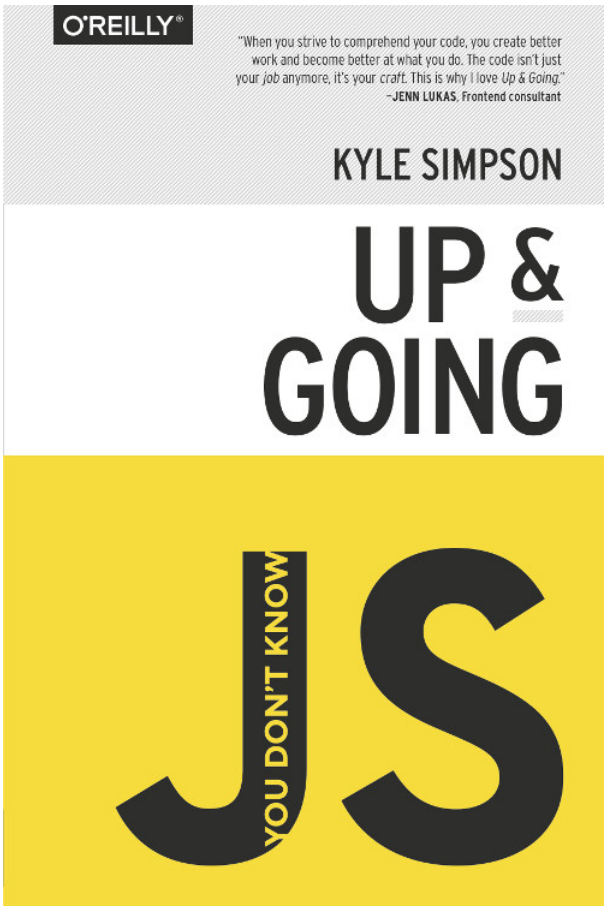


从 O'Reilly 购买数字/印刷版

目录

序

你学的最后一个新东西是什么?

也许是一门外语，比如意大利语或德语。或者可能是一种图像编辑器，比如 Photoshop。或者是一种烹饪技术，木工活，日常锻炼。我想让你回忆一下你最终学会它时的感觉：醍醐灌顶

的时刻。当事情从模糊不清变得豁然开朗，正如你掌握了如何使用台锯，或者理解了法语中雄性名词和雌性名词的区别。那种感觉怎么样？非常美妙，对吧？

现在我想让你再多向前回忆一些，找到你学会新技能之前的那一刻。它感觉如何？可能有点儿吓人，也可能有点儿沮丧，是吧？在某一个时刻，我们都还不知道我们现在知道的事情，而这完全没问题；我们是从某处开始的。学习新的东西是一次激动人心的冒险，特被是当你想高效地学习它时。

我教授过许多面向初学者的编程课程。上我课的学生们经常试着通过阅读博客或者拷贝粘贴代码来自学 HTML 或 JavaScript 这样的东西，但是他们都没能真正掌握能够使他们编写出自己渴望的结果的技能。而且，因为他们没有真正把握关于编程的特定问题的内在和外在，他们不能编写强大的代码或调试自己的程序，因为他们没有真正地理解发生的事情。

我总是相信教授我的课程的正确方法，意味着我教授 Web 标准，语义标记，良好注释的代码，和其他的最佳实践。我使用一种彻底的方式讲解问题来阐明如何做与为何做，而非通过复制粘贴来倒腾代码。当你努力理解你的代码时，你就在创造更好的成果，并在编程上变得更加纯熟。代码不再仅仅是你的 *工作*，而是你的 *作品*。这就是为什么我喜爱 *入门与进阶*。Kyle 通过深入讲解语法和术语给我们带来了一个对 JavaScript 的全面介绍。这本书不是浅尝辄止，而是让我们真正地理解我们将要编写的东西。

能够在你的网站中复制 JQuery 代码段是不够的，就像在 Photoshop 中仅仅学习如何打开，关闭和保存一个文档是不够的一样。确实，只要我学会了一些关于编程的基本我就可以制造并分享一些我的设计。但是没有合理地了解这些工具和它们背后的机制，我又如何定义一个网格，或者建造一个合理的类型系统，或者为 Web 优化图像呢？JavaScript 也一样。不知道循环如何工作，或者如何定义变量，或者作用域是什么，我们将不能写出最好的代码。我们不想安于这种次优的状态 —— 这毕竟是我们的作品。

你对 JavaScript 探索得越多，它就变得越清晰。闭包，对象，和方法这样的词现在可能看起来与你还有些距离，但是这本书将会帮你搞清楚这些术语。我希望你在开始阅读这本书时保持学会东西之前与之后的那两种感觉。它看起来可能有些令人望而却步，但是你已经拿起了这本书，你开启了一个了不起的旅程来磨练自己的知识。*入门与进阶*是我们理解编程之路的开端。享受醍醐灌顶的时刻吧！

Jenn Lukas jennlukas.com, [@jennlukas](https://twitter.com/jennlukas) 前端顾问

第一章：进入编程

欢迎来到 *你不懂 JS*（*YDKJS*）系列。

*入门与进阶*是一个对几种编程基本概念的介绍 —— 当然我们是特别倾向于 JavaScript（经常略称为 JS）的 —— 以及如何看待与理解本系列的其他书目。特别是如果你刚刚接触编程和/或 JavaScript，这本书将简要地探索你需要什么来 *入门与进阶*。

这本书从很高的角度来解释编程的基本原则开始。它基本上假定你是在没有或很少的编程经验的情况下开始阅读 *YDKJS* 的，而且你期待这些书可以透过 JavaScript 的镜头帮助你开启一条理解编程的道路。

第一章应当作为一个快速的概览来阅读，它讲述为了 进入编程你将想要多加学习和实践的东西。有许多其他精彩的编程介绍资源可以帮你在这个话题上走得更远，而且我鼓励你学习它们来作为这一章的补充。

一旦你对一般的编程基础感到适应了，第二章将指引你熟悉 JavaScript 风格的编程。第二章介绍了 JavaScript 是什么，但是同样的，它不是一个全面的指引 —— 那是其他 *YDKJS* 书目的任务！

如果你已经相当熟悉 JavaScript，那么就首先看一下第三章作为 *YDKJS* 内容的简要一瞥，然后一头扎进去吧！

代码

让我们从头开始。

一个程序，经常被称为 源代码或者只是 代码，是一组告诉计算机要执行什么任务的特殊指令。代码通常保存在文本文件中，虽然你也可以使用 JavaScript 在一个浏览器的开发者控制台中直接键入代码 —— 我们一会儿就会讲解。

合法的格式与指令的组合规则被称为一种 计算机语言，有时被称作它的 语法，这和英语教你如何拼写单词，和如何使用单词与标点创建合法的句子差不多是相同的。

语句

在一门计算机语言中，一组单词，数字，和执行一种具体任务的操作符构成了一个 语句。在 JavaScript 中，一个语句可能看起来像下面这样：

```
a = b * 2;
```

字符 `a` 和 `b` 被称为 变量（参见“变量”），它们就像简单的盒子，你可以把任何东西存储在其中。在程序中，变量持有将被程序使用的值（比如数字 `42`）。可以认为它们就是值本身的标志占位符。

相比之下，`2` 本身只是一个值，称为一个 字面值，因为它没有被存入一个变量，是独立的。

字符 `=` 和 `*` 是 操作符（见“操作符”）—— 它们使用值和变量实施动作，比如赋值和数学乘法。

在 JavaScript 中大多数语句都以末尾的分号（`;`）结束。

语句 `a = b * 2;` 告诉计算机，大致上，去取得当前存储在变量 `b` 中的值，将这个值乘以 `2`，然后将结果存回到另一个我们称为 `a` 变量里面。

程序只是许多这样的语句的集合，它们一起描述为了执行你的程序的意图所要采取的所有步骤。

表达式

语句是由一个或多个 表达式组成的。一个表达式是一个引用，指向变量或值，或者一组用操作符组合的变量和值。

例如：

```
a = b * 2;
```

这个语句中有四个表达式：

- `2` 是一个 *字面量表达式*
- `b` 是一个 *变量表达式*，它意味着取出它的当前值
- `b * 2` 是一个 *算数表达式*，它意味着执行乘法
- `a = b * 2` 是一个 *赋值表达式*，它意味着将表达式 `b * 2` 的结果赋值给变量 `a`（稍后有更多关于赋值的内容）

一个独立的普通表达式也被称为一个 *表达式语句*，比如下面的：

```
b * 2;
```

这种风格的表达式语句不是很常见也没什么用，因为一般来说它不会对程序的运行有任何影响 —— 它将取得 `b` 的值并乘以 `2`，但是之后不会对结果做任何事情。

一种更常见的表达式语句是 *调用表达式*语句（见 “函数”），因为整个语句本身是一个函数调用表达式：

```
alert( a );
```

执行一个程序

这些程序语句的集合如何告诉计算机要做什么？这个程序需要被 *执行*，也称为 *运行这个程序*。

在开发者们阅读与编写时，像 `a = b * 2` 这样的语句很有帮助，但是它实际上不是计算机可以直接理解的形式。所以一个计算机上的特殊工具（不是一个 *解释器*就是一个 *编译器*）被用于将你编写的代码翻译为计算机可以理解的命令。

对于某些计算机语言，这种命令的翻译经常是在每次程序运行时从上向下，一行接一行完成的，这通常成为代码的 *解释*。

对于另一些语言，这种翻译是提前完成的，成为代码的 *编译*，所以当程序稍后 *运行*时，实际上运行的东西已经是编译好，随时可以运行的计算机指令了。

JavaScript 通常被断言为是 *解释型*的，因为你的 JavaScript 源代码在它每次运行时都被处理。但这并不是完全准确的。JavaScript 引擎实际上在即时地 *编译*程序然后立即运行编译好的代码。

注意：更多关于 JavaScript 编译的信息，参见本系列的 *作用域与闭包*的前两章。

亲自尝试

这一章将用简单的代码段来介绍每一个编程概念，它们都是用 JavaScript 写的（当然!）。

有一件事情怎么强调都不过分：在你通读本章时 —— 而且你可能需要花时间读好几遍 —— 你应当通过自己编写代码来实践这些概念中的每一个。最简单的方法就是打开你手边的浏览器（Firefox，Chrome，IE，等等）的开发者工具控制台。

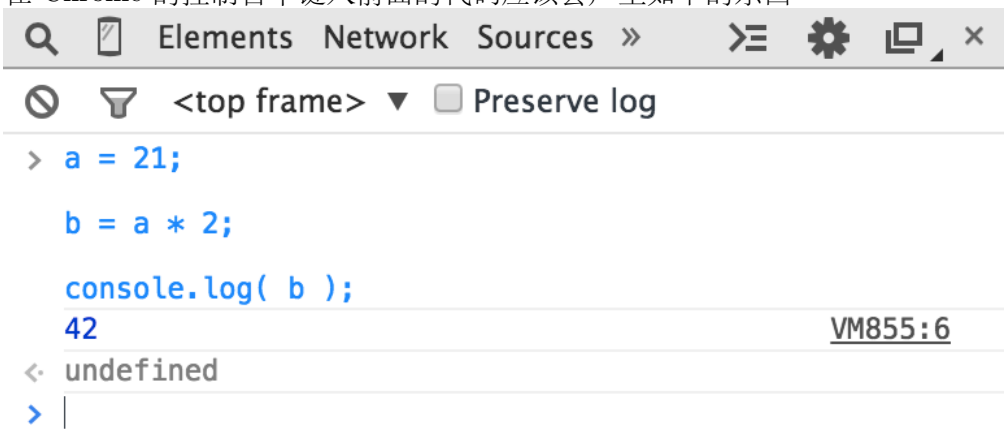
提示：一般来说，你可以使用快捷键或者菜单选项来启动开发者控制台。更多关于启动和使用你最喜欢的浏览器的控制台的细节，参见“精通开发者工具控制台”(http://blog.teamtreehouse.com/masterin developer-tools-console)。要在控制台中一次键入多行，可以使用 `<shift> + <enter>` 来移动到下一行。一旦你敲击 `<enter>`，控制台将运行你刚刚键入的任何东西。

让我们熟悉一下在控制台中运行代码的过程。首先，我建议你在浏览器中打开一个新的标签页。我喜欢在地址栏中键入 `about:blank` 来这么做。然后，确认你的开发者控制台是打开的，就像我们刚刚提到的那样。

现在，键入如下代码看看它是怎么运行的：

```
a = 21;

b = a * 2;

console.log( b );
```

在 Chrome 的控制台中键入前面的代码应该会产生如下的东西：



继续，试试吧。学习编程的最佳方式就是开始编码!

输出

在前一个代码段中，我们使用了 `console.log(..)`。让我们简单地看看这一行代码在做什么。

你也许已经猜到了，它正是我们如何在开发者控制台中打印文本（也就是向用户 输出）的方法。这个语句有两个性质，我们应当解释一下。

首先，`log( b )` 部分被称为一个函数调用（见“函数”）。这里发生的事情是，我们将变量 `b` 交给这个函数，它向变量 `b` 要来它的值，并在控制台中打印。

第二，`console.` 部分是一个对象引用，这个对象就是找到 `log(..)` 函数的地方。我们会在第二章中详细讲解对象和它们的属性。

另一种创建你可以看到的输出的方式是运行 `alert(..)` 语句。例如：

```
alert( b );
```

如果你运行它，你会注意到它不会打印输出到控制台，而是显示一个内容为变量 `b` 的“OK”弹出框。但是，一般来说与使用 `alert(..)` 相比，使用 `console.log(..)` 会使学习编码和在控制台运行你的程序更简单一些，因为你可以一次输出许多值，而不必干扰浏览器的界面。

在这本书中，我们将使用 `console.log(..)` 来输出。

输入

虽然我们在讨论输出，你也许还想知道 输入（例如，从用户那里获得信息）。

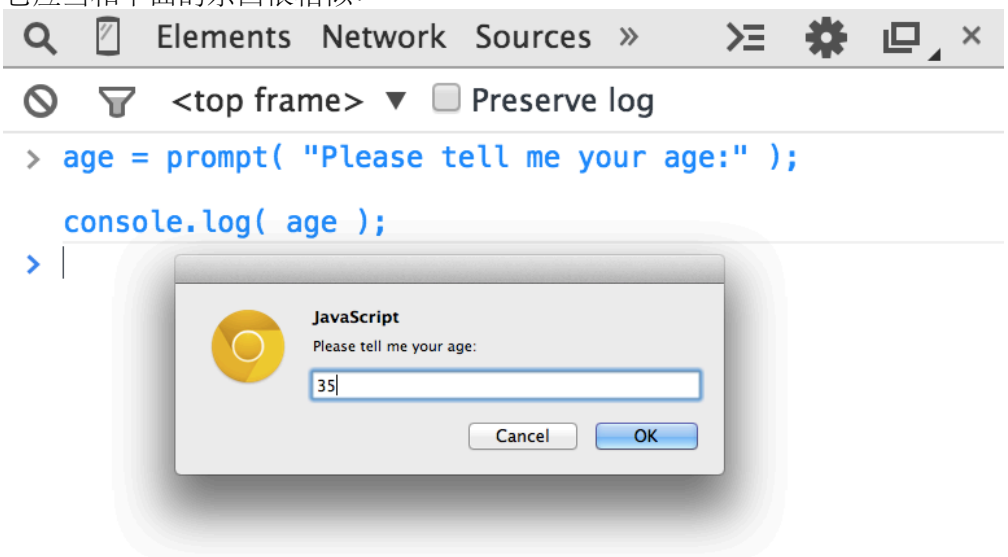
对于 HTML 网页来说，输入发生的最常见的方式是向用户显示一个他们可以键入的 form 元素，然后使用 JS 将这些值读入你程序的变量中。

但是为了单纯的学习和展示的目的 —— 也就是你在这本书中将通篇看到的 —— 有一个获取输入的更简单的方法。使用 `prompt(..)` 函数：

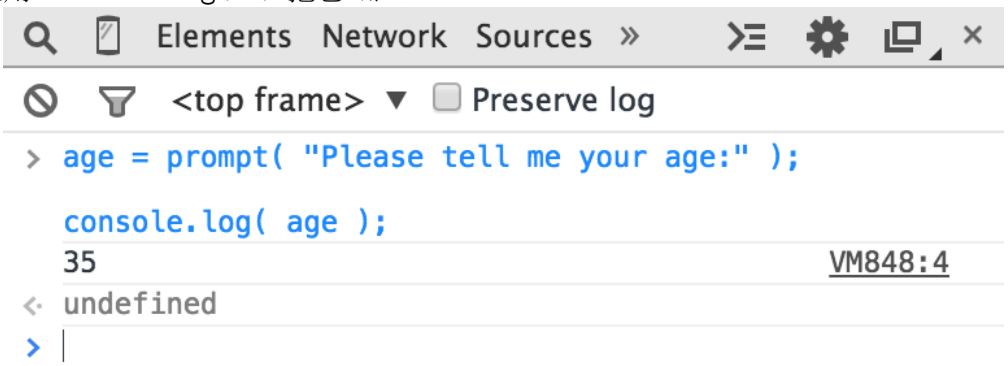
```
age = prompt( "Please tell me your age:" );

console.log( age );
```

正如你可能已经猜到的，你传递给 `prompt(..)` 的消息 —— 在这个例子中，`"Please tell me your age:"` —— 被打印在弹出框中。

它应当和下面的东西很相似：



一旦你点击 “OK” 提交输入的文本，你将会看到你输入的值被存储在变量 `age` 中，然后我们使用 `console.log(..)` 把它 *输出*：



为了让我们在学习基本编程概念时使事情保持简单，本书中的例子不要求输入。但是现在你已经看到了如何使用 `prompt(..)`，如果你想挑战一下自己，你可以试着在探索这些例子时使用输入。

操作符

操作符是我们如何在变量和值上实施操作的方式。我们已经见到了两种 JavaScript 操作符，`=` 和 `*`。

`*` 操作符实施数学乘法。够简单的，对吧？

`=` 操作符用于 *赋值* —— 我们首先计算 `=` *右手边*的值（源值）然后将它放进我们在 *左手边* 指定的变量中（目标变量）。

警告：对于指定赋值，这看起来像是一种奇怪的倒置。与 `a = 42` 不同，一些人喜欢把顺序反转过来，于是源值在左而目标变量在右，就像 `42 -> a`（这不是合法的 JavaScript！）。不幸的是，`a = 42` 顺序的形式，和与其相似的变种，在现代编程语言中是十分流行的。如果它让你觉得不自然，那么就花些时间在脑中演练这个顺序并习惯它。

考虑如下代码：

```
a = 2;
b = a + 1;
```

这里，我们将值 `2` 赋值给变量 `a`。然后，我们取得变量 `a` 的值（还是 `2`），把它加 `1` 得到值 `3`，然后将这个值存储到变量 `b` 中。

虽然在技术上说 `var` 不是一个操作符，但是你将在每一个程序中都需要这个关键字，因为它是你 *声明*（也就是 *创建*）变量（见 “变量”）的主要方式。

你应当总是在使用变量前用名称声明它。但是对于每个 *作用域*（见 “作用域”）你只需要声明变量一次；它可以根据需要使用任意多次。例如：

```
var a = 20;
```

```
a = a + 1;
a = a * 2;

console.log( a );   // 42
```

这里是一些在 JavaScript 中最常见的操作符：

- 赋值：比如 `a = 2` 中的 `=`。
- 数学：`+`（加法），`-`（减法），`*`（乘法），和`/`（除法），比如 `a * 3`。
- 复合赋值：`+=`，`-=`，`*=`，和`/=` 都是复合操作符，它们组合了数学操作和赋值，比如 `a += 2`（与 `a = a + 2` 相同）。
- 递增/递减：`++`（递增），`--`（递减），比如 `a++`（和 `a = a + 1` 很相似）。
- 对象属性访问：比如 `console.log()` 的`.`。

  对象是一种值，它可以在被称为属性的，被具体命名的位置上持有其他的值。`obj.a` 意味着一个称为 `obj` 的对象值有一个名为 `a` 的属性。属性可以用 `obj["a"]` 这种替代的方式访问。参见第二章。
- 等价性：`==`（宽松等价），`===`（严格等价），`!=`（宽松不等价），`!==`（严格不等价），比如 `a == b`。

  参见“值与类型”和第二章。
- 比较：`<`（小于），`>`（大于），`<=`（小于或宽松等价），`>=`（大于或宽松等价），比如 `a <= b`。

  参见“值与类型”和第二章。
- 逻辑：`&&`（与），`||`（或），比如 `a || b` 它选择 `a` 或 `b` 中的一个。

  这些操作符用于表达复合的条件（见“条件”），比如如果 `a` 或者 `b` 成立。

注意：更多细节，以及在此没有提到的其他操作符，可以参见 Mozilla 开发者网络（MDN）的“表达式与操作符”(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Oper

值与类型

如果你问一个手机店的店员一种特定手机的价格，而他们说“九十九块九毛九”（即，$99.99），他们给了你一个实际的美元数字来表示你需要花多少钱才能买到它。如果你想两部这种手机，你可以很容易地心算这个值的两倍来得到你需要花费的 $199.98。

如果同一个店员拿起另一部相似的手机说它是“免费的”（也许在用手比划引号），那么他们就不是在给你一个数字，而是你的花费（$0.00）的另一种表达形式 —— “免费”这个词。

当你稍后问到这个手机是否带充电器时，回答可能仅仅是“是”或者“不”。

以同样的方式，当你在程序中表达一个值时，你根据你打算对这些值做什么来选择不同的表达形式。

在编程术语中值的这些不同的表达形式称为 *类型*。JavaScript 中对这些所谓的 *基本类型*值都有内建的类型：

- 当你需要做数学计算时，你需要一个 `number`。
- 当你需要在屏幕上打印一个值时，你需要一个 `string`（一个或多个字符，单词，句子）。
- 当你需要在你的程序中做决定时，你需要一个 `boolean`（`true` 或 `false`）。

在源代码中直接包含的值称为 *字面量*。`string` 字面量被双引号`"..."` 或单引号（`'...'`）包围 —— 唯一的区别是风格上的偏好。`number` 和 `boolean` 字面量用它们本身来表示（即，`42`，`true`，等等）。

考虑如下代码：

```
"I am a string";
'I am also a string';

42;

true;
false;
```

在 `string`/`number`/`boolean` 值的类型以外，编程语言通常会提供 *数组*，*对象*，*函数*等更多的类型。我们会在本章和下一章中讲解更多关于值和类型的内容。

类型间转换

如果你有一个 `number` 但需要将它打印在屏幕上，那么你就需要将这个值转换为一个 `string`，在 JavaScript 中这种转换称为 “强制转换”。类似地，如果某些人在一个电商网页的 form 中输入一系列数字，那么它是一个 `string`，但是如果你需要使用这个值去做数学运算，那么你就需要将它 *强制转换*为一个 `number`。

为了在 *类型*之间强制转换，JavaScript 提供了几种不同的工具。例如：

```
var a = "42";
var b = Number( a );

console.log( a );   // "42"
console.log( b );   // 42
```

使用上面展示的 `Number(..)`（一个内建函数）是一种从任意其他类型到 `number` 类型的 *明确的*强制转换。这应当是相当直白的。

但是一个具有争议的话题是，当你试着比较两个还不是相同类型的值时发生的事情，它需要 *隐含的*强制转换。

当比较字符串`"99.99"` 和数字 `99.99` 时，大多数人同意它们是等价的。但是他们不完全相同，不是吗？它们是相同的值的两种不同表现形式，两个不同的 *类型*。你可以说它们是“宽松地等价”的，不是吗？

为了在这些常见情况下帮助你，JavaScript 有时会启动 *隐含的*强制转换来把值转换为匹配的类型。

所以如果你使用 `==` 宽松等价操作符来进行`"99.99" == 99.99` 比较，JavaScript 会将左手边的`"99.99"` 转换为它的 `number` 等价物 `99.99`。所以比较就变成了 `99.99 == 99.99`，这当然是成立的。

虽然隐含强制转换是为了帮助你而设计，但是它也可能把你搞糊涂，如果你没有花时间去学习控制它行为的规则。大多数开发者从没有这么做，所以常见的感觉是隐含的强制转换是令人困惑的，并且会产生意外的 bug 危害程序，因此应当避免使用。有时它甚至被称为这种语言中的设计缺陷。

然而，隐含强制转换是一种 *可以被学习*的机制，而且是一种 *应当*被所有想要认真对待 JavaScript 编程的人学习的机制。一旦你学习了这些规则，它不仅是消除了困惑，而且它实际上是你的程序变得更好！这种努力是值得的。

注意：关于强制转换的更多信息，参见本书第二章和本系列 *类型与文法*的第四章。

代码注释

手机店店员可能会写下一些笔记，记下新出的手机的特性或者他们公司推出的新套餐。这些笔记仅仅是给店员使用的 —— 他们不是给顾客读的。不管怎样，通过记录下为什么和如何告诉顾客他应当说的东西，这些笔记帮助店员更好的工作。

关于编写代码你要学的最重要的课程之一，就是它不仅仅是写给计算机的。代码的每一个字节都和写给编译器一样，也是写给开发者的。

你的计算机只关心机器码，一系列源自 *编译*的 0 和 1。你几乎可以写出无限多种可以产生相同 0 和 1 序列的代码。所以你对如何编写程序作出的决定很重要 —— 不仅是对你，也对你的团队中的其他成员，甚至是你未来的自己。

你不仅应当努力去编写可以正确工作的程序，而且应当努力编写检视起来有道理的程序。你可以通过给变量（见“变量”）和函数（见“函数”）起一个好名字在这条路上走很远。

但另外一个重要的部分是代码注释。它们纯粹是为了向人类解释一些事情而在你的程序中插入的一点儿文本。解释器/编译器将总是忽略这些注释。

关于什么是良好注释的代码有许多意见；我们不能真正地定义绝对统一的规则。但是一些意见和指导是十分有用的：

- 没有注释的代码是次优的。
- 过多的注释（比如，每行都有注释）可能是代码编写的很烂的标志。

- 注释应当解释 为什么，而不是 是什么。它们可以选择性地解释 如何做，如果代码特别令人困惑的话。

在 JavaScript 中，有两种可能的注释类型：单行注释和多行注释
考虑如下代码：

```
// 这是一个单行注释

/* 而这是
      一个多行
          注释。
                */
```

如果你想在一个语句的正上方，或者甚至是在行的末尾加一个注释，//单行注释是很合适的。这一行上//之后的所有东西都将被视为注释（因此被编译器忽略），一直到行的末尾。在单行注释内部可以出现的内容没有限制。
考虑：

```
var a = 42;      // 生命的意义是 42
```

如果你想在注释中用好几行来解释一些事情，/* .. */多行注释就很合适。
这是多行注释的一个常见用法：

```
/* 使用下面的值是因为
   它回答了
   全宇宙中所有的问题。 */
var a = 42;
```

它还可以出现在一行中的任意位置，甚至是一行的中间，因为 */终结了它。例如：

```
var a = /* 随机值 */ 42;

console.log( a );   // 42
```

在多行注释中唯一不能出现的就是 */，因为这将干扰注释的结尾。
你绝对会希望通过养成注释代码的习惯来开始学习编程。在本书剩余的部分中，你将看到我使用注释来解释事情，请也在你自己的实践中这么做。相信我，所有阅读你的代码的人都会感谢你！

变量

大多数有用的程序都需要在程序运行整个过程中，追踪由于你的程序所意图的任务被调用的底层不同的操作而发生的值的变化。

要这样做的最简单的方法是将一个值赋予一个符号容器，称为一个 *变量* —— 因为在这个容器中的值可以根据需要不时 *变化*而得名。

在某些编程语言中，你可以声明一个变量（容器）来持有特定类型的值，比如 `number` 或 `string`。因为防止了意外的类型转换，*静态类型*，也被称为 *类型强制*，通常被认为是对程序正确性有好处的。

另一些语言在值上强调类型而非在变量上。*弱类型*，也被称为 *动态类型*，允许变量在任意时刻持有任意类型的值。因为它允许一个变量在程序逻辑流程中代表一个值，而不论这个值在任意给定的时刻是什么类型，所以它被认为是对程序灵活性有好处的。

JavaScript 使用的是后者，*动态类型*，这意味着变量可以持有任意 *类型*的值而没有任何 *类型*强制约束。

正如我们刚才提到的，我们使用 `var` 语句来声明一个变量 —— 注意在这种声明中没有其他的 *类型*信息。考虑这段简单的代码：

```
var amount = 99.99;

amount = amount * 2;

console.log( amount );      // 199.98

// 将 `amount` 转换为一个字符串，
// 并在开头加一个 "$"
amount = "$" + String( amount );

console.log( amount );      // "$199.98"
```

变量 `amount` 开始时持有数字 `99.99`，然后持有 `amount * 2` 的 `number` 结果，也就是 `199.98`。

第一个 `console.log(..)` 命令不得不 *隐含地*将这个 `number` 值强制转换为一个 `string` 才能够打印出来。

然后语句 `amount = "$" + String(amount)` *明确地*将值 `199.98` 强制转换为一个 `string` 并且在开头加入一个`"$"` 字符。这时，`amount` 现在就持有这个 `string` 值 `$199.98`，所以第二个 `console.log(..)` 语句无需强制转换就可以把它打印出来。

JavaScript 开发者将会注意到为值 `99.99`，`199.98`，和`"$199.98"` 都使用变量 `amount` 的灵活性。静态类型的拥护者们将偏好于使用一个分离的变量，比如 `amountStr` 来持有这个值最后的`"$199.98"` 表达形式，因为它是一个不同的类型。

不管哪种方式，你将会注意到 amount 持有一个在程序运行过程中不断变化的值，这展示了变量的主要目地：管理程序 状态。

换句话说，在你程序运行的过程中 状态追踪着值的改变。

变量的另一种常见用法是将值的设定集中化。当你为一个在程序中通篇不打算改变的值声明了一个变量时，它更一般地被称为 常量。

你经常会在程序的顶部声明这些 常量，这样提供了一种方便：如果你需要改变一个值时你可以到唯一的地方去寻找。根据惯例，用做常量的 JavaScript 变量通常是大写的，在多个单词之间使用下划线 _ 连接。

这里是一个呆萌的例子：

```
var TAX_RATE = 0.08;    // 8% sales tax

var amount = 99.99;

amount = amount * 2;

amount = amount + (amount * TAX_RATE);

console.log( amount );              // 215.9784
console.log( amount.toFixed( 2 ) ); // "215.98"
```

注意： console.log(..) 是一个函数 log(..) 作为一个在值 console 上的对象属性被访问，与此类似，这里的 toFixed(..) 是一个可以在值 number 上被访问的函数。JavaScript number 不会被自动地格式化为美元 —— 引擎不知道你的意图，而且也没有通货类型。toFixed(..) 让我们指明四舍五入到小数点后多少位，而且它如我们需要的那样产生一个 string。

变量 TAX_RATE 只是因为惯例才是一个 常量 —— 在这个程序中没有什么特殊的东西可以防止它被改变。但是如果这座城市将它的消费税增至 9%，我们仍然可以很容地通过在一个地方将 TAX_RATE 被赋予的值改为 0.09 来更新我们的程序，而不是在程序通篇中寻找许多值 0.08 出现的地方然后更新它们全部。

在写作本书时，最新版本的 JavaScript（通常称为 “ES6”）引入了一个声明常量的新方法，用 const 代替 var：

```
// 在 ES6 中：
const TAX_RATE = 0.08;

var amount = 99.99;

// ..
```

常量就像带有不变的值的变量一样有用，常量还防止在初始设置之后的某些地方意外地改变它的值。如果你试着在第一个声明之后给 `TAX_RATE` 赋予一个不同的值，你的程序将会拒绝这个改变（而且在 Strict 模式下，会产生一个错误 —— 见第二章的 “Strict 模式”）。

顺带一提，这种防止编程错误的 “保护” 与静态类型的类型强制很类似，所以你可以看到为什么在其他语言中的静态类型很吸引人。

注意：更多关于如何在你程序的变量中使用不同的值，参见本系列的 *类型与文法*。

块儿

在你买你的新手机时，手机店店员必须走过一系列步骤才能完成结算。

相似地，在代码中我们经常需要将一系列语句一起分为一组，这就是我们常说的 *块儿*。在 JavaScript 中，一个块儿被定义为包围在一个大括号 `{ .. }` 中的一个或多个语句。考虑如下代码：

```
var amount = 99.99;

// 一个普通的块儿
{
    amount = amount * 2;
    console.log( amount );  // 199.98
}
```

这种独立的 `{ .. }` 块儿是合法的，但是在 JS 程序中并不常见。一般来说，块儿是添附在一些其他的控制语句后面的，比如一个 `if` 语句（见 “条件”）或者一个循环（见 “循环”）。例如：

```
var amount = 99.99;

// 数值够大吗?
if (amount > 10) {          // <-- 添附在 `if`上的块儿
    amount = amount * 2;
    console.log( amount );  // 199.98
}
```

我们将在下一节讲解 `if` 语句，但是如你所见，`{ .. }` 块儿带着它的两个语句被添附在 `if (amount > 10)` 后面；块儿中的语句将会仅在条件成立时被处理。

注意：与其他大多数语句不同（比如 `console.log(amount);`），一个块儿语句不需要分号（`;`）来终结它。

条件

“你想来一个额外的屏幕贴膜吗？只要 $9.99。”热心的手机店店员请你做个决定。而你也许需要首先咨询一下钱包或银行帐号的 *状态*才能回答这个问题。但很明显，这只是一个简单的“是与否”的问题。

在我们的程序中有好几种方式可以表达 *条件*（也就是决定）。

最常见的一个就是 `if` 语句。实质上，你在说，“*如果*这个条件成立，做后面的……”。例如：

```
var bank_balance = 302.13;
var amount = 99.99;

if (amount < bank_balance) {
    console.log( "I want to buy this phone!" );
}
```

`if` 语句在括号 `( )` 之间需要一个表达式，它不是被视作 `true` 就是被视作 `false`。在这个程序中，我们提供了表达式 `amount < bank_balance`，它确实会根据变量 `bank_balance` 中的值被求值为 `true` 或 `false`。

如果条件不成立，你甚至可以提供一个另外的选择，称为 `else` 子句。考虑下面的代码：

```
const ACCESSORY_PRICE = 9.99;

var bank_balance = 302.13;
var amount = 99.99;

amount = amount * 2;

// 我们买得起配件吗?
if ( amount < bank_balance ) {
    console.log( "I'll take the accessory!" );
    amount = amount + ACCESSORY_PRICE;
}
// 否则:
else {
    console.log( "No, thanks." );
}
```

在这里，如果 `amount < bank_balance` 是 `true`，我们将打印出`"I'll take the accessory!"` 并在我们的变量 `amount` 上加 `9.99`。否则，`else` 子句说我们将礼貌地回应`"No, thanks."`，并保持 `amount` 不变。

正如我们在早先的“值与类型”中讨论的，一个还不是所期望类型的值经常会被强制转换为那种类型。`if` 语句期待一个 `boolean`，但如果你传给它某些还不是 `boolean` 的东西，强制转换就会发生。

JavaScript 定义了一组特定的被认为是“falsy”的值，因为在强制转换为 `boolean` 时，它们将变为 `false` —— 这些值包括 `0` 和`""`。任何不再这个 `falsy` 列表中的值都自动是“truthy” —— 当强制转换为 `boolean` 时它们变为 `true`。truthy 值包括 `99.99` 和`"free"` 这样的东西。更多信息参见第二章的“Truthy 与 Falsy”。

除了 `if` 条件还以其他形式存在。例如，`switch` 语句可以被用作一系列 `if..else` 语句的缩写（见第二章）。循环（见“循环”）使用一个 条件来决定循环是否应当继续或停止。

注意：关于在 条件的测试表达式中可能发生的隐含强制转换的更深层的信息，参见本系列的 类型与文法的第四章。

循环

在繁忙的时候，有一张排队单，上面记载着需要和手机店店员谈话的顾客。虽然排队单上还有许多人，但是她只需要持续服务下一位顾客就好了。

重复一组动作直到特定的条件失败 —— 换句话说，仅在条件成立时重复 —— 就是程序循环的工作；循环可以有不同的形式，但是它们都符合这种基本行为。

一个循环包含测试条件和一个块儿（通常是 `{ .. }`）。每次循环块儿执行，都称为一次 迭代。

例如，`while` 循环和 `do..while` 循环形式就说明了这种概念 —— 重复一块儿语句直到一个条件不再求值得 `true`：

```
while (numOfCustomers > 0) {
    console.log( "How may I help you?" );

    // 服务顾客.....

    numOfCustomers = numOfCustomers - 1;
}

// 与

do {
    console.log( "How may I help you?" );

    // 服务顾客.....
```

```
    numOfCustomers = numOfCustomers - 1;
} while (numOfCustomers > 0);
```

这些循环之间唯一的实际区别是，条件是在第一次迭代之前（while）还是之后（do..while）被测试。

在这两种形式中，如果条件测试得 false，那么下一次迭代就不会运行。这意味着如果条件初始时就是 false，那么 while 循环就永远不会运行，但是一个 do..while 循环将仅运行一次。

有时你会为了计数一组特定的数字来进行循环，比如从 0 到 9（十个数）。你可以通过设定一个值为 0 的循环迭代变量，比如 i，并在每次迭代时将它递增 1。

警告：由于种种历史原因，编程语言几乎总是用从零开始的方式来计数的，这意味着计数开始于 0 而不是 1。如果你不熟悉这种思维模式，一开始它可能十分令人困惑。为了更适应它，花些时间练习从 0 开始数数吧！

条件在每次迭代时都会被测试，好像在循环内部有一个隐含的 if 语句一样。

你可以使用 JavaScript 的 break 语句来停止一个循环。另外，我们可以看到如果没有 break 机制，就会极其容易地创造一个永远运行的循环。

让我们展示一下：

```
var i = 0;

// 一个 `while..true` 循环将会永远运行，对吧?
while (true) {
    // 停止循环?
    if ((i <= 9) === false) {
        break;
    }

    console.log( i );
    i = i + 1;
}
// 0 1 2 3 4 5 6 7 8 9
```

警告：这未必是你想在你的循环中使用的实际形式。它是仅为了说明的目的才出现在这里的。

虽然一个 while（或 do..while）可以手动完成任务，但是为了同样的目的，还有一种称为 for 循环的语法形式：

```
for (var i = 0; i <= 9; i = i + 1) {
    console.log( i );
```

```
}
// 0 1 2 3 4 5 6 7 8 9
```

如你所见，对于这两种循环形式来说，前 10 次迭代（i 的值从 0 到 9）的条件 i <= 9 都是 true，而且一旦 i 值为 10 就变为 false。

for 循环有三个子句：初始化子句（var i=0），条件测试子句（i <= 9），和更新子句（i = i + 1）。所以如果你想要使用循环迭代来计数，for 是一个更紧凑而且更易理解和编写的形式。

还有一些意在迭代特定的值的特殊循环形式，比如迭代一个对象的属性（见第二章），它隐含的测试条件是所有的属性是否都被处理过了。无论循环是何种形式，“循环直到条件失败” 的概念是它们共有的。

函数

手机店的店员可能不会拿着一个计算器到处走，用它来搞清税费和最终的购物款。这是一个她需要定义一次然后一遍又一遍地重用的任务。很有可能的是，公司有一个带有内建这些 “功能” 的收银机（电脑，平板电脑，等等）。

相似地，几乎可以肯定你的程序想要将代码的任务分割成可以重用的片段，而不是频繁地多次重复自己。这么做的方法是定义一个 function。

一个函数一般来说是一段被命名的代码，它可以使用名称来被 “调用”，而每次调用它内部的代码就会运行。考虑如下代码：

```
function printAmount() {
    console.log( amount.toFixed( 2 ) );
}

var amount = 99.99;

printAmount(); // "99.99"

amount = amount * 2;

printAmount(); // "199.98"
```

函数可以选择性地接收参数值（也就是参数）—— 你传入的值。而且它们还可以选择性地返回一个值。

```
function printAmount(amt) {
    console.log( amt.toFixed( 2 ) );
}
```

```
function formatAmount() {
    return "$" + amount.toFixed( 2 );
}

var amount = 99.99;

printAmount( amount * 2 );      // "199.98"

amount = formatAmount();
console.log( amount );          // "$99.99"
```

函数 printAmount(..) 接收一个参数，我们称之为 amt。函数 formatAmount() 返回一个值。当然，你也可以在同一个函数中组合这两种技术。

函数经常被用于你打算多次调用的代码，但它们对于仅将有关联的代码组织在一个命名的集合中也很有用，即便你只打算调用它们一次。

考虑如下代码：

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // 计算带有税费的新费用
    amt = amt + (amt * TAX_RATE);

    // 返回新费用
    return amt;
}

var amount = 99.99;

amount = calculateFinalPurchaseAmount( amount );

console.log( amount.toFixed( 2 ) );     // "107.99"
```

虽然 calculateFinalPurchaseAmount(..) 只被调用了一次，但是将它的行为组织进一个分离的带名称的函数，让使用它逻辑的代码（amount = calculateFinal... 语句）更干净。如果函数中拥有更多的语句，这种好处将会更加明显。

作用域

如果你向手机店的店员询问一款她们店里没有的手机，那么她就不能卖给你你想要的。她只能访问她们店库房里的手机。你不得不到另外一家店里去看看能不能找到你想要的手机。

编程对这种概念有一个术语：**作用域**（技术上讲称为 **词法作用域**）。在 JavaScript 中，每个函数都有自己的作用域。作用域基本上就是变量的集合，也是如何使用名称访问这些变量的规则。只有在这个函数内部的代码才能访问这个函数 **作用域内**的变量。

在同一个作用域内变量名必须是唯一的 —— 不能有两个不同的变量 `a` 并排出现。但是相同的变量名 `a` 可以出现在不同的作用域中。

```
function one() {
    // 这个 `a` 仅属于函数 `one()`
    var a = 1;
    console.log( a );
}

function two() {
    // 这个 `a` 仅属于函数 `two()`
    var a = 2;
    console.log( a );
}

one();      // 1
two();      // 2
```

另外，一个作用域可以嵌套在另一个作用域中，就像生日 Party 上的小丑在一个气球的里面吹另一个气球一样。如果一个作用域嵌套在另一个中，那么在内部作用域中的代码就可以访问这两个作用域中的变量。

考虑如下代码：

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

    function inner() {
        var b = 2;

        // 我们可以在这里同时访问 `a` 和 `b`
        console.log( a + b );   // 3
    }
```

```
    inner();

    // 我们在这里只能访问 `a`
    console.log( a );           // 1
}

outer();
```

词法作用域规则说，在一个作用域中的代码既可以访问这个作用域中的变量，又可以访问任何在它外面的作用域的变量。

所以，在函数 `inner()` 内部的代码可以同时访问变量 `a` 和 `b`，但是仅在 `outer()` 中的代码只能访问 `a` —— 它不能访问 `b` 因为这个变量仅存在于 `inner()` 内部。

回忆一下先前的这个代码段：

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // 计算带有税费的新费用
    amt = amt + (amt * TAX_RATE);

    // 返回新费用
    return amt;
}
```

因为词法作用域，常数 `TAX_RATE`（变量）可以从 `calculateFinalPurchaseAmount(..)` 函数中访问，即便它没有被传入这个函数。

注意：关于词法作用域的更多信息，参见本系列的 *作用域与闭包* 的前三章。

练习

在编程的学习中绝对没有什么可以替代练习。我写的再好也不可能使你成为一个程序员。

带着这样的意识，让我们试着练习一下我们在本章学到的一些概念。我将给出“需求”，而你首先试着实现它。然后参考下面的代码清单来看看我是怎么处理它的。* 写一个程序来计算你购买手机的总价。你将不停地购买手机直到你的银行账户上的钱都用光（提示：循环!）。你还将为每个手机购买配件，只要你的花费低于你心理预算。* 在你计算完购买总价之后，加入税费，然后用合适的格式打印出计算好的购买总价。* 最后，将总价与你银行账户上的余额作比较，来看看那你是否买的起。* 你应当为“税率”，“手机价格”，“配件价格”和“花费预算”设置一些常数，也为你的“银行账户余额”设置一个变量。* 你应当为税费的计算和价格的格式化 —— 使用一个“$”并四舍五入到小数点后两位 —— 定义函数。* **加分挑战**：试着在这个程序中利用输

入，也许是使用在前面的“输入”中讲过的 `prompt(..)`。比如，你可能会提示用户输入它们的银行账户余额。发挥创造力好好玩儿吧！

好的，去吧。试试看。在你自己实践过之前不要偷看我的代码清单！

注意：因为这是一本 JavaScript 书，很明显我将使用 JavaScript 解决这个联系。但是目前你可使用其他的语言，如果你感觉更适应的话。

对于这个练习，这是我的 JavaScript 解决方案：

```
const SPENDING_THRESHOLD = 200;
const TAX_RATE = 0.08;
const PHONE_PRICE = 99.99;
const ACCESSORY_PRICE = 9.99;

var bank_balance = 303.91;
var amount = 0;

function calculateTax(amount) {
    return amount * TAX_RATE;
}

function formatAmount(amount) {
    return "$" + amount.toFixed( 2 );
}

// 只要你还有钱就不停地买手机
while (amount < bank_balance) {
    // 买个新手机
    amount = amount + PHONE_PRICE;

    // 还买得起配件吗?
    if (amount < SPENDING_THRESHOLD) {
        amount = amount + ACCESSORY_PRICE;
    }
}

// 也别忘了给政府交钱
amount = amount + calculateTax( amount );

console.log(
```

```
    "Your purchase: " + formatAmount( amount )
);
// Your purchase: $334.76

// 你买的起吗?
if (amount > bank_balance) {
    console.log(
        "You can't afford this purchase. :("
    );
}
// 你买不起 :(
```

注意：运行这个 JavaScript 程序的最简单的方法是将它键入到你手边的浏览器的开发者控制台中。

你做的怎么样？看了我的代码之后，现在再试一次也没什么不好。而且你可以改变某些常数来看看使用不同的值时这个程序运行的如何。

复习

学习编程不一定是个复杂而且巨大的过程。你只需要在脑中装进几个基本的概念。

它们就像构建块儿。要建一座高塔，你就要从堆砌构建块儿开始。编程也一样。这里是一些编程中必不可少的构建块儿：

- 你需要 *操作符*来在值上实施动作。
- 你需要值和 *类型*来试试不同种类的动作，比如在 `number` 上做数学，或者使用 `string` 输出。
- 你需要 *变量*在你程序执行的过程中存储数据（也就是 *状态*）。
- 你需要 *条件*，比如 `if` 语句来做决定。
- 你需要 *循环*来重复任务，直到一个条件不再成立。
- 你需要 *函数*来将你的代码组织为有逻辑的和可复用的块儿。

代码注释是一种编写更好可读性代码的有效方法，它使你的代码更易理解，维护，而且如果稍后出现问题的话更易修改。

最后，不要忽视练习的力量。学习写代码的最好方法就是写代码。

现在，我很高兴看到你在学习编码的道路上走得很好！保持下去。不要忘了看看其他编程初学者的资源（书，博客，在线教学，等等）。这一章和这本书是一个很好的开始，但它们只是一个简要的介绍。

下一章将会复习许多本章中的概念，但是是从更加专门于 JavaScript 的视角，这将突出将在本系列的剩余部分将要深度剖析的大多数主要话题。

第二章：进入 JavaScript

在前一章中，我介绍了编程的基本构建块儿，比如变量，循环，条件，和函数。当然，所有被展示的代码都是 JavaScript。但是在这一章中，为了作为一个 JS 开发者入门和进阶，我们想要特别集中于那些你需要知道的关于 JavaScript 的事情。

我们将在本章中介绍好几个概念，它们将会在后续的 *YDKJS* 丛书中全面地探索。你可以将这一章看作是这个系列的其他书目中将要详细讲解的话题的一个概览。

特别是如果你刚刚接触 JavaScript，那么你应当希望花相当一段时间来多次复习这里的概念和代码示例。任何好的基础都是一砖一瓦积累起来的，所以不要指望你会在第一遍通读后就立即理解了全部内容。

你深入学习 JavaScript 的旅途从这里开始。

注意：正如我在第一章中说过的，在你通读这一章的同时，你绝对应该亲自尝试这里所有的代码。要注意的是，这里的有些代码假定最新版本的 JavaScript（通常称为 “ES6”，ECMAScript 的第六个版本 —— ECMAScript 是 JS 语言规范的官方名称）中引入的功能是存在的。如果你碰巧在使用一个老版本的，前 ES6 时代的浏览器，这些代码可能不好用。应当使用一个更新版本的现代浏览器（比如 Chrome，Firefox，或者 IE）。

值与类型

正如我们在第一章中宣称的，JavaScript 拥有带类型的值，没有带类型的变量。下面是可用的内建类型：

- `string`
- `number`
- `boolean`
- `null` 和 `undefined`
- `object`
- `symbol` （ES6 新增类型）

JavaScript 提供了一个 `typeof` 操作符，它可以检查一个值并告诉你它的类型是什么：

```
var a;
typeof a;               // "undefined"

a = "hello world";
typeof a;               // "string"

a = 42;
typeof a;               // "number"
```

```
a = true;
typeof a;               // "boolean"

a = null;
typeof a;               // "object" -- 奇怪的 bug

a = undefined;
typeof a;               // "undefined"

a = { b: "c" };
typeof a;               // "object"
```

来自 `typeof` 的返回值总是六个（ES6 中是七个！—— “symbol” 类型）字符串值之一。也就是，`typeof "abc"` 返回`"string"`，不是 `string`。

注意在这个代码段中变量 `a` 是如何持有每种不同类型的值的，而且尽管表面上看起来很像，但是 `typeof a` 并不是在询问 “`a` 的类型”，而是 “当前 `a` 中的值的类型”。在 JavaScript 中只有值拥有类型；变量只是这些值的简单容器。

`typeof null` 是一个有趣的例子，因为当你期望它返回`"null"` 时，它错误地返回了`"object"`。

警告：这是 JS 中一直存在的一个 bug，但是看起来它永远都不会被修复了。在网络上有太多的代码依存于这个 bug，因此修复它将会导致更多的 bug！

另外，注意 `a = undefined`。我们明确地将 `a` 设置为值 `undefined`，但是在行为上这与一个还没有被设定值的变量没有区别，比如在这个代码段顶部的 `var a;`。一个变量可以用好几种不同的方式得到这样的 “undefined” 值状态，包括没有返回值的函数和使用 `void` 操作符。

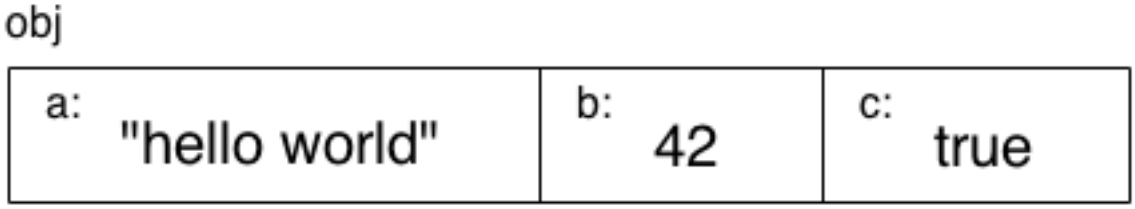
对象

`object` 类型指的是一种复合值，你可以在它上面设定属性（带名称的位置），每个属性持有各自的任意类型的值。它也许是 JavaScript 中最有用的类型之一。

```
var obj = {
    a: "hello world",
    b: 42,
    c: true
};

obj.a;      // "hello world"
obj.b;      // 42
obj.c;      // true
```

```
obj["a"];   // "hello world"
obj["b"];   // 42
obj["c"];   // true
```

可视化地考虑这个 obj 值可能会有所帮助：



属性既可以使用 *点号标记法*（例如，obj.a）访问，也可以使用 *方括号标记法*（例如，obj["a"]）访问。点号标记法更短而且一般来说更易于阅读，因此在可能的情况下它都是首选。

如果你有一个名称中含有特殊字符的属性名称，方括号标记法就很有用，比如 obj["hello world!"] —— 当通过方括号标记法访问时，这样的属性经常被称为 *键*。[ ] 标记法要求一个变量（下一节讲解）或者一个 string *字面量*（它需要包装进" .. " 或' .. '）。

当然，如果你想访问一个属性/键，但是它的名称被存储在另一个变量中时，方括号标记法也很有用。例如：

```
var obj = {
    a: "hello world",
    b: 42
};

var b = "a";

obj[b];         // "hello world"
obj["b"];       // 42
```

注意：更多关于 JavaScript 的 object 的信息，请参见本系列的 *this 与对象原型*，特别是第三章。

在 JavaScript 程序中有另外两种你将会经常打交道的值类型：*数组*和 *函数*。但与其说它们是内建类型，这些类型应当被认为更像是子类型 —— object 类型的特化版本。

数组

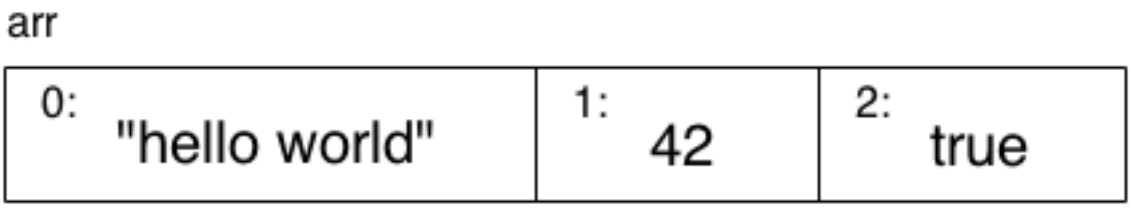
一个数组是一个 object，它不使用特殊的带名称的属性/键持有（任意类型的）值，而是使用数字索引的位置。例如：

```
var arr = [
    "hello world",
```

```
    42,
    true
];

arr[0];         // "hello world"
arr[1];         // 42
arr[2];         // true
arr.length;     // 3

typeof arr;     // "object"
```

注意：从零开始计数的语言，比如 JS，在数组中使用 0 作为第一个元素的索引。

可视化地考虑 arr 很能会有所帮助：



因为数组是一种特殊的对象（正如 typeof 所暗示的），所以它们可以拥有属性，包括一个可以自动被更新的 length 属性。

理论上你可以使用你自己的命名属性将一个数组用作一个普通对象，或者你可以使用一个 object 但是给它类似于数组的数字属性（0，1，等等）。然而，这么做一般被认为是分别误用了这两种类型。

最好且最自然的方法是为数字定位的值使用数组，而为命名属性使用 object。

函数

另一个你将在 JS 程序中到处使用的 object 子类型是函数：

```
function foo() {
    return 42;
}

foo.bar = "hello world";

typeof foo;         // "function"
typeof foo();       // "number"
typeof foo.bar;     // "string"
```

同样地，函数也是 object 的子类型 —— typeof 返回"function"，这暗示着"function"

是一种主要类型 —— 因此也可以拥有属性，但是你一般仅会在有限情况下才使用函数对象属性（比如 foo.bar）。

注意：更多关于 JS 的值和它们的类型的信息，参见本系列的 *类型与文法*的前两章。

内建类型的方法

我们刚刚讨论的内建类型和子类型拥有十分强大和有用的行为，它们作为属性和方法暴露出来。

例如：

```
var a = "hello world";
var b = 3.14159;

a.length;               // 11
a.toUpperCase();        // "HELLO WORLD"
b.toFixed(4);           // "3.1416"
```

使调用 a.toUpperCase() 成为可能的原因，要比这个值上存在这个方法的说法复杂一些。

简而言之，有一个 String（S 大写）对象包装器形式，通常被称为“原生类型”，与 string 基本类型配成一对儿；正是这个对象包装器的原型上定义了 toUpperCase() 方法。

当你通过引用一个属性或方法（例如，前一个代码段中的 a.toUpperCase()）将一个像"hello world" 这样的基本类型值当做一个 object 来使用时，JS 自动地将这个值“封箱”为它对应的对象包装器（这个操作是隐藏在幕后的）。

一个 string 值可以被包装为一个 String 对象，一个 number 可以被包装为一个 Number 对象，而一个 boolean 可以被包装为一个 Boolean 对象。在大多数情况下，你不担心或者直接使用这些值的对象包装器形式 —— 在所有实际情况中首选基本类型值形式，而 JavaScript 会帮你搞定剩下的一切。

注意：关于 JS 原生类型和“封箱”的更多信息，参见本系列的 *类型与文法*的第三章。要更好地理解对象原型，参见本系列的 *this 与对象原型*的第五章。

值的比较

在你的 JS 程序中你将需要进行两种主要的值的比较：*等价*和 *不等价*。任何比较的结果都是严格的 boolean 值（true 或 false），无论被比较的值的类型是什么。

强制转换

在第一章中我们简单地谈了一下强制转换，我们在此回顾它。

在 JavaScript 中强制转换有两种形式：*明确的*和 *隐含的*。明确的强制转换比较简单，因为你可以在代码中明显地看到一个类型转换到另一个类型将会发生，而隐含的强制转换更像是另外一些操作的不明显的副作用引发的类型转换。

你可能听到过像“强制转换是邪恶的”这样情绪化的观点，这是因为一个清楚的事实 —— 强制转换在某些地方会产生一些令人吃惊的结果。也许没有什么能比当一个语言吓到开发者时更能唤起他们的沮丧心情了。

强制转换并不邪恶，它也不一定是令人吃惊的。事实上，你使用类型强制转换构建的绝大部分情况是十分合理和可理解的，而且它甚至可以用来 增强你代码的可读性。但我们不会在这个话题上过度深入 —— 本系列的 类型与文法的第四章将会进行全面讲解。

这是一个 明确强制转换的例子：

```
var a = "42";

var b = Number( a );

a;              // "42"
b;              // 42 -- 数字!
```

而这是一个 隐含强制转换的例子：

```
var a = "42";

var b = a * 1;  // 这里 "42" 被隐含地强制转换为 42

a;              // "42"
b;              // 42 -- 数字!
```

Truthy 与 Falsy

在第一章中，我们简要地提到了值的“truthy”和“falsy”性质：当一个非 `boolean` 值被强制转换为一个 `boolean` 时，它是变成 `true` 还是 `false`。

在 JavaScript 中“falsy”的明确列表如下：

- `""` （空字符串）
- `0`, `-0`, `NaN` （非法的 `number`）
- `null`, `undefined`
- `false`

任何不在这个“falsy”列表中的值都是“truthy”。这是其中的一些例子：

- `"hello"`
- `42`
- `true`
- `[ ]`, `[ 1, "2", 3 ]` （数组）

- `{ }, { a: 42 }`（对象）
- `function foo() { .. }`（函数）

重要的是要记住，一个非 `boolean` 值仅在实际上被强制转换为一个 `boolean` 时才遵循这个 “truthy”/“falsy” 强制转换。把你搞糊涂并不困难 —— 当一个场景看起来像是将一个值强制转换为 `boolean`，可其实它不是。

等价性

有四种等价性操作符：`==`，`===`，`!=`，和`!==`。`!` 形式当然是与它们相对应操作符平行的 “不等” 版本；`不等`（*non-equality*）不应当与 `不等价性`（*inequality*）相混淆。

`==` 和 `===` 之间的不同通常被描述为，`==` 检查值的等价性而 `===` 检查值和类型两者的等价性。然而，这是不准确的。描述它们的合理方式是，`==` 在允许强制转换的条件下检查值的等价性，而 `===` 是在不允许强制转换的条件下检查值的等价性；因此 `===` 常被称为 “严格等价”。

考虑这个隐含强制转换，它在 `==` 宽松等价性比较中允许，而 `===` 严格等价性比较中不允许：

```
var a = "42";
var b = 42;

a == b;         // true
a === b;        // false
```

在 `a == b` 的比较中，JS 注意到类型不匹配，于是它经过一系列有顺序的步骤将一个值或者它们两者强制转换为一个不同的类型，直到类型匹配为止，然后就可以检查一个简单的值等价性。

如果你仔细想一想，通过强制转换 `a == b` 可以有两种方式给出 `true`。这个比较要么最终成为 `42 == 42`，要么成为`"42" == "42"`。那么是哪一种呢?

答案：`"42"` 变成 `42`，于是比较成为 `42 == 42`。在一个这样简单的例子中，只要最终结果是一样的，处理的过程走哪一条路看起来并不重要。但在一些更复杂的情况下，这不仅对比较的最终结果很重要，而且对你 *如何*得到这个结果也很重要。

`a === b` 产生 `false`，因为强制转换是不允许的，所以简单值的比较很明显将会失败。许多开发者感觉 `===` 更可靠，所以他们提倡一直使用这种形式而远离 `==`。我认为这种观点是非常短视的。我相信 `==` 是一种可以改进程序的强大工具，*如果你花时间去学习它的工作方式*。

我们不会详细地讲解强制转换在 `==` 比较中是如何工作的。它的大部分都是相当合理的，但是有一些重要的极端用例要小心。你可以阅读 ES5 语言规范的 11.9.3 部分（http://www.ecma-international.org/ecma-262/5.1/）来了解确切的规则，而且与围绕这种机制的所有负面炒作比起来，你会对这它是多么的直白而感到吃惊。

为了将这许多细节归纳为一个简单的包装，并帮助你在各种情况下判断是否使用 `==` 或 `===`，这是我的简单规则：

- 如果一个比较的两个值之一可能是 `true` 或 `false` 值，避免 `==` 而使用 `===`。
- 如果一个比较的两个值之一可能是这些具体的值（`0`，`""`，或 `[]` —— 空数组），避免 `==` 而使用 `===`。
- 在 *所有*其他情况下，你使用 `==` 是安全的。它不仅安全，而且在许多情况下它可以简化你的代码并改善可读性。

这些规则归纳出来的东西要求你严谨地考虑你的代码：什么样的值可能通过这个被比较等价性的变量。如果你可以确定这些值，那么 `==` 就是安全的，使用它！如果你不能确定这些值，就使用 `===`。就这么简单。

`!=` 不等价形式对应于 `==`，而`!==` 形式对应于 `===`。我们刚刚讨论的所有规则和注意点对这些非等价比较都是平行适用的。

如果你在比较两个非基本类型值，比如 `object`（包括 `function` 和 `array`），那么你应当特别小心 `==` 和 `===` 的比较规则。因为这些值实际上是通过引用持有的，`==` 和 `===` 比较都将简单地检查这个引用是否相同，而不是它们底层的值。

例如，`array` 默认情况下会通过使用逗号（`,`）连接所有值来被强制转换为 `string`。你可能认为两个内容相同的 `array` 将是 `==` 相等的，但它们不是：

```
var a = [1,2,3];
var b = [1,2,3];
var c = "1,2,3";

a == c;     // true
b == c;     // true
a == b;     // false
```

注意：更多关于 `==` 等价性比较规则的信息，参见 ES5 语言规范（11.9.3 部分），和本系列的 *类型与文法*的第四章；更多关于值和引用的信息，参见它的第二章。

不等价性

`<`，`>`，`<=`，和 `>=` 操作符用于不等价性比较，在语言规范中被称为 “关系比较”。一般来说它们将与 `number` 这样的可比较有序值一起使用。`3 < 4` 是很容易理解的。

但是 JavaScript`string` 值也可进行不等价性比较，它使用典型的字母顺序规则（`"bar" < "foo"`）。

那么强制转换呢？与 `==` 比较相似的规则（虽然不是完全相同！）也适用于不等价操作符。要注意的是，没有像 `===` 严格等价操作符那样不允许强制转换的 “严格不等价” 操作符。

考虑如下代码：

```
var a = 41;
var b = "42";
```

```
var c = "43";

a < b;      // true
b < c;      // true
```

这里发生了什么？在 ES5 语言规范的 11.8.5 部分中，它说如果 < 比较的两个值都是 string，就像 b < c，那么这个比较将会以字典顺序（也就是像字典中字母的排列顺序）进行。但如果两个值之一不是 string，就像 a < b，那么两个值就将被强制转换成 number，并进行一般的数字比较。

在可能不同类型的值之间进行比较时，你可能遇到的最大的坑 —— 记住，没有“严格不等价”可用 —— 是其中一个值不能转换为合法的数字，例如：

```
var a = 42;
var b = "foo";

a < b;      // false
a > b;      // false
a == b;     // false
```

等一下，这三个比较怎么可能都是 false？因为在 < 和 > 的比较中，值 b 被强制转换为了“非法的数字值”，而且语言规范说 NaN 既不大于其他值，也不小于其他值。

== 比较失败于不同的原因。如果 a == b 被解释为 42 == NaN 或者"42" == "foo" 都会失败 —— 正如我们前面讲过的，这里是前一种情况。

注意：关于不等价比较规则的更多信息，参见 ES5 语言规范的 11.8.5 部分，和本系列的 *类型与文法*第四章。

变量

在 JavaScript 中，变量名（包括函数名）必须是合法的 *标识符*（*identifiers*）。当你考虑非传统意义上的字符时，比如 Unicode，标识符中合法字符的严格和完整的规则就有点儿复杂。如果你仅考虑典型的 ASCII 字母数字的字符，那么这个规则还是很简单的。

一个标识符必须以 a-z，A-Z，$，或 _ 开头。它可以包含任意这些字符外加数字 0-9。

一般来说，变量标识符的规则也通用适用于属性名称。然而，有一些不能用作变量名，但是可以用作属性名的单词。这些单词被称为“保留字（reserved words）”，包括 JS 关键字（for，in，if，等等）和 null，true 和 false。

注意：更多关于保留字的信息，参见本系列的 *类型与文法*的附录 A。

函数作用域

你使用 var 关键字声明的变量将属于当前的函数作用域，如果声明位于任何函数外部的顶层，它就属于全局作用域。

提升

无论 var 出现在一个作用域内部的何处，这个声明都被认为是属于整个作用域，而且在作用域的所有位置都是可以访问的。

这种行为称为 *提升*，比喻一个 var 声明在概念上 *被移动*到了包含它的作用域的顶端。技术上讲，这个过程通过代码的编译方式进行解释更准确，但是我们先暂且跳过那些细节。

考虑如下代码：

```
var a = 2;

foo();                  // 可以工作，因为 `foo()` 声明被“提升”了

function foo() {
    a = 3;

    console.log( a );   // 3

    var a;              // 声明被“提升”到了 `foo()` 的顶端
}

console.log( a );   // 2
```

警告：在一个作用域中依靠变量提升来在 var 声明出现之前使用一个变量是不常见的，也不是个好主意；它可能相当使人困惑。而使用被提升的函数声明要常见得多，也更为人所接受，就像我们在 foo() 正式声明之前就调用它一样。

嵌套的作用域

当你声明了一个变量时，它就在这个作用域内的任何地方都是可用的，包括任何下层/内部作用域。例如：

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

    function bar() {
        var b = 2;

        function baz() {
            var c = 3;

            console.log( a, b, c ); // 1 2 3
```

```
        }

        baz();
        console.log( a, b );        // 1 2
    }

    bar();
    console.log( a );               // 1
}

foo();
```

注意 c 在 bar() 的内部是不可用的，因为它是仅在内部的 baz() 作用域中被声明的，并且 b 因为同样的原因在 foo() 内是不可用的。

如果你试着在一个作用域内访问一个不可用的变量的值，你就会得到一个被抛出的 ReferenceError。如果你试着为一个还没有被声明的变量赋值，那么根据“strict 模式”的状态，你会要么得到一个在顶层全局作用域中创建的变量（不好!），要么得到一个错误。让我们看一下：

```
function foo() {
    a = 1;  // `a` 没有被正式声明
}

foo();
a;          // 1 -- 噢，自动全局变量 :(
```

这是一种非常差劲儿的做法。别这么干！总是给你的变量进行正式声明。

除了在函数级别为变量创建声明，ES6 允许你使用 let 关键字声明属于个别块儿（一个 { .. }）的变量。除了一些微妙的细节，作用域规则将大致上与我们刚刚看到的函数相同：

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

    if (a >= 1) {
        let b = 2;

        while (b < 5) {
            let c = b * 2;
            b++;
```

```
            console.log( a + c );
        }
    }
}

foo();
// 5 7 9
```

因为使用了 `let` 而非 `var`，`b` 将仅属于 `if` 语句而不是整个 `foo()` 函数的作用域。相似地，`c` 仅属于 `while` 循环。对于以更加细粒度的方式管理你的变量作用域来说，块儿作用域是非常有用的，它将使你的代码随着时间的推移更加易于维护。

注意：关于作用域的更多信息，参见本系列的 *作用域与闭包*。更多关于 `let` 块儿作用域的信息，参见本系列的 *ES6 与未来*。

条件

除了我们在第一章中简要介绍过的 `if` 语句，JavaScript 还提供了几种其他值得我们一看的条件机制。

有时你可能发现自己在像这样写一系列的 `if..else..if` 语句：

```
if (a == 2) {
    // 做一些事情
}
else if (a == 10) {
    // 做另一些事请
}
else if (a == 42) {
    // 又是另外一些事情
}
else {
    // 这里是备用方案
}
```

这种结构好用，但有一点儿繁冗，因为你需要为每一种情况都指明 `a` 的测试。这里有另一种选项，`switch` 语句：

```
switch (a) {
    case 2:
        // 做一些事情
```

```
        break;
    case 10:
        // 做另一些事请
        break;
    case 42:
        // 又是另外一些事情
        break;
    default:
        // 这里是备用方案
}
```

如果你想仅让一个 case 中的语句运行，break 是很重要的。如果你在一个 case 中省略了 break，并且这个 case 成立或运行，那么程序的执行将会不管下一个 case 语句是否成立而继续执行它。这种所谓的“掉落”有时是有用/期望的：

```
switch (a) {
    case 2:
    case 10:
        // 一些很酷的事情
        break;
    case 42:
        // 另一些事情
        break;
    default:
        // 备用方案
}
```

这里，如果 a 是 2 或 10，它就会执行“一些很酷的事情”的代码语句。

在 JavaScript 中的另一种条件形式是“条件操作符”，经常被称为“三元操作符”。它像是一个单独的 if..else 语句的更简洁的形式，比如：

```
var a = 42;

var b = (a > 41) ? "hello" : "world";

// 与此相似：

// if (a > 41) {
//     b = "hello";
```

```
// }
// else {
//    b = "world";
// }
```

如果测试表达式（这里是 `a > 41`）求值为 `true`，那么就会得到第一个子句（`"hello"`），否则得到第二个子句（`"world"`），而且无论结果为何都会被赋值给 `b`。

条件操作符不一定非要用于赋值，但是这绝对是最常见的用法。

注意：关于测试条件和 `switch` 与`? :` 的其他模式的更多信息，参见本系列的 *类型与文法*。

Strict 模式

ES5 在语言中加入了一个 “strict 模式”，它收紧了一些特定行为的规则。一般来说，这些限制被视为使代码符合一组更安全和更合理的指导方针。另外，坚持 strict 模式一般会使你的代码对引擎有更强的可优化性。strict 模式对代码有很大的好处，你应当在你所有的程序中使用它。

根据你摆放 strict 模式注解的位置，你可以为一个单独的函数，或者是整个一个文件切换到 strict 模式：

```
function foo() {
    "use strict";

    // 这部分代码是 strict 模式的

    function bar() {
        // 这部分代码是 strict 模式的
    }
}

// 这部分代码不是 strict 模式的
```

将它与这个相比：

```
"use strict";

function foo() {
    // 这部分代码是 strict 模式的

    function bar() {
        // 这部分代码是 strict 模式的
```

```
    }
}

// 这部分代码是 strict 模式的
```

使用 strict 模式的一个关键不同（改善！）是，它不允许因为省略了 `var` 而进行隐含的自动全局变量声明：

```
function foo() {
    "use strict";   // 打开 strict 模式
    a = 1;          // 缺少 `var`, ReferenceError
}

foo();
```

如果你在代码中打开 strict 模式，并且得到错误，或者代码开始变得有 bug，这可能会诱使你避免使用 strict 模式。但是纵容这种直觉不是一个好主意。如果 strict 模式在你的程序中导致了问题，那么这标志着在你的代码中几乎可以肯定有应该修改的东西。

strict 模式不仅将你的代码保持在更安全的道路上，也不仅将使你的代码可优化性更强，它还代表着这种语言未来的方向。对于你来说，现在就开始习惯于 strict 模式要比一直回避它容易得多 —— 以后再进行这种转变只会更难！

注意：关于 strict 模式的更多信息，参见本系列的 *类型与文法*的第五章。

函数作为值

至此，我们已经将函数作为 JavaScript 中主要的 *作用域*机制讨论过了。你可以回想一下典型的 `function` 声明语法是这样的：

```
function foo() {
    // ..
}
```

虽然从这种语法中看起来不明显，`foo` 基本上是一个位于外围作用域的变量，它给了被声明的 `function` 一个引用。也就是说，`function` 本身是一个值，就像 `42` 或 `[1,2,3]` 一样。

这可能听起来像是一个奇怪的概念，所以花点儿时间仔细考虑一下。你不仅可以向一个 `function` 传递一个值（参数值），而且 *一个函数本身可以是一个值*，它能够赋值给变量，传递给其他函数，或者从其它函数中返回。

因此，一个函数值应当被认为是一个表达式，与任何其他的值或表达式很相似。

考虑如下代码：

```
var foo = function() {
    // ..
};

var x = function bar(){
    // ..
};
```

第一个被赋值给变量 foo 的函数表达式称为 匿名函数表达式，因为它没有“名称”。

第二个函数表达式是 命名的（bar），它还被赋值给变量 x 作为它的引用。命名函数表达式一般来说更理想，虽然 匿名函数表达式仍然极其常见。

更多信息参见本系列的 作用域与闭包。

立即被调用的函数表达式（IIFE）

在前一个代码段中，哪一个函数表达式都没有被执行 —— 除非我们使用了 foo() 或 x()。

有另一种执行函数表达式的方法，它通常被称为一个 立即被调用的函数表达式（IIFE）：

```
(function IIFE(){
    console.log( "Hello!" );
})();
// "Hello!"
```

围绕在函数表达式 (function IIFE(){ .. }) 外部的 (..) 只是一个微妙的 JS 文法，我们需要它来防止函数表达式被看作一个普通的函数声明。

在表达式末尾的最后的 () —— })(); 这一行 —— 才是实际立即执行它前面的函数表达式的东西。

这看起来可能很奇怪，但它不像第一眼看上去那么陌生。考虑这里的 foo 和 IIFE 之间的相似性：

```
function foo() { .. }

// `foo` 是函数引用表达式，然后用 `()`执行它
foo();

// `IIFE` 是函数表达式，然后用 `()`执行它
(function IIFE(){ .. })();
```

如你所见，在执行它的 () 之前列出 (function IIFE(){ .. })，与在执行它的 () 之前定义 foo 实质上是相同的；在这两种情况下，函数引用都使用立即在它后面的 () 执行。

因为 IIFE 只是一个函数，而函数可以创建变量 作用域，以这样的风格使用一个 IIFE 经常被用于定义变量，而这些变量将不会影响围绕在 IIFE 外面的代码：

```
var a = 42;

(function IIFE(){
    var a = 10;
    console.log( a );   // 10
})();

console.log( a );       // 42
```

IIFE 还可以有返回值：

```
var x = (function IIFE(){
    return 42;
})();

x;  // 42
```

值 42 从被执行的命名为 IIFE 的函数中 return，然后被赋值给 x。

闭包

闭包是 JavaScript 中最重要，却又经常最少为人知的概念之一。我不会在这里涵盖更深的细节，你可以参照本系列的 作用域与闭包。但我想说几件关于它的事情，以便你了解它的一般概念。它将是你的 JS 技术结构中最重要的技术之一。

你可以认为闭包是这样一种方法：即使函数已经完成了运行，它依然可以“记住”并持续访问函数的作用域。

考虑如下代码：

```
function makeAdder(x) {
    // 参数 `x` 是一个内部变量

    // 内部函数 `add()` 使用 `x`，所以它对 `x` 拥有一个“闭包”
    function add(y) {
        return y + x;
    };

    return add;
}
```

每次调用外部的 `makeAdder(..)` 所返回的对内部 `add(..)` 函数的引用可以记住被传入 `makeAdder(..)` 的 `x` 值。现在，让我们使用 `makeAdder(..)`：

```
// `plusOne` 得到一个指向内部函数 `add(..)` 的引用，
// `add()` 函数拥有对外部 `makeAdder(..)` 的参数 `x`
// 的闭包
var plusOne = makeAdder( 1 );

// `plusTen` 得到一个指向内部函数 `add(..)` 的引用，
// `add()` 函数拥有对外部 `makeAdder(..)` 的参数 `x`
// 的闭包
var plusTen = makeAdder( 10 );

plusOne( 3 );        // 4  <-- 1 + 3
plusOne( 41 );       // 42 <-- 1 + 41

plusTen( 13 );       // 23 <-- 10 + 13
```

这段代码的工作方式是：

1. 当我们调用 `makeAdder(1)` 时，我们得到一个指向它内部的 `add(..)` 的引用，它记住了 `x` 是 `1`。我们称这个函数引用为 `plusOne(..)`。
2. 当我们调用 `makeAdder(10)` 时，我们得到了另一个指向它内部的 `add(..)` 引用，它记住了 `x` 是 `10`。我们称这个函数引用为 `plusTen(..)`。
3. 当我们调用 `plusOne(3)` 时，它在 `3`（它内部的 `y`）上加 `1`（被 `x` 记住的），于是我们得到结果 `4`。
4. 当我们调用 `plusTen(13)` 时，它在 `13`（它内部的 `y`）上加 `10`（被 `x` 记住的），于是我们得到结果 `23`。

如果这看起来很奇怪和令人困惑，不要担心 —— 它确实是的！要完全理解它需要很多的练习。

但是相信我，一旦你理解了它，它就是编程中最强大最有用的技术之一。让你的大脑在闭包中煎熬一会是绝对值得的。在下一节中，我们将进一步实践闭包。

模块

在 JavaScript 中闭包最常见的用法就是模块模式。模块让你定义对外面世界不可见的私有实现细节（变量，函数），和对外面可访问的公有 API。

考虑如下代码：

```
function User(){
    var username, password;

    function doLogin(user,pw) {
        username = user;
        password = pw;

        // 做登录的工作
    }

    var publicAPI = {
        login: doLogin
    };

    return publicAPI;
}

// 创建一个 `User` 模块的实例
var fred = User();

fred.login( "fred", "12Battery34!" );
```

函数 User() 作为一个外部作用域持有变量 username 和 password，以及内部 doLogin() 函数；它们都是 User 模块内部的私有细节，是不能从外部世界访问的。

警告：我们在这里没有调用 new User()，这是有意为之的，虽然对大多数读者来说那可能更常见。User() 只是一个函数，不是一个要被初始化的对象，所以它只是被一般地调用了。使用 new 将是不合适的，而且实际上会浪费资源。

执行 User() 创建了 User 模块的一个 *实例* —— 一个全新的作用域会被创建，而每个内部变量/函数的一个全新的拷贝也因此而被创建。我们将这个实例赋值给 fred。如果我们再次运行 User()，我们将会得到一个与 fred 完全分离的新的实例。

内部的 doLogin() 函数在 username 和 password 上拥有闭包，这意味着即便 User() 函数已经完成了运行，它依然持有对它们的访问权。

publicAPI 是一个带有一个属性/方法的对象，login 是一个指向内部 doLogin() 函数的引用。当我们从 User() 中返回 publicAPI 时，它就变成了我们称为 fred 的实例。

在这个时候，外部的 User() 函数已经完成了执行。一般说来，你会认为像 username 和 password 这样的内部变量将会消失。但是在这里它们不会，因为在 login() 函数里有一个闭包使它们继续存活。

这就是为什么我们可以调用 fred.login(..) —— 和调用内部的 doLogin(..) 一样 ——

而且它依然可以访问内部变量 `username` 和 `password`。

这样对闭包和模块模式的简单一瞥，你很有可能还是有点儿糊涂。没关系！要把它装进你的大脑确实需要花些功夫。

以此为起点，关于更多深入细节的探索可以去读本系列的 *作用域与闭包*。

this 标识符

在 JavaScript 中另一个经常被误解的概念是 `this` 标识符。同样，在本系列的 *this 与对象原型*中有好几章关于它的内容，所以在这里我们只简要地介绍一下概念。

虽然 `this` 可能经常看起来是与 “面向对象模式” 有关的，但在 JS 中 `this` 是一个不同的概念。

如果一个函数在它内部拥有一个 `this` 引用，那么这个 `this` 引用通常指向一个 `object`。但是指向哪一个 `object` 要看这个函数是如何被调用的。

重要的是要理解 `this` *不是*指函数本身，这是最常见的误解。

这是一个快速的说明：

```
function foo() {
    console.log( this.bar );
}

var bar = "global";

var obj1 = {
    bar: "obj1",
    foo: foo
};

var obj2 = {
    bar: "obj2"
};

// --------

foo();              // "global"
obj1.foo();         // "obj1"
foo.call( obj2 );   // "obj2"
new foo();          // undefined
```

关于 `this` 如何被设置有四个规则，它们被展示在这个代码段的最后四行中：

1. `foo()` 最终在非 strict 模式中将 `this` 设置为全局对象 —— 在 strict 模式中，`this` 将会是 `undefined` 而且你会在访问 `bar` 属性时得到一个错误 —— 所以 `this.bar` 的值是 `global`。
2. `obj1.foo()` 将 `this` 设置为对象 `obj1`。
3. `foo.call(obj2)` 将 `this` 设置为对象 `obj2`。
4. `new foo()` 将 `this` 设置为一个新的空对象。

底线：要搞清楚 `this` 指向什么，你必须检视当前的函数是如何被调用的。它将是我们刚刚看到的四种中的一种，而这将会回答 `this` 是什么。

注意：关于 `this` 的更多信息，参见本系列的 *this 与对象原型*的第一和第二章。

原型

JavaScript 中的原型机制十分复杂。我们在这里仅仅扫它一眼。要了解关于它的所有细节，你需要花相当的时间来学习本系列的 *this 与对象原型*的第四到六章。

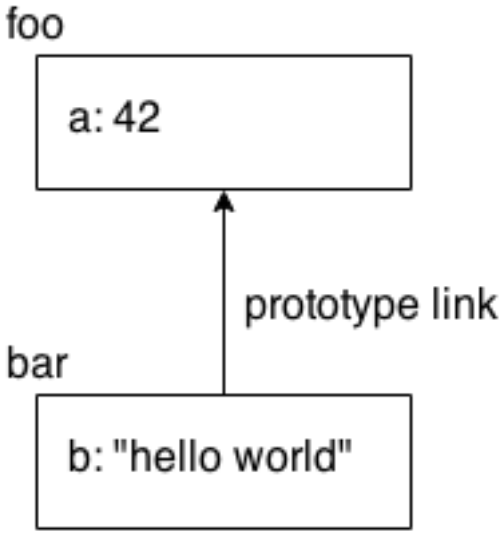
当你引用一个对象上的属性时，如果这个属性不存在，JavaScript 将会自动地使用这个对象的内部原型引用来寻找另外一个对象，在它上面查询你想要的属性。你可以认为它几乎是在属性缺失时的备用对象。

从一个对象到它备用对象的内部原型引用链接发生在这个对象被创建的时候。说明它的最简单的方法是使用称为 `Object.create(..)` 的内建工具。

考虑如下代码：

```
var foo = {
    a: 42
};

// 创建 `bar` 并将它链接到 `foo`
var bar = Object.create( foo );

bar.b = "hello world";

bar.b;      // "hello world"
bar.a;      // 42 <-- 委托到 `foo`
```

将对象 `foo` 和 `bar` 以及它们的关系可视化也许会有所帮助：



属性 `a` 实际上不存在于对象 `bar` 上，但是因为 `bar` 被原型链接到 `foo`，JavaScript 自动地退到对象 `foo` 上去寻找 `a`，而且在这里找到了它。

这种链接看起来是语言的一种奇怪的特性。这种特性最常被使用的方式 —— 我会争辩说这是一种滥用 —— 是用来模拟/模仿 “类” 机制的 “继承”。

使用原型的更自然的方式是一种称为 “行为委托” 的模式，在这种模式中你有意地将你的被链接的对象设计为可以从一个委托到另一个的部分所需的行为中。

注意：更多关于原型和行为委托的信息，参见本系列的 *this 与对象原型*的第四到六章。

旧的与新的

我们已经介绍过的 JS 特性，和将在这个系列的其他部分中讲解的相当一部分特性都是新近增加的，不一定在老版本的浏览器中可用。事实上，语言规范中的一些最新特性甚至在任何稳定的浏览中都没有被实现。

那么，你拿这些新东西怎么办？你只能等上几年或者十几年直到老版本浏览器归于尘土？

这确实是许多人认为的情况，但是它不是 JS 健康的进步方式。

有两种主要的技术可以将新的 JavaScript 特性 “带到” 老版本的浏览器中：填补和转译。

填补

“填补（Polyfilling）” 是一个人为发明的词（由 Remy Sharp 创造）（https://remysharp.com/2010/10/08/what-is-a-polyfill ）。它是指拿来一个新特性的定义并制造一段行为等价的代码，但是这段代码可以运行在老版本的 JS 环境中。

例如，ES6 定义了一个称为 `Number.isNaN(..)` 的工具，来为检查 `NaN` 值提供一种准确无误的方法，同时废弃原来的 `isNaN(..)` 工具。这个工具可以很容易填补，因此你可开始在你的代码中使用它，而不管最终用户是否在一个 ES6 浏览器中。

考虑如下代码：

```
if (!Number.isNaN) {
    Number.isNaN = function isNaN(x) {
        return x !== x;
    };
}
```

`if` 语句决定着在这个工具已经存在的 ES6 环境中不再进行填补。如果它还不存在，我们就定义 `Number.isNaN(..)`。

注意：我们在这里做的检查利用了 `NaN` 值的怪异之处，即它们是整个语言中唯一与自己不相等的值。所以 `NaN` 是唯一可能使 `x !== x` 为 `true` 的值。

并不是所有的新特性都可以完全填补。有时一种特性的大部分行为可以被填补，但是仍然存在一些小的偏差。在实现你自己的填补时你应当非常非常小心，来确保你尽可能严格地遵循语言规范。

或者更好地，使用一组你信任的，经受过检验的填补，比如那些由 ES5-Shim（https://github.com/es-shims/es5-shim ）和 ES6-Shim（https://github.com/es-shims/es6-shim ）提供的。

转译

没有任何办法可以填补语言中新增加的语法。在老版本的 JS 引擎中新的语法将因为不可识别/不合法而抛出一个错误。

所以更好的选择是使用一个工具将你的新版本代码转换为等价的老版本代码。这个处理通常被称为“转译（transpiling）”，表示转换 + 编译。

实质上，你的源代码是使用新的语法形式编写的，但是你向浏览器部署的是转译过的旧语法形式。你一般会将转译器插入到你的构建过程中，与你的代码 linter 和代码压缩器类似。

你可能想知道为什么要麻烦地使用新语法编写程序又将它转译为老版本代码 —— 为什么不直接编写老版本代码呢?

关于转译你应当注意几个重要的原因：

- 在语言中新加入的语法是为了使你的代码更具可读性和维护性而设计的。老版本的等价物经常会绕多得多的圈子。你应当首选编写新的和干净的语法，不仅为你自己，也为了开发团队的其他的成员。
- 如果你仅为老版本浏览器转译，而给最新的浏览器提供新语法，那么你就可以利用浏览器对新语法进行的性能优化。这也让浏览器制造商有更多真实世界的代码来测试它们的实现和优化方法。
- 提早使用新语法可以允许它在真实世界中被测试得更加健壮，这给 JavaScript 委员会（TC39）提供了更早的反馈。如果问题被发现的足够早，他们就可以在那些语言设计错误变得无法挽回之前改变/修改它。

这是一个转译的简单例子。ES6 增加了一个称为“默认参数值”的新特性。它看起来像是这样：

```
function foo(a = 2) {
    console.log( a );
}

foo();      // 2
foo( 42 );  // 42
```

简单，对吧？也很有用！但是这种新语法在前 ES6 引擎中是不合法的。那么转译器将会对这段代码做什么才能使它在老版本环境中运行呢？

```
function foo() {
    var a = arguments[0] !== (void 0) ? arguments[0] : 2;
    console.log( a );
}
```

如你所见，它检查 `arguments[0]` 值是否是 `void 0`（也就是 `undefined`），而且如果是，就提供默认值 `2`；否则，它就赋值被传递的任何东西。

除了可以现在就在老版本浏览器中使用更好的语法以外，观察转译后的代码实际上更清晰地解释了意图中的行为。

仅从 ES6 版本的代码看来，你可能还不理解 `undefined` 是唯一不能作为参数默认值的明确传递的值，但是转译后的代码使这一点清楚的多。

关于转译要强调的最后一个细节是，现在它们应当被认为是 JS 开发的生态系统和过程中的标准部分。JS 将继续以比以前快得多的速度进化，所以每几个月就会有新语法和新特性被加入进来。

如果你默认地使用一个转译器，那么你将总是可以在发现新语法有用时，立即开始使用它，而不必为了让今天的浏览器被淘汰而等上好几年。

有好几个了不起的转译器供你选择。这是一些在本书写作时存在的好选择：

- Babel (https://babeljs.io) (前身为 6to5)：将 ES6+ 转译为 ES5
- Traceur (https://github.com/google/traceur-compiler)：将 ES6，ES7，和以后特性转译为 ES5

非 JavaScript

至此，我们讨论过的所有东西都限于 JS 语言本身。现实是大多数 JS 程序都是在浏览器这样的环境中运行并与之互动的。你所编写的很大一部分代码，严格地说，不是直接由 JavaScript 控制的。这听起来可能有点奇怪。

你将会遇到的最常见的非 JavaScript 程序是 DOM API。例如：

```
var el = document.getElementById( "foo" );
```

当你的代码运行在一个浏览器中时，变量 `document` 作为一个全局变量存在。它不是由 JS 引擎提供的，也不为 JavaScript 语言规范所控制。它采取了某种与普通 JS `object` 极其相似的形式，但它不是真正的 `object`。它是一种特殊的 `object`，经常被称为“宿主对象”。

另外，`document` 上的 `getElementById(..)` 方法看起来像一个普通的 JS 函数，但它只是一个微微暴露出来的接口，指向由浏览器 DOM 提供的内建方法。在一些（新一代的）浏览器中，这一层可能也是由 JS 实现的，但是传统的 DOM 及其行为是由像 C/C++ 这样的语言实现的。

另一个例子是输入/输出（I/O）。

大家最喜爱的 `alert(..)` 在用户的浏览器窗口中弹出一个消息框。`alert(..)` 是由浏览器提供给你的 JS 程序的，而不是 JS 引擎本身。你进行的调用将消息发送给浏览器内部，它来处理消息框的绘制与显示。

`console.log()` 也一样；你的浏览器提供这样的机制并将它们挂在开发者工具中。

这本书，和整个这个系列，聚焦于 JavaScript 语言本身。这就是为什么你看不到任何涵盖这些非 JavaScript 机制的重要内容。不管怎样，你需要小心它们，因为它们将在你写的每一个 JS 程序中存在！

复习

学习 JavaScript 风格编程的第一步是对它的核心机制有一个基本的了解，比如：值，类型，函数闭包，`this`，和原型。

当然，这些话题中的每一个都会衍生出比你在这里见到的多得多的内容，这也是为什么它们在这个系列剩下的部分中拥有自己的章节和书目。在你对本章中的概念和代码示例感到相当适应之后，这个系列的其他部分正等着你真正地深入挖掘和了解这门语言。

这本书的最后一章将会对这个系列的每一卷的内容，以及它们所涵盖的我们在这里还没有探索过的概念，进行简单地总结。

第三章：进入 YDKJS

这个系列丛书到底是为了什么？简单地说，它的目的是认真地学习 *JavaScript 的所有部分*，不仅是这门语言的某些人称之为“好的部分”的子集，也不仅是让你在工作中搞定任务所需的最小部分的知识。

其他语言中，认真的开发者总是希望努力学习他们主要使用的语言的大部分或全部，但是 JS 开发者由于通常不太学习这门语言而在人群中显得很扎眼。这不是一件好事，而且我们也不应当继续将之视为常态。

你不懂 JS（*YDKJS*）系列的立场是与学习 JS 的通常方式形成鲜明的对比，而且与你将会读到的其他 JS 书籍不同。它挑战你超越自己的舒适区，对每一个你遇到的行为问一个更深入的“为什么”。你准备好接受挑战了吗？

我将用这最后一章的篇幅来简要地总结一下这个系列其他书目的内容，和如何在 *YDKJS* 的基础上最有效地建立学习 JS 的基础。

作用域与闭包

也许你需要快速接受的基础之一，就是在 JavaScript 中变量的作用域是如何工作的。关于作用域仅有传闻中的模糊 观念是不够的。

作用域与闭包从揭穿常见的误解开始：JS 是 “解释型语言” 因此是不被编译的。不对。

JS 引擎在你的代码执行的前一刻（有时是在执行期间！）编译它。所以我们首先深入了解编译器处理我们代码的方式，以此来理解它如何找到并处理变量和函数的声明。沿着这条道路，我们将见到 JS 变量作用域管理的特有隐喻，“提升”。

对 “词法作用域” 的极其重要的理解，是我们在这本书最后一章探索闭包时所需的基石。闭包也许是 JS 所有的概念中最重要的一个，但如果你没有首先牢牢把握住作用域的工作方式，那么闭包将很可能依然不在你的掌握之中。

闭包的一个重要应用是模块模式，正如我们在本书第二章中简要介绍过的那样。模块模式也许是 JavaScript 的所有代码组织模式中最流行的一种；深刻理解它应当是你的首要任务之一。

this 与对象原型

也许关于 JavaScript 传播得最广泛和持久的谬误之一是认为 `this` 关键字指代它所出现的函数。可怕的错误。

`this` 关键字是根据函数如何被执行而动态绑定的，而事实上有四种简单的规则可以用来理解和完全决定 `this` 绑定。

和 `this` 密切相关的是对象原型属性，它是一种属性的查询链，与查询词法作用域变量的方式相似。但是原型中包含的是另一个关于 JS 的巨大谬误：模拟（山寨）类和继承（所谓的 “原型继承”）的想法。

不幸的是，渴望将类和继承的设计模式思想带入 JavaScript 只是你能做的最差劲儿的事情，因为虽然语法可能欺骗你，使你认为有类这样的东西存在，但实际上原型机制在行为上是根本相反的。

目前的问题是，是忽略这种错位并假装你实现的是 “继承” 更好，还是学习并接纳对象原型系统实际的工作方式更恰当。后者被称为 “行为委托” 更合适。

这不光是语法上的偏好问题。委托是一种完全不同的，更强大的设计模式，其中的原因之一就是它取代了使用类和继承进行设计的需要。但是对于以谈论 JavaScript 的一生为主题的几乎所有的其他博客，书籍，和论坛来说，这些断言绝对是打脸的。

我对委托和继承做出的宣言不是源于对语言和其语法的厌恶，而是来自于渴望看到这门语言的真实力量被正确地利用，渴望看到无尽的困惑与沮丧被一扫而光。

但是我举出的关于原型和委托的例子可要比我在这里乱说的东西复杂得多。如果你准备好重新思考你认为你所了解的关于 JavaScript“类” 和 “继承” 的一切，我给你一个机会来 “服用红色的药丸”，并且看一看本系列的 *this 与对象原型* 的第四到六章。

类型与文法

这个系列的第三本书主要集中于解决另一个极具争议的话题：类型强制转换。也许没有什么话题能比你谈论隐含的强制转换造成的困惑更能使 JS 开发者感到沮丧了。

到目前为止，惯例的智慧说隐含强制转换是这门语言的 “坏的部分”，并且应当不计一切避免它。事实上，有些人已经到了将它称为语言设计的 “缺陷” 的地步了。确实存在这么一些工具，它们的全部工作就是扫描你的代码，并在你进行任何强制转换，甚至是做有些像强制转换的事情时报警。

但是强制转换真的如此令人困惑，如此的坏，如此的不可信，以至于只要你使用它，你的代码从一开始就灭亡了吗?

我说不。在第一到三章中建立了对类型和值真正的工作方式的理解后，第四章参与了这个辩论，并从强制转换的角落和缝隙全面地讲解它的工作方式。我们将看到强制转换的哪一部分真的令人惊讶，而且如果花时间去学习，哪一部分实际上完全是合理的。

但我不仅仅要说强制转换是合理的和可以学习的，我断言强制转换是一种 *你应当在代码中使用的*极其有用而且完全被低估的工具。我要说在合理使用的情况下，强制转换不仅可以工作，而且会使你的代码更好。所有唱反调的和怀疑的人当然会嘲笑这样的立场，但我相信它是让你玩儿好 JS 游戏的主要按键之一。

你是想继续人云亦云，还是想将所有的臆测放在一边，用一个全新的视角观察强制转换? 这个系列的 *类型与文法*将会强制转换你的想法。

异步与性能

这个系列的前三本书聚焦于这门语言的核心技术，但是第四本书稍稍开出一个分支来探讨在这门语言技术之上的管理异步编程的模式。异步不仅对于性能和我们的应用程序很关键，而且它日渐成为改进可写性和可维护性的关键因素。

这本书从搞清楚许多令人困惑的术语和概念开始，比如 “异步”，“并行” 和 “并发”。而且深入讲解了这些东西如何适用和不适用于 JS。

然后我们继续检视作为开启异步的主要方法：回调。但我们很快就会看到，对于现代异步编程的需求来说，单靠回调自身是远远不够的。我们将找出仅使用回调编码的两种主要的不足之处：*控制反转*（IoC）信任丢失和缺乏线性的可推理性。

为了解决这两种主要的不足，ES6 引入了两种新的机制（实际上也是模式）：promise 和 generator。

Prmise 是一个 “未来值” 的一种与时间无关的包装，它让你推理并组合这些未来值而不必关心它们是否已经准备好。另外，它们通过将回调沿着一个可信赖和可组装的 promise 机制传递，有效地解决了 IoC 信任问题。

Generator 给 JS 函数引入了一种新的执行模式，generator 可以在 `yield` 点被暂停而稍后异步地被继续。这种 “暂停-继续” 的能力让 generator 在幕后异步地被处理，使看起来同步，顺序执行的代码成为可能。如此，我们就解决了回调的非线性，非本地跳转的困惑，并因此使我们的异步代码看起来是更容易推理的同步代码。

但是，是 promise 与 generator 的组合给了我们 JavaScript 中最有效的异步代码模式。事实上，在即将到来的 ES7 与之后的版本中，大多数精巧的异步性肯定会建立在这个基础之上。为了认真地在一个异步的世界中高效地编程，你将需要对 promise 与 generator 的组合十分适应。

如果 promise 和 generator 是关于表达一些模式，这些模式让你的程序更加并发地运行，而因此在更短的时间内完成更多的处理，那么 JS 在性能优化上就拥有许多其他的方面值得探索。

第五章钻研的话题是使用 Web Worker 的程序并行性和使用 SIMD 的数据并行性，以及像 ASM.js 这样的底层优化技术。第六章从正确的基准分析技术的角度来观察性能优化，包括什么样的性能值得关心而什么应当忽略。

高效地编写 JavaScript 意味着编写的代码可以突破这种限制壁垒：在范围广泛的浏览器和其他环境中动态运行。这需要我们进行更多复杂的详细计划与努力，才能使一个程序从“可以工作”到“工作得很好”。

给你编写合理且高效的 JavaScript 代码所需的全部工具与技能，*异步与性能*就是为此而设计的。

ES6 与未来

至此，无论你感觉自己已经将 JavaScript 掌握的多么好，现实是 JavaScript 从来没有停止过进化，而且进化的频率正在飞快地增长。这个事实几乎就是本系列精神的含义，拥抱我们永远不会完全 *懂得*的 JS 的所有部分，因为只要你掌握了它的全部，就会有你需要学习的新的东西到来。

这本书专注于这门语言在中短期的发展前景，不仅是像 ES6 这样 *已知的*东西，还包括在未来 *可能的*东西。

虽然这个系列的所有书目采纳的是在编写它们时 JavaScript 的状态，也就是 ES6 正在被接纳的半途中，但是这个系列更主要地集中于 ES5。现在我们想要将注意力转移到 ES6，ES7，和……

因为在编写本书时 ES6 已经近于完成，*ES6 与未来*首先将 ES6 中确定的东西分割为几个关键的范畴，包括新的语法，新的数据结构（集合），和新的处理能力以及 API。我们将在各种细节的层面讲解这些新的 ES6 特性中的每一个，包括复习我们在本系列的其他书目中遇到过的细节。

这是一些值得一读的激动人心的 ES6 特性：解构，参数默认值，symbol，简洁方法，计算属性，箭头函数，块儿作用域，promise，generator，iterator，模块，代理，weakmap，以及很多，很多别的东西！呼，ES6 真是不容小觑！

这本书的第一部分是一张路线图，为了对你将要在以后几年中编写和探索的新改进的 JavaScript 做好准备，它指明了你需要学习的所有东西。

这本书稍后的部分将注意力转向简要地介绍一些我们将在近未来可能看到的 JavaScript 的新东西。在这里最重要的是，要理解在后 ES6 时代，JS 很可能将会一个特性一个特性地进化，而不是一个版本一个版本地进化，这意味着我们将在比你想象的早得多的时候，看到这些近未来的到来。

JavaScript 的未来是光明的。这不正是我们开始学习它好时机吗!?

复习

YDKJS 系列投身于这样的命题：所有的 JS 开发者都可以，也应该学习这门伟大语言的每一部分。没有任何个人意见，没有任何框架的设想，没有任何项目的期限可以作为你从没有学习和深入理解 JavaScript 的借口。

我们聚焦这门语言中的每一个重要领域，为之专著一本很短但是内容非常稠密的书，来全面地探索它的 —— 你也许认为自己知道但可能并不全面 —— 所有部分。

“你不懂 JS” 不是一种批评或羞辱。它是我们所有人，包括我自己，都必须正视的一种现实。学习 JavaScript 不是一个最终目标，而是一个过程。我们还不懂 JavaScript。但是我们会的!

附录 A: 鸣谢

为了这本书和整个系列的诞生，我有很多人要感谢。

首先，我必须感谢我的妻子 Christen Simpson，和我的两个孩子 Ethan 和 Emily，忍受着老爹总是在电脑上敲打。即使在没有写书时，我对 JavaScript 的痴迷也将我的眼睛粘在屏幕上太久了。我从家庭那里借来的时间是这些书可以如此深入和完整地向你，读者，解释 JavaScript 的原因。我欠我的家庭一切。

我要感谢我在 O’Reilly 的编辑，他们是 Simon St.Laurent 和 Brian MacDonald，还有其他的编辑和市场员工。和他们一起工作很棒，而且在这种 “开源” 写作，编辑，和生产的实验期间提供了特别的通融。

感谢许多通过提供编辑意见和订正来参与使这部丛书变得更好的朋友们，他们是 Shelley Powers，Tim Ferro，Evan Borden，Forrest L. Norvell，Jennifer Davis，Jesse Harlin，Kris Kowal，Rick Waldron，Jordan Harband，Benjamin Gruenbaum，Vyacheslav Egorov，David Nolen，和许多其他人。一个巨大感谢送给 Jenn Lukas 为本书作序。

感谢社区中无数的朋友们，包括 TC39 协会的成员，他们和我们分享了那么多的知识，特别是以耐心和细节容忍我无休止的问题和探究。John-David Dalton，Juriy “kangax” Zaytsev，Mathias Bynens，Axel Rauschmayer，Nicholas Zakas，Angus Croll，Reginald Braithwaite，Dave Herman，Brendan Eich，Allen Wirfs-Brock，Bradley Meck，Domenic Denicola，David Walsh，Tim Disney，Peter van der Zee，Andrea Giammarchi，Kit Cambridge，Eric Elliott，和其他许多我甚至不能接触到的人。

你不懂 JS 系列丛书诞生于 Kickstarter，所以我也要感谢我的所有（将近）500 位慷慨的支持者，没有他们这部丛书不可能诞生：

Jan Szpila, nokiko, Murali Krishnamoorthy, Ryan Joy, Craig Patchett, pdqtrader, Dale Fukami, ray hatfield, R0drigo Perez [Mx], Dan Petitt, Jack Franklin, Andrew Berry, Brian Grinstead, Rob Sutherland, Sergi Meseguer, Phillip Gourley, Mark

Watson, Jeff Carouth, Alfredo Sumaran, Martin Sachse, Marcio Barrios, Dan, AimelyneM, Matt Sullivan, Delnatte Pierre-Antoine, Jake Smith, Eugen Tudorancea, Iris, David Trinh, simonstl, Ray Daly, Uros Gruber, Justin Myers, Shai Zonis, Mom & Dad, Devin Clark, Dennis Palmer, Brian Panahi Johnson, Josh Marshall, Marshall, Dennis Kerr, Matt Steele, Erik Slagter, Sacah, Justin Rainbow, Christian Nilsson, Delapouite, D.Pereira, Nicolas Hoizey, George V. Reilly, Dan Reeves, Bruno Laturner, Chad Jennings, Shane King, Jeremiah Lee Cohick, od3n, Stan Yamane, Marko Vucinic, Jim B, Stephen Collins, Ægir Þorsteinsson, Eric Pederson, Owain, Nathan Smith, Jeanetteurphy, Alexandre ELISÉ, Chris Peterson, Rik Watson, Luke Matthews, Justin Lowery, Morten Nielsen, Vernon Kesner, Chetan Shenoy, Paul Tregoing, Marc Grabanski, Dion Almaer, Andrew Sullivan, Keith Elsass, Tom Burke, Brian Ashenfelter, David Stuart, Karl Swedberg, Graeme, Brandon Hays, John Christopher, Gior, manoj reddy, Chad Smith, Jared Harbour, Minoru TODA, Chris Wigley, Daniel Mee, Mike, Handyface, Alex Jahraus, Carl Furrow, Rob Foulkrod, Max Shishkin, Leigh Penny Jr., Robert Ferguson, Mike van Hoenselaar, Hasse Schougaard, rajan venkataguru, Jeff Adams, Trae Robbins, Rolf Langenhuijzen, Jorge Antunes, Alex Koloskov, Hugh Greenish, Tim Jones, Jose Ochoa, Michael Brennan-White, Naga Harish Muvva, Barkóczi Dávid, Kitt Hodsden, Paul McGraw, Sascha Goldhofer, Andrew Metcalf, Markus Krogh, Michael Mathews, Matt Jared, Juanfran, Georgie Kirschner, Kenny Lee, Ted Zhang, Amit Pahwa, Inbal Sinai, Dan Raine, Schabse Laks, Michael Tervoort, Alexandre Abreu, Alan Joseph Williams, NicolasD, Cindy Wong, Reg Braithwaite, LocalPCGuy, Jon Friskics, Chris Merriman, John Pena, Jacob Katz, Sue Lockwood, Magnus Johansson, Jeremy Crapsey, Grzegorz Pawłowski, nico nuzzaci, Christine Wilks, Hans Bergren, charles montgomery, Ariel - Fogel, Ivan Kolev, Daniel Campos, Hugh Wood, Christian Bradford, Frédéric Harper, Ionuţ Dan Popa, Jeff Trimble, Rupert Wood, Trey Carrico, Pancho Lopez, Joël kuijten, Tom A Marra, Jeff Jewiss, Jacob Rios, Paolo Di Stefano, Soledad Penades, Chris Gerber, Andrey Dolganov, Wil Moore III, Thomas Martineau, Kareem, Ben Thouret, Udi Nir, Morgan Laupies, jory carson-burson, Nathan L Smith, Eric Damon Walters, Derry Lozano-Hoyland, Geoffrey Wiseman, mkeehner, KatieK, Scott MacFarlane, Brian LaShomb, Adrien Mas, christopher ross, Ian Littman, Dan Atkinson, Elliot Jobe, Nick Dozier, Peter Wooley, John Hoover, dan, Martin A. Jackson, Héctor Fernando Hurtado, andy ennamorato, Paul Seltmann, Melissa Gore, Dave Pollard, Jack Smith, Philip Da Silva, Guy Israeli, @megalithic, Damian Crawford, Felix Gliesche, April Carter Grant, Heidi, jim tierney, Andrea Giammarchi, Nico Vignola, Don Jones, Chris Hartjes, Alex Howes, john gibbon, David J. Groom, BBox, Yu 'Dilys' Sun, Nate Steiner,

Brandon Satrom, Brian Wyant, Wesley Hales, Ian Pouncey, Timothy Kevin Oxley, George Terezakis, sanjay raj, Jordan Harband, Marko McLion, Wolfgang Kaufmann, Pascal Peuckert, Dave Nugent, Markus Liebelt, Welling Guzman, Nick Cooley, Daniel Mesquita, Robert Syvarth, Chris Coyier, Rémy Bach, Adam Dougal, Alistair Duggin, David Loidolt, Ed Richer, Brian Chenault, GoldFire Studios, Carles Andrés, Carlos Cabo, Yuya Saito, roberto ricardo, Barnett Klane, Mike Moore, Kevin Marx, Justin Love, Joe Taylor, Paul Dijou, Michael Kohler, Rob Cassie, Mike Tierney, Cody Leroy Lindley, tofuji, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes, Rhys Brett-Bowen, Dmitry, Aziz Khoury, Dean, Scott Tolinski - Level Up, Clement Boirie, Djordje Lukic, Anton Kotenko, Rafael Corral, Philip Hurwitz, Jonathan Pidgeon, Jason Campbell, Joseph C., SwiftOne, Jan Hohner, Derick Bailey, getify, Daniel Cousineau, Chris Charlton, Eric Turner, David Turner, Joël Galeran, Dharma Vagabond, adam, Dirk van Bergen, dave furf, Vedran Zakanj, Ryan McAllen, Natalie Patrice Tucker, Eric J. Bivona, Adam Spooner, Aaron Cavano, Kelly Packer, Eric J, Martin Drenovac, Emilis, Michael Pelikan, Scott F. Walter, Josh Freeman, Brandon Hudgeons, vijay chennupati, Bill Glennon, Robin R., Troy Forster, otaku_coder, Brad, Scott, Frederick Ostrander, Adam Brill, Seb Flippence, Michael Anderson, Jacob, Adam Randlett, Standard, Joshua Clanton, Sebastian Kouba, Chris Deck, SwordFire, Hannes Papenberg, Richard Woeber, hnzz, Rob Crowther, Jedidiah Broadbent, Sergey Chernyshev, Jay-Ar Jamon, Ben Combee, luciano bonachela, Mark Tomlinson, Kit Cambridge, Michael Melgares, Jacob Adams, Adrian Bruinhout, Bev Wieber, Scott Puleo, Thomas Herzog, April Leone, Daniel Mizieliński, Kees van Ginkel, Jon Abrams, Erwin Heiser, Avi Laviad, David newell, Jean-Francois Turcot, Niko Roberts, Erik Dana, Charles Neill, Aaron Holmes, Grzegorz Ziółkowski, Nathan Youngman, Timothy, Jacob Mather, Michael Allan, Mohit Seth, Ryan Ewing, Benjamin Van Treese, Marcelo Santos, Denis Wolf, Phil Keys, Chris Yung, Timo Tijhof, Martin Lekvall, Agendine, Greg Whitworth, Helen Humphrey, Dougal Campbell, Johannes Harth, Bruno Girin, Brian Hough, Darren Newton, Craig McPheat, Olivier Tille, Dennis Roethig, Mathias Bynens, Brendan Stromberger, sundeep, John Meyer, Ron Male, John F Croston III, gigante, Carl Bergenhem, B.J. May, Rebekah Tyler, Ted Foxberry, Jordan Reese, Terry Suitor, afeliz, Tom Kiefer, Darragh Duffy, Kevin Vanderbeken, Andy Pearson, Simon Mac Donald, Abid Din, Chris Joel, Tomas Theunissen, David Dick, Paul Grock, Brandon Wood, John Weis, dgrebb, Nick Jenkins, Chuck Lane, Johnny Megahan, marzsman, Tatu Tamminen, Geoffrey Knauth, Alexander Tarmolov, Jeremy Tymes, Chad Auld, Sean Parmelee, Rob Staenke, Dan Bender, Yannick derwa, Joshua Jones, Geert Plaisier, Tom LeZotte, Christen Simpson, Stefan Bruvik, Justin Falcone, Car-

los Santana, Michael Weiss, Pablo Villoslada, Peter deHaan, Dimitris Iliopoulos, seyDoggy, Adam Jordens, Noah Kantrowitz, Amol M, Matthew Winnard, Dirk Ginader, Phinam Bui, David Rapson, Andrew Baxter, Florian Bougel, Michael George, Alban Escalier, Daniel Sellers, Sasha Rudan, John Green, Robert Kowalski, David I. Teixeira (@ditma, Charles Carpenter, Justin Yost, Sam S, Denis Ciccale, Kevin Sheurs, Yannick Croissant, Pau Fracés, Stephen McGowan, Shawn Searcy, Chris Ruppel, Kevin Lamping, Jessica Campbell, Christopher Schmitt, Sablons, Jonathan Reisdorf, Bunni Gek, Teddy Huff, Michael Mullany, Michael Fürstenberg, Carl Henderson, Rick Yoesting, Scott Nichols, Hernán Ciudad, Andrew Maier, Mike Stapp, Jesse Shawl, Sérgio Lopes, jsulak, Shawn Price, Joel Clermont, Chris Ridmann, Sean Timm, Jason Finch, Aiden Montgomery, Elijah Manor, Derek Gathright, Jesse Harlin, Dillon Curry, Courtney Myers, Diego Cadenas, Arne de Bree, João Paulo Dubas, James Taylor, Philipp Kraeutli, Mihai Păun, Sam Gharegozlou, joshjs, Matt Murchison, Eric Windham, Timo Behrmann, Andrew Hall, joshua price, Théophile Villard

这部丛书是以开源的风格书写的，包括编辑和生产。我们感激 GitHub 使这样的事情在社区中成为可能!

再次感谢所有无数的朋友，尽管我不能叫上名字但是我依然亏欠感谢。但愿这部丛书被我们所有人 “拥有”，并为增进对 JavaScript 语言的意识和理解做出贡献，成为当下和未来所有社区贡献者的助益。