

回到Web镇可乐公司

该利用新学到的数组排序知识帮助Web镇可乐公司了。当然，我们需要做的只是为它们编写一个比较函数，但这样做之前，再来看一眼数组

但我们没必要让它们知道这一点。

products:

```
var products = [ { name: "Grapefruit", calories: 170, color: "red", sold: 8200 },
  { name: "Orange", calories: 160, color: "orange", sold: 12101 },
  { name: "Cola", calories: 210, color: "caramel", sold: 25412 },
  { name: "Diet Cola", calories: 0, color: "caramel", sold: 43922 },
  { name: "Lemon", calories: 200, color: "clear", sold: 14983 },
  { name: "Raspberry", calories: 180, color: "pink", sold: 9427 },
  { name: "Root Beer", calories: 200, color: "caramel", sold: 9909 },
  { name: "Water", calories: 0, color: "clear", sold: 62123 }
];
```

别忘了，数组products的每个元素都是一个对象。我们要比较的不是这些对象，而是它们的特定属性，如sold。

那么我们按什么排序呢？首先来按销量以升序排列，为此需要比较每个对象的属性sold。需要注意的一点是，这是一个产品对象数组，因此需要向比较函数传递两个对象，而不是两个数字：

```
function compareSold(colA, colaB) {
  if (colaA.sold > colaB.sold) {
    return 1;
  } else if (colaA.sold === colaB.sold) {
    return 0;
  } else {
    return -1;
  }
}
```

compareSold将两个可乐产品对象（colaA和colaB）作为参数，并对它们的属性sold进行比较。

这个函数将导致方法sort根据销量升序排列可乐产品。

如果你愿意，完全可以像前面的练习中那样简化这些代码！

当然，要使用函数compareSold来对数组products进行排序，只需调用该数组的方法sort：

```
products.sort(compareSold);
```

别忘了，可对任何数组（数字数组、字符串数组、对象数组）调用方法sort来按任何顺序（升序或降序）排序。通过传入一个比较函数，我们获得了灵活性，还可重用代码。

测试排序代码

该来测试我们为Web镇可乐公司编写的代码了。下面整合了前几页的所有代码，并新增了一些测试代码。因此你只需创建一个简单的HTML页面（cola.html），将这些代码加入其中，再进行测试：

```
var products = [ { name: "Grapefruit", calories: 170, color: "red", sold: 8200 },
  { name: "Orange", calories: 160, color: "orange", sold: 12101 },
  { name: "Cola", calories: 210, color: "caramel", sold: 25412 },
  { name: "Diet Cola", calories: 0, color: "caramel", sold: 43922 },
  { name: "Lemon", calories: 200, color: "clear", sold: 14983 },
  { name: "Raspberry", calories: 180, color: "pink", sold: 9427 },
  { name: "Root Beer", calories: 200, color: "caramel", sold: 9909 },
  { name: "Water", calories: 0, color: "clear", sold: 62123 }
];
```

```
function compareSold(colmA, colaB) {
  if (colaA.sold > colaB.sold) {
    return 1;
  } else if (colaA.sold === colaB.sold) {
    return 0;
  } else {
    return -1;
  }
}
```

这是将传递给方法
sort的比较函数。

```
function printProducts(products) {
  for (var i = 0; i < products.length; i++) {
    console.log("Name: " + products[i].name +
      ", Calories: " + products[i].calories +
      ", Color: " + products[i].color +
      ", Sold: " + products[i].sold);
  }
}
```

我们首先使用compareSold来对数
组products进行排序。

```
products.sort(compareSold);
printProducts(products);
```

然后将排序后的数组打印出来。

这是我们运行上述代码得到的输出，注意
到数组products是按销量排序的。

这是一个新增的函数，它以漂
亮的格式将数组products打印
到控制台。如果只是编写代码
console.log(products)，你也能
看到输出，但不那么漂亮。

JavaScript控制台

```
Name: Grapefruit, Calories: 170, Color: red, Sold: 8200
Name: Raspberry, Calories: 180, Color: pink, Sold: 9427
Name: Root Beer, Calories: 200, Color: caramel, Sold: 9909
Name: Orange, Calories: 160, Color: orange, Sold: 12101
Name: Lemon, Calories: 200, Color: clear, Sold: 14983
Name: Cola, Calories: 210, Color: caramel, Sold: 25412
Name: Diet Cola, Calories: 0, Color: caramel, Sold: 43922
Name: Water, Calories: 0, Color: clear, Sold: 62123
```



编写按属性sold对可乐产品进行排序的函数后，该来编写用于按其他各种属性(name、calories和color)进行排序的比较函数了。请仔细查看控制台输出，确认按每个属性排序时，可乐产品的排列顺序都是正确的。答案见本章末尾。

请补全下面的三个比较函数。



```
function compareName(colmA, colaB) {
```

提示：比较字符串时，
也可使用<、>和==。

```
}
```

```
function compareCalories(colmA, colaB) {
```

```
}
```

```
function compareColor(colmA, colaB) {
```

```
}
```

```
products.sort(compareName);  
console.log("Products sorted by name:");  
printProducts(products);
```

```
products.sort(compareCalories);  
console.log("Products sorted by calories:");  
printProducts(products);
```

```
products.sort(compareColor);  
console.log("Products sorted by color:");  
printProducts(products);
```

对于每个新的比较函数，我们将其传递给sort方法，再在控制台中显示结果。

你真棒！





要点

- 定义函数的方式有两种：使用**函数声明**和使用**函数表达式**。
- **函数引用**是一个指向函数的值。
- 函数声明在执行代码前处理。
- 函数表达式在运行阶段随其他代码一起执行。
- 处理函数声明时，浏览器创建一个函数以及一个与函数同名的变量，并将指向函数的引用存储到变量中。
- 处理函数表达式时，浏览器创建一个函数，但你必须显式地处理指向这个函数的引用。
- 可将作为**一等公民**的值赋给变量，包含在数据结构中，传递给函数或从函数返回。
- 函数引用是作为一等公民的值。
- 数组的**sort**方法将一个函数作为参数，这个函数知道如何对两个数组元素进行比较。
- 传递给**sort**方法的函数必须返回下面的其中一个值：大于0的数字、0或小于0的数字。

练习答案



基于前述浏览器处理函数`quack`和`fly`的代码的方式，你能得出哪些有关函数声明和函数表达式的结论呢？请选择你认为正确的所有说法。继续往下阅读前，务必查看本章末尾的答案。

- ☒ 在处理其他代码之前，先处理函数声明。
- ☒ 函数表达式随其他代码一起被处理。
- ☒ 函数声明不返回指向函数的引用；而是创建一个与函数同名的变量，并将指向函数的引用赋给它。
- ☒ 函数表达式返回一个引用，该引用指向函数表达式创建的函数。
- ☒ 可在变量中存储指向函数的引用。
- ☒ 函数声明是完整的语句，而函数表达式只是语句的一部分。
- ☒ 对于函数声明和函数表达式创建的函数，用相同的方式处理调用它们的语句。
- ☐ 函数声明是一种屡试不爽的函数创建方式。
- ☐ 应尽可能使用函数声明，因为它们首先被处理。

这两种说法
不一定正确！

变身浏览器答案



下面是一些JavaScript代码。你的任务是像浏览器那样执行它们，并在每个函数被创建时，在右边将其记录下来。别忘了分两遍处理这些代码：第一遍处理函数声明，第二遍处理函数表达式。

```
var midi = true;
var type= "piano";
var midiInterface;

function play(sequence) {
    // 函数体的代码
}

var pause = function() {
    stop();
}

function stop() {
    // 函数体的代码
}

function createMidi() {
    // 函数体的代码
}

if (midi) {
    midiInterface = function(type) {
        // 函数体的代码
    };
}
```

请按顺序列出所创建函数的名称。对于根据函数表达式创建的函数，列出它被赋给的变量的名称。我们已经写出了创建的第一个函数。

play
stop
createMidi
pause
midiInterface

磨笔上阵 答案

为牢牢地记住函数是值的概念，咱们来玩一个碰运气取胜的游戏。请尝试下面这个猜豆子游戏，你会赢还是输呢？请试一试，并找出结果。答案如下。

```
var winner = function() { alert("WINNER!") };
var loser = function() { alert("LOSER!") };
// 咱们来热身
winner();
// 将函数引用赋给其他变量
```

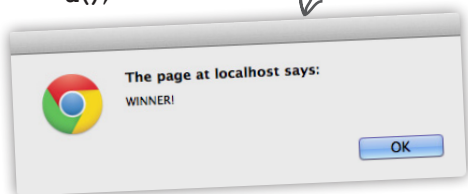
```
var a = winner;
var b = loser;
var c = loser;
a();
b();
```

// 来玩一个猜豆子游戏，看看你的运气怎样

```
c = a;      ← c指向函数winner
a = b;      ← a指向函数loser
b = c;      ← b指向函数winner
c = a;      ← c指向函数loser
a = c;      ← a指向函数loser
a = c;      ← a指向函数winner
a = b;      ← b指向函数loser
b = c;      ← 调用a，即函数winner!!
a();
```

别忘了，这些变量分别存储了指向函数winner和loser的引用。我们可以像其他任何值一样，将这些引用赋给其他变量。

别忘了，随时都可以调用指向函数的引用。





下面来热热身，尝试将一个函数传递给另一个函数。请在心中执行下面的代码，看看它们的结果是什么。答案如下。

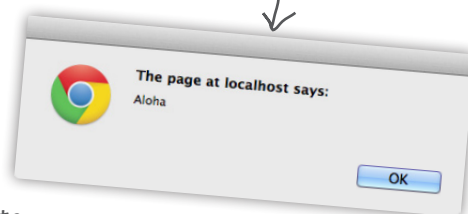
```
function sayIt(translator) {
  var phrase = translator("Hello");
  alert(phrase);
}
```

定义一个将函数作为参数的函数，并在其中调用传入的函数。

```
function hawaiianTranslator(word) {
  if (word == "Hello") return "Aloha";
  if (word == "Goodbye") return "Aloha";
}
```

```
sayIt(hawaiianTranslator);
```

将函数hawaiianTranslator传递给函数sayIt。



又轮到你了：编写一个函数，将乘客姓名以及是否已买票的信息打印到控制台。将这个函数传递给processPassengers，对其进行测试。我们编写了这个函数的函数头，你只需编写其代码即可。答案如下。

```
function printPassenger(passenger) {
  var message = passenger.name;
  if (passenger.paid === true) {
    message = message + " has paid";
  } else {
    message = message + " has not paid";
  }
  console.log(message);
  return false;
}

processPassengers(passengers, printPassenger);
```

这个返回值没多大意义，因为在processPassengers中，没有考虑这个返回的结果。

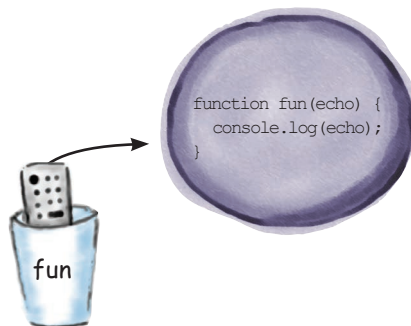
JavaScript控制台

```
Jane Doloop has paid
Dr. Evel has paid
Sue Property has not paid
John Funcall has paid
```



下面的代码创建了一个函数，并将其赋给变量fun。

```
function fun(echo) {
    console.log(echo);
};
```



请注意，你的浏览器在控制台中显示的函数fun和boo的值可能不同，请尝试在两三个不同的浏览器中执行这些代码。

请执行下面的代码，并将其输出写在右边。请先在心中执行这些代码；如果看不懂，再在计算机中执行它们。

```
fun("hello");
```

hello

```
function boo(aFunction) {
    aFunction("boo");
}
```

```
boo(fun);
```

boo

```
console.log(fun);
```

function fun(echo) { console.log(echo); }

```
fun(boo);
```

function boo(aFunction) { aFunction("boo"); }

```
var moreFun = fun;
```

```
moreFun("hello again");
```

hello again

选做题（这涉及
接下来要介绍的
内容）

```
function echoMaker() {
    return fun;
}

var bigFun = echoMaker();
bigFun("Is there an echo?");
```

Is there an echo?

继续往下阅读前，务必查看并理解答案，这非常重要！



练习答案

你的任务是修改前面的代码，以支持第三类乘客：豪华经济舱乘客。对于豪华经济舱乘客，除了可乐和开水外，还可以点葡萄酒。另外，请实现支持下述菜单的getDinnerOrderFunction。

头等舱：鸡肉或意大利面。

豪华经济舱：快餐或芝士拼盘。

经济舱：花生饼或椒盐脆饼。

答案如下。

```
var passengers = [ { name: "Jane Doloop", paid: true, ticket: "coach" },
                   { name: "Dr. Evel", paid: true, ticket: "firstclass" },
                   { name: "Sue Property", paid: false, ticket: "firstclass" },
                   { name: "John Funcall", paid: true, ticket: "premium" } ];
```

```
function createDrinkOrder(passenger) {
    var orderFunction;
    if (passenger.ticket === "firstclass") {
        orderFunction = function() {
            alert("Would you like a cocktail or wine?");
        };
    } else if (passenger.ticket === "premium") {
        orderFunction = function() {
            alert("Would you like wine, cola or water?");
        };
    } else {
        orderFunction = function() {
            alert("Your choice is cola or water.");
        };
    }
    return orderFunction;
}
```

为方便测试这里的新代码，我们将John Funcall购买的机票类型改成了豪华经济舱。

这是处理豪华经济舱的新代码。将根据乘客购买的机票类型，在三个让乘客点饮料的函数中返回一个。

将所有相关的逻辑都封装在一个函数中，这个函数知道如何为乘客创建正确的点饮料函数。这提供了极大的便利。

让乘客点饮料时，无需执行这些逻辑，因为已经有了一个专为该乘客定制的点饮料函数。



练习
答案

```
function createDinnerOrder(passenger) {
    var orderFunction;
    if (passenger.ticket === "firstclass") {
        orderFunction = function() {
            alert("Would you like chicken or pasta?");
        };
    } else if (passenger.ticket === "premium") {
        orderFunction = function() {
            alert("Would you like a snack box or cheese plate?");
        };
    } else {
        orderFunction = function() {
            alert("Would you like peanuts or pretzels?");
        };
    }
    return orderFunction;
}
```

← 我们添加了一个全新的函数 `createDinnerOrder`，用于创建让乘客点餐的函数。

← 这个函数的工作原理与 `createDrinkOrder` 相同：检查乘客购买的机票类型，并据此返回一个为乘客定制的点餐函数。

```
function serveCustomer(passenger) {
    var getDrinkOrderFunction = createDrinkOrder(passenger);
    var getDinnerOrderFunction = createDinnerOrder(passenger);

    getDrinkOrderFunction();

    // 让乘客点餐
    getDinnerOrderFunction();

    getDrinkOrderFunction();
    getDrinkOrderFunction();
    // 播放电影
    getDrinkOrderFunction();
    // 清理垃圾
}
```

← 为乘客创建正确的点餐函数……

← 每当需要让乘客点餐时，我们都调用这个函数。

```
function servePassengers(passengers) {
    for (var i = 0; i < passengers.length; i++) {
        serveCustomer(passengers[i]);
    }
}
```

```
servePassengers(passengers);
```



你认为下面的代码是做什么的？你能通过几个示例来说明如何使用这个函数吗？答案如下。

```
function addN(n) {
    var adder = function(x) {
        return n + x;
    };
    return adder;
}
```

这个函数接受一个参数 n 。它创建并返回一个这样的函数：也接受一个参数 x ，并返回 n 与 x 的和。

因此，可使用它来创建一个总是将传入的数字加上2的函数，如这里所示：

```
var add2 = addN(2);
console.log(add2(10));
console.log(add2(100));
```



方法`sort`为何按升序排列数组`numbersArray`呢？因为我们比较函数中返回1、0和-1时，实际上是在这样告诉`sort`方法：

如果返回值为1，就将第一项放在第二项后面；

如果返回值为0，就说明两项相等，可保留它们的位置不变；

如果返回值为-1，就将第一项放在第二项前面。

要修改代码，以便按降序排列，只需反转上述逻辑，让返回值1表示将第二项放在第一项后面，返回值-1表示将第二项放在第一项前面（返回值0表示保持位置不变）。请编写一个按降序排列的新比较函数。

```
function compareNumbersDesc(num1, num2) {
    if (num2 > num1) {
        return 1;
    } else if (num1 == num2) {
        return 0;
    } else {
        return -1;
    }
}
```



你知道，传递给`sort`的比较函数需要根据比较结果返回一个大于0、等于0或小于0的数字：如果第一项大于第二项，就返回一个大于0的值；如果第一项等于第二项，就返回0；如果第一项小于第二项，就返回一个小于0的值。

利用这些知识，你能够重写对两个数字进行比较的`compareNumbers`，让其包含更少的代码吗？

答案如下。

```
function compareNumbers(num1, num2) {
    return num1 - num2;
}
```

通过返回将`num2`和`num1`相减的结果，可将这个函数简化为只包含一行代码。请通过两三个示例来理解其中的工作原理。别忘了，方法`sort`要求比较函数返回一个大于、等于或小于0的数字，而不是1、0或-1（虽然很多比较函数都返回这些值）。



编写按属性sold对可乐产品进行排序的函数后，该来编写用于按其他各种属性（name、calories和color）进行排序的比较函数了。请仔细查看控制台输出，确认按每个属性排序时，可乐产品的排列顺序都是正确的。答案如下。

下面是各个比较函数的实现。



```
function compareName(colmA, colaB) {
  if (colaA.name > colaB.name) {
    return 1;
  } else if (colaA.name === colaB.name) {
    return 0;
  } else {
    return -1;
  }
}
```

```
function compareCalories(colmA, colaB) {
  if (colaA.calories > colaB.calories) {
    return 1;
  } else if (colaA.calories === colaB.calories) {
    return 0;
  } else {
    return -1;
  }
}
```

```
function compareColor(colmA, colaB) {
  if (colaA.color > colaB.color) {
    return 1;
  } else if (colaA.color === colaB.color) {
    return 0;
  } else {
    return -1;
  }
}
```

```
products.sort(compareName);
console.log("Products sorted by name:");
printProducts(products);

products.sort(compareCalories);
console.log("Products sorted by calories:");
printProducts(products);

products.sort(compareColor);
console.log("Products sorted by color:");
printProducts(products);
```

全部完成了!



你真棒!

对于每个新的比较函数，我们将其传递给sort方法，再在控制台中显示结果。



11 匿名函数、作用域和闭包

系统地讨论函数

自从学了匿名函数，我的
表达能力强了三倍。



你已经全面了解了函数，但还需更深入地学习。本章将再进一步，深入函数的核心，演示如何娴熟地利用函数。这一章虽然不太长，但涵盖的知识点非常多；阅读完本章后，你的JavaScript表达能力将超乎你的想象。你还将为与人协作编写代码或使用开源JavaScript库作好准备，因为我们将介绍一些与函数相关的常见编码习惯和约定。如果你从未听说过匿名函数和闭包，那真是来对了地方。



如果你听说过闭包，但不太确定它是什么，那么更是来对了地方！

函数的另一面

你已经见识了函数的两个方面：你见识了函数声明正规、说明性的一面，还见识了函数表达式更宽松、表达力更强的一面。现在向你介绍函数有趣的另一面：匿名性。

所谓匿名（anonymous），指的是没有名称的函数。怎么可能出现这样的情况呢？使用函数声明定义函数时，必须给它指定名称；但使用函数表达式定义函数时，不必给它指定名称。

你可能会说，这确实很有趣，也是有可能的，但那又如何呢？通过使用匿名函数，可让代码更简洁精练，可读性更强，效率更高，甚至更易于维护。

下面来看看如何创建和使用匿名函数。我们将从以前见过的一个代码片段着手，看看匿名函数可提供什么样的帮助。



这是一个加载事件处理程序，指定方式与以前相同。

我们首先定义了一个函数。这个函数有名称handler。

```
function handler() { alert("Yeah, that page loaded!"); }  
window.onload = handler;
```

接下来，我们使用这个函数的名称（handler）将其赋给window对象的属性onload。

这样，网页加载完毕后，将调用函数handler。



磨笔上阵

请根据你已有的函数和变量知识，判断下述哪些说法是正确的。

- ☐ 变量handler存储了一个函数引用。

☐ 将handler赋给window.onload时，实际上是将一个函数引用赋给了它。

☐ 变量handler仅用于将其存储的函数引用赋给window.onload。

☐ 我们绝不会再使用handler，因为其代码仅在网页加载完毕后执行一次。
- ☐ 调用onload处理程序两次是个馊主意，因为这种处理程序通常用于为整个网页做一些初始化工作，多次调用它可能引发问题。

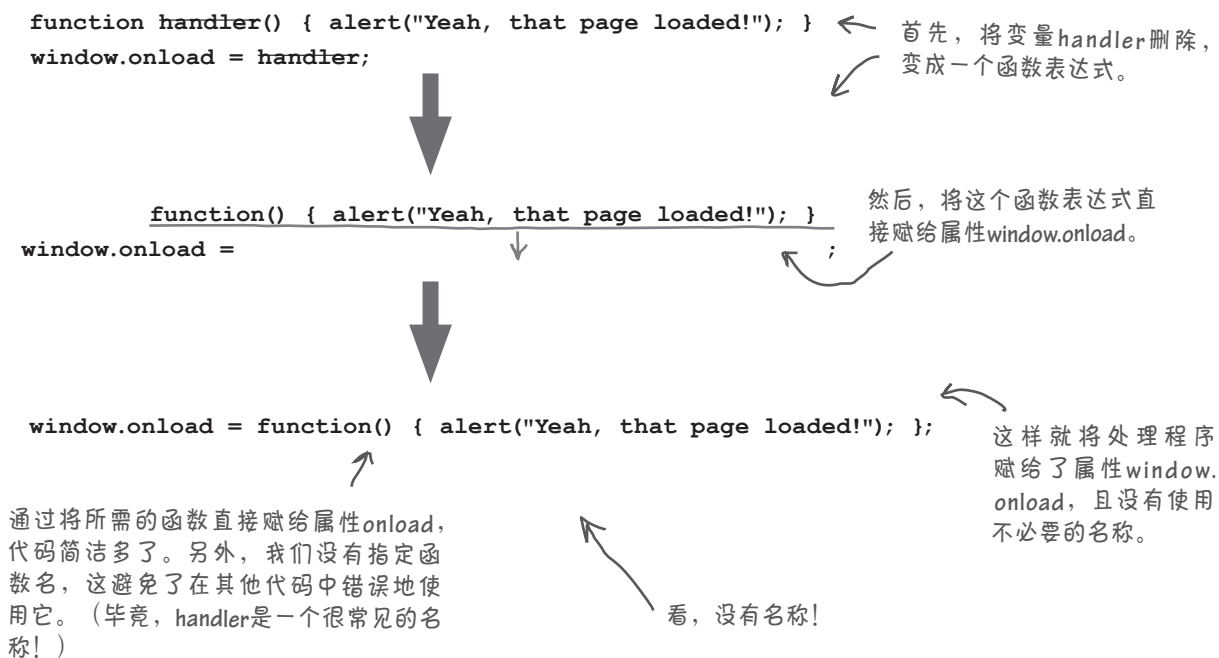
☐ 函数表达式创建函数引用。

☐ 我们是不是说过，将handler赋给window.onload时，实际上是将一个函数引用赋给了它？

如何使用匿名函数

我们创建了一个处理加载事件的函数，但我们知道这是一个一次性函数，因为在每个网页中，加载事件只发生一次。我们还注意到，将一个函数引用（具体地说是handler存储的函数引用）赋给了属性window.onload。然而，由于handler是个一次性函数，这个名称显得有点多余，因为我们仅使用它来将其存储的引用赋给属性window.onload。

匿名函数提供了一种简化这些代码的途径。匿名函数是没有名称的函数表达式，用于通常需要函数引用的地方。为明白这一点，来看看如何在代码中以匿名方式使用函数表达式。



考考你的
脑力



在本书前面的代码中，是否有你没有意识到的匿名函数？

提示：它们是不是隐藏在对象中？



磨笔上阵

在下面的代码中，有几个地方都可使用匿名函数。请尽可能地使用匿名函数重新编写这些代码。你可将旧代码删除，并写入新代码。你还需完成另一个任务：将这些代码中的匿名函数圈出来。

```
window.onload = init;

var cookies = {
  instructions: "Preheat oven to 350...",
  bake: function(time) {
    console.log("Baking the cookies.");
    setTimeout(done, time);
  }
};

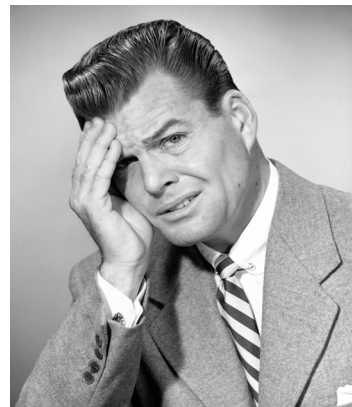
function init() {
  var button = document.getElementById("bake");
  button.onclick = handleButton;
}

function handleButton() {
  console.log("Time to bake the cookies.");
  cookies.bake(2500);
}

function done() {
  alert("Cookies are ready, take them out to cool.");
  console.log("Cooling the cookies.");
  var cool = function() {
    alert("Cookies are cool, time to eat!");
  };
  setTimeout(cool, 1000);
}
```

别这么啰嗦好不好

我不想旧话重提，因为你已经深入地学习了函数，知道如何传递函数、如何将其赋给变量、如何将其传递给函数以及如何从函数返回它们，但你使用函数的方式还是太啰嗦（换言之，你的表达能力依然没有达到应有的水平）。我们来看一个示例吧。



这个名为`cookieAlarm`的函数看起来很正常，它显示一个提示框，指出小甜饼做好了。

```
function cookieAlarm() {
    alert("Time to take the cookies out of the oven");
};
```

```
setTimeout(cookieAlarm, 600000);
```

将前述函数作为实参传递给`setTimeout`。

小甜饼将在10分钟后做好。

提醒一下，这个数字的单位为毫秒，因此 $60\,000/1000/60 = 10$ 分钟。

这些代码看起来挺好，但通过使用匿名函数，可使其更加紧凑。如何做到呢？看看调用`setTimeout`时使用的变量`cookieAlarm`吧。这个变量是一个函数引用，也就是说，调用`setTimeout`时，传入了一个函数引用。要获取函数引用，一种方式是使用指向函数的变量，另一种方式与两页前的`window.onload`示例一样，使用函数表达式。下面使用函数表达式来重写上述代码。

这里调用`setTimeout`时没有使用变量，而是直接使用函数表达式。

请注意这里的语法。我们书写了以右花括号结尾的整个函数表达式；然后像其他实参一样，在它后面加上逗号，再添下一个实参。

```
setTimeout(function() { alert("Time to take the cookies out of the oven"); }, 600000);
```

我们指定了要调用的函数的名称`setTimeout`，在它后面加上圆括号，再加上第一个实参——一个函数表达式。

函数表达式的后面是第二个实参。

你骗谁呢？这太乱了。谁喜欢阅读这么长的一行代码呢？如果函数更长、更复杂呢？



对于简短的代码片段，写成一行挺好。但在其他情况下，你说的没错，这样编写代码太难看了。不过你知道，在JavaScript中，可大量使用空白，因此可以根据需要插入任意数量的空格和回车，以提高代码的可读性。下面调整了前一页的setTimeout代码的格式。

我们只是插入了一些空白，
即添加一些空格和回车。

```
setTimeout(function() {  
    alert("Time to take the cookies out of the oven");  
}, 600000);
```

这样，代码的可读性高得多了。
很高兴你提出这个问题。

等等，我想我明白了。函数表达式的结果是一个函数引用，因此在任何需要函数引用的地方，都可以使用函数表达式。是这样的吗？

说得有点拗口，不过你理解的没错。这是理解函数是一等公民的关键之一。在需要函数引用的地方，都可使用函数表达式，因为其结果就是一个函数引用。正如你刚才看到的，如果需要将一个函数作为实参，可将实参指定为函数表达式（同样，传递前将获取其结果——一个函数引用）。同样，如果需要从函数返回一个函数，也可返回一个函数表达式。





为确保你理解了将匿名函数表达式传递给函数的语法，请对下述将变量（这里是vaccine）用作实参的代码进行修改，将匿名函数表达式用作实参。

```
function vaccine(dosage) {  
    if (dosage > 0) {  
        inject(dosage);  
    }  
}  
  
administer(patient, vaccine, time);
```

← 将转换后的版本写在这里，再查看本章末尾的答案，然后接着往下阅读。

世上没有愚蠢的问题

问：这样使用匿名函数看起来确实晦涩难懂，我真的需要熟悉这种做法吗？

答：你必须熟悉。在JavaScript代码中，经常会使用匿名函数表达式。要读懂别人编写的代码或JavaScript库，你必须知道其中的工作原理，并能够识别匿名函数表达式。

问：使用匿名函数表达式真的更好吗？在我看来，这样做只会让代码更复杂，更难阅读和理解。

答：这需要时间。随着时间的推移，你将能够轻松地分析这样的代码。另外，在很多情况下，这种语法都可降低代码的复杂度，让代码更整洁，其意图更清晰。虽然如此，过度地使用这种技巧绝对可能导致代码极难理解。但只要坚持，你就能掌握这种语法，届时它将变得更容易理解，也更有用。你将遇到很多大量使用匿名函数的代码，因此最好将这种技巧加入你的代码工具腰带。

问：既然作为一等公民的函数用途这么大，为何其他语言不支持它？

答：实际上支持（那些当前不支持的也正在考虑这样做）。例如，Scheme和Scala等语言与JavaScript一样，全面支持将函数作为一等公民；而PHP、最新的Java版本、C#和Objective-C等语言在一定程度上支持将函数作为一等公民。随着越来越多的人认识到将函数作为一等公民在编程语言中的价值，将有更多的语言提供这样的支持。然而，每种语言支持的方式都稍有不同，因此当你探索其他语言的这种功能时，一定要有心理准备。

函数是在什么时候定义的？这要看情况

函数有一个很微妙的地方，我们还未提及。别忘了，浏览器分两遍来处理JavaScript代码：第一遍，浏览器分析所有的函数声明，并定义这些函数声明创建的函数；第二遍，浏览器按从上到下的顺序执行代码，并定义函数表达式创建的函数。有鉴于此，使用函数声明创建的函数是在使用函数表达式创建的函数之前定义的，而这又决定了你可在什么时候和什么地方调用函数。

为明白这一点，来看一个具体的示例。下面的代码摘自上一章，但稍微调整了一下顺序。下面来处理这些代码。

请务必按数字标示的顺序阅读，从1开始，再到2，依此类推。这很重要。

❶ 从代码开头开始，查找所有的函数声明。

❷ 再次从代码开头开始，这次执行代码。

`var migrating = true;` ❸ 创建变量migrating，并将其设置为true。

注意到我们将这条if语句从代码末尾移到了这里。

```
if (migrating) {
    quack(4);
    fly(4);
}
```

❹ 条件为true，因此执行后面的代码块。

❺ 获取变量quack存储的函数引用，并使用实参4调用它。

❻ 获取变量fly存储的函数引用。糟糕，fly还没有定义！

```
var fly = function(num) {
    for (i = 0; i < num; i++) {
        console.log("Flying!");
    }
};
```

❼ 找到一个函数声明。定义它创建的函数，并将其赋给变量quack。

```
function quack(num) {
    for (i = 0; i < num; i++) {
        console.log("Quack!");
    }
}
```

❽ 已到达代码末尾。只找到了一个函数声明。



为何fly未定义

你知道，试图调用函数fly时它还未定义，为什么会这样呢？毕竟quack一点问题都没有。你可能猜到了，不同于函数声明quack，在第一遍处理代码时被定义，fly要等到第二遍从上到下执行代码时才被定义。我们再来看一下：

执行这些代码，并尝试调用quack时，一切都如预期的那样，因为quack在第一遍处理代码时就定义了。

```
var migrating = true;
if (migrating) {
  quack(4);
  fly(4);
}

var fly = function(num) {
  for (var i = 0; i < num; i++) {
    console.log("Flying!");
  }
};

function quack(num) {
  for (var i = 0; i < num; i++) {
    console.log("Quack!");
  }
}
```

当尝试调用函数fly时出现了错误，因为它还未定义……

这是由于fly要等到这条语句执行时才会被定义，而这条语句位于调用函数fly的语句后面。

试图调用未定义的函数时出现的情况。

根据你使用的浏览器，出现的错误可能类似于下面这样：TypeError: Property 'fly' of object [object Object] is not a function。

这到底意味着什么呢？首先，这意味着可将函数声明放在任何地方（代码的开头、末尾、中间），且可在任何地方调用它们。在代码的任何地方，函数声明创建的函数都是已定义的，这被称为提升（hoisting）。

函数表达式显然不同，因为它创建的函数要等到它执行后才被定义。因此，即便将函数表达式赋给全局变量（如前面的fly），也要等到它创建的函数被定义后，才能使用这个全局变量来调用该函数。

在这个示例中，两个函数的作用域都是全局。这意味着这两个函数被定义后，在代码的任何地方都是可见的。我们还需要考虑嵌套函数（在其他函数中定义的函数），因为这会影响函数的作用域。下面就来看看。

如何嵌套函数

在函数中定义其他函数完全可行，这意味着你可以在函数中使用函数声明或函数表达式。这其中的工作原理是什么呢？简单地说，在代码顶层定义的函数与在函数中定义的函数之间的唯一差别在于，它们的作用域不同。换言之，将函数放在另一个函数中时，将影响它在代码的哪些地方可见。

为明白这一点，来稍微扩展一下前面的示例，添加一些嵌套（nested）的函数声明和函数表达式。

```
var migrating = true;
var fly = function(num) {
  var sound = "Flying";
  function wingFlapper() {
    console.log(sound);
  }
  for (var i = 0; i < num; i++) {
    wingFlapper();
  }
};

function quack(num) {
  var sound = "Quack";
  var quacker = function() {
    console.log(sound);
  };
  for (var i = 0; i < num; i++) {
    quacker();
  }
}

if (migrating) {
  quack(4);
  fly(4);
}
```

在函数表达式fly中，添加了一个名为wingFlapper的函数声明。

调用wingFlapper。

在函数声明quack中，添加了一个函数表达式，并将其赋给变量quacker。

调用quacker。

将这些代码移到最后，以免调用fly时引发错误。



练习

在上述代码中，用铅笔标出函数fly、quack、wingFlapper和quacker的作用域。另外，标出你认为这些函数可见但还未定义的地方。

嵌套对作用域的影响

在代码顶层定义的函数是全局的，而在函数中定义的函数是局部的。下面来研究一下这些代码，看看每个函数的作用域，同时确定各个函数在什么地方是已定义的（即不是undefined）。

```
var migrating = true;
var fly = function(num) {
  var sound = "Flying";
  function wingFlapper() {
    console.log(sound);
  }
  for (var i = 0; i < num; i++) {
    wingFlapper();
  }
};
function quack(num) {
  var sound = "Quack";
  var quacker = function() {
    console.log(sound);
  };
  for (var i = 0; i < num; i++) {
    quacker();
  }
}
if (migrating) {
  quack(4);
  fly(4);
}
```

- 在代码顶层定义的东西都是全局的，因此fly和quack都是全局变量。
- 但别忘了，仅当执行这个函数表达式后，fly才是已定义的。
- wingFlapper是在函数fly中由一个函数声明定义的，因此其作用域为整个fly函数，即在fly函数的整个函数体内都是已定义的。
- quacker是在函数quack中由一个函数表达式定义的，因此其作用域为整个quack函数；但仅在这个函数表达式被执行后且在到达函数quack末尾前，它才是已定义的。
- quacker仅在这个范围内是已定义的。

世上没有愚蠢的问题

问： 将一个函数传递给另一个函数时，这个函数将存储在一个形参中，进而被视为所调用函数的一个局部变量。这样理解对吗？

答： 完全正确。将函数作为实参传递给另一个函数时，传入的函数引用将被复制到所调用函数的一个形参变量中。与其他形参一样，存储函数引用的形参也是局部变量。

请注意，在函数内部的什么地方可引用嵌套函数的规则，与在整个代码的什么地方可引用全局函数的规则相同。也就是说，在函数内部，如果你使用函数声明创建了一个全套函数，那么在这个函数的函数体的任何地方，嵌套函数都是已定义的；如果你使用函数表达式创建了一个嵌套函数，则在这个函数的函数体内，仅当函数表达式执行后，嵌套函数才是已定义的。

严峻的JavaScript挑战

我们需要一位一等函数方面的专家，听说你就是！我们被下面两个代码片段难住了，需要你的帮助。在我们看来，这两段代码几乎完全相同，只是其中一段使用了一个一等函数，而另一段没有。根据对JavaScript作用域的认识，我们认为代码片段#1的结果为008，而代码片段#2的结果为007。但实际上，它们的结果都是008！你能帮我们弄明白为什么会这样吗？

往下阅读前，建议在本页将
你的看法记录下来。

代码片段 #1

```
var secret = "007";  
  
function getSecret() {  
    var secret = "008";  
  
    function getValue() {  
        return secret;  
    }  
    return getValue();  
}  
  
getSecret();
```

代码片段 #2

```
var secret = "007";  
  
function getSecret() {  
    var secret = "008";  
  
    function getValue() {  
        return secret;  
    }  
    return getValue;  
}  
  
var getValueFun = getSecret();  
getValueFun();
```

现在不要查看本章末尾的答案，
稍后我们将重温这个挑战。



词法作用域简介

词法 (lexical) 意味着只需查看代码的结构就可确定变量的作用域，而不是等到代码执行时才明白。

既然说到作用域，下面来简单地介绍一下词法作用域的工作原理。

```
var justAVar = "Oh, don't you worry about it, I'm GLOBAL";
```

这里定义了一个名为 justAVar 的全局变量。

```
function whereAreYou() {
  var justAVar = "Just an every day LOCAL";
```

这个函数定义了一个新的词法作用域……

```
  return justAVar;
}
```

其中有一个局部变量 justAVar，它遮住了同名的全局变量。

```
var result = whereAreYou();
console.log(result);
```

这个函数被调用时返回 justAVar，不过是哪个 justAVar 呢？我们使用的是词法作用域，因此在最近的函数作用域内查找 justAVar；如果在这个作用域内没有找到，再在全局作用域内查找。

因此，调用 whereAreYou 时，它将返回局部变量 justAVar（而不是全局变量 justAVar）的值。

JavaScript 控制台

Just an every day LOCAL

下面来引入一个嵌套函数。

```
var justAVar = "Oh, don't you worry about it, I'm GLOBAL";
```

```
function whereAreYou() {
```

这个函数与前面相同。

```
  var justAVar = "Just an every day LOCAL";
```

这个局部变量遮住了同名的全局变量。

```
    function inner() {
      return justAVar;
    }
```

但现在定义了一个嵌套函数，并在其中引用了 justAVar。这引用的是哪个变量呢？同样，引用的总是最近的外部函数中的变量。因此，这里引用的变量与前面相同。

```
  return inner();
}
```

注意到这里调用了 inner，并返回其结果。

```
var result = whereAreYou();
console.log(result);
```

因此调用 whereAreYou 时，将调用函数 inner，进而返回局部变量 justAVar（而不是全局变量 justAVar）的值。

JavaScript 控制台

Just an every day LOCAL

词法作用域的有趣之处

下面再来作一点调整，请瞪大眼睛，情况将发生翻天覆地的变化。

```
var justAVar = "Oh, don't you worry about it, I'm GLOBAL";
```

```
function whereAreYou() {
```

```
    var justAVar = "Just an every day LOCAL";
```

```
    function inner() {
```

```
        return justAVar;
```

```
    }
```

```
    return inner;
```

```
}
```

没有任何变化，变量还是原来的变量，函数还是原来的函数。

但这里没有调用函数inner，而是返回它。

```
var innerFunction = whereAreYou();
```

```
var result = innerFunction();
```

```
console.log(result);
```

因此调用whereAreYou时，将获得一个指向函数inner的引用。我们将这个函数引用赋给变量innerFunction，再调用innerFunction，将其结果赋给变量result，并打印这个变量。

那么这里通过innerFunction调用inner时，将返回哪个justAVar变量呢？局部变量justAVar还是全局变量justAVar？

重要的是调用函数的时机。我们调用返回的函数inner时，全局变量justAVar在作用域内，因此结果为“**Oh don't worry about it, I'm GLOBAL**”。

别这么快下结论。在词法作用域中，重要的是函数是在什么地方定义的，因此结果肯定是局部变量justAVar的值，即“**Just an every day LOCAL**”。





Frank: 你凭什么说你是对的? 这有悖于物理定律。局部变量根本就不存在了……我是说离开其作用域后, 变量便不再存在了。它消失 (derezzed) 了! 你难道没看过电影《创: 战纪》(TRON) 吗?

Judy: 在不那么强大的C++和Java语言中也许如此, 但在JavaScript中不是这样的。

Jim: 别逗了, 这怎么可能? 函数whereAreYou来去匆匆, 局部变量justAVar已不复存在。

Judy: 我刚才不是跟你说了嘛, 在JavaScript中, 局部变量不是这样的。

Frank: Judy, 你说说是什么样的。

Judy: 定义函数inner时, 局部变量justAVar在这个函数的作用域内。不过词法作用域指出, 重要的是定义的方式; 既然我们使用的是词法作用域, 那么每当inner被调用时, 它都认为这个局部变量还存在, 需要时可直接使用。

Frank: 可是我前面说过, 这看起来有悖于物理定律。定义局部变量justAVar的函数whereAreYou已执行完毕, 这个变量已不复存在。

Judy: 没错, 函数whereAreYou已执行完毕, 但其作用域依然存在, 可供inner使用。

Jim: 这怎么可能?

Judy: 咱们来看看定义并返回函数时发生的情况吧。

编辑说明: 一页之间, Joe 怎么就换了衬衫呢? !

再谈函数

实在抱歉，到现在我们还没有将有关函数的一切和盘托出。即便在你问及“函数引用指向的到底是什么”时，我们也有点回避问题，说“将其视为封装好的函数，其中包含函数的代码块就好了”。

现在该将一切都告诉你了。

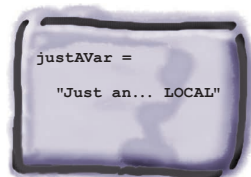
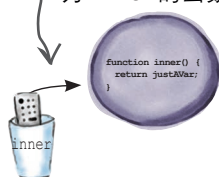
为此，咱们从函数whereAreYou着手，看看执行这些代码时实际发生的情况。

```
function whereAreYou() {  
    var justAVar = "Just an every day LOCAL";  
  
    function inner() {  
        return justAVar;  
    }  
  
    return inner;  
}
```

① 我们首先遇到了局部变量justAVar，并将字符串"Just an every day LOCAL"赋给了这个变量。

② 所有的局部变量都存储在一个环境（environment）中，这一点我们以前没有提及。

③ 接下来，我们创建一个名为inner的函数。

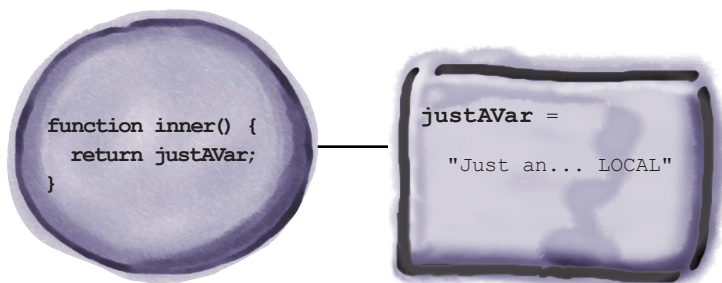


这就是环境，它存储了在局部作用域内定义的所有变量。

在这个示例中，环境中只有一个变量——变量justAVar。

④ 最后，当我们返回这个函数时，返回的不仅仅是函数，还有与之相关联的环境。

每个函数都有与之相关联的环境，其中包含它所处作用域内的局部变量。



下面来看看调用函数inner时，如何使用其环境。

再谈函数调用

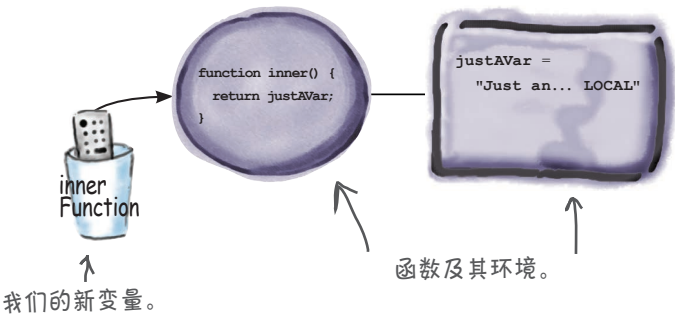
有了函数inner及其环境后，我们来调用函数inner，看看将发生的情况。
我们要执行的代码如下。

```
var innerFunction = whereAreYou();  
var result = innerFunction();  
console.log(result);
```

- 1
- 首先，我们调用whereAreYou。我们知道，这将返回一个函数引用，因此我们创建变量innerFunction，并将返回的函数引用赋给它。别忘了，有一个环境与这个函数相关联。

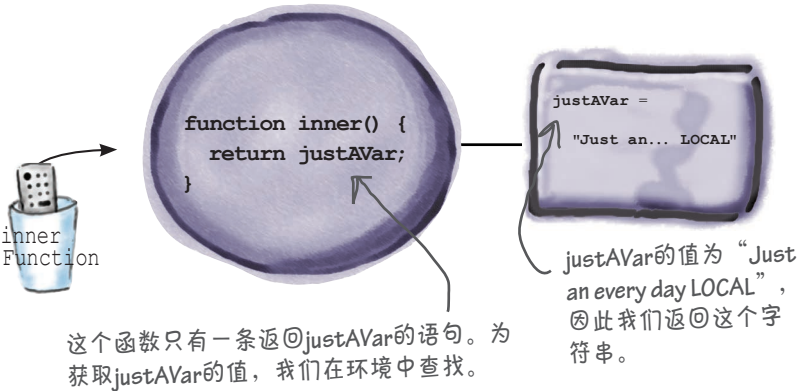
```
var innerFunction = whereAreYou();
```

这条语句执行完毕后，我们就有了变量innerFunction，它指向whereAreYou返回的函数（及其环境）。



- 2
- 接下来，我们调用innerFunction。这将在相应的环境中执行它指向的函数的代码，如下所示。

```
var result = innerFunction();
```



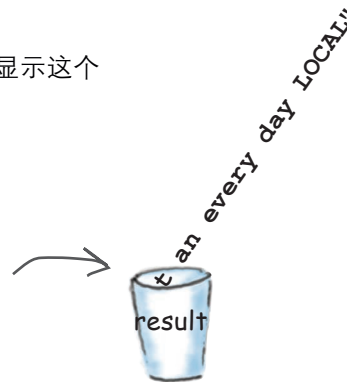
- ③ 最后，我们将这个函数的结果赋给变量`result`，再在控制台中显示这个变量。

```
var result = innerFunction();
```

```
console.log(result);
```

innerFunction返回从相应的环境中获取的值“Just an every day LOCAL”，因此我们将这个字符串存储到变量`result`中。

接下来，我们只需在控制台中显示这个字符串即可。



JavaScript控制台

Just an every day LOCAL

可恶！Judy又对了。

等等，Judy没有提到闭包，而我们这里做的好像与闭包相关。咱们来学习闭包，看看能否利用闭包来战胜她。

伙计们，这就是闭包，你们最好仔细地研究一下。



世上没有愚蠢的问题

问：你说词法作用域决定了变量是在哪里定义的，这是什么意思？

答：所谓词法作用域，指的是JavaScript的作用域规则完全基于代码的结构，而不是一些动态的运行阶段属性。这意味着只需查看代码的结构，就能确定变量是在什么地方定义的。另外别忘了，在JavaScript中，只有函数会引入新的作用域。因此，对于在函数中引用的变量，要确定它是在哪里定义的，可从最里面（当前函数）开始依次向最外面进行查找，直到找到它为止。如果在这些函数中都找不到它，则它要么是全局的，要么未定义。

问：在函数被嵌套很多层的情况下，环境的工作原理是什么呢？

答：前面解释环境时，简化了环境的表示，实际上，你可以认为每个嵌套函数都有自己的小环境，其中包含它自己的变量。这样将形成一个环境链，从内到外依次为各个嵌套函数的环境。

因此，在环境中查找变量时，你将从最近的环境着手，沿环境链不断往下查找，直到找到变量为止。如果在环境链中没有找到，再在全局环境中查找。

问：为何说词法作用域和函数环境是好东西？我原本以为前述示例代码的结果为“`Oh, don't you worry about it, I'm GLOBAL`”。在我看来，这更合乎情理。正确的答案令人迷惑，也有悖于直觉。

答：我们明白你为何这么想，但词法作用域的优点是，总是可以通过查看代码来确定变量是在哪里定义的，进而确定它的值。正如你看到的，即便在你返回函数并过后在定义函数的作用域外面调用它时，情况亦如此。

词法作用域和函数环境是好东西的另一个原因是，使用这种功能可做很多事情，稍后进行介绍。

问：形参变量是否也包含在环境中？

答：是的。前面说过，可将形参视为函数的局部变量，因此它们也包含在环境中。

问：我需要详细了解环境的工作原理吗？

答：不需要。你只需明白有关JavaScript变量的词法作用域规则即可，我们在前面已经介绍过。你现在已经知道，从函数返回的函数携带了其周边环境。

别忘了，JavaScript函数都是在定义它的环境中执行的。在函数中，要确定变量来自何方，可按从内到外的顺序依次在包含它的函数中搜索。

闭包到底是什么

人人都在谈论闭包（closure），认为它是**必不可少**的语言特性，但有多少人真正明白闭包是什么以及如何使用它们呢？少之又少。闭包是人人都想理解的语言特性，也是所有传统语言都想添加的语言特性。

闭包很难学，业界很多受过良好教育的人都这样说；但对本书的读者来说，这根本就不是问题。想知道为什么？不是因为本书通俗易懂，也不是因为我们要创建一个杀手级应用程序来向你介绍闭包，而是因为你**早就学习过了**，只是那时我们没有称之为闭包而已。

闲话少说，给闭包一个非常正式的定义吧。

如果你已深受本书的影响，此时此刻你可能这样想：这是一个会让我的薪水大幅提升的知识点。

闭包：名词，指的是函数和引用环境。

必须承认，上述定义并不能让人茅塞顿开。为何称之为**闭包**呢？简单地说，因为它可能是一道决定成败的面试题，也可能在未来的某个时候让你获得加薪——我一点都没开玩笑。

要理解闭包一词，需要明白“敲定”（close）函数的概念。



磨笔上阵

你的任务有以下两个。（1）在下面的代码中，找出所有的**自由变量**，并将它们圈出来。自由变量指的是不是在本地图作用域内定义的变量。（2）从右边选择一个可敲定这个函数的环境。可敲定函数的环境指的是给所有自由变量都提供了值的环境。

```
function justSayin(phrase) {
  var ending = "";
  if (beingFunny) {
    ending = " -- I'm just sayin!";
  } else if (notSoMuch) {
    ending = " -- Not so much.";
  }
  alert(phrase + ending);
}
```

将这些代码中的所有自由变量都圈出来。自由变量不是在本地图作用域内定义的变量。

```
beingFunny = true;
notSoMuch = false;
inConversationWith = "Paul";
```

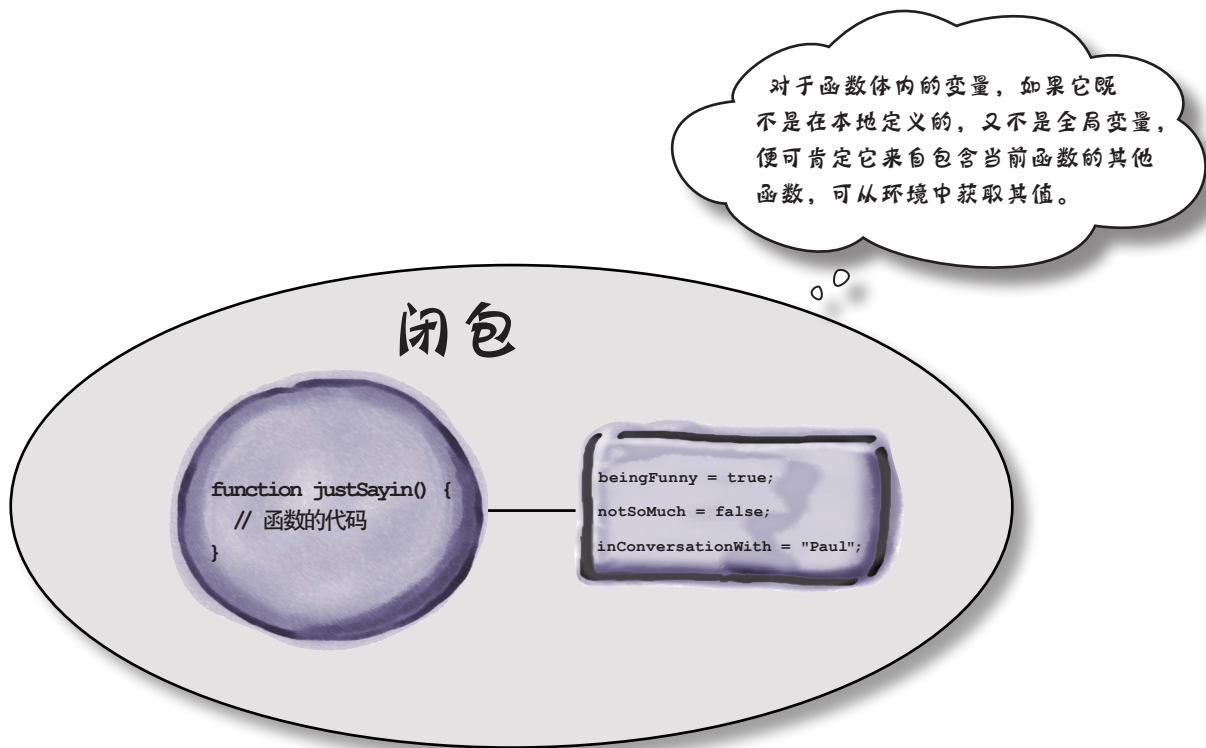
```
beingFunny = true;
justSayin = false;
oocoder = true;
```

选择一个能够将这个函数敲定的环境。

```
notSoMuch = true;
phrase = "Do do da";
band = "Police";
```

敲定函数

通过前面的练习，你可能明白了，函数通常包含局部变量（它们是在函数体中定义的，包括所有的形参），还可能包含不是在本地图定义的变量，这些变量被称为自由变量。自由一词源于这样一点：在函数体内，自由变量没有绑定到任何值（换言之，它们不是在本地图声明的）。有了给每个自由变量都提供了值的环境后，便将函数敲定了；而函数和环境一起被称为闭包。



包含自由变量的函数与为所有这些自由变量提供了变量绑定的环境一起，被称为闭包。

你已经花了将近10页的篇幅探讨这个主题，还打不打算介绍实用的JavaScript代码呀？换句话说，你要没完没了地介绍理论吗？我干嘛要关心函数的这些底层工作原理？我只要能够编写并调用函数不就成了吗？

要不是闭包这么有用，我完全同意你的看法。很抱歉强行让你去学习闭包，不过我向你保证，这样做非常值得。你知道，闭包绝非理论性的函数式编程语言结构，而是一款极其强大的编程工具。现在你明白了闭包的工作原理（毫不夸张地说，明白闭包绝对会让你在经理和同事面前脱颖而出），该学习如何使用它们了。

现实情况是，闭包无处不在；事实上，你将对它们了如指掌，进而在代码中大量地使用它们。下面来介绍一些闭包示例，帮助你明白前面讲述的所有内容。



使用闭包实现神奇的计数器

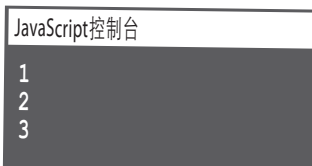
有过实现计数器函数的念头吗？通常以类似于下面的方式来实现：

```
var count = 0; // 声明一个全局变量count。

function counter() {
  count = count + 1; // 每次调用counter时，都将全局变量count加1，再返回结果。
  return count;
}
```

可以像下面这样使用这个counter函数：

```
console.log(counter()); // 因此可以像这样计数并显示计数器的值。
console.log(counter());
console.log(counter());
```



这种做法存在的唯一问题是，使用全局变量count，协作开发代码时，大家常常会使用相同的变量名，进而导致冲突。

这里要告诉你的是，可以使用受保护的局部变量实现计数器。这样，计数器将不会与任何代码发生冲突，且只能通过调用相应的函数（也叫闭包）来增加计数器的值。

要使用闭包实现这种计数器，可重用前面的大部分代码。别眨眼，奇迹就要发生了：

```
function makeCounter() { // 我们在函数makeCounter中声明变量count。这样它就是局部变量，而不是全局变量。
  var count = 0;

  function counter() { // 接下来，我们创建函数counter，它将变量count加1。
    count = count + 1;
    return count;
  }
  return counter; // 然后，返回函数counter。
} // 这是一个闭包，在其环境中存储了变量count。
```

这种办法管用吗？咱们来试一试。



测试神奇的计数器

添加一些测试计数器的代码，再试一试！

```
function makeCounter() {
  var count = 0;

  function counter() {
    count = count + 1;
    return count;
  }
  return counter;
}

var doCount = makeCounter();
console.log(doCount());
console.log(doCount());
console.log(doCount());
```

JavaScript控制台

```
1
2
3
```

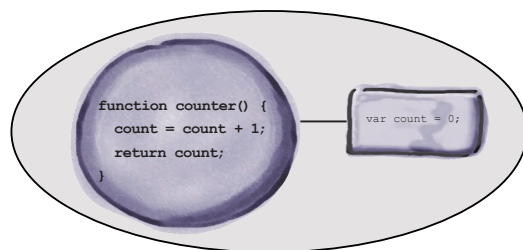
这个计数器确实管用，计数结果准确无误。

揭秘

下面来分步剖析这些代码，看看这个计数器是如何工作的。

- 1 我们调用makeCounter，它创建函数counter，并将其与包含自由变量count的环境一起返回。换句话说，它创建了一个闭包。从makeCounter返回的函数存储在doCount中。
- 2 我们调用函数doCount，这将执行函数counter的函数体。
- 3 遇到变量count时，我们在环境中查找并获取它的值。我们将count的值加1，将结果存回到环境中，再将结果返回到调用doCount的地方。
- 4 每次调用doCount时，我们都重复第2~3步。

调用doCount（指向函数counter的引用），进而需要获取count的值时，我们使用这个闭包的环境中的变量count。在外部世界（全局作用域中），根本看不到变量count，但每次调用doCount时，我们都可以使用它。除了调用doCount外，没有其他任何办法能够获取count。



```
function makeCounter() {
  var count = 0;

  function counter() {
    count = count + 1;
    return count;
  }
  return counter;
}
```

这是一个闭包。

```
1 var doCount = makeCounter();
2 console.log(doCount());
3 console.log(doCount());
4 console.log(doCount());
```

调用makeCounter时，我们获得的是一个闭包：一个函数及其环境。



轮到你了。请尝试创建下面的闭包。我们知道这对初学者来说不容易，必要时请参阅本章末尾的答案。重要的是不断地思考这些示例，直到完全明白为止。

第一题（10分）

`makePassword`将实际密码作为参数，它返回一个这样的函数：将猜测的密码作为参数，并在猜测的密码与实际密码相同时返回`true`。（对于这些有关闭包的描述，可能需要读几遍才能明白。）

```
function makePassword(password) {  
    return _____ {  
        return (passwordGuess === password);  
    };  
}
```

第二题（20分）

函数`multN`将一个数字（`n`）作为参数，并返回一个这样的函数：将一个数字作为参数，并返回它与`n`的乘积。

```
function multN(n) {  
    return _____ {  
        return _____;  
    };  
}
```

第三题（30分）

修改前面的计数器，让`makeCounter`与以前一样不接受任何参数，但使其定义变量`count`，再创建并返回一个对象。这个对象包含一个方法`increment`，它将变量`count`加1，再返回这个变量。

通过将函数表达式用作实参来创建闭包

并非只能通过从函数返回函数来创建闭包。如果函数使用了自由变量，则每当你在创建该函数的上下文外面执行它时，都将创建一个闭包。

将函数传递给函数时，也将创建闭包。在这种情况下，传递的函数将在完全不同于定义它的上下文中执行。下面是一个这样的示例。



```
function makeTimer(doneMessage, n) {
    setTimeout(function() {
        alert(doneMessage);
    }, n);
}

makeTimer("Cookies are done!", 1000);
```

我们定义了一个函数。

它使用了一个自由变量。

我们将它用作setTimeout的处理程序。

这个函数将在函数makeTimer调用完毕1000毫秒后执行。

这里向函数setTimeout传入了一个函数表达式，而这个函数表达式使用了自由变量doneMessage。你知道，函数表达式的结果是一个函数引用，而该函数引用将被传递给setTimeout。方法setTimeout存储该函数引用（这是一个函数及其环境，即闭包），并在1000毫秒后调用它。

再说一遍，我们传递给setTimeout的函数是一个闭包，因为它带有将自由变量doneMessage绑定到字符串"Cookies are done!"的环境。

考考你的脑力

如果将前面的代码改成下面这样，结果将如何？

```
function handler() {
    alert(doneMessage);
}

function makeTimer(doneMessage, n) {
    setTimeout(handler, n);
}

makeTimer("Cookies are done!", 1000);
```

考考你的脑力²

对于第9章中第412页的代码，你能使用闭包对其进行修改，避免向setTimeout传递第三个参数吗？

闭包包含的是实际环境，而非环境的副本

学习闭包时，大家常常错误地认为闭包的环境包含所有变量及其值的副本。实际上不是这样的。环境引用的是实时变量，因此，如果闭包函数外面的代码修改了变量，闭包函数执行时看到的将是变量的新值。

为明白这一点，下面来修改前面的示例。

```
function setTimer(doneMessage, n) {  
  
    setTimeout(function() {  
        alert(doneMessage);  
    }, n);  
  
    doneMessage = "OUCH!";  
}  
setTimer("Cookies are done!", 1000);
```

← 在这里创建了闭包。

← 调用setTimeout后再修改doneMessage的值。

- 1 调用setTimeout并向它传递一个函数表达式时，将创建一个闭包，其中包含相应的函数以及指向环境的引用。

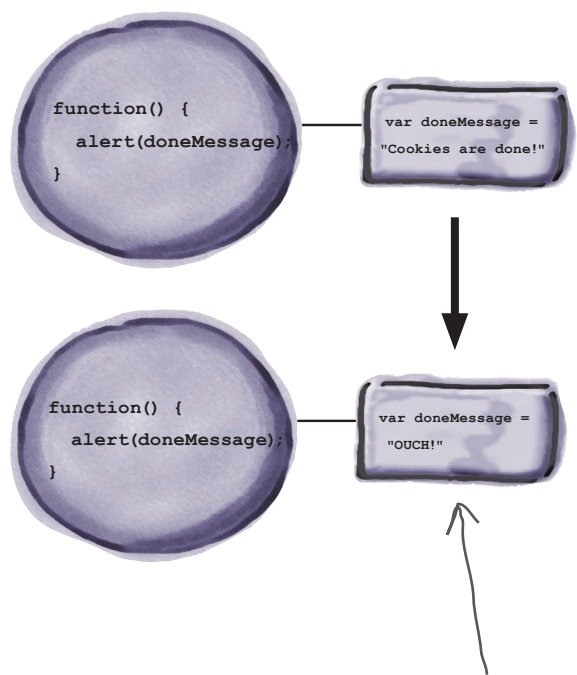
```
setTimeout(function() {  
    alert(doneMessage);  
}, n);
```

- 2 接下来，我们在闭包外面将doneMessage的值改为"OUCH!"。在闭包使用的环境中，这个变量的值也发生了变化。

```
doneMessage = "OUCH!";
```

- 3 1000毫秒后，闭包中的函数被调用。这个函数引用变量doneMessage，而在环境中，这个变量被设置成了"OUCH!"，因此提示框中显示的是"OUCH!"。

```
function() { alert(doneMessage); }
```



这个函数被调用时，它使用环境中变量doneMessage的值，即在setTimer中给这个变量设置的新值。

使用事件处理程序来创建闭包

来看另一种创建闭包的方式：使用事件处理程序来创建闭包。这种做法在JavaScript代码中很常见。我们将首先创建一个简单的网页，它包含一个按钮和一个显示消息的<div>元素。我们将跟踪用户单击按钮的次数，并在<div>元素中显示出来。

下面是用于创建该网页的HTML和一些CSS。请新建一个文件，将其命名为divClosure.html，并在其中输入这些代码。

```

<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Click me!</title>
<style>
    body, button { margin: 10px; }
    div { padding: 10px; }
</style>
<script>
    // 在这里添加JavaScript代码
</script>
</head>
<body>
    <button id="clickme">Click me!</button>
    <div id="message"></div>
</body>
</html>

```

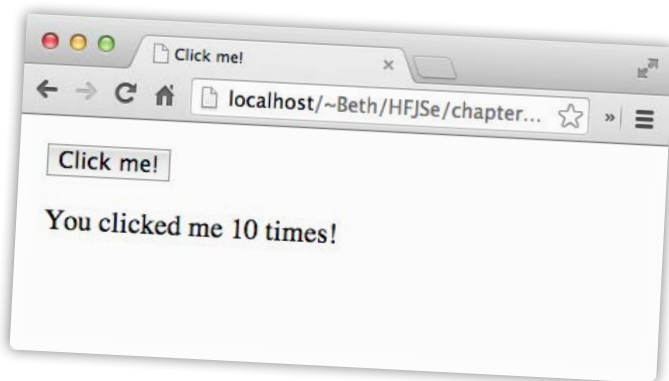
典型的简单网页。

一些设置网页元素样式的CSS。

将在这里添加我们的代码。

这个网页中有一个按钮和一个显示消息的<div>元素，每当用户单击按钮时，我们都将更新<div>元素显示的消息。

这就是我们要实现的效果：每当用户单击按钮时，<div>元素显示的消息都将更新，以指出用户单击按钮的次数。



下面来编写JavaScript代码。你完全可以不使用闭包，但正如你将看到的，通过使用闭包，代码将更简洁，甚至效率也更高一些。

不使用闭包

先来看看在不使用闭包的情况下如何实现这个示例。

```
var count = 0;

window.onload = function() {
    var button = document.getElementById("clickme");
    button.onclick = handleClick;
};

function handleClick() {
    var message = "You clicked me ";
    var div = document.getElementById("message");
    count++;
    div.innerHTML = message + count + " times!";
}
```

必须将变量`count`声明为全局的，因为如果将其作为`handleClick`（按钮的单击事件处理程序，参见后面的代码）的局部变量，则用户每次单击时，它都将被初始化。

在加载事件处理程序中获取按钮元素，并通过其属性`onclick`指定一个单击事件处理程序。

这是按钮的单击事件处理程序。

我们定义变量`message`……

获取网页中的`<div>`元素……

将计数器加1……

并使用指出用户单击按钮次数的消息更新`<div>`元素。

使用闭包

不使用闭包的版本看起来合情合理，只是其中的全局变量可能会带来麻烦。下面来使用闭包重写这些代码，看看结果有何不同。重写后的代码如下，稍后我们将对其进行测试，更深入地探讨。

```
window.onload = function() {
    var count = 0;
    var message = "You clicked me ";
    var div = document.getElementById("message");

    var button = document.getElementById("clickme");
    button.onclick = function() {
        count++;
        div.innerHTML = message + count + " times!";
    };
};
```

现在，所有变量都是在赋给`window.onload`的函数中定义的，消除了名称冲突的问题。

我们通过将一个函数表达式赋给按钮的属性`onclick`来指定单击处理程序，因此可以在这个单击处理程序中引用`div`、`message`和`count`。（别忘了词法作用域！）

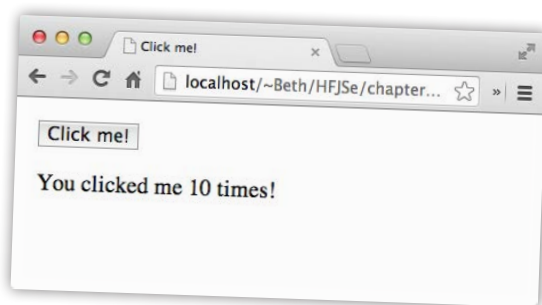
这个函数有三个自由变量：`div`、`message`和`count`。因此将创建一个闭包。也就是说，将把一个闭包赋给按钮的属性`onclick`。

测试按钮单击计数器

将上述代码添加到divClosure.html，并对其进行测试吧。加载这个网页，再单击按钮来增加计数器的数值，<div>元素显示的消息将更新。再来看一眼代码，确保你完全明白其中的工作原理。然后翻到下一页，我们将详细探讨这些代码。

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Click me!</title>
<style>
    body, button { margin: 10px; }
    div { padding: 10px; }
</style>
<script>
    window.onload = function() {
        var count = 0;
        var message = "You clicked me ";
        var div = document.getElementById("message");

        var button = document.getElementById("clickme");
        button.onclick = function() {
            count++;
            div.innerHTML = message + count + " times!";
        };
    };
</script>
</head>
<body>
    <button id="clickme">Click me!</button>
    <div id="message"></div>
</body>
</html>
```



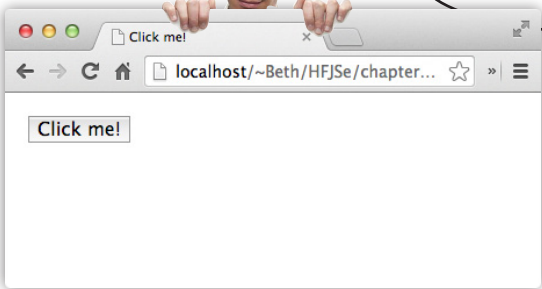
← 这是我们的测试效果。

← 将divClosure.html修改成这样。

用作按钮单击处理程序的闭包的工作原理

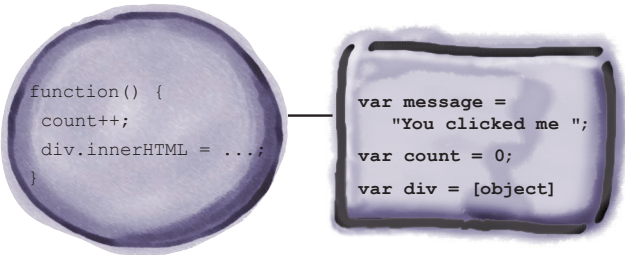
为理解这个闭包的工作原理，我们再次跟随浏览器，看看它是如何执行这些代码的。

网页已加载完毕，可以运行加载事件处理程序了。我需要定义一些变量……哦，我遇到了一个函数表达式。在这个函数表达式中，引用了三个自由变量，还是创建一个闭包好了。



```
window.onload = function() {  
    var count = 0;  
    var message = "You clicked me ";  
    var div = document.getElementById("message");  
  
    var button = document.getElementById("clickme");  
    button.onclick = function() {  
        count++;  
        div.innerHTML = message + count + " times!";  
    };  
};
```

浏览器为赋给属性button.onclick的函数创建一个闭包，其中的环境包含变量div、message和count。



现在，我将这个闭包赋给网页中按钮Click me!的属性onclick。至此，加载事件处理程序就执行完毕了，我只需静待用户单击这个按钮即可。



用户单击了这个按钮！该执行我前面存储的单击事件处理程序了……

虽然变量button已不复存在（加载事件处理程序执行完毕后，它就消失了），但按钮对象还在DOM中，其属性onclick存储了我们的闭包。

我发现有一个闭包。太好了，这意味着我可以在环境中查找自由变量的值。

```
function() {
  count++;
  div.innerHTML = ...;
}
```

```
var message =
  "You clicked me ";
var count = 0;
var div = [object]
```

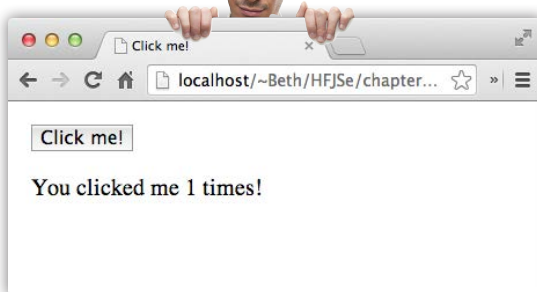
```
function() {
  count++;
  div.innerHTML = ...;
}
```

```
var message =
  "You clicked me ";
var count = 1;
var div = [object]
```

请注意，闭包中的变量div存储了一个对象。在加载事件处理程序中初始化div时，我们将document.getElementById返回的对象存储到了变量div中，因此我们不用再次从DOM中获取这个对象。这节省了一点计算时间，让代码的速度更快。

这个闭包将一直存在，直到你关闭网页为止。它已准备就绪，每当用户单击按钮时就会行动起来！

我已将变量count加1，并在环境中更新了这个变量的值。我还更新了网页显示的消息，现在要做的就是静候用户再次单击。



严峻的JavaScript挑战

我们需要一位闭包专家，听说你就是！知道闭包的工作原理后，你现在明白下述两个代码片段的結果都是008的原因吗？为理解这一点，请在下面写出闭包的环境中存储的变量。请注意，环境完全可以空的。答案见本章末尾。

代码片段 #1

```
var secret = "007";

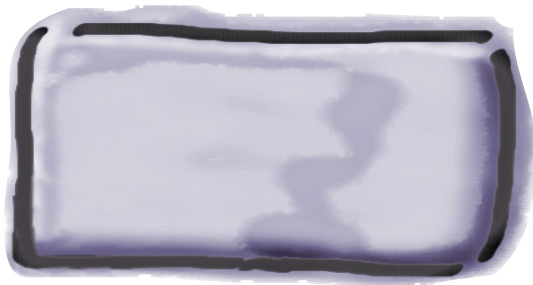
function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }

    return getValue();
}

getSecret();
```

环境



代码片段 #2

```
var secret = "007";

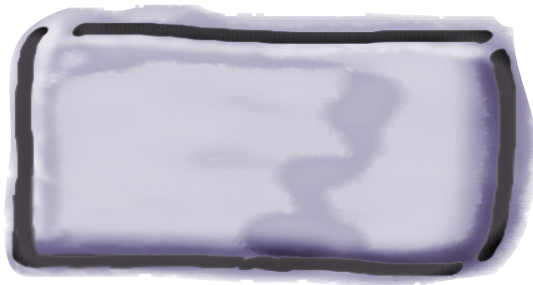
function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }

    return getValue;
}

var getValueFun = getSecret();
getValueFun();
```

环境





磨笔上阵

请看下面的代码：

```
(function(food) {  
    if (food === "cookies") {  
        alert("More please");  
    } else if (food === "cake") {  
        alert("Yum yum");  
    }  
})( "cookies");
```

← 在需要使用函数引用的地方使用函数表达式，将代码简化到极致。

你的任务不仅是明白这段代码是做什么的，还要明白它是**怎么做**的。为此，你需要进行还原，即将其中的匿名函数提取出来，将其赋给一个变量，再在原本使用函数表达式的地方使用这个变量。这样处理后，代码是不是更容易理解了？它到底是做什么的呢？



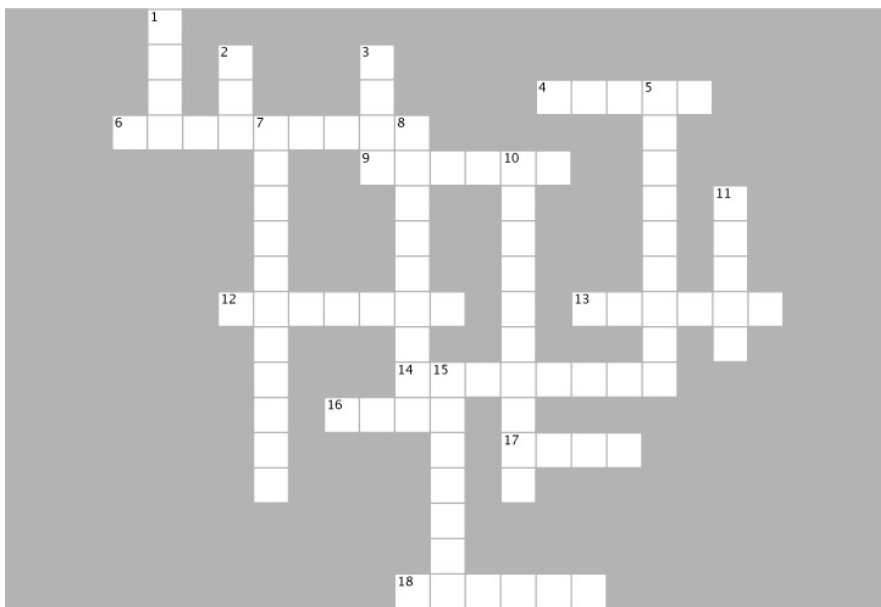
要点

- **匿名函数**是没有名称的函数表达式。
- 使用匿名函数可让代码更简洁。
- **函数声明**创建的函数是在执行其他代码前定义的。
- **函数表达式**是在运行阶段与其他代码一起执行的，因此在函数表达式所在的语句执行前，它创建的函数是未定义的。
- 可将函数表达式传递给函数，还可从函数返回函数表达式。
- 函数表达式的结果是一个**函数引用**，因此在可以使用函数引用的任何地方，都可使用函数表达式。
- **嵌套函数**是在其他函数中定义的函数。
- 与局部变量一样，嵌套函数的作用域也是局部的。
- **词法作用域**意味着通过阅读代码就能确定变量的作用域。
- 在嵌套函数中，为确定变量的值，将在最近的外部函数中查找；如果没有找到，再在全局作用域中查找。
- **闭包**指的是函数及其引用的环境。
- 闭包捕获其创建时所处作用域内的变量的值。
- **自由变量**指的是在函数体内未绑定的变量。
- 在创建闭包的上下文外部执行它时，自由变量的值由引用的环境决定。
- 通常使用闭包来为事件处理程序捕获状态。



JavaScript填字游戏

请完成下面的填字游戏，将本章介绍的JavaScript知识深深地刻在脑海中。



横向

4. 嵌套在函数中的函数声明的作用域。
6. 在fly定义前试图调用它时出现的错误。
9. wingFlapper是一个_____函数。
12. 本章使用了setTimeout来创建一个制作什么的定时器？
13. 在代码顶层被赋给变量的函数表达式拥有_____作用域。
14. 要升职加薪，你必须明白什么的工作原理？
16. 不是在当前作用域内定义的变量。
17. 在本章中，在闭包外面将变量doneMessage的值修改成了什么？
18. 为所有自由变量都提供了值的环境可以_____函数。

纵向

1. 在本章中，每次都说对了的人。
2. 在本章中，一页之间就换了衬衫的人。
3. 本章提及的一部电影，其中使用了“消失”(derezzed)一词。
5. 没有名称的函数表达式。
7. 函数和与之相关联的什么称为闭包？
8. 函数表达式的结果是一个函数_____。
10. 设置小甜饼制作定时器时，我们向setTimeout传递了一个函数_____。
11. 形参是_____变量，因此包含在定义变量的环境中。
15. _____作用域意味着通过查看代码的结构就能确定变量的作用域。



在下面的代码中，有几个地方都可使用匿名函数。请尽可能地使用匿名函数重新编写这些代码。你可将旧代码删除，并写入新代码。你还需完成另一个任务：将这些代码中的匿名函数圈出来。答案如下。

```
window.onload = init;

var cookies = {
  instructions: "Preheat oven to 350...",
  bake: function(time) {
    console.log("Baking the cookies.");
    setTimeout(done, time);
  }
};

function init() {
  var button = document.getElementById("bake");
  button.onclick = handleButton;
}

function handleButton() {
  console.log("Time to bake the cookies.");
  cookies.bake(2500);
}

function done() {
  alert("Cookies are ready, take them out to cool.");
  console.log("Cooling the cookies.");
  var cool = function() {
    alert("Cookies are cool, time to eat!");
  };
  setTimeout(cool, 1000);
}
```

我们重新编写了上述代码，使用了两个匿名函数表达式：一个对应于函数init，另一个对应于函数handleButton。

```

window.onload = function() {
    var button = document.getElementById("bake");
    button.onclick = function() {
        console.log("Time to bake the cookies.");
        cookies.bake(2500);
    };
};

var cookies = {
    instructions: "Preheat oven to 350...",
    bake: function(time) {
        console.log("Baking the cookies.");
        setTimeout(done, time);
    }
};

function done() {
    alert("Cookies are ready, take them out to cool.");
    console.log("Cooling the cookies.");
    var cool = function() {
        alert("Cookies are cool, time to eat!");
    };
    setTimeout(cool, 1000);
}

```

我们将一个函数表达式赋给了属性 window.onload……

并将另一个函数表达式赋给了属性 button.onclick。

如果你发现了可以像这样直接将函数 cool 传递给 setTimeout，就可以加分。

```

setTimeout(function() {
    alert("Cookies are cool, time to eat!");
}, 1000);

```



为确保你理解了将匿名函数表达式传递给函数的语法，请对下述将变量（这里是 vaccine）用作实参的代码进行修改，将匿名函数表达式用作实参。答案如下。

```
administer(patient, function(dosage) {
  if (dosage > 0) {
    inject(dosage);
  }
}, time);
```

请注意，用作实参的函数表达式完全可以横跨多行。不过，这样做时一定要注意语法，因为很容易出错。



轮到你了。请尝试创建下面的闭包。我们知道这对初学者来说不容易，必要时请参阅本章末尾的答案。重要的是不断地思考这些示例，直到完全明白为止。

答案如下。

第一题（10分）

makePassword将实际密码作为参数，它返回一个这样的函数：将猜测的密码作为参数，并在猜测的密码与实际密码相同时返回true。（对于这些有关闭包的描述，可能需要读几遍才能明白。）

```
makePassword(password) {
  return function guess(passwordGuess) {
    return (passwordGuess === password);
  };
}

var tryGuess = makePassword("secret");
console.log("Guessing 'nope': " + tryGuess("nope"));
console.log("Guessing 'secret': " + tryGuess("secret"));
```

从makePassword返回的函数是一个闭包，其环境中包含自由变量password。

我们向makePassword传入了值“secret”，因此在闭包的环境中存储的是这个值。

注意到这里使用了一个带名称的函数表达式！并非必须这样做，但这提供了一种引用内部函数的便利途径。请注意，调用返回的函数时，必须使用tryGuess，而不能使用guess。

调用tryGuess时，将对传入的单词（“nope”或“secret”）与环境中变量password的值进行比较。

转下一页



轮到你了。请尝试创建下面的闭包。我们知道这对初学者来说不容易，必要时请参阅本章末尾的答案。重要的是不断地思考这些示例，直到完全明白为止。

答案如下（接上一页）。

第二题（20分）

函数`multN`将一个数字（`n`）作为参数，并返回一个这样的函数：将一个数字作为参数，并返回它与`n`的乘积。

```
function multN(n) {
  return function multBy(m) {
    return n*m;
  };
}
var multBy3 = multN(3);
console.log("Multiplying 2: " + multBy3(2));
console.log("Multiplying 3: " + multBy3(3));
```

从`multN`返回的函数是一个闭包，其环境包含自由变量`n`。

因此调用`multN(3)`时，将返回一个把传递给它的数字与3相乘的函数。

第三题（30分）

修改前面的计数器，让`makeCounter`与以前一样不接受任何参数，但使其定义变量`count`，再创建并返回一个对象。这个对象包含一个方法`increment`，它将变量`count`加1，再返回这个变量。

```
function makeCounter() {
  var count = 0;
  return {
    increment: function() {
      count++;
      return count;
    }
  };
}
var counter = makeCounter();
console.log(counter.increment());
console.log(counter.increment());
console.log(counter.increment());
```

这个函数与前面的`makeCounter`类似，只是它返回一个包含方法`increment`的对象，而不是直接返回一个函数。

方法`increment`包含一个自由变量`count`，因此它是一个闭包，其环境中包含变量`count`。

这里调用`makeCounter`，获得一个对象，它包含一个为闭包的方法。

我们像往常那样调用这个方法。当我们这样做时，这个方法将引用其环境中的变量`count`。

磨笔上阵 答案

请根据你已有的函数和变量知识，判断下述哪些说法是正确的。答案如下。

- ✓ 变量handler存储了一个函数引用。
- ✓ 将handler赋给window.onload时，实际上是将一个函数引用赋给了它。
- ✓ 变量handler仅用于将其存储的函数引用赋给window.onload。
- ✓ 我们绝不会再使用handler，因为其代码仅在网页加载完毕后执行一次。
- ✓ 调用onload处理程序两次是个馊主意，因为这种处理程序通常用于为整个网页做一些初始化工作，多次调用它可能引发问题。
- ✓ 函数表达式创建函数引用。
- ✓ 我们是不是说过，将handler赋给window.onload时，实际上是将一个函数引用赋给了它？

磨笔上阵 答案

你的任务有以下两个。（1）在下面的代码中，找出所有的自由变量，并将它们圈出来。自由变量指的是并非在本地作用域内定义的变量。（2）从右边选择一个可敲定这个函数的环境。可敲定函数的环境指的是给所有自由变量都提供了值的环境。答案如下。

```
function justSayin(phrase) {
  var ending = "";
  if (beingFunny) {
    ending = " -- I'm just sayin!";
  } else if (notSoMuch) {
    ending = " -- Not so much.";
  }
  alert(phrase + ending);
}
```

将这些代码中的所有自由变量都圈出来。自由变量不是在本地图作用域内定义的变量。

这个环境给两个自由变量beingFunny和notSoMuch都提供了值。

选择一个能够将这个函数敲定的环境。

```
beingFunny = true;
notSoMuch = false;
inConversationWith = "Paul";
```

```
beingFunny = true;
justSayin = false;
oocoder = true;
```

```
notSoMuch = true;
phrase = "Do do da";
band = "Police";
```

严峻的JavaScript挑战

我们需要一位闭包专家，听说你就是！知道闭包的工作原理后，你现在明白下述两个代码片段的结果都是008的原因吗？为搞明白这一点，请在下面写出闭包的环境中存储的变量。请注意，环境完全可以是空的。答案如下。

代码片段 #1

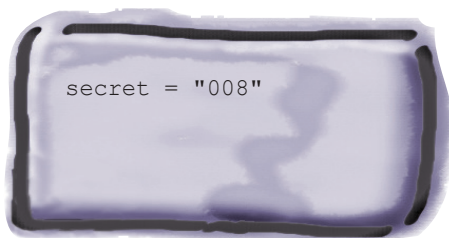
```
var secret = "007";
```

```
function getSecret() {
  var secret = "008";

  function getValue() {
    return secret;
  }

  return getValue();
}
getSecret();
```

环境



因此，它被记录到`getValue`的环境中。但我们没有从`getSecret`返回`getValue`，因此在创建这个闭包的上下文外部，根本看不到它。

代码片段 #2

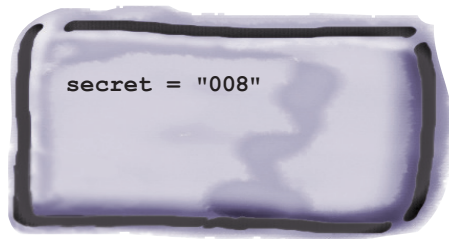
```
var secret = "007";
```

```
function getSecret() {
  var secret = "008";

  function getValue() {
    return secret;
  }

  return getValue;
}
var getValueFun = getSecret();
getValueFun();
```

环境



在这里，我们创建了一个闭包，并将其从`getSecret`返回。因此，当我们在另一个上下文（全局作用域）中调用`getValueFun`（即`getValue`）时，将使用其环境中的变量`secret`的值。



这是我们给出的答案：

```
(function(food) {
    if (food === "cookies") {
        alert("More please");
    } else if (food === "cake") {
        alert("Yum yum");
    }
})( "cookies");
```

你的任务不仅是明白这段代码是做什么的，还要明白它是怎么做的。为此，你需要进行还原，即将其中的匿名函数提取出来，将其赋给一个变量，再在原本使用函数表达式的地方使用这个变量。这样处理后，代码是不是更容易理解了？它到底是做什么的呢？

```
var eat = function(food) {
    if (food === "cookies") {
        alert("More please");
    } else if (food === "cake") {
        alert("Yum yum");
    }
};
(eat)("cookies");
```

这是提取出来的函数，我们将其赋给了变量`eat`。如果愿意，也可以使用函数声明来创建这个函数。

当然，也可将其写为`eat ("cookies")`，但这里的做法旨在演示如何用`eat`来替换前面的函数表达式。

这里所做的是对“cookies”调用`eat`，但多出来的圆括号是做什么的呢？

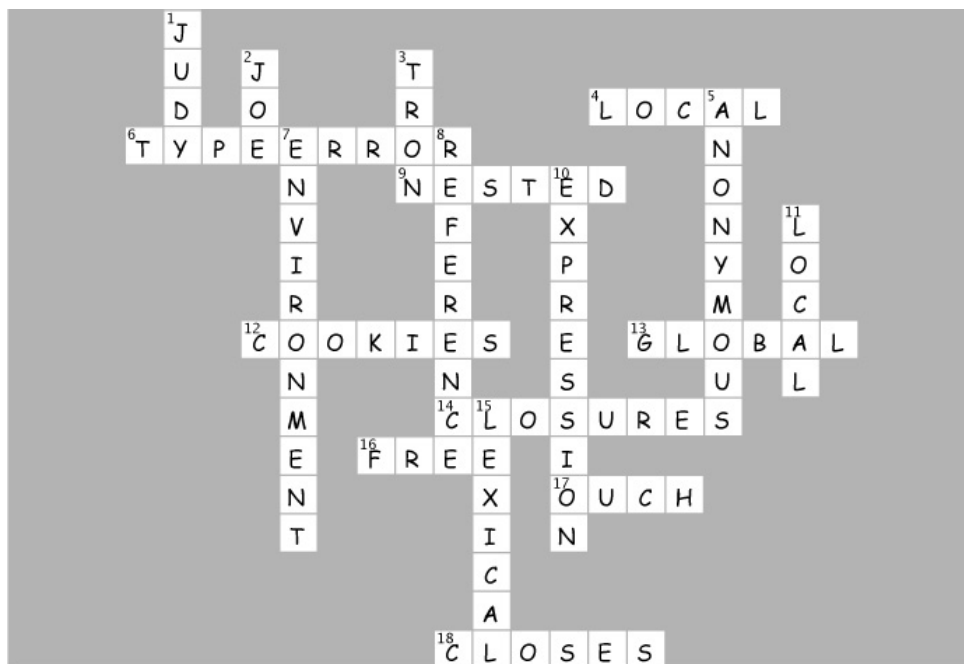
根据上述分析可知，这些代码所做的是，内嵌一个函数表达式，然后立即调用它并向它传递一个实参。

原因如下：别忘了，函数声明以关键字`function`打头，接下来是函数名，而函数表达式必须是语句的一部分。对于上述函数表达式，如果不用圆括号括起，JavaScript解释器将认为它是一个函数声明，而不是函数表达式。但调用`eat`时，不需要用圆括号将其括起，因此你可以将这些圆括号删除。

因此这些代码在一个提示框中显示消息“More please”。



JavaScript填字游戏答案



12 高级对象构造技巧

✦ 创建对象



到目前为止，我们都以手动方式创建对象：对于每个对象，都使用对象字面量来指定其所有属性。小规模地创建对象时，这没有问题，但要编写正式的代码，需要使用更好的方式。这正是对象构造函数的用武之地。使用构造函数，可更轻松地创建对象，还可让所有对象都采用相同的设计蓝图；也就是说，使用构造函数可确保所有对象都包含相同的属性和方法。通过使用构造函数，编写的对象代码将简洁得多，而且创建大量对象时不容易出错。阅读完本章后，你谈论起构造函数，就会像是在对象镇长大的一样。

使用对象字面量创建对象

到目前为止，本书都是使用对象字面量来创建对象的。使用对象字面量创建对象时，你逐个地指定其属性，如下所示：

```
var taxi = {
  make: "Webville Motors",
  model: "Taxi",
  year: 1955,
  color: "yellow",
  passengers: 4,
  convertible: false,
  mileage: 281341,
  started: false,

  start: function() { this.started = true;},
  stop: function() { this.started = false;},
  drive: function {
    // 方法drive的代码
  }
};
```



使用对象字面量创建对象时，你在花括号内列出对象的各个部分。将各部分都列出后，就得到了一个JavaScript对象。你通常将其赋给一个变量，供以后使用。

对象字面量提供了一个便利方式，让你能够在代码中随时随地创建对象，但需要创建大量对象（如一个出租车车队）时，你不想输入数百个对象字面量，不是吗？

考考你的脑力

想想看，使用对象字面量创建大量出租车对象时，还可能导致哪些问题？

- ☐ 大量输入让你手抽筋。
- ☐ 你能确保所有出租车都包含相同的属性吗？如果出现输入错误或遗漏了某个属性呢？
- ☐ 大量的对象字面量意味着大量的代码，这会不会延长浏览器的下载时间呢？
- ☐ 必须在很多地方复制方法start、stop和drive的代码。
- ☐ 要增删属性或修改方法start和stop的工作方式时，必须对所有的出租车对象进行相应的修改。
- ☐ 有了打车软件，谁还需要出租车？

按约定创建对象

到目前为止，我们创建对象时都遵守约定。例如，我们将属性和方法放在一起，并称之为“汽车对象”或“小狗对象”。它们成为对象的唯一原因是遵守了约定。

小规模地创建对象时，这种做法也许可行，但需要创建大量对象，或者有大量开发人员参与开发而他们不完全了解或遵守约定时，这种做法就行不通了。

耳听为虚，眼见为实。来看看本书前面所谓的汽车对象。

这个绝对是我们一直处理的汽车对象，它包含约定的所有属性和方法。



```
var taxi = {
  make: "Webville Motors",
  model: "Taxi",
  year: 1955,
  color: "yellow",
  passengers: 4,
  convertible: false,
  mileage: 281341,
  started: false,

  start: function() {
    this.started = true;
  },

  stop: function() {
    this.started = false;
  }
};
```

这个对象与其他汽车对象很像，但它有一个thrust方法。真不敢确定它就是汽车对象。



```
var rocketCar = {
  make: "Galaxy",
  model: "4000",
  year: 2001,
  color: "white",
  passengers: 6,
  convertible: false,
  mileage: 60191919,
  started: false,

  start: function() {
    this.started = true;
  },

  stop: function() {
    this.started = false;
  },

  drive: function() {
    // 方法drive的代码
  },

  thrust: function(amount) {
    // 方法thrust的代码
  }
};
```

这看起来是一个不错的汽车对象，但它缺少一些基本属性，如mileage和color。它还有一些额外的属性。这可能是个问题。



```
var toyCar = {
  make: "Mattel",
  model: "PeeWee",
  color: "blue",
  type: "wind up",
  price: "2.99"
};
```

这可能是一个汽车对象，但与其他汽车对象一点都不像。它有属性make、model和color，但看起来像玩具，而不是汽车。它怎么会在这里呢？



```
var tbird = {
  make: "Ford",
  model: "Thunderbird",
  year: 1957,
  passengers: 4,
  convertible: true,
  started: false,
  oilLevel: 1.0,

  start: function() {
    if (oilLevel > .75) {
      this.started = true;
    }
  },

  stop: function() {
    this.started = false;
  },

  drive: function() {
    // 方法drive的代码
  }
};
```

如果有一种像饼干模具的对象创建方式，可帮助创建基本结构相同的对象就好了。这样，创建的所有对象都将看起来一样，它们包含在一个地方定义的所有属性和方法。但我知道，这不过是白日做梦。

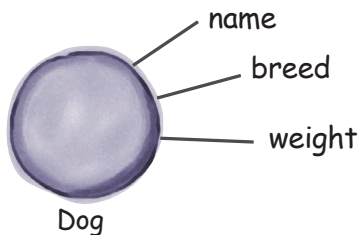


对象构造函数简介

对象构造函数（简称为构造函数）让你能够更好地创建对象。构造函数犹如一个小型工厂，能够创建无数类似的对象。

从代码的角度看，构造函数很像返回对象的函数：定义后，每当需要创建新对象时都可调用它。但正如你将看到的，构造函数并非这里说得那么简单。

要明白构造函数的工作原理，最佳方式是创建一个。我们以本书前面的小狗对象为例，编写一个构造函数，让我们能够根据需要创建任意数量的小狗对象。下面是本书前面使用的小狗对象，它包含属性name、breed和weight。



要使用对象字面量创建这样的小狗对象，代码将类似于下面这样：

```
var dog = {
  name: "Fido",
  breed: "Mixed",
  weight: 38
};
```

使用对象字面量创建的一个简单的小狗对象。现在我们需要想出创建大量小狗对象的办法。

但我们要做的不是创建一个名为Fido的小狗对象，而是找到一种办法，创建任何包含名称、品种和体重的小狗对象。为此，我们将编写一些代码，它类似于函数，同时在语法上又有点像对象。

听了前面的介绍，你心里肯定痒痒的。请翻到下一页，看看如何创建这样的构造函数。

对象构造函数和函数关系紧密，学习如何编写和使用构造函数时务必牢记这一点。

在我看来，构造函数就是个弗兰肯斯坦的怪物，由函数和对象的特征拼凑而成。还有比这更神奇的吗？



要知道其中的原因，请翻到下一页。

如何创建构造函数



要使用构造函数，可以分两步走：先定义一个构造函数，再使用它来创建对象。下面先来介绍如何创建构造函数。

我们需要的是一个可用于创建小狗对象的构造函数，更具体地说是用于创建包含名称、品种和体重的小狗对象。因此，我们定义一个被称为构造函数的函数，它知道如何创建小狗对象，如下所示：

注意到我们给构造函数命名时，采用了首字母大写的方式。并非必须这样做，但人人都将此视为一种约定加以遵守。

构造函数与常规函数看起来没什么两样。

这个函数的形参对应于要给每个小狗对象提供的属性。

```
function Dog(name, breed, weight) {  
  this.name = name;  
  this.breed = breed;  
  this.weight = weight;  
}
```

这部分看起来更像对象，我们将每个形参都赋给了看起来像属性的东西。

属性名和形参名不必相同，但通常相同——这也是约定。

注意到这个构造函数什么都没有返回。

与大多数函数中不同，我们没有使用局部变量，而是使用关键字黑色。（到目前为止，我们只在对象中这样做过。）

请接着往下看。接下来我们将介绍如何使用这个构造函数，届时这样做的原因将一目了然。



磨笔上阵

我们需要你的帮助。我们以前一直使用对象字面量来创建鸭子对象。利用刚学到的知识，你能否为我们编写一个创建鸭子对象的构造函数？下面是我们使用的一个对象字面量，你可以据此来编写构造函数：

```
var duck = {  
  type: "redheaded",  
  canFly: true  
}
```

一个鸭子对象字面量。

编写一个创建鸭子对象的构造函数。

备注：我们知道你还没有完全明白构造函数的工作原理，现在请暂时将重点放在语法上。

如何使用构造函数

前面说过，要使用构造函数，可分两步走：先创建一个构造函数，再使用它。前面创建了构造函数Dog，下面就来使用它吧，如下所示：

为创建小狗对象，我们使用了运算符new。
再调用构造函数Dog。
然后指定实参。
请将下面的话大声读出来：为创建fido，我创建一个名称为Fido、品种为Mixed、体重为38磅的小狗对象。

```
var fido = new Dog("Fido", "Mixed", 38);
```

要创建一个名称为Fido、品种为Mixed、体重为38磅的小狗对象，我们首先指定关键字new，再使用合适的实参调用构造函数Dog。这条语句执行完毕后，变量fido将包含一个指向新小狗对象的引用。

有了构造函数Dog后，我们就可以不断地创建小狗对象：

```
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
```

相比于使用对象字面量，这种对象创建方式是不是更容易些？通过这种方式创建小狗对象，可确保每个小狗对象都包含相同的属性：name、breed和weight。



```
function Dog(name, breed, weight) {
  this.name = name;
  this.breed = breed;
  this.weight = weight;
}
var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var dogs = [fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
  var size = "small";
  if (dogs[i].weight > 10) {
    size = "large";
  }
  console.log("Dog: " + dogs[i].name
    + " is a " + size
    + " " + dogs[i].breed);
}
```

下面来做一个简单的练习，帮助你完全消化这些知识。请将左边的代码加入一个网页中，再进行测试，并将输出记录在下面。



构造函数的工作原理

幕后花絮



你知道了如何声明构造函数以及如何使用它来创建对象，但还需看看幕后的情况，以了解构造函数的工作原理。要明白构造函数的工作原理，关键在于了解运算符new都做了些什么。

先来看看前面用来创建对象fido的语句：

```
var fido = new Dog("Fido", "Mixed", 38);
```

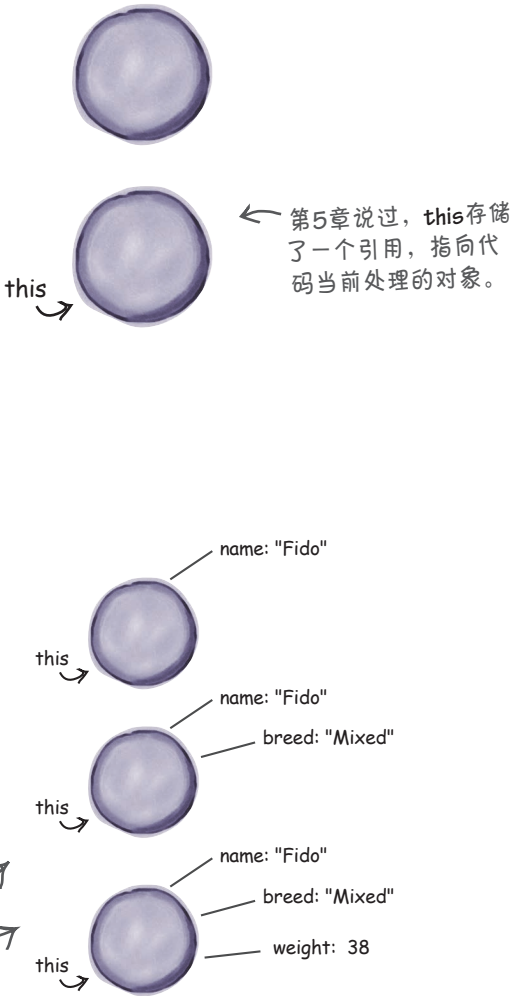
请看赋值运算符的右边，所有的操作都是在这里进行的。我们来跟踪一下其执行过程。

- ① new首先创建一个新的空对象。
- ② 接下来，new设置this，使其指向这个新对象。
- ③ 设置this后，调用函数Dog，并将"Fido"、"Mixed"和38作为实参传递给它。

```
      "Fido"      "Mixed"      38
       |           |           |
       v           v           v
function Dog(name, breed, weight) {
  this.name = name;
  this.breed = breed;
  this.weight = weight;
}
```

- ④ 接下来，执行这个函数的代码。与大多数构造函数一样，Dog给新创建的this对象的属性赋值。

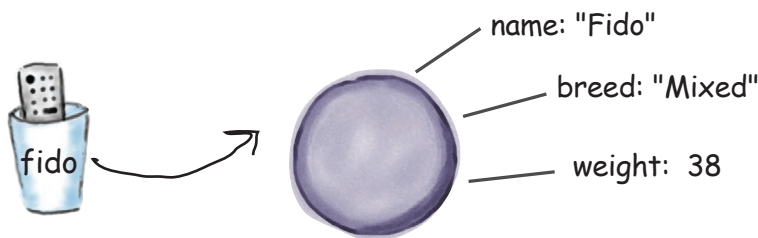
通过执行Dog函数的代码，将相应的形参赋给三个属性，从而对新对象进行定制。



幕后花絮



- ⑤ 最后，Dog函数执行完毕后，运算符new返回this——指向新创建的对象引用。请注意，它会自动为你返回this，你无需在代码中显式地返回。指向新对象的引用被返回后，我们将其赋给变量fido。



变身浏览器



下面的JavaScript代码存在一些错误，你的任务是变身浏览器，将这些错误找出来。完成这个练习后，翻到本章末尾，看看你是否将所有的错误都找出来了。顺便说一句，这是第12章。如果你愿意，也可在其中添加注释。你早就有这样的权利了。

```
function widget(partNo, size) {
    var this.no = partNo;
    var this.breed = size;
}

function FormFactor(material, widget) {
    this.material = material,
    this.widget = widget,
    return this;
}

var widgetA = widget(100, "large");
var widgetB = new widget(101, "small");
var formFactorA = new FormFactor("plastic", widgetA);
var formFactorB = new ForumFactor("metal", widgetB);
```

还能在构造函数中定义方法

构造函数Dog创建的小狗对象与本书前面的小狗对象类似，只是不会叫，因为它们没有bark方法。这个问题很容易解决，因为在构造函数中，除了能给属性指定值外，还可以定义方法。下面就来扩展前面的代码，以添加bark方法：

顺便说一句，在对象中，方法也是属性，只是将函数赋给了这种属性。

```
function Dog(name, breed, weight) {
  this.name = name;
  this.breed = breed;
  this.weight = weight;
  this.bark = function() {
    if (this.weight > 25) {
      alert(this.name + " says Woof!");
    } else {
      alert(this.name + " says Yip!");
    }
  };
}
```

要添加bark方法，只需将一个函数（这里是一个匿名函数）赋给属性this.bark。

现在，每个小狗对象都有bark方法，你可以调用它。

请注意，与以前创建的其他所有对象一样，我们使用this来表示当前对象。

快速测试bark方法

构造函数介绍得差不多了，我们将上述代码添加到一个HTML页面中，再在该页面中添加如下测试代码：

```
var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var dogs = [fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
  dogs[i].bark();
}
```

确认小狗对象像预期的那样叫。





有人给我们提供了一个点咖啡的构造函数，但没有定义方法。我们需要方法`getSize`，它根据加入的咖啡量返回一个字符串：

- 咖啡量为8盎司^①时返回`small`；
- 咖啡量为12盎司时返回`medium`；
- 咖啡量为16盎司时返回`large`。

我们还需要方法`toString`，它返回一个字符串，指出你点的是哪种咖啡，如"`You've ordered a small House Blend coffee.`"。

请补全下面的代码，再在浏览器中进行测试。尝试点几杯大小各异的咖啡。查看本章末尾的答案，再继续往下阅读。

```
function Coffee(roast, ounces) {
    this.roast = roast;
    this.ounces = ounces;
```

← 编写在这个构造函数中定义上述两个方法的代码。

```
}
```

```
var houseBlend = new Coffee("House Blend", 12);
console.log(houseBlend.toString());
```

```
var darkRoast = new Coffee("Dark Roast", 16);
console.log(darkRoast.toString());
```

这是我们得到的输出。你得到的输出应与此类似。

JavaScript控制台

```
You've ordered a medium House Blend coffee.
You've ordered a large Dark Roast coffee.
```

①盎司为英制重量单位。1盎司 $\approx$ 28.35克。——编者注



世上没有愚蠢的问题

问：构造函数名的首字母为何要大写？

答：这是JavaScript开发人员遵守的一种约定，让开发人员能够一眼就看出哪些函数是构造函数，哪些函数是常规函数。为什么要区分这一点呢？因为调用构造函数时，需要使用运算符new。一般而言，将构造函数名的首字母大写能够让代码阅读者更容易识别它们。

问：也就是说，构造函数与常规函数没什么两样，只是前者设置this对象的属性？

答：如果你说的是计算方面，答案是肯定的。在常规函数中可以做的任何事情，在构造函数中都可以做，如声明和使用变量、使用for循环、调用其他函数等；但有一件事情不能做，那就是从构造函数中返回值，因为除非返回的是this，否则这将导致构造函数不返回它创建的对象。

问：构造函数的形参必须与属性同名吗？

答：不是，可使用任何形参名。形参仅用于存储要赋给对象的属性，以定制对象的值。重要的是给对象的属性指定的名称。虽然如此，出于提高清晰度的考虑，常常让形参与属性同名。这样只需查看构造函数的定义，就知道要给哪些属性赋值。

问：使用构造函数创建的对象与使用对象字面量创建的对象没什么两样，对吗？

答：在不涉及下一章将讨论的更高级的对象设计时，是这样的。

问：为何需要使用new来创建对象？不是可以在常规函数中创建并返回对象吗？（就像第5章的makeCar所做的那样。）

答：是的，可以这样创建对象，但正如前面所说，使用new时会执行一些额外的操作，这将在本章后面和第13章介绍。

问：我对构造函数中的this还是感到有点迷惑。我们使用this来给对象的属性赋值，还在对象的方法中使用this，它们指的是一回事吗？

答：调用构造函数来创建对象时，this被设置为一个引用，指向正在创建的新对象，因此构造函数的所有代码针对的都是这个新对象。

对象创建后，当你对其调用方法时，this被设置为方法被调用的对象。因此，在方法中，this总是表示方法被调用的对象。

问：使用构造函数创建对象是否胜过使用对象字面量创建对象？

答：这两种方式都有其用武之地。需要创建大量包含相同属性和方法的对象时，构造函数很有帮助。使用构造函数很方便，可重用代码，还可确保对象的一致性。

但有时候，需要创建一次性的简单对象。在这种情况下，使用对象字面量显得更简洁、更具表达力。

因此，使用哪种方式完全取决于你的需要。它们都是很不错的对象创建方式。

↖
稍后你将看到一个这样的示例。



危险地带

使用构造函数时，有一点需要特别小心：千万别忘了使用关键字new。你可能很容易忘记这样做，因为构造函数也是函数，不使用关键字new也能够调用它。但调用构造函数时如果忘记了使用new，可能导致代码出现难以找出的bug。来看看忘记使用关键字new时可能发生的情况：

```
function Album(title, artist, year) {
  this.title = title;
  this.artist = artist;
  this.year = year;
  this.play = function() {
    // 其他代码
  };
}
```

← 这个构造函数看起来一点问题没有。

↓ 这也许没有关系，因为Album也是一个函数。

↓ 忘记使用new!
var darkside = Album("Dark Side of the Cheese", "Pink Mouse", 1971);
darkside.play();

← 尝试调用方法play。麻烦来了！

Uncaught TypeError: Cannot call method 'play' of undefined

安检清单

请阅读下面的检查清单，了解为何会发生这种情况。

- ❑ 别忘了，new首先创建一个新对象，将其赋给this，再调用构造函数。如果忘记使用new，根本就不会创建新对象。
- ❑ 这意味着在构造函数中，this指向的不是新的唱片对象，而是表示应用程序的全局对象。
- ❑ 如果忘记使用new，就不会有对象从构造函数返回。这意味着没有可供赋给变量darkside的对象，因此darkside是未定义的。正是因为这一点，我们试图调用方法play时出现了错误，指出方法被调用的对象未定义。

这个对象是存储全局变量的顶级对象，在浏览器中为window对象。



如果你使用构造函数来创建对象，而且在试图引用这些对象时总是发现它们是未定义的，请检查你的代码，确认调用构造函数时使用了关键字new。

00



当装有液体的开口试管在昂贵的笔记本电脑上方倾斜时，也请三思而行！



起底构造函数

本周访谈：认识关键字`new`

Head First: `new`，你躲到哪里去了，怎么到第12章才看到你？

new: 当前，依然有很多脚本没有用到我；还有人在没有理解的情况下使用我。

Head First: 为什么会这样？

new: 因为很多脚本编写人员只使用对象字面量，或者直接复制并粘贴使用我的代码，而不明白我的工作原理。

Head First: 确实如此。对象字面量很方便，我自己也不太明白该在什么时候以及如何使用你。

new: 没错，我是一种高级功能。毕竟，要明白如何使用我，必须先明白对象的工作原理、函数的工作原理、`this`的工作原理等。只有掌握了大量的知识后，你才会考虑了解我！

Head First: 你能简单地介绍一下自己吗？我们的读者熟悉对象、函数和`this`，如果能够激发出他们学习你的动力，那就太好了。

new: 让我想一想。是这样的：我是一个与构造函数一起用来创建对象的运算符。

Head First: 我不想让你扫兴，但这样的简介好像不是最佳的。

new: 你饶了我吧，我只是一个运算符，又不是公关高手。

Head First: 算了，你的简介确实提出了几个问题。首先，你是个运算符？

new: 没错，我是个运算符。将我放在函数调用的前面，就能带来翻天覆地的变化。运算符对操作数进行操作。就我而言，我只操作一种操作数，那就是函数调用。

Head First: 那你是如何操作的呢？

new: 首先，我创建一个新对象。人人都以为这是构造函数做的，但实际上是我做的。真是吃力不讨好。

Head First: 请接着说。

new: 接下来，我调用构造函数，并确保在构造函数的函数体内，关键字`this`指向了我创建的新对象。

Head First: 你为何要这样做？

new: 让构造函数中的语句能够引用这个对象。毕竟，给这个对象添加属性和方法才是构造函数的全部意义所在。使用构造函数来创建小狗和汽车等对象时，你希望这些对象包含一些属性，对不对？

Head First: 没错。接下来呢？

new: 接下来，我确保从构造函数返回创建的新对象。这提供了极大的便利，让开发人员不用显式地返回它。

Head First: 听起来确实非常方便。既然如此，为何有人在学习了你后还使用对象字面量呢？

new: 我和对象字面量交情匪浅。这个家伙很了不起，需要快速创建对象时，我也会毫不犹豫地使用它。但需要创建大量类似的对象，确保对象能够利用代码重用，需要确保对象一致以及支持一些高级对象用法时，就要用到我了。

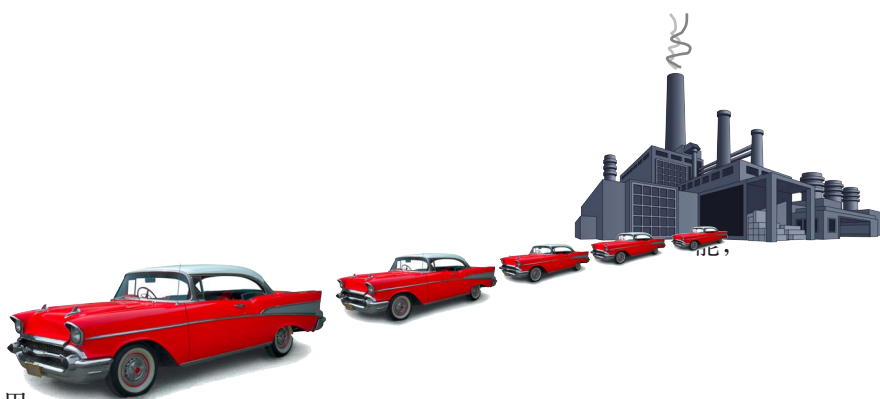
Head First: 更高级的用法？快说说！

new: 还是别分散读者的注意力，到下一章再介绍吧。

Head First: 我想我先得再看一遍采访！

生产时间到了

还好你及时学到了对象构建技
因为我们刚接到一个很大的汽
车订单，没法手工制作。为按
时完成订单，需要使用一个构造
函数。为此，我们将以本书前面使用
的汽车对象字面量为蓝本，创建一个生产汽车的
构造函数。



下面是我们需要制造的各种汽车，注意到我们擅自让每款汽车
都包含相同的属性和方法。当前，我们没有考虑每种汽车的独特之处，
也没有考虑玩具车和火箭车，这些情况将在后面处理。下面来查看这些汽车对象字面量，并据此
创建一个构造函数，以便能够使用它来创建汽车对象，用于表示包含这些属性和方法的各种汽车。



```
var chevy = {
  make: "Chevy",
  model: "Bel Air",
  year: 1957,
  color: "red",
  passengers: 2,
  convertible: false,
  mileage: 1021,
  started: false,

  start: function() {
    this.started = true;
  },

  stop: function() {
    this.started = false;
  },

  drive: function() {
    if (this.started) {
      console.log(this.make + " " +
        this.model + " goes zoom zoom!");
    } else {
      console.log("Start the engine first.");
    }
  }
};
```



```
var cadil = {
  make: "GM",
  model: "Cadillac",
  year: 1955,
  color: "tan",
  passengers: 5,
  convertible: false,
  mileage: 12892,
  started: false,
  start: function() {...},
  stop: function() {...},
  drive: function() {...}
};
```



```
var fiat = {
  make: "Fiat",
  model: "500",
  year: 1957,
  color: "Medium Blue",
  passengers: 2,
  convertible: false,
  mileage: 88000,
  started: false,
  start: function() {...},
  stop: function() {...},
  drive: function() {...}
};
```



```
var taxi = {
  make: "Webville Motors",
  model: "Taxi",
  year: 1955,
  color: "yellow",
  passengers: 4,
  convertible: false,
  mileage: 281341,
  started: false,
  start: function() {...},
  stop: function() {...},
  drive: function() {...}
};
```



请利用你学到的知识创建一个Car构造函数。建议你采取如下步骤。

- ① 首先，指定关键字`function`（这项工作我们已经替你完成了）和构造函数名。再指定形参：对于每个要为其提供初始值的属性，都需要一个形参。
- ② 接下来，给对象的每个属性设置初始值（务必结合使用属性名和关键字`this`）。
- ③ 最后，添加三个方法：`start`、`drive`和`stop`。

```
function _____(_____){
```

← 在这里编写构造函数的代码。

```
}
```

继续往下阅读前，务必查看本章末尾的答案，看看你编写的代码是否正确。

试驾一些新车



有了大批量生产汽车的方法后，咱们来生成一些汽车，并测试一下。为此，首先在一个HTML页面中添加构造函数Car，再添加一些测试代码。

请注意：要这样做，你必须先完成前一页的练习。 😊

这是我们使用的代码，请随意修改和扩展。

我们首先使用这个构造函数创建第5章涉及的所有汽车对象。

```
var chevy = new Car("Chevy", "Bel Air", 1957, "red", 2, false, 1021);
var cadi = new Car("GM", "Cadillac", 1955, "tan", 5, false, 12892);
var taxi = new Car("Webville Motors", "Taxi", 1955, "yellow", 4, false, 281341);
var fiat = new Car("Fiat", "500", 1957, "Medium Blue", 2, false, 88000);
```

为何就此止步呢？

```
var testCar = new Car("Webville Motors", "Test Car", 2014, "marine", 2, true, 21);
```

创建本书使用的试驾车！ →



使用构造函数创建新对象非常容易，你注意到了吗？下面来试驾一下这些汽车。

你也可以创建自己喜欢的汽车或虚构的汽车。

```
var cars = [chevy, cadi, taxi, fiat, testCar];

for(var i = 0; i < cars.length; i++) {
    cars[i].start();
    cars[i].drive();
    cars[i].drive();
    cars[i].stop();
}
```

这是我们得到的输出。你添加了自己的汽车吗？请尝试调整方法的调用顺序（如在调用方法start前调用方法drive）。你还可以将方法drive的调用次数设置为随机的。

JavaScript控制台

```
Chevy Bel Air goes zoom zoom!
Chevy Bel Air goes zoom zoom!
GM Cadillac goes zoom zoom!
GM Cadillac goes zoom zoom!
Webville Motors Taxi goes zoom zoom!
Webville Motors Taxi goes zoom zoom!
Fiat 500 goes zoom zoom!
Fiat 500 goes zoom zoom!
Webville Motors Test Car goes zoom zoom!
Webville Motors Test Car goes zoom zoom!
```

不要将对象字面量弃若敝屣

前面简要地比较了对象构造函数和对象字面量，并指出了对象字面量依然很有用，但你没有看到这方面的示例。下面对构造函数Car进行细微的修改，让你知道在什么情况下通过使用一些对象字面量，可让代码更整洁，使其更易于阅读和维护。

再来看一眼构造函数Car，看看如何使其代码更整洁。

注意到我们使用了大量的形参——总共7个。

添加的形参越多（随着需求的增长，总是需要这样做），这些代码阅读起来越困难。



```
function Car(make, model, year, color, passengers, convertible, mileage) {
    this.make = make;
    this.model = model;
    this.year = year;
    this.color = color;
    this.passengers = passengers;
    this.convertible = convertible;
    this.mileage = mileage;
    this.started = false;

    this.start = function() {
        this.started = true;
    };
    // 其他方法
}
```



编写调用这个构造函数的代码时，必须按正确的顺序指定所有实参。

这里的问题是，构造函数Car包含大量的形参，难以阅读和维护。另外，调用这个构造函数的代码编写起来也比较困难。这看似只是小小的不便，但可能导致的bug比你想象得多；不仅如此，这些bug还常常难以发现。

这些bug为何难以发现呢？因为如果你颠倒了两个实参的顺序，代码从语义上看像是正确的，却不能正确地运行。

传递大量实参时，可使用一种适用于任何函数的常见技巧——无论它是常规函数还是构造函数。这种技巧的工作原理如下：将所有实参都放到一个对象字面量中，再将这个对象字面量传递给函数。这将通过一个容器（对象字面量）传递所有的值，从而不必操心实参与形参的顺序问题。

另外，如果遗漏了一个实参，将引发连锁错误！

为明白其中的工作原理，下面来重新编写调用构造函数Car的代码，再稍微调整一下这个构造函数的代码。

用一个对象字面量替代所有实参

对于调用构造函数Car的代码，用一个对象字面量替代所有的实参：



只需将所有实参都存储到对象字面量的恰当属性中即可。我们使用了构造函数使用的属性名。

```
var cadi = new Car("GM", "Cadillac", 1955, "tan", 5, false, 12892);
```

```
var cadiParams = {make: "GM",  
                  model: "Cadillac",  
                  year: 1955,  
                  color: "tan",  
                  passengers: 5,  
                  convertible: false,  
                  mileage: 12892};
```

这里按原来的顺序存储实参，但并非必须这样做。

因此，对于调用构造函数Car的代码，可将其重写为下面这样：

```
var cadiParams = {make: "GM",  
                  model: "Cadillac",  
                  year: 1955,  
                  color: "tan",  
                  passengers: 5,  
                  convertible: false,  
                  mileage: 12892};
```

完成了重大的转变。代码不仅更整洁，阅读起来也容易得多，至少在我们看来如此。

```
var cadi = new Car(cadiParams);
```

← 现在只需向构造函数Car传递一个实参。

还没完，因为构造函数Car本身依然要求向它传递7个实参，而不是1个对象。下面来修改构造函数Car的代码，再进行测试。

修改构造函数Car

现在，需要删除构造函数Car的所有形参，用所传入对象的属性替代。我们将把这个形参命名为params。为使用这个对象，还需对这个构造函数的代码稍作修改，如下所示：

```
var cadiParams = {make: "GM",
                  model: "Cadillac",
                  year: 1955,
                  color: "tan",
                  passengers: 5,
                  convertible: false,
                  mileage: 12892};
```

这里再次列出了前一页的对象字面量以及调用构造函数Car的代码。

```
var cadi = new Car(cadiParams);
```

首先，将构造函数Car的7个形参替换为1个，用于表示要传入的对象。

```
function Car(params) {
    this.make = params.make;
    this.model = params.model;
    this.year = params.year;
    this.color = params.color;
    this.passengers = params.passengers;
    this.convertible = params.convertible;
    this.mileage = params.mileage;
    this.started = false;

    this.start = function() {
        this.started = true;
    };
    this.stop = function() {
        this.started = false;
    };
    this.drive = function() {
        if (this.started) {
            alert("Zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    };
}
```

然后，将每个形参引用都替换为传递给函数的对象的相应属性。

在方法中，没有直接使用形参。这合乎情理，因为在方法中应总是通过关键字this使用对象的属性。因此，这些代码无需作任何修改。

试驾

请更新创建cadi和其他汽车对象的代码，并对修改后的代码进行测试。

```
cadi.start();
cadi.drive();
cadi.drive();
cadi.stop();
```



将构造函数Car和Dog复制到一个文件中，再添加下面的代码。然后运行这个文件，并将其输出记录下来。

```
var limoParams = {make: "Webville Motors",
                  model: "limo",
                  year: 1983,
                  color: "black",
                  passengers: 12,
                  convertible: true,
                  mileage: 21120};

var limo = new Car(limoParams);
var limoDog = new Dog("Rhapsody In Blue", "Poodle", 40);

console.log(limo.make + " " + limo.model + " is a " + typeof limo);
console.log(limoDog.name + " is a " + typeof limoDog);
```

构造函数Dog的代码
可在第530页找到。

← 将输出记录在这里。



假设有人给你提供了一个对象，而你想确定它是哪种类型的对象（Car、Dog或Superman），或者想判断它是否与另一个对象的类型相同，请问运算符typeof能够提供帮助吗？

世上没有愚蠢的问题

问：能再跟我说说typeof是做什么的吗？

答：运算符typeof返回其操作数的类型。如果向它传递一个字符串，它将返回"string"；如果向它传递一个对象，它将返回"object"。你可以向它传递任何类型：数字、字符串、布尔值或对象和函数等更复杂的类型。然而，typeof不会提供更具体的信息，比如指出对象是小狗对象还是汽车对象。

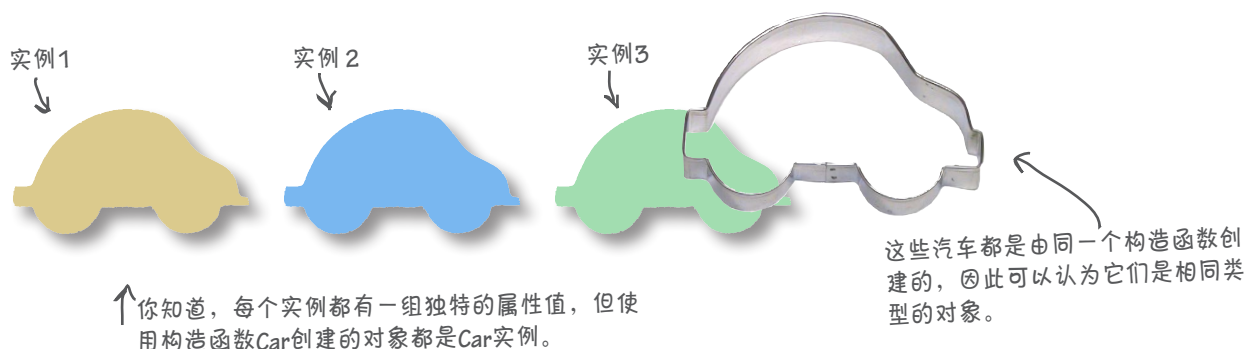
问：既然typeof不会指出对象是小狗对象还是汽车对象，如何确定对象是哪种类型的呢？

答：Java和C++等很多其他面向对象的语言都有严格的对象类型概念。在这些语言中，你可以确定对象到底是哪种类型的。但JavaScript以更动态、更宽松的方式处理对象及其类型。有鉴于此，很多开发人员认为JavaScript的对象系统不那么强大；可实际上，其对象系统更宽泛、更灵活。鉴于JavaScript的类型系统更动态，要确定对象是否是小狗或汽车更难些：这取决于你对小狗或汽车的定义。然而，有另一个运算符可提供更详细的信息。要知道是哪个运算符，请接着往下看。

理解对象实例

你无法通过观察确定JavaScript对象是哪种类型的对象，如小狗或汽车。在JavaScript中，对象是一个动态的结构，无论对象包含哪些属性和方法，其类型都是object。不过，如果知道对象是由哪个**构造函数**创建的，就能获悉一些有关它的信息。

别忘了，每当你使用运算符new调用构造函数时，都将创建一个新的对象实例。因此，如果你使用构造函数Car创建一个对象，这个对象就是汽车；更准确地说，这个对象就是一个**Car实例**。



说对象是某个构造函数的实例并非纸上谈兵，实际上，可以在代码中使用运算符instanceof来确定对象是由哪个构造函数创建的。来看一些代码：

```
var cadiParams = {make: "GM", model: "Cadillac", year: 1955, color: "tan",
  passengers: 5, convertible: false, mileage: 12892};
```

```
var cadi = new Car(cadiParams);
```

```
if (cadi instanceof Car) {
  console.log("Congrats, it's a Car!");
};
```

如果对象是由指定的构造函数创建的，运算符instanceof将返回true。

在这里，我们说对象cadi是构造函数Car创建的一个实例。

实际上，创建对象时，运算符new在幕后存储了一些信息，让你随时都能确定对象是由哪个构造函数创建的。运算符instanceof就是根据这些信息来确定对象是否是指定构造函数的实例。

情况比这里描述的要复杂些，这将在下一章讨论。





我们需要一个名为dogCatcher的函数，它在传入的对象是小狗时返回true，否则返回false。请编写这个函数，并使用下面的代码进行测试。接着往下阅读前，别忘了查看本章末尾的答案，看看你编写的代码是否正确。

```
function dogCatcher(obj) {
```

在这里编写实现函数
dogCatcher的代码。



```
}
```

这是测试代码。



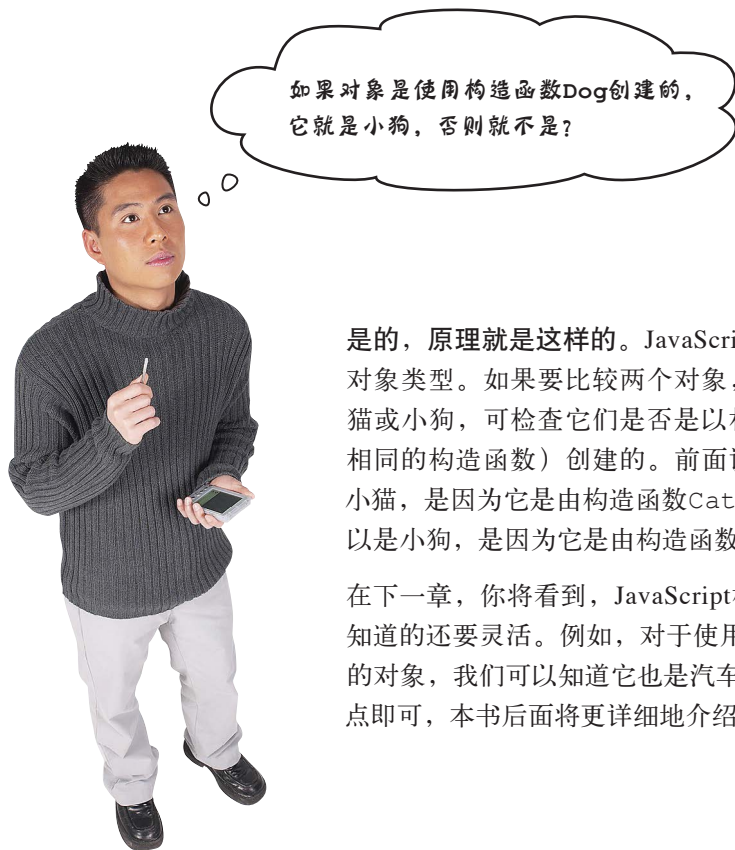
```
function Cat(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}
var meow = new Cat("Meow", "Siamese", 10);
var whiskers = new Cat("Whiskers", "Mixed", 12);

var fido = {name: "Fido", breed: "Mixed", weight: 38};

function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.bark = function() {
        if (this.weight > 25) {
            alert(this.name + " says Woof!");
        } else {
            alert(this.name + " says Yip!");
        }
    };
}
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var dogs = [meow, whiskers, fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
    if (dogCatcher(dogs[i])) {
        console.log(dogs[i].name + " is a dog!");
    }
}
```





是的，原理就是这样的。JavaScript没有严格意义上的对象类型。如果要比较两个对象，看它们是否都是小猫或小狗，可检查它们是否是以相同的方式（即使用相同的构造函数）创建的。前面说过，小猫之所以是小猫，是因为它是由构造函数Cat创建的；而小狗之所以是小狗，是因为它是由构造函数Dog创建的。

在下一章，你将看到，JavaScript构造函数和对象比你知道的还要灵活。例如，对于使用构造函数Taxi创建的对象，我们可以知道它也是汽车。现在只需了解这一点即可，本书后面将更详细地介绍。

即便是创建好的对象，也可以有独特的属性

我们花了很大的篇幅讨论如何使用构造函数来创建一致的对象，即包含相同属性和方法的对象，但没有提到这样一点：使用构造函数创建对象后，可对其进行修改，因为使用构造函数创建的对象是可以修改的。

这到底是什么意思呢？回忆一下，介绍对象字面量时我们演示过，可在对象创建后增删其属性。对于使用构造函数创建的对象，也可这样做。

使用构造函数Dog创建的小狗对象fido。

```
var fido = new Dog("Fido", "Mixed", 38);  
fido.owner = "Bob";  
delete fido.weight;
```

要给这个对象添加新属性，只需给这个新属性赋值即可。

也可使用运算符`delete`删除既有的属性。

如果愿意，还可添加新方法：

要添加新方法，只需将这个函数赋给一个新属性。

```
fido.trust = function(person) {  
    return (person === "Bob");  
};
```

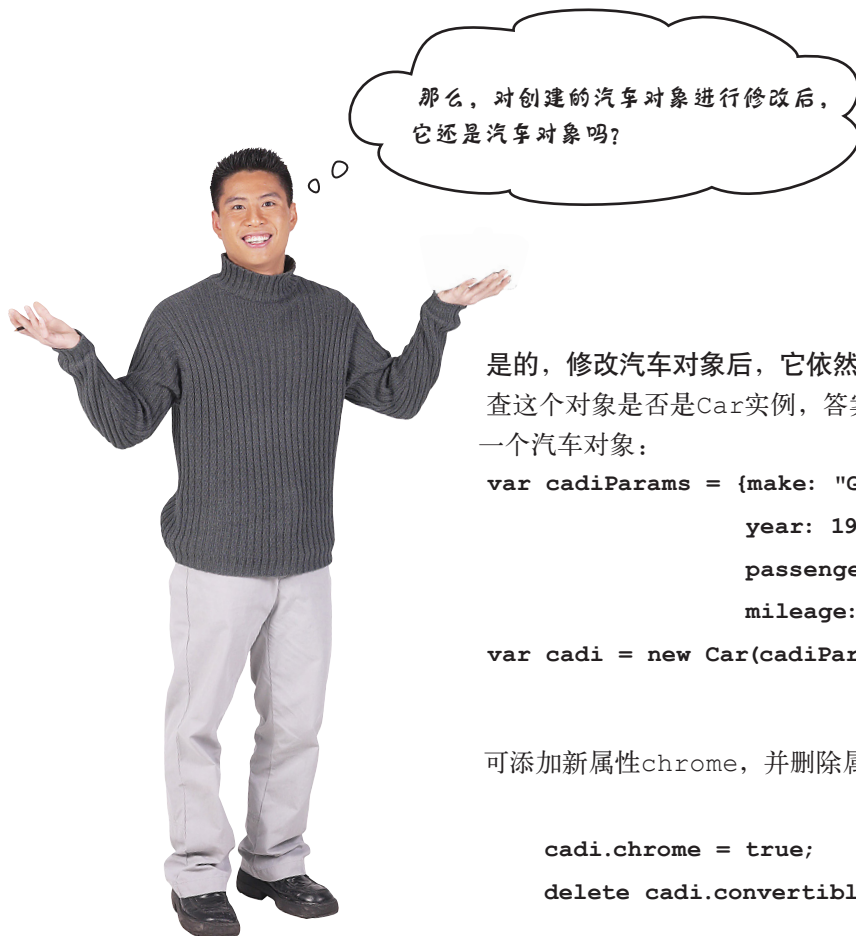
使用的是匿名函数！看到了吧，匿名函数无处不在！

请注意，这只修改了对象fido。如果给对象fido添加一个方法，那么只有fido对象有这个方法，其他小狗对象都不会包含这个方法：

```
var notBite = fido.trust("Bob");  
  
var spot = new Dog("Spot", "Chihuahua", 10);  
notBite = spot.trust("Bob");
```

这行代码没问题，因为fido对象包含方法trust。因此，notBite的值为true。

这些代码行不通，因为spot对象没有方法trust。这将引发错误如下错误：TypeError: Object #<Dog> has no method "trust"。



那么，对创建的汽车对象进行修改后，它还是汽车对象吗？

是的，修改汽车对象后，它依然是汽车对象。也就是说，如果检查这个对象是否是Car实例，答案是肯定的。例如，如果我们创建一个汽车对象：

```
var cadiParams = {make: "GM", model: "Cadillac",
                  year: 1955, color: "tan",
                  passengers: 5, convertible: false,
                  mileage: 12892};

var cadi = new Car(cadiParams);
```

可添加新属性chrome，并删除属性convertible：

```
cadi.chrome = true;
delete cadi.convertible;
```

但对象cadi依然是汽车对象：

```
cadi instanceof Car
```

← 结果为true。

前面说JavaScript的类型系统是动态的时，指的就是这个意思。

从现实意义上说，cadi还是汽车吗？如果我们删除这个对象的所有属性，它还是汽车吗？运算符instanceof说是，但从你的标准判断来看，它可能不是。

通常，你不会使用构造函数来创建一个对象，再将它修改得面目全非，看起来根本不像使用这个构造函数创建的。一般而言，你使用构造函数来创建相当一致的对象；但如果你希望对象更灵活，JavaScript也提供了这样的支持。作为代码设计人员，由你决定如何以你和同事认为合理的方式使用构造函数和对象。

这些内置对象确实能够节省时间。
现在，我每天下班都很早，回到家都能看上一段《黄金女郎》了。

内置构造函数

JavaScript自带了一系列构造函数，可用于实例化一些便利的对象，如知道如何处理日期和时间的对象、擅于在文本中查找模式的对象以及颠覆你对数组的认识的对象。你知道构造函数的工作原理，还知道关键字new的用法，完全可以使用这些构造函数——更重要的是它们创建的对象。下面简要地介绍两个这样的构造函数，为你自行探索它们作好充分准备。

下面先来看看JavaScript内置的日期对象。要创建日期对象，只需使用其构造函数即可：

```
var now = new Date();
```

← 新建一个表示当前日期和时间的日期对象。

构造函数Date返回一个表示本地当前日期和时间的Date实例。有了日期对象后，便可使用其方法来操作日期和时间，还可获取其各种属性。下面是几个这样的示例：

```
var dateString = now.toString();
```

← 返回一个表示日期的字符串，如“Thu Feb 06 2014 17:29:29 GMT-0800 (PST)”。

```
var theYear = now.getFullYear();
```

← 返回日期中的年份。

```
var theDayOfWeek = now.getDay();
```

← 返回一个数字，指出日期对象表示的是星期几，如1（表示星期一）。

通过向构造函数Date传递额外的实参，可轻松地创建表示任何日期和时间的日期对象。例如，要创建一个表示1983年5月1日的日期对象，可像下面这样做：

```
var birthday = new Date("May 1, 1983");
```

← 可像这样向构造函数传递一个表示日期的简单字符串。

还可在传入的字符串中包含时间，从而创建更具体的日期对象：

```
var birthday = new Date("May 1, 1983 08:03 pm");
```

← 在传入的字符串中包含时间。

这里只简要地介绍了日期对象，要详尽地了解其属性和方法，请参阅《JavaScript权威指南》。



数组对象

下面介绍另一个有趣的内置对象：数组对象。本书前面创建数组时，使用的都是方括号表示法（`[1, 2, 3]`），但也可以使用下面的构造函数来创建：

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

← 创建一个长度为零的空数组。

上述代码新建了一个空数组。然后，可随时像下面这样添加元素：

```
emptyArray[0] = 99;
```

← 你应该对这些代码不陌生。本书前面给数组添加元素时，都是这样做的。

还可创建特定长度的数组对象。假设要创建一个包含3个元素的数组：

```
var oddNumbers = new Array(3);
oddNumbers[0] = 1;
oddNumbers[1] = 3;
oddNumbers[2] = 5;
```

← 创建一个长度为3的数组，再用值填充它。

这里创建了一个长度为3的数组。刚创建时，数组`oddNumbers`的3个元素是未定义的，但随后给每个元素都设置了值。如果愿意，可在这个数组中轻松添加更多的元素。

你对这些都不应该感到陌生。数组对象最有趣的地方是包含一系列方法，你已经知道了数组的方法`sort`，下面是其他几个有趣的方法：

```
oddNumbers.reverse();
```

← 按相反的顺序排列数组的所有元素（现在数组`oddNumbers`依次包含5、3和1）。请注意，这个方法修改原始数组。

```
var aString = oddNumbers.join(" - ");
```

← 方法`join`将数组`oddNumbers`中的值合并成一个字符串（这里在值之间加上-），并返回这个字符串。因此，这将返回字符串“5-3-1”。

```
var areAllOdd = oddNumbers.every(function(x) {
    return ((x % 2) !== 0);
});
```

← 方法`every`将一个函数作为参数；对于数组中的每个值，都调用这个函数，并判断该函数返回的值是`true`还是`false`。如果对于所有的元素，这个函数都返回`true`，那么方法`every`将返回`true`。

这里只涉及数组对象的冰山一角，但就本书而言，这些知识足够了。同样，要全面了解对象数组，请参阅《JavaScript权威指南》。



可是，本书前面创建数组时，使用的方法截然不同。

好眼力。本书前面创建数组时，使用的都是方括号表示法（[]），这实际上是直接使用构造函数Array的简写。请看下面两种创建空数组的方式，它们是等价的：

```
var items = new Array(); ← 这两种方式等价。JavaScript语言支持方括号表示法，旨在让你能够更轻松创建数组。  
var items = [];
```

同样，如果你编写下面的代码：

```
var items = ["a", "b", "c"];
```

我们称之为数组字面量语法。

这不过是另一种使用构造函数的方式的简写：

```
var items = new Array("a", "b", "c");
```

如果传入了多个实参，创建的数组将以这些值作为元素。

使用字面量表示法和构造函数创建的对象没什么两样。无论使用哪种方式创建对象后，都可使用其方法。

你可能会问，为何要使用构造函数（而不是字面量表示法）来创建数组呢？需要创建在运行阶段确定的特定长度的数组，再在该数组中添加元素时，使用构造函数很方便，如下所示：

```
var n = getNumberOfWidgetsFromDatabase();  
var widgets = new Array(n);  
for(var i=0; i < n; i++) {  
    widgets[i] = getDatabaseRecord(i);  
}
```

这些代码可能创建一个很大的数组，其长度要等到运行阶段才知道。

创建简单数组时，使用数组字面量语法的效果很好；但创建要等到运行阶段才能确定长度的数组时，使用构造函数Array更合适。你可以根据需要使用其中任意一种方式，也可结合使用这两种方式。

其他内置对象

除了日期和数组外，JavaScript还包含很多其他的内置对象，它们有时候可能为你提供很大的便利。下面列出了其中的几个。如果你对其他内置对象感兴趣，可在网上搜索“JavaScript标准内置对象”。

Object 你可使用构造函数Object来创建对象。与数组一样，对象字面量表示法{}与new Object()等价。这将在本书后面更详细地介绍。

Math 这个对象包含用于执行数学运算任务的属性和方法，如Math.PI和Math.random()。

RegExp 使用这个构造函数可创建正则表达式对象，让你能够在文本中搜索非常复杂的模式。

Error 这个构造函数创建标准错误对象，为你在代码中捕获错误提供了极大的便利。

世上没有愚蠢的问题

问：我不太明白构造函数Date和Array的工作原理。它们好像支持零个或更多的实参。就拿Date来说吧，如果没有提供任何实参，它将返回当前日期，但也可以通过传入实参来指定其他的日期。这是如何实现的呢？

答：好眼力。可编写根据实参数量执行不同操作的函数。就拿构造函数Array来说吧，如果没有传入任何实参，它将创建一个空数组；如果传入了一个实参，它就认为该实参指定的是数组的长度；如果有多个实参，它就认为这些实参指定的是元素的初始值。

问：在自己编写的构造函数中，也可以这样做吗？

答：当然可以。这方面的知识我们没有介绍过：在每个函数中，都有一个arguments对象，其中包含传递给该函数的所有实参。你可使用这个对象来确定传入了哪些实参，进而采取相

应的措施（有关arguments对象的详细信息，请参阅附录）。另外，还可通过检查这个对象来确定哪些形参是未定义的。

问：本书前面使用了Math，但使用前为何不需要使用new Math来创建Math对象呢？

答：问得好。实际上，Math并非构造函数，甚至不是函数，而是一个对象。你知道，Math是一个内置对象，可用来获取pi的值（使用Math.PI）、生成随机数（使用Math.random）等。可将Math视为一个包含一系列属性和方法的对象字面量，可供你编写JavaScript代码时随时使用。Math的首字母之所以大写，是为了让你知道它是JavaScript内置的。

问：我知道如何检查对象是否是使用特定构造函数创建的，但如何编写代码以判断两个对象是否是使用同一个构造函数创建的呢？

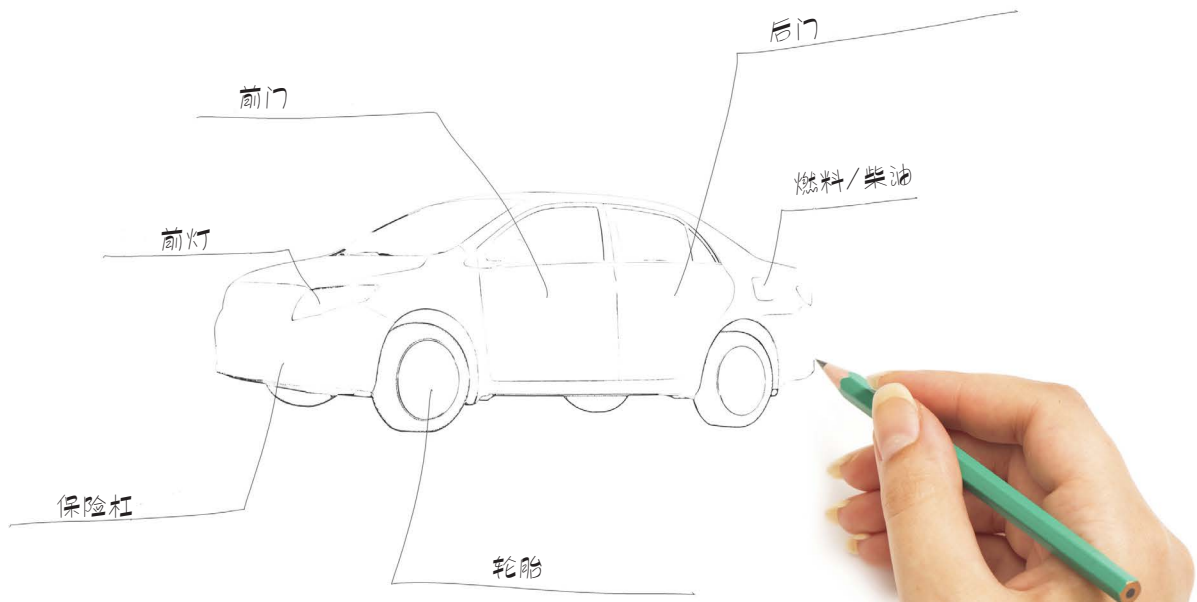
答：要判断两个对象是否是使用同一个构造函数创建的，可像下面这样做：

```
((fido instanceof Dog) &&
 (spot instanceof Dog))
```

如果这个表达式的结果为true，就说明fido和spot是使用同一个构造函数创建的。

问：如果使用对象字面量来创建对象，它将是哪个构造函数的实例呢？还是说它不是任何构造函数的实例？

答：对象字面量是Object的实例。可将Object视为用于创建最通用的JavaScript对象的构造函数。下一章将更深入地讨论Object在JavaScript对象系统中所处的位置。

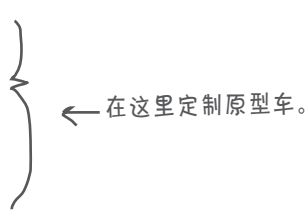


Web镇汽车制造公司根据原型车来制造所有汽车，彻底改变了汽车制造方式。原型车提供了包括发动、驾驶和停车方式在内的基本功能以及制造商和生产年份这两个属性，但其他方面都需要你去定制。希望车身是红色还是蓝色的？没问题，自己去定制好了。想装备高级立体声系统？没问题，添加就是了。

现在给你一个机会，设计一辆你心目中完美无缺的汽车。在下面创建一个CarPrototype对象，并让它成为一辆你梦寐以求的汽车。继续往下阅读前，请查看我们在本章末尾提供的设计。



```
function CarPrototype() {  
  this.make = "Webville Motors";  
  this.year = 2013;  
  this.start = function() {...};  
  this.stop = function() {...};  
  this.drive = function() {...};  
}
```



这有什么用途呢？下一章你就会知道了！顺便说一句，本章到这里就结束了，不过别忘了要点和填字游戏！



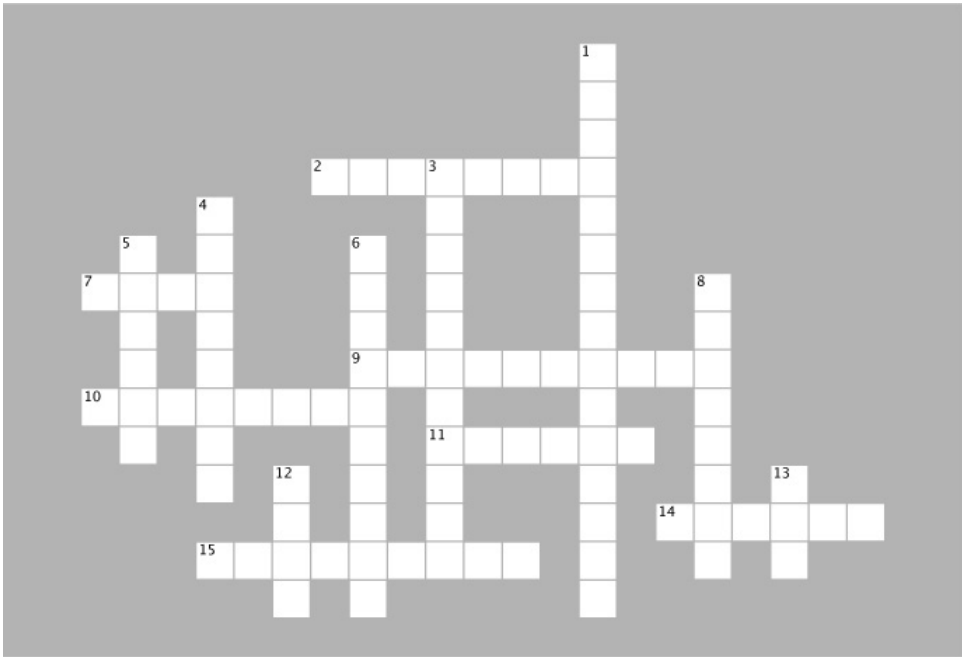
要点

- 需要创建少量对象时，适合使用对象字面量。
- 需要创建大量类似的对象时，适合使用构造函数。
- 构造函数是使用运算符`new`进行调用的函数。根据约定，将构造函数名的首字母大写。
- 使用构造函数可创建包含相同属性和方法的一致对象。
- 要创建对象，可使用运算符`new`调用构造函数。
- 使用`new`来调用构造函数时，将新建一个空对象，并在构造函数中将其赋给`this`。
- 在构造函数中，可使用`this`来访问正在创建的对象，进而给它添加属性。
- 构造函数自动返回它创建的新对象。
- 调用构造函数时，如果忘记使用`new`，将不会创建任何对象。这将导致难以调试的错误。
- 要定制对象，可向构造函数传递实参，并使用这些值来初始化要创建的对象属性。
- 如果构造函数有很多形参，应考虑将它们合并为单个对象形参。
- 要判断对象是否是使用特定构造函数创建的，可使用运算符`instanceof`。
- 与对象字面量一样，也可对使用构造函数创建的对象进行修改。
- JavaScript自带很多构造函数，可使用它们来创建很有用的对象，如日期对象、正则表达式和数组。



JavaScript填字游戏

请完成下面的填字游戏，让新学的知识深深地印在脑海中。



横向

- 2. 构造函数是什么？
- 7. 一个构造函数，想将生日存储在变量中时需要用到它。
- 9. 构造函数包含大量的_____时，可使用一个对象字面量来向它传递实参。
- 10. 使用构造函数来创建对象时，我们说这个对象是构造函数的一个什么？
- 11. 构造函数有点像制作什么的模子？
- 14. 构造函数返回新创建的什么？
- 15. 调用构造函数时，如果忘记使用new，可能出现的错误类型。

纵向

- 1. 构造函数有点像什么？
- 3. 对于使用_____创建的对象，可随时给它添加属性。
- 4. new不是公关高手，而只是一个什么？
- 5. Web镇汽车制造公司试驾车的颜色。
- 6. 使用构造函数，可让所有汽车的什么都相同？
- 8. 绝不要将什么放在笔记本电脑上方？
- 12. limo和limoDog的什么是相同的？
- 13. 使用构造函数创建对象时，需要使用哪个运算符？

磨笔上阵 答案

我们需要你的帮助。我们以前一直使用对象字面量来创建鸭子对象。利用刚学到的知识，你能否为我们编写一个创建鸭子对象的构造函数？下面是我们使用的一个对象字面量，你可以据此来编写构造函数。答案如下。

```
var duck = {
  type: "redheaded",
  canFly: true
}
```

一个鸭子对象字面量。

编写一个创建鸭子对象的构造函数。

```
function Duck(type, canFly) {
  this.type = type;
  this.canFly = canFly;
}
```

备注：我们知道你还没有完全明白构造函数的工作原理，现在请暂时将重点放在语法上。



练习 答案

```
function Dog(name, breed, weight) {
  this.name = name;
  this.breed = breed;
  this.weight = weight;
}

var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var dogs = [fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
  var size = "small";
  if (dogs[i].weight > 10) {
    size = "large";
  }
  console.log("Dog: " + dogs[i].name
    + " is a " + size
    + " " + dogs[i].breed);
}
```

下面来做一个简单的练习，帮助你完全消化这些知识。请将左边的代码加入一个网页中，再进行测试，并将输出记录在下面。

JavaScript控制台

```
Dog: Fido is a large Mixed
Dog: Fluffy is a large Poodle
Dog: Spot is a small Chihuahua
```

变身浏览器



下面的JavaScript代码存在一些错误，你的任务是变身浏览器，将这些错误找出来。答案如下。

如果要将widget用作构造函数，就需要将首字母大写。采用小写不会导致错误，但最好遵守将构造函数名首字母大写的约定。

在this前面无需使用var。这里是给对象添加属性，而不是声明新变量。

```
function widget(partNo, size) {
    var this.no = partNo;
    var this.breed = size;
}
```

另外，根据约定，通常让形参与属性同名。因此，改为this.partNo和this.size可能更合适。

这里使用的是逗号而不是分号。别忘了，在构造函数中，应使用常规语句，而不是用逗号分隔的属性名/值对。

```
function FormFactor(material, widget) {
    this.material = material,
    this.widget = widget,
    return this;
}
```

这里返回了this，这没有必要。构造函数会自动完成这项任务。这条语句不会导致错误，但它是多余的。

遗漏了new!

```
var widgetA = widget(100, "large");
var widgetB = new widget(101, "small");
var formFactorA = newFormFactor("plastic", widgetA);
var formFactorB = new ForumFactor("metal", widgetB);
```

new和构造函数名之间必须有一个空格。

构造函数名的拼写不正确。



有人给我们提供了一个点咖啡的构造函数，但没有定义方法。我们需要方法`getSize`，它根据加入的咖啡量返回一个字符串：

- 咖啡量为8盎司时返回`small`；
- 咖啡量为12盎司时返回`medium`；
- 咖啡量为16盎司时返回`large`。

我们还需要方法`toString`，它返回一个字符串，指出你点的是哪种咖啡。请补全下面的代码，再在浏览器中进行测试。尝试点几杯大小各异的咖啡。答案如下。

```
function Coffee(roast, ounces) {
    this.roast = roast;
    this.ounces = ounces;
    this.getSize = function() {
        if (this.ounces === 8) {
            return "small";
        } else if (this.ounces === 12) {
            return "medium";
        } else if (this.ounces === 16) {
            return "large";
        }
    };
    this.toString = function() {
        return "You've ordered a " + this.getSize() + " "
            + this.roast + " coffee.";
    };
}
```

别忘了，表示方法被调用的对象。因此，如果我们调用`houseBlend.size`，`this`将为对象`houseBlend`。

方法`getSize`查看对象的属性`ounces`，并返回相应的字符串。

方法`toString`返回一个描述对象的字符串。它使用方法`getSize`获取咖啡的大小。

我们创建了两个咖啡对象。调用方法`toString`，并显示它返回的字符串。

```
var houseBlend = new Coffee("House Blend", 12);
console.log(houseBlend.toString());

var darkRoast = new Coffee("Dark Roast", 16);
console.log(darkRoast.toString());
```

这是我们到的输出。你得到的输出应与此类似。

JavaScript控制台

```
You've ordered a medium House Blend coffee.
You've ordered a large Dark Roast coffee.
```



请利用你学到的知识创建一个Car构造函数。建议你采取如下步骤。

- ① 首先，指定关键字function（这项工作我们已经替你完成了）和构造函数名。再指定形参：对于每个要为其提供初始值的属性，都需要一个形参。
- ② 接下来，给对象的每个属性设置初始值（务必结合使用属性名和关键字this）。
- ③ 最后，添加三个方法：start、drive和stop。

答案如下。

构造函数名为Car。

①

```
function Car(make, model, year, color, passengers, convertible, mileage) {
```

7个形参，分别对应于7个要定制的属性。

②

```
this.make = make;
this.model = model;
this.year = year;
this.color = color;
this.passengers = passengers;
this.convertible = convertible;
this.mileage = mileage;
this.started = false;
```

在新的汽车对象中，把要根据形参定制的属性都设置为与相应的形参同名。这里按照约定，让属性和形参同名。

将属性started初始化为false。

③

```
this.start = function() {
    this.started = true;
};
this.stop = function() {
    this.started = false;
};
this.drive = function() {
    if (this.started) {
        alert("Zoom zoom!");
    } else {
        alert("You need to start the engine first.");
    }
};
}
```

这些方法与以前相同，但当它们赋给对象的属性时，使用的语法稍有不同，因为这是在构造函数而不是对象字面量中。



练习 答案

将构造函数Car和Dog复制到一个文件中，再添加下面的代码。然后运行这个文件，并将其输出记录下来。结果如下：

```
var limoParams = {make: "Webville Motors",
                  model: "limo",
                  year: 1983,
                  color: "black",
                  passengers: 12,
                  convertible: true,
                  mileage: 21120};

var limo = new Car(limoParams);
var limoDog = new Dog("Rhapsody In Blue", "Poodle", 40);

console.log(limo.make + " " + limo.model + " is a " + typeof limo);
console.log(limoDog.name + " is a " + typeof limoDog);
```

JavaScript控制台

```
Webville Motors limo is a object
Rhapsody In Blue is a object
```

← 我们得到的输出。



我们需要一个名为dogCatcher的函数，它在传入的对象是小狗时返回true，否则返回false。请编写这个函数，并使用下面的代码进行测试。答案如下：

```
function dogCatcher(obj) {
  if (obj instanceof Dog) {
    return true;
  } else {
    return false;
  }
}
```

更简洁的版本

```
function dogCatcher(obj) {
  return (obj instanceof Dog);
}
```

```
function Cat(name, breed, weight) {
  this.name = name;
  this.breed = breed;
  this.weight = weight;
}
var meow = new Cat("Meow", "Siamese", 10);
var whiskers = new Cat("Whiskers", "Mixed", 12);

var fido = {name: "Fido", breed: "Mixed", weight: 38};

function Dog(name, breed, weight) {
  this.name = name;
  this.breed = breed;
  this.weight = weight;
  this.bark = function() {
    if (this.weight > 25) {
      alert(this.name + " says Woof!");
    } else {
      alert(this.name + " says Yip!");
    }
  };
}

var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var dogs = [meow, whiskers, fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
  if (dogCatcher(dogs[i]) {
    console.log(dogs[i].name + " is a dog!");
  }
}
```

JavaScript控制台

Fluffy is a dog!
Spot is a dog!





Web镇汽车制造公司根据原型车来制造所有汽车，彻底改变了汽车制造方式。原型车提供了包括发动、驾驶和停车方式在内的基本功能以及制造商和生产年份这两个属性，但其他方面都需要你去定制。希望车身是红色还是蓝色的？没问题，自己去定制好了。想装备高级立体声系统？没问题，添加就是了。

现在给你一个机会，设计一辆你心目中完美无缺的汽车。我们的设计如下。



在这里画出你梦想的汽车。

```
var taxi = new CarPrototype();
taxi.model = "DeLorean Remake";
taxi.color = "silver";
taxi.currentTime = new Date();
taxi.fluxCapacitor = {type: "Mr. Fusion"};
taxi.timeTravel = function(date) {...};
```

← 在这里定制原型车。

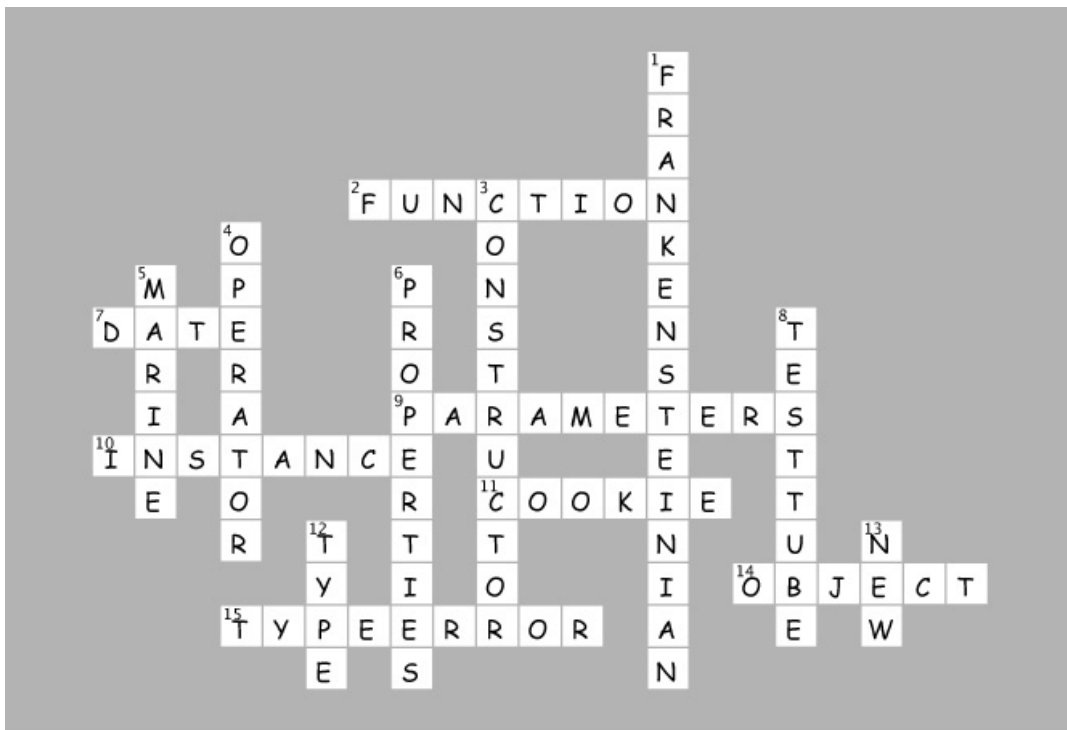


```
function CarPrototype() {
  this.make = "Webville Motors";
  this.year = 2013;
  this.start = function() {...};
  this.stop = function() {...};
  this.drive = function() {...};
}
```

这有什么用途呢？下一章你就会知道了！顺便说一句，本章到这里就结束了。



JavaScript填字游戏答案



超强的对象创建方式



学会如何创建对象仅仅是个开始。该充实一些关于对象的内容了。我们需要在对象之间建立关系和共享代码的方法，需要扩展和改进既有对象的方法。换句话说，我们需要更多的工具。在本章中你将看到，JavaScript的对象模型非常强大，但它与标准面向对象语言的对象模型稍有不同。JavaScript采用的不是基于类的面向对象系统，而是更强大的原型模型，其中的对象可继承和扩展其他对象的行为。这有何优点呢？你马上就会看到。现在就开始吧。



如果你以前使用过Java、C++或其他任何基于传统面向对象编程的语言，咱们有必要谈一谈。

如果你没有使用过，而且有个约会，也请坐下来慢慢听我讲，说不定也能学到点东西。

跟你直说吧：JavaScript没有传统的面向对象模型，即从类创建对象的模型。事实上，JavaScript根本就没有类。在JavaScript中，对象从其他对象那里继承行为，我们称之为原型式继承（prototypal inheritance）或基于原型的继承。

如果你接受过面向对象编程训练，可能对JavaScript多有抱怨，也深感困惑，但别忘了，相比于经典面向对象语言，基于原型的语言更通用。它们更灵活，效率更高，表达力更强——只要你愿意，就可使用JavaScript来实现经典继承。

因此，如果你接受过经典面向对象编程训练，请坐下来，放松心情，打开心扉，为学习一些不同的东西作好准备。如果你对“经典面向对象编程”毫无概念，就说明你是一张白纸，这通常是天大的好事。

这在未来可能发生变化：下一个JavaScript版本可能添加类。因此，请关注<http://wickedlysmart.com/hfjs>，以获取这方面的最新情况。

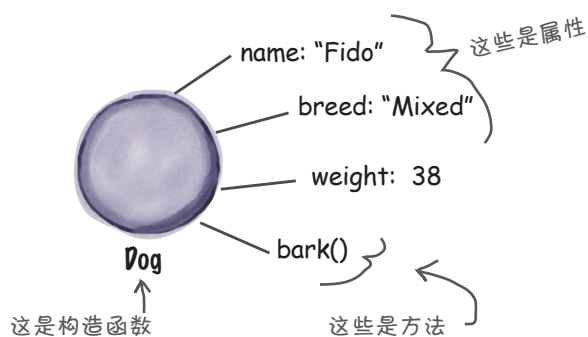
我们将此作为练习留给读者去完成。

先来介绍一种更好的对象图

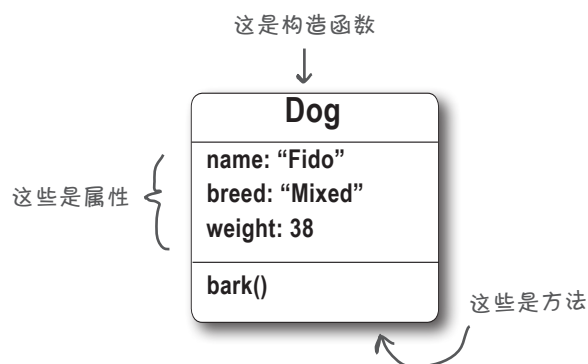
本书前面使用的对象图很精巧，但本章要严肃地讨论对象，因此必须更严肃地对待对象图。实际上，我们非常喜欢原来的对象图，而本章使用的对象图非常复杂，无法让其包罗万象。

闲话少说，上新的对象图。

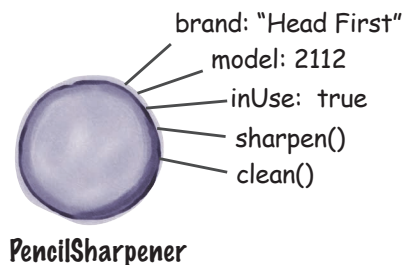
旧对象图



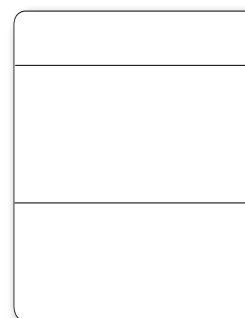
改进后的新对象图



来做个小练习，确保你完全明白新的对象图。请将下面的旧对象图改成新对象图。



补全这个对象图。



再谈构造函数：它能让我们重用代码，但效率如何呢

还记得前一章创建的构造函数Dog吗？咱们再来看一眼，看看使用构造函数有何好处：

```
function Dog(name, breed, weight) {
  this.name = name;
  this.breed = breed;
  this.weight = weight;
  this.bark = function() {
    if (this.weight > 25) {
      alert(this.name + " says Woof!");
    } else {
      alert(this.name + " says Yip!");
    }
  };
}
```

所有小狗对象都包含相同的属性，但这些属性的值各不相同。

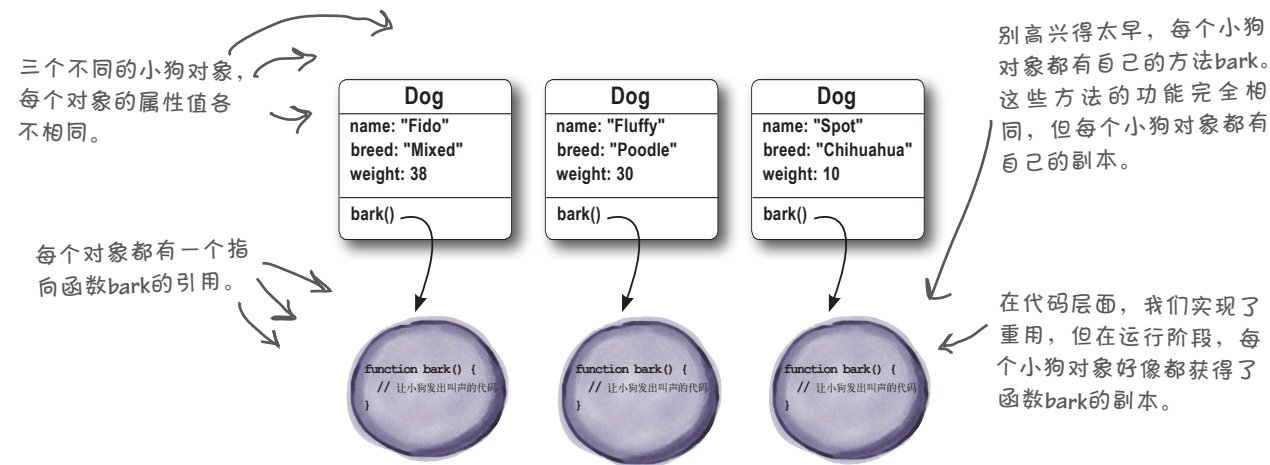
每个小狗对象都有方法bark。

更重要的是，在所有小狗对象之间重用了这些代码。

通过使用这个构造函数，可创建一致的小狗对象，并根据喜好进行定制；还可利用这个构造函数中定义的方法（这里只有一个bark）。另外，每个小狗对象都从构造函数那里获得了相同的代码，未来需要修改代码时，这可避免很多麻烦。这很好，但在运行阶段执行下面的代码时，情况将如何呢？

```
var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
```

这些代码创建三个小狗对象。使用新对象图表示时，这些对象类似于下面这样。





在我看来，每个小狗对象都应该有自己的bark方法。我只是说说而已。



重复的方法真是个问题吗

确实是个问题。一般而言，我们不希望每次使用构造函数实例化一个对象时，都创建一组新的方法。这样会影响应用程序的性能，占用计算机资源。这可能是个大问题，在移动设备上尤其如此。你将看到，还有更灵活、更强大的JavaScript对象创建方式。

我们回过头去想使用构造函数的主要目的：试图重用行为。例如，创建大量的小狗对象时，我们希望这些对象都使用相同的bark方法。通过使用构造函数，我们只需将bark方法放在构造函数Dog中，这样每次实例化对象时，都将重用方法bark的代码，从而在代码层面实现重用行为的目的。但在运行阶段，这种解决方案的效果并不好，因为每个小狗对象都将获得自己的bark方法副本。

为何会出现这种问题呢？这是因为我们没有充分利用JavaScript的对象模型。JavaScript对象模型基于原型的概念，在这种模型中，可通过扩展其他对象（即原型对象）来创建对象。

为演示原型式继承，下面首先来创建小狗原型。

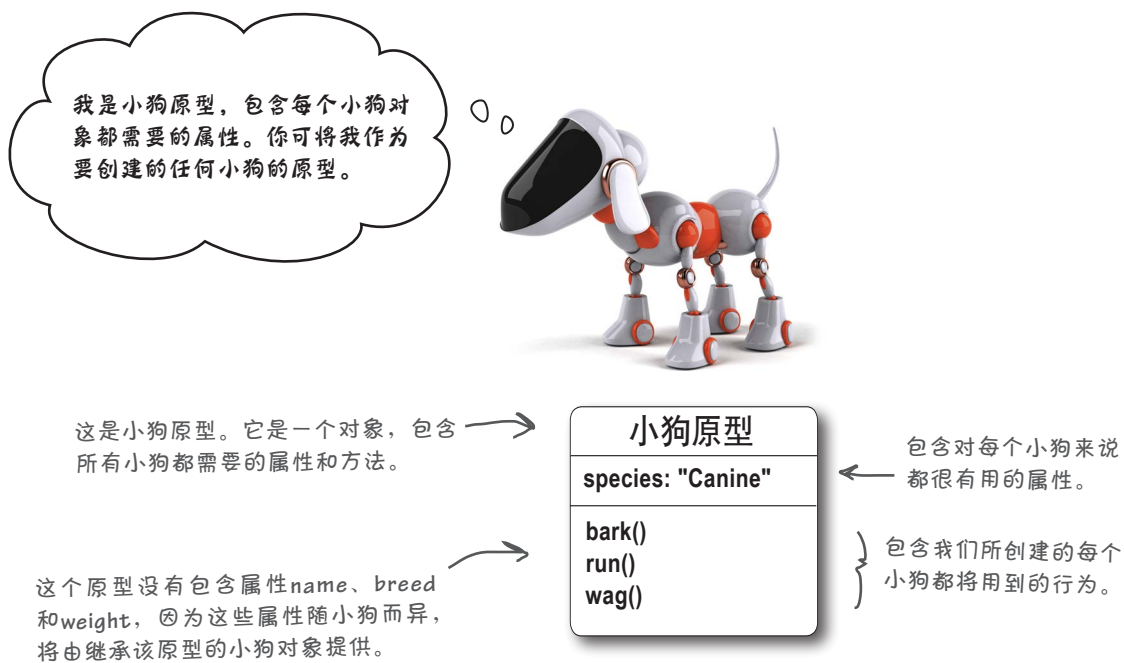
说到对象的行为时，我们通常指的是它支持的一系列方法。

原型是什么

JavaScript对象可从其他对象那里继承属性和行为。更具体地说，JavaScript使用原型式继承，其中其行为被继承的对象称为原型。这旨在继承既有属性（包括方法），同时在新对象中添加属性。这说得过于抽象，我们来看一个示例。

对象继承另一个对象后，便可访问其所有方法和属性。

我们从用于创建小狗对象的原型开始，它可能类似于下面这样。

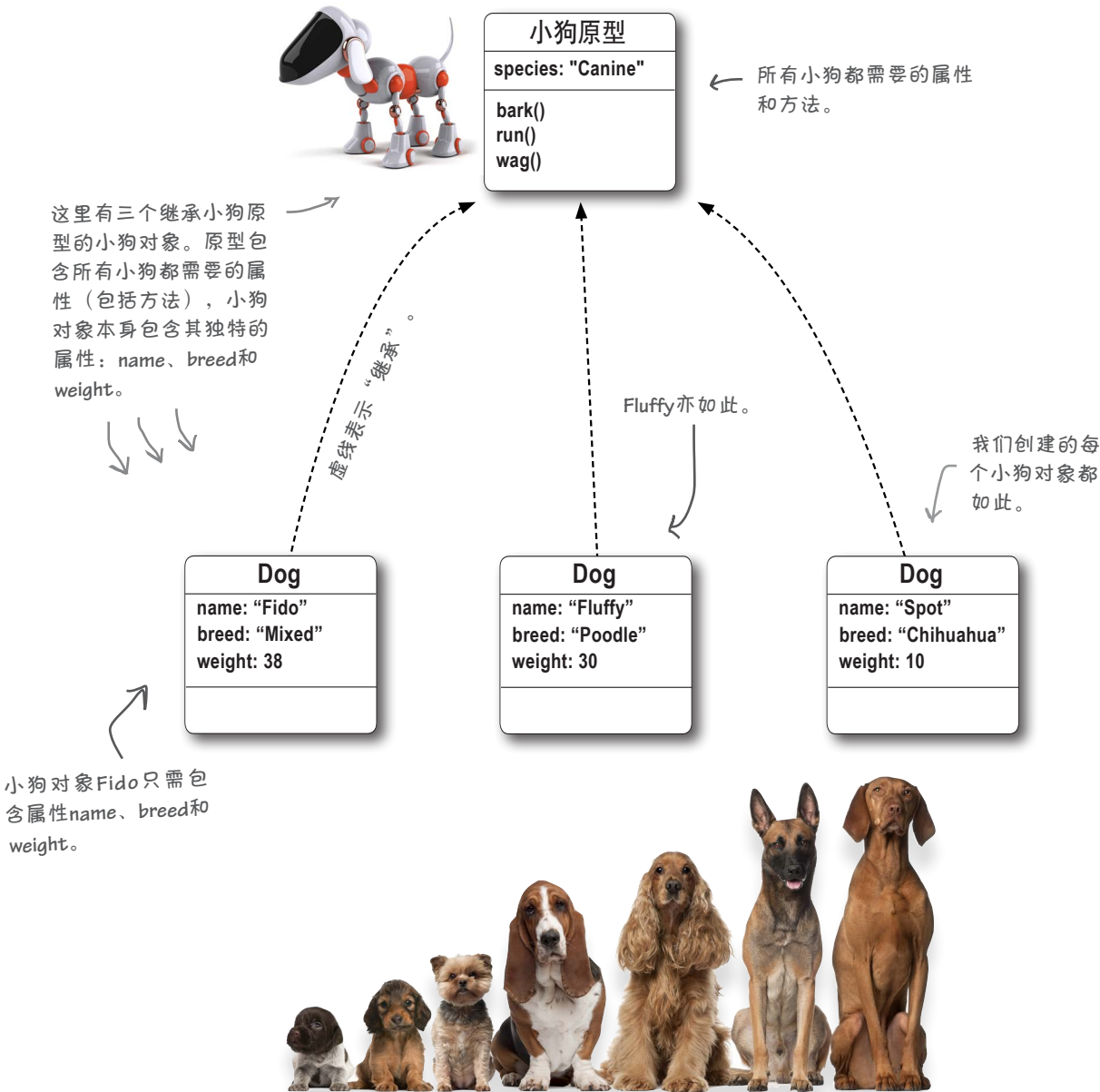


有了不错的小狗原型后，便可创建从该原型继承属性的小狗对象了。对于这些小狗对象，还可根据其具体需求添加属性和行为。例如，对于每个小狗对象，我们都将添加属性name、breed和weight。

这些小狗对象需要发出叫声、奔跑或摇尾巴时，都可使用原型提供的这些行为，因为它们从原型那里继承了这些行为。为了让你明白其中的工作原理，下面来创建几个小狗对象。

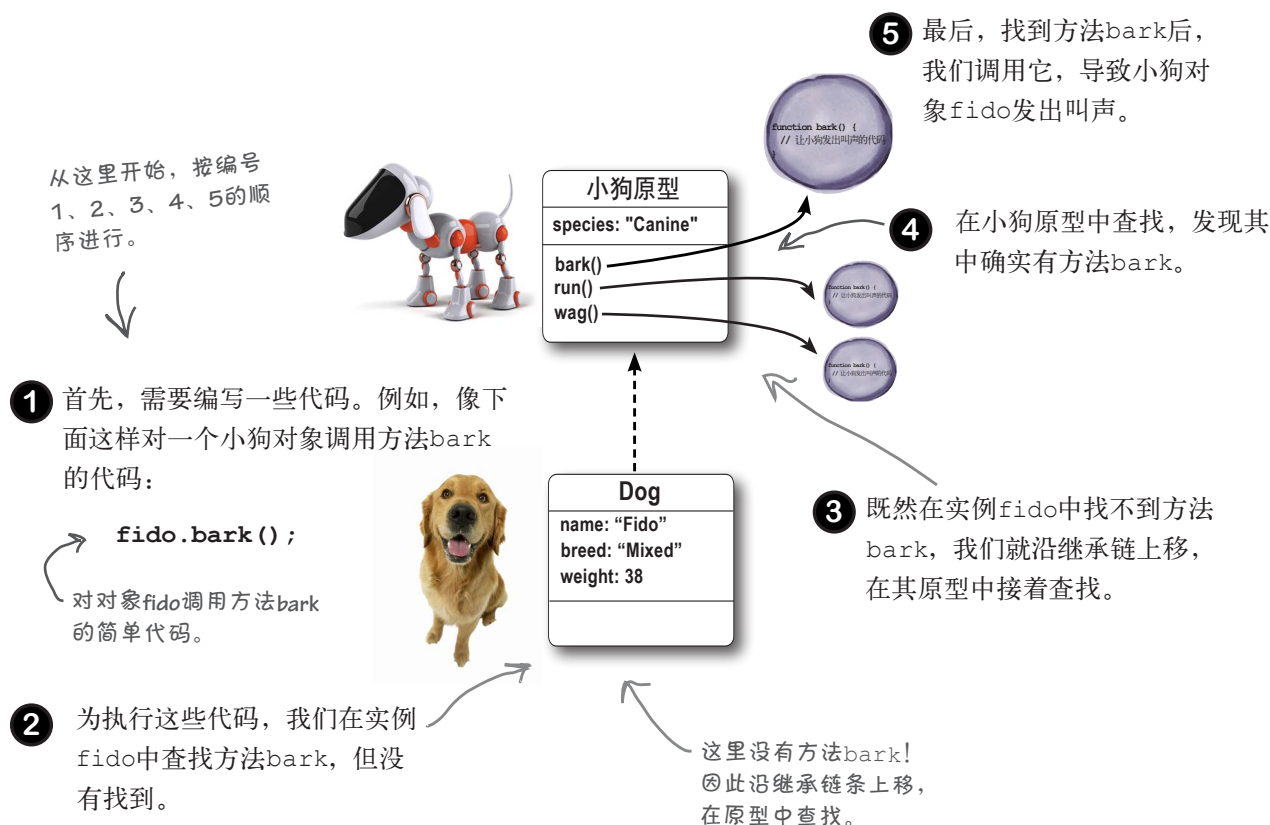
继承原型

首先，需要创建小狗对象Fido、Fluffy和Spot的对象图，让它们继承新创建的小狗原型。为表示继承关系，我们将绘制从小狗实例到原型的虚线。别忘了，我们只将**所有**小狗都需要的方法和属性放在小狗原型中，因为**所有**小狗都将继承它们。对于所有随小狗对象而异的属性，如name，我们都将其都放在小狗实例中，因为每条小狗的这些属性都各不相同。

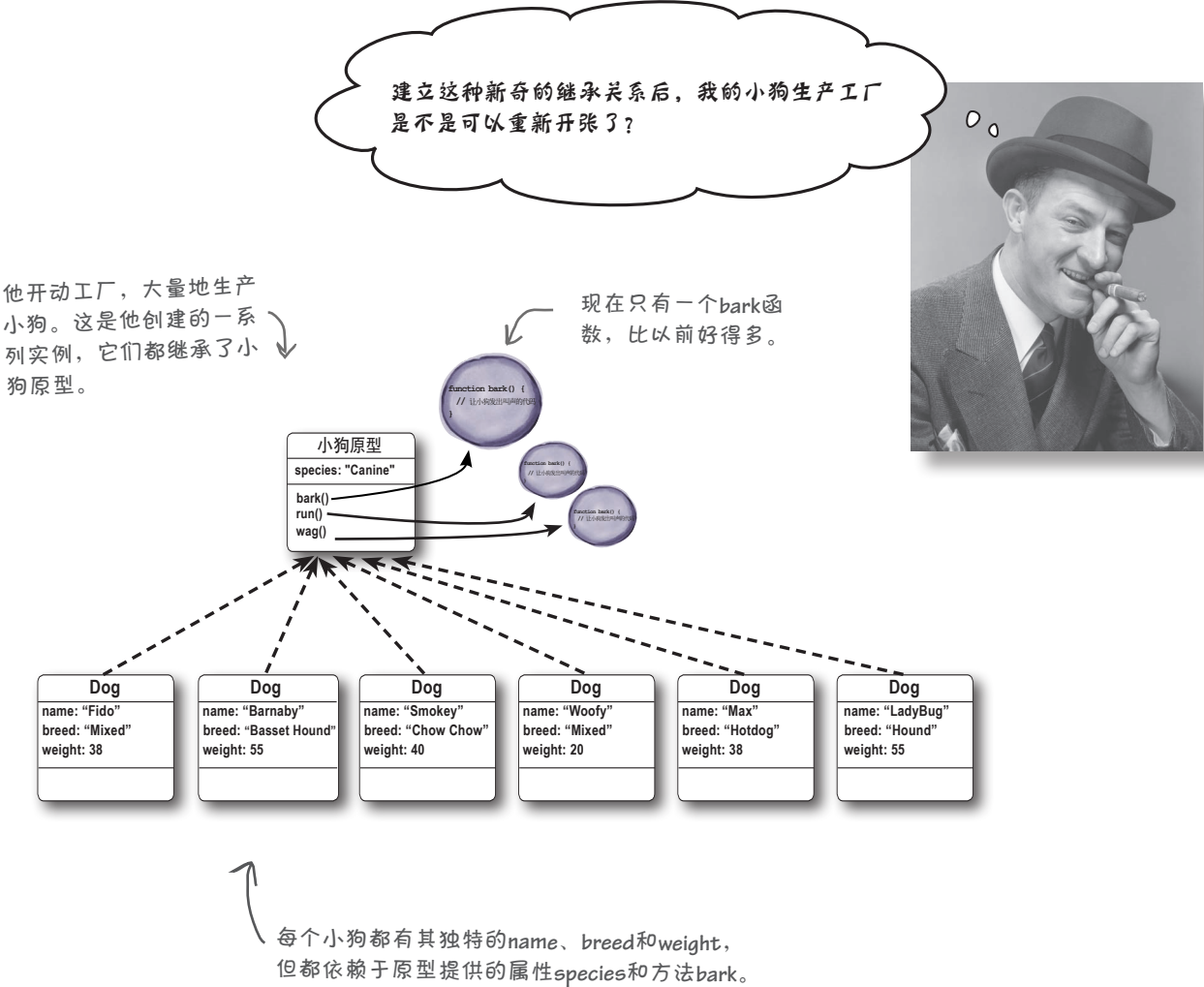


继承的工作原理

既然方法bark并不包含在各个小狗对象中，而是包含在原型中，如何让小狗发出叫声呢？这正是继承的用武之地。对对象调用方法时，如果在对象中找不到，将在原型中查找它，如下所示。

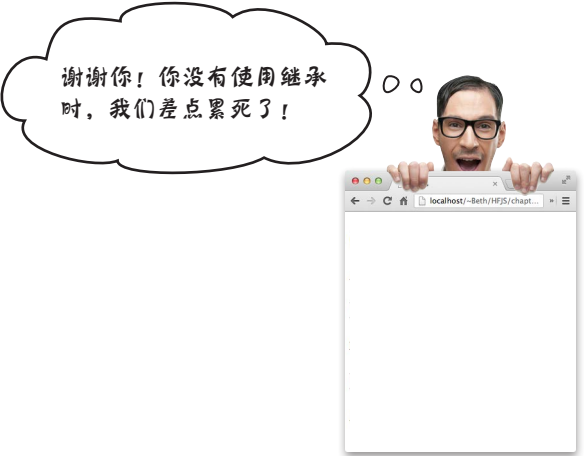


属性的情况也一样。如果我们编写了需要获取fido.name的代码，将从fido对象中获取这个值。如果要获取fido.species的值，将首先在对象fido中查找；在这里找不到后，将接着在小狗原型中查找（结果是找到了）。



明白如何使用继承后，便可以创建大量的小狗了。这些小狗都能发出叫声，但依赖于小狗原型提供的方法bark。我们实现了代码重用：不仅只需在一个地方编写代码，而且让所有小狗实例都在运行阶段使用同一个bark方法，从而避免了庞大的运行阶段开销。

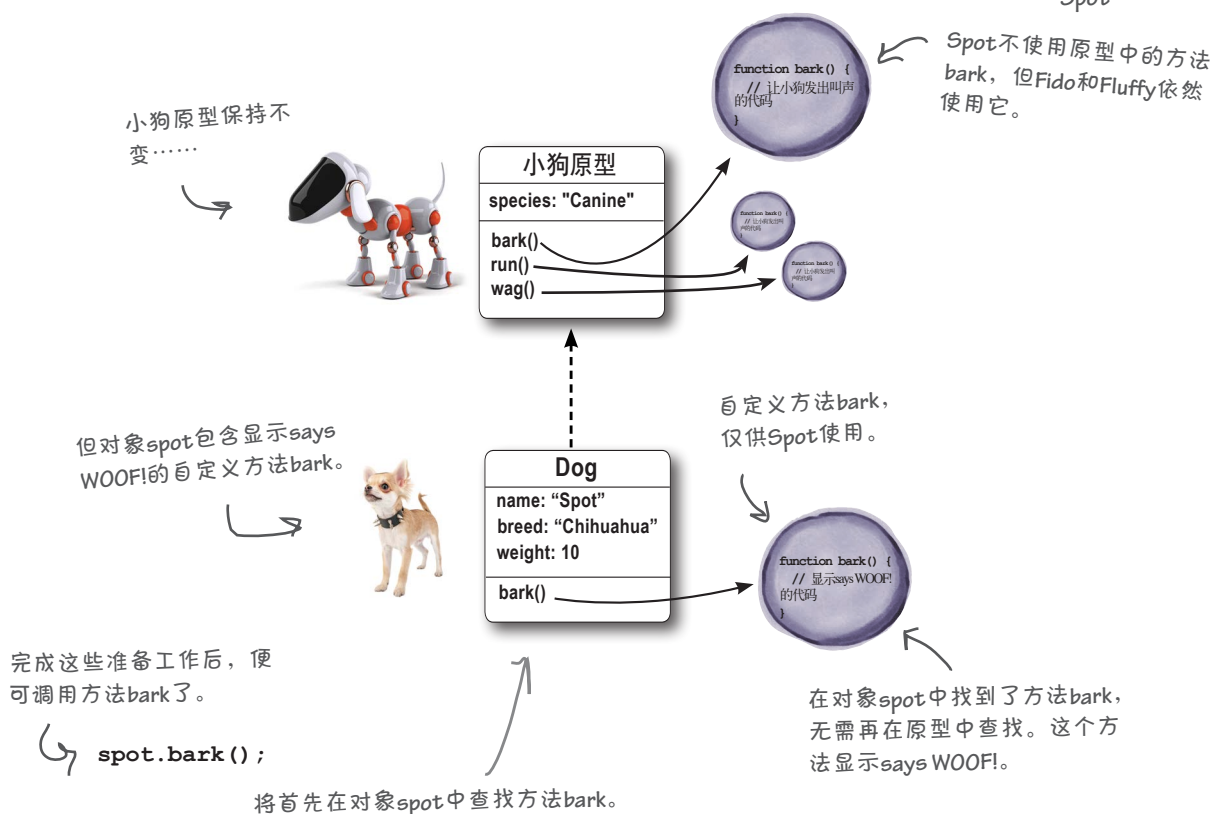
你将看到，通过使用原型，可快速地创建对象，这些对象不仅能够重用代码，还能新增行为和属性。

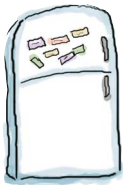


重写原型

继承原型并不意味着必须与它完全相同。在任何情况下，都可重写原型的属性和方法，为此只需在对象实例中提供它们即可。这之所以可行，是因为JavaScript总是先在对象实例（即具体的小狗对象）中查找属性；如果找不到，再在原型中查找。因此，要为对象spot定制方法bark，只需在其中包含自定义的方法bark。这样，JavaScript查找方法bark以便调用它时，将在对象spot中找到它，而不用劳神去原型中查找。

下面来看看如何在对象spot中重写方法bark，让它发出叫声时显示says WOOF!。





代码冰箱贴

冰箱上有一个对象图，但有人将它打乱了，你能帮忙将它组装起来吗？在这个对象图中，有两个机器人原型的实例：其中一个1956年生产的Robby，它归Dr. Morbius所有，拥有一个开关，喝咖啡时去星巴克；另一个是1962年生产的Rosie，它归George Jetson所有，能够打扫房间。需要指出的是，有些冰箱贴可能是多余的，祝你好运！

可供机器人继承的原型。

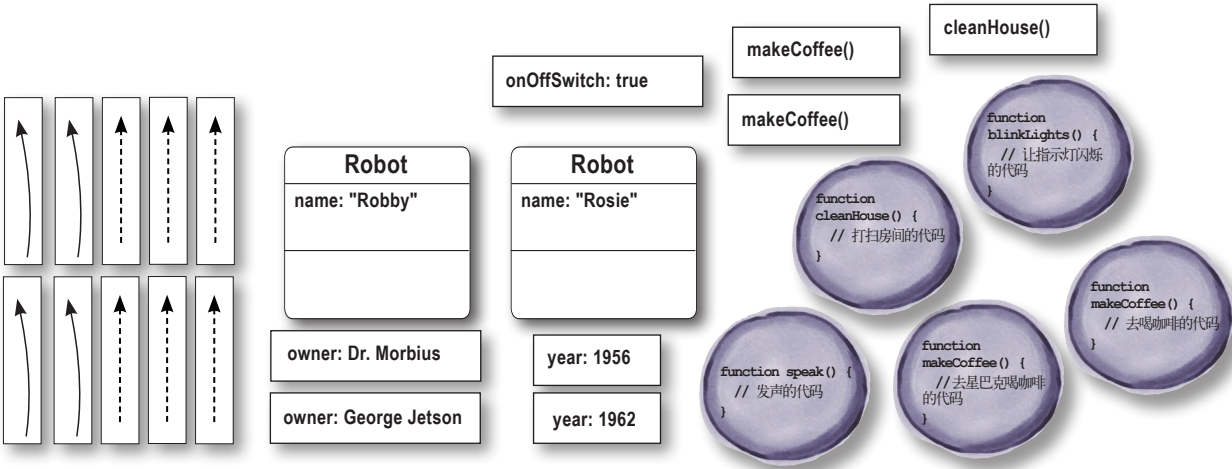


机器人原型

maker: "ObjectsRUs"

speake()
makeCoffee()
blinkLights()

请在这里组装对象图。



原型从哪里来

前面花了很多篇幅讨论小狗原型，你现在可能想看的是代码示例，而不是对象图示例。那么，如何创建或获取小狗原型呢？实际上，你已经有了一个这样的原型，只是你没有意识到而已。

下面演示了如何在代码中访问这个原型：

`Dog.prototype`



如果你查看构造函数`Dog`，将发现它有一个`prototype`属性。这是一个指向原型的引用。

属性`prototype`？

等等，`Dog`是个构造函数，即函数。你是说它有属性吗？

〇〇

幕后的东西还是不要知道的好！

跟你开玩笑呢，你说得没错，但我们一直故意隐瞒了这一点。简单地说，在JavaScript中，函数也是对象。实际上，在JavaScript中，几乎所有的东西都是对象，数组也是——你可能还没有意识到这一点。

但就目前而言，我们不想节外生枝。你只需知道，除了具备你知道的各种功能外，函数还可以有属性，而构造函数都包含属性`prototype`。这里向你保证，本书后面将更深入地讨论函数以及其他对象。



如何设置原型

前面说过，可通过构造函数Dog的属性prototype来访问原型对象，但这个原型对象包含哪些属性和方法呢？默认包含的不多。换句话说，你需要给原型添加属性和方法，这通常是在使用构造函数前进行的。

下面来设置小狗原型。为此，得有一个可供使用的构造函数。下面来看看如何根据对象图创建这样的构造函数：

```
function Dog(name, breed, weight) {  
  this.name = name;  
  this.breed = breed;  
  this.weight = weight;  
}
```

这是用于创建小狗实例的构造函数。每个实例都有自己的属性name、breed和weight，因此需要在构造函数中添加这些属性。

Dog
name: "Spot"
breed: "Chihuahua"
weight: 10

方法来自原型，因此不需要在构造函数中定义它们。

创建构造函数后，便可以设置小狗原型了。我们希望它包含属性species以及方法bark、run和wag，如下所示：

```
Dog.prototype.species = "Canine";  
  
Dog.prototype.bark = function() {  
  if (this.weight > 25) {  
    console.log(this.name + " says Woof!");  
  } else {  
    console.log(this.name + " says Yip!");  
  }  
};  
  
Dog.prototype.run = function() {  
  console.log("Run!");  
};  
  
Dog.prototype.wag = function() {  
  console.log("Wag!");  
};
```

将字符串“Canine”赋给原型的属性species。
为定义各个方法，我们分别将合适的函数赋给原型的属性bark、run和wag。



编码技巧

别忘了串接表示法：

```
Dog.prototype.species
```

通过构造函数Dog获取其属性prototype，后者是一个引用，指向的对象包含属性species。

创建几个小狗对象并对原型进行测试

为测试这个原型，请在文件（dog.html）中输入下面的代码，再在浏览器中加载它。这里再次列出了前一页的代码，并添加了一些测试代码。请确保所有的小狗对象都像预期的那样发出叫声、奔跑和摇尾。

```
function Dog(name, breed, weight) {
  this.name = name;
  this.breed = breed;
  this.weight = weight;
}
```

← 这是构造函数Dog。

```
Dog.prototype.species = "Canine";
```

↪ 这里给小狗原型添加了属性和方法。

```
Dog.prototype.bark = function() {
  if (this.weight > 25) {
    console.log(this.name + " says Woof!");
  } else {
    console.log(this.name + " says Yip!");
  }
};
```

↪ 我们给小狗原型添加了一个属性和三个方法。

```
Dog.prototype.run = function() {
  console.log("Run!");
};
```

```
Dog.prototype.wag = function() {
  console.log("Wag!");
};
```

像通常那样创建小狗对象。

```
var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
```

```
fido.bark();
fido.run();
fido.wag();
```

↪ 然后，像通常那样对每个小狗对象调用方法。每个小狗对象都从原型那里继承了这些方法。

```
fluffy.bark();
fluffy.run();
fluffy.wag();
```

```
spot.bark();
spot.run();
spot.wag();
```

每个小狗对象都能发出叫声、奔跑和摇尾。很好！

等等，不是说Spot发出叫声时应显示says WOOF!吗？

JavaScript控制台

```
Fido says Woof!
Run!
Wag!
Fluffy says Woof!
Run!
Wag!
Spot says Yip!
Run!
Wag!
```



别把我给忘了，我要求在发出叫声时显示says WOOF!呀！

编写让Spot发出叫声时显示says WOOF!的代码

别担心，我们可没忘记Spot。Spot要求在发出叫声时显示says WOOF!，因此我们需要重写原型，给Spot提供自定义方法bark。下面来修改代码：

```
... }  
  
var spot = new Dog("Spot", "Chihuahua", 10);  
  
spot.bark = function() {  
    console.log(this.name + " says WOOF!");  
};  
  
// 对fido和fluffy调用方法的代码  
  
spot.bark();  
spot.run();  
spot.wag();
```

这里省略了其他的代码，这旨在节省纸张或者说减少碳足迹……

我们所作的唯一修改是，给Spot提供自定义方法bark。

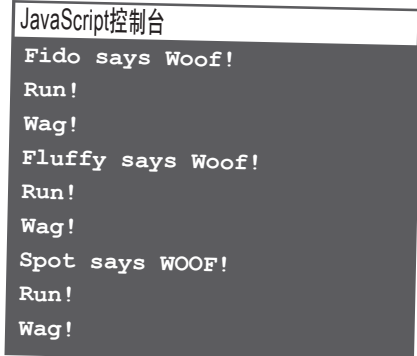
对Spot调用方法bark的代码根本不需要修改。

测试自定义方法bark



添加上述代码后，进行简单的测试……

Spot得偿所愿，在发出叫声时显示了says WOOF!





还记得前面包含机器人Robby和Rosie的对象图吗？现在就来实现它。我们编写了构造函数Robot和一些测试代码，而你需要做的是设置机器人原型并创建这两个机器人对象，再通过运行对这些代码进行测试。

```
function Robot(name, year, owner) {
  this.name = name;
  this.year = year;
  this.owner = owner;
}
```

这是构造函数Robot，你还需设置其原型。

```
Robot.prototype.maker =
```

请在这里设置机器人原型。

```
Robot.prototype.speak =
```

```
Robot.prototype.makeCoffee =
```

```
Robot.prototype.blinkLights =
```

```
var robby =
```

```
var rosie =
```

```
robby.onOffSwitch =
```

```
robby.makeCoffee =
```

请在这里编写创建机器人对象Robby和Rosie的代码。请务必给这些实例添加所有的自定义属性。

```
rosie.cleanHouse =
```

```
console.log(robby.name + " was made by " + robby.maker +
  " in " + robby.year + " and is owned by " + robby.owner);
```

```
robby.makeCoffee();
```

```
robby.blinkLights();
```

```
console.log(rosie.name + " was made by " + rosie.maker +
```

```
  " in " + rosie.year + " and is owned by " + rosie.owner);
```

```
rosie.cleanHouse();
```

使用这段代码测试这些实例，确保它们继承了原型并能正确地工作。

我有点疑惑，鉴于方法**bark**位于原型而不是对象中，其中的**this.name**怎么不会导致问题呢？

问得好。在没有使用原型的情况下，这很容易解释，因为**this**指的是方法被调用的对象。调用原型中的方法**bark**时，你可能认为**this**指的是原型对象，但情况并非如此。

调用对象的方法时，**this**被设置为方法被调用的对象。即便在该对象中没有找到调用的方法，而是在原型中找到了它，也不会修改**this**的值。在任何情况下，**this**都指向原始对象，即方法被调用的对象，即便该方法位于原型中亦如此。因此，即便方法**bark**位于原型中，调用这个方法时，**this**也将被设置为原始小狗对象，得到的结果也是我们期望的，如显示**Fluffy says Woof!**。



所有的 让 ~~一只~~ 小狗都学会新技能

该让所有小狗都学会新技能了。没错，就是所有的小狗。使用原型后，如果给原型添加一个方法，所有的小狗对象都将立即从原型那里继承这个方法并自动获得这种新行为，包括添加方法前已创建的小狗对象。

假设我们要让所有小狗都会坐下，只需在原型中添加一个坐下的方法即可。

```
var barnaby = new Dog("Barnaby", "Basset Hound", 55);
```

创建另一个小狗对象，以测试这一点。

```
Dog.prototype.sit = function() {  
    console.log(this.name + " is now sitting");  
}
```

再添加方法sit。

下面来让小狗Barnaby坐下：

```
barnaby.sit();
```

首先检查对象barnaby是否有方法sit，结果发现它没有。接着在原型中查找，找到方法sit后调用它。

JavaScript控制台

Barnaby is now sitting



深入研究

下面来更深入地研究其中的工作原理。请务必按步骤1、2、3、4的顺序进行。

- 4 在原型中找到了方法sit，并且调用它。



小狗原型	
species:	"Canine"
bark()	
run()	
wag()	
sit()	



- 2 接下来，在原型中添加新方法sit。

```
function sit() {  
    // 让小狗坐下的代码  
}
```

- 3 调用方法barnaby.sit，但在对象barnaby中没有找到方法sit。



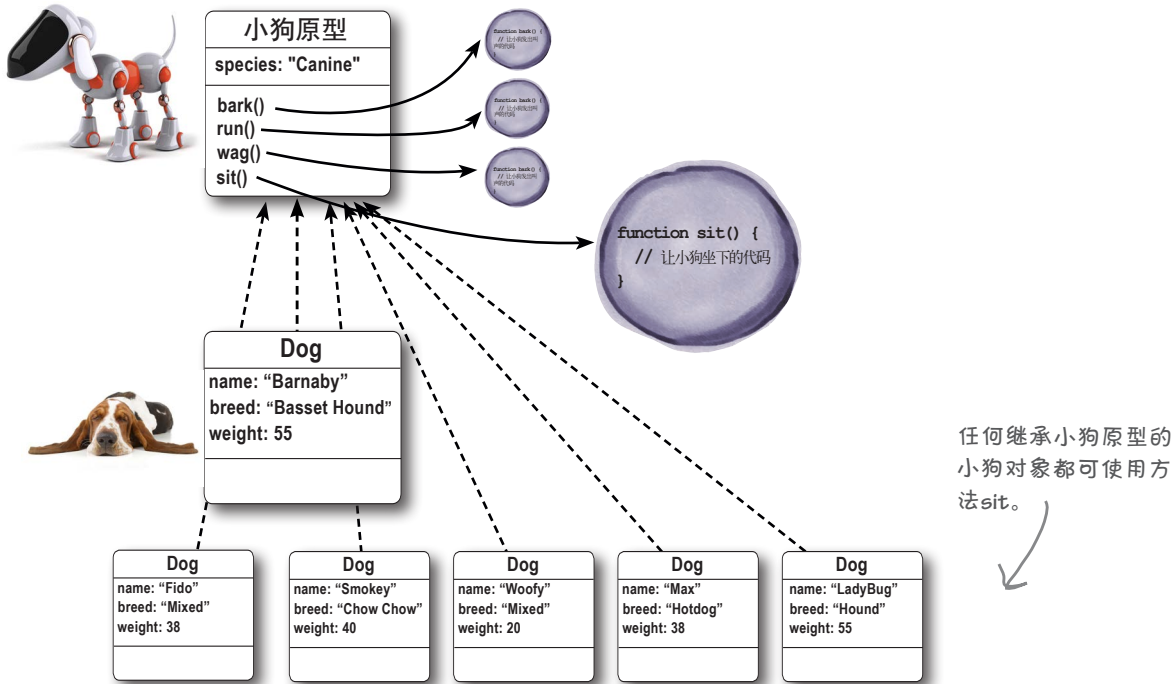
Dog	
name:	"Barnaby"
breed:	"Basset Hound"
weight:	55

- 1 创建一个新的小狗对象barnaby。

原型是动态的

Barnaby能够坐下了，看到这一点我们很高兴。实际上，现在所有的小狗都能够坐下，因为在原型中添加方法后，继承该原型的任何对象都能使用这个方法。

当然，对属性来说，情况亦如此。



世上没有愚蠢的问题

问：也就是说，给原型添加新的方法或属性后，继承该原型的所有对象实例都将立即看到它？

答：如果你说的“看到”是继承的意思，那你说的完全正确。请注意，这提供了一个途径，让你只需在运行阶段修改原型，就可扩展或修改其所有实例的行为。

问：我知道，给原型添加新属性后，继承该原型的所有对象都将包含这个属性，但修改原型的既有属性呢？这是否也会影响继承原型的所有对象？比方说，如果我将属性species的值从Canine改为Feline，会不会导致所有既有小狗对象的属性species都变成Feline？

答：是的。修改原型的任何属性时，都将影响继承该原型的所有对象——只要它们没有重写这个属性。



有一个机器人游戏使用了机器人Robby和Rosie。这个机器人游戏的代码如下。在这个游戏中，玩家的等级达到42后，机器人将具备一项新功能：发射激光束。请补全下面的代码，使得在玩家的等级达到42后，机器人Robby和Rosie都能发射激光束。继续往下阅读前，请查看本章末尾的答案。

```
function Game() {
    this.level = 0;
}

Game.prototype.play = function() {
    // 让玩家玩游戏的代码
    this.level++;
    console.log("Welcome to level " + this.level);
    this.unlock();
}

Game.prototype.unlock = function() {

}

function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}

var game = new Game();
var robby = new Robot("Robby", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");

while (game.level < 42) {
    game.play();
}

robby.deployLaser();
rosie.deployLaser();
```

JavaScript控制台

```
Welcome to level 1
Welcome to level 2
Welcome to level 3
...
Welcome to level 41
Welcome to level 42
Rosie is blasting you with
laser beams.
```

这是我们得到的输出示例。补全代码后，请尝试玩玩这个游戏，看看是哪个机器人先发射激光束。



方法sit更有趣的实现

下面来让方法sit更有趣些：小狗开始处于非坐着（即站立）状态。在方法sit中，判断小狗是否是坐着的。如果不是，就让它坐着；如果是，就告诉用户小狗已经是坐着的。为此，需要一个额外的属性sitting，用于跟踪小狗是否是坐着的。下面来编写这样的代码：

首先，在原型中添加属性sitting。

```
Dog.prototype.sitting = false;
```

通过在原型中将属性sitting
设置为false，让所有的小狗
一开始都是站着的。

```
Dog.prototype.sit = function() {  
  if (this.sitting) {  
    console.log(this.name + " is already sitting");  
  } else {  
    this.sitting = true;  
    console.log(this.name + " is now sitting");  
  }  
};
```

接下来，在方法sit中检查小狗是
否是坐着的。如果这个方法
是首次被调用，检查this.sitting时，
将在小狗原型中查找其值。

如果小狗是坐着的，就指出小狗已
经是坐着的。

但如果小狗不是坐着的，就说它现在坐下，再
将this.sitting设置为true。这将重写原型的属性
sitting，并在实例中设置这个属性的值。

请注意，实例现在有自己的sitting属性，
其值为true。

这些代码的有趣之处在于，小狗实例刚创建时，从原型那里继承了属性sitting，该属性的值默认为false；但调用方法sit后，就给小狗实例添加了属性sitting的值，导致在小狗实例中创建了属性sitting。这让我们能够给所有小狗对象指定默认值，并在需要时对各个小狗进行定制。

测试新的sit方法

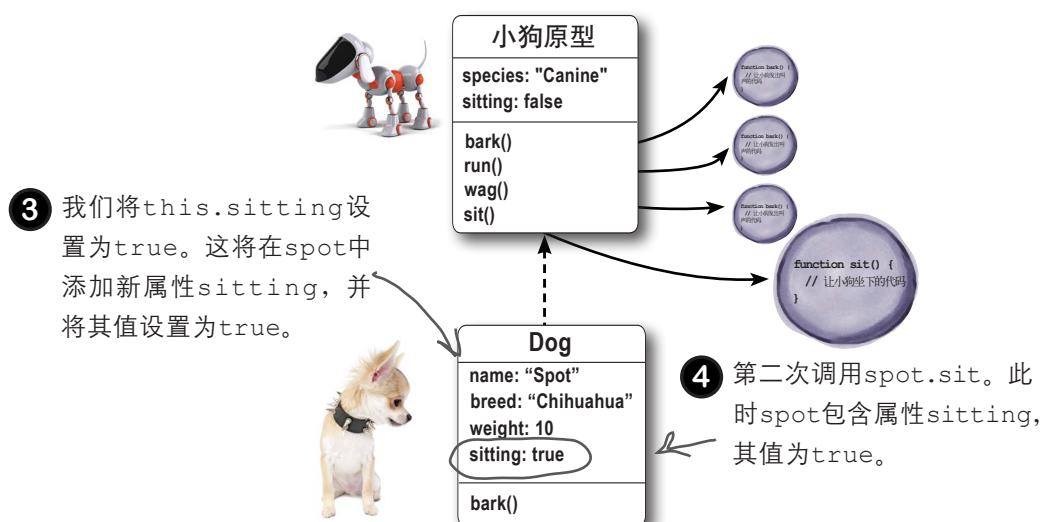
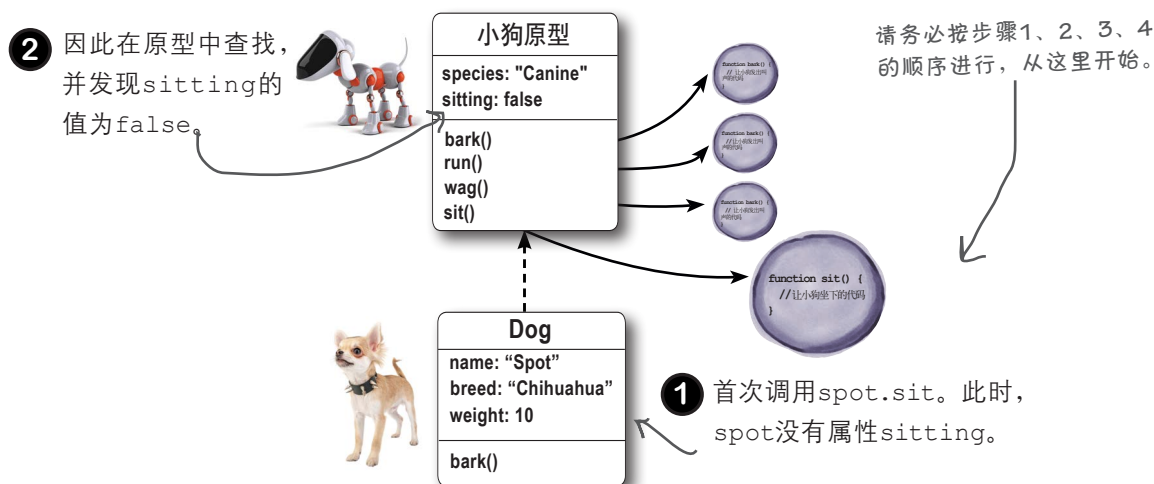
下面来尝试使用这个方法。请在你的代码中添加上述新属性sitting以及方法sit的新实现，再对代码进行测试。你将发现，现在可以让barnaby坐下，再让spot坐下，且每个小狗对象都独立地跟踪自己是否是坐着的。

```
barnaby.sit() → Barnaby is now sitting  
barnaby.sit() → Barnaby is already sitting  
spot.sit() → Spot is now sitting  
spot.sit() → Spot is already sitting
```



再谈属性sitting的工作原理

下面来确保你明白了其中的工作原理，因为如果你没有仔细分析前述实现，可能遗漏重要的细节。要点如下：首次获取sitting的值时，是从原型中获取的；但接下来将sitting设置为true时，是在对象实例而不是原型中进行的。在对象实例中添加这个属性后，接下来每次获取sitting的值时，都将从对象实例中获取，因为它重写了原型中的这个属性。下面再次详细地介绍这一点。





既然说到属性，在代码中是否有办法判断使用的属性包含在实例还是原型中呢？

有办法，可使用每个对象都有的方法hasOwnProperty。如果属性是在对象实例中定义的，这个方法将返回true。如果属性不是在对象实例中定义的，但能够访问它，就可认为它肯定是在原型中定义的。

下面来对fido和spot调用这个方法。首先，我们知道，在小狗原型中定义了属性species，而且spot和fido都没有重写这个属性。因此，如果我们对这两个对象调用方法hasOwnProperty，并以字符串的方式传入属性名"species"，结果都将为false：

```
spot.hasOwnProperty("species");
fido.hasOwnProperty("species");
```

这两条语句都返回false，因为species是在原型而不是对象实例spot和fido中定义的。

下面来尝试对属性sitting进行这种判断。我们知道，在原型中定义了属性sitting，并将其初始化为false，因此将spot.sitting设置为true时，将重写原型中的属性sitting，并在实例spot中定义属性sitting。下面来询问spot和fido自己是否定义了属性sitting：

首次检查spot是否有自己的sitting属性时，结果为false。

```
spot.hasOwnProperty("sitting");
```

```
spot.sitting = true;
```

```
spot.hasOwnProperty("sitting");
```

```
fido.hasOwnProperty("sitting");
```

接下来，我们将spot.sitting设置为true，这将在实例spot中添加属性sitting。

这次调用hasOwnProperty时，结果为true，因为spot现在有自己的sitting属性。

但对fido调用hasOwnProperty时，结果为false，因为实例fido没有sitting属性。这意味着fido使用的sitting属性是在原型中定义的，而fido从原型那里继承了这个属性。



我们给机器人Robby和Rosie添加了一项新功能，现在它们能够在发生错误时通过方法reportError进行报告。请研究下面的代码，特别要注意这个方法是从什么地方获取错误信息的，以及这个方法是否是在机器人原型中定义的。

请在下面写出这些代码的输出：

```
function Robot(name, year, owner) {
  this.name = name;
  this.year = year;
  this.owner = owner;
}

Robot.prototype.maker = "ObjectsRUs";
Robot.prototype.errorMessage = "All systems go.";
Robot.prototype.reportError = function() {
  console.log(this.name + " says " + this.errorMessage);
};
Robot.prototype.spillWater = function() {
  this.errorMessage = "I appear to have a short circuit!";
};

var robby = new Robot("Robby", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");

rosie.reportError();
robby.reportError();
robby.spillWater();
rosie.reportError();
robby.reportError();
```

```
console.log(robby.hasOwnProperty("errorMessage")); _____
console.log(rosie.hasOwnProperty("errorMessage")); _____
```

Robby有自己的errorMessage
属性吗？



Rosie呢？



最优秀的表演犬



在本章中，你所做的艰苦工作带来了回报。Web镇小狗俱乐部见到你有关小狗对象的工作后，立即意识到你就是他们要找的人，决定邀请你来帮助他们实现小狗表演模拟器。他们要求你做的唯一工作是更新构造函数Dog，以便能够创建表演犬。归根结底，表演犬可不是普通的狗，它们不仅会奔跑，还会各种步法；它们不去翻垃圾箱，而是喜欢寻找有香味的东西；它们不摇尾乞食，而是喜欢追着诱饵不放。

更具体地说，Web镇小狗俱乐部需要的是下面这样的表演犬。



你的构造函数Dog编写得很好！我们竭诚邀请你来开发小狗表演模拟器。表演犬有点独特，需要有额外的方法（参见下面的清单）。

谢谢！——Web镇小狗俱乐部

stack(): 立正。

gait(): 类似于奔跑。这个方法接受一个字符串参数，如walk、trot、pace或gallop。

bait(): 让小狗饱餐一顿。

groom(): 让小狗洗个澡。

如何设计表演犬

我们将如何设计表演犬呢？显然，我们希望利用既有的小狗代码，毕竟这正是Web镇小狗俱乐部最初求助于我们的原因所在。但如何利用呢？下面就来研究研究。

如果将这些新方法添加到既有的构造函数Dog中，所有的小狗对象都将能够做这些事情。这有悖于我们的初衷。

我们可以只在表演犬实例中添加这些方法，但这就回到了老路上，本章开头揭示的所有问题都会出现。

如果我们从头开始创建构造函数ShowDog，就得重新实现所有的基本方法：bark、run、sit等。





大家别着急上火。使用
JavaScript时，可以有多个
原型。

Joe: 多个原型? 什么意思?

Judy: 就像你得到的遗产。

Joe: 什么遗产? 我要是有遗产的话, 就不会在这里打工了。开什么玩笑?

Judy: 你并非只继承了父母的特质, 不是吗? 你还继承了祖父母、外祖父母、曾祖父母、曾外祖父母等的一些特质。

Joe: 没错, 明白你的意思。

Judy: 在JavaScript中, 可建立供对象继承的原型链。

Frank: 举个例子吧, 这样可能会有所帮助。

Judy: 假设你有一个小鸟原型, 知道如何做大多数小鸟都会做的事情, 如飞翔。

Frank: 明白, 就像我们的小狗原型。

Judy: 现在假设你需要实现各种鸭子——绿头鸭、红头鸭等。

Frank: 别忘了还有红嘴树鸭。

Judy: 谢谢Frank提醒。

Frank: 不客气。我也是刚阅读《Head First 设计模式》, 才知道这些种类的鸭子。

Judy: 好吧, 但鸭子与一般的鸟不同。它们会游泳, 我们不想将这个放在小鸟原型中。不过, 在JavaScript中, 可创建一个继承小鸟原型的鸭子原型。

Joe: 你的意思是说, 可以创建指向鸭子原型的构造函数Duck, 而鸭子原型又指向小鸟原型?

Frank: 太复杂了, 请详细说说。

Judy: Frank, 你这样想一想。假设你创建了一个鸭子对象, 并对其调用了方法fly。如果在该对象中查找时, 没有找到这个方法, 你将如何做呢? 你接着在鸭子原型中查找, 可这里也没有方法fly。因此, 你继续在鸭子原型继承的小鸟原型中查找, 并在这里找到了方法fly。

Joe: 如果我们调用方法swim, 将首先在当前鸭子实例中查找; 但找不到。因此我们接着在鸭子原型中查找, 并找到了它。

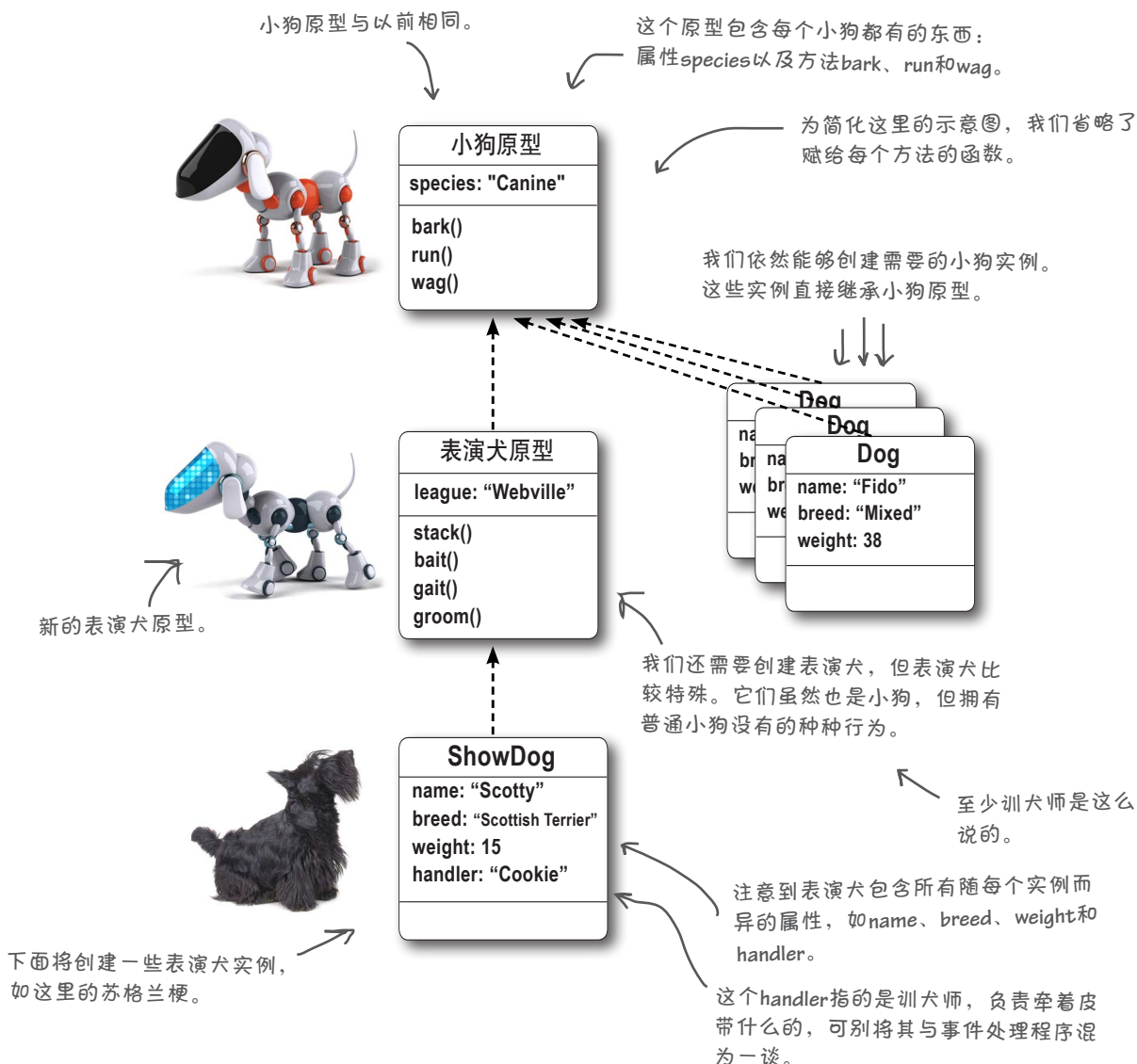
Judy: 没错。这样, 我们不仅重用了鸭子原型的行为, 必要时还可沿原型链往上走, 进而使用小鸟原型的行为。

Joe: 这听起来非常完美, 完全可以通过扩展小狗原型来创建表演犬原型。下面就来看看如何做。

建立原型链

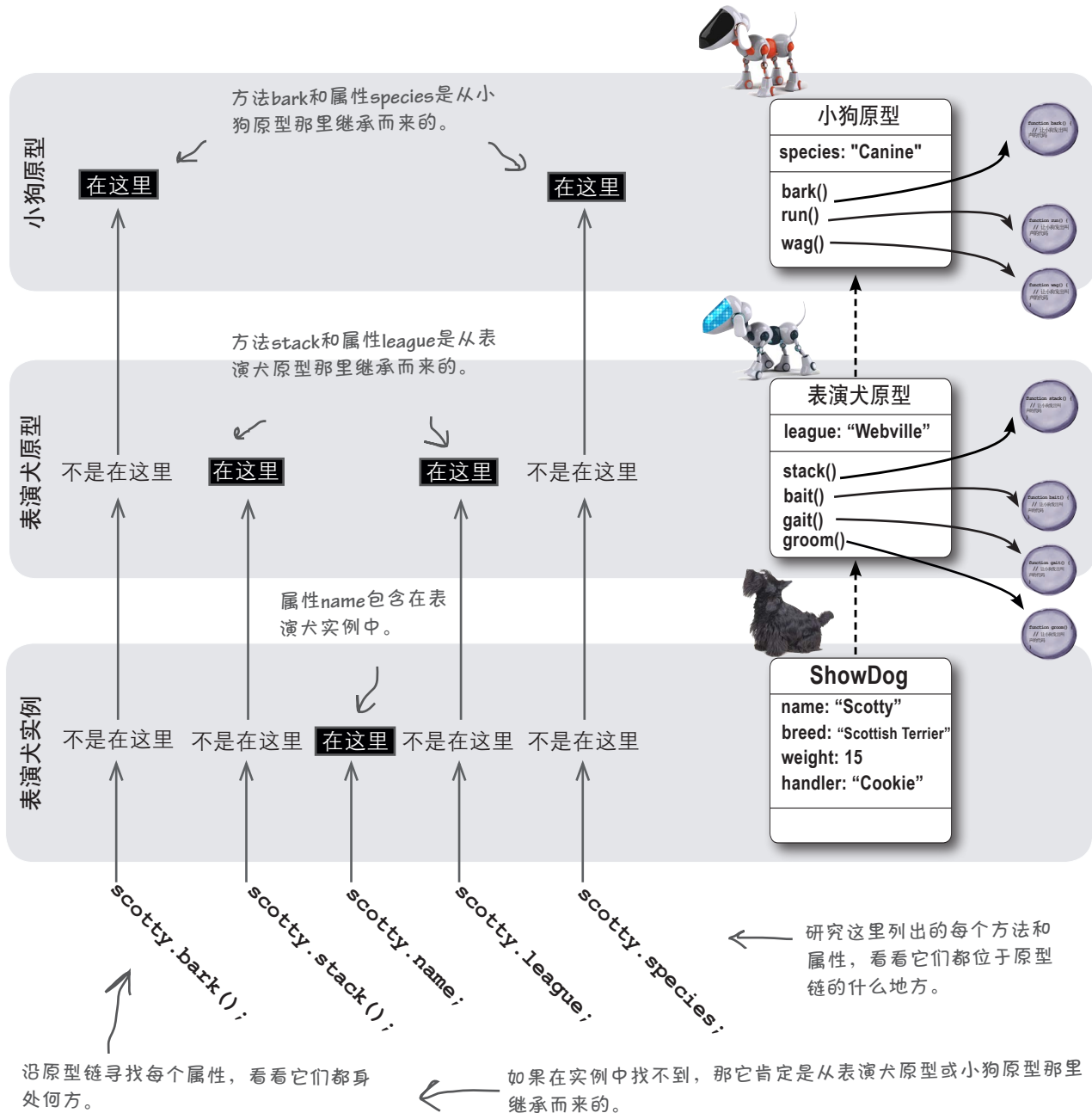
咱们来考虑如何建立原型链。对象不仅可以继承一个原型的属性，还可继承一个原型链。基于前面考虑问题的方式，这并不难理解。

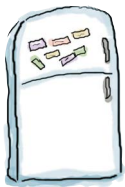
假设我们需要一个用于创建表演犬的表演犬原型，并希望这个原型依赖于小狗原型提供的方法bark、run和wag。下面就来建立这样的原型链，体会一下其中的各个部分是如何协同工作的。



原型链中的继承原理

为表演犬建立原型链后，下面来看看其中的继承原理。对于本页下方的每个属性和方法，请沿原型链向上找出它们都是在哪里定义的。





代码冰箱贴

我们在冰箱上又贴了一个对象图，但也被人弄乱了。你能帮忙将其重新组合起来吗？与前面的对象图相比，这个对象图新增了一些太空机器人，它们继承了机器人的属性，重写了机器人的方法speak，还新增了方法pilot()和属性homePlanet。请注意，有些冰箱贴可能是多余的，祝你好运！

在这里组装对象图。↓

这是机器人原型。→



机器人原型
maker: "ObjectsRUs"
speak()
makeCoffee()
blinkLights()

function speak() {
 // 发声
}
function makeCoffee() {
 // 去喝咖啡
}
function blinkLights() {
 // 让指示灯闪烁的代码
}

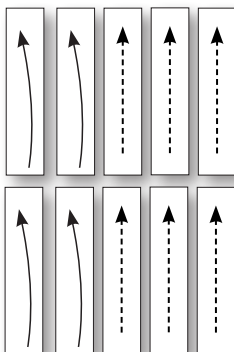
这是太空机器人原型。↙



太空机器人原型

Robot
name: "Robby"
year: 1956
owner: "Dr. Morbius"

Robot
name: "Rosie"
year: 1962
owner: "George Jetson"



speak() year: 1977
pilot() year: 2009
homePlanet: "Earth"
homePlanet: "Tatooine"

function speak() {
 // 在太空中发声的代码
}
function pilot() {
 // 驾驶太空飞船的代码
}

Space Robot
name: "C3PO"
year: 1977
owner: "L. Skywalker"

Space Robot
name: "Simon"
year: 2009
owner: "Carla Diana"

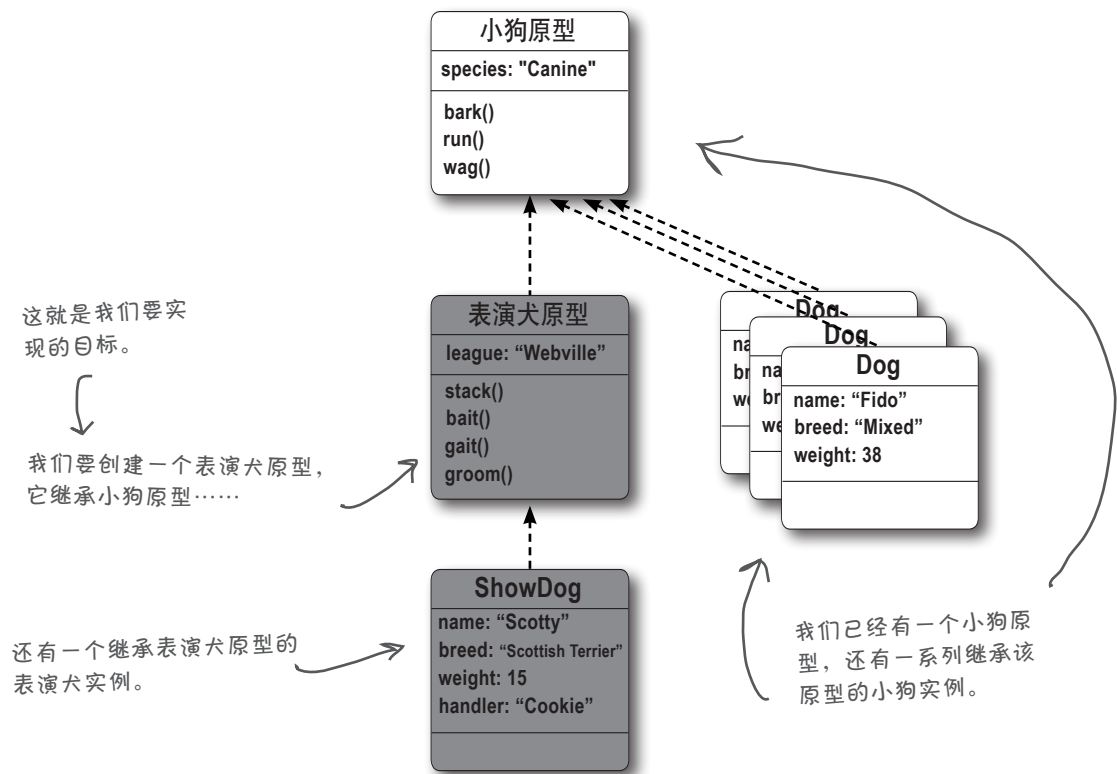
创建表演犬原型

创建小狗原型时，只需直接使用构造函数Dog的属性prototype提供的空对象，在其中添加要让每个小狗实例都继承的属性和方法即可。

但创建表演犬原型时，我们必须做更多的工作，因为我们需要的是一个继承另一个原型（小狗原型）的原型对象。为此，我们必须创建一个继承小狗原型的对象，再亲自动手建立关联。

当前，我们有一个小狗原型，还有一系列继承这个原型的小狗实例，而目标是创建一个继承小狗原型的表演犬原型以及一系列继承表演犬原型的表演犬实例。

为此，需要一步一步来完成。



首先，需要一个继承小狗原型的对象

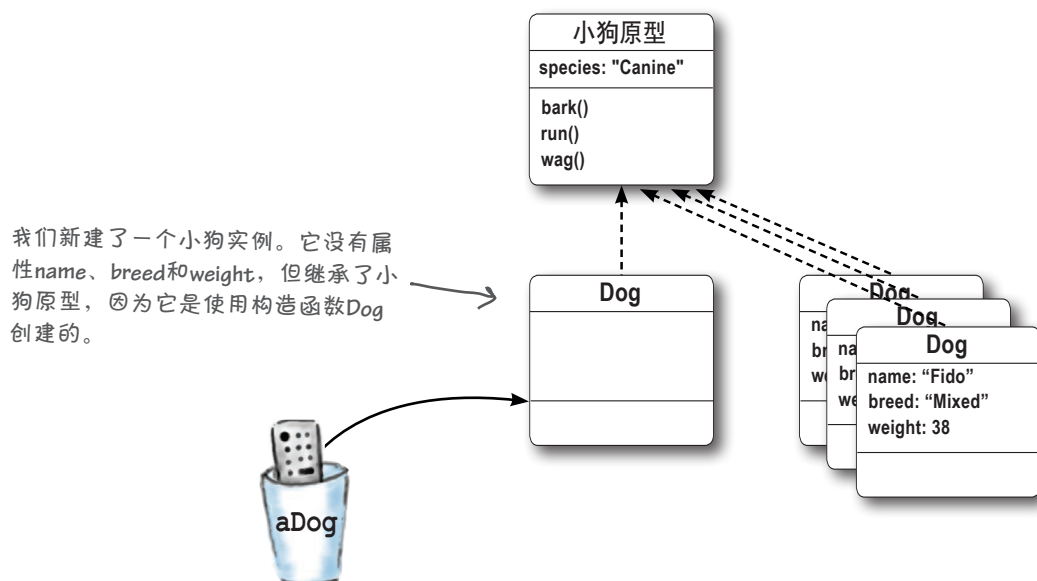
前面说过，表演犬原型是一个继承小狗原型的对象。要创建继承小狗原型的对象，最佳方式是什么呢？其实就是前面创建小狗实例时一直采用的方式。你可能还记得，这种方式类似于下面这样：

要创建继承小狗原型的对象，只需结合使用new和构造函数Dog。

```
var aDog = new Dog();
```

这里没有给构造函数提供任何实参，其中的原因将稍后讨论。

上述代码创建一个继承小狗原型的对象，因为它与以前创建小狗实例时使用的代码完全相同，只是没有向构造函数提供任何实参。为什么这样做呢？因为在这里，我们只需要一个继承小狗原型的小狗对象，而不关心其细节。



当前，我们需要的是一个表演犬原型。与其他小狗实例一样，它也是一个继承小狗原型的对象。下面来看看如何将这个空的小狗实例变成所需的表演犬原型。

接下来，将新建的小狗实例变成表演犬原型

至此，我们有了一个小狗实例，但如何使其成为表演犬原型呢？为此，只需将它赋给构造函数ShowDog的属性prototype。等等，我们还没有构造函数ShowDog呢，下面就来创建它：

```
function ShowDog(name, breed, weight, handler) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.handler = handler;
}
```

这个构造函数接受各种实参，用于设置小狗的属性（name、breed、weight）和表演犬的属性（handler）。

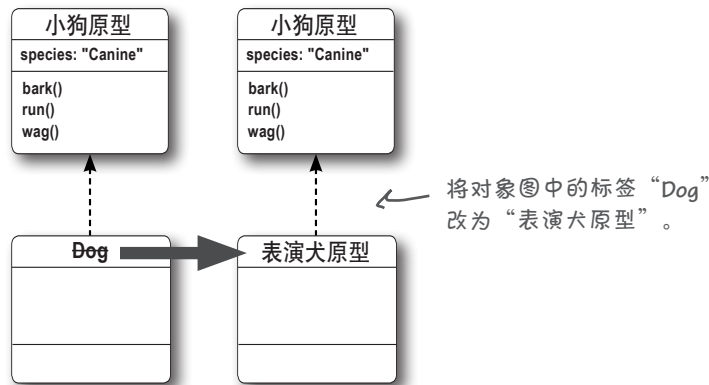
有了这样的构造函数后，便可将其属性prototype设置为一个新的小狗实例了：

我们原本可以使用前一页创建的小狗实例，但为少使用一个变量，这里没有这样做，而是直接将一个新小狗实例赋给属性 `prototype`。

```
ShowDog.prototype = new Dog();
```

来看看我们到了哪一步：我们有构造函数ShowDog，可用来创建表演犬实例。我们还有一个表演犬原型，它是一个小狗实例。

下面来将对象图中的标签“Dog”改为“表演犬原型”，确保它准确地反映了这些对象扮演的角色。但别忘了，表演犬原型依然是一个小狗实例。



有了构造函数ShowDog和表演犬原型后，我们需要回过头去补充一些细节。我们将深入研究这个构造函数，并给表演犬原型添加一些属性和方法，让表演犬具备所需的额外行为。

该补全原型了

我们设置了表演犬原型，但当前它只是一个空的小狗实例。现在该给它添加属性和行为，让它更像表演犬原型了。

要给表演犬添加的属性和方法如下：

```
function ShowDog(name, breed, weight, handler) {  
  this.name = name;  
  this.breed = breed;  
  this.weight = weight;  
  this.handler = handler;  
}
```

别忘了，构造函数ShowDog看起来与构造函数Dog很像。表演犬也有属性name、breed和weight，但还有一个额外的属性handler，表示负责训练它的人。这些属性都是在表演犬实例中定义的。

```
ShowDog.prototype = new Dog();
```

```
Showdog.prototype.league = "Webville";  
  
ShowDog.prototype.stack = function() {  
  console.log("Stack");  
};  
  
ShowDog.prototype.bait = function() {  
  console.log("Bait");  
};  
  
ShowDog.prototype.gait = function(kind) {  
  console.log(kind + "ing");  
};  
  
ShowDog.prototype.groom = function() {  
  console.log("Groom");  
};
```

所有的表演犬都将加入Web镇联盟，因此我们在原型中添加这个属性。

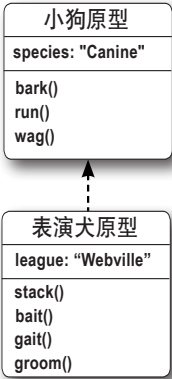
这些是表演犬支持的所有方法。这里让这些方法尽可能简单。

我们将这些属性都添加到表演犬原型中，让所有表演犬都继承它们。

在这里，我们获取充当表演犬原型的小狗实例，并给它添加新的属性和方法。

添加这些属性和方法后，表演犬原型看起来像表演犬了。下面再次修改对象图，创建一个表演犬并对其进行详尽的测试。看到结果后，Web镇小狗俱乐部肯定会激动万分。

我们说表演犬原型“扩展”了小狗原型。它继承了小狗原型的属性，并添加了一些新属性。



创建表演犬实例

至此，我们只需做最后一件事：创建一个ShowDog实例。这个实例将从表演犬原型那里继承表演犬特有的属性和方法。另外，由于表演犬原型是一个小狗实例，这个表演犬也将从小狗原型那里继承所有的小狗行为和属性。因此它像其他小狗一样，也能够发出叫声、奔跑和摇尾。

下面列出了前面编写的所有代码，还有创建表演犬实例的代码：

```
function ShowDog(name, breed, weight, handler) {
  this.name = name;
  this.breed = breed;
  this.weight = weight;
  this.handler = handler;
}

ShowDog.prototype = new Dog();

Showdog.prototype.league = "Webville";

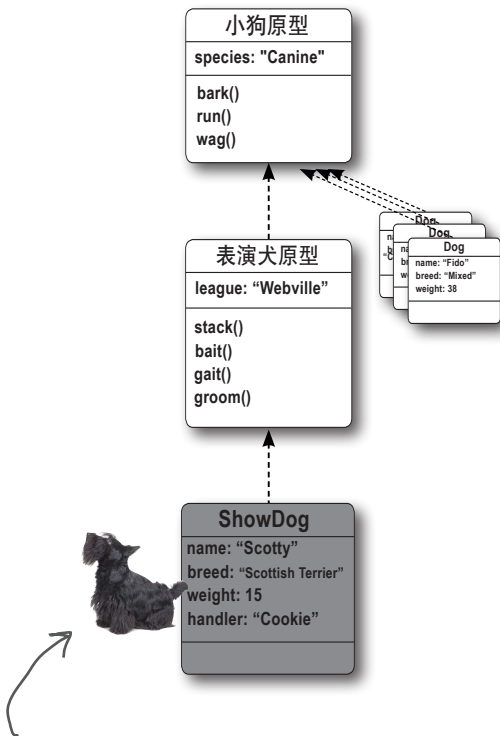
ShowDog.prototype.stack = function() {
  console.log("Stack");
};

ShowDog.prototype.bait = function() {
  console.log("Bait");
};

ShowDog.prototype.gait = function(kind) {
  console.log(kind + "ing");
};

ShowDog.prototype.groom = function() {
  console.log("Groom");
};
```

```
var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
```



这就是我们的表演犬实例。它继承了表演犬原型，而表演犬原型又继承了小狗原型。这正是我们希望的。如果你回过头去看第592页，将发现我们建立了所需的原型链。

新创建的表演犬实例scotty。

测试表演犬

请将前一页的所有代码以及下面的测试代码添加到一个网页中，对scotty进行详尽的测试。另外，顺便创建几个表演犬实例，并对它们进行测试。

```
scotty.stack();
scotty.bark();
console.log(scotty.league);
console.log(scotty.species);
```

这是我们得到的结果。→

```
JavaScript控制台
Stack
Scotty says Yip!
Webville
Canine
```



练习

轮到你了。在机器人行列中添加一些太空机器人。这些太空机器人能做到机器人能做的所有事，但它们还有一些额外的行为。请补全下面的代码，并对其进行测试。继续往下阅读前，请查看本章末尾的答案。

```
function SpaceRobot(name, year, owner, homePlanet) {

}

SpaceRobot.prototype = new _____;

_____.speak = function() {
    alert(this.name + " says Sir, If I may venture an opinion...");
};

_____.pilot = function() {
    alert(this.name + " says Thrusters? Are they important?");
};

var c3po = new SpaceRobot("C3PO", 1977, "Luke Skywalker", "Tatooine");
c3po.speak();
c3po.pilot();
console.log(c3po.name + " was made by " + c3po.maker);

var simon = new SpaceRobot("Simon", 2009, "Carla Diana", "Earth");
simon.makeCoffee();
simon.blinkLights();
simon.speak();
```



我们来更深入地研究一下前面创建的各个小狗。前面对Fido进行了测试，发现它确实是小狗，下面来看看它是否也是表演犬（我们认为它不是）。Scotty呢？前面通过测试确定了它是表演犬，但它也是小狗吗？我们不确定。下面顺便来测试一下Fido和Scotty，看看它们都是使用哪个构造函数创建的。

```
var fido = new Dog("Fido", "Mixed", 38);
if (fido instanceof Dog) {
    console.log("Fido is a Dog");
}
if (fido instanceof ShowDog) {
    console.log("Fido is a ShowDog");
}

var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
if (scotty instanceof Dog) {
    console.log("Scotty is a Dog");
}
if (scotty instanceof ShowDog) {
    console.log("Scotty is a ShowDog");
}

console.log("Fido constructor is " + fido.constructor);
console.log("Scotty constructor is " + scotty.constructor);
```

← 请运行这些代码，并在下面记录你得到的输出。

将你得到的输出记录在这里。



下一页列出了我们得到的输出。 →

研究练习结果

前一个练习的输出如下：

Fido是小狗，这符合我们的预期。输出没有指出Fido是表演犬，因此它肯定不是表演犬，这也合情合理。

Scotty既是小狗又是表演犬，这合情合理。但instanceof是怎么知道这点的呢？

这有点不可思议。结果表明，Fido和Scotty都是使用构造函数Dog创建的，但我们创建Scotty时，使用的可是构造函数ShowDog呀！

JavaScript控制台

```
Fido is a Dog
Scotty is a Dog
Scotty is a ShowDog
Fido constructor is function Dog...
Scotty constructor is function Dog...
```



想想为何会是这样的结果。首先，Fido显然只是小狗，而不是表演犬。事实上，这完全符合我们的预期，毕竟Fido是使用构造函数Dog创建的，而这个构造函数与表演犬一点关系都没有。

JavaScript控制台

```
Fido is a Dog
Scotty is a Dog
Scotty is a ShowDog
Fido constructor is function Dog...
Scotty constructor is function Dog...
```

接下来，Scotty既是小狗又是表演犬。这也合情合理，不过怎么会出现这样的结果呢？这是因为instanceof不仅考虑当前对象的类型，还考虑它继承的所有对象。Scotty虽然是作为表演犬创建的，但表演犬继承了小狗，因此Scotty也是小狗。

JavaScript控制台

```
Fido is a Dog
Scotty is a Dog
Scotty is a ShowDog
Fido constructor is function Dog...
Scotty constructor is function Dog...
```

再接下来，Fido的构造函数为Dog。这合情合理，因为它就是使用这个构造函数创建的。

JavaScript控制台

```
Fido is a Dog
Scotty is a Dog
Scotty is a ShowDog
Fido constructor is function Dog...
Scotty constructor is function Dog...
```

最后，Scotty的构造函数也是Dog。这不合理，因为它是使用构造函数ShowDog创建的。到底是怎么回事呢？先来看看这是如何得到的：查看属性scotty.constructor。由于我们没有显式地为表演犬设置这个属性，它将从小狗原型那里继承该属性。

JavaScript控制台

```
Fido is a Dog
Scotty is a Dog
Scotty is a ShowDog
Fido constructor is function Dog...
Scotty constructor is function Dog...
```

为何会这样呢？坦率地说，这是一个需要修复的漏洞。正如你看到的，如果我们不显式地设置表演犬原型的属性constructor，就没人会这样做。不过，即便我们不这样做，一切也都将正常运行，但访问scotty.constructor时，结果将不是预期的ShowDog，让人感到迷惑。

不用担心，下面就来修复这个问题。

最后的整理

只要对代码做最后的整理，就可以将其交付给Web镇小狗俱乐部了。总共需要修复两个小问题。

首先，正如你看到的，没有正确地设置表演犬实例的属性constructor；它们从小狗原型那里继承了这个属性。需要澄清的一点是，虽然代码都没问题，但给对象设置正确的构造函数是一种最佳实践，以免有一天另一位开发人员接手这些代码并查看表演犬对象的情况时感到迷惑。

为修复属性constructor不正确的问题，需要在表演犬原型中正确地设置它。这样，创建表演犬实例时，它将继承正确的constructor属性，如下所示：



```
function ShowDog(name, breed, weight, handler) {  
  this.name = name;  
  this.breed = breed;  
  this.weight = weight;  
  this.handler = handler;  
}
```

```
ShowDog.prototype = new Dog();  
ShowDog.prototype.constructor = ShowDog;
```

这就是需要做的全部工作。现在，如果再次检查Scotty，其属性constructor将正确无误，其他所有表演犬实例也如此。

获取表演犬原型，将其属性constructor显式地设置为构造函数ShowDog。

别忘了，这只是一种最佳实践，即便不这样做，代码也将像预期的那样运行。

请注意，无需对小狗原型这样做，因为其属性constructor的默认设置就是正确的。



练习

这是我们得到的输出，注意到Scotty的构造函数现在是ShowDog了。

请再次运行前面的测试，并核实表演犬实例Scotty的构造函数正确无误。

```
JavaScript控制台  
Fido is a Dog  
Scotty is a Dog  
Scotty is a ShowDog  
Fido constructor is function Dog...  
Scotty constructor is function ShowDog...
```

继续清理

还有一个地方可以清理：构造函数ShowDog的代码。再来看一眼这个构造函数：

```
function ShowDog(name, breed, weight, handler) {
  this.name = name;
  this.breed = breed;
  this.weight = weight;
  this.handler = handler;
}
```

← 你可能没有注意到，构造函数Dog也包含这些代码。

正如你在本书中看到的，每当我们发现重复的代码时，警报就会大响。就这里而言，既然构造函数Dog已经知道如何完成这些工作，为何不让它去做呢？另外，虽然这个示例的代码很简单，但有些构造函数可能使用复杂的代码来计算属性的初始值。因此创建继承另一个原型的构造函数时，都不应重复既有的代码。下面来修复这个问题——先重写代码，再详细介绍它们：

有个缩略语可以表示这种消除重复代码的理念：DRY (Don't Repeat Yourself, 不要自我重复)。所有厉害的程序员都这么说。

```
function ShowDog(name, breed, weight, handler) {
  Dog.call(this, name, breed, weight);
  this.handler = handler;
}
```

← 这行代码重用构造函数Dog中处理属性name、breed和weight的代码。

← 但这里依然需要处理属性handler，因为构造函数Dog对这个属性一无所知。

正如你看到的，在构造函数ShowDog中，我们调用了方法Dog.call，以此替换了那些重复的代码。这里的原理如下：call是一个内置方法，可对任何函数调用它（别忘了，Dog是一个函数）。Dog.call调用函数Dog，将一个用作this的对象以及函数Dog的所有实参传递给它。下面来详细介绍这行代码：

Dog是要调用的函数。

参数this指定了函数Dog中this指向的对象。

其他的实参与正常调用Dog时指定的实参相同。

这行代码调用构造函数Dog，并让其中的this指向当前的ShowDog实例。这样，构造函数Dog将设置当前ShowDog对象的属性name、breed和weight。

```
Dog.call(this, name, breed, weight);
```

这里调用了Dog的方法call，导致函数Dog被调用。这里之所以调用方法call，而不直接调用Dog，是因为这样可以控制this的值。

Dog.call详解

为何使用Dog.call来调用Dog呢？这有点不太好理解，下面来详细解释，先列出修改后的代码：

```
function ShowDog(name, breed, weight, handler) {  
    Dog.call(this, name, breed, weight);  
    this.handler = handler;  
}
```

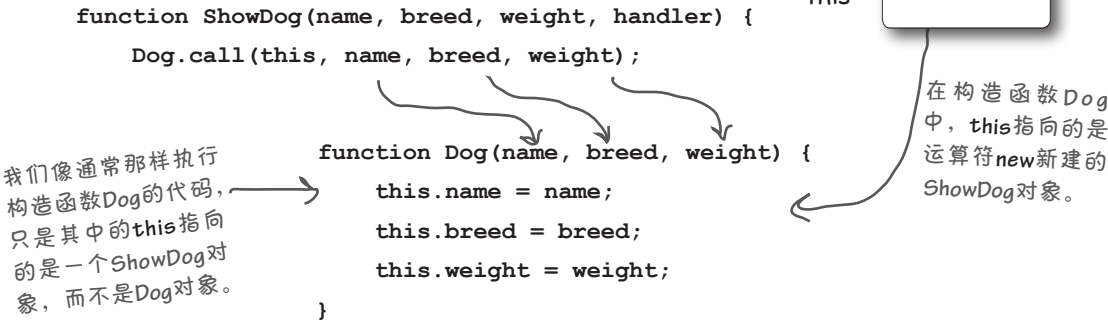
重用构造函数Dog中的代码来给属性name、breed和weight赋值。

但构造函数Dog对属性handler一无所知，因此必须在构造函数ShowDog中处理它。

其中的工作原理是什么样的呢？你可以这么想：首先，使用运算符new调用ShowDog。前面说过，运算符new新建一个空对象，并将其赋给构造函数ShowDog中的变量this。

```
var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
```

接下来，我们执行构造函数ShowDog的代码。在这个构造函数中，我们首先做的是使用方法call来调用Dog。这样调用Dog时，我们传入了this，还将形参name、breed和weight作为实参传递给它。



构造函数Dog执行完毕后（别忘了，我们调用它时没有使用运算符new，因此它不会返回任何对象），接着执行ShowDog中的其他代码，将形参handler的值赋给属性this.handler。接下来，因为我们调用ShowDog时使用了运算符new，所以将返回一个设置了属性name、breed、weight和handler的ShowDog实例。



最后的测试



Web镇小狗俱乐部肯定会喜欢我们的设计!



很好，你的设计精彩绝伦，Web镇小狗俱乐部肯定很满意。下面对各个小狗做最后一次测试，展示一下它们具备的能力。

```
function ShowDog(name, breed, weight, handler) {
    Dog.call(this, name, breed, weight);
    this.handler = handler;
}

ShowDog.prototype = new Dog();
ShowDog.prototype.constructor = ShowDog;
ShowDog.prototype.league = "Webville";
ShowDog.prototype.stack = function() {
    console.log("Stack");
};

ShowDog.prototype.bait = function() {
    console.log("Bait");
};

ShowDog.prototype.gait = function(kind) {
    console.log(kind + "ing");
};

ShowDog.prototype.groom = function() {
    console.log("Groom");
};

var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
var beatrice = new ShowDog("Beatrice", "Pomeranian", 5, "Hamilton");
fido.bark();
fluffy.bark();
spot.bark();
scotty.bark();
beatrice.bark();
scotty.gait("Walk");
beatrice.groom();
```

将ShowDog的所有代码整合在一起，并加入包含Dog代码的文件中，以便对其进行测试。

下面是一些测试代码。

创建几个小狗和几个表演犬。

将这些小狗和表演犬拉出来遛遛，核实它们都能做到正确的事。

JavaScript控制台

```
Fido says Woof!
Fluffy says Woof!
Spot says Yip!
Scotty says Yip!
Beatrice says Yip!
Walking
Groom
```

世上没有愚蠢的问题

问：前面调用构造函数Dog来创建用作表演犬原型的小狗实例时，没有指定任何实参。这是为什么？

答：因为对于这个小狗实例，我们唯一的要求是它继承了小狗原型。这个小狗实例不像Fido和Fluffy那样是具体的小狗，而只是一个继承小狗原型的通用小狗实例。

另外，所有继承表演犬原型的小狗都定义了自己的属性name、breed和weight。因此即便用作表演犬原型的小狗实例给这些属性设置了值，我们也根本看不到这些值，因为表演犬实例总是重写这些属性。

问：那么，在用作表演犬原型的小狗实例中，这些属性是怎么样呢？

答：根本没有给它们赋值，因此它们都是未定义的。

问：如果没有将ShowDog的属性prototype设置为一个小狗实例，结果将如何？

答：所有的表演犬都不会有问题，但它们不会继承小狗原型的任何行为。这意味着它们不能发出叫声、奔跑和摇尾，也不包含值为Canine的属性species。你可以自己试一试：把将ShowDog.prototype设置为new Dog()的那行代码注释掉，再尝试让Scotty发出叫声。结果将如何呢？

问：可以创建一个对象字面量，再将其用作原型吗？

答：可以。可将任意对象用作表演犬原型；当然，如果你这样做，表演犬将不会从小狗原型那里继承任何东西，而是继承你在这个对象字面量中定义的属性和方法。

问：对于将ShowDog.prototype设置为一个小狗实例的代码，我不小心将其放在了创建表演犬实例Scotty的代码之后。这会导致有些代码不能正确运行吗？请说明其中的原因。

答：创建表演犬实例Scotty时，将获取赋给ShowDog.prototype的原型。因此，对于将ShowDog.prototype设置为一个小狗实例的代码，如果你将其放在创建Scotty的代码后面，Scotty将是一个不同于其原型（构造函数ShowDog默认创建的对象）的对象，它不包含小狗原型的任何属性。创建构造函数ShowDog后，应首先设置其原型并在这个原型中添加所需的属性和方法，然后创建ShowDog实例。

问：如果我修改小狗原型的属性，如将属性species的值从Canine改为Feline，是否会影响到我之前创建的表演犬？

答：会影响。对原型作任何修改都将影响直接或间接继承该原型的所有实例。

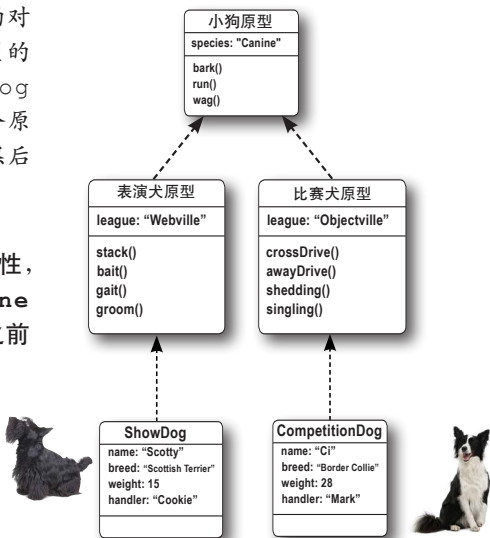
问：对原型链的长度有限制吗？

答：理论上没有，但实际上可能有。原型链越长，解析方法或属性时需要做的工作越多。虽然如此，但运行时系统通常极其擅长优化这种查找工作。

一般而言，设计不需要包含大量的继承层级。如果出现这样的情况，也许就该重新审视一下你的设计了。

问：如果还需创建另一类小狗，如比赛犬，该怎么办呢？可创建一个比赛犬原型，让它像表演犬原型一样继承小狗原型吗？

答：可以。为此，你需要再创建一个用作比赛犬原型的小狗实例。这样准备工作就做好了，然后就可以像创建表演犬原型那样创建比赛犬原型了。



小狗原型并非原型链的终点

前面介绍了两个原型链。第一个原型链包含从中派生出小狗对象的小狗原型；第二个原型链包含从中派生出表演犬的表演犬原型，而表演犬原型又是从小狗原型派生出来的。

在这两个原型链中，终点都是小狗原型吗？实际上不是，因为小狗原型是从Object派生出来的。

事实上，你创建的每个原型链的终点都是Object。这是因为对于你创建的任何实例，其默认原型都是Object，除非你对其进行了修改。

Object是什么

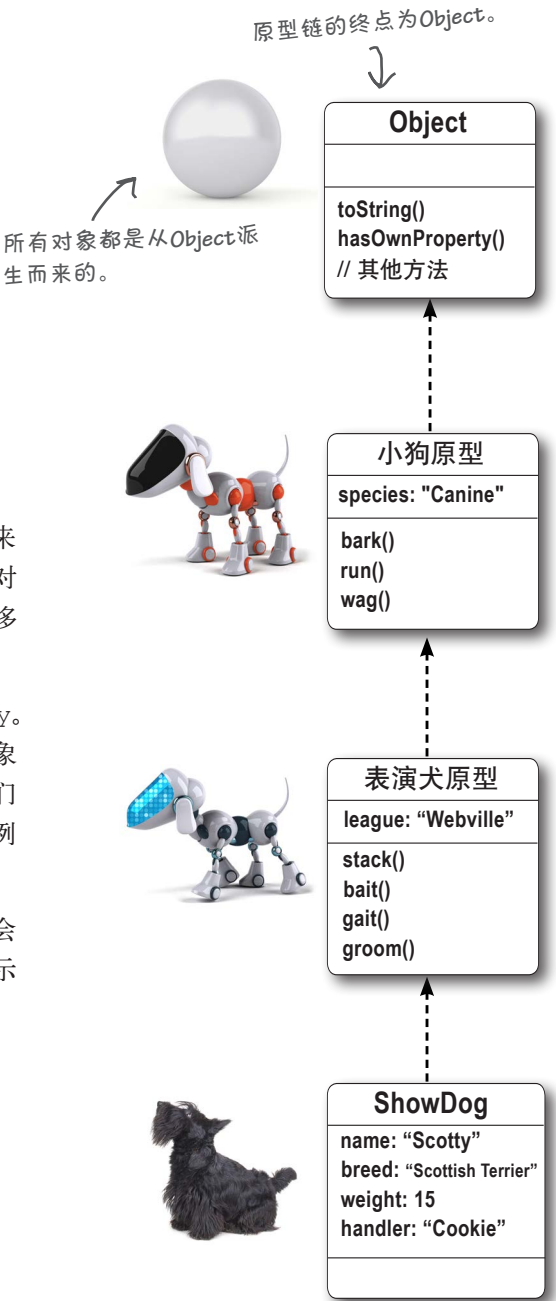
可将Object视为对象始祖，所有对象都是从它派生而来的。Object实现了多个重要的方法，它们是JavaScript对象系统的核心部分。在日常工作中，这些方法中的很多你都不会用到，但有几个经常会用到。

你在本章前面就见到过其中一个：hasOwnProperty。每个对象都继承了这个方法，因为归根结底，每个对象都是从Object派生而来的。别忘了，在本章前面，我们使用了方法hasOwnProperty来确定属性是在对象实例还是其原型中定义的。

Object定义的另一个方法是toString，但实例通常会重写它。这个方法返回对象的字符串表示。稍后将演示如何重写这个方法，为对象提供更准确的描述。

作为原型的Object

你可能没有意识到，你创建的每个对象都有原型，该原型默认为Object。你可将对象的原型设置为其他对象，就像我们对表演犬原型所做的那样，但所有原型链的终点都是Object。



充分发挥继承的威力之重写内置行为

继承内置对象时，可重写这些对象定义的方法。一种常见的情形是，重写Object定义的方法toString。所有对象都是从Object派生而来的，因此所有对象都可使用方法toString来获取其简单的字符串表示。例如，要在控制台中显示对象，可结合使用console.log和方法toString：

```
function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}

var toy = new Robot("Toy", 2013, "Avary");

console.log(toy.toString());
```

JavaScript控制台

[Object object]

从Object继承而来的方法toString在显示对象方面做得并不好。

如你所见，在将机器人对象toy转换为字符串方面，方法toString做得并不好。为解决这个问题，可重写方法toString，让其为机器人对象创建独特的字符串：

```
function Robot(name, year, owner) {
    // 相同的代码
}

Robot.prototype.toString = function() {
    return this.name + " Robot belonging to " + this.owner;
};

var toy = new Robot("Toy", 2013, "Avary");
```

```
console.log(toy.toString());
```

JavaScript控制台

Toy Robot belonging to Avary

情况好多了！现在使用的是自定义方法toString。

请注意，在有些情况下，会自动调用方法toString，无需你直接调用它。例如，如果你使用运算符+来拼接字符串和对象，JavaScript将自动调用方法toString将对象转换为字符串，再将该字符串与另一个字符串拼接起来。

```
console.log("Robot is: " + toy);
```

调用方法toString将对象toy转换为一个字符串，再进行拼接。如果toy重写了方法toString，将使用重写后的方法。

玩具？这个机器人运行着一个Arduino栈，可使用JavaScript进行控制！





危险地带

一旦你开始重写属性和方法，就很容易上瘾。重写内置对象的属性和方法时，一定要特别小心，否则可能改变其他依赖于这些属性来完成特定工作的代码的行为。

因此，如果你打算重写Object的属性，请先阅读这里的安全指南，否则可能会以意想不到的方式破坏既有的代码。（换言之，可能引发难以发现的bug。）



不可重写

千万不要重写Object的如下属性：

constructor

属性constructor指向与这个原型相关联的构造函数。

hasOwnProperty

你知道这个方法的作用。

isPrototypeOf

isPrototypeOf是一个方法，用于判断一个对象是否是另一个对象的原型。

propertyIsEnumerable

方法propertyIsEnumerable用于判断通过迭代对象的所有属性是否可访问指定的属性。

可以重写

熟悉原型并知道如何安全地重写属性后，就可以重写Object的如下属性了：

toString

toLocaleString是一个方法。它类似于toString，也将对象转换为字符串。通过重写这个方法，可提供描述对象的本地化字符串（如用你的母语表示的字符串）。

toLocaleString

valueOf

valueOf是另一个可以重写的方法。它默认情况下在返回当前对象，但通过重写，可让它返回你希望的其他值。

充分发挥继承的威力之扩展内置对象

你知道，通过给原型添加方法，可给其所有实例添加新功能。这不仅适用于自定义对象，还适用于内置对象。

就拿String对象来说吧：你在代码中使用了String对象的substring等方法，但如果要添加新方法，让所有String实例都能使用它，该如何办呢？可将前面介绍的对象扩展技巧用于String原型。

我们通常将字符串视为基本类型，但它们也有对象的形式。必要时，JavaScript会负责将字符串转换为对象。

假设我们想用一個名为cliche的方法扩展String原型。这个方法会在字符串包含一个众所周知的俗语时返回true。下面就是我们的代码：

```
String.prototype.cliche = function() {
    var cliche = ["lock and load", "touch base", "open the kimono"];

    for (var i = 0; i < cliche.length; i++) {
        var index = this.indexOf(cliche[i]);
        if (index >= 0) {
            return true;
        }
    }
    return false;
};
```

在这里，我们给String的原型添加了方法cliche。

我们定义了一些俗语。

接下来，我们使用String的方法indexOf来检查字符串中是否包含上述俗语。如果包含，就返回true。

请注意，this指的是方法cliche被调用的字符串。

下面来编写一些测试这个方法的代码：

为测试这个方法，我们来创建一些句子，其中的两个包含俗语。

```
var sentences = ["I'll send my car around to pick you up.",
    "Let's touch base in the morning and see where we are",
    "We don't want to open the kimono, we just want to inform them."];

for (var i = 0; i < sentences.length; i++) {
    var phrase = sentences[i];
    if (phrase.cliche()) {
        console.log("CLICHE ALERT: " + phrase);
    }
}
```

每个句子都是一个字符串，因此我们可以调用其方法cliche。

注意到创建字符串时，我们并没有结合使用new和构造函数String。我们调用方法cliche时，JavaScript会在幕后将每个字符串转换为String对象。

如果返回的是true，我们就知道字符串包含俗语。

测试俗语检查器

将前面的代码加入一个HTML文件中，然后打开浏览器并加载这个文件。

效果很好。要是我们能够说服美国的企业都安装这些代码就好了。

JavaScript控制台

```
CLICHE ALERT: Let's touch base in the morning
and see where we are
CLICHE ALERT: We don't want to open the kimono,
we just want to inform them.
```



小心!

给String等内置对象添加新方法时，千万要小心。

务必确保你为新方法选择的名称不与对象的既有方法发生冲突。链接其他代码时，一定要清楚这些代码包含的自定义扩展（同样，要注意可能存在的名称冲突）。最后，有些内置对象是不能扩展的，如Array。因此，着手给内置对象添加方法前，务必要做足功课。



轮到你了。请编写方法palindrome，它在字符串从前往后读和从后往前读一样时返回true。（只考虑字符串只包含一个单词的情况，不用考虑字符串为短语的情况。）将这个方法加入String.prototype中，再对其进行测试。答案见本章末尾。

JavaScript大统一理论

你接受了学习一门全新语言（也可能是你接触的第一门语言）的任务，祝贺你成功地完成了这项任务。到这里，你对JavaScript的了解几乎比你遇到的每个人都深。

不开玩笑了，如果你阅读到了这里，那么你就在成为JavaScript专家的路上走了很远，唯一欠缺的是设计和编写Web应用程序（也可以说是任何JavaScript应用程序）的经验。

这里的逻辑是这样的：大约有59亿人对JavaScript一无所知，对JavaScript有所了解的人只是极小一部分。这意味着你对JavaScript的了解几乎比你遇到的每个人都深。

使用对象改善生活

学习JavaScript这样的复杂主题时，很容易只见树木，不见森林。但对JavaScript有了全面了解后，再回过头来研究整个森林将容易得多。

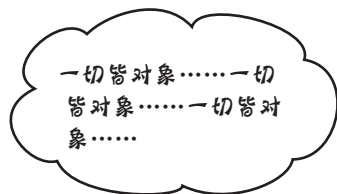
学习JavaScript时，你每次都只学习其一个方面：基本类型（可随时像使用对象一样使用它们）、数组（它们有点像对象）、函数（真是奇怪，它们像对象一样包含属性和方法）、构造函数（既像对象又像函数），还有对象本身。这些看起来都非常复杂。

掌握这些知识后，可以坐下来放松放松，深呼吸，回味“一切皆对象”的说法。

正如你看到的，一切皆对象。诚然，有一些基本类型，如布尔值、数字和字符串，但只要需要，你随时都可将它们视为对象。还有一些内置类型，如Date、Math和RegExp，但它们也都是对象。即便数组也是对象。正如你看到的，它们之所以看起来不同，只是因为JavaScript提供了一些出色的语法糖，让我们能够更轻松创建和访问对象。当然，还有对象本身，其中对象字面量简单易用，原型对象系统则提供了强大的功能。

函数呢？它们真的是对象吗？我们来验证这一点：

```
function meditate() {  
    console.log("Everything is an object...");  
}  
alert(meditate instanceof Object);
```

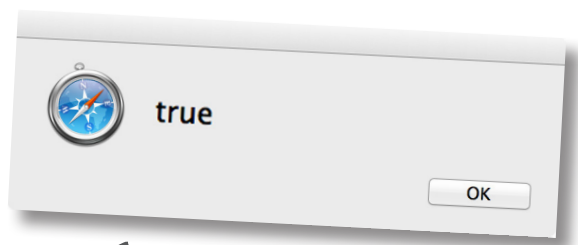


没错，函数确实是对象；但现在你不应该觉得这有什么奇怪的。毕竟，我们可以将函数赋给变量（就像将对象赋给变量一样），将其作为实参传递给函数（就像对象一样），从函数返回它们（就像对象一样）。我们还发现，函数甚至包含属性，如下所示：

Dog.constructor

↑
别忘了，这是一个函数。

↖
这是一个属性。



没错！函数也是对象。

另外，你完全可以给函数添加新属性——如果这样做有所帮助的话。最后，顺便说一句，方法也是对象的一个属性，只是该属性被设置为一个匿名函数表达式而已。

整合起来

从很大程度上说，JavaScript的威力和灵活性都要归功于可以将函数和对象作为一等值使用。只要想想我们学习过的编程概念（构造函数、闭包、创建可重用和扩展的对象、参数化函数的行为、等等），你就会发现，它们的强大威力都取决于你对高级对象和函数的认识。

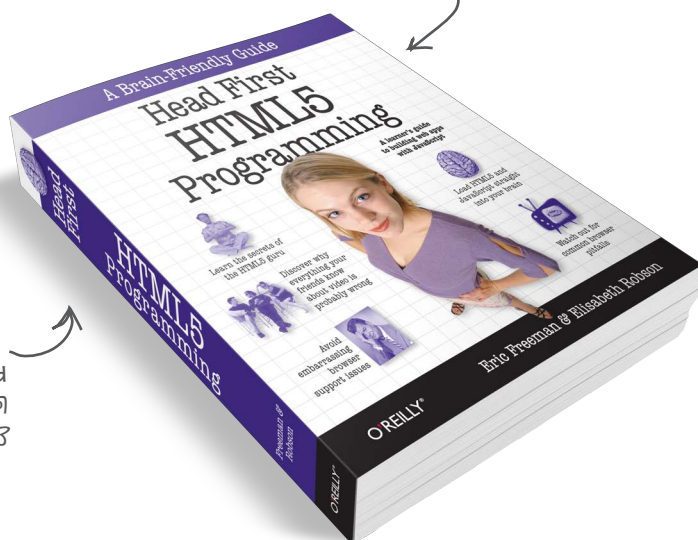
好了，现在你可以更进一步了。

继续学习

至此，你掌握了所有的基本知识，该接着往下走了。现在，你可以将自己具备的浏览器及其编程接口方面的经验付诸应用了。请拿起《Head First HTML5 Programming中文版》一书，它将引领你学会在应用程序中添加定位、画布绘画、本地存储和后台异步化（web worker）等功能。不过，放下本书前，务必阅读附录，其中列出了其他等待你去探索的重要主题。

这是一个变化迅速的主题，因此着手阅读《Head First HTML5 Programming中文版》前，请务必访问 <http://wickedlysmart.com/javascript>，其中列出了最新的推荐读物以及本书的增补和修订。

请务必访问<http://wickedlysmart.com/javascript>，阅读其中列出的增补内容。另外，你的下一项任务是阅读这本书。请接受这项任务。

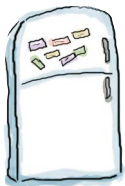




要点

- JavaScript对象系统使用原型式继承。
- 使用构造函数创建对象实例时，实例包含自己的自定义属性，还有构造函数中方法的副本。
- 给构造函数的原型添加属性后，使用这个构造函数创建的实例都将继承这些属性。
- 通过在原型中定义属性，可减少对象包含的重复代码。
- 要重写原型中的属性，只需在实例中添加该属性即可。
- 构造函数有默认的原型，你可通过构造函数的属性prototype来访问它。
- 可将你自己创建的对象赋给构造函数的属性prototype。
- 使用自定义的原型对象时，务必将原型的属性constructor设置为相应的构造函数，以保持一致性。
- 给原型添加属性后，继承该原型的所有实例都将立即继承这些属性，即便是以前创建的实例也不例外。
- 要确定属性是否是在实例中定义的，可对实例调用方法hasOwnProperty。
- 要调用函数并指定函数体中this指向的对象，可调用其方法call。
- 归根结底，所有原型和对象都是从Object派生而来的。
- Object包含所有对象都将继承的属性和方法，如toString和hasOwnProperty。
- 可给内置对象（如Object和String）添加属性，也可重写它们的既有属性，但这样做时必须小心，因为你所作的修改可能带来深远的影响。
- 在JavaScript中，一切几乎皆是对象，包括函数、数组和众多的内置对象，还有你自己创建的自定义对象。



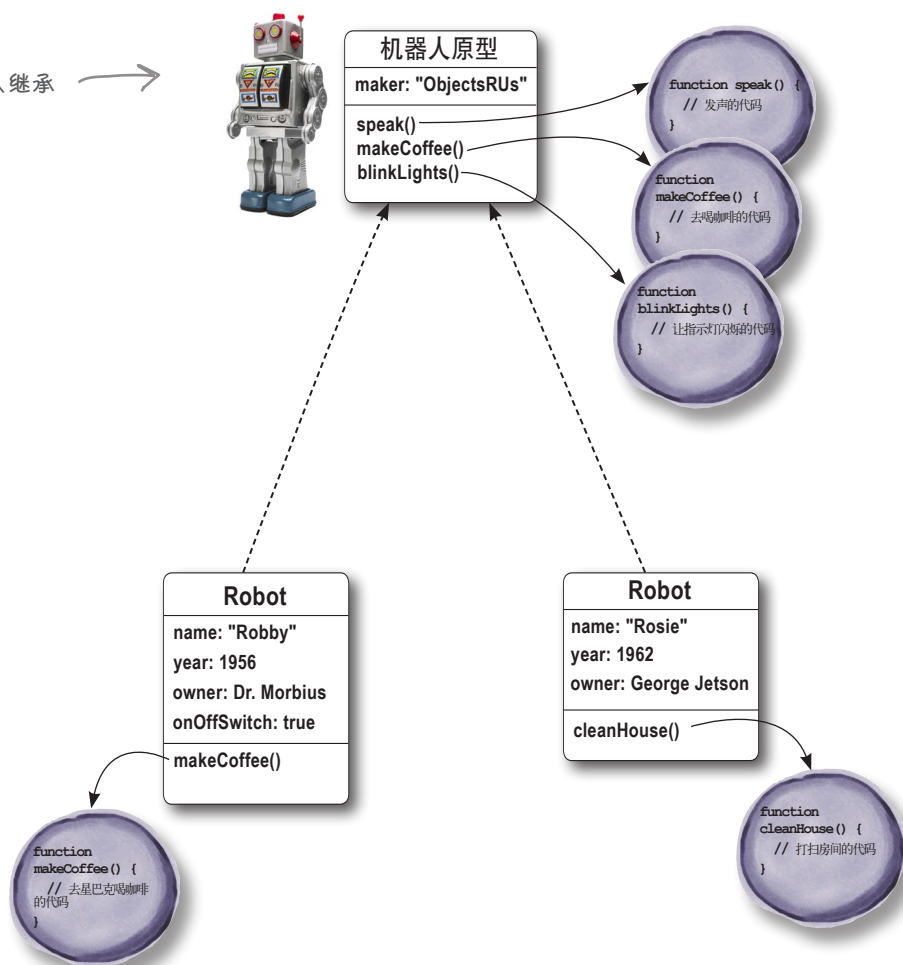


代码冰箱贴答案

冰箱上有一个对象图，但有人将它打乱了，你能帮忙将它组装起来吗？在这个对象图中，有两个机器人原型的实例：其中一个1956年生产的Robby，它归Dr. Morbius所有，拥有一个开关，喝咖啡时去星巴克；另一个是1962年生产的Rosie，它归George Jetson所有，能够打扫房间。需要指出的是，有些冰箱贴可能是多余的，祝你好运！

答案如下。

可供机器人继承
的原型。





练习答案

还记得前面包含机器人Robby和Rosie的对象图吗？现在就来实现它。我们编写了构造函数Robot和一些测试代码，而你需要做的是设置机器人原型并创建这两个机器人对象，再通过运行对这些代码进行测试。答案如下。

```
function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}
Robot.prototype.maker = "ObjectsRUs";
Robot.prototype.speak = function() {
    alert("Warning warning!!");
};
Robot.prototype.makeCoffee = function() {
    alert("Making coffee");
};
Robot.prototype.blinkLights = function() {
    alert("Blink blink!");
};
var robbie = new Robot("Robbie", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");
robbie.onOffSwitch = true;
robbie.makeCoffee = function() {
    alert("Fetching a coffee from Starbucks.");
};
rosie.cleanHouse = function() {
    alert("Cleaning! Spic and Span soon...");
};
console.log(robbie.name + " was made by " + robbie.maker +
    " in " + robbie.year + " and is owned by " + robbie.owner);
robbie.makeCoffee();
robbie.blinkLights();
console.log(rosie.name + " was made by " + rosie.maker +
    " in " + rosie.year + " and is owned by " + rosie.owner);
rosie.cleanHouse();
```

这是基本的构造函数Robot。

这里给原型添加了属性maker。

还添加了所有机器人都将继承的三个方法。

这里创建机器人Robbie和Rosie。

这里给Robbie添加了一个自定义属性，还添加了一个去星巴克喝咖啡的自定义方法。

Rosie也有一个打扫房间的自定义方法。（为何打扫房间的事就得女机器人做？）

这是我们得到的输出
(除这些输出外，还有一些这里没显示的提示框)。

JavaScript控制台

```
Robbie was made by ObjectsRUs in 1956
and is owned by Dr. Morbius
Rosie was made by ObjectsRUs in 1962
and is owned by George Jetson
```



有一个机器人游戏使用了机器人Robby和Rosie。这个机器人游戏的代码如下。在这个游戏中，玩家的等级达到42后，机器人将具备一项新功能：发射激光束。请补全下面的代码，使得在玩家的等级达到42后，机器人Robby和Rosie都能发射激光束。答案如下。

```
function Game() {
    this.level = 0;
}

Game.prototype.play = function() {
    // 让玩家玩游戏的代码
    this.level++;
    console.log("Welcome to level " + this.level);
    this.unlock();
};

Game.prototype.unlock = function() {
    if (this.level === 42) {
        Robot.prototype.deployLaser = function () {
            console.log(this.name + " is blasting you with laser beams.");
        };
    }
};

function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}

var game = new Game();
var robbie = new Robot("Robby", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");

while (game.level < 42) {
    game.play();
}

robbie.deployLaser();
rosie.deployLaser();
```

每次玩游戏时都调用unlock，但这个方法的作用仅当等级达到42时才会显现出来。

这个游戏的诀窍在这里：等级达到42后，在原型中添加一个新方法。这意味着所有机器人都将继承发射激光的能力！

JavaScript控制台

```
Welcome to level 1
Welcome to level 2
Welcome to level 3
...
Welcome to level 41
Welcome to level 42
Rosie is blasting you with laser beams.
```

这是我们得到的输出示例。补全代码后，请尝试玩玩这个游戏，看看是哪个机器人先发射激光束。





我们给机器人Robby和Rosie添加了一项新功能，现在它们能够在发生错误时通过方法reportError进行报告。请研究下面的代码，特别要注意这个方法是从什么地方获取错误信息的，以及这个方法是否是在机器人原型中定义的。

答案如下。

```
function Robot(name, year, owner) {
  this.name = name;
  this.year = year;
  this.owner = owner;
}

Robot.prototype.make = "ObjectsRUs";
Robot.prototype.errorMessage = "All systems go.";
Robot.prototype.reportError = function() {
  console.log(this.name + " says " + this.errorMessage);
};

Robot.prototype.spillWater = function() {
  this.errorMessage = "I appear to have a short circuit!";
};

var robby = new Robot("Robby", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");
```

方法reportError只是使用了errorMessage的值，因此它并没有重写这个属性。

方法spillWater给this.errorMessage赋值。这将在这个方法被调用的机器人对象中重写原型中的这个属性。

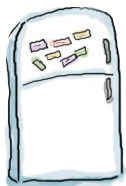
```
rosie.reportError();
robby.reportError();
robby.spillWater();
rosie.reportError();
robby.reportError();
```

我们对Robby调用方法spillWater，因此Robby将获得自己的属性errorMessage，即重写原型中的这个属性。

```
console.log(robby.hasOwnProperty("errorMessage"));
console.log(rosie.hasOwnProperty("errorMessage"));
```

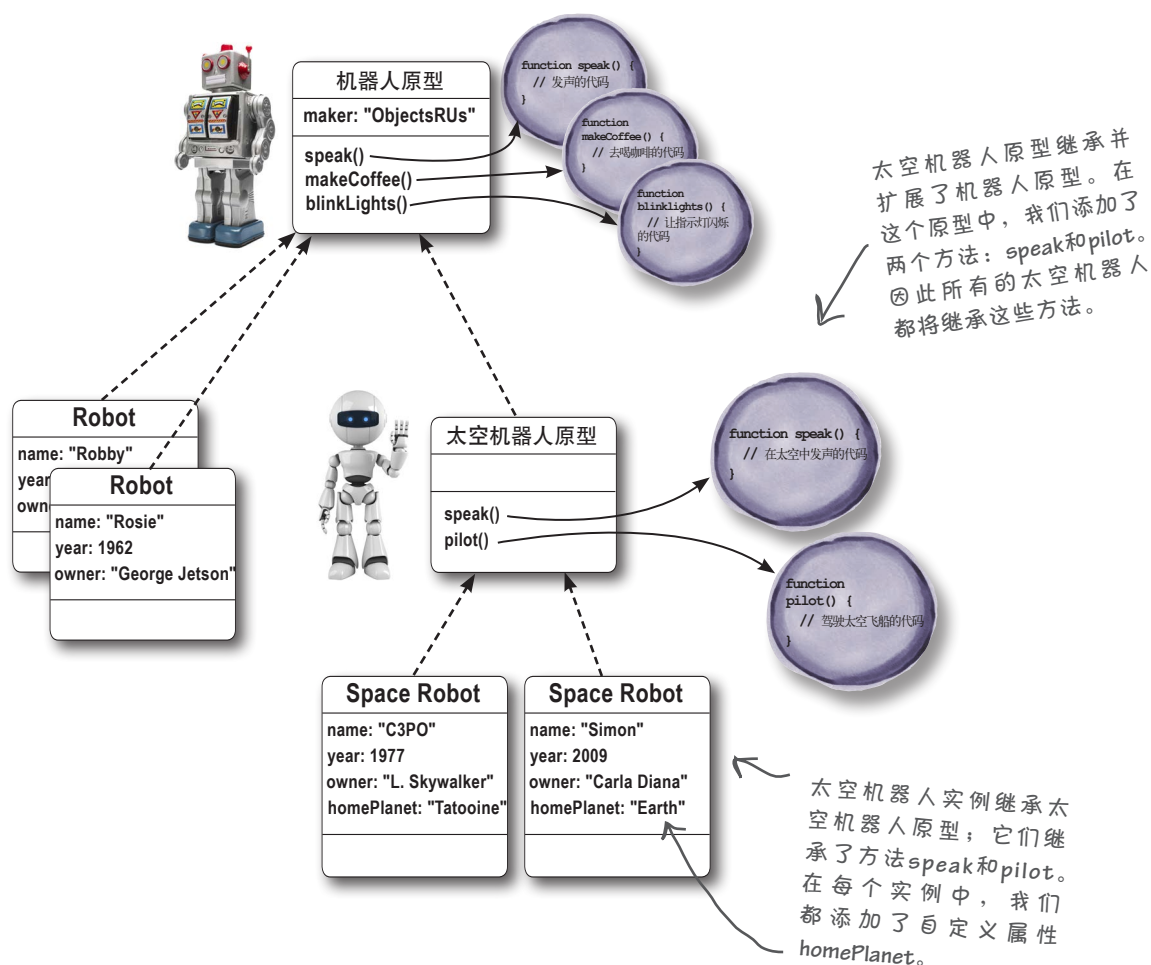
true
false

但我们从未对Rosie调用方法spillWater，因此它继承了原型中的这个属性。



代码冰箱贴答案

我们在冰箱上又贴了一个对象图，但也被人弄乱了。你能帮忙将其重新组合起来吗？与前面的对象图相比，这个对象图新增了一些太空机器人，它们继承了机器人的属性，重写了机器人的方法`speak`，还新增了方法`pilot()`和属性`homePlanet`。答案如下。





轮到你了。在机器人行列中添加一些太空机器人。这些太空机器人能做到机器人能做的所有事，但它们还有一些额外的行为。请补全下面的代码，并对其进行测试。答案如下。

```
function SpaceRobot(name, year, owner, homePlanet) {
    this.name = name;
    this.year = year;
    this.owner = owner;
    this.homePlanet = homePlanet;
}
```

← 构造函数SpaceRobot与构造函数Robot类似，但处理了太空机器人实例特有的额外属性homePlanet。

```
SpaceRobot.prototype = new Robot();
```

← 我们希望太空机器人原型继承机器人原型，因此将一个机器人实例赋给构造函数SpaceRobot的属性prototype。

```
SpaceRobot.prototype.speak = function() {
    alert(this.name + " says Sir, If I may venture an opinion...");
};
```

← 在太空机器人原型中
← 添加了这两个方法。

```
SpaceRobot.prototype.pilot = function() {
    alert(this.name + " says Thrusters? Are they important?");
};
```

```
var c3po = new SpaceRobot("C3PO", 1977, "Luke Skywalker", "Tatooine");
c3po.speak();
c3po.pilot();
console.log(c3po.name + " was made by " + c3po.maker);
```

```
var simon = new SpaceRobot("Simon", 2009, "Carla Diana", "Earth");
simon.makeCoffee();
simon.blinkLights();
simon.speak();
```

这是我们得到的输出（还有一些提示框，这里没有显示出来）。 →

JavaScript控制台

C3PO was made by ObjectsRUs



轮到你了。请编写方法`palindrome`，它在字符串从前往后读和从后往前读一样时返回`true`。将这个方法加入`String.prototype`中，再对其进行测试。答案如下（只考虑了字符串只包含一个单词的情况）。

```
String.prototype.palindrome = function() {
  var len = this.length-1;
  for (var i = 0; i <= len; i++) {
    if (this.charAt(i) !== this.charAt(len-i)) {
      return false;
    }
    if (i === (len-i)) {
      return true;
    }
  }
  return true;
};
```

首先，确定字符串的长度。

接下来，迭代字符串中的每个字符，并测试位于索引`i`处的字符，看它是否与位于索引`len - i`处的字符（即从后往前数的相应字符）是否相同。

如果它们不同，就返回`false`，因为这意味着这个字符串不是回文。

如果到达字符串正中央或者循环已结束，就返回`true`，因为这意味着这个字符串是回文。

如果到达字符串正中央或者循环已结束，就返回`true`，因为这意味着这个字符串是回文。

```
var phrases = ["eve", "kayak", "mom", "wow", "Not a palindrome"];
for (var i = 0; i < phrases.length; i++) {
  var phrase = phrases[i];
  if (phrase.palindrome()) {
    console.log("'" + phrase + "' is a palindrome");
  } else {
    console.log("'" + phrase + "' is NOT a palindrome");
  }
}
```

这是一些用于测试的单词。

我们迭代数组中的每个单词，并对其调用方法`palindrome`。如果结果为`true`，就说明这个单词是回文。



高阶解决方案

```
String.prototype.palindrome = function() {
  var r = this.split("").reverse().join("");
  return (r === this.valueOf());
}
```

在这里，我们首先将字符串转换为一个字符数组，其中包含字符串中的所有字符。接下来，我们反转该数组中字符的排列顺序，再将其组合成一个字符串。如果原来的字符串与新字符串相同，就说明它是回文。注意，这里必须使用`valueOf`，因为`this`是一个对象，而不像`r`那样是基本类型。如果不使用`valueOf`，比较的将是一个字符串和一个对象，而在这种情况下，即便`this`是回文，它们也不相等。

如果这是本书末尾，那该多好！
要是再也没有了要点、填字游
戏、JavaScript代码清单什么的，那该
多好！但这也许不过是白日做梦。



祝贺你终于读完了！

当然，还有附录。

还有索引。

还有配套网站等。

实际上，永远都没有解脱的时候。

附录 遗漏内容

✧未涉足的十大主题✧



我们介绍了大量的基本知识，本书也即将接近尾声。我们会想念你的。放手让你去独闯天涯前，我们想在你的行囊中再准备点东西，不然会不放心。这个附录篇幅较短，无法囊括你需要知道的一切。实际上，它最初确实涵盖了你需要知道但本书前面未介绍的一切JavaScript编程知识，但字体小得谁都看不清。因此，我们删除了其中的大部分内容，只留下最重要的十大主题。

等你阅读完这个附录后，本书就真的结束了。不过别忘了索引。（也是必读的！）

#1. jQuery

jQuery是一个JavaScript库，旨在减少和简化处理DOM和添加视觉效果的JavaScript代码。jQuery极受欢迎，被大家广泛使用，其插件模型使其是可扩展的。

当前，使用jQuery能做的事情，使用JavaScript也都能做（前面说过，jQuery只是一个JavaScript库），但使用jQuery确实可以减少需要编写的代码量。

jQuery的流行说明了一切，但如果你对它一无所知，还是需要花些时间才能熟悉的。这里只介绍可使用jQuery来完成的几项工作，如果你觉得它不错，建议你进行更深入的研究。

还记得本书前面编写的window.onload函数吗？它们类似于下面这样：

```
window.onload = function() {
    alert("the page is loaded!");
}
```

下面是等效的jQuery代码：

```
$(document).ready(function() {
    alert("the page is loaded!");
});
```

在文档准备就绪后，调用这里定义的函数。

你还可以进一步简化：

```
$(function() {
    alert("the page is loaded!");
});
```

非常简单，但正如你看到的，需要一段时间才能习惯。不用担心，你很快就会对它了如指掌。

从DOM获取元素呢？这正是jQuery的闪光点。假设网页中有一个id为buynow的<a>元素，而你想给它指定一个单击事件处理程序（本书前面多次这样做过），可以像下面这样做：

```
$(function() {
    $("#buynow").click(function() {
        alert("I want to buy now!");
    });
});
```

这些代码是什么意思呢？我们首先指定了一个要在网页加载完毕后调用的函数。

接下来，我们获取id为buynow的元素。（请注意，jQuery采用CSS中的元素选择语法。）

然后，对获得的元素调用jQuery方法click，以指定该元素的单击事件处理程序。

当前，熟悉jQuery对就业和理解他人编写的代码都大有裨益。

这只是开始，要给网页中的所有元素指定单击处理程序，同样易如反掌：

```
$(function() {
    $("a").click(function() {
        alert("I want to buy now!");
    });
});
```

为此，我们需要使用相应的标签名。

请将这些代码与实现相同功能的JavaScript代码进行比较。

使用jQuery，还可以完成复杂得多的工作：

```
$(function() {
    $("#playlist > li").addClass("favorite");
});
```

获取id为playlist的元素中所有的元素。

再给这些元素都添加类favorite。

这里提供这些jQuery代码只是想让你热热身，jQuery实际上能够让你完成比这复杂得多的任务。

jQuery还能够让你对界面元素进行有趣的变换：

```
$(function() {
    $("#specialoffer").click(function() {
        $(this).fadeOut(800, function() {
            $(this).fadeIn(400);
        });
    });
});
```

这些代码让id为specialoffer的元素以不同的速度渐隐再渐现。

正如你看到的，使用jQuery可以做很多事情。这里没有提及的是，使用jQuery还可以与Web服务通信；另外，还有很多支持jQuery的插件。如果你对jQuery感兴趣，最佳的选择是通过浏览器访问<http://jquery.com/>，并详细研究其中的教程和文档。

另外，也请阅读《Head First jQuery 中文版》一书。

#2. 更多地使用DOM

本书前面提及了使用DOM可做的一些事情，但需要学习的还有很多。表示网页中文档的对象（即document）以及各种元素对象都包含大量的属性和方法，你可使用它们来操作网页以及与网页交互。

你已经知道如何使用document.getElementById和document.getElementsByTagName来获取网页中的元素，但对象document还提供了下述用于获取元素的方法：

`document.getElementsByClassName` 向这个方法传入类名，它将返回一个NodeList，其中包含class特性为指定值的所有元素。

`document.getElementsByName` 这个方法获取name特性为指定值的所有元素。

`document.querySelector` 这个方法将一个选择器（类似于CSS选择器）作为参数，并返回匹配的第一个元素。

`document.querySelectorAll` 这个方法也将一个选择器作为参数，但返回一个NodeList，其中包含与选择器匹配的所有元素。

下面演示了如何使用document.querySelector来获取一个列表项元素。这个元素的class特性为song，嵌套在一个id为playlist的元素中：

```
var li = document.querySelector("#playlist .song");
```

这个选择器的含义是，先查找id为playlist的元素，再在其中查找第一个特性class为song的元素。

注意到这个选择器很像CSS选择器。

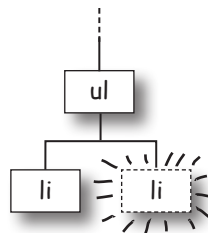
如果要使用代码在网页中添加新元素，该如何办呢？可结合使用对象document的方法以及元素对象的方法，如下所示：

```
var newItem = document.createElement("li");
newItem.innerHTML = "Your Random Heart";

var ul = document.getElementById("playlist");
ul.appendChild(newItem);
```

首先，创建一个新的元素，并将其内容设置为一个字符串。

然后，获取要将该元素作为子元素加入其中的元素，再将元素附加到这个元素的末尾。



可使用JavaScript对DOM执行的操作还有很多。有关这方面的深入介绍，请参阅《Head First HTML5 Programming中文版》。

#3. 对象window

你听说过DOM，但应该知道还有BOM，即浏览器对象模型（browser object model）。这其实并非官方标准，但所有浏览器都通过对象window来支持它。使用属性window.onload时，你顺便熟悉了对象window。你肯定还记得，可将一个事件处理程序赋给这个属性，以便在网页加载完毕后调用它。

使用方法alert和prompt时，也涉及了对象window，但这一点不那么明显。不明显的原因是，window是一个充当全局命名空间的对象。当你声明全局变量或定义全局函数时，它将被作为一个属性存储在对象window中。因此，每次调用alert时，其实也都可调用window.alert，因为它们都是等价的。

另外，使用对象document（如使用document.getElementById从DOM中获取元素）时，也涉及了对象window，只是你没有意识到而已。对象document是对象window的一个属性，因此可将document.getElementById改写为window.document.getElementById。但与调用alert时一样，可以不这样做，因为window是全局对象。我们使用属性和方法时，如果没有指定它所属的对象，默认指的都是window。

除了充当全局对象，提供属性onload以及方法alert和prompt外，对象window还提供了其他既有浏览器的有趣属性和方法。例如，大家常常获取浏览器窗口的宽度和高度，以根据浏览器的尺寸来定制网页。要访问这些值，可以像下面这样做：

```
window.innerWidth
window.innerHeight
```

使用这些属性来获取浏览器窗口的宽度和高度（单位为像素）。请注意，老式浏览器可能没有暴露这些属性。

有关对象window的详细信息，请参阅W3C文档（<http://www.w3.org/html/wg/drafts/html/CR/browsers.html#the-window-object>）。下面是对象window中几个常用的方法和属性：

```
window.close()
```

这个方法关闭浏览器窗口。

```
window.setTimeout()
window.setInterval()
```

这些方法你都很熟悉，它们是对象window提供的。

```
window.print()
```

使用打印机打印网页。

```
window.confirm()
```

这个方法类似于prompt，但给用户提供了选择，让你能够单击OK或Cancel按钮。

```
window.history
```

这个属性是一个对象，包含浏览历史记录。

```
window.location
```

这个属性是当前所显示网页的URL。你也可设置这个属性，让浏览器加载指定的新网页。

#4. arguments

在每个函数中，都有一个名为arguments的对象可供使用。形参列表中没有这个对象，但每当函数被调用时，你都可以通过变量arguments来使用它。

对象arguments包含传递给函数的所有实参，可像使用数组那样使用它。通过使用arguments，你可创建这样的函数：接受数量可变的实参，并根据传入的实参数量执行不同的操作。下面的代码演示了如何使用arguments：

在这个函数中，没有定义任何形参，只使用了对象arguments。

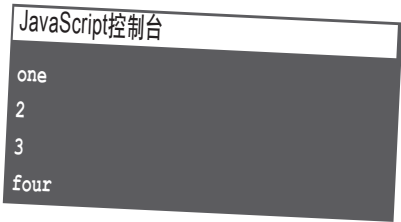
与数组一样，arguments也包含属性length。

```
function printArgs() {
  for (var i = 0; i < arguments.length; i++) {
    console.log(arguments[i]);
  }
}
```

可使用数组表示法来访问每个实参。

调用printArgs，并向它传递4个实参。

```
printArgs("one", 2, 1+2, "four");
```



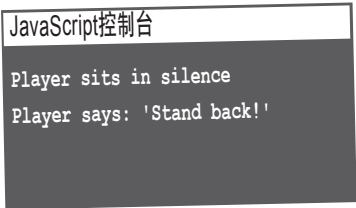
虽然arguments看起来像数组，但它其实并不是数组，而是一个对象。它包含属性length，你可迭代它并使用方括号表示法来访问其中的实参，但它与数组的类似性仅此而已。另外，注意你可以在同一个函数中同时使用形参和对象arguments。下面再来看一段代码，看看如何编写实参数量可变的函数：

可像通常那样定义形参。在这里，我们使用了一个形参来指定该如何使用这个函数。

```
function emote(kind) {
  if (kind === "silence") {
    console.log("Player sits in silence");
  } else if (kind === "says") {
    console.log("Player says: " + arguments[1] + "");
  }
}
```

如果第一个实参为silence，就不需要其他的实参。
如果第一个实参为says，就使用arguments[1]来获取第二个实参。

```
emote("silence");
emote("says", "Stand back!");
```





#5. 处理异常

JavaScript是一种非常宽容的语言，但还是会时不时地出问题——严重到浏览器无法继续执行代码。在这种情况下，网页将停止工作。如果你此时查看控制台，很可能会看到错误消息。来看一个将导致错误的代码示例。首先，创建一个只包含一个元素的简单HTML页面：

```
<div id="message"></div>
```

接下来，添加如下JavaScript代码：

```
window.onload = function() {
    var message = document.getElementById("message");
    message.innerHTML = "Here's the message!";
};
```

← 这些代码中存在一个错误，你看出来了吗？

在浏览器中加载这个网页，确保打开了控制台。你将看到一条错误消息。你能看出来问题出在什么地方吗？<div>元素的id拼写不正确，导致代码尝试获取<div>元素时以失败告终。因此变量message为null，进而无法访问这个变量的属性innerHTML。

JavaScript控制台
Uncaught TypeError: Cannot set property 'innerHTML' of null

像这样导致代码无法继续执行的错误被称为异常。JavaScript提供了一种名为try/catch的机制，让你能够发现并捕获异常。这里的基本理念是，如果能够捕获异常，就可避免代码停止执行，从而采取补救措施（尝试执行其他操作，向用户提供不同的体验等）。

try/catch

try/catch的用法如下：将要尝试执行的代码放在一个try块中，再编写一个catch块，其中包含try块中的代码出现错误时将执行的代码。在关键字catch后面，紧跟着用括号括起的变量名（这个变量很像函数形参）。只要出现问题，就将捕获到异常，并将一个与异常相关的值（通常是一个Error对象）赋给这个变量。下面演示了try/catch语句的用法：

```
window.onload = function() {
    try {
        var message = document.getElementById("message");
        message.innerHTML = "Here's the message!";
    } catch (error) {
        console.log("Error! " + error.message);
    }
};
```

将需要执行的代码移到try块中。

尝试将message（其值为null）的属性innerHTML设置为一个字符串。

如果try块中的代码引发了异常，将执行这行代码：在控制台中显示对象error的属性message。接下来，将接着执行try/catch语句后面的代码。

在这里，可根据error的值采取更明智的措施。

#6. 使用addEventListener添加事件处理程序

在本书中，我们使用对象属性来给事件指定处理程序。例如，要处理加载事件时，我们将一个事件处理程序赋给属性window.onload；要处理按钮单击事件时，我们将一个事件处理程序赋给按钮的属性onclick。

这种指定事件处理程序的方式很方便，但在有些情况下，可能需要更通用的事件处理程序指定方式。例如，如果要给一个事件指定多个处理程序，就不能使用属性（如onload）来完成这种任务，而必须使用方法addEventListener：

对对象window调用方法
addEventListener，以注册
一个加载事件处理程序。

向它传递了三个参数：用字符串“load”表示的事件名称……

指向事件处理程序的引用……

还有一个标志，它指出我们是否要让事件向上传递（这将在稍后解释）。

```

window.addEventListener("load", init, false);
function init() {
    // 网页加载完毕
}

```

因此加载事件发生时，
将调用函数init。

你可以再指定一个加载事件处理程序。为此，可再次调用addEventListener，并将第二个实参设置为指向另一个事件处理程序的引用。在你想将初始化代码放在两个函数中时，这提供了极大的方便；但别忘了，你无法知道先调用哪个处理程序，因此设计代码时务必考虑这一点。

addEventListener的第三个实参指定是否将事件向上传递给父元素。就加载事件而言，这无关紧要，因为对象window位于最顶层；但如果你在<div>元素中嵌套了一个元素，并希望用户单击元素时，<div>元素也将收到这个事件，就可将这个实参设置为true（而不是false）。

完全可以在使用事件属性（如onload）的同时使用addEventListener，这不会有任何问题。另外，对于使用addEventListener添加的事件处理程序，还可使用removeEventListener来删除它，如下所示：

使用对象window的属性onload指定了一个加载事件处理程序。

并使用addEventListener给<div>元素指定了一个单击事件处理程序。

用户单击<div>元素时，我们使用removeEventListener删除已指定的单击事件处理程序。

```

window.onload = function() {
    var div = document.getElementById("clickme");
    div.addEventListener("click", handleClick, false);
};
function handleClick(e) {
    var target = e.target;
    alert("You clicked on " + target.id);
    target.removeEventListener("click", handleClick, false);
}

```

IE8和更早版本中的事件处理

本书处理过多种不同的事件：鼠标单击事件、网页加载事件、按键事件等。但愿你使用的是现代浏览器，让这些代码能够正常地运行。然而，如果你要编写处理事件的网页（哪个网页不需要这样做呢？），并担心有些用户使用的是Internet Explorer（IE）8或更早的版本，就需要知道一个与事件处理相关的问题。

可惜在IE9之前，IE处理事件的方式不同于其他浏览器。无论用户使用的是哪种浏览器，你都可使用onclick和onload等属性来指定事件处理程序，但较旧的IE浏览器处理事件对象的方式不同。另外，在IE9之前，IE都不支持标准化方法addEventListener。你必须对下面的问题心中有数。

- ❑ IE8和更早的版本确实支持大多数可用于指定事件处理程序的“on”属性。
- ❑ IE8和更早的版本使用方法attachEvent而不是addEventListener。
- ❑ 事件触发导致事件处理程序被调用时，IE8和更早的版本将事件对象存储在对象window中，而不将其传递给事件处理程序。

因此，要确保代码在包括IE8和更早版本的所有浏览器中都能正确地运行，可像下面这样处理这些差异：

```

window.onload = function() {
    var div = document.getElementById("clickme");
    if (div.addEventListener) {
        div.addEventListener("click", handleClick, false);
    } else if (div.attachEvent) {
        div.attachEvent("onclick", handleClick);
    }
};

function handleClick(e) {
    var evt = e || window.event;
    var target;
    if (evt.target) {
        target = evt.target;
    } else {
        target = evt.srcElement;
    }
    alert("You clicked on " + target.id);
}

```

IE8支持用于指定加载事件处理程序的属性onload，因此这行代码不会有任何问题。

如果你使用方法addEventListener来添加事件处理程序，就必须检查浏览器是否支持它。

如果浏览器不支持它，就应转而使用方法attachEvent。请注意，方法attachEvent不接受第3个参数，另外，它要求使用“onclick”来表示单击事件。

如果向事件处理程序传递了事件对象，就说明用户使用的浏览器为IE9以上版本或其他浏览器。否则，你必须从window中获取该事件对象。

如果事件对象是向事件处理程序传递的，其属性target将为触发事件的元素。如果用户使用的是IE8或更早的版本，包含该元素的属性将为srcElement。

#7. 正则表达式

本书前面提到过RegExp对象。RegExp表示**正则表达式**，是一种描述文本模式的语法。例如，通过使用正则表达式，可编写匹配以下文本的表达式：以t打头、以e结束，且至少包含一个a和至多两个u。

正则表达式可能很复杂。事实上，刚见到正则表达式时，它们可能看起来就像天书。不过，你很快就会熟悉简单的正则表达式。如果你对正则表达式感兴趣，请参阅这方面的优秀资料。

构造函数RegExp

下面来介绍两个正则表达式。要创建正则表达式，可调用构造函数RegExp，并传入放在两个斜杠之间的搜索模式，如下所示：

```
var areaCode = new RegExp(/[0-9]{3}/);
```

```
var phoneNumber = new RegExp(/^d{3}-?d{4}$/);
```

还记得第7章的座椅争夺战吗？Amy就是凭借这个正则表达式获胜的。

构造函数RegExp的实参为搜索模式。如何解读这两个搜索模式呢？

像这样解读……

要理解正则表达式，关键在于学会如何解读搜索模式。搜索模式是正则表达式中最复杂的部分，下面将详细解读这两个示例，其他的就留给你自己去探索吧。

这部分的意思是与0~9的任何数字都匹配。[]用于指定与什么范围内的字母或数字匹配。

这部分的意思是与三个字符匹配。换言之，这个搜索模式与三个0~9的数字匹配。

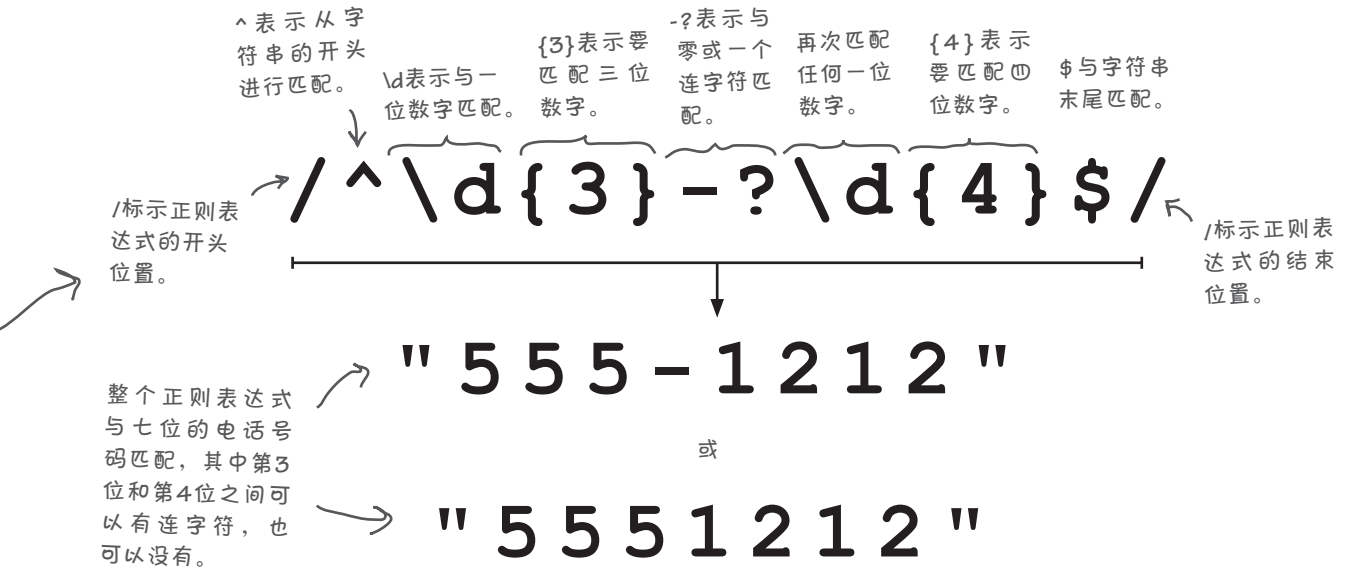
/标示正则表达式的开头位置。

/ [0 - 9] { 3 } /

/标示正则表达式的结束位置。

这个正则表达式与由三个数字组成的字符串（如“201”或“503”）匹配。

" 2 0 1 "



使用RegExp对象

要使用正则表达式，首先必须有一个要搜索的字符串：

```
var amyHome = "555-1212";
```

接下来，判断这个字符串是否与正则表达式匹配。为此，可对这个字符串调用方法match，并将正则表达式对象作为实参传递给它：

```
var result = amyHome.match(phoneNumber);
```

result的值为[“555-1212”]，因为指定的正则表达式与变量amyHome存储的整个字符串匹配。

结果为一个数组，其中包含与正则表达式匹配的子串。如果结果为null，就说明字符串的任何部分都不与正则表达式匹配。

```
var invalid = "5556-1212";
var result2 = invalid.match(phoneNumber);
```

result2的值为null，因为变量invalid存储的字符串的任何部分都不与指定的正则表达式搜索模式匹配。

创建正则表达式后，就可以不断地使用它来匹配字符串，想匹配多少个就能匹配多少个。

#8. 递归

给函数指定名称后，就可以做一件有趣的事情：在这个函数中调用它自己。我们称之为递归或递归函数调用。

为何需要这样做呢？因为有些问题就是递归性的。下面是一个来自数学领域的例子：计算斐波那契数列的算法。斐波那契数列是这样的：

0、1、1、2、3、5、8、13、21、34、55、89、144……

为计算斐波那契数列，首先做如下两个假设：

$\text{Fibonacci}(0) = 1$

$\text{Fibonacci}(1) = 1$

然后，对于斐波那契数列中的任何数字，都通过将前两个数字相加来得到它：

$\text{Fibonacci}(2) = \text{Fibonacci}(1) + \text{Fibonacci}(0) = 2$

$\text{Fibonacci}(3) = \text{Fibonacci}(2) + \text{Fibonacci}(1) = 3$

$\text{Fibonacci}(4) = \text{Fibonacci}(3) + \text{Fibonacci}(2) = 5$

依此类推。计算斐波那契数的算法天然具有递归性，因为你将前两个斐波那契数相加来得到下一个斐波那契数。

我们可创建一个这样的递归函数来计算斐波那契数：要计算 $\text{Fibonacci}(n)$ ，我们两次调用函数`fibonacci`，并分别将 $n-1$ 和 $n-2$ 作为实参，再将这两次调用函数`fibonacci`的结果相加。

下面编写这个函数的代码。首先，处理实参为0或1的情形：

```
function fibonacci(n) {
    if (n === 0) return 1;
    if (n === 1) return 1;
}
```

编写一个函数，它将 n 作为参数。该参数表示要计算斐波那契数列中的第几个数字。

我们知道，如果 n 为0或1，我们都应返回1。这被称为函数的基线条件（base case），因为满足这种条件后，就无需再递归调用了。

这些被称为基线条件，即无需根据以前的斐波那契数来计算它们；通常最好先将实现它们的代码编写出来。接下来，你就可以这样想：要计算 $\text{Fibonacci}(n)$ ，只需将 $\text{Fibonacci}(n-1)$ 和 $\text{Fibonacci}(n-2)$ 相加，并返回结果即可。

下面来编写这样的代码：

```
function fibonacci(n) {
  if (n === 0) return 1;
  if (n === 1) return 1;
  return (fibonacci(n-1) + fibonacci(n-2));
}
```

如果n不为0或1，只需将Fibonacci(n-1)和Fibonacci(n-2)相加，并返回结果即可。

如果你从未见过递归，这些代码看起来有点不可思议，但它们确实能够计算出斐波那契数。下面整理了这些代码，并对其进行测试：

```
function fibonacci(n) {
  if (n === 0 || n === 1) {
    return 1;
  } else {
    return (fibonacci(n-1) + fibonacci(n-2));
  }
}
```

与前面的代码等效，只是更简洁。

```
for (var i = 0; i < 10; i++) {
  console.log("The fibonacci of " + i + " is " + fibonacci(i));
}
```

这是一些测试代码。



一定要有基线条件。

如果递归代码满足不了导致计算结束的基线条件，它将不断地执行下去，就像无限循环。换句话说，函数将不断地调用自己，进而不断地消耗资源，直到浏览器难以应付。因此，如果包含递归代码的网页没有响应，请核实这些代码最终能满足基线条件。

JavaScript控制台

```
The fibonacci of 0 is 1
The fibonacci of 1 is 1
The fibonacci of 2 is 2
The fibonacci of 3 is 3
The fibonacci of 4 is 5
The fibonacci of 5 is 8
The fibonacci of 6 is 13
The fibonacci of 7 is 21
The fibonacci of 8 is 34
The fibonacci of 9 is 55
```

#9. JSON

JavaScript不仅是一种Web编程语言，还正在逐渐成为一种常用的对象存储和传输格式。JSON是一个缩略语，表示JavaScript Object Notation。这种格式让你能够以字符串的方式表示JavaScript对象，以便对其进行存储和传输：

一个JSON字符串。

```
var fidoString = '{ "name": "Fido", "breed": "Mixed", "weight": 38 }';
```

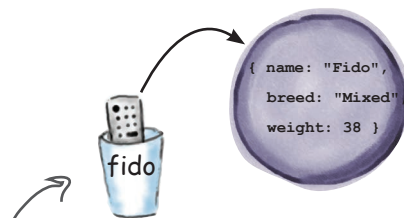
请注意，这里使用了单引号来括起这个JSON字符串。我们必须这样做，因为这个字符串包含双引号。如果使用双引号来括起它，将让JavaScript感到迷惑。单引号让JavaScript知道这是一个很长的字符串，其中包含其他字符串。

是不是觉得眼熟？本应如此。这个字符串很像本书前面创建的对象fido。

JSON的优点之一是，它让我们能够将类似于上面的字符串转换为对象。为此，需要使用JavaScript对象JSON提供的两个方法：JSON.parse和JSON.stringify。下面使用方法parse来分析前面的fidoString，并将其转换为真正的小狗对象（总之是JavaScript对象）：

```
var fido = JSON.parse(fidoString);
```

调用对象JSON的方法parse，并将前面的字符串传递给它。结果是一个JavaScript对象。



将指向这个对象的引用存储到变量fido中。

请注意，这里用到了JSON对象。JSON既是一种字符串格式的名称，也是一种JavaScript对象。

你还可执行相反的操作。如果你有一个对象fido，并想将其转换为字符串，只需调用方法JSON.stringify即可，如下所示：

```
var fido = {
  name: "Fido",
  breed: "Mixed",
  weight: 38
};
var fidoString = JSON.stringify(fido);
```

这里创建了一个JavaScript对象……

并将其转换为字符串。

请注意，JSON格式不支持方法（因此不能在JSON字符串中包含方法bark），但它支持所有的基本类型，还有对象和数组。

#10. 服务器端JavaScript

本书只介绍了浏览器和客户端编程，但你也可将学到的JavaScript技能用于另一个完全不同的领域——服务器端编程。对你在Internet上使用的Web和云服务来说，服务器端编程通常必不可少。如果你要新建一个Web镇玉米卷在线订购系统，或者认为下一个重大创意是反社会网络，就需要编写在云端（Internet服务器上）运行的代码。

Node.js是当前流行的一种JavaScript服务器端技术，它包含自己的运行环境和库（就像客户端JavaScript使用浏览器提供的库一样）。与浏览器一样，Node.js运行JavaScript时使用的也是单线程模型。在这种模型中，每次只有一个执行线程。因此，其编程模型与基于异步事件和事件循环的浏览器类似。

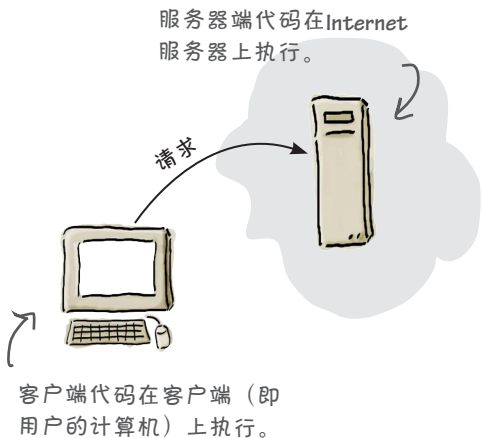
例如，下面的方法启动Web服务器，以侦听到来的Web请求。它将一个负责处理请求的处理程序作为参数。注意，要指定处理请求的事件处理程序，可向方法createServer传递一个匿名函数。

Node.js库中的方法http.createServer
接受一个实参，这个实参是一个以
匿名函数的方式指定的处理程序。

```
http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

这个匿名函数负责处理请求。
它在有请求到来时作出响应，
将字符串“Hello World”发送
给客户端。

当然，要理解Node.js的工作原理还要学习很多东西，不过你已经掌握了很多有关对象和函数的知识，这些学起来都不难。要详细了解Node.js，需要阅读相关的专著，除此之外，<http://nodejs.org>也提供了很多教程、文章和演示。



索引

符号

\$ (美元符号)

jQuery中的函数, 624

用于JavaScript变量名开头, 13

0 (零), 被视为假值, 292

&& (与) 运算符, 55, 62–63, 74

* (星号) 运算符, 为乘法算术运算符, 15, 286

: (冒号), 分隔属性名和属性值, 179

, (逗号), 分隔对象的不同属性, 177, 179

{ } (花括号)

定义代码块, 17

定义对象字面量, 177, 522

定义函数体, 97

将对象的属性括起, 177, 179

在代码中必须配套, 59

. (句点表示法)

访问对象的属性, 181, 209, 230

与this对象一起使用, 202

与引用变量一起使用, 186

= (等号) 运算符, 用于给变量赋值, 11, 16, 275

== (相等) 运算符

一种比较运算符, 16, 55, 275–285, 311, 459

与===的异同, 289

=== (严格相等) 运算符

一种比较运算符, 55, 280–285, 311

与==的异同, 289

// (双斜杠), 在JavaScript中用于注释开头, 13

/ (斜杠) 运算符, 用作除法算术运算符, 15, 286

> (大于) 运算符, 16, 55, 459

>= (大于或等于) 运算符, 16, 55

< (小于) 运算符, 55, 459

<= (小于或等于) 运算符, 55

- (减号或符号) 运算符

用作处理字符串和数字的算术运算符, 286–287, 312

用作单目运算符, 287

!= (不等) 运算符, 16, 55

! (非) 运算符, 55

|| (或) 运算符, 54, 55, 62–63, 74

() (圆括号)

调用函数68, 430, 439

指定参数 97

+(加号)

算术运算符, 286–287, 312

用于拼接字符串, 15, 133, 142, 354

-- (后递减运算符), 146–147

++ (后递增运算符), 146–147

“ ” (双引号)

用于将属性名括起, 179

在JavaScript用于将字符串括起, 13

;(分号), 在JavaScript中结束语句, 11, 13

[] (方括号)

用于定义和访问数组, 127, 129, 550

用于访问属性, 209

!== (严格不等) 运算符, 281

_ (下划线), 用于JavaScript变量名开头, 13

A

action特性, 328

addEventListener方法, 630

alert函数

用于与用户交流, 25–26, 42, 46

在简单的战舰游戏中指出是否击中了目标, 59–60, 76

alt特性, 256

Array对象, 549–551

按引用传递, 192

按值传递, 92–93, 192

B

<body>元素

简介, 4

使用属性innerHTML替换元素的内容, 245

- 添加代码, 32
- BOM (浏览器对象模型), 627
- 保留字, 12
- 背景图像
 - 创建HTML页面, 320–321
 - 单元格, 322
- 闭包
 - 创建练习, 500, 514–515
 - 简介, 493, 495–497
 - 将函数表达式作为实参, 501
 - 实际环境, 502
 - 实现神奇的计数器, 498–499
 - 使用事件处理程序来创建, 503–507
- 比较运算符, 55, 71
- 编程语言, 学习, 9
- 变量
 - undefined值和变量 268
 - 变量的词法作用域, 488–490, 494
 - 变量的默认值, 50
 - 变量中的对象, 186
 - 查找与变量相关的错误, 39
 - 代码卫生指南, 111
 - 短暂的生命周期, 102
 - 简介, 11
 - 将对象赋给变量, 192
 - 将函数赋给变量, 439, 449, 469
 - 命名, 12–13, 100, 103, 108
 - 局部变量
 - 代码卫生指南, 111
 - 方法中的局部变量, 206
 - 局部变量的词法作用域, 488–490, 494
 - 嵌套的影响, 486, 488–490, 494
 - 声明, 103, 108
 - 识别, 105, 120
 - 与全局变量比较, 99, 106–107
 - 作用域, 101
 - 全局变量
 - 代码卫生指南, 111
 - 滥用, 391
 - 嵌套的影响, 486, 488–490, 494
 - 全局变量的词法作用域, 488–490, 494
 - 识别, 105, 120
 - 与局部变量比较, 99, 106–107
 - 作用域, 101
 - 设置为null, 270
 - 声明局部变量, 103, 108
 - 声明中的关键字var, 97, 99
 - 识别, 91, 119
 - 使用伪代码来确定所需的变量, 50
 - 数组中的变量, 127, 129
 - 网页重新加载和变量, 108
 - 用作条件, 23
 - 在for语句中声明, 152
 - 在函数内和函数外声明, 98
 - 遮住, 104
 - 指向函数的变量, 436
 - 自由变量, 495–496, 501, 516
 - 作为实参传递给函数, 89
 - 作用域, 101
 - 变量存储的内容, 186–187
 - 变量的词法作用域, 488, 494
 - 变量的生命周期, 102
 - 变量的作用域, 101
 - 编写JavaScript代码。另请参见“货真价实的战舰游戏”“简单的战舰游戏”和“代码”
 - else if语句, 用于作决策, 22–23
 - null, 256, 270–271, 274, 278
 - prompt函数, 46, 53
 - while语句, 用于执行代码, 18–21
 - 帮助设计程序的流程图, 45
 - 变量的默认值undefined, 50
 - 表达式, 15–16, 40
 - 查找代码错误
 - 练习, 14, 39, 203, 224
 - 使用控制台, 68
 - 重构代码, 156, 159
 - 重用代码, 71, 79, xii
 - 串接, 345, 348
 - 代码卫生指南, 111
 - 对数字和字符串进行比较, 275–277, 281
 - 对象中的重复代码, 208
 - 分析代码, 80–81, 116
 - 函数和代码重用, 83–88
 - 简介, 3, 7–8
 - 生成随机数, 67–68
 - 实施质量保证, 61, 69–70
 - 使用=运算符给变量赋值, 11, 16, 275
 - 使用==运算符
 - 与===运算符比较, 289
 - 作为比较运算符, 16, 55, 275–285, 311, 459
 - 使用===运算符
 - 使用变量, 11–13
 - 与==运算符比较, 289
 - 作为比较运算符, 55, 280–285, 311

- 伪代码
 - 编写, 47–48, 73
 - 确定所需的变量, 50
 - 实现循环, 51
 - 在简单的战舰游戏中判断是否击中, 59–60, 76
 - 转换为JavaScript代码, 52, 164–165, 171
- 语句, 10
- 值, 272–274, 281
- 指定条件时别太啰嗦, 65, 74
- 编写函数, 25–26, 42
- 编写清晰的代码, 12
- 编译代码, 5
- 表达式, 15–16, 40
- 播放列表
 - jQuery中, 625
 - 使用document对象, 626
 - 使用JavaScript数组在播放列表中填充歌曲名, 253, 262
- 布尔表达式
 - true和false, 13, 15–16, 291–293, 313
 - 使用布尔表达式来作决策, 22
- 布尔运算符
 - 比较运算符和逻辑运算符, 71
 - 编写复杂的语句, 62–64, 74–75
 - 指定条件时别太啰嗦, 65, 74
 - 指南, 55
- 布尔值
 - 简介, 11, 23
 - 使用==运算符将其与其他类型进行比较, 277–278, 281
 - 作为对象, 296
 - 作为基本类型, 266
- 不等 (!=) 运算符, 16, 55
- 不要自我重复 (DRY), 603
- C**
- call方法, 603–604
- charAt方法, 297
- cliche方法, 610
- close()方法, 627
- collision方法, 364, 368
- concat方法, 300
- confirm方法, 627
- console.log函数
 - 与用户交流, 25–27, 42
 - 在控制台中显示对象, 185, 608
- controller对象, 货真价实的战舰游戏
 - 处理玩家的猜测, 350–354
 - 记录猜测并开火的次数, 355–356
 - 简介, 329
 - 开发函数parseGuess, 351–354
 - 判断游戏是否结束, 356
 - 试驾, 357
 - 实现, 349–357
 - 提供输入, 360
- CSS (级联样式表)
 - 编写, 254, 263
 - 标记文本, 10
 - 创建交互式网页, 319
 - 定位, 324–326, 328
 - 简介, 2
 - 属性background, 326
- CSS属性background, 326
- 猜图游戏
 - 创建, 388–392, 411–414, 422, 426
 - 添加图像
 - 简介, 393–397
 - 将事件处理程序赋给属性onclick, 390–391, 393–398
 - 指定单击事件处理程序, 396–397
- 藏宝图游戏, 405–406, 424
- 重构代码, 156, 159
- 处理事件
 - 事件处理程序
 - onload, 249
 - timerHandler, 407
 - 创建, 385–386
 - 赋给属性, 407
 - 回调函数和事件处理程序, 250
 - 简介, 383
 - 使用addEventListener来添加, 630
 - 使用函数setInterval, 410, 425
 - 使用函数setTimeout, 407–413
 - 用来创建闭包 503–507
 - 在货真价实的战舰游戏中的应用, 358–359, 361
 - 种类, 252
 - 事件对象, 399–402, 423
 - 事件通知练习, 382, 383, 421
 - 使用函数setInterval, 410, 425
 - 使用函数setTimeout, 407–413
 - 响应事件
 - 创建猜图游戏, 388–392, 411–414, 422, 426

- 简介, 387
- 将事件处理程序赋给属性onclick, 390–391, 393–398
- 事件响应练习, 415, 427
- 在猜图游戏中添加图像, 393–397
- 指定单击事件处理程序, 396–397
- 异步编码
 - onload, 249
 - timerHandler, 407
 - 创建, 385–386
 - 赋给属性, 407
 - 简介, 383
 - 使用函数setInterval, 410, 425
 - 使用函数setTimeout, 407–413
 - 用来创建闭包, 503–507
 - 在货真价实的战舰游戏中的应用, 358–359, 361
 - 种类, 253
 - 与浏览器讨论事件, 403
- 传递实参, 88–89
- 串接, 345, 348
- 传统面向对象编程, 564
- 错误对象, 551, 629

D

- Date对象, 214, 548, 551
- delete关键字, 184
- displayHit方法, 331–334
- displayMessage方法, 331–334
- displayMiss方法, 331–334
- DOM (文档对象模型)
 - DOM为树形结构, 235
 - NodeList, 397
 - 创建, 234–235
 - 创建、添加或删除元素, 258
 - 访问图像, 389–390
 - 分析HTML并创建DOM, 236, 261
 - 根据id获取元素, 245
 - 更新, 247–248
 - 获取元素, 241
 - 简介, xv
 - 设置特性
 - 使用代码, 254
 - 使用方法setAttribute, 255, 333
 - 事件, 413
 - 事件对象, 399–402, 423
 - 使用JavaScript与之交互, 233
 - 使用jQuery获取元素, 624
 - 文档对象, 235, 626
 - 修改, 244
 - 隐藏的密电, 246
 - 用于与用户交流, 25–26, 42
- do while循环, 364, 373
- dragstart事件, 419
- drop事件, 419
- DRY (不要自我重复), 603
- 大于 (>) 运算符, 16, 55, 459
- 大于或等于 (>=) 运算符, 16, 55
- 代码。另请参见“货真价实的战舰游戏”“简单的战舰游戏”和“编写JavaScript代码”
 - JavaScript应用程序, 29–35
 - 编译, 5
 - 查找错误
 - 练习, 14, 39, 203, 224
 - 使用函数console.log, 25, 27
 - 使用控制台, 68
 - 重构, 156, 159
 - 重用, 71, 79, xii
 - 递归, 635
 - 对象中重复的代码, 208
 - 分析, 80, 116
 - 函数和重用, 83–88
 - 加载到浏览器中, 31
 - 将JavaScript从HTML页面移到文件中, 33–34
 - 浏览器执行, 433
 - 卫生指南, 111
- 代码块, 17, 23
- 代码文件
 - 在HTML页面的<body>元素中链接到外部文件, 32
 - 在HTML页面的<head>元素中链接, 32
- 代码中的错误, 查找
 - 处理异常, 629
 - 练习, 14, 39, 203, 224
 - 使用函数console.log, 25, 27
 - 使用控制台查找错误, 68
- 单击事件, 407, 419, 624
- 等号 (=) 运算符, 用于给变量赋值, 11, 16, 275
- 递归, 634–635
- 调用函数
 - 传递实参, 90, 430
 - 递归函数, 634–635

- 简介, 85–86, 492–493
- 练习, 117
- 浏览器如何处理函数声明和函数表达式, 431–437
- 圆括号 (), 68, 430, 439
- 指向函数的变量, 436
- 定时器事件, 407–409
- 定位, CSS, 324–326, 328
- 定义包含形参的函数, 90
- 逗号 (,), 分隔对象的不同属性, 177, 179
- 队列和事件, 404
- 对象
 - arguments对象, 628
 - Array对象, 549–551
 - Array构造函数, 549–551
 - controller对象 (货真价实的战舰游戏)
 - 处理玩家的猜测, 350–354
 - 记录猜测并开火的次数, 355–356
 - 简介, 329
 - 开发函数parseGuess, 351–354
 - 判断游戏是否结束, 356
 - 试驾, 357
 - 实现, 349–357
 - 提供输入, 360
 - document对象
 - DOM中的document对象, 235, 626
 - 方法getElementById, 238。另请参见“getElementById”
 - 简介, 214, 240
 - 使用document编写函数与用户交流, 25–26, 42
 - JSON, 214
 - Math, 214, 551
 - model (货真价实的战舰游戏)。另请参见“model对象, 货真价实的战舰游戏”
 - RegExp, 214, 551, 632–633
 - this
 - 用法, 202
 - 用于构造函数中, 528–529, 532
 - 与对象结合起来使用, 204–206, 219
 - 原型和this, 580
 - window
 - history方法, 627
 - innerHeight属性, 627
 - innerWidth属性, 627
 - location方法, 627
 - onload属性。参见“onload属性”
 - onresize属性, 415, 427
 - print方法, 627
 - setInterval方法, 627
 - setTimeout方法, 407–413, 627
 - 包含的方法和属性, 627
 - 创建onload事件处理程序, 249, 402
 - 对BOM的支持, 627
 - 简介, 214
- 按约定在对象中添加方法, 523
- 比较对象, 459
- 比较方法和函数, 206
- 遍历对象的属性, 209
- 创建对象, 177–179, 183, 185, 188–191, 220–222
- 错误对象, 551
- 代码错误查找练习, 203
- 对象变量, 186
- 对象变量声明, 177
- 对象创建规则, 179
- 对象的属性
 - 遍历属性, 209
 - 访问, 181
 - 简介, 175
 - 使用属性进行计算, 182
 - 属性的值, 176, 219
 - 随时添加或删除, 184
 - 添加新属性, 182
 - 未定义的值和属性, 268
 - 修改, 182
- 对象相等, 288–290, 314
- 对象中重复的代码, 208
- 方法, 198
- 回调函数, 250
- 绘制对象图, 565, 570
- 继承对象, 569–572
- 建立模型, 3
- 简介, 173–174, 215, xiv–xv
- 控制台, 214
- 理解对象实例, 543–545, 560
- 没有愚蠢的问题, 185
- 内置对象, 548–551, 608
- 汽车自动制造机应用程序 (示例), 195–197
- 日期对象, 214, 548
- 事件对象
 - 事件处理程序和事件对象, 399–402, 423
 - 属性, 401
 - 在IE8和更早的版本中设置事件对象, 399, 631
- 使用null, 270–271, 274
- 使用this, 202, 204–206, 219
- 使用对象改善生活, 612–613
- 使用对象字面量创建对象, 522, 532, 539–542, 559
- 数组嵌套层级数, 348

- 提供的JavaScript, 214
 - 添加行为, 198–201
 - 未定义的值和对象, 268
 - 向函数传递对象, 192, 192–194, 198
 - 行为, 568
 - 与基本类型比较, 187
 - 元素, 231
 - 状态和行为, 210–212, 226
 - 作为对象的布尔值, 296
 - 作为对象的函数, 612–613
 - 作为对象的数字, 296
 - 作为基本类型和对象的字符串, 294–296
 - 对象arguments, 628
 - 对象变量的声明, 177
 - 对象创建规则, 179
 - 对象的行为和状态, 210–212, 226
 - 对象构造函数
 - 创建, 525–526, 536–538, 555, 558, 566–567
 - 创建对象
 - 按约定, 523
 - 使用对象字面量, 522
 - 代码错误查找练习, 529, 556
 - 定义方法, 530–532, 557
 - 独特的属性, 546–547
 - 更新构造函数
 - 创建继承其他原型的原型, 594–599, 620
 - 清洁代码, 600–605
 - 设计, 588–590
 - 原型链, 591–593, 606, 619
 - 工作原理, 528
 - 简介, 521, 525, xxi
 - 理解对象实例, 543–545, 560
 - 命名, 532
 - 内置构造函数, 548
 - 使用, 527, 555
 - 同关键字new一起使用, 528, 532–535, 543, 548
 - 形参名, 532
 - 与对象字面量比较, 532, 539–542, 559
 - 对象构造函数Array, 549–551
 - 对象实例, 543–545, 560
 - 对象图, 565, 570
 - 对象引用, 187, 192
 - 对象原型模型
 - Object
 - 简介, 607
 - 作为原型, 607
 - 改变内置行为, 608–609
 - 重复的方法, 568
 - 对象图练习, 574, 615
 - 更新构造函数
 - 创建继承其他原型的原型, 594–599, 620
 - 清洁代码, 600–605
 - 设计, 588–590
 - 原型链, 591–593, 606, 619
 - 继承
 - 对象中的继承, 569–572
 - 改变内置行为, 608–609
 - 扩展内置的对象, 610
 - 一个原型继承另一个原型, 594–599, 606, 620
 - 原型链中的继承, 592–593, 619
 - 简介, xxii–xxiii
 - 判断属性包含在实例还是原型中, 586–587, 592, 618
 - 使用对象改善生活, 612–613
 - 使用方法扩展原型String, 610–611, 621
 - 实现代码, 579, 583, 616–617
 - 原型
 - 重写, 573, 578
 - 定义方法, 581–582, 584
 - 动态的, 582
 - 获取, 575
 - 简介, 569
 - 设置, 575
 - 修改属性, 582, 585
 - 原型链
 - 建立, 591
 - 原型链中的继承, 592–593, 619
 - 多个单词组成的变量名的骆驼式拼写法, 13
- ## E
- ECMAScript 5, 6
 - else if语句
 - 编写, 56
 - 作决策, 22–23
 - else语句, 23
- ## F
- false (布尔值), 13
 - Flash, 用于创建动态网页, 5
 - for/in迭代器, 209
 - for循环

- 重新编写, 47
- 用于迭代数组, 140–142, 169
- 与while循环比较, 144
- 在货真价实的战舰游戏中猜测战舰的位置, 343
- 在货真价实的战舰游戏中添加新的战舰位置, 365, 367

<form>元素

- Action特性, 328
- 添加<input>元素, 323
- 使用CSS来定位, 325

fire方法, 货真价实的战舰游戏, 342, 346

方法

- 编写, 199–200, 207, 225
- 重复, 568
- 简介, 198
- 可用的JavaScript方法, 296
- 扩展原型String, 610–611, 621
- 设计货真价实的战舰游戏, 330–331, 376
- 与函数比较, 206
- 原型链, 592–593, 619
- 在方法中使用this对象, 202, 204–206, 219
- 在构造函数中定义, 530–532, 557
- 在原型中添加, 581–582, 584
- 字符串方法简明教程, 297–300

方括号 ([])

- 用于定义数组, 127, 129, 550
- 用于访问属性, 209

非 (!) 运算符, 55

分号 (;), 在JavaScript中用于结束语句, 11, 13

分析函数的声明, 432

封装, 200

服务器端JavaScript, 637

G

generateShip方法, 364–365, 367

generateShipLocations方法, 364, 367, 369

getAttribute方法, 256, 391

getDay方法, 548

getElementById方法

- 查找元素并修改其内容, 238–239
- 传入不存在的id, 245
- 返回null, 256, 270–271
- 访问图像, 388–392
- 获取元素, 231, 240, 247–248

区分大小写, 230

- 在货真价实的战舰游戏中获取执行按钮fire的引用, 359
- 作为文档对象, 237

getElementsByClassName方法, 245, 626

getElementsByName方法, 626

getElementsByTagName方法, 245, 396, 397

getFullYear方法, 548

getSize方法, 531, 557

guesses属性, 货真价实的战舰游戏, 349

跟踪执行流程(跟踪), 96

关键字, 12

关键字new, 与构造函数结合使用, 528, 532–535, 543, 548

过程型设计, 329

H

handleFireButton函数, 359

handleKeyPress函数, 361

hasOwnProperty方法, 586, 607

<head>元素

- 简介, 4
- 链接到代码文件, 32
- 添加代码, 32

《Head First HTML5 Programming中文版》, 413, 613, 626

《Head First HTML与CSS》, 320, 328

history方法, 627

HTML

- 编辑器, 31
- 标记文本, 10
- 创建交互式网页
 - 背景, 320–321
 - 表格, 322
 - 简介, 319–320
 - 使用hit和miss类, 327, 375
 - 添加CSS, 324–326
 - 与用户交互, 322
 - 指出是否击中了战舰, 326
- 分析HTML并创建DOM, 236, 261
- 简介, 2
- 使用id标识元素, 243
- 在简单的战舰游戏中链接代码, 49

HTML5

- 标准文档类型, 4

- 将id特性设置为数字字符串, 328
- 结合使用JavaScript, 5
- HTML编辑器Coda, 31
- HTML编辑器Dreamweaver, 31
- HTML页面
 - 加载代码, 32
 - 将代码放在外部文件中, 230–232
 - 使用JavaScript与页面交互, 233
- <html>元素, 4
- 函数
 - alert
 - 用于与用户交流, 25–26, 42, 46
 - 在简单的战舰游戏中判断是否击中, 59, 76
 - arguments对象, 628
 - console.log
 - 用于与用户交流, 25, 42
 - 用于在控制台中显示对象, 185
 - init, 249, 251, 359, 361, 369, 389–391
 - isNaN, 273, 353
 - jQuery中的\$ (美元符号), 624
 - Math.floor, 68, 131, 365–367
 - Math.random, 67–68, 88, 131, 365–367
 - onload事件处理程序, 249–250, 359, 389–390
 - prompt, 46, 53, 55
 - return语句, 95–97
 - setInterval, 410, 413, 425, 627
 - setTimeout, 407–413, 410, 480, 501, 627
 - timerHandler, 407
 - 闭包
 - 闭包中的实际环境, 502
 - 创建练习, 500, 514–515
 - 简介, 493, 495–497
 - 使用事件处理程序来创建闭包, 503–507
 - 通过将函数表达式作为实参来创建, 501
 - 用于实现计数器, 498–499
 - 比较形参和实参, 90
 - 变量的生命周期, 102
 - 编写函数, 25–26, 42
 - 标准声明方式, 430
 - 创建, 83–87
 - 从函数中返回函数, 450–456, 470, 472
 - 错误地调用函数, 94
 - 代码卫生指南, 111
 - 调用函数, 85–86, 117
 - 递归地调用函数, 634–635
 - 定义, 132
 - 定义顺序, 389
 - 赋给变量, 439, 449, 469
 - 跟踪函数的执行过程, 96
 - 关键字, 430
 - 函数表达式
 - 赋给变量, 439
 - 浏览器如何处理, 434–436, 465
 - 替代, 481
 - 与函数声明比较, 431, 437
 - 执行代码, 438, 466
 - 函数调用中的圆括号 (), 68, 430, 439
 - 函数剖析, 97
 - 函数声明
 - 分析, 431
 - 赋给变量, 439
 - 简介, 430
 - 浏览器如何处理, 431–433, 436, 465, 483
 - 匿名函数中的函数声明, 483–485
 - 与函数表达式比较, 431, 437
 - 执行代码, 438, 466
 - 函数体
 - 其中的形参, 84–85
 - 在其中声明变量, 97
 - 函数形参
 - 传递实参, 89
 - 简介, 84, 88
 - 将实参的值赋给形参, 85
 - 命名, 97
 - 识别, 91, 105, 119, 120
 - 未在函数中使用, 94
 - 与实参比较, 90
 - 函数中的变量
 - 比较局部变量和全局变量, 99, 106–107
 - 变量的词法作用域, 488–490, 494
 - 变量的默认值, 50
 - 重新加载网页, 108
 - 关键字var, 97, 99
 - 局部和全局变量的作用域, 101
 - 命名, 12–13, 100, 103, 108
 - 嵌套, 486, 488–490, 494
 - 声明局部变量, 103, 108
 - 在函数内和函数外声明, 97–99
 - 遮住, 103
 - 作为实参进行传递, 89
 - 简介, 79, 88, xii
 - 将JavaScript放在什么地方, 109
 - 将处理数组的代码组织为函数, 157–162
 - 将对象传递给函数, 192–194, 198
 - 将函数传递给函数, 409, 443–448, 468, 482, 486, 514
 - 命名, 97

- 内置函数, 71, 91, 119
- 匿名函数
 - 创建, 477–478, 512–513
 - 简介, 475–476, 482, xx–xxi
 - 可读性, 480
 - 脑筋急转弯, 509, 518
 - 让代码更紧凑, 479
 - 向函数传递函数, 482, 486, 514
 - 在构造函数中将函数赋给方法属性, 530–532, 557
- 没有愚蠢的问题, 97, 108
- 嵌套, 485–486, 488–490, 494
- 识别练习, 91, 119
- 实参
 - 按值传递, 92–93
 - 传递, 88–89, 92–94
 - 简介, 85
 - 识别, 91, 105
 - 顺序不正确, 97
 - 与形参的异同, 90
- 严峻的JavaScript挑战, 487, 508, 517
- 引用
 - 传递引用, 479
 - 给引用赋值, 477
 - 函数调用中的引用, 491
 - 简介, 430, 476
 - 替代函数表达式, 481
 - 用于将函数作为实参传递给另一个函数, 486
 - 在函数中使用引用, 491, 494
- 与方法比较, 206
- 作用域
 - 变量的词法作用域, 488–490, 494
 - 局部变量和全局变量的作用域, 101
- 作为对象, 612–613
- 作为一等公民
 - 从函数返回函数, 450–456, 470, 472
 - 简介, 442–443
 - 使用数组的方法sort, 457–463, 472–473
 - 替换函数表达式, 481
 - 向函数传递函数, 443–448, 468, 482, 486, 514
 - 严峻的JavaScript挑战, 486, 508, 517
- 作为值, 439–440, 467
- 函数表达式
 - 赋给变量, 439
 - 浏览器如何处理, 434–436, 465
 - 替代, 481
 - 与函数声明比较, 431, 437
 - 执行代码, 438, 466
- 函数嵌套, 485–486, 488–490, 494
- 函数声明
 - 分析, 431
 - 赋给变量, 439
 - 浏览器如何处理, 431–433, 436, 465, 483
 - 匿名函数, 483–485
 - 与函数表达式比较, 431, 437
 - 执行代码, 438, 466
- 函数实参
 - 按值传递, 92–93
 - 传递, 88–89, 92–94
 - 简介, 85
 - 识别, 91, 105, 119, 120
 - 顺序不正确, 97
 - 为对象时, 192
 - 与形参的异同, 90
- 函数体
 - 声明变量, 97
 - 形参, 84–85
- 函数引用
 - 传递, 479
 - 赋值, 477
 - 简介, 430, 476
 - 替代函数表达式, 481
 - 用于将函数作为实参传递给其他函数, 486
 - 在函数调用中的应用, 491
 - 在函数中的应用, 491, 494
- 后递减运算符 (--), 146–147
- 后递增运算符 (++), 146–147
- 花括号 ({ })
 - 定义代码块, 17
 - 定义对象字面量, 177, 522
 - 定义函数体, 97
 - 将对象的属性括起, 177, 179
 - 在代码中必须配套, 59
- 回调函数。另请参见“事件处理程序”, 250
- 或 (||) 运算符, 54, 55, 62–63, 74
- 货真价实的战舰游戏
 - QA, 370–371
 - 按钮fire的事件处理程序, 359
 - 创建HTML页面
 - 表格, 322
 - 背景, 320–321
 - 简介, 320
 - 使用hit和miss类, 327, 375
 - 添加CSS, 324–326
 - 指出是否击中战舰, 326

- 对象controller
 - 处理玩家的猜测, 350–354
 - 记录猜测并开火的次数, 355–356
 - 简介, 329
 - 将输入交给控制器, 360
 - 开发函数parseGuess, 351–354
 - 判断游戏是否结束, 356
 - 试驾, 357
 - 实现, 349–357
- 对象model
 - 方法fire, 342
 - 告诉视图是否击中了战舰, 347
 - 获取游戏板尺寸的函数parseGuess, 353
 - 简介, 329, 336
 - 试驾, 348
 - 实现, 341–348
 - 判断是否击中, 344–345
 - 判断是否击中了战舰, 343
 - 判断战舰是否被击沉, 346
 - 与视图交互, 336
 - 战舰的表示, 338–340, 378–379
 - 战舰状态的表示, 337, 377
- 对象view
 - 简介, 329
 - 试驾, 335
 - 实现, 329, 331–334
 - 与模型交互, 336
 - 指出是否击中了, 347
- 跟踪战舰, 337
- 工具包, 319
- 获取玩家的猜测, 358–361
- 简介, xvii–xix
- 设计这款游戏, 329–330, 376
- 试驾, 370
- 随机放置战舰, 362–368, 380
- 战舰的起始位置, 366
- 作弊, 369
- 货真价实的战舰游戏中的表格单元格, 322
- I**
- id属性
 - 设置为数字字符串, 328
 - 用于标识元素, 243
 - 用于访问元素, 238
- IE8 (Internet Explorer 8)
 - 在IE8和更早的版本中设置事件对象, 399, 631
- if/else语句
 - 作决策, 23
 - 编写, 56
- if语句, 用于作决策, 22
- indexOf方法, 298, 343, 345, 352, 368, 610
- Infinity (或-Infinity), JavaScript, 274
- init函数, 249, 251, 359, 361, 369, 389–391, 630
- innerHTML属性, 627
- innerHTML属性
 - 简介, 245
 - 设置文本, 331
 - 替换元素<body>中元素的内容, 245
 - 修改DOM, 244
 - 修改元素内容, 239, 241–242, 247–248
- innerWidth属性, 627
- <input>元素
 - 添加到<form>中, 323
 - 使用CSS来设置其样式, 325
- instanceof运算符, 305, 315, 543, 547
- Internet Explorer 8 (IE8)
 - 在IE8和更早的版本中设置事件对象, 399, 631
- I/O事件, 413
- isNaN函数, 273, 353
- isSunk方法, 货真价实的战舰游戏, 346
- J**
- Java
 - 简介, 6
 - 与JavaScript比较, 5
- Java, 传统的面向对象编程, 564
- JavaScript
 - Infinity, 274
 - jQuery, 624
 - node.js库, 637
 - 保留字, 12
 - 编写. 参见“编写JavaScript”
 - 不区分大小写, 12
 - 处理异常, 629
 - 大统一理论, 612
 - 发展历程, 6
 - 服务器端, 637
 - 加入到网页中, 2–3
 - 简介, 1–2, x–xi
 - 蠕虫洞getElementById, 237

- 数组, 127, 132–134
- 思考, xxix
- 提供的对象, 214
- 通过DOM与网页交互, 237
- 学习, 9
- 应用, 5
- 与API相关的事件, 413
- 与DOM交互, 233
- 与Java比较, 5
- 语法, 13–14, 39
- 在网页中添加, 4–5
- 值, 272–274, 281, 292
- 重要性, 5
- JavaScript代码中的空白, 13
- JavaScript区分大小写, 12
 - 《JavaScript权威指南》, 296
- JavaScript语法
 - 查找错误, 14, 39
 - 简介, 13
- JavaScript中的注释, 13
- JavaScript中的字符, 297
- jQuery
 - 简介, 624–625
 - 在线文档和教程, 625
- Jscript, 6
- JSON对象, 214, 636
- 基本类型
 - 简介, 23, 266
 - 识别未定义的值, 267–268, 308
 - 与对象比较, 187
 - 作为对象和基本类型的字符串, 294–296
- 继承
 - 对象中的继承, 569–572
 - 改变内置行为, 608–609
 - 扩展内置的对象, 610
 - 一个原型继承另一个原型, 594–599, 606, 620
 - 原型链中的继承, 592–593, 619
- 级联样式表 (CSS)。另请参见 “CSS (级联样式表)”
- 计数器初始化, 迭代数组, 140
- 击中, 货真价实的战舰游戏
 - 表示击中和未击中的类, 326–327, 375
 - 告诉视图击中了战舰, 347
 - 判断是否击中, 344–345
 - 判断战舰是否被击沉, 346
 - 确定战舰的位置, 343
- 加号 (+)
 - 用于拼接字符串, 15, 133, 142, 354
 - 作为算术运算符, 286–287, 312
- 假值和真值, 291–293, 313
- 简单的战舰游戏
 - 过程型设计, 329
 - 检查用户的猜测, 54
 - 检查战舰是否被击沉, 57
 - 简介, 44
 - 设计, 45–46
 - 随机地指定战舰位置, 66–68
 - 添加HTML以链接代码, 45–46
 - 添加检测是否击中了战舰的代码, 56–57, 77
 - 伪代码
 - 编写, 47–48, 73
 - 判断是否击中, 59–60, 76
 - 确定需要的变量, 50
 - 实现循环, 51
 - 转换为JavaScript代码, 52
 - 向用户显示统计数据, 58
 - 找出代码中的错误, 68
 - 质量保证, 61, 69–70
- 减号 (-) 运算符
 - 用作处理字符串和数字的算术运算符, 286–287, 312
 - 用作单目运算符, 287
- 建立对象模型, 3
- 脚本语言, 5
- 解释型语言, 5
- 局部变量
 - 代码卫生指南, 111
 - 方法中的局部变量, 206
 - 局部变量的词法作用域, 488–489, 494
 - 嵌套的影响, 486, 488–490, 494
 - 声明, 103, 108
 - 生命周期, 102
 - 识别, 105, 120
 - 与全局变量比较, 99, 106–107
 - 遮住全局变量, 104
 - 作用域, 101
- 句点表示法 (.)
 - 访问对象的属性, 181, 209, 230
 - 与this对象一起使用, 202
 - 与引用变量一起使用, 186

K

keypress事件, 419

空数组, 134, 153–154, 365, 367, 549–550

控制台

查找错误, 68

打开浏览器, 28

简介, 214

在货真价实的战舰游戏中作弊, 370

L

lastIndexOf方法, 300

length属性, 297, 301, 303

LiveScript, 6

load事件

onload属性

设置为匿名函数, 476–477

设置为一个函数, 249, 359, 392, 394–395, 406–407, 422

指定处理程序, 385–386, 389–390, 410

简介, 419

location方法, 627

类

标识一组元素, 243

货真价实的战舰游戏中的类hit和miss, 326–327, 375

特性class, 255

类型

instanceof, 305, 315, 543, 547

typeof练习, 269

对象相等, 288–290, 314

简介, xvi–xvii

将==和===运算符用于不同的类型, 275–285, 311

使用null, 55, 270–271, 274, 310

熟练掌握类型, 301–304

值和类型, 272–274, 281, 292

执行算术运算时的自动类型转换, 286–287, 312

转换, 277–280

练习, xxxiii

列表

NodeList, 397

返回一个元素对象列表, 397

使用JavaScript在播放列表中添加歌曲(示例), 253, 262

使用jQuery找出id为playlist的元素的所有子元素, 625

零 (0), 被视假值, 292

流程图, 用于设计程序, 45

浏览器对象模型 (BOM), 627

浏览器

处理函数表达式, 434–436, 465

处理函数声明, 431–433, 436, 465, 483

处理事件, 383, 403

处理条件语句, 434

打开控制台, 28

分析HTML并创建DOM, 236, 261

根据网页设置窗口的尺寸, 627

加载并执行JavaScript, 3

加载代码, 31

事件, 404

提供的对象, 214

推荐, xxvi

运行JavaScript, xxxii

在IE8和更早的版本中设置事件对象, 399, 631

执行代码, 433

浏览器之战 6

逻辑运算符, 55, 71

M

Mac文本编辑器TextEdit, 31

match方法, 300, 633

Math.floor函数, 68, 131, 365–367

Math对象, 214, 551

Math.random函数, 67–68, 88, 131, 365–367

Microsoft, Jscript简介, 6

model对象, 货真价实的战舰游戏

fire方法, 342–343, 346

告诉视图是否击中, 347

跟踪战舰, 337

简介, 329, 336

判断是否击中, 344–345

判断是否击沉了战舰, 346

确定游戏板尺寸的函数parseGuess, 353

试驾, 348

实现, 341–348

与视图交互, 336

战舰的表示, 338–340, 378–379

战舰状态的表示, 337, 377

mousemove事件, 405–406, 419, 424

mouseout事件, 419

mouseover事件, 419

面向对象的编程, 180, 564

冒号 (:), 分隔属性名和属性值, 179

美元符号 (\$)

- jQuery中的函数, 624
- 用于JavaScript变量名开头, 13

命名

- 变量, 12–13, 100, 103, 108
- 构造函数, 532
- 函数, 97
- 属性, 179
- 同名的局部变量和全局变量, 104

目标, 398–399, 401

N

NaN (不是数字) 值, 272–274, 281, 292

Netscape, 6

node.js库, 637

NodeList, 397

null, 55, 256, 270–271, 274, 278, 292

内置对象, 548–551, 608–610

内置函数, 71, 91, 119

匿名函数

- 创建, 477–478, 512–513
- 简介, 475–476, 482, xx–xxi
- 可读性, 480
- 让代码更紧凑, 479
- 向函数传递函数, 482, 486, 514
- 用于访问属性, 509, 518
- 在构造函数中赋给方法, 530–532, 557

O

Object对象

- 改变内置行为, 608–609
- 简介, 551, 607
- 作为原型, 607

onclick属性, 390–391, 393–398, 504–507, 630

onload事件处理程序, 249, 359, 402

onload事件处理函数, 249–250, 359, 389–390

onload属性

- 将事件处理程序赋给它, 385–386, 389–390, 410, 630
- 设置为匿名函数, 476–477
- 设置为函数, 249, 359, 392, 394–395, 406–407, 422
- 用于支持BOM, 627

onmousemove属性, 405–406, 414, 424, 426

onmouseout属性, 414, 426

onresize属性, 415, 427

P

parseGuess函数, 351–354

parse方法, JSON中, 636

pause事件, 419

position: absolute, 324–325, 328

position: relative, 324, 328

<p> (段落) 元素

- 使用JavaScript来修改, 247
- 修改属性innerHTML, 242

print方法, 627

processGuess方法, 349–350, 357

prompt函数, 46, 53, 55

push方法, 152

拼接字符串, 15, 133, 142, 286–287, 312, 354

Q

querySelectorAll方法, 626

querySelector方法, 626

汽车自动制造机应用程序 (示例), 195–197

全局变量

- 代码卫生指南, 111
- 词法作用域, 488–490, 494
- 滥用, 391
- 嵌套的影响, 486, 488–490, 494
- 少用的原因, 108
- 生命周期, 102
- 识别, 105, 120
- 与局部变量比较, 99, 106–107
- 在JavaScript中滥用, 108
- 遮住, 104
- 作用域, 101

R

RegExp对象, 214, 551, 632
 removeEventListener, 630
 replace方法, 300
 reportError方法, 587, 618
 resize事件, 419
 return语句, 95–97

S

<script>元素

src特性, 34–35
 简介, 4
 剖析, 35

<script>标签

放在HTML页面的<head>或 <body>元素中, 32
 起始标签和结束标签, 35

setAttribute方法, 255, 333, 391

setInterval函数, 410, 413, 425

setInterval方法, 627

setTimeout函数, 410, 480, 501

setTimeout方法, 407–413, 627

showAnswer处理程序, 413

slice方法, 300

split方法, 299

<script>元素的src特性, 34–35

src属性, 390, 391

substring方法, 299–300, 302, 304

Sun Microsystems, 6

设计

货真价实的战舰游戏的设计, 329–330, 376
 简单的战舰游戏, 45–46

示例使用的HTML骨架, xxxiv

示例文件, xxxiv

示例文件, 下载, xxxiii

事件

DOM, 413
 dragstart, 419
 drop, 419

I/O, 413

keypress, 419

load 419。另请参见“加载事件”

mousemove, 405–406, 419, 424

mouseout, 419

mouseover, 419

onclick属性, 390–391, 393–398

pause, 419

resize, 419

touchend, 419

touchstart, 419

unload, 419

创建响应事件的游戏, 388–392

单击, 407, 419, 624

定时器, 407–409

队列和事件, 404

简介, 383, 419

与API相关的事件, 413

事件通知练习, 382, 421

事件响应练习, 415, 427

图像游戏和事件

添加图像, 393–397

指定单击事件处理程序, 396–397

将事件处理程序赋给属性onclick, 390–391, 393–398

创建, 411–414, 422, 426

事件类型, 407

事件列表, 419

使用函数setInterval, 410, 425

响应事件, 387

与浏览器讨论事件, 403

事件处理程序

onload, 249

timerHandler, 407

创建, 385–386

赋给属性, 407

回调函数和事件处理程序, 250

货真价实的战舰游戏中的事件处理程序, 358–359, 361

简介, 384

事件对象, 399–402, 423

使用addEventListener来添加, 630

使用函数setInterval, 410, 425

使用函数setTimeout, 407–413

异步编码

onload, 249

timerHandler, 407

创建, 385–386

赋给属性, 407

- 回调函数和异步编码, 250
 - 货真价实的战舰游戏中的异步编码, 358–359, 361
 - 使用函数setInterval, 410, 425
 - 使用函数setTimeout, 407–413
 - 种类, 252
 - 用来创建闭包, 503–507
 - 种类, 252
- 事件对象
 - 事件处理程序和事件对象, 399–402, 423
 - for属性, 401
 - 在IE8和更早的版本中设置事件对象, 399, 631
- 事件列表, 419
- 使用catch/try来处理异常, 629
- 使用jQuery变换界面元素, 625
- 使用方法扩展原型String, 610–611, 621
- 树, DOM的结构, 235
- 属性
 - JavaScript支持的属性, 296
 - 比较方法和函数, 206
 - 遍历对象的属性, 209
 - 代码错误查找练习, 203
 - 独特的属性, 546–547
 - 访问, 181
 - 继承, 571
 - 简介, 175
 - 没有愚蠢的问题, 185
 - 命名, 179
 - 判断来自实例还是原型, 586–587, 592, 618
 - 使用this对象, 202, 204–206, 219
 - 随时添加或删除, 184
 - 添加, 182
 - 未定义的值和属性, 268
 - 修改, 182
 - 原型链中的属性, 592–593, 619
 - 用于计算, 182
 - 在原型中修改, 582, 585
 - 字符串属性简明教程, 297–300
 - 值和属性, 176, 219
 - 作为属性集合的对象, 174
- 数字
 - NaN（不是数字）值和数字, 273–274, 281, 292
 - 函数prompt返回数字的字符串表示, 55
 - 将其与字符串比较, 275–277, 281
 - 使用运算符+来处理字符串和数字, 286–287
 - 作为对象, 296
 - 作为基本类型, 266
- 数组字面量语法, 550
- 数组
 - for循环, 140–142, 144–145, 147, 169
 - while循环, 17–21
 - 重用代码, 156
 - 初始化计数器, 140
 - 创建
 - 空, 151
 - 值, 128
 - 最有效的气泡配方代码, 164–165, 171
 - 代码排列练习, 139, 168
 - 迭代, 138, 140
 - 对象嵌套层级数, 348
 - 多少个元素, 134
 - 访问元素, 129
 - 更新元素, 129
 - 简介, xiii–xv
 - 将处理数组的代码组织为函数, 157–162
 - “汽车自动制造机”应用程序（示例）, 195–197
 - 空, 134, 153–154, 365, 367, 549–550
 - 密室商谈, 148–149, 170
 - 声明变量, 152
 - 试驾, 143, 147, 155
 - 属性length, 130
 - 索引, 129, 134, 152, 163
 - 填充播放列表, 253, 262
 - 未定义的值, 268
 - 稀疏, 152
 - 元素的类型, 134
 - 元素的排列顺序, 134
 - 在货真价实的战舰游戏中判断是否击中, 344–345
 - 在货真价实的战舰游戏中判断是否击中以及指定战舰的位置, 338
 - 自动造句应用程序（示例）, 131–133
 - 字面量, 151–152
- 数组的length属性, 130
- 数组的sort方法, 457–463, 472
- 双斜杠（//），用于JavaScript注释开头, 13
- 双引号（“ ”）
 - 用于将属性名括起, 179
 - 在JavaScript中用于将字符串括起, 13
- 随机地生成战舰的位置, 在货真价实的战舰游戏中, 362–368, 380
- 随机数的生成, 67–68

索引, 数组中, 129, 134, 152, 163

T

<table>元素

创建HTML页面, 322

使用CSS进行定位, 325

<td>元素, 322, 325, 327–328, 333

this对象

用法, 202, 204–206, 219

用于构造函数中, 528–529, 532

原型和this, 580

timerHandler函数, 407–409

toLowerCase方法, 300

toString方法, 531, 548, 557, 607–608

touchend事件, 419

touchstart事件, 419

toUpperCase方法, 300

trim方法, 300

true和false (布尔值), 13, 15–16, 291–293, 313

try/catch, 用于处理异常 629

typeof运算符, 269, 305, 309, 315, 542

特性

action, 328

alt, 256

class, 255

使用getAttribute来获取, 256

使用setAttribute来设置, 255

使用代码来设置, 254

元素没有时, 256

条件

别太啰嗦, 65, 74

浏览器如何处理, 434

使用布尔表达式和条件来作决策, 22

使用布尔运算符进行组合, 62–63, 74

在while语句中使用, 17–21

在数组中测试, 140

作为变量或字符串, 23

U

undefined, 变量的默认值, 50

undefined值

简介, 267–268, 308

没有return语句返回的值, 97

为假值, 292

与null比较, 278

与null的异同, 270, 310

Unicode值, 281

unload事件, 419

V

var关键字

简介, 11

用于变量声明, 97, 99

view对象, 货真价实的战舰游戏

简介, 329

试驾, 335

实现, 331–334

与模型交互, 336

指出是否击中了战舰, 347

W

W3C文档网站, 627

Web浏览器

处理函数表达式, 434–436, 465

处理函数声明, 431–433, 436, 465, 483

处理事件, 383, 403

处理条件语句, 434

打开控制台, 28

分析HTML并创建DOM, 261

根据网页调整窗口的尺寸, 627

加载代码, 31

加载并执行JavaScript, 3

事件, 404

提供的对象, 214

推荐的Web浏览器, xxvi

运行JavaScript, xxxii

在IE8和更早的版本中设置事件对象, 399, 631

执行代码, 433

Web浏览器之战, 6

while循环

简介, 17

用于执行代码, 18–21

与for比较, 144–145

wickedlysmart.com, viii

window对象

- addEventListener方法, 630
- confirm方法, 627
- history方法, 627
- innerHeight属性, 627
- innerWidth属性, 627
- location方法, 627
- onload属性
 - 将事件处理程序赋给它, 385–386, 389–390, 410, 630
 - 设置为匿名函数, 476–477
 - 设置为一个函数, 249, 359, 392, 394–395, 406–407, 422
 - 用于支持BOM, 627
- onresize属性, 415, 427
- print方法, 627
- setInterval方法, 627
- setTimeout函数, 407–413
- setTimeout方法, 627
- 包含的方法和属性, 627
- 创建onload事件处理程序, 249, 359, 402
- 对BOM的支持 627
- 简介, 214

完全相等运算符, 284。另请参见“严格相等(===)运算符”

网页

- 创建动态的网页, 5–6
- 创建交互式网页, 319
- 根据网页调整窗口的尺寸, 627
- 使用JavaScript与网页交互, 233
- 网页中的JavaScript, 2–3
- 在HTML页面中添加代码, 32
- 在网页中使用<script>元素, 4
- 作为应用程序, 9

伪代码

- 编写, 47–48, 73
- 确定需要的变量, 50
- 实现循环, 51
- 在简单的战舰游戏中判断是否击中, 59–60, 76
- 转换为JavaScript代码, 52, 164–165, 171

未击中, 货真价实的战舰游戏

- 表示击中和未击中的类, 326–327, 375
- 告诉视图击中了战舰, 347

文本编辑器, 31

文本编辑器“记事本”(Windows), 31

文档对象

DOM中的文档对象, 235, 626

getElementById方法

- 传递不存在的id, 245
- 返回null, 256, 270–271
- 访问图像, 388–392
- 获取元素, 231, 240, 247–248
- 区分大小写, 230
- 用于查找元素并修改其内容, 238–239
- 在货真价实的战舰游戏中获取指向按钮fire的引用, 359
- 作为文档对象, 237
- getElementsByClassName方法, 245, 626
- getElementsByName方法, 626
- getElementsByTagName方法, 245, 397
- 简介, 214, 240
- 用来编写函数以便与用户交流, 25–26, 42

无限循环, 55

X

稀疏数组, 152

下划线(_), 可用于变量名开头, 13

相等(==)运算符

- 一种比较运算符, 16, 55, 275–285, 311, 459
- 与===运算符的异同, 289

小于(<)运算符, 55, 459

小于或等于(<=)运算符, 55

斜杠(/)运算符, 用作除法算术运算符, 15, 286

星号(*)运算符, 用作乘法算术运算符, 15, 286

学习

JavaScript, 9

小提示, xxxi

学习原则, xxviii

循环

do while循环, 364, 373

for循环

- 重新编写, 147
- 用来迭代数组, 140–142, 169
- 与while循环比较, 144–145
- 在货真价实的战舰游戏中猜测战舰的位置, 343
- 在货真价实的战舰游戏中添加新战舰, 365, 367

while循环

- 简介, 17
- 用来执行代码, 18–21
- 与for循环比较, 144–145

将伪代码转换为JavaScript代码, 52
 使用内部循环和外部循环, 368
 使用伪代码来实现循环, 51
 无限循环, 55

Y

严格相等 (===) 运算符

一种比较运算符, 55, 280–285, 311
 与==运算符的异同, 289

异步编码

alt特性, 256
 DOM中的事件对象, 399–402, 423
 事件, 404
 事件处理程序
 onload, 249
 timerHandler, 407
 创建, 385–386
 赋给属性, 407
 回调函数, 250
 货真价实的战舰游戏, 358–359, 361
 简介, 383
 类型, 252
 使用addEventListener添加, 630
 使用函数setInterval, 410, 425
 使用函数setTimeout, 407–413

事件简介, 383

事件通知练习, 382, 421

响应事件

创建猜图游戏, 388–392, 411–414, 422, 426
 简介, 387
 将事件处理程序赋给, 390–391, 393–398
 练习, 415, 427
 在猜图游戏中添加图像, 393–397
 指定单击处理程序, 396–397

与浏览器讨论事件, 403

异步编码事件处理程序

事件处理程序
 创建闭包, 503–507

异常, 处理, 629

应用程序。另请参见“货真价实的战舰游戏”和“简单的战舰游戏”

JavaScript, 5
 编写JavaScript代码, 29–35
 创建交互式, 319
 作为应用程序的网页, 9

用户

与用户交流

使用函数alert, 25–26, 46
 使用函数console.log, 25–27
 使用函数prompt, 46, 53
 使用文档对象模型, 25

与API相关的事件, 413

语句

编写语句, 10
 以分号结尾, 11, 13
 语句中的变量, 11–13

与(&&)运算符, 55, 62–63, 74

圆括号 (())

函数调用中, 68, 430, 439
 形参列表中, 97

元认知, xxix

元素<div>

指定独一无二的id, 243
 使用CSS进行定位, 324–325

元素。另请参见“getElementById方法”

标识, 243
 从DOM中获取, 241
 根据id从DOM中获取, 245
 获取、创建、添加或删除, 258
 没有指定的属性, 256
 设置为目标, 398–399, 401
 使用CSS来定位, 324–326
 使用ElementById来获取, 238, 240
 使用类标识一组元素, 243
 用于标识一组元素的类, 243

元素对象

返回元素对象列表, 397
 简介, 231, 245

原型链

创建, 591
 继承, 592

原型式继承, 569–572

Z

在JavaScript中将字符串括起, 13

自动造句应用程序(示例), 131–133

字符串

prompt函数返回数字的字符串表示, 55
 包含的属性, 296
 空字符串为假值, 292
 排序运算符, 459

- 拼接, 15, 133, 142, 286–287, 312, 354
- 神秘生活, 294–296
- 用作条件, 23
- 与数字比较, 275–277, 281
- 字符串方法和属性简明教程, 297
- 字符串数组, 132
- 作为基本类型, 266
- 作为基本类型和对象, 294–296
- 字面量
 - 用来创建对象, 522
 - 与构造函数比较, 532, 539–542, 559
- 字面量, 数组中, 151–152
- 自由变量, 495–496, 501, 516
- 遮住变量, 104
- 侦听器, 384。另请参见“事件处理程序”
- 真值和假值, 291–293, 313
- 值
 - NaN（不是数字）, 272–274, 281, 292
 - null, 270
 - undefined, 268
 - 变量和值, 11
 - 对象, 266
 - 对象的属性及其值, 176, 219
 - 函数为一等公民
 - 从函数返回函数, 450–456, 470, 472
 - 简介, 442–443
 - 将函数传递给其他函数, 443–448, 468, 482, 486, 514
 - 替换函数表达式, 481
 - 使用数组的sort方法, 457–463, 472–473
 - 严峻的JavaScript挑战, 487, 508, 517
 - 基本类型, 266
 - 类型转换和值, 276–281
 - 排序运算符, 459
 - 识别undefined, 267–268, 308
 - 数组元素的类型, 134
 - 相等和值, 276
 - 像函数传递值, 92–93, 192
 - 严格相等, 280
 - 运算符和值, 286
 - 真值和假值, 291–293
 - 字符串, 291–297
 - 作为值的函数, 439–440, 467
- 指定单击事件处理程序, 396–397
- 质量保证（QA）
 - 货真价实的战舰游戏中的质量保证, 370–371
 - 简介, 61
 - 实施, 61, 69–70
- 作为学习者的读者, xxviii
- 作为一等公民的函数
 - 从函数返回函数, 450–456, 470, 472
 - 简介, 442–443
 - 使用数组的方法sort, 457–463, 472–473
 - 替换函数表达式, 481
 - 向函数传递函数, 443–448, 468, 482, 486, 514
 - 严峻的JavaScript挑战, 486, 508, 517

版 权 声 明

Copyright © 2014 by Eric T. Freeman, Elisabeth Robson

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2017. Authorized translation of the English edition, 2014 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2014。

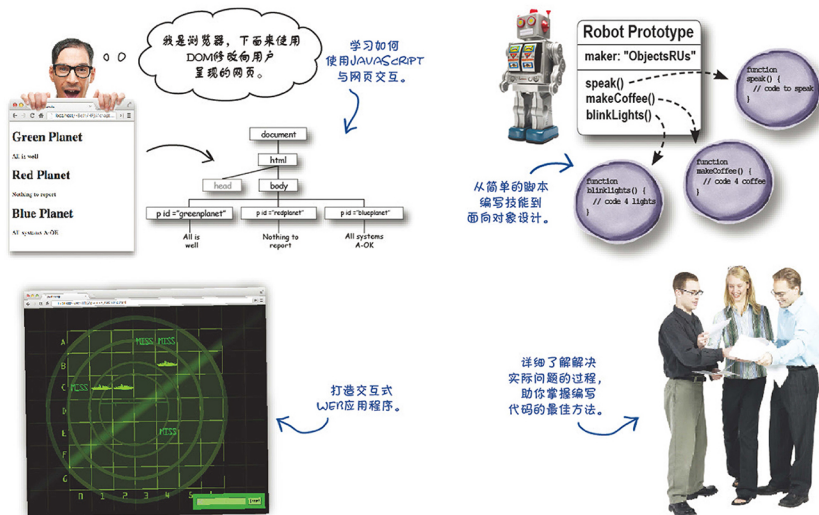
简体中文版由人民邮电出版社出版 2017。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

Head First JavaScript程序设计

你能从这本书中学到什么?

在这本趣学指南中,你将学习JavaScript语言的方方面面,包括基本知识以及对象、函数、浏览器文档对象模型等高级主题。在阅读过程中,你还将完成填字游戏、探查案情、以想象不到的方式与JavaScript交互。另外,你还将编写大量的代码,为独立打造Web应用程序打下坚实的基础。



这本书有什么特别之处?

本书基于神经生物学、认知学和学习理论领域的最新研究成果,采用了大量激发大脑的视觉元素,而不是充斥着让人昏昏欲睡的文字。

“这是一本优秀的编程入门教程,同时涵盖了对象创建、继承和闭包等高阶主题,帮助读者在掌握现代计算机编程基本知识的同时涉猎一些最有趣的概念。”

——Peter Casey,
中俄勒冈社区学院教授

“这本书引领读者进入JavaScript的幕后,深入了解这门非凡编程语言的工作原理。”

——Chris Fuselier,
工程咨询师

“要是我最初学JavaScript时有这本书就好了!”

——Daniel Konpacki,
华特迪士尼公司
资深软件工程师

JavaScript / Programming

封面设计: Randy Comer 张健

图灵社区: www.it-ebooks.com.cn

新浪微博: @图灵教育 @图灵社区

反馈/投稿/推荐信箱: contact@turingbook.com

热线: (010)51095186转604

分类建议 计算机/程序设计/JavaScript

O'Reilly Media, Inc. 授权人民邮电出版社出版

此简体中文版仅限于在中华人民共和国境内(但不允许在中国香港、澳门特别行政区和中国台湾地区)销售发行

This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-45841-4



ISBN 978-7-115-45841-4

定价: 129.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：
ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报

微博 @图灵社区：电子书和好文章的消息

微博 @图灵新知：图灵教育的科普小组

微信 图灵访谈：ituring_interview，讲述码农精彩人生

微信 图灵教育：turingbooks