JavaScript
面向对象编程

随书赠送

目　录

JavaScript 面向对象编程

本讲将系统、简略的讲解 JavaScript 中对象编程。包括如何创建并使用对象，什么是构造器函数，JavaScript 内置对象及其运用。

【学习重点】

- 理解数组和对象的结构异同。
- 正常操作对象。
- 灵活使用对象的属性和方法。
- 熟悉内置对象。

E.1 从数组到对象

数组实际上就是一组值的列表。该列表中的每一个值都有自己的索引值（即数字键名），索引值从 0 开始，依次递增。例如：

```
var myarr = ['red', 'blue', 'yellow', 'purple'];
myarr;                                  //["red", "blue", "yellow", "purple"];
myarr[0];                                   //"red"
myarr[3];                                   //"purple"
```

如果我们将索引键单独排成一列，再把对应的值排成另一列，就会列出一个键/值表。

```
0       red
1       blue
2       yellow
3       purple
```

对象的结构跟数组很相似，唯一的不同是它的键值类型是自定义的。也就是说，我们的索引方式不再局限于数字了，而可以使用一些更为友好

的键名，如 first_name、age 等。

【示例 1】下面示例演示对象是由哪几部分组成的。

```
var hero = {
    breed: 'Turtle',
    occupation: 'Ninja'
};
```

总结：

- 有一个用于表示该对象的变量名 hero。
- 与定义数组时所用的中括号[]不同，对象使用的是大括号{}。
- 括号中用逗号分隔的是组成该对象的元素（通常被称之为属性）。
- 键/值对之间用冒号分割，例如，key: value。
- 有时候，还可以在键名（属性名）上面加一对引号，例如，下面三行代码所定义的内容是完全相同的。

```
var hero = {occupation: 1};
var hero = {"occupation": 1};
var hero = {'occupation': 1};
```

通常情况下，不建议在属性名上面加引号（这也能减少一些输入），但在以下这些情境中，引号是必需的。

- 如果属性名是 JavaScript 中的保留字之一的话。
- 如果属性名中包含空格或其他特殊字符的话（包括任何除字母、数字、下划线及美元符号以外的字符）。
- 如果属性名以数字开头的话。

总而言之，如果所选的属性名不符合 JavaScript 中的变量命名规则，就必须对其施加一对引号。

【示例 2】下面定义一个怪异的对象。

```
var o = {
    $omething: 1,
    'yes or no': 'yes',
    '!@#$%^&*': true
```

```
};
```

虽然这个对象的属性名看起来很另类，但该对象是合法的，因为它的第二和第三个属性名上加了引号，否则一定会出错。

E.1.1 元素、属性、方法与成员

说到数组的时候，其中包含的是元素。而说对象时，就会说其中包含的是属性。实际上对于 JavaScript 来说，它们并没有多大的区别，只是在技术术语上的表达习惯有所不同罢了。这也是它区别于其他程序设计语言的地方。

另外，对象的属性也可以是函数，因为函数本身也是一种数据。在这种情况下，我们称该属性为方法。例如，下面的 talk 就是一个方法。

```
var dog = {
    name: 'Benji',
    talk: function(){
        alert('Woof, woof!');
    }
};
```

也可以在数组中存储一些函数元素，并在需要时调用它们，但这在实践中并不多见。

```
var a = [];
a[0] = function(what){ alert(what); };
a[0]('Boo!');
```

有时候可能还会看到一个对象的属性指向另一个对象属性的情况。而且所指向的属性也可以是函数。

E.1.2 哈希表、关联型数组

在一些语言中，会存在着两种不同的数组形式。

- 一般性数组，也叫作索引型数组或者枚举型数组（通常以数字为键名）。
- 关联型数组，也叫作哈希表或者字典（通常以字符串为键值）。

在 JavaScript 中，我们会用数组来表示索引型数组，而用对象来表示关联型数组。因此，如果想在 JavaScript 中使用哈希表，就必须要用到对象。

E.1.3 访问对象属性

我们可以通过以下两种方式来访问对象的属性。

- 中括号表示法，例如：hero['occupation']。
- 点号表示法，例如：hero.occupation。

相对而言，点号表示法更易于读写，但也不是总能适用的。这一规则也适用于引用属性名，如果我们所访问的属性不符合变量命名规则，它就不能通过点号表示法来访问。

【示例】下面示例演示一下这两种表示法。

```
var hero = {
    breed: 'Turtle',
    occupation: 'Ninja'
};
```

点号表示法来访问属性：

```
hero.breed;                          //"Turtle"
```

中括号表示法来访问属性：

```
hero['occupation'];                  //"Ninja"
```

如果访问的属性不存在，代码就会返回 undefined。

```
'Hair color is ' + hero.hair_color;          //"Hair color is undefined"
```

由于对象中可以包含任何类型的数据，自然也包括其他对象：

```
var book = {
    name: 'Catch-22',
    published: 1961,
    author: {
        firstname: 'Joseph',
        lastname: 'Heller'
    }
};
```

在这里，如果想访问 book 对象的 author 属性对象的 firstname 属性，就需要这样：

```
book.author.firstname;                       //"Joseph"
```

当然，也可以连续使用中括号表示法：

```
book['author']['lastname'];                  //"Heller"
```

甚至可以混合使用这两种表示法：

```
book.author['lastname'];                     //"Heller"
book['author'].lastname;                     //"Heller"
```

如果要访问的属性名是不确定的，就必须使用中括号表示法了，它允许我们在运行时通过变量来实现相关属性的动态存取。

```
var key = 'firstname';
book.author[key];                            //"Joseph"
```

E.1.4 调用对象方法

由于对象方法实际上只是一个函数类型的属性，因此访问方式与属性完全相同，即用点号表示法或中括号表示法均可。其调用（请求）方式也与其他函数相同，在指定的方法名后加一对括号即可。例如：

```
var hero = {
    breed: 'Turtle',
    occupation: 'Ninja',
    say: function() {
        return 'I am ' + hero.occupation;
    }
};
hero.say();                                    //"I am Ninja"
```

如果调用方法时需要传递一些参数，做法也和一般函数一样。例如：

```
hero.say('a', 'b', 'c');
```

由于我们可以像访问数组一样用中括号来访问属性，因此这意味着我们同样可以用中括号来调用方法。

```
hero['say']();
```

使用中括号来调用方法在实践中并不常见，除非属性名是在运行时定义的。

```
var method = 'say';
hero[method]();
```

【提示】

尽量别使用引号（除非别无他法）。尽量使用点号表示法来访问对象的方法与属性。不要在对象中使用带引号的属性标识。

E.1.5 修改属性与方法

由于 JavaScript 是一种动态语言，所以它允许我们随时对现存对象的属性和方法进行修改。其中自然也包括添加与删除属性。因此，我们也可以先创建一个空对象，稍后再为它添加属性。

首先，创建一个“空”对象：

```
var hero = {};
```

实际上这个对象并不是空的。虽然我们并没有为它定义属性，但它本身有一些继承的属性。在 ES5 中，我们确实是可以真正创建一个不继承任何属性的空对象的。

这时要访问一个不存在的属性：

```
typeof hero.breed;                    //"undefined"
```

现在，我们来为该对象添加一些属性和方法：

```
hero.breed = 'turtle';
hero.name = 'Leonardo';
hero.sayName = function() {
    return hero.name;
};
```

然后调用该方法：

```
hero.sayName();                       //"Leonardo"
```

接下来，删除一个属性：

```
delete hero.name;                     //true
```

然后再调用该方法，它就找不到 name 属性了：

```
hero.sayName();                       //"undefined"
```

在 JavaScript 中，对象在任何时候都是可以改变的，例如,增加、删除、修改属性。但这种规则也有例外的情况：某些内建对象的一些属性是不可改变的，如 Math.PI。另外，ES5 允许创建不可改变的对象。

E.1.6 使用 this

在上面示例中，方法 sayName()是直接通过 hero.name 来访问 hero 对象的 name 属性的。而事实上，当我们处于某个对象的方法内部时，还可以用另一种方法来访问同一对象的属性，即该对象的特殊值 this。例如：

```
var hero = {
    name: 'Rafaelo',
```

```
        sayName: function() {
            return this.name;
        }
    };
    hero.sayName();                                    //"Rafaelo"
```

也就是说，当我们引用 this 值时，实际上所引用的就是“这个对象”或者“当前对象”。

E.1.7 构造器函数

我们还可以通过构造器函数（constructor function）的方式来创建对象。下面来看一个例子。

```
    function Hero() {
        this.occupation = 'Ninja';
    }
```

为了能使用该函数来创建对象，我们需要使用 new 操作符，例如：

```
    var hero = new Hero();
    hero.occupation;                                   //"Ninja"
```

使用构造器函数的好处之一是它可以在创建对象时接收一些参数。下面，我们就来修改一下上面的构造器函数，使它可以通过接收参数的方式来设定 name 属性。

```
    function Hero(name) {
        this.name = name;
        this.occupation = 'Ninja';
        this.whoAreYou = function() {
            return "I'm " + this.name + " and I'm a " +this.occupation;
        };
    }
```

现在，我们就能利用同一个构造器来创建不同的对象了。

```
var h1 = new Hero('Michelangelo');
var h2 = new Hero('Donatello');
h1.whoAreYou();                    //"I'm Michelangelo and I'm a Ninja"
h2.whoAreYou();                    //"I'm Donatello and I'm a Ninja"
```

【提示】

依照惯例，我们应该将构造器函数的首字母大写，以便显著地区别于其他一般函数。

如果在调用一个构造器函数时忽略了 new 操作符，尽管代码不会出错，但它的行为可能会令人出乎预料，例如：

```
var h = Hero('Leonardo');
typeof h;                          //"undefined"
```

由于这里没有使用 new 操作符，因此我们不是在创建一个新的对象。这个函数调用与其他函数并没有区别，这里的 h 值应该就是该函数的返回值。

而由于该函数没有显式返回值（它没有使用关键字 return），所以它实际上返回的是 undefined 值，并将该值赋值给了 h。那么，在这种情况下 this 引用的是全局对象。

E.1.8 全局对象

事实上，程序所在的宿主环境一般都会为其提供一个全局对象，而所谓的全局变量其实都只不过是该对象的属性罢了。

例如，当程序的宿主环境是 Web 浏览器时，它所提供的全局对象就是 window。另一种获取全局对象的方法（这种方法在浏览器以外的大多数其他环境也同样有效）是在构造器函数之外使用 this 关键字。例如，可以在任何函数之外的全局代码部分这么做。

下面，我们来看一个具体示例。首先，我们在所有函数之外声明一个

全局变量，例如：

```
var a = 1;
```

然后，我们就可以通过各种不同的方式来访问该全局变量了。

- 可以当作一个变量 a 来访问。
- 可以当作全局对象的一个属性来访问，例如 window['a']或者 window.a。
- 可以通过 this 所指向的全局对象属性来访问。例如：

```
var a = 1;
window.a;                                    //1
this.a;                                      //1
```

当我们声明了一个构造函数，但又不通过 new 来调用它时，代码就会返回 undefined。

```
function Hero(name) {
    this.name = name;
}
var h = Hero('Leonardo');
typeof h;                                    //"undefined"
typeof h.name;      //TypeError: Cannot read property 'name' of undefined
```

由于我们在 Hero 中使用了 this，所以这里就会创建一个全局变量（同时也是全局对象的一个属性）。

```
name;                                        //"Leonardo"
window.name;                                 //"Leonardo"
```

而如果我们使用 new 来调用相同的构造器函数，就会创建一个新对象，并且 this 也会自动指向该对象。

```
var h2 = new Hero('Michelangelo');
typeof h2;                                   //"object"
h2.name;                                     //"Michelangelo"
```

下面两个调用的效果完全相同：

```
parseInt('101 dalmatians');                  //101
```

```
window.parseInt('101 dalmatians');          //101
```

如果在所有函数之外，这样使用也是可以的：

```
this.parseInt('101 dalmatians');            //101
```

E.1.9 构造器属性

当我们创建对象时，实际上同时也赋予了该对象一种特殊的属性—即构造器属性（constructor property）。该属性实际上是一个指向用于创建该对象的构造器函数的引用。

【示例】继续之前的例子：

```
h2.constructor;
function Hero(name){
    this.name = name;
}
```

由于构造器属性所引用的是一个函数，因此我们也可以利用它来创建一个其他新对象。例如像下面这样。

```
var h3 = new h2.constructor('Rafaello');
h3.name;                                    //"Rafaello"
```

另外，如果对象是通过对象文本标识法所创建的， 那么实际上它就是由内建构造器 Object()函数所创建的。

```
var o = {};
o.constructor;
function Object(){ [native code] }
typeof o.constructor;                       //"function"
```

E.1.10 instanceof 操作符

使用 instanceof 操作符可以测试一个对象是不是由某个指定的构造器函数所创建的。例如：

```
function Hero(){}
var h = new Hero();
var o = {};
h instanceof Hero;                    //true
h instanceof Object;                  //true
o instanceof Object;                  //true
```

注意，这里的函数名后面没有加括号（即不是 h instanceof Hero()），因为这里不是函数调用，所以只需要像使用其他变量一样，引用该函数的名字即可。

E.1.11 返回对象的函数

除了使用 new 操作符调用构造器函数以外，我们也可以抛开 new 操作符，只用一般函数来创建对象。这就需要一个能执行某些预备工作，并以对象为返回值的函数。

【示例 1】下面就有一个用于产生对象的简单函数 factory()。

```
function factory(name) {
    return {
        name: name
    };
}
```

然后，调用 factory()来生成对象。

```
var o = factory('one');
```

```
o.name;                                    //"one"
o.constructor;                             //function Object(){ [native code] }
```

实际上，构造器函数也是可以返回对象的，只不过在 this 值的使用上会有所不同。这意味着我们需要修改构造器函数的默认行为。

【示例 2】下面示例是构造器的一般用法。

```
function C() {
    this.a = 1;
}
var c = new C();
c.a;                                       //1
```

但现在要考虑的是这种用法：

```
function C2() {
    this.a = 1;
        return {b: 2};
    }
}
var c2 = new C2();
typeof c2.a;                               //"undefined"
c2.b;                                      //2
```

这里构造器返回的不再是包含属性 a 的 this 对象，而是另一个包含属性 b 的对象。但这也只有在函数的返回值是一个对象时才会发生，而当我们企图返回的是一个非对象类型时，该构造器将会照常返回 this。

关于对象在构造器函数内部是如何创建出来的，您可以设想在函数开头处存在一个叫作 this 的变量，这个变量会在函数结束时被返回，就像这样：

```
function C() {
// var this = {}; //pseudo code, you can't do this
this.a = 1;
// return this;
```

```
}
```

E.1.12 传递对象

当我们拷贝某个对象或者将它传递给某个函数时，往往传递的都是该对象的引用。注意，return 语句中使用的是大括号，也就是说{b:2}是一个独立的对象。因此在引用上所做的任何改动，实际上都会影响它所引用的原对象。

【示例】在下面示例中，对象是如何赋值给另一个变量的，并且，如果我们对该变量做一些改变操作的话，原对象也会跟着被改变。

```
var original = {howmany: 1};
var mycopy = original;
mycopy.howmany;                    //1
mycopy.howmany = 100;              //100
original.howmany;                  //100
```

同样，将对象传递给函数的情况也是如此：

```
var original = {howmany: 100};
var nullify = function(o) {o.howmany = 0;}
nullify(original);
original.howmany;                  //0
```

E.1.13 比较对象

当我们对对象进行比较操作时，当且仅当两个引用指向同一个对象时，结果为 true。如果是不同的对象，即使它们碰巧拥有相同的属性和方法，比较操作也会返回 false。

【示例】在下面示例中，创建两个看上去完全相同的对象。

```
var fido = {breed: 'dog'};
var benji = {breed: 'dog'};
```

然后，对它们进行比较，操作将会返回 false。

```
benji === fido;                    //false
benji == fido;                     //false
```

可以新建一个变量 mydog，并将其中一个对象赋值给它。这样一来 mydog 实际上就指向了这个变量。

```
var mydog = benji;
```

在这种情况下，mydog 与 benji 所指向的对象是相同的。也就是说，改变 mydog 的属性就等同于改变 benji），比较操作就会返回 true。

```
mydog === benji;                   //true
```

由于 fido 是一个与 mydog 不同的对象，所以它与 mydog 的比较结果仍为 false。

```
mydog === fido;                    //false
```

E.1.14 Webkit 控制台中的对象

Webkit 控制台为我们提供了一个叫作 console 的对象和一系列的方法，例如 console.log()和 console.error()。通过这些函数，我们可以在控制台中显示想要查看的值。

console.log()既可以在我们想进行某种快速测试时提供一些便利，也可以在我们处理某些真实脚本时记录一些中间调试信息。

【示例】在下面示例中，演示如何在循环中使用该函数。

```
for(var i = 0; i < 5; i++) {
    console.log(i);
}
```

E.2 内建对象

无论是函数，还是构造器函数，最后都是对象。下面一一介绍 JavaScript 内建的各种类型的构造器和内建对象。内建对象大致上可以分为 3 大类。

- 数据封装类对象，包括 Object、Array、Boolean、Number 和 String。这些对象代表着 JavaScript 中不同的数据类型，并且都拥有各自不同的 typeof 返回值，以及 undefined 和 null 状态。
- 工具类对象，包括 Math、Date、RegExp 等用于提供便利的对象。
- 错误类对象，包括一般性错误对象，以及其他各种更特殊的错误类对象。它们可以在某些异常发生时帮助我们纠正程序工作状态。

E.2.1 Object

Object 是 JavaScript 中所有对象的父级对象，这意味着我们创建的所有对象都继承于此。为了新建一个空对象，我们既可以用对象文本标识法，也可以调用 Object()构造器函数。

【示例】下面这两行代码的执行结果是等价的。

```
var o = {};
var o = new Object();
```

我们之前提到过，所谓的“空”对象，实际上并非是完全无用的，它还是包含了一些继承来的方法和属性的。这里的“空”对象指的是像{}这种除继承来的属性之外，不含任何自身属性的对象。

例如，“空”对象 o 中的部分属性。

- constructor：返回构造器函数的引用。
- toString()：返回对象的描述字符串。
- valueOf()：返回对象的单值描述信息，通常返回的就是对象本身。

下面创建一个对象：

```
var o = new Object();
```

然后调用 toString()方法，返回该对象的描述字符串：

```
o.toString();                                    //"[object Object]"
```

toString()方法会在某些需要用字符串来表示对象的时候被 JavaScript 内部调用。例如，alert()的工作就需要用到这样的字符串。所以，如果我们将对象传递给了一个 alert()函数，toString()方法就会在后台被调用，也就是说，下面两行代码的执行结果是相同的。

```
alert(o);
alert(o.toString());
```

另外，字符串连接操作也会使用字符串描述文本，如果我们将某个对象与字符串进行连接，那么该对象就先调用自身的 toString()方法。

```
"An object: " + o;
"An object: [object Object]"
```

valueOf()方法也是为所有对象共有的一个方法。对于简单对象（即以 Object()为构造器的对象）来说，valueOf()方法所返回的就是对象自己。

```
o.valueOf() === o;                               //true
```

总之，创建对象时既可以用 var o = {}的形式（即执行对象文本标识法，推荐这种方法），也可以用 var o = new Object()；. 无论是多复杂的对象，它都是继承自 Object 对象的，并且拥有其所有的方法（如 toString()）和属性（如 constructor）。

E.2.2 Array

Array()是一个用来构建数组的内建构造器函数，例如：

```
var a = new Array();
```

这与下面的数组文本标识法是等效的：

```
var a = [];
```

无论数组是以什么方式创建的，我们都能照常往里添加元素。

```
a[0] = 1;
a[1] = 2;
a;                                          //[1, 2]
```

当我们使用 Array()构造器创建新数组时，也可以通过传值的方式为其设定元素。

```
var a = new Array(1,2,3,'four');
a;                                          // [1, 2, 3, "four"]
```

但是如果我们传递给该构造器的是一个单独数字，就会出现一种异常情况，即该数值会被认为是数组的长度。

```
var a2 = new Array(5);
a2;                                          //[undefined x 5]
```

既然数组是由构造器来创建的，那么这是否意味着数组实际上是一个对象。我们可以用 typeof 操作符来验证一下。

```
typeof [1, 2, 3];                           //"object"
```

由于数组也是对象，那么就说明它也继承了 Object 的所有方法和属性。

```
var a = [1, 2, 3, 'four'];
a.toString();                               //"1,2,3,four"
a.valueOf();                                //[1, 2, 3, "four"]
a.constructor;                          //function Array(){ [native code] }
```

尽管数组也是一种对象，但还是有一些特殊之处，原因有以下几个封面。

- 数组的属性名是从 0 开始递增，并自动生成数值。
- 数组拥有一个用于记录元素数量的 length 属性。
- 数组在父级对象的基础上扩展了更多额外的内建方法。

下面来实际验证一下对象与数组之间的区别，让我们从创建空对象 o 和空数组 a 开始。

```
var a = [], o = {};
```

首先，定义数组对象时会自动生成一个 length 属性。而这在一般对象中是没有的。

```
a.length;                                    //0
typeof o.length;                             //"undefined"
```

在为数组和对象添加以数字或非数字为键名的属性操作上，两者间并没有多大的区别。

```
a[0] = 1;
o[0] = 1;
a.prop = 2;
o.prop = 2;
```

length 属性通常会随着数字键名属性的数量而更新，而忽略非数字键名属性。

```
a.length;                                    //1
```

我们也可以手动设置 length 属性。如果设置的值大于当前数组中元素数量，剩下的那部分会被自动创建（值为 undefined）的空元素所填充。

```
a.length = 5;                                //5
a;                                           //[1, undefined x 4]
```

而如果我们设置的 length 值小于当前元素数，多出的那部分元素将会被移除。

```
a.length = 2;                                //2
a;                                           // [1, undefined x 1]
```

除了从父级对象那里继承的方法以外，数组对象中还有一些更为有用的方法，例如，sort()、join()和 slice()等。

下面，我们将通过一个数组来试验一下这些方法。

```
var a = [3, 5, 1, 7, 'test'];
```

push()方法会在数组的末端添加一个新元素，而 pop()方法则会移除最后一个元素，也就是说 a.push("new")就相当于 a[a.length] = "new"，而 a.pop()则与 a.length--的结果相同。

另外，push()返回的是改变后的数组长度，而 pop 所返回的则是被移除的元素。

```
a.push('new');                               //6
```

```
a;                                  //[3, 5, 1, 7, "test", "new"]
a.pop();                            //"new"
a;                                  //[3, 5, 1, 7, "test"]
```

而 sort()方法则是用于给数组排序的，它会返回排序后的数组，在下面的示例中，排序完成后，a 和 b 所指向的数组是相同的。

```
var b = a.sort();                   //b;
[1, 3, 5, 7, "test"]                //
a === b;                            //true
```

join()方法会返回一个由目标数组中所有元素值用逗号连接而成的字符串，我们可以通过该方法的参数来设定这些元素之间用什么字符（串）连接。例如：

```
a.join(' is not ');                 //"1 is not 3 is not 5 is not 7 is not test"
```

slice()方法会在不修改目标数组的情况下返回其中的某个片段，该片段的首尾索引位置将由 slice()的头两个参数来指定（都以 0 为基数）。

```
b = a.slice(1, 3);                  //[3, 5]
b = a.slice(0, 1);                  //[1]
b = a.slice(0, 2);                  //[1, 3]
```

所有的截取完成之后，原数组的状态不变。

```
a;                                  //[1, 3, 5, 7, "test"]
```

splice()则是会修改目标数组的。它会移除并返回指定切片，并且在可选情况下，它还会用指定的新元素来填补被切除的空缺。该方法的头两个参数所指定的是要移除切片的首尾索引位置，其他参数则用于填补的新元素值。

```
b = a.splice(1, 2, 100, 101, 102);  //[3, 5]
a;                                  //[1, 100, 101, 102, 7, "test"]
```

当然，用于填补空缺的新元素是可选的，我们也可以直接跳过。

```
a.splice(1, 3);                     //[100, 101, 102]
a;                                  //[1, 7, "test"]
```

E.2.3 Function

函数是一种特殊的数据类型，实际上它也是一种对象。函数对象的内建构造器是 Function()，你可以将它作为创建函数的一种备选方式。

下面展示了三种定义函数的方式。

```
function sum(a, b) { // function declaration
    return a + b;
}
sum(1, 2);                                    //3
var sum = function(a, b) { // function expression
    `return a + b;
};
sum(1, 2)                                     //3
var sum = new Function('a', 'b', 'return a + b;');
sum(1, 2)                                     //3
```

如果我们使用的是 Function()构造器的话，就必须要通过参数传递的方式来设定函数的参数名（通常是用字符串）以及函数体中的代码（也是用字符串）。JavaScript 引擎自会对这些源代码进行解析，并随即创建新函数，这样一来，就会带来与 eval()相似的缺点。因此我们要尽量避免使用 Function()构造器来定义函数。

如果一定想用 Function()构造器来创建一个拥有许多参数的函数，可了解一点：这些参数可以是一个由逗号分隔而成的单列表，所以，下面例子中的这些函数定义是相同的。

```
var first = new Function(
    'a, b, c, d',
    'return arguments;'
);
first(1,2,3,4);                               //[1, 2, 3, 4]
```

```
var second = new Function(
    'a, b, c',
    'd',
    'return arguments;'
);
second(1,2,3,4);                    //[1, 2, 3, 4]
var third = new Function(
    'a',
    'b',
    'c',
    'd',
    'return arguments;'
);
third(1,2,3,4);                     //[1, 2, 3, 4]
```

【提示】

请尽量避免使用 Function()构造器。因为它与 eval()和 setTimeout()一样，始终会以字符串的形式通过 JavaScript 的代码检查。

1. 函数对象的属性

- constructor 的属性

与其他对象相同的是，函数对象中也含有名为 constructor 的属性，其引用的就是 Function()这个构造器函数。

```
function myfunc(a){
    return a;
}
myfunc.constructor;          //function Function(){[native code]}
```

- length 属性

函数对象中也有一个 length 属性，用于记录该函数声明时所决定的参数数量。

```
function myfunc(a, b, c){
    return true;
}
myfunc.length;                          //3
```

- prototype 属性

prototype 属性是 JavaScript 中使用得最为广泛的函数属性。

- 每个函数的 prototype 属性中都指向了一个对象。
- 它只有在该函数是构造器时才会发挥作用。
- 该函数创建的所有对象都会持有一个该 prototype 属性的引用，并可以将其当作自身的属性来使用。

【示例】下面示例演示一下 prototype 属性的使用。先创建一个简单对象，对象中只有一个 name 属性和一个 say()方法。

```
var ninja = {
    name: 'Ninja',
    say: function(){
        return 'I am a ' + this.name;
    }
};
```

这方面的验证很简单，因为任何一个新建函数（即使这个函数没有函数体）中都会有一个 prototype 属性，而该属性会指向一个新对象。

```
function F(){}
typeof F.prototype;                     //"object"
```

如果我们现在对该 prototype 属性进行修改，就会发生一些有趣的变化：当前默认的空对象被直接替换成了其他对象。下面我们将变量 ninja 赋值给这个 prototype：

```
F.prototype = ninja;
```

现在，如果我们将 F()当作一个构造器函数来创建对象 baby_ninja，那么新对象 baby_ninja 就会拥有对 F.prototype 属性（也就是 ninja）的访问权。

```
var baby_ninja = new F();
```

```
baby_ninja.name;                    //"Ninja"
baby_ninja.say();                   //"I am a Ninja"
```

2. 函数对象的方法

所有的函数对象都是继承自顶级父对象 Object 的，因此它也拥有 Object 对象的方法。例如，toString()。当我们对一个函数调用 toString()方法时，所得到的就是该函数的源代码。

```
function myfunc(a, b, c) {
    return a + b + c;
}
myfunc.toString();                  //"function myfunc(a, b, c) {
                                    //"return a + b + c;
                                    //}"
```

但如果我们想用这种方法来查看那些内建函数的源码的话，就只会得到一个毫无用处的字符串[native code]。

```
parseInt.toString();                //"function parseInt() {[native code]}"
```

我们可以用 toString()函数来区分本地方法和自定义方法。toString()方法的行为与运行环境有关，浏览器之间也会有差异，比如空格和空行的多少。

3. call()与 apply()

在 JavaScript 中，每个函数都有 call()和 apply()两个方法，可以用它们来触发函数，并指定相关的调用参数。此外，这两个方法还有另外一个功能，它可以让一个对象去“借用”另一个对象的方法，并为己所用。这也是一种非常简单而实用的代码重用。

【示例】下面示例定义一个 some_obj 对象，该对象中有一个 say()方法。

```
var some_obj = {
    name: 'Ninja',
    say: function(who){
```

```
            return 'Haya ' + who + ', I am a ' + this.name;
        }
    };
```

这样一来，我们就可以调用该对象的 say()方法，并在其中使用 this.name 来访问其 name 属性了。

```
some_obj.say('Dude');                    //"Haya Dude, I am a Ninja"
```

下面再创建一个 my_obj 对象，它只有一个 name 属性。

```
var my_obj = {name: 'Scripting guru'};
```

显然，some_obj 的 say()方法也适用于 my_obj，因此希望将该方法当作 my_obj 自身的方法来调用。在这种情况下，我们就可以试试 say()函数中的对象方法 call()。

```
some_obj.say.call(my_obj, 'Dude');   //"Haya Dude, I am a Scripting guru"
```

由于我们在调用 say()函数的对象方法 call()时传递了两个参数：对象 my_obj 和字符串"Dude"。这样一来，当 say()被调用时，其中的 this 就被自动设置成了 my_obj 对象的引用。因而 this.name 返回的不再是"Ninja"，而是"Scripting guru"了。

如果我们调用 call 方法时需要传递更多的参数，可以在后面依次加入它们。

```
some_obj.someMethod.call(my_obj, 'a', 'b', 'c');
```

如果我们没有将对象传递给 call()的首参数，或者传递给它的是 null，它的调用对象将会被默认为全局对象。

apply()的工作方式与 call()基本相同，唯一的不同之处在于参数的传递形式，这里目标函数所需要的参数都是通过一个数组来传递。所以，下面两行代码的作用是等效的。

```
some_obj.someMethod.apply(my_obj, ['a', 'b', 'c']);
some_obj.someMethod.call(my_obj, 'a', 'b', 'c');
```

因而，对于之前的示例，我们也可以这样写。

```
some_obj.say.apply(my_obj, ['Dude']);
//"Haya Dude, I am a Scripting guru"
```

4. arguments 对象

在一个函数中通过 arguments 来访问传递给该函数所需的全部参数。例如：

```
function f() {
    return arguments;
}
f(1,2,3);                                    //[1, 2, 3]
```

尽管 arguments 看上去像是一个数组，但它实际上是一个类似数组的对象。它和数组相似。是因为其中也包含了索引元素和 length 属性。但相似之处也就到此为止了，因为 arguments 不提供一些像 sort()、slice()这样的数组方法。

但我们可以把 arguments 转换成数组，这样就可以对它使用各种各样的数组方法了。

【示例】在下面示例中，学习 call()方法。

```
function f(){
    var args = [].slice.call(arguments);
    return args.reverse();
}
f(1,2,3,4);                                  //[4,3,2,1]
```

这里新建一个空数组[]，再使用它的 slice 属性。当然，也可以通过 Array.prototype.slice 来调用同一个函数。

5. 推断对象类型

arguments 对象跟数组之间的不同之处。但二者之间具体应该如何区分呢？既然数组的 typeof 返回值也为"object"，那么要如何区分对象与数组呢？

我们可以使用 Object 对象的 toString()方法。这个方法会返回所创建对象的内部类名。

```
Object.prototype.toString.call({});          //"[object Object]"
Object.prototype.toString.call([]);          //"[object Array]"
```

在这里，toString()方法必须要来自于 Object 构造器的 prototype 属性。直接调用 Array 的 toString()方法是不行的，因为在 Array 对象中。这个方法已经出于其他目的被重写了。

```
 [1, 2, 3].toString();                       //"1,2,3"
```

也可以写为：

```
Array.prototype.toString.call([1, 2, 3]);    //"1,2,3"
```

也可以单独为 Object.prototype.toString 设置一个引用变量，以便让代码显得更简短一些。

```
var toStr = Object.prototype.toString;
```

如果用这个方法调用 arguments，很快就能发现它与 Array 之间的区别。

```
(function () {
    return toStr.call(arguments);
}());                                         "[object Arguments]"
```

同样，这个方法也适用于 DOM 元素。

```
toStr.call(document.body);                    //"[object
HTMLBodyElement]"
```

E.2.4 Boolean

JavaScript 基本数据类型主要包括 Boolean、Number、String 等。Boolean()构造器用法如下。

```
var b = new Boolean();
```

在这里最重要的一点是，这里所新创建的 b 是一个对象，而不是一个基本数据类型的布尔值。如果想将 b 转换成基本数据类型的布尔值，可以调用它的 valueOf()方法（继承自 Object 对象）。

```
var b = new Boolean();
```

```
typeof b;                                    //"object"
typeof b.valueOf();                          //"boolean"
b.valueOf();                                 //false
```

总体而言，用 Boolean()构造器所创建的对象并没有多少实用性，因为它并没有提供来自父级对象以外的任何方法和属性。

不使用 new 操作符而单独作为一般函数使用时，Boolean()可以将一些非布尔值转换为布尔值（其效果相当于进行两次取反操作：!!value）。

```
Boolean("test");                             //true
Boolean("");                                 //false
Boolean({});                                 //true
```

在 JavaScript 中，除了六种假值：空字符串""、0、false、null、undefined、NaN，其他所有的都属于真值，其中也包括所有的对象。这就意味着所有由 new Boolean()语句创建的布尔对象都等于 true，因为它们都是对象。

```
Boolean(new Boolean(false) );                //true
```

这种情况确实很容易让人混淆。而且考虑到 Boolean 对象中并没有很特别的方法，建议还是一直使用基本类型来表示布尔值比较妥当。

E.2.5 Number

Number()函数的用法与 Boolean()类似。

- 在被当作构造器函数时（即用于 new 操作符），它会创建一个对象。
- 在被当作一般函数时，它会试图将任何值转换为数字，这与 parseInt()或 parseFloat()起到的作用基本相同。

```
var n = Number('12.12');
n;                                           //12.12
typeof n;                                    //"number"
var n = new Number('12.12');
```

```
typeof n;                                    //"object"
```

由于函数本身也是对象，所以会拥有一些属性。在 Number()函数中，有一些内置属性是值得我们注意的（它们是不可更改的）。

```
Number.MAX_VALUE;                    //1.7976931348623157e+308
Number.MIN_VALUE;                    //5e-324
Number.POSITIVE_INFINITY;            //Infinity
Number.NEGATIVE_INFINITY;            //-Infinity
Number.NaN;                          //NaN
```

此外，Number 对象中还提供了三个方法，它们分别是：toFixed()、toPrecision()和 toExponential()。

```
var n = new Number(123.456);
n.toFixed(1);                                //"123.5"
```

注意，可以在事先未创建 Number 对象的情况下使用这些方法。在这些例子中，Number 对象均在后台完成创建和销毁。

```
(12345).to Exponential();                    //"1.2345e+4"
```

与所有的对象一样，Number 对象也提供了自己的 toString()方法。但值得注意的是，该对象的 toString()方法有一个可选的 radix 参数（它的默认值是 10）。

```
var n = new Number(255);
n.toString();                                //"255"
n.toString(10);                              //"255"
n.toString(16);                              //"ff"
(3).toString(2);                             //"11"
(3).toString(10);                            //"3"
```

E.2.6 String

我们可以通过 String()构造器函数来新建 String 对象。该对象为我们提

供了一系列用于文本操作的方法，但最好还是使用基本的字符串类型。

【示例】下面示例演示 String 对象与基本的字符串类型之间有什么区别。

```
var primitive = 'Hello';
typeof primitive;                    //"string"
var obj = new String('world');
typeof obj;                          //"object"
```

String 对象实际上就像是一个字符数组，其中也包括用于每个字符的索引属性（虽然这个特性在 ES5 开始才引入，但早已被各大浏览器支持，除了早期版本的 IE），以及整体的 length 属性。

```
obj[0];                              //"w"
obj[4];                              //"d"
obj.length;                          //5
```

如果想获得 String 对象的基本类型值，可以调用该对象的 valueOf()或 toString()方法（都继承自 Object 对象）。不过可能很少有机会这么做，因为在很多场景中，String 对象都会被自动转换为基本类型的字符串。

```
obj.valueOf();                       //"world"
obj.toString();                      //"world"
obj + "";                            //"world"
```

而基本类型的字符串就不是对象了，因此它们不含有任何属性和方法。但 JavaScript 还是为我们提供了一些将基本字符串类型转换为 String 对象的语法（就像我们之前转换基本类型的数字一样）。

【示例】在下面示例中，当我们将一个基本字符串当作对象来使用时，后台就会相应的创建 String 对象，在调用完之后又把 String 对象给立即销毁。

```
"potato".length;                     //6
"tomato"[0];                         //"t"
"potato"["potatoes".length - 1];     //"s"
```

最后我们再来看一个说明基本字符串与 String 对象之间区别的例子：

当它们被转换成布尔值时，尽管空字符串属于 falsy 值，但所有的 String 对象都是 truthy 值（因为所有的对象都是 truthy 值）。

```
Boolean("");                                    //false
Boolean(new String(""));                        //true
```

与 Number()和 Boolean()类似，如果我们不通过 new 操作符来调用 String()，它就会试图将其参数转换为一个基本字符串。

```
String(1);                                      //"1"
```

如果其参数是一个对象的话，这就等于调用该对象的 toString()方法。

```
String({p: 1});                                 //"[object Object]"
String([1,2,3]);                                //"1,2,3"
String([1, 2, 3]) === [1, 2, 3].toString();     //true
```

下面演示一下部分 String 对象方法的调用。

首先，从新建 String 对象开始。

```
var s = new String("Couch potato");
```

接下来是用于字符串大小写转换的方法，toUpperCase() 与 toLowerCase()。

```
s.toUpperCase();                                //"COUCH POTATO"
s.toLowerCase();                                //"couch potato"
```

charAt()方法返回的是我们指定位置的字符，这与中括号的作用相当（字符串本身就是一个字符数组）。

```
s.charAt(0);                                    //"C"
s[0];                                           //"C"
```

如果我们传递给 charAt()方法的位置并不存在，它就会返回一个空字符串。

```
s.charAt(101);                                  //""
```

indexOf()方法可以帮助我们实现字符串内部搜索，该方法在遇到匹配字符时会返回第一次匹配位置的索引值。由于该索引值是从 0 开始计数的，所以字符串"Couch"中第二个字符"o"的索引值为 1。

```
s.indexOf('o');                                 //1
```

另外，我们也可以通过可选参数指定搜索开始的位置（以索引值的形式）。例如下面所找到的就是字符串中的第二个"o"，因为我们指定的搜索是从索引 2 处开始的。

```
s.indexOf('o', 2);                    //7
```

如果我们想让搜索从字符串的末端开始，可以调用 lastIndexOf()方法（但返回的索引值仍然是从前到后计数的）。

```
s.lastIndexOf('o');                   //11
```

当然，上述方法的搜索对象不仅仅局限于字符，也可以用于字符串搜索。并且搜索是区分大小写的。

```
s.indexOf('Couch');                   //0
```

如果方法找不到匹配对象，返回的位置索引值就为-1。

```
s.indexOf('couch');                   //-1
```

如果我们想进行一次大小写无关的搜索，可以将字符串转换为小写后再执行搜索。

```
s.toLowerCase().indexOf('couch');     //0
```

如果相关的搜索方法返回的索引值是 0，就说明字符串的匹配部分是从 0 处开始的。这有可能会给 if 语句的使用带来某些混淆因素，当我们像下面这样使用 if 语句，就会将索引值 0 隐式地转换为布尔值 false，虽然这种写法没有什么语法错误，但在逻辑上却完全错了。

```
if (s.indexOf('Couch')) {...}
```

正确的做法是：当我们用 if 语句检测一个字符串中是否包含另一个字符串时，可以用数字-1 来做 indexOf()结果的比较参照。

```
if (s.indexOf('Couch') !== -1) {...}
```

接下来，我们要介绍的是 slice()和 substring()，这两个方法都可以用于返回目标字符串中指定的区间。

```
s.slice(1, 5);                        //"ouch"
s.substring(1, 5);                    //"ouch"
```

注意，这两个方法的第二个参数所指定的都是区间的末端位置，而不是该区间的长度。这两个方法的不同之处在于对负值参数的处理方式，

substring()方法会将负值视为 0，而 slice()方法则会将它与字符串的长度相加。因此，如果我们传给它们的参数是(1,-1)的话，它们的实际情况分别是 substring(1, 0)和 slice(1,s.length-1)。

```
s.slice(1, -1);                    //"ouch potat"
s.substring(1, -1);                //"C"
```

还有一个方法叫 substr()，但由于它不在 JavaScript 的标准中，所以应该尽量用 substring()去代替它。split()方法可以根据我们所传递的分割字符串，将目标字符串分割成一个数组。例如：

```
s.split(" ");                      // ["Couch", "potato"]
```

split()是 join()的反操作，后者则会将一个数组合并成一个字符串。例如：

```
s.split(' ').join(' ');            //"Couch potato"
```

concat()方法通常用于合并字符串，它的功能与基本字符串类型的+操作符类似。

```
s.concat("es");                    //
"Couch potatoes"
```

注意，到目前为止，我们所讨论的方法返回的都是一个新的基本字符串，它们所做的任何修改都不会改动源字符串。所有的方法调用都不会影响原始字符串的值。

```
s.valueOf();                       //"Couch potato"
```

通常情况下，我们会用 indexOf()和 lastIndexOf()方法进行字符串内搜索，但除此之外还有一些功能更为强大的方法（如 search()、match()、replace()等），它们可以以正则表达式为参数来执行搜索任务。

E.2.7 Math

Math 与我们之前所见过的其他全局内建对象是有些区别的。Math 对象不是函数对象，所以我们不能对它调用 new 操作符，以创建别的对象。实际上，Math 只是一个包含一系列方法和属性、用于数学计算的全局内建

对象。

Math 的属性都是一些不可修改的常数，因此它们都以名字大写的方式来表示自己与一般属性变量的不同（这类似于 Number()构造器的常数属性）。下面我们一起看看这些属性。

- 数字常数 π

```
Math.PI;                    //3.141592653589793
```

- 2 的平方根

```
Math.SQRT2;                 //1.4142135623730951
```

- 欧拉常数 e

```
Math.E;                     //2.718281828459045
```

- 2 的自然对数

```
Math.LN2;                   //0.6931471805599453
```

- 10 的自然对数

```
Math.LN10;                  //2.302585092994046
```

接下来，我们再来看看 Math 对象所提供的一些方法，完整的方法列表请参考 JavaScript 参考手册。

- 生成随机数

```
Math.random();              //0.3649461670235814
```

random()所返回的是 0 到 1 之间的某个数，所以如果我们想要获得 0 到 100 之间的某个数的话，就可以这样：

```
100 * Math.random();
```

如果我们需要获取的是某 max 和 min 之间的值，可以通过一个公式((max - min) *Math.random()) + min 来获取，例如，我们想获取的是 2 到 10 之间的某个数，就可以这样：

```
8 * Math.random() + 2;      //9.175650496668485
```

如果这里需要的是一个整数的话，可以调用以下取整方法。

- floor()：取小于或等于指定值的最大整数。
- ceil()：取大于或等于指定值的最小整数。
- round()：取最靠近指定值的整数。

例如，下面的执行结果不是 0 就是 1。

```
Math.round(Math.random());
```

如果想获得一个数字集合中的最大值或最小值，则可以调用 max()和 min()方法。所以，当我们在一个表单中需要一个合法的月份值时，可以用下面的方式来确保相关的数据能正常工作：

```
Math.min(Math.max(1, input), 12);
```

除此之外，Math 对象还提供了一些用于执行数学计算的方法，这些计算是我们不需要去专门设计即可使用的。这意味着当我们想要执行指数运算时只需要调用 pow()方法即可，而求平方根时只需要调用 sqrt()，另外还包括所有的三角函数计算—sin()、cos()、atan()等。例如，求 2 的 8 次方。

```
Math.pow(2, 8);                                    //256
```

求 9 的平方根：

```
Math.sqrt(9);                                      //3
```

E.2.8 Date

Date()是用于创建 Date 对象的构造器函数，我们在用它创建对象时可以传递以下几种参数。

- 无参数（默认为当天的日期）。
- 一个用于表现日期的字符串。
- 分开传递的日、月、时间等值。
- 一个 timestamp 值。

【示例】下面是一个表示当天日期和时间的对象示例。

```
new Date();                    //Wed Feb 27 2018 23:49:28 GMT-0800 (PST)
```

UNIX 时间，或称 POSIX 时间，是 UNIX 或类 UNIX 系统使用的时间表示方式：从协调世界时 1970 年 1 月 1 日 0 时 0 分 0 秒起至现在的总秒数，不包括闰秒。

【示例】用字符串初始化 Date 对象的示例，请注意它们各自不同的格

式以及所指定的时间。

```
new Date('2015 11 12');      //Thu Nov 12 2015 00:00:00 GMT-0800 (PST)
new Date('1 1 2016');        //Fri Jan 01 2016 00:00:00 GMT-0800 (PST)
new Date('1 mar 2016 5:30');
//Tue Mar 01 2016 05:30:00 GMT-0800 (PST)
```

Date 构造器可以接受各种不同格式的字符串日期输入表示法，但如要定义一个精确的日期，例如将用户输入直接传递给 Date 构造器，这样做显然不够可靠。更好的选择是向 Date()构造器传递一些具体的数值，其中包括：

- 年份；
- 月份：从 0（1 月）到 11（12 月）；
- 日期：从 1 到 31；
- 时数：从 0 到 23；
- 分钟：从 0 到 59；
- 秒钟：从 0 到 59；
- 毫秒数：从 0 到 999。

【示例】下面是一个具体示例。

如果我们传递所有参数：

```
new Date(2015, 0, 1, 17, 05, 03, 120);
//Tue Jan 01 2015 17:05:03 GMT-0800 (PST)
```

如果只传递日期和时钟值：

```
new Date(2015, 0, 1, 17);   //Tue Jan 01 2015 17:00:00 GMT-0800 (PST)
```

注意，由于月份是从 0 开始的，所以这里的 1 指的是 2 月：

```
new Date(2016, 1, 28);      //Sun Feb 28 2016 00:00:00 GMT-0800 (PST)
```

如果我们所传递的值越过了被允许的范围，Date 对象会自行启动“溢出式”前进处理。例如，由于 2016 年 2 月不存在 30 日这一天，所以它会自动解释为该年的 3 月 1 日（2016 年为闰年）。

```
new Date(2016, 1, 29);      //Mon Feb 29 2016 00:00:00 GMT-0800 (PST)
new Date(2016, 1, 30);      //Tue Mar 01 2016 00:00:00 GMT-0800 (PST)
```

类似地，如果我们传递的是 12 月 32 日，就会被自动解释为来年的 1 月

1 日：

```
new Date(2012, 11, 31);   //Mon Dec 31 2012 00:00:00 GMT-0800 (PST)
new Date(2012, 11, 32);   //Tue Jan 01 2013 00:00:00 GMT-0800 (PST)
```

最后，我们也可以通过 timestamp 的方式来初始化一个 Date 对象（这是一个以毫秒为单位的 UNIX 纪元方式，开始于 1970 年 1 月 1 日）。

```
new Date(1357027200000); //Tue Jan 01 2013 00:00:00 GMT-0800 (PST)
```

如果我们在调用 Date()时没有使用 new 操作符，那么无论是否传递了参数，所得字符串的内容始终都将是当前的日期和时间（就像下面示例所运行的那样）。

```
Date();                    //Wed Feb 27 2013 23:51:46 GMT-0800 (PST)
Date(1, 2, 3, "it doesn't matter");
//Wed Feb 27 2013 23:51:52 GMT-0800 (PST)
typeof Date();                          //"string"
typeof new Date();                      //"object"
```

Date 对象的方法

一旦创建了 Date 对象，就可以调用该对象中的许多方法。其中使用最多的都是一些名为 set*()或 get*()的方法，例如 getMonth()、setMonth()、getHours()、setHours()等。

【示例】下面来看一些具体的示例。

首先，新建一个 Date 对象。

```
var d = new Date(2015, 1, 1);
d.toString();              //Sun Feb 01 2015 00:00:00 GMT-0800 (PST)
```

然后，将其月份设置成 3 月（记住，月份数是从 0 开始的）。

```
d.setMonth(2);                          //1425196800000
d.toString();              //Sun Mar 01 2015 00:00:00 GMT-0800 (PST)
```

接着，读取月份数。

```
d.getMonth();                           //2
```

除了这些实例方法以外，Date()函数/对象中还有另外两个方法（ES5 中又新增了一个）。这两个属性不需要在实例化情况下使用，工作方式与 Math

的方法基本相同。在基于 class 概念的程序设计语言中，它们往往被称之为“静态”方法，因为它们的调用不需要依托对象实例。

例如，Date.parse()方法会将其所接收的字符串转换成相应的 timestamp 格式，并返回。

```
Date.parse('Jan 11, 2018');                    //1515657600000
```

而 Date.UTC()方法则可以接受包括年份、月份、日期等在内的所有参数，并以此产生一个相应的、符合格林尼治时标准的 timestamp 值。

```
Date.UTC(2018, 0, 11);                         //1515628800000
```

由于用 Date 创建对象时可以接受一个 timestamp 参数，因此我们也可以直接将 Date.UTC()的结果传递给该构造器。

【示例】在下面的示例中，我们演示了如何在新建 Date 对象的过程中，将 UTC()返回的格林尼治时间转换为本地时间。

```
new Date(Date.UTC(2018, 0, 11));
//Wed Jan 10 2018 16:00:00 GMT-0800 (PST)
new Date(2018, 0, 11);
//Thu Jan 11 2018 00:00:00 GMT-0800 (PST)
```

此外，ES5 还为 Date 构造器新增了 now()方法，以用于返回当前 timestamp。比起在 ES3 中对着一个 Date 对象调用 getTime()方法而言，这种新方法显然更为简洁。

```
Date.now();                                    //1362038353044
Date.now() === new Date().getTime();           //true
```

日期的内部表达形式就是一个整数类型的 timestamp，而它的其他表达形式只不过是这种内部形式的包装。这么一来，我们就很容易理解为什么 Date 对象的 valueOf()返回的是一个 timestamp 数据。

```
new Date().valueOf();                          //
1362418306432
```

而将 Date 转换为整型则只需要一个+号。

```
+ new Date();                                  //1362418318311
```

例子：计算生日下面，我们再来看最后一个关于 Date 对象的工作示例。

假如，我很好奇自己 2016 年的生日（6 月 20 日）是星期几，就可以这样：

```
var d = new Date(2016, 5, 20);
d.getDay();                                    //1
```

由于星期数是从 0（星期日）开始计数的，因此，1 应该代表了星期一。我们来验证一下。

```
d.toDateString();                              //"Mon Jun 20 2016"
```

接下来弄一个循环，看看从 2016 年到 3016 年有多少个 6 月 20 日是星期一，并查看一下这些日子在一周当中的分布情况。

首先，我们来初始化一个包含七个元素的数组，每个元素都分别对应着一周中的一天，以充当计数器。也就是说，在循环到 3016 年的过程中，我们将会根据执行情况递增相关的计数器：

```
var stats = [0,0,0,0,0,0,0];
```

接下来就是该循环的实现：

```
for (var i = 2016; i < 3016; i++) {
    stats[new Date(i, 5, 20).getDay()]++;
}
```

然后，看看结果：

```
stats;                                  //[140, 146, 140, 145, 142, 142, 145]
```

E.2.9 RegExp

正则表达式（regular expression）提供了一种强大的文本搜索和处理方式。对于正则表达式，不同的语言有着不同的实现（就像“方言”），JavaScript 所采用的是 Perl 5 的语法。另外，为简便起见，人们经常会将 regular expression 缩写成 regex 或者 regexp。

一个正则表达式通常由以下部分组成。

- 一个用于匹配的模式文本。
- 用 0 个或多个修饰符（也叫作标志）描述的匹配模式细节。

该匹配模式也可以是简单的全字符文本，但这种情况极少，而且此时我们多半会使用 indexOf()这样的方法，而很少会用到正则表达式。在大多数情况下，匹配模式往往都很复杂，也更难以理解。

在 JavaScript 中，我们通常会利用内建构造器 RegExp()来创建正则表达式对象，例如：

```
var re = new RegExp("j.*t");
```

另外，RegExp 对象还有一种更为简便的正则文本标记法（regex literal notation）。

```
var re = /j.*t/;
```

在上面的示例中，“j.*t”就是我们之前说的正则表达式模式。其具体含义是：“匹配任何以 j 开头、t 结尾的字符串，且这两个字符之间可以包含 1 个或多个字符。”其中的*号的意思就是“0 个或多个单元”，而这里的点号（.）所表示的是“任意字符”。当然，当我们向 RegExp 构造器传递该模式时，还必须将它放在一对引号中。

1 RegExp 对象的属性

以下是一个正则表达式对象所拥有的属性。

- global：如果该属性值为 false（这也是默认值），相关搜索在找到第一个匹配时就会停止。如果需要找出所有的匹配，将其设置为 true 即可。
- ignoreCase：设置大小写相关性，默认为 false。
- multiline：设置是否跨行搜索，默认为 false。
- lastIndex：搜索开始的索引位，默认值为 0。
- source：用于存储正则表达式匹配模式。

另外，除了 lastIndex 外，上面所有属性在对象创建之后就都不能再被修改了。而且，前三个属性是可以通过 regex 修饰符来表示的。当我们通过构造器来创建 regex 对象时，可以向构造器的第二参数传递下列字符中的任意组合。

- “g”代表 global。

- “i”代表 ignoreCase。
- “m”代表 multiline。

这些字符可以以任意顺序传递，只要它们被传递给了构造器，相应的修饰符就会被设置为 true。例如在下面的示例中，我们将所有的修饰符都设置成了 true。

```
var re = new RegExp('j.*t', 'gmi');
re.global;                                    //true
```

不过，这里的修饰符一旦被设置了就不能更改。

```
re.global = false;
re.global;                                    //true
```

另外，我们也可以通过文本方式来设置这种 regex 的修饰符，只需将它们加在斜线后面。

```
var re = /j.*t/ig;
re.global;                                    //true
```

2 RegExp 对象的方法

RegExp 对象中有两种可用于查找匹配内容的方法：test()和 exec()。这两个方法的参数都是一个字符串，但 test()方法返回的是一个布尔值（找到匹配内容时为 true，否则就为 false），而 exec()返回的则是一个由匹配到的字符串组成的数组。显然，exec()能做的工作更多，而 test()只有在我们不需要匹配的具体内容时才会有所用处。人们通常会用正则表达式来执行某些验证操作，在这种情况下往往使用 test()就足够了。

【示例】下面的表达式是不匹配的，因为目标中是大写的 J。

```
/j.*t/.test("Javascript");                    //false
```

如果将其改成大小写无关的，结果就返回 true 了。

```
/j.*t/i.test("Javascript");                   //true
```

同样的，我们也可以用测试一下 exec()方法，并访问它所返回数组的首元素。

```
/j.*t/i.exec("Javascript")[0];                //"Javascript"
```

3 以正则表达式为参数的字符串方法

String 对象的 IndexOf()和 lastIndexOf()方法来搜索文本。但这些方法只能用于纯字符串式的搜索，如果想获得更强大的文本搜索能力就需要用到正则表达式了。String 对象也为我们提供了这种能力。

在 String 对象中，以正则表达式对象为参数的方法主要有以下这些。

- match()方法：返回的是一个包含匹配内容的数组。
- search()方法：返回的是第一个匹配内容所在的位置。
- replace()方法：该方法能将匹配的文本替换成指定的字符串。
- split()方法：能根据指定的正则表达式将目标字符串分割成若干个数组元素。

4 search()与 match()

下面来看一些 search()与 match()方法的用例。首先，我们来新建一个 String 对象。

```
var s = new String('HelloJavaScriptWorld');
```

然后调用其 match()方法，这里返回的结果数组中只有一个匹配对象。

```
s.match(/a/);                    //["a"]
```

接下来，我们对其施加 g 修饰符，进行 global 搜索，这样一来返回的数组中就有了两个结果。

```
s.match(/a/g);                   //["a", "a"]
```

下面进行大小写无关的匹配操作。

```
s.match(/j.*a/i);                //["Java"]
```

而 search()方法则会返回匹配字符串的索引位置。

```
s.search(/j.*a/i);               //5
```

5 replace()

replace()方法可以将相关的匹配文本替换成某些其他字符串。在下面的示例中，我们移除了目标字符串中的所有大写字符（实际上是替换为空字

符串）。

```
s.replace(/[A-Z]/g, '');                    //"elloavacriptorld"
```

如果我们忽略了 g 修饰符，结果就只有首个匹配字符被替换掉。

```
s.replace(/[A-Z]/, '');                     //"elloJavaScriptWorld"
```

当某个匹配对象被找到时，如果我们想让相关的替换字符串中包含匹配的文本，可以使用$&来代替所找到的匹配文本。例如，下面我们在每一个匹配字符前面加了一个下划线。

```
s.replace(/[A-Z]/g, '_$&');                 //"_Hello_Java_Script_World"
```

如果正则表达式中分了组（即带括号），那么可以用$1 来表示匹配分组中的第一组，而$2 则表示第二组，以此类推。

```
s.replace(/([A-Z])/g, '_$1');               //"_Hello_Java_Script_World"
```

假设我们的 Web 页面上有一个注册表单，上面会要求用户输入 E-mail 地址、用户名和密码。当用户输入他们的 E-mail 地址时，我们可以利用 JavaScript 将 E-mail 的前半部分提炼出来，作为后面用户名字段的建议。

```
var email = "stoyan@phpied.com";
var username = email.replace(/(.*)@.*/, "$1");
username;                                   //"stoyan"
```

6 回调式替换

当我们需要执行一些特定的替换操作时，也可以通过返回字符串的函数来完成。这样，我们就可以在执行替换操作之前实现一些必要的处理逻辑。

```
function replaceCallback(match){
    return "_" + match.toLowerCase();
}
s.replace(/[A-Z]/g, replaceCallback);       //"_hello_java_script_world"
```

该回调函数可以接受一系列的参数（在上面的示例中，我们忽略了所有参数，但首参数是依然存在的）。

- 首参数是正则表达式所匹配的内容。

- 尾参数则是被搜索的字符串。
- 尾参数之前的参数表示的是匹配内容所在的位置。
- 剩下的参数可以是由 regex 模式所分组的所有匹配字符串组。

首先，我们新建一个变量，用于存储之后传递给回调函数的整个 arguments 对象。

```
var glob;
```

下一步是定义一个正则表达式，我们将 E-mail 地址分成三个匹配组，具体格式形如 something@something.something。

```
var re = /(.*)@(.*)\.(.*)/;
```

最后就是定义相应的回调函数了，它会接受 glob 数组中的参数，并返回相应的替换内容。

```
var callback = function(){
glob = arguments;
return arguments[1] + ' at ' + arguments[2] + ' dot ' +
arguments[3];
};
```

然后我们就可以这样调用它们了：

```
"stoyan@phpied.com".replace(re, callback);
"stoyan at phpied dot com"
```

下面是该回调函数返回的参数内容：

```
glob;
//["stoyan@phpied.com", "stoyan", "phpied", "com", 0,"stoyan@phpied.com"]
```

7 split()

split()方法能根据指定的分割字符串将我们的输入字符串分割成一个数组。下面就是我们用逗号将字符串分割的结果。

```
var csv = 'one, two,three ,four';
csv.split(',');                                    //["one", " two", "three ", "four"]
```

由于上面的输入字符串中存在逗号前后的空格不一致的情况，这导致

生成的数组也会出现多余的空格。如果我们使用正则表达式，就可以在这里用\s*修饰符来解决，意思就是“匹配 0 个或多个空格”。

```
csv.split(/\s*,\s*/);                    //["one", "two", "three", "four"]
```

8 用字符串来代替过于简单的 regexp 对象

split()、match()、search()和 replace()）这些方法可以接受的参数不仅仅是一些正则表达式，也包括字符串。它们会将接收到的字符串参数自动转换成 regex 对象，就像我们直接传递 new RegExp()一样。

【示例】下面的 replace()方法直接使用字符串参数来执行替换。

```
"test".replace('t', 'r');                //"rest"
```

它与下面的调用是等价的：

```
"test".replace(new RegExp('t'), 'r');    //"rest"
```

当然，在执行这种字符串传递时，我们就不能像平时使用构造器或者 regex 文本法那样设置表达式修饰符了。使用字符串而不是正则表达式来替换文本比较常见的错误是，使用者往往会误以为原字符串中所有的匹配都会替换。然而如上所述，以字符串为参数的 replace()其 global 修饰符的值将为 false，即只有第一个被匹配到的字符串才会被替换。这与其他一些编程语言不同，从而容易导致混淆。例如：

```
"pool".replace('o', '*');                //"p*ol"
```

而使用者大多数情况下的意图是替换所有的匹配：

```
"pool".replace(/o/g, "*");               //"p**l"
```

E.2.10 Error 对象

当代码中有错误发生时，一个好的处理机制可以帮助我们理解错误发生的原因，并且能以一种较为优雅的方式来纠正错误。在 JavaScript 中，将会使用 try、catch 及 finally 语句组合来处理错误。当程序中出现错误时，就会抛出一个 Error 对象，该对象可能由以下几个内建构造器中的一个产生

而成，它们包括 EvalError、RangeError、ReferenceError、SyntaxError、TypeError 和 URIError 等，所有这些构造器都继承自 Error 对象。

错误捕获很容易，只需要我们使用 try 语句后接一个 catch 语句即可。例如添加下面代码，我们就不会看到之前截图中的那个错误显示了。

```
try {
    iDontExist();
} catch (e){
    // do nothing
}
```

这里包含两部分内容。

- try 语句及其代码块。
- catch 语句及其参数变量和代码块。
- finally 语句并没有在这个例子中出现，这是一个可选项，主要用于执行一些无论如何（无论有没有错误发生）都要执行的内容。

在上面示例中，我们并没有在 catch 语句后面的代码块中写入任何内容，但实际上我们可以在这里加入一些用于修复错误的代码，或者至少可以将该应用程序错误的一些特定情况反馈给用户。

catch 语句的参数（括号中的）e 实际上是一个 Error 对象。跟其他对象一样，它也提供一系列有用的方法与属性。不同的浏览器对于这些方法与属性都有着各自不同的实现，但其中有两个属性的实现还是基本相同的，那就是 e.name 和 e.message。

【示例】下面示例是一个简单的演示。

```
try {
    iDontExist();
} catch (e){
    alert(e.name + ': ' + e.message);
} finally {
    alert('Finally!');
}
```

这里的第一个 alert()显示了 e.name 和 e.message，而后一个则显示了 Finally！字样。

在 Firefox 和 Chrome 中，第一个 alert()将显示的内容是 ReferenceError: iDontExist is not defined。而在 Internet Explorer 中则是 TypeError: Object expected。总之，这里向我们传递了两个信息。

- e.name 所包含的是构造当前 Error 对象的构造器名称。
- 由于 Error 对象在各宿主环境（浏览器）中的表现并不一致，因此在这里我们需要使用一些技巧，以便我们的代码能处理各种类型的错误（即 e.name 的值）。

当然，我们也可以用 new Error()或者其他 Error 对象构造器来自定义一个 Error 对象，然后告诉 JavaScript 引擎某个特定的条件，并使用 throw 语句来抛出该对象。

【示例】下面来看一个具体的示例，假设我们需要调用一个 maybeExists()函数，并将函数返回结果作为除数来执行除法运算。我们想统一进行错误处理， 无论错误原因是 maybeExists()函数不存在，还是返回值不是我们想要的，那么代码都应该这样写。

```
try {
    var total = maybeExists();
    if (total === 0) {
        throw new Error('Division by zero!');
    } else {
        alert(50 / total);
    }
} catch (e){
    alert(e.name + ': ' + e.message);
} finally {
    alert('Finally!');
}
```

根据 maybeExists()函数的存在与否及其返回值，这段代码会弹出几种

不同的信息。

- 如果 maybeExists() 函数不存在，我们在 Firefox 中将会得到信息“ReferenceError: maybeExists() is not defined”，而在 IE 中则为“TypeError: Object expected”。
- 如果 maybeExists()返回值为 0，我们将得到的信息是“Error: Division by zero!”。
- 如果 maybeExists()的返回值为 2，我们将得到的 alert 信息是 25。

在以上所有的情况下，程序都会弹出第二个 alert 窗口，内容为“Finally!”。另外，这里抛出的是一般性的错误提示，使用的是 throw new Error('Division by zero!')语句，然而我们也可以根据自身的需要来明确错误类型。例如，可以利用 throw new RangeError('Division by zero!')语句来抛出该错误，或者不用任何构造器，直接定义一个一般对象抛出。

```
throw {
    name: "MyError",
    message: "OMG! Something terrible has happened"
}
```

这样一来，我们就可以使用自定义的 Error 名，从而解决了浏览器之间由于抛出错误不相同所导致的问题。