重新介绍 JavaScript（JS 教程）

这篇翻译不完整。请帮忙从英语翻译这篇文章。

为什么会有这一篇“重新介绍”呢？因为 JavaScript 堪称 世界上被人误解最深的编程语言。虽然常被嘲为“玩具语言”，但在它看似简洁的外衣下，还隐藏着强大的语言特性。 JavaScript 目前广泛应用于众多知名应用中，对于网页和移动开发者来说，深入理解 JavaScript 就尤为必要。

先从这门语言的历史谈起是有必要的。在1995 年 Netscape 一位名为 Brendan Eich 的工程师创造了 JavaScript，随后在 1996 年初，JavaScript 首先被应用于 Netscape 2 浏览器上。最初的 JavaScript 名为 LiveScript，后来，因为 Sun Microsystem 的 Java 语言兴起，被广泛使用，Netscape 出于宣传和推广的考虑，将它的名字从最初的 LiveScript 更改为 JavaScript——尽管两者之间并没有什么共同点。这便是之后混淆产生的根源。

几个月后，Microsoft 随 IE 3 发布推出了一个与之基本兼容的语言 JScript。又过了几个月，Netscape 将 JavaScript 提交至 Ecma International（一个欧洲标准化组织）， ECMAScript 标准第一版便在 1997 年诞生了，随后在 1999 年以 ECMAScript 第三版的形式进行了更新，从那之后这个标准没有发生过大的改动。由于委员会在语言特性的讨论上发生分歧，ECMAScript 第四版尚未推出便被废除，但随后于 2009 年 12 月发布的 ECMAScript 第五版引入了第四版草案加入的许多特性。第六版标准已经于 2015 年 6 月发布。

> 注意：为熟悉起见，从这里开始，我们将使用 “JavaScript” 代替 ECMAScript 。

与大多数编程语言不同，JavaScript 没有输入或输出的概念。它是一个在主机环境（host environment）下运行的脚本语言，任何与外界沟通的机制都是由主机环境提供的。浏览器是最常见的主机环境，但在非常多的其他程序中也包含 JavaScript 解释器，如 Adobe Acrobat、

Photoshop、SVG 图像、Yahoo! 的 Widget 引擎，以及 Node.js 之类的服务器端环境。JavaScript 的实际应用远不止这些，除此之外还有 NoSQL 数据库（如开源的 Apache CouchDB）、嵌入式计算机，以及包括 GNOME （注：GNU/Linux 上最流行的 GUI 之一）在内的桌面环境等等。

概览

JavaScript 是一种多范式的动态语言，它包含类型、运算符、标准内置（ built-in）对象和方法。它的语法来源于 Java 和 C，所以这两种语言的许多语法特性同样适用于 JavaScript。JavaScript 通过原型链而不是类来支持面向对象编程（有关 ES6 类的内容参考这里`Classes`，有关对象原型参考见此继承与原型链）。JavaScript同样支持函数编程-因为它们也是对象，函数也可以被保存在变量中，并且像其他对象一样被传递。

先从任何编程语言都不可缺少的组成部分——“类型”开始。JavaScript 程序可以修改值（value），这些值都有各自的类型。JavaScript 中的类型包括：

- `Number`（数字）
- `String`（字符串）
- `Boolean`（布尔）
- `Function`（函数）
- `Object`（对象）
- `Symbol`（ES2015 新增）

…哦，还有看上去有些…奇怪的 `undefined`（未定义）类型和 `null`（空）类型。此外还有 `Array`（数组）类型，以及分别用于表示日期和正则表达式的 `Date`（日期）和 `RegExp`（正则表达式），这三种类型都是特殊的对象。严格意义上说，Function（函数）也是一种特殊的对象。所以准确来说，JavaScript 中的类型应该包括这些：

- `Number`（数字）
- `String`（字符串）
- `Boolean`（布尔）
- `Symbol`（符号）（ES2015 新增）
- `Object`（对象）
    - `Function`（函数）
    - `Array`（数组）

    - `Date`（日期）
    - `RegExp`（正则表达式）
- `null`（空）
- `undefined`（未定义）

JavaScript 还有一种内置的 `Error`（错误）类型。但是，如果我们继续使用上面的分类，事情便容易得多；所以，现在，我们先讨论上面这些类型。

数字

根据语言规范，JavaScript 采用“遵循 IEEE 754 标准的双精度 64 位格式”（"double-precision 64-bit format IEEE 754 values"）表示数字。据此我们能得到一个有趣的结论，和其他编程语言（如 C 和 Java）不同，JavaScript 不区分整数值和浮点数值，所有数字在 JavaScript 中均用浮点数值表示，所以在进行数字运算的时候要特别注意。看看下面的例子：

```
0.1 + 0.2 = 0.30000000000000004
```

在具体实现时，整数值通常被视为32位整型变量，在个别实现（如某些浏览器）中也以32位整型变量的形式进行存储，直到它被用于执行某些32位整型不支持的操作，这是为了便于进行位操作。

JavaScript 支持标准的算术运算符，包括加法、减法、取模（或取余）等等。还有一个之前没有提及的内置对象 `Math`（数学对象），用以处理更多的高级数学函数和常数：

```
Math.sin(3.5);
var circumference = 2 * Math.PI * r;
```

你可以使用内置函数 `parseInt()` 将字符串转换为整型。该函数的第二个可选参数表示字符串所表示数字的基（进制）：

```
parseInt("123", 10); // 123
parseInt("010", 10); // 10
```

一些老版本的浏览器会将首字符为“0”的字符串当做八进制数字，2013 年以前的 JavaScript 实现会返回一个意外的结果：

```
parseInt("010");  //  8
parseInt("0x10"); // 16
```

这是因为字符串以数字 0 开头，`parseInt()`函数会把这样的字符串视作八进制数字；同理，0x开头的字符串则视为十六进制数字。

如果想把一个二进制数字字符串转换成整数值，只要把第二个参数设置为 2 就可以了：

```
parseInt("11", 2); // 3
```

JavaScript 还有一个类似的内置函数 `parseFloat()`，用以解析浮点数字符串，与`parseInt()`不同的地方是，parseFloat()只应用于解析十进制数字。

单元运算符 + 也可以把数字字符串转换成数值：

```
+ "42";   // 42
+ "010";  // 10
+ "0x10"; // 16
```

如果给定的字符串不存在数值形式，函数会返回一个特殊的值 `NaN`（Not a Number 的缩写）：

```
parseInt("hello", 10); // NaN
```

要小心NaN：如果把 `NaN` 作为参数进行任何数学运算，结果也会是 `NaN`：

```
NaN + 5; //NaN
```

可以使用内置函数 `isNaN()` 来判断一个变量是否为 `NaN`：

```
isNaN(NaN); // true
```

JavaScript 还有两个特殊值：`Infinity`（正无穷）和 `-Infinity`（负无穷）：

```
1 / 0; //  Infinity
-1 / 0; // -Infinity
```

可以使用内置函数 `isFinite()` 来判断一个变量是否是一个有穷数， 如果类型为 `Infinity`, `-Infinity` 或 `NaN`则返回`false`：

```
isFinite(1/0); // false
isFinite(Infinity); // false
isFinite(-Infinity); // false
isFinite(NaN); // false

isFinite(0); // true
isFinite(2e64); // true

isFinite("0"); // true
// 如果是纯数值类型的检测，则返回 false:
Number.isFinite("0"); // false
```

> 备注： `parseInt()` 和 `parseFloat()` 函数会尝试逐个解析字符串中的字符，直到遇上一个无法被解析成数字的字符，然后返回该字符前所有数字字符组成的数字。然而如果使用运算符 "+"， 只要字符串中含有无法被解析成数字的字符，该字符串都将被转换成 `NaN` 。可分别使用这两种方法解析“10.2abc”这一字符串，并比较得到的结果，来理解这两种方法的区别。

字符串

JavaScript 中的字符串是一串Unicode 字符序列。这对于那些需要和多语种网页打交道的开发者来说是个好消息。更准确地说，它们是一串UTF-16编码单元的序列，每一个编码单元由一个 16 位二进制数表示。每一个Unicode字符由一个或两个编码单元来表示。

如果想表示一个单独的字符，只需使用长度为 1 的字符串。

通过访问字符串的 `length` （编码单元的个数）属性，可以得到它的长度。

```
"hello".length; // 5
```

这是我们第一次碰到 JavaScript 对象。我们有没有提过你可以像 object 一样使用字符串？是的，字符串也有 methods（方法）能让你操作字符串和获取字符串的信息。

```
"hello".charAt(0); // "h"
```

```
"hello, world".replace("world", "mars"); // "hello, mars"
"hello".toUpperCase(); // "HELLO"
```

其他类型

与其他类型不同，JavaScript 中的 `null` 表示一个空值（non-value），必须使用 null 关键字才能访问，`undefined` 是一个“undefined（未定义）”类型的对象，表示一个未初始化的值，也就是还没有被分配的值。我们之后再具体讨论变量，但有一点可以先简单说明一下，JavaScript 允许声明变量但不对其赋值，一个未被赋值的变量就是 `undefined` 类型。还有一点需要说明的是，`undefined` 实际上是一个不允许修改的常量。

JavaScript 包含布尔类型，这个类型的变量有两个可能的值，分别是 `true` 和 `false`（两者都是关键字）。根据具体需要，JavaScript 按照如下规则将变量转换成布尔类型：

1. `false`、`0`、空字符串（`""`）、`NaN`、`null` 和 `undefined` 被转换为 `false`
2. 所有其他值被转换为 `true`

也可以使用 `Boolean()` 函数进行显式转换：

```
Boolean(""); // false
Boolean(234); // true
```

不过一般没必要这么做，因为 JavaScript 会在需要一个布尔变量时隐式完成这个转换操作（比如在 `if` 条件语句中）。所以，有时我们可以把转换成布尔值后的变量分别称为 真值（true values）——即值为 true 和 假值（false values）——即值为 false；也可以分别称为“真的”（truthy）和“假的”（falsy）。

JavaScript 支持包括 `&&`（逻辑与）、`||`（逻辑或）和 `!`（逻辑非）在内的一些逻辑运算符。下面会有所提到。

变量

在 JavaScript 中声明一个新变量的方法是使用关键字 `let`、`const` 和 `var`：

let 语句声明一个块级作用域的本地变量，并且可选的将其初始化为一个值。

```
let a;
let name = 'Simon';
```

下面是使用 **let** 声明变量作用域的例子：

```
// myLetVariable is *not* visible out here

for (let myLetVariable = 0; myLetVariable < 5; myLetVariable++) {
  // myLetVariable is only visible in here
}

// myLetVariable is *not* visible out here
```

const 允许声明一个不可变的常量。这个常量在定义域内总是可见的。

```
const Pi = 3.14; // 设置 Pi 的值
Pi = 1; // 将会抛出一个错误因为你改变了一个常量的值。
```

var 是最常见的声明变量的关键字。它没有其他两个关键字的种种限制。这是因为它是传统上在 JavaScript 声明变量的唯一方法。使用 **var** 声明的变量在它所声明的整个函数都是可见的。

```
var a;
var name = "simon";
```

一个使用 **var** 声明变量的语句块的例子：

```
// myVarVariable *is* visible out here

for (var myVarVariable = 0; myVarVariable < 5; myVarVariable++) {
  // myVarVariable is visible to the whole function
}

// myVarVariable *is* visible out here
```

如果声明了一个变量却没有对其赋值，那么这个变量的类型就是 `undefined`。

JavaScript 与其他语言的（如 Java）的重要区别是在 JavaScript 中语句块（blocks）是没有作用域的，只有函数有作用域。因此如果在一个复合语句中（如 if 控制结构中）使用 var 声明一个变量，那么它的作用域是整个函数（复合语句在函数中）。 但是从 ECMAScript Edition 6 开始将有所不同的， `let` 和 `const` 关键字允许你创建块作用域的变量。

运算符

JavaScript的算术操作符包括 `+`、`-`、`*`、`/` 和 `%` ——求余（与模运算相同）。赋值使用 `=` 运算符，此外还有一些复合运算符，如 `+=` 和 `-=`，它们等价于 `x = x operator y`。

```
x += 5; // 等价于 x = x + 5;
```

可以使用 `++` 和 `--` 分别实现变量的自增和自减。两者都可以作为前缀或后缀操作符使用。

`+` 操作符还可以用来连接字符串：

```
"hello" + " world"; // hello world
```

如果你用一个字符串加上一个数字（或其他值），那么操作数都会被首先转换为字符串。如下所示：

```
"3" + 4 + 5; // 345
3 + 4 + "5"; // 75
```

这里不难看出一个实用的技巧——通过与空字符串相加，可以将某个变量快速转换成字符串类型。

JavaScript 中的比较操作使用 `<`、`>`、`<=` 和 `>=`，这些运算符对于数字和字符串都通用。相等的比较稍微复杂一些。由两个“`=`（等号）”组成的相等运算符有类型自适应的功能，具体例子如下：

```
123 == "123" // true
1 == true; // true
```

如果在比较前不需要自动类型转换，应该使用由三个“= （等号）”组成的相等运算符：

```
1 === true; //false
123 === "123"; // false
```

JavaScript 还支持 `!=` 和 `!==` 两种不等运算符，具体区别与两种相等运算符的区别类似。

JavaScript 还提供了 位操作符。

控制结构

JavaScript 的控制结构与其他类 C 语言类似。可以使用 `if` 和 `else` 来定义条件语句，还可以连起来使用：

```
var name = "kittens";
if (name == "puppies") {
  name += "!";
} else if (name == "kittens") {
  name += "!!";
} else {
  name = "!" + name;
}
name == "kittens!!"; // true
```

JavaScript 支持 `while` 循环和 `do-while` 循环。前者适合常见的基本循环操作，如果需要循环体至少被执行一次则可以使用 `do-while`：

```
while (true) {
  // 一个无限循环!
}

var input;
do {
  input = get_input();
} while (inputIsNotValid(input))
```

JavaScript 的 `for` 循环与 C 和 Java 中的相同：使用时可以在一行代码中提供控制信息。

```
for (var i = 0; i < 5; i++) {
  // 将会执行五次
}
```

JavaScript 也还包括其他两种重要的 for 循环： `for…of`

```
for (let value of array) {
  // do something with value
}
```

和 `for…in` ：

```
for (let property in object) {
  // do something with object property
}
```

`&&` 和 `||` 运算符使用短路逻辑（short-circuit logic），是否会执行第二个语句（操作数）取决于第一个操作数的结果。在需要访问某个对象的属性时，使用这个特性可以事先检测该对象是否为空：

```
var name = o && o.getName();
```

或用于缓存值（当错误值无效时）：

```
var name = cachedName || (cachedName = getName());
```

类似地，JavaScript 也有一个用于条件表达式的三元操作符：

```
var allowed = (age > 18) ? "yes" : "no";
```

在需要多重分支时可以使用 基于一个数字或字符串的`switch` 语句：

```
switch(action) {
    case 'draw':
        drawIt();
        break;
    case 'eat':
        eatIt();
        break;
    default:
        doNothing();
}
```

如果你不使用 `break` 语句，JavaScript 解释器将会执行之后 `case` 中的代码。除非是为了调试，一般你并不需要这个特性，所以大多数时候不要忘了加上 `break`。

```
switch(a) {
    case 1: // 继续向下
    case 2:
        eatIt();
        break;
    default:
        doNothing();
}
```

`default` 语句是可选的。`switch` 和 `case` 都可以使用需要运算才能得到结果的表达式；在 `switch` 的表达式和 `case` 的表达式是使用 `===` 严格相等运算符进行比较的：

```
switch(1 + 3){
    case 2 + 2:
        yay();
        break;
    default:
        neverhappens();
}
```

对象

JavaScript 中的对象，Object，可以简单理解成“名称-值”对（而不是键值对：现在，ES 2015 的映射表（Map），比对象更接近键值对），不难联想 JavaScript 中的对象与下面这些概念类似：

- Python 中的字典（Dictionary）
- Perl 和 Ruby 中的散列/哈希（Hash）
- C/C++ 中的散列表（Hash table）
- Java 中的散列映射表（HashMap）
- PHP 中的关联数组（Associative array）

这样的数据结构设计合理，能应付各类复杂需求，所以被各类编程语言广泛采用。正因为 JavaScript 中的一切（除了核心类型，core object）都是对象，所以 JavaScript 程序必然与大量的散列表查找操作有着千丝万缕的联系，而散列表擅长的正是高速查找。

“名称”部分是一个 JavaScript 字符串，“值”部分可以是任何 JavaScript 的数据类型——包括对象。这使用户可以根据具体需求，创建出相当复杂的数据结构。

有两种简单方法可以创建一个空对象：

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

和：

```
var obj = {};
```

这两种方法在语义上是相同的。第二种更方便的方法叫作“对象字面量（object literal）”法。这种也是 JSON 格式的核心语法，一般我们优先选择第二种方法。

“对象字面量”也可以用来在对象实例中定义一个对象：

```
var obj = {
    name: "Carrot",
    "for": "Max",//'for' 是保留字之一，使用'_for'代替
    details: {
        color: "orange",
        size: 12
    }
}
```

对象的属性可以通过链式（chain）表示方法进行访问：

```
obj.details.color; // orange
obj["details"]["size"]; // 12
```

下面的例子创建了一个对象原型，**Person**，和这个原型的实例，**You**。

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

// 定义一个对象
var You = new Person("You", 24);
// 我们创建了一个新的 Person，名称是 "You"
// ("You" 是第一个参数，24 是第二个参数..)
```

完成创建后，对象属性可以通过如下两种方式进行赋值和访问：

```
obj.name = "Simon"
var name = obj.name;
```

和：

```
// bracket notation
obj['name'] = 'Simon';
var name = obj['name'];
// can use a variable to define a key
var user = prompt('what is your key?')
obj[user] = prompt('what is its value?')
```

这两种方法在语义上也是相同的。第二种方法的优点在于属性的名称被看作一个字符串，这就意味着它可以在运行时被计算，缺点在于这样的代码有可能无法在后期被解释器优化。它也可以被用来访问某些以预留关键字作为名称的属性的值：

```
obj.for = "Simon"; // 语法错误，因为 for 是一个预留关键字
obj["for"] = "Simon"; // 工作正常
```

> 注意：从 EcmaScript 5 开始，预留关键字可以作为对象的属性名（reserved words may be used as object property names "in the buff"）。这意味着当定义对象字面量时不需要用双引号了。参见 ES5 Spec.

关于对象和原型的详情参见： Object.prototype. 解释对象原型和对象原型链可以参见：继承与原型链。

> 注意：从 EcmaScript 6 开始，对象键可以在创建时使用括号表示法由变量定义。`{[phoneType]: 12345}` 可以用来替换 `var userPhone = {}; userPhone[phoneType] = 12345 .`

数组

JavaScript 中的数组是一种特殊的对象。它的工作原理与普通对象类似（以数字为属性名，但只能通过 `[]` 来访问），但数组还有一个特殊的属性——`length`（长度）属性。这个属性的值通常比数组最大索引大 1。

创建数组的传统方法是：

```
var a = new Array();
a[0] = "dog";
a[1] = "cat";
a[2] = "hen";
a.length; // 3
```

使用数组字面量（array literal）法更加方便：

```
var a = ["dog", "cat", "hen"];
a.length; // 3
```

注意，`Array.length` 并不总是等于数组中元素的个数，如下所示：

```
var a = ["dog", "cat", "hen"];
a[100] = "fox";
a.length; // 101
```

记住：数组的长度是比数组最大索引值多一的数。

如果试图访问一个不存在的数组索引，会得到 `undefined`：

```
typeof(a[90]); // undefined
```

可以通过如下方式遍历一个数组：

```
for (var i = 0; i < a.length; i++) {
    // Do something with a[i]
}
```

ES2015 引入了更加简洁的 `for…of` 循环，可以用它来遍历可迭代对象，例如数组：

```
for (const currentValue of a) {
  // Do something with currentValue
}
```

遍历数组的另一种方法是使用 `for...in` 循环， 然而这并不是遍历数组元素而是数组的索引。注意，如果哪个家伙直接向 `Array.prototype` 添加了新的属性，使用这样的循环这些属性也同样会被遍历。所以并不推荐使用这种方法遍历数组：

```
for (var i in a) {
  // Do something with a[i]
}
```

ECMAScript 5 增加了另一个遍历数组的方法，`forEach()`：

```
["dog", "cat", "hen"].forEach(function(currentValue, index, array) {
  // Do something with currentValue or array[index]
});
```

如果想在数组后追加元素，只需要：

```
a.push(item);
```

除了 forEach() 和 push()，Array（数组）类还自带了许多方法。建议查看 Array 方法的完整文档。

方法名称	描述
a.toString()	返回一个包含数组中所有元素的字符串，每个元素通过逗号分隔。
a.toLocaleString()	根据宿主环境的区域设置，返回一个包含数组中所有元素的字符串，每个元素通过逗号分隔。
a.concat(item1[, item2[, ...[, itemN]]])	返回一个数组，这个数组包含原先 a 和 item1、item2、……、itemN 中的所有元素。
a.join(sep)	返回一个包含数组中所有元素的字符串，每个元素通过指定的 sep 分隔。
a.pop()	删除并返回数组中的最后一个元素。
a.push(item1, ..., itemN)	将 item1、item2、……、itemN 追加至数组 a。
a.reverse()	数组逆序（会更改原数组 a）。
a.shift()	删除并返回数组中第一个元素。
a.slice(start, end)	返回子数组，以 a[start] 开头，以 a[end] 前一个元素结尾。
a.sort([cmpfn])	依据可选的比较函数 cmpfn 进行排序，如果未指定比较函数，则按字符顺序比较（即使被比较元素是数字）。
a.splice(start, delcount[, item1[, ...[, itemN]]])	从 start 开始，删除 delcount 个元素，然后插入所有的 item。
a.unshift(item1[, item2[, ...[, itemN]]])	将 item 插入数组头部，返回数组新长度（考虑 undefined）。

函数

学习 JavaScript 最重要的就是要理解对象和函数两个部分。最简单的函数就像下面这个这么简单：

```
function add(x, y) {
    var total = x + y;
    return total;
}
```

这个例子包括你需要了解的关于基本函数的所有部分。一个 JavaScript 函数可以包含 0 个或多个已命名的变量。函数体中的表达式数量也没有限制。你可以声明函数自己的局部变量。`return` 语句在返回一个值并结束函数。如果没有使用 `return` 语句，或者一个没有值的 `return` 语句，JavaScript 会返回 `undefined`。

已命名的参数更像是一个指示而没有其他作用。如果调用函数时没有提供足够的参数，缺少的参数会被 `undefined` 替代。

```
add(); // NaN
// 不能在 undefined 对象上进行加法操作
```

你还可以传入多于函数本身需要参数个数的参数：

```
add(2, 3, 4); // 5
 // 将前两个值相加，4 被忽略了
```

这看上去有点蠢。函数实际上是访问了函数体中一个名为 `arguments` 的内部对象，这个对象就如同一个类似于数组的对象一样，包括了所有被传入的参数。让我们重写一下上面的函数，使它可以接收任意个数的参数：

```
function add() {
    var sum = 0;
    for (var i = 0, j = arguments.length; i < j; i++) {
        sum += arguments[i];
    }
    return sum;
}

add(2, 3, 4, 5); // 14
```

这跟直接写成 `2 + 3 + 4 + 5` 也没什么区别。我们还是创建一个求平均数的函数吧：

```
function avg() {
    var sum = 0;
    for (var i = 0, j = arguments.length; i < j; i++) {
        sum += arguments[i];
    }
    return sum / arguments.length;
}
avg(2, 3, 4, 5); // 3.5
```

这个就有用多了，但是却有些冗长。为了使代码变短一些，我们可以使用剩余参数来替换arguments的使用。在这方法中，我们可以传递任意数量的参数到函数中同时尽量减少我们的代码。这个**剩余参数操作符**在函数中以：**...variable** 的形式被使用，它将包含在调用函数时使用的未捕获整个参数列表到这个变量中。我们同样也可以将 **for** 循环替换为 **for...of** 循环来返回我们变量的值。

```
function avg(...args) {
  var sum = 0;
  for (let value of args) {
    sum += value;
  }
  return sum / args.length;
}

avg(2, 3, 4, 5); // 3.5
```

> 在上面这段代码中，所有被传入该函数的参数都被变量 **args** 所持有。
>
> 需要注意的是，无论“剩余参数操作符”被放置到函数声明的哪里，它都会把除了自己之前的所有参数存储起来。比如函数：function avg(**firstValue**, ...args) 会把传入函数的第一个值存入 **firstValue**，其他的参数存入 **args**。这是虽然一个很有用的语言特性，却也会带来新的问题。`avg()` 函数只接受逗号分开的参数列表 -- 但是如果你想要获取一个数组的平均值怎么办？一种方法是将函数按照如下方式重写：

```
function avgArray(arr) {
    var sum = 0;
    for (var i = 0, j = arr.length; i < j; i++) {
        sum += arr[i];
    }
    return sum / arr.length;
```

```
}
avgArray([2, 3, 4, 5]); // 3.5
```

但如果能重用我们已经创建的那个函数不是更好吗？幸运的是 JavaScript 允许你通过任意函数对象的 `apply()` 方法来传递给它一个数组作为参数列表。

```
avg.apply(null, [2, 3, 4, 5]); // 3.5
```

传给 `apply()` 的第二个参数是一个数组，它将被当作 `avg()` 的参数列表使用，至于第一个参数 `null`，我们将在后面讨论。这也正说明了一个事实——函数也是对象。

> 通过使用展开语法，你也可以获得同样的效果。
>
> 比如说：`avg(...numbers)`

JavaScript 允许你创建匿名函数：

```
var avg = function() {
    var sum = 0;
    for (var i = 0, j = arguments.length; i < j; i++) {
        sum += arguments[i];
    }
    return sum / arguments.length;
};
```

这个函数在语义上与 `function avg()` 相同。你可以在代码中的任何地方定义这个函数，就像写普通的表达式一样。基于这个特性，有人发明出一些有趣的技巧。与 C 中的块级作用域类似，下面这个例子隐藏了局部变量：

```
var a = 1;
var b = 2;
(function() {
    var b = 3;
    a += b;
})();

a; // 4
b; // 2
```

JavaScript 允许以递归方式调用函数。递归在处理树形结构（比如浏览器 DOM）时非常有用。

```
function countChars(elm) {
    if (elm.nodeType == 3) { // TEXT_NODE 文本节点
        return elm.nodeValue.length;
    }
    var count = 0;
    for (var i = 0, child; child = elm.childNodes[i]; i++) {
        count += countChars(child);
    }
    return count;
}
```

这里需要说明一个潜在问题——既然匿名函数没有名字，那该怎么递归调用它呢？在这一点上，JavaScript 允许你命名这个函数表达式。你可以命名立即调用的函数表达式（IIFE——Immediately Invoked Function Expression），如下所示：

```
var charsInBody = (function counter(elm) {
    if (elm.nodeType == 3) { // 文本节点
        return elm.nodeValue.length;
    }
    var count = 0;
    for (var i = 0, child; child = elm.childNodes[i]; i++) {
        count += counter(child);
    }
    return count;
})(document.body);
```

如上所提供的函数表达式的名称的作用域仅仅是该函数自身。这允许引擎去做更多的优化，并且这种实现更可读、友好。该名称也显示在调试器和一些堆栈跟踪中，节省了调试时的时间。

需要注意的是 JavaScript 函数是它们本身的对象——就和 JavaScript 其他一切一样——你可以给它们添加属性或者更改它们的属性，这与前面的对象部分一样。

自定义对象

> 备注：关于 JavaScript 中面向对象编程更详细的信息，请参考 JavaScript 面向对象简介。

在经典的面向对象语言中，对象是指数据和在这些数据上进行的操作的集合。与 C++ 和 Java 不同，JavaScript 是一种基于原型的编程语言，并没有 class 语句，而是把函数用作类。那么让我们来定义一个人名对象，这个对象包括人的姓和名两个域（field）。名字的表示有两种方法：“名 姓（First Last）”或“姓, 名（Last, First）”。使用我们前面讨论过的函数和对象概念，可以像这样完成定义：

```
function makePerson(first, last) {
    return {
        first: first,
        last: last
    }
}
function personFullName(person) {
    return person.first + ' ' + person.last;
}
function personFullNameReversed(person) {
    return person.last + ', ' + person.first
}
s = makePerson("Simon", "Willison");
personFullName(s); // Simon Willison
personFullNameReversed(s); // Willison, Simon
```

上面的写法虽然可以满足要求，但是看起来很麻烦，因为需要在全局命名空间中写很多函数。既然函数本身就是对象，如果需要使一个函数隶属于一个对象，那么不难得到：

```
function makePerson(first, last) {
    return {
        first: first,
        last: last,
        fullName: function() {
            return this.first + ' ' + this.last;
        },
        fullNameReversed: function() {
            return this.last + ', ' + this.first;
        }
    }
}
s = makePerson("Simon", "Willison");
s.fullName(); // Simon Willison
s.fullNameReversed(); // Willison, Simon
```

上面的代码里有一些我们之前没有见过的东西：关键字 this。当使用在函数中时，this 指代当前的对象，也就是调用了函数的对象。如果在一个对象上使用点或者方括号来访问属性或方法，这个对象就成了 this。如果并没有使用“点”运算符调用某个对象，那么 this 将指向全局对象（global object）。这是一个经常出错的地方。例如：

```
s = makePerson("Simon", "Willison");
var fullName = s.fullName;
fullName(); // undefined undefined
```

当我们调用 fullName() 时，this 实际上是指向全局对象的，并没有名为 first 或 last 的全局变量，所以它们两个的返回值都会是 undefined。

下面使用关键字 this 改进已有的 makePerson 函数：

```
function Person(first, last) {
    this.first = first;
    this.last = last;
    this.fullName = function() {
        return this.first + ' ' + this.last;
    }
    this.fullNameReversed = function() {
        return this.last + ', ' + this.first;
    }
}
var s = new Person("Simon", "Willison");
```

我们引入了另外一个关键字：new，它和 this 密切相关。它的作用是创建一个崭新的空对象，然后使用指向那个对象的 this 调用特定的函数。注意，含有 this 的特定函数不会返回任何值，只会修改 this 对象本身。new 关键字将生成的 this 对象返回给调用方，而被 new 调用的函数称为构造函数。习惯的做法是将这些函数的首字母大写，这样用 new 调用他们的时候就容易识别了。

不过，这个改进的函数还是和上一个例子一样，在单独调用 fullName() 时，会产生相同的问题。

我们的 Person 对象现在已经相当完善了，但还有一些不太好的地方。每次我们创建一个 Person 对象的时候，我们都在其中创建了两个新的函数对象——如果这个代码可以共享不是更好吗?

```
function personFullName() {
    return this.first + ' ' + this.last;
}
function personFullNameReversed() {
    return this.last + ', ' + this.first;
}
function Person(first, last) {
    this.first = first;
    this.last = last;
    this.fullName = personFullName;
    this.fullNameReversed = personFullNameReversed;
}
```

这种写法的好处是，我们只需要创建一次方法函数，在构造函数中引用它们。那是否还有更好的方法呢？答案是肯定的。

```
function Person(first, last) {
    this.first = first;
    this.last = last;
}
Person.prototype.fullName = function() {
    return this.first + ' ' + this.last;
}
Person.prototype.fullNameReversed = function() {
    return this.last + ', ' + this.first;
}
```

`Person.prototype` 是一个可以被`Person`的所有实例共享的对象。它是一个名叫原型链（prototype chain）的查询链的一部分：当你试图访问一个 `Person` 没有定义的属性时，解释器会首先检查这个 `Person.prototype` 来判断是否存在这样一个属性。所以，任何分配给 `Person.prototype` 的东西对通过 `this` 对象构造的实例都是可用的。

这个特性功能十分强大，JavaScript 允许你在程序中的任何时候修改原型（prototype）中的一些东西，也就是说你可以在运行时(runtime)给已存在的对象添加额外的方法：

```
s = new Person("Simon", "Willison");
s.firstNameCaps();  // TypeError on line 1: s.firstNameCaps is not a

Person.prototype.firstNameCaps = function() {
    return this.first.toUpperCase()
```

```
}
s.firstNameCaps(); // SIMON
```

有趣的是，你还可以给 JavaScript 的内置函数原型（prototype）添加东西。让我们给 `String` 添加一个方法用来返回逆序的字符串：

```
var s = "Simon";
s.reversed(); // TypeError on line 1: s.reversed is not a function

String.prototype.reversed = function() {
    var r = "";
    for (var i = this.length - 1; i >= 0; i--) {
        r += this[i];
    }
    return r;
}
s.reversed(); // nomiS
```

定义新方法也可以在字符串字面量上用（string literal）。

```
"This can now be reversed".reversed(); // desrever eb won nac sihT
```

正如我前面提到的，原型组成链的一部分。那条链的根节点是 `Object.prototype`，它包括 `toString()` 方法——将对象转换成字符串时调用的方法。这对于调试我们的 `Person` 对象很有用：

```
var s = new Person("Simon", "Willison");
s; // [object Object]

Person.prototype.toString = function() {
    return '<Person: ' + this.fullName() + '>';
}
s.toString(); // <Person: Simon Willison>
```

你是否还记得之前我们说的 `avg.apply()` 中的第一个参数 `null`？现在我们可以回头看看这个东西了。`apply()` 的第一个参数应该是一个被当作 `this` 来看待的对象。下面是一个 `new` 方法的简单实现：

```
function trivialNew(constructor, ...args) {
    var o = {}; // 创建一个对象
    constructor.apply(o, args);
    return o;
}
```

这并不是 new 的完整实现，因为它没有创建原型（prototype）链。想举例说明 new 的实现有些困难，因为你不会经常用到这个，但是适当了解一下还是很有用的。在这一小段代码里，...args（包括省略号）叫作剩余参数（rest arguments）。如名所示，这个东西包含了剩下的参数。

因此，调用

```
var bill = trivialNew(Person, "William", "Orange");
```

可认为和调用如下语句是等效的

```
var bill = new Person("William", "Orange");
```

apply() 有一个姐妹函数，名叫 call，它也可以允许你设置 this，但它带有一个扩展的参数列表而不是一个数组。

```
function lastNameCaps() {
    return this.last.toUpperCase();
}
var s = new Person("Simon", "Willison");
lastNameCaps.call(s);
// 和以下方式等价
s.lastNameCaps = lastNameCaps;
s.lastNameCaps();
```

内部函数

JavaScript 允许在一个函数内部定义函数，这一点我们在之前的 makePerson() 例子中也见过。关于 JavaScript 中的嵌套函数，一个很重要的细节是，它们可以访问父函数作用域中的变量：

```
function parentFunc() {
  var a = 1;

  function nestedFunc() {
    var b = 4; // parentFunc 无法访问 b
    return a + b;
  }
  return nestedFunc(); // 5
}
```

如果某个函数依赖于其他的一两个函数，而这一两个函数对你其余的代码没有用处，你可以将它们嵌套在会被调用的那个函数内部，这样做可以减少全局作用域下的函数的数量，这有利于编写易于维护的代码。

这也是一个减少使用全局变量的好方法。当编写复杂代码时，程序员往往试图使用全局变量，将值共享给多个函数，但这样做会使代码很难维护。内部函数可以共享父函数的变量，所以你可以使用这个特性把一些函数捆绑在一起，这样可以有效地防止“污染”你的全局命名空间——你可以称它为“局部全局（local global）”。虽然这种方法应该谨慎使用，但它确实很有用，应该掌握。

闭包

闭包是 JavaScript 中最强大的抽象概念之一——但它也是最容易造成困惑的。它究竟是做什么的呢?

```
function makeAdder(a) {
  return function(b) {
    return a + b;
  }
}
var add5 = makeAdder(5);
var add20 = makeAdder(20);
add5(6); // ?
add20(7); // ?
```

`makeAdder` 这个名字本身，便应该能说明函数是用来做什么的：它会用一个参数来创建一个新的“adder”函数，再用另一个参数来调用被创建的函数时，`makeAdder` 会将一前一后两个参数相加。

从被创建的函数的视角来看的话，这两个参数的来源问题会更显而易见：新函数自带一个参数——在新函数被创建时，便钦定、钦点了前一个参数（如上方代码中的 a、5 和 20，参考 `makeAdder` 的结构，它应当位于新函数外部）；新函数被调用时，又接收了后一个参数（如上方代码中的 b、6 和 7，位于新函数内部）。最终，新函数被调用的时候，前一个参数便会和由外层函数传入的后一个参数相加。

这里发生的事情和前面介绍过的内嵌函数十分相似：一个函数被定义在了另外一个函数的内部，内部函数可以访问外部函数的变量。唯一的不同是，外部函数已经返回了，那么常识告诉我们局部变量“应该”不再存在。但是它们却仍然存在——否则 `adder` 函数将不能工作。也就是说，这里存在 `makeAdder` 的局部变量的两个不同的“副本”——一个是 `a` 等于 5，另一个是 `a` 等于 20。那些函数的运行结果就如下所示：

```
x(6); // 返回 11
y(7); // 返回 27
```

下面来说说，到底发生了什么了不得的事情。每当 JavaScript 执行一个函数时，都会创建一个作用域对象（scope object），用来保存在这个函数中创建的局部变量。它使用一切被传入函数的变量进行初始化（初始化后，它包含一切被传入函数的变量）。这与那些保存的所有全局变量和函数的全局对象（global object）相类似，但仍有一些很重要的区别：第一，每次函数被执行的时候，就会创建一个新的，特定的作用域对象；第二，与全局对象（如浏览器的 `window` 对象）不同的是，你不能从 JavaScript 代码中直接访问作用域对象，也没有 可以遍历当前作用域对象中的属性 的方法。

所以，当调用 `makeAdder` 时，解释器创建了一个作用域对象，它带有一个属性：`a`，这个属性被当作参数传入 `makeAdder` 函数。然后 `makeAdder` 返回一个新创建的函数（暂记为 `adder`）。通常，JavaScript 的垃圾回收器会在这时回收 `makeAdder` 创建的作用域对象（暂记为 `b`），但是，`makeAdder` 的返回值，新函数 `adder`，拥有一个指向作用域对象 `b` 的引用。最终，作用域对象 `b` 不会被垃圾回收器回收，直到没有任何引用指向新函数 `adder`。

作用域对象组成了一个名为作用域链（scope chain）的（调用）链。它和 JavaScript 的对象系统使用的原型（prototype）链相类似。

一个**闭包**，就是 一个函数 与其 被创建时所带有的作用域对象 的组合。闭包允许你保存状态——所以，它们可以用来代替对象。　这个 StackOverflow 帖子里有一些关于闭包的详细介绍。