象都可以当做数组来对待（在很多情形中）。

译注5： 参照4.3节。

然而必须要了解的是，真正数组的length属性有一些独有的行为：当添加新的元素时，数组的长度会自动更新，并且当给length属性设置一个更小的整数时，数组会被自动截断。我们说这些特征是“会游泳”和“嘎嘎叫”。如果所实现的代码需要“会游泳”且能“嘎嘎叫”，则不能使用只“会走路”的类似数组的对象。

上文所讲到的鸭式辨型的例子提到了进行对象的“<”运算符的职责以及length属性的特殊行为。但当我们提到鸭式辨型时，往往是说检测对象是否实现了一个或多个方法。一个强类型的triathlon()函数所需要的参数必须是TriAthlete对象。而一种“鸭式辨型”式的做法是，只要对象包含walk()、swim()和bike()这三个方法就可以作为参数传入。同理，可以重新设计Range类，使用结束点对象的compareTo()和succ()（successor）方法来代替“<”和“++”运算符。

鸭式辨型的实现方法让人感觉太“放任自流”：仅仅是假设输入对象实现了必要的方法，根本没有执行进一步的检查。如果输入对象没有遵循“假设”，那么当代码试图调用那些不存在的方法时就会报错。另一种实现方法是对输入对象进行检查。但不是检查它们的类，而是用适当的名字来检查它们所实现的方法。这样可以将非法输入尽可能早地拦截在外，并可给出带有更多提示信息的报错。

例9-5中按照鸭式辨型的理念定义了quacks()函数（函数名叫“implements”会更加合适，但implements是保留字）。quacks()用以检查一个对象（第一个实参）是否实现了剩下的参数所表示的方法。对于除第一个参数外的每个参数，如果是字符串的话则直接检查是否存在以它命名的方法；如果是对象的话则检查第一个对象中的方法是否在这个对象中也具有同名的方法；如果参数是函数，则假定它是构造函数，函数将检查第一个对象实现的方法是否在构造函数的原型对象中也具有同名的方法。

例9-5：利用鸭式辨型实现的函数

```
// 如果o实现了除第一个参数之外的参数所表示的方法，则返回true
function quacks(o /*, ... */) {
    for (var i = 1; i < arguments.length; i++) { // 遍历o之后的所有参数
        var arg = arguments[i];
        switch (typeof arg) {     // 如果参数是：
        case 'string':            // string: 直接用名字做检查
          if (typeof o[arg] !== "function") return false;
          continue;
        case 'function':          // function: 检查函数的原型对象上的方法
          // 如果实参是函数，则使用它的原型
          arg = arg.prototype;    // 进入下一个case
        case 'object':            // object: 检查匹配的方法
          for (var m in arg) {    // 遍历对象的每个属性
              if (typeof arg[m] !== "function") continue; // 跳过不是方法的属性
              if (typeof o[m] !== "function") return false;
          }
        }
    }
```

```
    // 如果程序能执行到这里，说明o实现了所有的方法
    return true;
}
```

关于这个quacks()函数还有一些地方是需要尤为注意的。首先，这里只是通过特定的名称来检测对象是否含有一个或多个值为函数的属性。我们无法得知这些已经存在的属性的细节信息，比如，函数是干什么用的？它们需要多少参数？参数类型是什么？然而这是鸭式辩型的本质所在，如果使用鸭式辩型而不是强制的类型检测的方式定义API，那么创建的API应当更具灵活性才可以，这样才能确保你提供给用户的API更加安全可靠。关于quacks()函数还有另一问题需要注意，就是它不能应用于内置类。比如，不能通过quacks(o,Array)来检测o是否实现了Array中所有同名的方法。原因是内置类的方法都是不可枚举的，quacks()中的for/in循环无法遍历到它们（注意，在ECMAScript 5中有一个补救办法，就是使用Ojbect.getOwnPropertyNames()）。

9.6 JavaScript中的面向对象技术

到目前为止，我们讨论了JavaScript中类的基础知识：原型对象的重要性、它和构造函数之间的联系、instanceof运算符如何工作等。本节将目光转向一些实际的例子（尽管这不是基础知识），包括如何利用JavaScript中的类进行编程。我们从两个重要的例子开始，这两个例子中实现的类非常有意思，接下来的讨论都将基于此作展开。

9.6.1 一个例子：集合类

集合（set）是一种数据结构，用以表示非重复值的无序集合。集合的基础方法包括添加值、检测值是否在集合中，这种集合需要一种通用的实现，以保证操作效率。JavaScript的对象是属性名以及与之对应的值的基本集合。因此将对象只用做字符串的集合是大材小用。例子9-6用JavaScript实现了一个更加通用的Set类，它实现了从JavaScript值到唯一字符串的映射，然后将字符串用做属性名。对象和函数都不具备如此简明可靠的唯一字符串表示。因此集合类必须给集合中的每一个对象或函数定义一个唯一的属性标识。

例9-6：Set.js：值的任意集合

```
function Set() {          // 这是一个构造函数
    this.values = {};     // 集合数据保存在对象的属性里
    this.n = 0;           // 集合中值的个数
    this.add.apply(this, arguments); // 把所有参数都添加进这个集合
}

// 将每个参数都添加至集合中
Set.prototype.add = function () {
    for (var i = 0; i < arguments.length; i++) {    // 遍历每个参数
        var val = arguments[i];                     // 待添加到集合中的值
```

```
        var str = Set._v2s(val);                    // 把它转换为字符串
        if (!this.values.hasOwnProperty(str)) {     // 如果不在集合中
            this.values[str] = val;                 // 将字符串和值对应起来
            this.n++;                               // 集合中值的计数加一
        }
    }
    return this;                                    // 支持链式方法调用
};

// 从集合删除元素，这些元素由参数指定
Set.prototype.remove = function () {
    for (var i = 0; i < arguments.length; i++) {    // 遍历每个参数
        var str = Set._v2s(arguments[i]);           // 将字符串和值对应起来
        if (this.values.hasOwnProperty(str)) {      // 如果它在集合中
            delete this.values[str];                // 删除它
            this.n--;                               //集合中值的计数减一
        }
        }
        return this; // 支持链式方法调用
};

// 如果集合包含这个值，则返回true，否则，返回false
Set.prototype.contains = function (value) {
    return this.values.hasOwnProperty(Set._v2s(value));
};
// 返回集合的大小
Set.prototype.size = function () {
    return this.n;
};

// 遍历集合中的所有元素，在指定的上下文中调用f
Set.prototype.foreach = function (f, context) {
    for (var s in this.values)                      // 遍历集合中的所有字符串
        if (this.values.hasOwnProperty(s))          // 忽略继承的属性
        f.call(context, this.values[s]);            // 调用f，传入value
};

// 这是一个内部函数，用以将任意JavaScript值和唯一的字符串对应起来
Set._v2s = function (val) {
    switch (val) {
        case undefined:    return 'u';      //特殊的原始值
        case null:         return 'n';      //值只有一个字母
        case true:         return 't';      // 代码
        case false:        return 'f';
        default: switch (typeof val) {
            case 'number': return '#' + val;           // 数字都带有 # 前缀
            case 'string': return '"' + val;           // 字符串都带有" 前缀
                 default: return '@' + objectId(val);  // Objs and funcs get @
            }
        }

        // 对任意对象来说，都会返回一个字符串
        // 针对不同的对象，这个函数会返回不同的字符串
        // 对于同一个对象的多次调用，总是返回相同的字符串
        // 为了做到这一点，它给o创建了一个属性，在ES5中，这个属性是不可枚举且是只读的
```

```
        function objectId(o) {
            var prop = "|**objectid**|";   //私有属性，用以存放id
            if (!o.hasOwnProperty(prop))   //如果对象没有id
                o[prop] = Set._v2s.next++; //将下一个值赋给它
            return o[prop];                // 返回这个id
        }
};
Set._v2s.next = 100; // 设置初始id的值
```

9.6.2 一个例子：枚举类型

枚举类型（enumerated type）是一种类型，它是值的有限集合，如果值定义为这个类型则该值是可列出（或“可枚举”）的。在C及其派生语言中，枚举类型是通过关键字enum声明的。Enum是ECMAScript 5中的保留字（还未使用），很有可能在将来JavaScript就会内置支持枚举类型。到那时，例9-7展示了如何在JavaScript中定义枚举类型的数据。需要注意的是，这里用到了例6-1中的inherit()函数。

例9-7包含一个单独函数enumeration()。但它不是构造函数，它并没有定义一个名叫“enumeration”的类。相反，它是一个工厂方法，每次调用它都会创建并返回一个新的类，比如：

```
// 使用4个值创建新的Coin类： Coin.Penny, Coin.Nickel等
var Coin = enumeration({Penny: 1, Nickel:5, Dime:10, Quarter:25});
var c = Coin.Dime;                      // 这是新类的实例
c instanceof Coin                       // => true: instanceof正常工作
c.constructor == Coin                   // => true: 构造函数的属性正常工作
Coin.Quarter + 3*Coin.Nickel            // => 40: 将值转换为数字
Coin.Dime == 10                         // => true: 更多转换为数字的例子
Coin.Dime > Coin.Nickel                 // => true: 关系运算符正常工作
String(Coin.Dime) + ":" + Coin.Dime     // => "Dime:10": 强制转换为字符串
```

这个例子清楚地展示了JavaScript类的灵活性，JavaScript的类要比C++和Java语言中的静态类要更加灵活。

例 9-7：JavaScript中的枚举类型

```
// 这个函数创建一个新的枚举类型，实参对象表示类的每个实例的名字和值
// 返回值是一个构造函数，它标识这个新类
// 注意，这个构造函数也会抛出异常：不能使用它来创建该类型的新实例
// 返回的构造函数包含名/值对的映射表
// 包括由值组成的数组，以及一个foreach()迭代器函数
function enumeration(namesToValues) {
    // 这个虚拟的构造函数是返回值
    var enumeration = function () { throw "Can't Instantiate Enumerations"; };

    // 枚举值继承自这个对象
    var proto = enumeration.prototype = {
        constructor: enumeration,                    // 标识类型
```

```
        toString: function () { return this.name; },        // 返回名字
        valueOf: function () { return this.value; },        // 返回值
        toJSON: function () { return this.name; }           // 转换为JSON
    };

    enumeration.values = []; // 用以存放枚举对象的数组

    // 现在创建新类型的实例
    for (name in namesToValues) {           // 遍历每个值
        var e = inherit(proto);             // 创建一个代表它的对象
        e.name = name;                      // 给它一个名字
        e.value = namesToValues[name];      // 给它一个值
        enumeration[name] = e;              // 将它设置为构造函数的属性
        enumeration.values.push(e);         // 将它存储到值数组中
    }
    // 一个类方法，用来对类的实例进行迭代
    enumeration.foreach = function (f, c) {
        for (var i = 0; i < this.values.length; i++) f.call(c, this.values[i]);
    };

    // 返回标识这个新类型的构造函数
    return enumeration;
}
```

如果用这个枚举类型来实现一个“hello world”小程序的话，就可以使用枚举类型来表示一副扑克牌。例9-8中使用enumeration()函数实现了这个表示一副扑克牌的类[注1]。

例9-8：使用枚举类型来表示一副扑克牌

```
// 定义一个表示"玩牌"的类
function Card(suit, rank) {
    this.suit = suit;     // 每张牌都有花色
    this.rank = rank;     // 以及点数
}

// 使用枚举类型定义花色和点数
Card.Suit = enumeration({Clubs: 1, Diamonds: 2, Hearts: 3, Spades: 4});
Card.Rank = enumeration({Two: 2, Three: 3, Four: 4, Five: 5, Six: 6,
                         Seven: 7, Eight: 8, Nine: 9, Ten: 10,
                         Jack: 11, Queen: 12, King: 13, Ace: 14});

// 定义用以描述牌面的文本
Card.prototype.toString = function () {
    return this.rank.toString() + " of " + this.suit.toString();
};
// 比较扑克牌中两张牌的大小
Card.prototype.compareTo = function (that) {
    if (this.rank < that.rank) return -1;
    if (this.rank > that.rank) return 1;
    return 0;
};
```

注1：这个例子的作者是Joshua Bloch，最初是基于Java写的，可以在这里查看到：*http://jcp.org/aboutJava/communityprocess/jsr/tiger/enum.html*。

```
// 以扑克牌的玩法规则对牌进行排序的函数
Card.orderByRank = function (a, b) { return a.compareTo(b); };

// 以桥牌的玩法规则对扑牌进行排序的函数
Card.orderBySuit = function (a, b) {
    if (a.suit < b.suit) return -1;
    if (a.suit > b.suit) return 1;
    if (a.rank < b.rank) return -1;
    if (a.rank > b.rank) return 1;
    return 0;
};

// 定义用以表示一副标准扑克牌的类
function Deck() {
    var cards = this.cards = [];            // 一副牌就是由牌组成的数组
    Card.Suit.foreach(function (s) {        // 初始化这个数组
                          Card.Rank.foreach(function (r) {
                                                cards.push(new Card(s, r));
                                            });
                      });
}

// 洗牌的方法：重新洗牌并返回洗好的牌
Deck.prototype.shuffle = function () {
    // 遍历数组中的每个元素，随机找出牌面最小的元素，并与之（当前遍历的元素）交换
    var deck = this.cards, len = deck.length;
    for (var i = len - 1; i > 0; i--) {
        var r = Math.floor(Math.random() * (i + 1)), temp;          // 随机数
        temp = deck[i], deck[i] = deck[r], deck[r] = temp;          // 交换
    }
    return this;
};

// 发牌的方法：返回牌的数组
Deck.prototype.deal = function (n) {
    if (this.cards.length < n) throw "Out of cards";
    return this.cards.splice(this.cards.length - n, n);
};

// 创建一副新扑克牌，洗牌并发牌
var deck = (new Deck()).shuffle();
var hand = deck.deal(13).sort(Card.orderBySuit);
```

9.6.3 标准转换方法

3.8.3和6.10节讨论了对象类型转换所用到的重要方法，有一些方法是在需要做类型转换时由JavaScript解释器自动调用的。不需要为定义的每个类都实现这些方法，但这些方法的确非常重要，如果没有为自定义的类实现这些方法，也应当是有意为之，而不应当因为疏忽而漏掉了它们。

最重要的方法首当`toString()`。这个方法的作用是返回一个可以表示这个对象的字符

串。在希望使用字符串的地方用到对象的话（比如将对象用做属性名或使用“+”运算符来进行字符串连接运算），JavaScript会自动调用这个方法。如果没有实现这个方法，类会默认从Object.prototype中继承toString()方法，这个方法的运算结果是“[object Object]”，这个字符串用处不大。toString()方法应当返回一个可读的字符串，这样最终用户才能将这个输出值利用起来，然而有时候并不一定非要如此，不管怎样，可以返回可读字符串的toString()方法也会让程序调试变得更加轻松。例9-2和例9-3中的Range类和Complex类都定义了toString()方法，例9-7中的枚举类型也定义了toString()。下面我们会给例9-6中的Set类也定义toString()方法。

toLocaleString()和toString()极为类似：toLocaleString()是以本地敏感性（locale-sensitive）的方式来将对象转换为字符串。默认情况下，对象所继承的toLocaleString()方法只是简单地调用toString()方法。有一些内置类型包含有用的toLocaleString()方法用以实际上返回本地化相关的字符串。如果需要为对象到字符串的转换定义toString()方法，那么同样需要定义toLocaleString()方法用以处理本地化的对象到字符串的转换。下面的Set类的定义中会有相关代码。

第三个方法是valueOf()，它用来将对象转换为原始值。比如，当数学运算符（除了“+”运算符）和关系运算符作用于数字文本表示的对象时，会自动调用valueOf()方法。大多数对象都没有合适的原始值来表示它们，也没有定义这个方法。但在例9-7中的枚举类型的实现则说明valueOf()方法是非常重要的。

第四个方法是toJSON()，这个方法是由JSON.stringify()自动调用的。JSON格式用于序列化良好的数据结构，而且可以处理JavaScript原始值、数组和纯对象。它和类无关，当对一个对象执行序列化操作时，它会忽略对象的原型和构造函数。比如将Range对象或Complex对象作为参数传入JSON.stringify()，将会返回诸如{"form":1, "to":3}或{"r":1, "i":-1}这种字符串。如果将这些字符串传入JSON.parse()，则会得到一个和Range对象和Complex对象具有相同属性的纯对象，但这个对象不会包含从Range和Complex继承来的方法。

这种序列化操作非常适用于诸如Range和Complex这种类，但对于其他一些类则必须自定义toJSON()方法来定制个性化的序列化格式。如果一个对象有toJSON()方法，JSON.stringify()并不会对传入的对象做序列化操作，而会调用toJSON()来执行序列化操作（序列化的值可能是原始值也可能是对象）。比如，Date对象的toJSON()方法可以返回一个表示日期的字符串。例9-7中的枚举类型也是如此：它们的toJSON()方法和toString()方法完全一样。如果要模拟一个集合，最接近JSON的表示方法就是数组，因此在下面的例子中将定义toJSON()方法用以将集合对象转换为值数组。

例9-6中的Set类并没有定义上述方法中的任何一个。JavaScript中没有哪个原始值

可以表示集合，因此也没必要定义valueOf()方法，但该类应当包含toString()、toLocaleString()和toJSON()方法。可以用如下代码来实现。注意extend()函数（例6-2）的用法，这里使用extend()来向Set.prototype来添加方法：

```
// 将这些方法添加至Set类的原型对象中
extend(Set.prototype, {
          // 将集合转换为字符串
          toString: function () {
              var s = "{",
              i = 0;
              this.foreach(function (v) {s += ((i++>0) ? ", " : "") + v; });
              return s + "}";
          },
          // 类似 toString，但是对于所有的值都将调用toLocaleString()
          toLocaleString: function () {
              var s = "{",i = 0;
              this.foreach(function (v) {
                              if (i++>0) s += ", ";
                              if (v == null) s += v; // null 和 undefined
                              else s += v.toLocaleString(); // 其他情况
                          });
              return s + "}";
          },
          // 将集合转换为值数组
          toArray: function () {
              var a = [];
              this.foreach(function (v) { a.push(v); });
              return a;
          }
      });

// 对于要从JSON转换为字符串的集合都被当做数组来对待
Set.prototype.toJSON = Set.prototype.toArray;
```

9.6.4 比较方法

JavaScript的相等运算符比较对象时，比较的是引用而不是值。也就是说，给定两个对象引用，如果要看它们是否指向同一个对象，不是检查这两个对象是否具有相同的属性名和相同的属性值，而是直接比较这两个单独的对象是否相等，或者比较它们的顺序（就像“<”和“>”运算符进行的比较一样）。如果定义一个类，并且希望比较类的实例，应该定义合适的方法来执行比较操作。

Java编程语言有很多用于对象比较的方法，将Java中的这些方法借用到JavaScript中是一个不错的主意。为了能让自定义类的实例具备比较的功能，定义一个名叫equals()实例方法。这个方法只能接收一个实参，如果这个实参和调用此方法的对象相等的话则返回true。当然，这里所说的“相等”的含义是根据类的上下文来决定的。对于简单的类，可以通过简单地比较它们的constructor属性来确保两个对象是相同类型，然后比较两

个对象的实例属性以保证它们的值相等。例9-3中的Complex类就实现了这样的equals()方法，我们可以轻易地为Range类也实现类似的方法：

```
// Range类重写它的constructor属性，现在将它添加进去
Range.prototype.constructor = Range;

// 一个Range对象和其他不是Range的对象均不相等
// 当且仅当两个范围的端点相等，它们才相等
Range.prototype.equals = function(that) {
    if (that == null) return false;                    // 处理 null 和 undefined
    if (that.constructor !== Range) return false;      // 处理非Range对象
    // 当且仅当两个端点相等，才返回true
    return this.from == that.from && this.to == that.to;
}
```

给Set类定义equals()方法稍微有些复杂。不能简单地比较两个集合的values属性，还要进行更深层次的比较：

```
Set.prototype.equals = function (that) {
    // 一些次要情况的快捷处理
    if (this === that) return true;

    // 如果that对象不是一个集合，它和this不相等
    // 我们用到了instanceof，使得这个方法可以用于Set的任何子类
    // 如果希望采用鸭式辩型的方法，可以降低检查的严格程度
    // 或者可以通过 this.constructor == that.constructor 来加强检查的严格程度
    // 注意，null和undefined两个值是无法用于instanceof运算的
    if (! (that instanceof Set)) return false;

    // 如果两个集合的大小不一样，则它们不相等
    if (this.size() != that.size()) return false;

    // 现在检查两个集合中的元素是否完全一样
    // 如果两个集合不相等，则通过抛出异常来终止foreach循环
    try {
        this.foreach(function (v) { if (!that.contains(v)) throw false; });
        return true;                    // 所有的元素都匹配：两个集合相等
    } catch(x) {
        if (x === false) return false; // 如果集合中有元素在另外一个集合中不存在
        throw x;                        // 重新抛出异常
    }
};
```

按照我们需要的方式比较对象是否相等常常是很有用的。对于某些类来说，往往需要比较一个实例“大于”或者“小于”另外一个示例。比如，你可能会基于Range对象的下边界来定义实例的大小关系。枚举类型可以根据名字的字母表顺序来定义实例的大小，也可以根据它包含的数值（假设它包含的都是数字）来定义大小。另一方面，Set对象其实是无法排序的。

如果将对象用于JavaScript的关系比较运算符，比如“<”和“<=”，JavaScript会首先调

用对象的valueOf()方法，如果这个方法返回一个原始值，则直接比较原始值。例9-7中由enumeration()方法所返回的枚举类型包含valueOf()方法，因此可以使用关系运算符对它们做有意义的比较。但大多数类并没有valueOf()方法，为了按照显式定义的规则来比较这些类型的对象，可以定义一个名叫compareTo()的方法（同样，这里遵照Java中的命名约定）。

compareTo()方法应当只能接收一个参数，这个方法将这个参数和调用它的对象进行比较。如果this对象小于参数对象，compareTo()应当返回比0小的值。如果this对象大于参数对象，应当返回比0大的值。如果两个对象相等，应当返回0。这些关于返回值的约定非常重要，这样我们可以用下面的表达式替换掉关系比较和相等性运算符：

待替换	替换为
a<b	a.compareTo(b)<0
a<=b	a.compareTo(b)<=0
a>b	a.compareTo(b)>0
a>=b	a.compareTo(b)>=0
a==b	a.compareTo(b)==0
a!=b	a.compareTo(b)!=0

例9-8中的Card类定义了该类的compareTo()方法，可以给Range类添加一个类似的方法，用以比较它们的下边界：

```
Range.prototype.compareTo = function(that) {
    return this.from - that.from;
};
```

需要注意的是，这个方法中的减法操作根据两个Range对象的关系正确地返回了小于0、等于0和大于0的值。例9-8中的Card.Rank枚举值包含valueOf()方法，其实也可以给Card类实现类似的compareTo()方法。

上文所提到的equals()方法对其参数执行了类型检查，如果参数类型不合法则返回false。compareTo()方法并没有返回一个表示“这两个值不能比较”的值，由于compareTo()没有对参数做任何类型检查，因此如果给compareTo()方法传入错误类型的参数，往往会抛出异常。

注意，如果两个范围对象的下边界相等，为Range类定义的compareTo()方法会返回0。这意味着就compareTo()而言，任何两个起始点相同[译注6]的Range对象都相等。这个相等概念的定义和equals()方法定义的相等概念是相背的，equals()要求两个端点均相

译注6：这里所说的起始点相同就是下边界相同。

等才算相等。这种相等概念上的差异性会造成很多bug，最好将Range类的equals()和compareTo()方法中处理相等的逻辑保持一致。这里是Range类修正后的compareTo()方法，它的比较逻辑和equals()保持一致，但当传入不可比较的值时仍然会报错：

```
// 根据下边界来对Range对象排序，如果下边界相等则比较上边界
// 如果传入非Range值，则抛出异常
// 当且仅当this.equals(that)时，才返回0
Range.prototype.compareTo = function(that) {
    if (!(that instanceof Range))
        throw new Error("Can't compare a Range with " + that);
    var diff = this.from - that.from;          // 比较下边界
    if (diff == 0) diff = this.to - that.to;   // 如果相等，比较上边界
    return diff;
};
```

给类定义了compareTo()方法，这样就可以对类的实例组成的数组进行排序了。Array.sort()方法可以接收一个可选的参数，这个参数是一个函数，用来比较两个值的大小，这个函数返回值的约定和compareTo()方法保持一致。假定有了上文提到的compareTo()方法，就可以很方便地对Range对象组成的数组进行排序了：

```
ranges.sort(function(a,b) { return a.compareTo(b); });
```

排序运算非常重要，如果已经为类定义了实例方法compareTo()，还应当参照这个方法定义一个可传入两个参数的比较函数。使用compareTo()方法可以非常轻松地定义这个函数，比如：

```
Range.byLowerBound = function(a,b) { return a.compareTo(b); };
```

使用这个方法可以让数组排序的操作变得非常简单：

```
ranges.sort(Range.byLowerBound);
```

有些类可以有很多方法进行排序。比如Card类，可以定义两个方法分别按照花色排序和按照点数排序。

9.6.5 方法借用

JavaScript中的方法没有什么特别：无非是一些简单的函数，赋值给了对象的属性，可以通过对象来调用它。一个函数可以赋值给两个属性，然后作为两个方法来调用它。比如，我们在Set类中就这样做了，将toArray()方法创建了一个副本，并让它可以和toJSON()方法一样完成同样的功能。

多个类中的方法可以共用一个单独的函数。比如， Array类通常定义了一些内置方法，如果定义了一个类，它的实例是类数组的对象，则可以从Array.prototype中将函数复

制至所定义的类的原型对象中。如果以经典的面向对象语言的视角来看JavaScript的话，把一个类的方法用到其他的类中的做法也称做“多重继承”（multiple inheritance）。然而，JavaScript并不是经典的面向对象语言，我更倾向于将这种方法重用更正式地称为“方法借用”（borrowing）。

不仅Array的方法可以借用，还可以自定义泛型方法（generic method）。例9-9定义了泛型方法toString()和equals()，可以被Range、Complex和Card这些简单的类使用。如果Range类没有定义equals()方法，可以这样借用泛型方法equals()：

```
Range.prototype.equals = generic.equals;
```

注意，generic.equals()只会执行浅比较，因此这个方法并不适用于其实例太复杂的类，它们的实例属性通过其equals()方法指代对象。同样需要注意，这个方法包含一些特殊情况的程序逻辑，以处理新增至Set对象中的属性（见例9-6）。

例 9-9：方法借用的泛型实现

```
var generic = {
    // 返回一个字符串，这个字符串包含构造函数的名字（如果构造函数包含名字）
    // 以及所有非继承来的、非函数属性的名字和值
    toString: function () {
        var s = '[';
        // 如果这个对象包含构造函数，且构造函数包含名字
        // 这个名字会作为返回字符串的一部分
        // 需要注意的是，函数的名字属性是非标准的，并不是在所有的环境中都可用
        if (this.constructor && this.constructor.name)
            s += this.constructor.name + ": ";

        // 枚举所有非继承且非函数的属性
        var n = 0;
        for (var name in this) {
            if (!this.hasOwnProperty(name)) continue; //跳过继承来的属性
            var value = this[name];
            if (typeof value === "function") continue; // 跳过方法
            if (n++) s += ", ";
            s += name + '=' + value;
        }
        return s + ']';
    },

    // 通过比较this和that的构造函数和实例属性来判断它们是否相等
    // 这种方法只适合于那些实例属性是原始值的情况，原始值可以通过"==="来比较
    // 这里还处理一种特殊情况，就是忽略由Set类添加的特殊属性
    equals: function (that) {
        if (that == null) return false;
        if (this.constructor !== that.constructor) return false;
        for (var name in this) {
            if (name === "|**objectid**|") continue;          // 跳过特殊属性
            if (!this.hasOwnProperty(name)) continue;         // 跳过继承来的属性
            if (this[name] !== that[name]) return false;      // 比较是否相等
```

```
        }
        return true; // 如果所有属性都匹配 ，两个对象相等
    }
};
```

9.6.6 私有状态

在经典的面向对象编程中，经常需要将对象的某个状态封装或隐藏在对象内，只有通过对象的方法才能访问这些状态，对外只暴露一些重要的状态变量可以直接读写。为了实现这个目的，类似Java的编程语言允许声明类的“私有”实例字段，这些私有实例字段只能被类的实例方法访问，且在类的外部是不可见的。

我们可以通过将变量（或参数）闭包在一个构造函数内来模拟实现私有实例字段，调用构造函数会创建一个实例。为了做到这一点，需要在构造函数内部定义一个函数（因此这个函数可以访问构造函数内部的参数和变量），并将这个函数赋值给新创建对象的属性。例9-10展示了对Range类的另一种封装，新版的类的实例包含from()和to()方法用以返回范围的端点，而不是用from和to属性来获取端点。这里的from()和to()方法是定义在每个Range对象上的，而不是从原型中继承来的。其他的Range方法还是和之前一样定义在原型中，但获取端点的方式从之前直接从属性读取变成了通过from()和to()方法来读取。

例9-10：对Range类的读取端点方法的简单封装

```
function Range(from, to) {
    // 不要将端点保存为对象的属性，相反
    // 定义存取器函数来返回端点的值
    // 这些值都保存在闭包中
    this.from = function () { return from; };
        this.to = function () { return to; };
}

// 原型上的方法无法直接操作端点
// 它们必须调用存取器方法
Range.prototype = {
    constructor: Range,
    includes: function (x) { return this.from() <= x && x <= this.to(); },
    foreach: function (f) {
        for (var x = Math.ceil(this.from()), max = this.to(); x <= max; x++) f(x);
    },
    toString: function () { return "(" + this.from() + "..." + this.to() + ")"; }
};
```

这个新的Range类定义了用以读取范围端点的方法，但没有定义设置端点的方法或属性。这让类的实例看起来是不可修改的，如果使用正确的话，一旦创建Range对象，端点数据就不可修改了。除非使用ECMAScript 5（参照9.3节）中的某些特性，但from和to属性依然是可写的，并且Range对象实际上并不是真正不可修改的：

```
var r = new Range(1,5);                 //一个不可修改的范围
r.from = function() { return 0; };      //通过方法替换来修改它
```

但需要注意的是，这种封装技术造成了更多系统开销。使用闭包来封装类的状态的类一定会比不使用封装的状态变量的等价类运行速度更慢，并占用更多内存。

9.6.7 构造函数的重载和工厂方法

有时候，我们希望对象的初始化有多种方式。比如，我们想通过半径和角度（极坐标）来初始化一个Complex对象，而不是通过实部和虚部来初始化，或者通过元素组成的数组来初始化一个Set对象，而不是通过传入构造函数的参数来初始化它。

有一个方法可以实现，通过重载（overload）这个构造函数让它根据传入参数的不同来执行不同的初始化方法。下面这段代码就是重载Set()构造函数的例子：

```
function Set() {
    this.values = {};          // 用这个对象的属性来保存这个集合
    this.n = 0;                // 集合中值的个数

    // 如果传入一个类数组的对象，将这个元素添加至集合中
    // 否则，将所有的参数都添加至集合中
    if (arguments.length == 1 && isArrayLike(arguments[0]))
        this.add.apply(this, arguments[0]);
    else if (arguments.length > 0)
        this.add.apply(this, arguments);
}
```

这段代码所定义的Set()构造函数可以显式将一组元素作为参数列表传入，也可以传入元素组成的数组。但是这个构造函数有多义性，如果集合的某个成员是一个数组就无法通过这个构造函数来创建这个集合了（为了做到这一点，需要首先创建一个空集合，然后显式调用add()方法）。

在使用极坐标来初始化复数的例子中，实际上并没有看到有函数重载。代表复数两个维度的数字都是浮点数，除非给构造函数传入第三个参数，否则构造函数无法识别到底传入的是极坐标参数还是直角坐标参数。相反，可以写一个工厂方法——一个类的方法用以返回类的一个实例。下面的例子即是使用工厂方法来返回一个使用极坐标初始化的Complex对象：

```
Complex.polar = function(r, theta) {
    return new Complex(r*Math.cos(theta), r*Math.sin(theta));
};
```

下面这个工厂方法用来通过数组初始化Set对象：

```
Set.fromArray = function(a) {
    s = new Set();              //创建一个空集合
    s.add.apply(s, a);          //将数组a的成员作为参数传入add()方法
    return s;                   // 返回这个新集合
};
```

可以给工厂方法定义任意的名字，不同名字的工厂方法用以执行不同的初始化。但由于构造函数是类的公有标识，因此每个类只能有一个构造函数。但这并不是一个“必须遵守”的规则。在JavaScript中是可以定义多个构造函数继承自一个原型对象的，如果这样做的话，由这些构造函数的任意一个所创建的对象都属于同一类型。并不推荐这种技术，但下面的示例代码使用这种技术定义了该类型的一个辅助构造函数：

```
// Set类的一个辅助构造函数
function SetFromArray(a) {
    // 通过以函数的形式调用Set()来初始化这个新对象
    // 将a的元素作为参数传入[译注7]
    Set.apply(this, a);
}
// 设置原型，以便SetFromArray能创建Set的实例
SetFromArray.prototype = Set.prototype;

var s = new SetFromArray([1,2,3]);
s instanceof Set // => true
```

9.7 子类

在面向对象编程中，类B可以继承自另外一个类A。我们将A称为父类（superclass），将B称为子类（subclass）。B的实例从A继承了所有的实例方法。类B可以定义自己的实例方法，有些方法可以重载类A中的同名方法，如果B的方法重载了A中的方法，B中的重载方法可能会调用A中的重载方法，这种做法称为“方法链”（method chaining）。同样，子类的构造函数B()有时需要调用父类的构造函数A()，这种做法称为“构造函数链”（constructor chaining）。子类还可以有子类，当涉及类的层次结构时，往往需要定义抽象类（abstract class）。抽象类中定义的方法没有实现。抽象类中的抽象方法是在抽象类的具体子类中实现的。

在JavaScript中创建子类的关键之处在于，采用合适的方法对原型对象进行初始化。如果类B继承自类A，B.prototype必须是A.prototype的后嗣。B的实例继承自B.prototype，后者同样也继承自A.prototype。本节将会对刚才提到的子类相关的术语做一一讲解，还会介绍类继承的替代方案：“组合”（composition）。

我们从例9-6中的Set类开始讲解，本节将会讨论如何定义子类，如何实现构造函数链并

译注7：apply()的第二个参数是一个数组，数组成员就是参数列表。

重载方法，如何使用组合来代替继承，以及最后如何通过抽象类从实现中提炼出接口。本节以一个扩展的例子结束，这个例子定义了Set类的层次结构。注意，本节开始的几个例子着重讲述了实现子类的基础技术。其中某些技术有着重要的缺陷，后续几节会讲到。

9.7.1 定义子类

JavaScript的对象可以从类的原型对象中继承属性（通常继承的是方法）。如果O是类B的实例，B是A的子类，那么O也一定从A中继承了属性。为此，首先要确保B的原型对象继承自A的原型对象。通过inherit()函数（例6-1），可以这样来实现：

```
B.prototype = inherit(A.prototype);    // 子类派生自父类
B.prototype.constructor = B;           //重载继承来的constructor属性
```

这两行代码是在JavaScript中创建子类的关键。如果不这样做，原型对象仅仅是一个普通对象，它只继承自Object.prototype，这意味着你的类和所有的类一样是Object的子类。如果将这两行代码添加至defineClass()函数中（参照9.3节），可以将它变成例9-11中的defineSubclass()函数和Function.prototype.extend()方法：

例9-11：定义子类

```
// 用一个简单的函数创建简单的子类
function defineSubclass(superclass,        // 父类的构造函数
                        constructor,       // 新的子类的构造函数
                        methods,           // 实例方法：复制至原型中
                        statics)           // 类属性：复制至构造函数中
{
    // 建立子类的原型对象
    constructor.prototype = inherit(superclass.prototype);
    constructor.prototype.constructor = constructor;
    // 像对常规类一样复制方法和类属性
    if (methods) extend(constructor.prototype, methods);
    if (statics) extend(constructor, statics);
    // 返回这个类
    return constructor;
}

// 也可以通过父类构造函数的方法来做到这一点
Function.prototype.extend = function (constructor, methods, statics) {
    return defineSubclass(this, constructor, methods, statics);
};
```

例9-12展示了不使用defineSubclass()函数如何“手动”实现子类。这里定义了Set的子类SingletonSet。SingletonSet是一个特殊的集合，它是只读的，而且含有单独的常量成员。

例9-12：SingletonSet: 一个简单的子类

```
// 构造函数
function SingletonSet(member) {
    this.member = member;          //记住集合中这个唯一的成员
```

```
}

// 创建一个原型对象，这个原型对象继承自Set的原型
SingletonSet.prototype = inherit(Set.prototype);

// 给原型添加属性
// 如果有同名的属性就覆盖Set.prototype中的同名属性
extend(SingletonSet.prototype, {
            // 设置合适的constructor属性
            constructor: SingletonSet,
            // 这个集合是只读的：调用add()和remove()都会报错
            add: function () { throw "read-only set"; },
            remove: function () { throw "read-only set"; },
            // SingletonSet的实例中永远只有一个元素
            size: function () { return 1; },
            // 这个方法只调用一次，传入这个集合的唯一成员
            foreach: function (f, context) { f.call(context, this.member); },
            // contains()方法非常简单：只须检查传入的值是否匹配这个集合唯一的成员即可
            contains: function (x) { return x === this.member; }
        });
```

这里的SingletonSet类是一个比较简单的实现，它包含5个简单的方法定义。它实现了5个核心的Set方法，但从它的父类中继承了toString()、toArray()和equals()方法。定义子类就是为了继承这些方法。比如，Set类的equals()方法（在9.4节中定义）用来对Set实例进行比较，只要Set的实例包含size()和foreach()方法，就可以通过equals()比较。因为SingletonSet是Set的子类，所以它自动继承了equals()的实现，不用再实现一次。当然，如果想要最简单的实现方式，那么给SingletonSet类定义它自己的equals()版本会更高效一些：

```
SingletonSet.prototype.equals = function(that) {
    return that instanceof Set && that.size()==1 && that.contains(this.member);
};
```

需要注意的是，SingletonSet不是将Set中的方法列表静态地借用过来，而是动态地从Set类继承方法。如果给Set.prototype添加新的方法，Set和SingletonSet的所有实例就会立即拥有这个方法（假定SingletonSet没有定义与之同名的方法）。

9.7.2 构造函数和方法链

最后一节的SingletonSet类定义了全新的集合实现，而且将它继承自其父类的核心方法全部替换。然而定义子类时，我们往往希望对父类的行为进行修改或扩充，而不是完全替换掉它们。为了做到这一点，构造函数和子类的方法需要调用或链接到父类构造函数和父类方法。

例9-13 对此做了展示。它定义了Set的子类NonNullSet，它不允许null和undefined作为

它的成员。为了使用这种方式对成员做限制，NonNullSet需要在其add()方法中对null和undefined值做检测。但它需要完全重新实现一个add()方法，因此它调用了父类中的这个方法。注意，NonNullSet()构造函数同样不需要重新实现，它只须将它的参数传入父类构造函数（作为函数来调用它，而不是通过构造函数来调用），通过父类的构造函数来初始化新创建的对象。

例9-13：在子类中调用父类的构造函数和方法

```
/*
 * NonNullSet 是Set的子类，它的成员不能是null 和undefined
 */
function NonNullSet() {
    //仅链接到父类
    //作为普通函数调用父类的构造函数来初始化通过该构造函数调用创建的对象
    Set.apply(this, arguments);
}

// 将NonNullSet设置为Set的子类
NonNullSet.prototype = inherit(Set.prototype);
NonNullSet.prototype.constructor = NonNullSet;

// 为了将null和undefined排除在外，只须重写add()方法
NonNullSet.prototype.add = function() {
    //检查参数是不是null或undefined
    for (var i = 0; i < arguments.length; i++)
        if (arguments[i] == null)
            throw new Error("Can't add null or undefined to a NonNullSet");

    //调用父类的add()方法以执行实际插入操作
    return Set.prototype.add.apply(this, arguments);
};
```

让我们将这个非null集合的概念推而广之，称为“过滤后的集合”，这个集合中的成员必须首先传入一个过滤函数再执行添加操作。为此，定义一个类工厂函数（类似例9-7中的enumeration()函数），传入一个过滤函数，返回一个新的Set子类。实际上，可以对此做进一步的通用化的处理，定义一个可以接收两个参数的类工厂：子类和用于add()方法的过滤函数。这个工厂方法称为filteredsetSubclass()，并通过这样的代码来使用它：

```
// 定义一个只能保存字符串的"集合"类
var StringSet = filteredSetSubclass(Set,
                                    function(x) {return typeof x==="string";});

// 这个集合类的成员不能是null、undefined或函数
var MySet = filteredSetSubclass(NonNullSet,
                                function(x) {return typeof x !== "function";});
```

例9-14是这个类工厂函数的实现代码。注意，这个例子中的方法链和构造函数链和NonNullset中的实现是一样的。

例 9-14：类工厂和方法链

```
/*
 * 这个函数返回具体Set类的子类
 * 并重写该类的add()方法用以对添加的元素做特殊的过滤
 */
function filteredSetSubclass(superclass, filter) {
    var constructor = function() {          // 子类构造函数
        superclass.apply(this, arguments); // 调用父类构造函数
    };
    var proto = constructor.prototype = inherit(superclass.prototype);
    proto.constructor = constructor;
    proto.add = function() {
        //在添加任何成员之前首先使用过滤器将所有参数进行过滤
        for (var i = 0; i < arguments.length; i++) {
            var v = arguments[i];
            if (!filter(v)) throw ("value " + v + " rejected by filter");
        }
        //调用父类的add()方法
        superclass.prototype.add.apply(this, arguments);
    };
    return constructor;
}
```

例9-14中一个比较有趣的事情是，用一个函数将创建子类的代码包装起来，这样就可以在构造函数和方法链中使用父类的参数，而不是通过写死某个父类的名字来使用它的参数。也就是说如果想修改父类，只须修改一处代码即可，而不必对每个用到父类类名的地方都做修改。已经有充足的理由证明这种技术的可行性，即使在不是定义类工厂的场景中，这种技术也是值得提倡使用的。比如，可以这样使用包装函数和例9-11的Function.prototype.extend()方法来重写NonNullSet：

```
    var NonNullSet = (function() {          // 定义并立即调用这个函数
        var superclass = Set;               // 仅指定父类
        return superclass.extend(
            function() { superclass.apply(this, arguments); },      // 构造函数
            {                                                       // 方法
                add: function() {
                    // 检查参数是否是null或undefined
                    for (var i = 0; i < arguments.length; i++)
                        if (arguments[i] == null)
                            throw new Error("Can't add null or undefined");

                    //调用父类的add()方法以执行实际插入操作
                    return superclass.prototype.add.apply(this, arguments);

                }
            });
    } ());
```

最后，值得强调的是，类似这种创建类工厂的能力是JavaScript语言动态特性的一个体现，类工厂是一种非常强大和有用的特性，这在Java和C++等语言中是没有的。

9.7.3 组合 vs 子类

在前一节中，定义的集合可以根据特定的标准对集合成员做限制，而且使用了子类的技术来实现这种功能，所创建的自定义子类使用了特定的过滤函数来对集合中的成员做限制。父类和过滤函数的每个组合都需要创建一个新的类。

然而还有另一种更好的方法来完成这种需求，即面向对象编程中一条广为人知的设计原则："组合优于继承"[注2]。这样，可以利用组合的原理定义一个新的集合实现，它"包装"了另外一个集合对象，在将受限制的成员过滤掉之后会用到这个（包装的）集合对象。例9-15展示了其工作原理：

例9-15：使用组合代替继承的集合的实现

```
/*
 * 实现一个FilteredSet，它包装某个指定的"集合"对象，
 * 并对传入add()方法的值应用了某种指定的过滤器
 * "范围"类中其他所有的核心方法延续到包装后的实例中
 */
var FilteredSet = Set.extend(
    function FilteredSet(set, filter) { // 构造函数
        this.set = set;
        this.filter = filter;
    },
    { // 实例方法
     add: function() {
        // 如果已有过滤器，直接使用它
        if (this.filter) {
            for (var i = 0; i < arguments.length; i++) {
                 var v = arguments[i];
                 if (!this.filter(v))
                     throw new Error("FilteredSet: value " + v +
                                     " rejected by filter");
                 }
            }

            // 调用set中的add()方法
            this.set.add.apply(this.set, arguments);
            return this;
        },
        // 剩下的方法都保持不变
        remove: function() {
            this.set.remove.apply(this.set, arguments);
            return this;
        },
        contains: function(v) { return this.set.contains(v); },
        size: function() { return this.set.size(); },
        foreach: function(f, c) { this.set.foreach(f, c); }
});
```

注2：　可参照Erich Gamma et al所著《Design Patterns》和Joshua Bloch所著《Effective Java》。

在这个例子中使用组合的一个好处是，只须创建一个单独的FilteredSet子类即可。可以利用这个类的实例来创建任意带有成员限制的集合实例。比如，不用上文中定义的NonNullSet类，可以这样做：

```
var s = new FilteredSet(new Set(), function(x) { return x !== null; });
```

甚至还可以对已经过滤后的集合进行过滤：

```
var t = new FilteredSet(s, { function(x) { return !(x instanceof Set); }};
```

9.7.4 类的层次结构和抽象类

在上一节中给出了“组合优于继承”的原则，但为了将这条原则阐述清楚，创建了Set的子类。这样做的原因是最终得到的类是Set的实例[译注8]，它会从Set继承有用的辅助方法，比如toString()和equals()。尽管这是一个很实际的原因，但不用创建类似Set类这种具体类的子类也可以很好的用组合来实现“范围”。例9-12中的SingletonSet类可以有另外一种类似的实现，这个类还是继承自Set，因此它可以继承很多辅助方法，但它的实现和其父类的实现完全不一样。SingletonSet并不是Set类的专用版本，而是完全不同的另一种Set。在类层次结构中SingletonSet和Set应当是兄弟的关系，而非父子关系。

不管是在经典的面向对象编程语言中还是在JavaScript中，通行的解决办法是[译注9]“从实现中抽离出接口”。假定定义了一个AbstractSet类，其中定义了一些辅助方法比如toString()，但并没有实现诸如foreach()的核心方法。这样，实现的Set、SingletonSet和FilteredSet都是这个抽象类的子类，FilteredSet和SingletonSet都不必再实现为某个不相关的类的子类了。

例9-16在这个思路上更进一步，定义了一个层次结构的抽象的集合类。AbstractSet只定义了一个抽象方法：contains()。任何类只要“声称”自己是一个表示范围的类，就必须至少定义这个contains()方法。然后，定义AbstractSet的子类AbstractEnumerableSet。这个类增加了抽象的size()和foreach()方法，而且定义了一些有用的非抽象方法（toString()、toArray()、equals()等），AbstractEnumerableSet并没有定义add()和remove()方法，它只代表只读集合。SingletonSet可以实现为非抽象子类。最后，定义了AbstractEnumerableSet的子类AbstractWritableSet。这个final抽象集合定义了抽象方法add()和remove()，并实现了诸如union()和intersection()等非具体方法，这两个方法

译注8：作者这里的表述稍有含混，作者的意思应该是“Set子类的实例也是Set的实例”，而不是“子类是Set的实例”。

译注9：这里指的是实现类的不同定制版本的解决办法，更直接地讲就是实现多态的方法。

调用了add()和remove()。AbstractWritableSet是Set和FilteredSet类相应的父类。但这个例子中并没有实现它，而是实现了一个新的名叫ArraySet的非抽象类。

例9-16中的代码很长，但还是应当完整地阅读一遍。注意这里用到了Function.prototype.extend()作为创建子类的快捷方式。

例9-16：抽象类和非抽象Set类的层次结构

```
// 这个函数可以用做任何抽象方法，非常方便
function abstractmethod() { throw new Error("abstract method"); }

/*
 * AbstractSet类定义了一个抽象方法：contains()
 */
function AbstractSet() { throw new Error("Can't instantiate abstract classes"); }
AbstractSet.prototype.contains = abstractmethod;

/*
 * NotSet是AbstractSet的一个非抽象子类
 * 所有不在其他集合中的成员都在这个集合中
 * 因为它是在其他集合是不可写 的条件下定义的
 * 同时由于它的成员是无限个，因此它是不可枚举的
 * 我们只能用它来检测元素成员的归属情况
 * 注意，我们使用了Function.prototype.extend()方法来定义这个子类
 */
var NotSet = AbstractSet.extend(
    function NotSet(set) { this.set = set; },
    {
        contains: function (x) { return !this.set.contains(x); },
        toString: function (x) { return "~" + this.set.toString(); },
        equals: function (that) {
            return that instanceof NotSet && this.set.equals(that.set);
        }
    }
);

/*
 * AbstractEnumerableSet 是AbstractSet的一个抽象子类
 * 它定义了抽象方法size()和foreach()
 * 然后实现了非抽象方法isEmpty()、toArray()、to[Locale]String()和equals()方法
 * 子类实现了contains()、size()和foreach()，这三个方法可以很轻易地调用这5个非抽象方法
 */
var AbstractEnumerableSet = AbstractSet.extend(
    function () { throw new Error("Can't instantiate abstract classes"); },
    {
        size: abstractmethod,
        foreach: abstractmethod,
        isEmpty: function () { return this.size() == 0; },
        toString: function () {
            var s = "{", i = 0;
            this.foreach(function (v) {
                            if (i++>0) s += ", ";
                            s += v;
                        });
```

```
            return s + "}";
        },
        toLocaleString: function () {
            var s = "{", i = 0;
            this.foreach(function (v) {
                            if (i++>0) s += ", ";
                            if (v == null) s += v; // null和undefined
                        else s += v.toLocaleString(); // 其他的情况
                    });
            return s + "}";
        },
        toArray: function () {
            var a = [];
            this.foreach(function (v) { a.push(v); });
            return a;
        },
        equals: function (that) {
            if (! (that instanceof AbstractEnumerableSet)) return false;
            // 如果它们的大小不同，则它们不相等
            if (this.size() != that.size()) return false;
            //检查每一个元素是否也在that中
            try {
                this.foreach(function (v) { if (!that.contains(v)) throw false; });
                return true; // 所有的元素都匹配：集合相等
            } catch(x) {
                if (x === false) return false; // 集合不相等
                throw x; //发生了其他的异常：重新抛出异常
            }
        }
    });

/*
 * SingletonSet是AbstractEnumerableSet的非抽象子类
 * singleton集合是只读的，它只包含一个成员
 */
var SingletonSet = AbstractEnumerableSet.extend(
    function SingletonSet(member) { this.member = member; },
    {
        contains: function (x) { return x === this.member; },
        size: function () { return 1; },
        foreach: function (f, ctx) { f.call(ctx, this.member); }
    }
);

/*
 * AbstractWritableSet是AbstractEnumerableSet的抽象子类
 * 它定义了抽象方法add()和remove()
 * 然后实现了非抽象方法union()、intersection()和 difference()
 */
var AbstractWritableSet = AbstractEnumerableSet.extend(
    function () { throw new Error("Can't instantiate abstract classes");},
    {
        add: abstractmethod,
        remove: abstractmethod,
        union: function (that) {
```

```
            var self = this;
            that.foreach(function (v) { self.add(v); });
            return this;
        },
        intersection: function (that) {
            var self = this;
            this.foreach(function (v) { if (!that.contains(v)) self.remove(v);});
            return this;
        },
        difference: function (that) {
            var self = this;
            that.foreach(function (v) {self.remove(v);});
            return this;
        }
    });
/*
 * ArraySet是AbstractWritableSet的非抽象子类
 * 它以数组的形式表示集合中的元素
 * 对于它的contains()方法使用了数组的线性查找
 * 因为contains()方法的算法复杂度是O(n)而不是O(1)
 * 它非常适用于相对小型的集合，注意，这里的实现用到了ES5的数组方法indexOf()和forEach()
 */
var ArraySet = AbstractWritableSet.extend(
    function ArraySet() {
        this.values = [];
        this.add.apply(this, arguments);
    },
    {
        contains: function (v) { return this.values.indexOf(v) != -1; },
        size: function () { return this.values.length; },
        foreach: function (f, c) { this.values.forEach(f, c); },
        add: function () {
            for (var i = 0; i < arguments.length; i++) {
                var arg = arguments[i];
                if (!this.contains(arg)) this.values.push(arg);
            }
            return this;
        },
        remove: function () {
            for (var i = 0; i < arguments.length; i++) {
                var p = this.values.indexOf(arguments[i]);
                if (p == -1) continue;
                this.values.splice(p, 1);
            }
            return this;
        }
    }
);
```

9.8 ECMAScript 5 中的类

ECMAScript 5给属性特性增加了方法支持（getter、setter、可枚举性、可写性和可配置

性），而且增加了对象可扩展性的限制。这些方法在6.6节、6.7节和6.8.3节都有详细的讨论，然而这些方法非常适合用于类的定义。下面几节讲述了如何使用ECMAScript 5的特性来使类更加健壮。

9.8.1 让属性不可枚举

例9-6中的Set类使用了一个小技巧，将对象存储为“集合”的成员：它给添加至这个“集合”的任何对象定义了“对象id”属性。之后如果在`for/in`循环中对这个对象做遍历，这个新添加的属性[译注10]也会遍历到。ECMAScript 5可以通过设置属性为“不可枚举”（nonenumerable）来让属性不会遍历到。例9-17展示了如何通过`Object.defineProperty()`来做到这一点，同时也展示了如何定义一个getter函数以及检测对象是否是可扩展的（extensible）。

例9-17：定义不可枚举的属性

```
// 将代码包装在一个匿名函数中，这样定义的变量就在这个函数作用域内
(function() {
    //定义一个不可枚举的属性objectId，它可以被所有对象继承
    //当读取这个属性时调用getter函数
    //它没有定义setter，因此它是只读的
    //它是不可配置的，因此它是不能删除的
    Object.defineProperty(Object.prototype, "objectId", {
                          get: idGetter,            // 取值器
                          enumerable: false,        // 不可枚举的
                          configurable: false       // 不可删除的
                        });

    //当读取objectId的时候直接调用这个getter函数
    function idGetter() {          // getter函数返回该id
        if (! (idprop in this)) { // 如果对象中不存在id
            if (!Object.isExtensible(this)) // 并且可以增加属性
                throw Error("Can't define id for nonextensible objects");
            Object.defineProperty(this, idprop, {            // 给它一个值
                                  value: nextid++,           // 就是这个值
                                  writable: false,           // 只读的
                                  enumerable: false,         // 不可枚举的
                                  configurable: false        // 不可删除的
                                });
            }
            return this[idprop];   // 返回已有的或新的值
        };

        // idGetter()用到了这些变量，这些都属于私有变量
        var idprop = "|**objectId**|"; //假设这个属性没有用到
        var nextid = 1; // 给它设置初始值

} ()); // 立即执行这个包装函数
```

译注10：这里指的是“对象id”属性。

9.8.2 定义不可变的类

除了可以设置属性为不可枚举的，ECMAScript 5还可以设置属性为只读的，当我们希望类的实例都是不可变的，这个特性非常有帮助。例9-18使用Object.defineProperties()和Object.create()定义不可变的Range类。它同样使用Object.defineProperties()来为类创建原型对象，并将（原型对象的）实例方法设置为不可枚举的，就像内置类的方法一样。不仅如此，它还将这些实例方法设置为“只读”和“不可删除”，这样就可以防止对类做任何修改（monkey-patching）[译注11]。最后，例9-18展示了一个有趣的技巧，其中实现的构造函数也可以用做工厂函数，这样不论调用函数之前是否带有new关键字，都可以正确地创建实例。

例 9-18：创建一个不可变的类，它的属性和方法都是只读的

```
// 这个方法可以使用new调用，也可以省略new，它可以用做构造函数也可以用做工厂函数
function Range(from, to) {
    // 这些是对from和to只读属性的描述符
    var props = {
        from: {value: from, enumerable: true, writable: false, configurable: false},
        to: { value: to, enumerable: true, writable: false, configurable: false}
    };

    if (this instanceof Range)                    // 如果作为构造函数来调用
        Object.defineProperties(this, props);     // 定义属性
    else                                          // 否则，作为工厂方法来调用
        return Object.create(Range.prototype,     // 创建并返回这个新Range对象，
                             props);              // 属性由props指定
}

// 如果用同样的方法给Range.prototype对象添加属性
// 那么我们需要给这些属性设置它们的特性
// 因为我们无法识别出它们的可枚举性、可写性或可配置性，这些属性特性默认都是false
Object.defineProperties(Range.prototype, {
    includes: {
        value: function(x) { return this.from <= x && x <= this.to; }
    },
    foreach: {
        value: function(f) {
            for (var x = Math.ceil(this.from); x <= this.to; x++) f(x);
        }
    },
    toString: {
        value: function() { return "(" + this.from + "..." + this.to + ")"; }
    }
});
```

例9-18用到了Object.defineProperties()和Object.create()来定义不可变的和不可枚

译注11：Monkey-patching是指修改现有对象的原型，在JavaScript中，修改对象的原型就相当于修改了实例化它的类。

举的属性。这两个方法非常强大，但属性描述符对象让代码的可读性变得更差。另一种改进的做法是将修改这个已定义属性的特性的操作定义为一个工具函数，例9-19展示了两个这样的工具函数：

例9-19：属性描述符工具函数

```
// 将o的指定名字（或所有）的属性设置为不可写的和不可配置的
function freezeProps(o) {
    var props = (arguments.length == 1)                    // 如果只有一个参数
        ? Object.getOwnPropertyNames(o)                    // 使用所有的属性
        : Array.prototype.splice.call(arguments, 1);       // 否则传入了指定名字的属性
    props.forEach(function(n) {    // 将它们都设置为只读的和不可变的
        // 忽略不可配置的属性
        if (!Object.getOwnPropertyDescriptor(o, n).configurable) return;
        Object.defineProperty(o, n, { writable: false, configurable: false });
    });
    return o; // 所以我们可以继续使用它
}

// 将o的指定名字（或所有）的属性设置为不可枚举的和可配置的
function hideProps(o) {
    var props = (arguments.length == 1)                    // 如果只有一个参数
        ? Object.getOwnPropertyNames(o)                    // 使用所有的属性
        : Array.prototype.splice.call(arguments, 1);       // 否则传入了指定名字的属性
    props.forEach(function(n) { // 将它们设置为不可枚举的
        // 忽略不可配置的属性
        if (!Object.getOwnPropertyDescriptor(o, n).configurable) return;
        Object.defineProperty(o, n, { enumerable: false });
    });
    return o;
}
```

Object.defineProperty()和Object.defineProperties()可以用来创建新属性，也可以修改已有属性的特性。当用它们创建新属性时，默认的属性特性的值都是false。但当用它们修改已经存在的属性时，默认的属性特性依然保持不变。比如，在上面的hideProps()函数中，只指定了enumerable特性，因为我们只想修改enumerable特性。

使用这些工具函数，就可以充分利用ECMAScript 5的特性来实现一个不可变的类，而且不用动态地修改这个类。例9-20中不可变的Range类就用到了刚才定义的工具函数。

例9-20：一个简单的不可变的类

```
function Range(from, to) {          // 不可变的类Range的构造函数
    this.from = from;
    this.to = to;
    freezeProps(this);              // 将属性设置为不可变的
}

Range.prototype = hideProps({       // 使用不可枚举的属性来定义原型
    constructor: Range,
    includes: function(x) { return this.from <= x && x <= this.to; },
```

```
    foreach: function(f) {for(var x=Math.ceil(this.from);x<=this.to;x++) f(x);},
    toString: function() { return "(" + this.from + "..." + this.to + ")"; }
});
```

9.8.3 封装对象状态

如9.6.6节和例9-10所示，构造函数中的变量和参数可以用做它创建的对象的私有状态。该方法在ECMAScript 3中的一个缺点是，访问这些私有状态的存取器方法是可以替换的。在ECMAScript 5中可以通过定义属性getter和setter方法将状态变量更健壮地封装起来，这两个方法是无法删除的，如例9-21所示。

例9-21：将Range类的端点严格封装起来

```
// 这个版本的Range类是可变的，但将端点变量进行了良好的封装
// 但端点的大小顺序还是固定的：from <= to
function Range(from, to) {
    // 如果from大于to
    if (from > to) throw new Error("Range: from must be <= to");

    // 定义存取器方法以维持不变
    function getFrom() { return from; }
    function getTo() { return to; }
    function setFrom(f) {          // 设置from的值时，不允许from大于to
        if (f <= to) from = f;
        else throw new Error("Range: from must be <= to");
    }
    function setTo(t) {            // 设置to的值时，不允许to小于from
        if (t >= from) to = t;
        else throw new Error("Range: to must be >= from");
    }

    // 将使用取值器的属性设置为可枚举的、不可配置的
    Object.defineProperties(this, {
        from: { get: getFrom, set: setFrom, enumerable: true, configurable: false },
        to: { get: getTo, set: setTo, enumerable: true, configurable: false }
    });
}

// 和前面的例子相比，原型对象没有做任何修改
// 实例方法可以像读取普通的属性一样读取from和to
Range.prototype = hideProps({
    constructor: Range,
    includes: function(x) { return this.from <= x && x <= this.to; },
    foreach: function(f) {for (var x = Math.ceil(this.from); x <= this.to; x++) f(x);},
    toString: function() { return "(" + this.from + "..." + this.to + ")"; }
});
```

9.8.4 防止类的扩展

通常认为，通过给原型对象添加方法可以动态地对类进行扩展，这是JavaScript本身的

特性。ECMAScript 5可以根据需要对此特性加以限制。Object.preventExtensions()可以将对象设置为不可扩展的（见6.8.3节），也就是说不能给对象添加任何新属性。Object.seal()则更加强大，它除了能阻止用户给对象添加新属性，还能将当前已有的属性设置为不可配置的，这样就不能删除这些属性了（但不可配置的属性可以是可写的，也可以转换为只读属性）。可以通过这样一句简单的代码来阻止对Object.prorotype的扩展：

```
Object.seal(Object.prototype);
```

JavaScript的另外一个动态特性是“对象的方法可以随时替换”（或称为“monkey-patch”）：

```
var original_sort_method = Array.prototype.sort;
Array.prototype.sort = function() {
    var start = new Date();
    original_sort_method.apply(this, arguments);
    var end = new Date();
    console.log("Array sort took " + (end - start) + " milliseconds.");
};
```

可以通过将实例方法设置为只读来防止这类修改，一种方法就是使用上面代码所定义的freezeProps()工具函数。另外一种方法是使用Object.freeze()，它的功能和Object.seal()完全一样，它同样会把所有属性都设置为只读的和不可配置的。

理解类的只读属性的特性至关重要。如果对象o继承了只读属性p，那么给o.p的赋值操作将会失败，就不会给o创建新属性。如果你想重写一个继承来的只读属性，就必须使用Object.definePropertiy()、Object.defineProperties()或Object.create()来创建这个新属性。也就是说，如果将类的实例方法设置为只读的，那么重写它的子类的这些方法的难度会更大。

这种锁定原型对象的做法往往没有必要，但的确有一些场景是需要阻止对象的扩展的。回想一下例9-7中的enumeration()，这是一个类工厂函数。这个函数将枚举类型的每个实例都保存在构造函数对象的属性里，以及构造函数的values数组中。这些属性和数组是表示枚举类型实例的正式实例列表，是可以执行“冻结”（freezing）操作的，这样就不能给它添加新的实例，已有的实例也无法删除或修改。可以给enumeration()函数添加几行简单的代码：

```
Object.freeze(enumeration.values);
Object.freeze(enumeration);
```

需要注意的是，通过在枚举类型中调用Object.freeze()，例9-17中定义的objectId属性之后也无法使用了。这个问题的解决办法是，在枚举类型被“冻结”之前读取一次它的objectId属性（调用潜在的存取器方法并设置内部属性）。

9.8.5 子类和ECMAScript 5

例9-22使用ECMAScript 5的特性实现子类。这里使用例9-16中的AbstractWritableSet类来做进一步说明，来定义这个类的子类StringSet。下面这个例子的最大特点是使用Object.create()创建原型对象，这个原型对象继承自父类的原型，同时给新创建的对象定义属性。这种实现方法的困难之处在于，正如上文所提到的，它需要使用难看的属性描述符。

这个例子中另外一个有趣之处在于，使用Object.create()创建对象时传入了参数null，这个创建的对象没有任何继承任何成员。这个对象用来存储集合的成员，同时，这个对象没有原型，这样我们就能对它直接使用in运算符[译注12]，而不须使用hasOwnProperty()方法。

例9-22：StringSet：利用ECMAScript 5的特性定义的子类

```
function StringSet() {
    this.set = Object.create(null); // 创建一个不包含原型的对象
    this.n = 0;
    this.add.apply(this, arguments);
}

// 注意，使用Object.create()可以继承父类的原型
// 而且可以定义单独调用的方法，因为我们没有指定属性的可写性、可枚举性和可配置性
// 因此这些属性特性的默认值都是false
// 只读方法让这个类难于子类化（被继承）
StringSet.prototype = Object.create(AbstractWritableSet.prototype, {
    constructor: { value: StringSet },
    contains: { value: function(x) { return x in this.set; } },
    size: { value: function(x) { return this.n; } },
    foreach: { value: function(f, c) { Object.keys(this.set).forEach(f, c); } },
    add: {
        value: function() {
            for (var i = 0; i < arguments.length; i++) {
                if (!(arguments[i] in this.set)) {
                    this.set[arguments[i]] = true;
                    this.n++;
                }
            }
            return this;
        }
    },
    remove: {
```

译注12：使用in运算符可以对对象成员进行遍历，包括对原型对象中的非内置成员进行遍历。

```
        value: function() {
            for (var i = 0; i < arguments.length; i++) {
                if (arguments[i] in this.set) {
                    delete this.set[arguments[i]];
                    this.n--;
                }
            }
            return this;
        }
    }
});
```

9.8.6 属性描述符

6.7节讨论了ECMAScript 5中的属性描述符，但没有给出它们的示例代码。本节给出一个例子，用来讲述基于ECMAScript 5如何对属性进行各种操作。在例9-23中给Object.prototype添加了properties()方法（这个方法是不可枚举的）。这个方法的返回值是一个对象，用以表示属性的列表，并定义了有用的方法用来输出属性和属性特性（对于调试非常有用），用来获得属性描述符（当复制属性同时复制属性特性时非常有用）以及用来设置属性的特性（是上文定义的hideProps()和freezeProps()函数不错的替代方案）。这个例子展示了ECMAScript 5的大多数属性相关的特性，同时使用了一种模块编程技术，这将在下一节讨论。

例9-23：ECMAScript 5属性操作

```
/*
 * 给Object.prototype定义properties()方法，
 * 这个方法返回一个表示调用它的对象上的属性名列表的对象
 * （如果不带参数调用它，就表示该对象的所有属性）
 * 返回的对象定义了4个有用的方法：toString()、 descriptors()、hide()和 show()
 */
(function namespace() { // 将所有逻辑闭包在一个私有函数作用域中

    // 这个函数成为所有对象的方法
    function properties() {
        var names; // 属性名组成的数组
        if (arguments.length == 0) // 所有的自有属性
            names = Object.getOwnPropertyNames(this);
        else if (arguments.length == 1 && Array.isArray(arguments[0]))
            names = arguments[0]; // 名字组成的数组
        else                      // 参数列表本身就是名字
            names = Array.prototype.splice.call(arguments, 0);

        // 返回一个新的Properties对象，用以表示属性名字
        return new Properties(this, names);
    }
    // 将它设置为Object.prototpye的新的不可枚举的属性
    // 这是从私有函数作用域导出的唯一一个值
    Object.defineProperty(Object.prototype, "properties", {
        value: properties,
```

```
    enumerable: false, writable: true, configurable: true
});

// 这个构造函数是由上面的properties()函数所调用的
// Properties类表示一个对象的属性集合
function Properties(o, names) {
    this.o = o;                 // 属性所属的对象
    this.names = names;         // 属性的名字
}

// 将代表这些属性的对象设置为不可枚举的
Properties.prototype.hide = function() {
    var o = this.o, hidden = { enumerable: false };
    this.names.forEach(function(n) {
                        if (o.hasOwnProperty(n))
                            Object.defineProperty(o, n, hidden);
                    });
    return this;
};

// 将这些属性设置为只读的和不可配置的
Properties.prototype.freeze = function() {
    var o = this.o, frozen = { writable: false, configurable: false };
    this.names.forEach(function(n) {
                            if (o.hasOwnProperty(n))
                                Object.defineProperty(o, n, frozen);
                        });
    return this;
};

// 返回一个对象，这个对象是名字到属性描述符的映射表
// 使用它来复制属性，连同属性特性一起复制
// Object.defineProperties(dest, src.properties().descriptors());
Properties.prototype.descriptors = function() {
    var o = this.o, desc = {};
    this.names.forEach(function(n) {
                            if (!o.hasOwnProperty(n)) return;
                            desc[n] = Object.getOwnPropertyDescriptor(o, n);
                        });
    return desc;
};

// 返回一个格式化良好的属性列表
// 列表中包含名字、值和属性特性，使用"permanent"表示不可配置
// 使用"readonly"表示不可写，使用"hidden"表示不可枚举
// 普通的可枚举、可写和可配置属性不包含特性列表
Properties.prototype.toString = function() {
    var o = this.o; // 在下面嵌套的函数中使用
    var lines = this.names.map(nameToString);
    return "{\n " + lines.join(",\n ") + "\n}";

    function nameToString(n) {
        var s = "", desc = Object.getOwnPropertyDescriptor(o, n);
        if (!desc) return "nonexistent " + n + ": undefined";
        if (!desc.configurable) s += "permanent ";
```

```
            if ((desc.get && !desc.set) || !desc.writable) s += "readonly ";
            if (!desc.enumerable) s += "hidden ";
            if (desc.get || desc.set) s += "accessor " + n
            else s += n + ": " + ((typeof desc.value === "function") ? "function"
                                                                    : desc.value);
            return s;
        }
    };

    // 最后，将原型对象中的实例方法设置为不可枚举的
    // 这里用到了刚定义的方法
    Properties.prototype.properties().hide();
} ()); // 立即执行这个匿名函数
```

9.9 模块

将代码组织到类中的一个重要原因是，让代码更加“模块化”，可以在很多不同场景中实现代码的重用。但类不是唯一的模块化代码的方式。一般来讲，模块是一个独立的JavaScript文件。模块文件可以包含一个类定义、一组相关的类、一个实用函数库或者是一些待执行的代码。只要以模块的形式编写代码，任何JavaScript代码段就可以当做一个模块[译注13]。JavaScript中并没有定义用以支持模块的语言结构（但`imports`和`exports`的确是JavaScript保留的关键字，因此JavaScript的未来版本可能会支持），这也意味着在JavaScript中编写模块化的代码更多的是遵循某一种编码约定。

很多JavaScript库和客户端编程框架都包含一些模块系统。比如，Dojo工具包和Google的Closure库定义了`provide()`和`require()`函数，用以声明和加载模块。并且，CommonJS服务器端JavaScript标准规范（参照*http://commonjs.org*）创建了一个模块规范，后者同样使用`require()`函数。这种模块系统通常用来处理模块加载和依赖性管理，这些内容已经超出本书的讨论范围。如果使用这些框架，则必须按照框架提供的模块编写约定来定义模块。本节仅对模块约定做一些简单的讨论。

模块化的目标是支持大规模的程序开发，处理分散源中代码的组装，并且能让代码正确运行，哪怕包含了作者所不期望出现的模块代码，也可以正确执行代码。为了做到这一点，不同的模块必须避免修改全局执行上下文，因此后续模块应当在它们所期望运行的原始（或接近原始）上下文中执行[译注14]。这实际上意味着模块应当尽可能少地定义全局

译注13：作者这里的表述是围绕“模块是一个可重用的代码片段”这一观念的，不论是从代码语法结构上解耦，还是将代码拆分至不同的文件中，只要用某种方法将代码“分离”，就认为是一个模块，因此作者说任何代码都可以处理为一个模块。

译注14：这里的“原始上下文”是指调用模块时所在的上下文，可能处在一个很深的闭包当中，但这个模块的逻辑不应该影响到其他的上下文特别是全局上下文。

标识，理想状况是，所有模块都不应当定义超过一个（全局标识）。接下来我们给出的一种简单的方法可以做到这一点。你会发现在JavaScript中实现一个模块代码并不困难：在本书中很多示例代码都用到了这种技术。

9.9.1 用做命名空间的对象

在模块创建过程中避免污染全局变量的一种方法是使用一个对象作为命名空间。它将函数和值作为命名空间对象属性存储起来（可以通过全局变量引用），而不是定义全局函数和变量。拿例9-6的Set类来说，它定义了一个全局构造函数Set()。然后给这个类定义了很多实例方法，但将这些实例方法存储为Set.prototype的属性，因此这些方法不是全局的。示例代码也包含一个_v2s()工具函数，但也没有定义它为全局函数，而是把它存储为Set的属性。

接下来看一下例9-16，这个例子定义了很多抽象类和非抽象类。每个类都只包含一个全局标识，但整个模块（这个JavaScript文件）定义了很少的全局变量。基于这种“保持干净的全局命名空间”的观点，一种更好的做法是将“集合”类定义为一个单独的全局对象：

```
var sets = {};
```

这个sets对象是模块的命名空间，并且将每个“集合”类都定义为这个对象的属性：

```
sets.SingletonSet = sets.AbstractEnumerableSet.extend(...);
```

如果想使用这样定义的类，需要通过命名空间来调用所需的构造函数：

```
var s = new sets.SingletonSet(1);
```

模块的作者并不知道他的模块会和哪些其他模块一起工作，因此尤为注意这种命名空间的用法带来的命名冲突。然而，使用这个模块的开发者是知道它用了哪些模块、用到了哪些名字的。程序员并不一定要严格遵守命名空间的写法，只需将常用的值“导入”到全局命名空间中。程序员如果要经常使用sets命名空间中的Set类，可以这样将它导入：

```
var Set = sets.Set;          // 将Set导入到全局命名空间中
var s = new Set(1,2,3);      // 这样每次使用它就不必加set前缀了
```

有时模块作者会使用更深层嵌套的命名空间。如果sets模块是另外一组更大的模块集合的话，它的命名空间可能会是collections.sets，模块代码的开始会这样写：

```
var collections;             // 声明（或重新声明）这个全局变量
if (!collections)            // 如果它原本不存在
    collections = {};        // 创建一个顶层的命名空间对象
collections.sets = {}        // 将sets命名空间创建在它的内部
```

```
// 在collections.sets内定义set类
collections.sets.AbstractSet = function() { ... }
```

最顶层的命名空间往往用来标识创建模块的作者或组织，并避免命名空间的命名冲突。比如，Google的Closure库在它的命名空间goog.structs中定义了Set类。每个开发者都反转互联网域名的组成部分，这样创建的命名空间前缀是全局唯一的，一般不会被其他模块作者采用。比如我的网站是*davidflanagan.com*，我可以通过这个命名空间来发布我的sets模块：`com.davidflanagan.clooectinos.sets`。

使用很长的命名空间来导入模块的方式非常重要，然而程序员往往将整个模块导入全局命名空间，而不是导入（命名空间中的某个）单独的类。

```
var sets = com.davidflanagan.collections.sets;
```

按照约定，模块的文件名应当和命名空间匹配。sets模块应当保存在文件*sets.js*中。如果这个模块使用命名空间collections.sets，那么这个文件应当保存在目录*collections/*下（这个目录还应当包含另一个文件*maps.js*）。并且使用命名空间 *com.davidflanagan.collections.sets*的模块应当在文件*com/davidflanagan/collections/sets.js*中。

9.9.2 作为私有命名空间的函数

模块对外导出一些公用API，这些API是提供给其他程序员使用的，它包括函数、类、属性和方法。但模块的实现往往需要一些额外的辅助函数和方法，这些函数和方法并不需要在模块外部可见。比如，例9-6中的`Set._v2s()`函数，模块作者不希望Set类的用户在某时刻调用这个函数，因此这个方法最好在类的外部是不可访问的。

可以通过将模块（本例中的Set类）定义在某个函数的内部来实现。正如8.5节所描述的一样，在一个函数中定义的变量和函数都属于函数的局部成员，在函数的外部是不可见的。实际上，可以将这个函数作用域用做模块的私有命名空间（有时称为“模块函数”）。例9-24展示了如何使用“模块函数”来实现Set类：

例9-24：模块函数中的Set类

```
// 声明全局变量Set，使用一个函数的返回值给它赋值
// 函数结束时紧跟的一对圆括号说明这个函数定义后立即执行
// 它的返回值将赋值给Set，而不是将这个函数赋值给Set
// 注意它是一个函数表达式，不是一条语句，因此函数"invocation"并没有创建全局变量
var Set = (function invocation() {

    function Set() { // 这个构造函数是局部变量
        this.values = {};          // 这个对象的属性用来保存这个集合
        this.n = 0;                // 集合中值的个数
        this.add.apply(this, arguments); // 将所有的参数都添加至集合中
    }
```

```
    // 给Set.prototype定义实例方法
    // 这里省略了详细代码
    Set.prototype.contains = function(value) {
        // 注意我们调用了v2s(), 而不是调用带有笨重的前缀的set._v2s()
        return this.values.hasOwnProperty(v2s(value));
    };
    Set.prototype.size = function() { return this.n; };
    Set.prototype.add = function() { /* ... */ };
    Set.prototype.remove = function() { /* ... */ };
        Set.prototype.foreach = function(f, context) { /* ... */ };

    // 这里是上面的方法用到的一些辅助函数和变量
    // 它们不属于模块的共有API, 但它们都隐藏在这个函数作用域内
    // 因此我们不必将它们定义为Set的属性或使用下划线作为其前缀
    function v2s(val) { /* ... */ }
    function objectId(o) { /* ... */ }
    var nextId = 1;
    // 这个模块的共有API是Set()构造函数
    // 我们需要把这个函数从私有命名空间中导出来
    // 以便在外部也可以使用它, 在这种情况下, 我们通过返回这个构造函数来导出它
    // 它变成第一行代码所指的表达式的值
    return Set;
} ()); // 定义函数后立即执行
```

注意，这里使用了立即执行的匿名函数，这在JavaScript中是一种惯用法。如果想让代码在一个私有命名空间中运行，只须给这段代码加上前缀“(function(){“和后缀”}())”。开始的左圆括号确保这是一个函数表达式，而不是函数定义语句，因此可以给该前缀添加一个函数名来让代码变得更加清晰。在例9-24中使用了名字“invocation”，用以强调这个函数应当在定义之后立即执行。名字“namespace”也可以用来强调这个函数被用做命名空间。

一旦将模块代码封装进一个函数，就需要一些方法导出其公用API，以便在模块函数的外部调用它们。在例9-24中，模块函数返回构造函数，这个构造函数随后赋值给一个全局变量。将值返回已经清楚地表明API已经导出在函数作用域之外。如果模块API包含多个单元，则它可以返回命名空间对象。对于sets模块来说，可以将代码写成这样：

```
    // 创建一个全局变量用来存放集合相关的模块
    var collections;
    if (!collections) collections = {};

    // 定义sets模块
    collections.sets = (function namespace() {
        // 在这里定义多种"集合"类, 使用局部变量和函数
        // ……这里省略很多代码……

        // 通过返回命名空间对象将API导出
        return {
            // 导出的属性名: 局部变量名字
            AbstractSet: AbstractSet,
```

```
        NotSet: NotSet,
        AbstractEnumerableSet: AbstractEnumerableSet,
        SingletonSet: SingletonSet,
        AbstractWritableSet: AbstractWritableSet,
        ArraySet: ArraySet
    };
}());
```

另外一种类似的技术是将模块函数当做构造函数，通过new来调用，通过将它们[译注15]赋值给this来将其导出[译注16]：

```
var collections;
if (!collections) collections = {};
collections.sets = (new function namespace() {
    //  ……这里省略很多代码……

    // 将API导出至this对象
    this.AbstractSet = AbstractSet;
    this.NotSet = NotSet;      // ……

    // 注意，这里没有返回值
}());
```

作为一种替代方案，如果已经定义了全局命名空间对象，这个模块函数可以直接设置那个对象的属性，不用返回任何内容：

```
var collections;
if (!collections) collections = {};
collections.sets = {};
(function namespace() {
    // ……这里省略很多代码……

    // 将共用API导出到上面创建的命名空间对象上
    collections.sets.AbstractSet = AbstractSet;
    collections.sets.NotSet = NotSet; // ……

    // 导出的操作已经执行了，这里不需要再写return语句了
}());
```

有些框架实现了模块加载功能，其中包括其他一些导出模块API的方法。比如，使用provides()函数来注册其API，提供exports对象[译注17]用以存储模块API。由于JavaScript目前还不具备模块管理的能力，因此应当根据所使用的框架和工具包来选择合适的模块创建和导出API的方式。

译注15：这里作者所说的“它们”是指构造函数创建的新实例。

译注16：使用构造函数和模块函数来实现私有成员的原理是一模一样的，只是调用的方式不一样。

译注17：可以参照CommonJS规范*http://commonjs.org*。