华章科技

世界级JavaScript程序员力作，JavaScript之父Brendan Eich高度评价并强力推荐。

JavaScript编程原理与运用规则完美融合，读者将在游戏式开发中学会JavaScript程序设计，是系统学习JavaScript程序设计的首选之作。

华章程序员书库

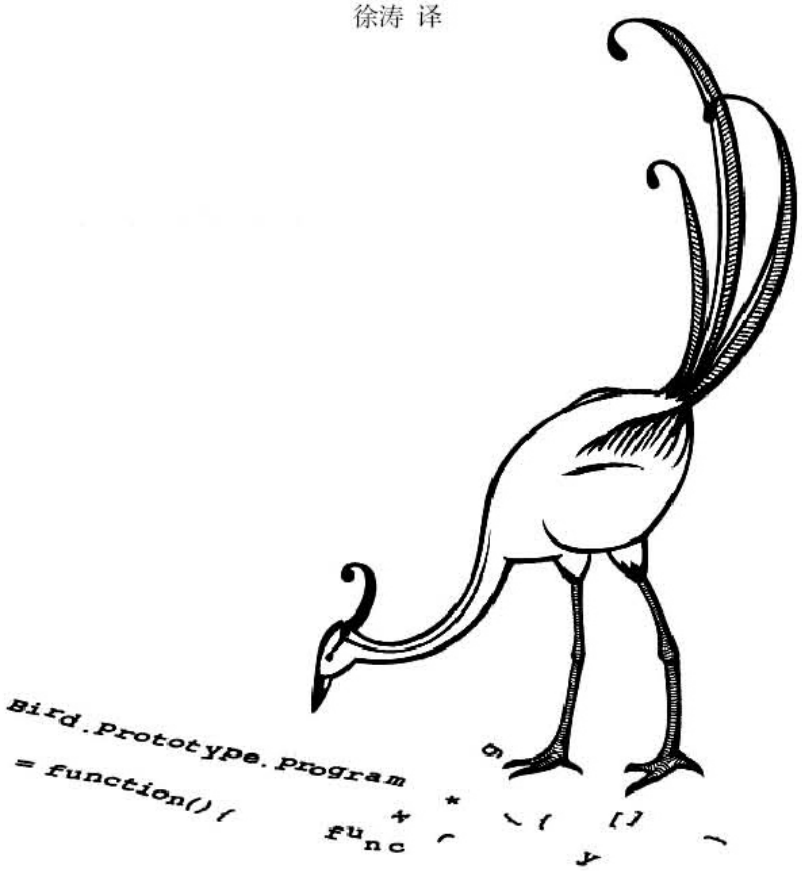
Eloquent JavaScript

A Modern Introduction to Programming

JavaScript编程精解

（美） Marijn Haverbeke 著

徐涛 译



机械工业出版社
China Machine Press

华章专业开发者丛书

JavaScript 编程精解

Eloquent JavaScript: A Modern Introduction to Programming

（美） Marijn Haverbeke 著
徐涛 译

机械工业出版社
China Machine Press

如果你只想阅读一本关于 JavaScript 的图书，那么本书应该是你的首选。本书由世界级 JavaScript 程序员撰写，JavaScript 之父和多位 JavaScript 专家鼎力推荐。本书适合作为系统学习 JavaScript 的参考书，它在写作思路上几乎与现有的所有同类书都不同，打破常规，将编程原理与运用规则完美地结合在一起，而且将所有知识点与一个又一个经典的编程故事融合在一起，读者可以在轻松的游戏式开发中学会 JavaScript 程序设计，趣味性十足，可操作性极强。

全书一共 12 章：第 1~3 章介绍了 JavaScript 的基本语法，旨在帮助读者编写出正确的 JavaScript 程序，包含数字、算术、字符串、变量、程序结构、控制流程、类型、函数、对象和数组等最基础和最核心的内容；第 4~7 章讲解了 JavaScript 编程中的高级技术，目的是帮助读者编写更复杂的 JavaScript 程序，主要涉及错误处理、函数式编程、面向对象编程、模块化等重要内容；第 8~12 章则将重心转移到 JavaScript 环境中可用的工具上，分别详细讲解了正则表达式、与 Web 编程相关的知识、文档对象模型、浏览器事件和 HTTP 请求等。

本书版权登记号：图字：01-2012-2098

图书在版编目（CIP）数据

JavaScript 编程精解 /（美）哈弗贝克（Haverbeke，M.）著；徐涛译 . —北京：机械工业出版社，2012.9
（华章专业开发者丛书）
书名原文：Eloquent JavaScript: A Modern Introduction to Programming

ISBN 978-7-111-39665-9

Ⅰ. J…　Ⅱ. ①哈…　②徐…　Ⅲ. JAVA 语言－程序设计　Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2012）第 210644 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）
责任编辑：秦　健
印刷
2012 年 10 月第 1 版第 1 次印刷
186mm×240mm • 11 印张
标准书号：ISBN 978-7-111-39665-9
定价：49.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

对本书的赞誉

编程原理与运用规则的简练、完美融合。我喜欢游戏式的程序开发教程。本书再次点燃了我学习编程的热情。对了，是 JavaScript！

——Brendan Eich，JavaScript 之父

因为这本书，我成为了更棒的架构师、作家、咨询师和开发者。

——Angus Croll，Twitter 开发者

如果你决定只买一本有关 JavaScript 的书，那么就应是 Marijn Haverbeke 的这本书。

——Joey deVilla，Global Nerdy

本书不仅是学习 JavaScript 最棒的教材之一，也是通过学习 JavaScript 进而学习现代编程的优秀图书。当有人问我如何学好 JavaScript 时，我会推荐这本书。

——Chris Williams，美国 JSConf 组织者

我读过的最棒的 JavaScript 书籍之一。

——Rey Bango，微软 Client-Web 社区项目经理和 jQuery 团队成员

这本书不仅是一本非常不错的 JavaScript 指导书，而且是一本很棒的编程指导书。

——Ben Nadel，Epicenter 咨询公司首席软件工程师

真是本好书。

——Design Shack

这本书对编程基本原理的详述以及对栈和环境等概念的解释非常到位。注重细节使本书从其他的 JavaScript 书中脱颖而出。

——Designorati

学习 JavaScript 的好书。

——Craig Buckler，OptimalWorks Web Design 公司

译 者 序

当我第一次阅读这本书的时候，就深深地喜欢上了这本书的写作风格。游戏式的章节、完整且连贯的故事，这些都使我在阅读过程中真正有了读书的快感。

与其他的 JavaScript 书籍不同，本书没有列表式的数据类型讲解，也没有枯燥的概念和老掉牙的例子，更没有流行的 Ajax 专题。本书通过设计一个个由浅入深的小游戏，让读者更加深入而轻松地学习如何应用 JavaScript 编程技术。因此，建议读者在阅读过程中，每次阅读一个完整章节，以便更好地理解编程故事的情节。

在翻译过程中，除了对 JavaScript 语言本身有了深刻理解之外，我也学到了如何从一个需求（游戏设计）进行分析，进而细化到可编程的 JavaScript 代码部分，尤其是本书中虚拟生态圈游戏的设计，使我真正体会到了 JavaScript 原来还可以这么做。

我非常荣幸能够参与本书的翻译工作，感谢机械工业出版社的编辑在翻译过程中给予的鼓励和信任。与此同时，也要感谢我的家人在翻译过程中对我的支持和理解，尤其是我的爱人对本书进行了无数次的阅读，并给出了大量的修改建议。

由于译者水平有限，翻译错误或者风格不合口味在所难免，对你造成的阅读上的不便我深表歉意。

针对本书的任何意见你都可以在我的博客上（http://www.cnblogs.com/TomXu），或者通过邮件直接发给我，我的邮件地址是 taoxu@live.com。

感谢并期待你的批评和指正！

前　言

20 世纪 70 年代，业界首次推出个人计算机时，大多数计算机都内置一种简单的编程语言——通常是 BASIC 的变体，人与计算机之间的互动需要通过这种语言实现。这意味着，对天生喜欢钻研技术的人来说，从单纯使用计算机到编程的过渡非常容易。

现在的计算机相比 20 世纪 70 年代的功能更加强大，价格也更加便宜，软件接口呈现的是使用鼠标操作的灵活图形界面，而不是语言界面。这使计算机更容易使用，总的来说，这是一个巨大的进步。然而，这也在计算机用户与编程世界之间制造了一个障碍——业余爱好者必须积极**寻找**自己的编程环境，而不是一打开电脑就呈现的环境。

实质上，计算机系统仍然被各种编程语言控制。大多数的编程语言都比早期个人计算机中的 BASIC 语言更加先进。例如，本书的主题——JavaScript 语言，就存在于每一款主流 Web 浏览器中。

关于编程

不愤不启，不悱不发。举一隅不以三隅反，则不复也。

——孔子

本书除了介绍 JavaScript 外，也致力于介绍编程的基本原理。事实上，这种编程还是比较难的。编程的基本规则通常都简单明了，但计算机程序构建在这些基本规则之上后，会变得很复杂，产生了其自身的规则和复杂性。正因为如此，编程并不是那么简单或可预测的。正如计算机科学的鼻祖高德纳（Donald Knuth）所说，编程是一门**艺术**，而不是一门科学。

要想从本书里获取最大收获，不能仅仅依靠被动阅读。一定要集中注意力去理解示例代码，只有确定自己真正理解了前面的内容，才能继续往下阅读。

程序员对其创造的宇宙负全部责任，因为他们是创造者。以计算机程序的形式，可创造出无限复杂的宇宙。

——Joseph Weizenbaum，《Computer Power and Human Reason》

一个程序包含很多含义。它是程序员敲出的一串字符，是计算机运行的指向力，是计算机内存中的数据，还控制同一个内存上的执行动作。仅使用熟悉的类推法比较程序与对象往往还不够，因为从表面上看适合该操作的是机器。机械表的齿轮巧妙地啮合在一起，如果表的制造者技术很棒，它就能够连续多年准确地显示时间。计算机程序的元素也以类似的方式组合在一起，如果程序员知道自己在做什么，那么这个程序就能够正常运行而不会崩溃。

计算机作为这些无形机器的载体而存在。计算机本身只会做简单直接的工作。它们之所以

如此有用，是因为它们能够以惊人的速度完成这些工作。程序可以巧妙地把许多简单动作结合起来，去完成非常复杂的工作。

对有些人来说，编写计算机程序是一种很有趣的游戏。程序是思想的构筑，它零成本、零重量，在我们的敲打中不断发展。如果我们不细心，它的规模和复杂性将失去控制，甚至创造者也会感到混乱。这就是编程的主要问题：控制好程序。程序在工作时是很不可思议的，编程的艺术就是控制复杂性的技巧，好的程序其复杂性也会降低。

如今，很多程序员认为只要在程序中使用少量易于理解的技术，就可以最有效地降低复杂性。他们制定了严格的编程规则（**最佳实践**）及书写格式，那些破坏规则的人被称为“差劲”的程序员。

丰富多彩的编程世界里包含了太多的复杂性！让我们努力将程序变得简单和可预测，并为所有奇妙和优美的程序制定禁忌规则。编程技术的前景是广阔的，其多样性使人着迷，它的世界仍有很多未被探索的部分。编程过程中有很多陷阱和圈套，缺乏经验的程序员会犯各类糟糕的错误，告诫我们需要谨慎，并保持头脑清醒。学习编程时总是需要探索新的挑战、新的领域，拒绝不断探索的程序员必定会停滞不前、忘记编程的快乐、并失去编程的意志（或成为管理人员）。

语言为何很重要

在计算机诞生初期并没有编程语言。程序看起来就像这样：

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

这是一个从 1 加到 10 并输出结果（1 + 2 +…+ 10 = 55）的程序。它可以在一个非常简单、理想化的计算机上运行。为早期的计算机编制程序时，必须在正确的位置设置一排排的开关或者在纸带上打上一系列有规律的孔点，这样才能将程序传递给计算机。可以想象这个过程有多么繁琐和易出错。即使编写简单的程序也需要使用很多脑力和规则，编写复杂的程序更是不可想象。

当然，手动输入这些二进制**位**（即以上这些 1 和 0 的统称）的神秘组合，让程序员感觉自己像巫师一样拥有强大的魔力，而且还能够获得工作满足感，因此这点还是很值得的。

程序的每一行都包含一条单独的指令。可以用语言这样描述：

1）将数字 0 保存在第 0 个存储单元；

2）将数字 1 保存在第 1 个存储单元；

3）将第 1 个存储单元的值保存在第 2 个存储单元；

4）将第 2 个存储单元中的值减去数字 11 ；

5）如果第 2 个存储单元中的值是数字 0，则继续执行指令 9 ；

6）将第 1 个存储单元的值添加至第 0 个存储单元 ；

7）将数字 1 添加至第 1 个存储单元 ；

8）继续执行指令 3 ；

9）输出第 0 个存储单元的值。

虽然这比二进制位易读，但仍然令人不快。用名称代替指令和存储单元的数字或许更有帮助。

```
 Set 'total' to 0
 Set 'count' to 1
[loop]
 Set 'compare' to 'count'
 Subtract 11 from 'compare'
 If 'compare' is zero, continue at [end]
 Add 'count' to 'total'
 Add 1 to 'count'
 Continue at [loop]
[end]
 Output 'total'
```

在这里不难看出程序是如何运行的。前两行代码为两个存储单元赋予初始值 ：total 用于创建计算的结果，count 记录当前看到的数字。使用 compare 的行可能是最令人费解的地方。该程序的目的是判断 count 是否等于 11，从而确定能否停止运行。由于该机器相当原始，它只能测试一个数字是否为零，并在此基础上做出判断（跳转），因此它使用标记 compare 的存储单元来计算 count 的值，即 11，并在该值的基础上做出判断。后面两行代码将 count 的值添加到结果 total 上，当程序判断 count 不是 11 时，为 count 加 1。

下面是用 JavaScript 编写的有同样效果的程序。

```
var total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
print(total);
```

这段程序有了更多的改进。最重要的是，不再需要指定程序来回转换的方式，while 这个神奇的单词会帮助解决这个问题。只要满足给定的条件 ：count <= 10（意思是“count 小于或等于 10”），它就会继续执行下面几行代码。不再需要创建临时的值并将该值与零比较——这是一个没有意义的细节，编程语言就是用于解决这些无意义的细节。

最后，如果我们可以使用方便的 range 和 sum 操作（分别在区间范围（range）里创建一组数字，并计算该组数字的总和（sum）），代码应该是这样的 ：

```
print(sum(range(1, 10)));
```

从上例可以看出，同样的程序可以使用长或短、不可读或可读的方式来表达。该程序的第一个版本非常晦涩，而最后一个版本基本上都是语言描述：打印（print）出从 1 到 10 这个区间范围（range）的总和（sum）。（我们将在后面的章节中讲述如何创建 sum 及 range 等函数。）

优秀的编程语言会提供一种更为抽象的表达方式来帮助程序员表达其意图。这种语言隐藏无意义的细节，提供方便的程序块（例如 while 语句），在很多时候，它允许程序员自己添加程序块（例如 sum 和 range 操作）。

什么是 JavaScript

JavaScript 语言目前主要用于解决万维网页面的各种难题。近年来，该语言也开始应用于其他环境中，如 node.js 框架（一种使用 JavaScript 语言编写快速服务器端程序的方式），最近吸引了不少关注。如果对编程感兴趣，JavaScript 绝对是值得学习的语言。即使以后不会从事大量的网络编程工作，本书中展示的一些程序也会一直伴随着你，影响你使用其他语言编写的程序。

有些人指出了 JavaScript 语言不好的地方，其中很多观点都是对的。当我第一次用 JavaScript 写程序时，我立刻就开始轻视这种语言了，它接受我输入的大部分代码，但其解释代码的方式与我的意图完全不同。无可否认，这与我做事没有头绪有很大的关系，但也存在着一个真正的问题：JavaScript 允许的操作实在是太多了。这种设计的初衷是让初学者更易掌握 JavaScript 编程方法。实际上，这使得更难发现程序中的问题，因为系统不会指出问题所在。

然而，语言的灵活性也是一种优势。它为很多无法使用更强大语言的技术留下了发展空间，正如我们将在后面几章中看到的，它能够克服 JavaScript 的一些缺点。当正确学习 JavaScript 并使用了一段时间之后，我发现自己真正的喜欢上了这种语言。

与这种语言的名字所暗示的不同，JavaScript 与 Java 编程语言并没有多大关系。相似的名字是基于营销的考虑，而不是为了获得好评。网景公司于 1995 年推出 JavaScript 时，Java 语言正在极力推广中，并且很受欢迎。很显然，当时有人认为借助 Java 语言的成功进行营销是个好主意。于是，就有了现在看到的这个名字。

与 JavaScript 有关的是 ECMAScript。当网景浏览器以外的浏览器开始支持 JavaScript 或类似语言时，出现了一份准确描述该语言应该如何工作的文档。此文档里所描述的语言称为 ECMAScript，它是以制定该设计语言标准的欧洲计算机制造商协会（ECMA）的名称命名的。ECMAScript 描述了一种多用途编程语言，但并没有提及它与 Web 浏览器的集成。

现在有多个版本的 JavaScript。本书讲的是 ECMAScript 3 版本，是第一个各种浏览器都支持的版本。在过去的几年中，人们进一步发展这种语言，但是，这些扩展版本只有受到浏览器的广泛支持才是有用的（至少在网络编程方面），而浏览器要跟上编程语言的发展也需要很长一段时间。幸运的是，新版本的 JavaScript 基本上是 ECMAScript 3 的扩展版本，因此本书中的大部分内容都不会过时。

试运行程序

如果想要执行并练习本书中的代码，一种方法是可以登录 http://eloquentjavascript.net/，使用该网站提供的工具。

另一种方法是创建一个包含该程序的 HTML 文件，并将其加载到浏览器中。例如，可以创建一个名为 test.html 的文件，内容如下。

```
<html><body><script type="text/javascript">

var total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
document.write(total);

</script></body></html>
```

后面的章节将讲述更多有关 HTML 的知识以及浏览器解释 HTML 的方式。请注意，示例中的 print 操作已被替换为 document.write。我们将在第 10 章中介绍如何创建 print 函数。

本书主要内容

本书前 3 章介绍了 JavaScript 语言，并指导如何编写语法正确的 JavaScript 程序，还介绍了控制结构（如在前面看到的 while 关键词）、函数（自己编写的操作）和数据结构。这些内容可以指导编写简单的程序。

在基本了解了编程的基础上，后面 4 章将讨论更高级的技术，指导如何编写更复杂的程序，而不是将程序写得难以理解。第 4 章将讨论如何处理错误和意想不到的情况。第 5 章和第 6 章将介绍两个主要的抽象方式：函数式编程和面向对象编程。第 7 章给出了一些如何组织程序的指导。

其余的章节重点关注了 JavaScript 环境中可用的工具，理论部分相对较少。第 8 章介绍了一种文字处理的子语言，第 9~12 章将讲述程序在浏览器内部运行时可利用的工具，教会大家控制网页、应对用户的操作，并与 Web 服务器通信。

排版规范

在本书中，采用等宽字体的文本应被理解为代表程序的元素，这些元素有时是各自独立的片段，有时只是指靠近程序的一部分。这段我们已经看过的程序的一部分，可写成如下形式：

```
function fac(n) {
  return n == 0 ? 1 : n * fac(n - 1);
}
```

有时，为了演示某些表达式求值时会发生的情况，这些表达式会采用黑体字，其产生的值写

X

在下面，前面有一个箭头：

```
1 + 1
→ 2
```

更新

访问 http://nostarch.com/ejs.htm 获得更新、勘误表以及运行本书示例代码的交互式代码沙盒。

目　录

第 1 章

JavaScript 基础：值、变量、控制流程

计算机世界里只有数据，没有数据计算机就不存在。所有数据实质上都是由 bit 序列构成的，因此基本上都是相似的。bit 序列通常是由 0 和 1 两种数字排列组合而成的，它们在计算机内的形式就如一个高电荷或一个低电荷、一个强信号或一个弱信号，或光盘表面的一个亮点或一个暗点。

1.1 值

虽然构成相同，但每一部分数据都扮演着自己的角色。在 JavaScript 系统中，大多数数据都被有序地分成了各种**值**。每个值都有一个类型，用于确定它扮演的角色类型。JavaScript 里有 6 种基本类型的值：number、string、Boolean、object、function 和 undefined。

创建值的时候，只需调用它的名称即可，非常方便。无需为创建的值收集构建素材或是支付费用，只要调用某个值，便可立即获得该值。当然，值也不是凭空创建的，每个值都需要存储在某个地方，如果在同一时间使用大量的值，就有可能耗尽内存，幸运的是，只有在同时使用大量数据的时候才会出现这个问题。一旦不再需要这个值，它将会消失，只剩下一些 bit 数据，用于再次生成值。

1.1.1 数字

number 类型的值就是数字值，它们通常写成如下这样的数字：

```
144
```

将 144 输入程序，144 就会在计算机内部存储起来。在 bit 里如下面所示存放 144：

```
01000000011000100000000000000000000000000000000000000000000000000
```

如果读者认为像 10010000（144 的整数表示法）这样的存放方式还不错，在某些情况下可能确实需要采用这样的表示方法，但标准的 JavaScript 数字描述是 64 位的浮点型值。因此，这些值也可以包括分数和指数。

但是，本书不会深入研究二进制表示法，我们更感兴趣的是这种表示法对数字产生的实际影响。首先，数字表示实际上是有 bit 数量限制的，也就是有精度限制的。64 个 1 或 0 的值只能表

示 2^{64} 个数字。这个数字已经很大了，超过了 10^{19}。

能够在 JavaScript 中使用的数字并非所有小于 10^{19} 的正整数，还包括负数，因此需要使用其中一个 bit 来存储数字符号。更大的难题在于：必须将非整数表示出来。为此，还需要使用 11 个 bit 来存储数字的小数点位置。

这样就剩余 52 个 bit[⊖]，任何小于 2^{52} 的整数（大于 10^{15}）都可以安全地写成 JavaScript 数字。大多数情况下使用的数字是小于该数值的。

使用点（.）来写入小数：

```
9.81
```

对于任何非常大或者非常小的数字，可以添加一个 e，用科学计数法来表示该数值，e 后面跟的是数字的指数。

```
2.998e8
```

这里表示 2.998×10^8=299800000。

用于整数计算时整数（integer）存放在 52 个 bit 内，计算结果通常十分精确，但小数计算的精确度一般不高。例如 π 无法通过有限的小数数字来精确表示，当只有 64 个 bit 用于存放数字时，很多数字的精度就会降低。这是件很糟糕的事情，不过只有在极特殊情况下才会出现这样的问题。了解这一点很重要，因此应将小数视为近似值而不是精确值。

1.1.2 算术

与数字密切相关的就是算术，加法或乘法等算术运算需要使用两个数字，并利用它们产生一个新数字。JavaScript 的算术运算如下所示：

```
100 + 4 * 11
```

符号“+”和“*”被称为**运算符**（操作符），第一个符号代表加法，第二个符号代表乘法。在两个数字之间加上运算符后，这两个数字将应用该运算符计算出一个新数字。

上面的示例是说“4 加 100 以后将结果和 11 相乘”，还是在加法之前先做乘法？正如我们猜到的，先做乘法。但在数学运算上，可以用括号把加法括起来改变运算顺序。

```
(100 + 4) * 11
```

减法使用“−”运算符，除法使用“/”运算符。如果出现多个运算符，而又没有括号，运算的顺序是按照运算符的优先级来确定的。在上面第一个示例中，乘法的优先级高于加法。除法的优先级与乘法的优先级相同，加减法的优先级也相同。如果同时出现多个同优先级的运算符（如 1−2 + 1），应按照从左到右的顺序运算。

不需过多考虑这些优先级规则，如果不确定时，就使用括号。

有一个运算符可能不太熟悉，“%”符号表示**余数**，X%Y 表示 X 除以 Y 的余数。例如，314 % 100 的结果是 14，10 % 3 的结果是 1，144 % 12 的结果是 0。% 运算符的优先级与乘除法相同。

⊖ 实际上是 53 个 bit，因为有一个方法可以免费获取一个 bit，详情可以查看 IEEE 754 格式。

1.1.3 字符串

另外一个数据类型是字符串（string），它不像数字那样从名称就能明显看出其用途，但也起到非常基础的作用。string 用于表示文本（其名称可能是因它能够串起一堆字符而得的）。string 的书写方式是用引号将内容括起来。

```
"Patch my boat with chewing gum."
'You ain\'t never seen a donkey fly!'
```

单引号和双引号均可以用来标记 string，只要引号前后一致就可以。

几乎任何字符都可以放在引号里，JavaScript 会将它解析成 string。但有些字符放在引号里就比较复杂，例如，在引号里放引号就是很有难度的。按下回车键产生的新行也不能放在引号里，string 只能放在一行里。

要将这些特殊字符放在 string 里，需要遵守下列规则：当在加引号的文字里发现反斜杠（\），那就意味着其后面的字符有特殊意义。反斜杠后加引号（\”）并不意味着字符串的结束，而是字符串的一部分。反斜杠后面加 n（\n）表示一个新行，反斜杠后面加 t（\t）表示一个 tab 字符。可以参考下面的示例。

```
"This is the first line\nAnd this is the second"
```

其中包含的真正文字是：

```
This is the first line
And this is the second
```

还有一些情况就是，我们希望字符串的反斜杠只是代表反斜杠，而不是一个特殊的代码。如果两个连续的“\”出现在字符串里，输出的结果里只会有一个“\”。如下就是字符串 A newline character is written like"\n" 的写法：

```
"A newline character is written like \"\\n\"."
```

字符串不能相除、相乘或相减，但可以使用“+”运算符，“+”不表示相加，而表示连接——将两个字符串连接在一起。下面的代码行将产生字符串“concatenate”。

```
"con" + "cat" + "e" + "nate"
```

还有更多操作字符串的方式，稍后讨论。

1.1.4 一元操作符

并不是所有的运算符都是符号，有些运算符是单词。例如 typeof 运算符，它产生一个字符串值，该字符串命名给定值的类型。

```
typeof 4.5
→ "number"
typeof "x"
→ "string"
```

其他运算符都是对两个值执行运算，而 typeof 只对一个值执行运算。使用两个值的运算符称为**二元运算符**，而使用一个值的运算符则称为**一元运算符**。减法“-”运算符可同时用作上述两种运算符。

```
- (10 - 2)
→ -8
```

1.1.5 布尔值、比较和布尔逻辑

接下来了解**布尔类型**的值，布尔类型只有两个值：true 和 false。以下是产生这两个值的方式。

```
3 > 2
→ true
3 < 2
→ false
```

之前已经看到“>”和“<”符号了，它们的意思分别是：大于和小于。“>”和“<”是二元运算符，应用该运算符的结果是布尔值，表示在此情况下运算符产生的是否为 true。

字符串也可以用同样的方式进行比较。

```
"Aardvark" < "Zoroaster"
→ true
```

字符串基本是按照字母顺序来进行比较的。大写字母始终小于小写字母，所以“Z”<“a”是 true，非字母字符（!、@ 等）也包括在此排序中。比较这些字符的实际方式是基于 Unicode 标准。该标准为每个字符赋予一个数字（包括希腊语、阿拉伯语、日语、泰米尔语等字符）。使用数字对应字符主要用于在计算机中存储字符串——可以将字符表示成一堆数字。比较字符串时，JavaScript 实际上是从左向右逐个比较每个字符所对应的数字代码。

其他类似的运算符还有 >=（大于或等于）、<=（小于或等于）、==（等于）和 !=（不等于）。

```
"Itchy" != "Scratchy"
→ true
```

还有一些运算符可以应用于布尔值本身，JavaScript 支持三种逻辑运算符：与、或、非。它们可以用于推理布尔值。

&& 运算符表示逻辑**与**，它是一个二元运算符，只有在两个值都是 true 的情况下其结果才是 true。

```
true && false
→ false
true && true
→ true
```

|| 运算符是逻辑**或**，如果两个值的任何一个值为 true，其结果就为 true。

```
false || true
→ true
false || false
→ false
```

非运算符用感叹号“!”表示，它是一元运算符，能够反转给定的值。!true 的结果是 false，

而 ! false 的结果是 true。

当布尔运算符与算术以及其他运算符混合使用时，使用括号后的运算顺序并不是很明显。在实践中，了解了目前为止所见到的运算符的用法，就可以理解运算符的意义。| | 的优先级最低，其次是 &&，然后是比较运算符（>、== 等），最后是其他的运算符。在典型的编程情况下，选择运算符时应尽可能少用括号。

1.1.6 表达式与语句

到目前为止，所有示例使用的语言都好像在使用一个袖珍计算器：将一些值通过应用运算符得到新的值。像这样创建值是每个 JavaScript 程序都有的基本内容，但仅仅是一部分。一段创建值的代码称为表达式。每一个直接写出来的值（数字“22”或者字符“psychoanalysis”）都是一个**表达式**，括号之间的表达式也是表达式。将二元运算符应用于两个表达式或者将一元运算符应用于一个表达式，这些也都是表达式。利用这些规则，可以建立任意大小和难度的表达式。（JavaScript 实际上有更多建立表达式的方式，在恰当的时候会揭示这些方式。）

有一个比表达式更大的单位，称为**语句**。一个程序是由一组语句构成的，大多数语句都是以分号（;）结束，最简单的语句是由一个表达式及其后面的分号构成。

```
1;
!false;
```

这是一个没有任何意义的程序。一个表达式可仅用于创建一个值，但一个语句只有发挥改变“世界”的作用，才能算是成功的语句。一个语句可以将某些信息输出到屏幕上（这就是其改变“世界”之举），或者改变程序的内部状态以此来影响后面的语句。这些改变称为**副作用**（side effect），上面示例中的语句仅仅是用于创建两个值——1 和 true，然后立即又将它们扔进 bit 库，一点影响“世界”的痕迹也没有，因此它不是一个副作用。

在某些情况下，JavaScript 允许省略语句后面的分号；而在其他情况下，则必须有分号，否则就会发生异常。关于何时能够安全省略分号的规则十分复杂，其基本思想是：如果一个程序本来是无效的，但是插入分号后就变得有效了，那么该程序能否正常运行就要看它是否有分号。本书中的每个语句后面都用分号结束，强烈建议在程序里也这么做。

1.2 变量

程序如何保持内部状态？又如何存储东西？我们已经了解了如何用旧值产生新值，但这并没有改变旧值，新值必须立即使用，否则就会再次丢失。为了捕获并且拥有这些值，JavaScript 提供了一种功能叫做**变量**。

```
var caught = 5 * 5;
```

变量通常都是有名称的，并且指向一个值（也就是拥有它）。上面的语句表示创建一个叫 caught 的变量，然后指向 5×5 的结果值（也就是 25）上。

声明变量之后，变量的名称可作为一个表达式，用于产生其拥有的值。下面是一个示例。

```
var ten = 10;
ten * ten;
→ 100
```

单词 var 用于创建一个新的变量，var 之后是变量的名称。除了空格之外，几乎任何单词都可以作为变量名。数字也可以作为变量名的一部分（catch22 就是一个合法的名称），但变量不能以数字开头。字符“$”和“_”也可作为字母用于变量名中，因此 $_$ 也是一个合法的变量名。

如果想在声明一个变量之后立即获取一个值（大部分情况都是这样的），可以使用“=”操作符将某些表达式的值赋给这个变量。

当一个变量指向一个值时，并不意味着它永远都是这个值。“=”操作符在任何时候都可以将现有的变量指向新值。

```
caught;
→ 25
caught = 4 * 4;
caught;
→ 16;
```

应该将变量想象成触手而不是盒子，它不“容纳”值，而是“抓取”值——两个变量可以引用同一个值。只有受程序支配的值才能被变量访问。当需要记住某些值的时候，就需要多出一支触手来抓取该值，或者在原有触手的基础上重新接上一支来抓取新的值。

例如，要记住 Luigi 还欠你多少美元，为此创建一个变量，当他还你 35 美元的时候，为该变量赋一个新值。

```
var luigisDebt = 140;
luigisDebt = luigisDebt - 35;
luigisDebt;
→ 105
```


关键字与保留字

请注意，有些名称有特殊的意义，比如 var 不能用作变量名。这些名称称为**关键字**。还有一些单词是在 JavaScript 未来版本里使用的“保留字”。还有一些官方规定的不允许声明为变量名的关键字（尽管某些浏览器允许其运行），这张列表相当长：

```
abstract boolean break byte case catch char class const continue debugger
default delete do double else enum export extends false final finally float
for function goto if implements import in instanceof int interface long native
new null package private protected public return short static super switch
synchronized this throw throws transient true try typeof var void volatile
while with
```

不用记住这些关键字，但要记住如果某些代码不能按预期运行，问题可能就出在关键字上。char（保存一个字符的字符串）和 class 是最常见的，只在 JavaScript 中偶尔使用。

1.3 环境

在给定时间存在的变量和变量值的集合叫做**环境**。当程序开始运行的时候，环境并不是空的，它通常包含一些标准的变量。当浏览器加载一个页面时，它创建一个新环境，并将这些标准的变量存于新环境中。在当前页面中，程序创建和修改的变量直到浏览器打开新页面才会消失。

1.3.1 函数

标准环境提供的很多值都是**函数**类型的，函数是包含在值中的一段程序。通常情况下，这段程序会执行一些有用的操作，使用包含的函数值调用这段程序。在浏览器环境中，变量 alert 拥有一个函数，用于弹出带有消息的小对话框，使用方法如下：

```
alert("Good morning!");
```

执行函数里的代码称为**调用**或**应用**，完成这一过程所使用的符号是括号。生成函数值的每个表达式都可以通过在后面添加括号来执行，尽管通常会直接引用包含该函数的变量。在上述代码中，括号中的字符串传递给了 alert 函数，作为文本用于显示在对话框上。给函数传递的值称为**形参**或**实参**，alert 只需要其中一个参数，但其他函数则有可能需要不同数量或不同形式的参数。

弹出对话框是一个副作用，很多函数由于产生了副作用而变得非常有用。另外，函数也可以产生值，这种情况下它就不需要副作用了。例如，函数 Math.max 有两个参数，它返回两个给定数值中较大的值。

```
Math.max(2, 4);
→ 4
```

当函数产生一个值的时候也称为**返回**一个值。因为使用 JavaScript 时，值都是由表达式产生的，所以函数调用也可以作为较长表达式的一部分。

```
Math.min(2, 4) + 100;
→ 102
```

第 2 章将讨论如何编写我们自己的函数。

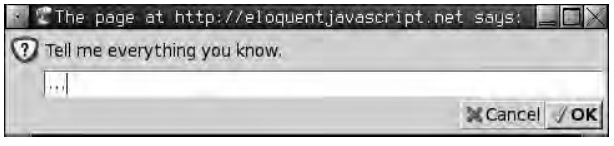
1.3.2 prompt 和 confirm

浏览器提供的标准环境包含了更多用于弹出窗口的函数，可以使用 confirm 函数让用户选择 OK/Cancel 问题。该函数返回布尔值：如果用户单击 OK，则返回 true；如果单击 Cancel，则返回 false。

```
confirm("Shall we, then?");
```

prompt 函数可用于询问一个开放式问题，第一个参数就是该问题；第二个参数是用户需要输入文本的开头部分，可以在对话窗口里输入一行文本，该函数会将其返回作为一个字符串。

```
prompt("Tell me everything you know.", "...");
```



1.3.3 print 函数

如前所述，浏览器提供了 alert 函数用于在小窗口上显示字符串。在这样做测试代码时很有帮助，但每次都单击这些小窗口也会让人感到厌烦，在本书中，假设有一个现成的函数 print，它能够将参数显示在未指定的文本输出设备上，方便编写代码实例。但要注意，Web 浏览器提供的 JavaScript 上下文环境里并不包含该函数。

例如，下面的代码将输入字母 X：

```
print("X");
```

1.3.4 修改环境

在该练习环境中，理论上可以给任何变量赋一个新值，这样做很有用但也很危险。如果给 print 函数赋值为 8，它就不再是一个函数，而且也不能再用于显示信息。第 7 章将讨论如何避免发生重新定义变量的偶然事件。

1.4 程序结构

一行代码的程序意义不大。在程序里编写多行语句时，这些语句都会像预想的一样从上到下逐条执行。下面的程序包含两个语句，第一个语句是让用户输入一个数字，第二个语句显示这个数字的平方值。

```
var theNumber = Number(prompt("Pick a number", ""));
alert("Your number is the square root of " + (theNumber * theNumber));
```

Number 函数将用户输入的值转化成数字（此例中需要使用 Number 函数，是因为 prompt 函数的结果是一个字符串值）。类似的函数还有 String 和 Boolean，它们也都是将值转换成相应的类型。

1.4.1 条件执行

有时并不希望程序里所有的语句都按照同样的顺序执行。例如，在上面的程序中，只想在输入的内容是数字的情况下才计算平方值。

关键字 if 可以根据布尔表达式的值执行或跳过执行语句，可以这样编写：

```
var theNumber = Number(prompt("Pick a number", ""));
if (!isNaN(theNumber))
  alert("Your number is the square root of " + (theNumber * theNumber));
```

if 后面的括号里提供的是条件表达式（本例中是 !isNaN(theNumber)），只有在该表达式的返回值是 true 的情况下，if 后面的语句才会执行。

如果调用 Number 函数传入的参数不是一个数字（如“moo”），则返回的结果是一个特殊值 NaN，用于代替“该结果不是数字”。函数 isNaN 用于判断传入的参数是否为 NaN，因此，如果 theNumber 是数字，则 !isNaN(theNumber) 的值就是 true。

通常情况下，不仅要在条件成立的情况下执行代码，也要在该条件不成立的情况下执行代码。可以将关键字 else 和 if 一起使用，分成两部分执行代码（并联路径）。

```
if (true == false)
  print("How confusing!");
else
  print("True still isn't false.");
```

如果要用多条路径分支，可以多次使用 if/else 对“连”在一起，示例如下。

```
var num = prompt("Pick a number:", "0");

if (num < 10)
  print("Small");
else if (num < 100)
  print("Medium");
else
  print("Large");
```

该程序首先会判断 num 是否小于 10，如果小于 10，则执行输出“small”的分支，然后结束；如果不小于 10，则执行包含第二个 if 的分支。如果第二个条件（小于 100）成立，则表示该数字在 10 和 100 之间，输出“Medium”; 如果不成立，则会执行第二个（即最后一个）else 分支。

1.4.2 while 循环与 do 循环

编写一个程序输出 0 到 12 之间所有的偶数。其中一种编写方式如下所示：

```
print(0);
print(2);
print(4);
print(6);
```

```
print(8);
print(10);
print(12);
```

上面的代码是有效的，但编程的目的是**减少**工作，而不是增加工作。如果要打印小于 1000 的所有偶数，上面的代码就不起作用了。编程就是要让一些代码自动重复执行。

```
var currentNumber = 0;
while (currentNumber <= 12) {
  print(currentNumber);
  currentNumber = currentNumber + 2;
}
```

while 开头的语句创建了一个**循环**，循环很像一个条件语句，影响着语句的顺序——它不是不执行语句或仅执行一次语句，而是有可能让这些语句重复执行多次。while 后面的括号里是表达式，用于判断 loop 循环是继续还是结束。只要该表达式产生的布尔值是 true，循环里的代码就会重复执行；如果是 false，程序就会跳到循环的底部，继续执行其他语句。

变量 currentNumber 用于演示变量跟踪程序运行过程的方式。循环每重复一次，变量 currentNumber 的值就会增加 2。每次重复循环开始时，都会和 12 进行比较，判断程序运行是否全部完成。

while 语句的第三部分称为**循环体**，是重复执行的行为代码。如果不使用 print 函数输出数字，程序就如下所示：

```
var currentNumber = 0;
while (currentNumber <= 12)
  currentNumber = currentNumber + 2;
```

此处，currentNumber = currentNumber + 2; 是组成循环体的语句。由于必须输出这个数字，因此循环语句需要有多个，用大括号（{}）将一组语句放到同一个代码**块**中。对于代码块外面的环境来说，这个代码块就相当于一个语句。在本例中，代码块包括 print 输出语句和 currentNumber 更新语句。

编写一个用于计算和显示 2^{10} 的程序作为一个有效的程序示例。可以使用两个变量：一个用于记录结果，另一个用于记录乘了多少次 2。循环判断第二个变量是否达到 10，如果没有达到，则更新这两个变量。

```
var result = 1;
var counter = 0;
while (counter < 10) {
  result = result * 2;
  counter = counter + 1;
}
result;
→ 1024
```

Counter 也是从 1 开始，判断小于等于 10。不过最好是从 0 开始计算，其原因将在后面的内容中解释。

另外一个非常相似的结构是 do 循环。它与 while 循环仅有一点区别：它的循环体至少执行一次，才开始验证是否需要停止循环。下面的循环代码展示了 do 循环的功能。

```
do {
  var input = prompt("Who are you?");
} while (!input);
```

1.4.3 缩进代码

前面的示例中，在一些语句前面输入了空格。这些空格不是必须的——没有空格程序也会正常运行。实际上，即使是程序里的换行符也是可选的，可以将所有的代码都写成一行很长的代码。缩进代码的作用是突出代码结构，代码块里可以有新的代码块，如果在一个非常复杂的程序中不缩进，很难区分某一个代码块是在哪里结束而另一个代码块是从哪里开始。使用代码行缩进以后，程序的外观形状就与其内部代码块的形状相对应。笔者喜欢为每个代码块使用两个空格进行缩进，但每个人的偏好不同，有些人喜欢使用 4 个空格，而有的人则使用 Tab 键。

1.4.4 for 循环

到目前为止，while 循环展示的都是相同的模式。首先，创建 counter 变量用于跟踪循环过程，while 本身包含了一个条件检查用于判断 counter 是否达到边界值。然后，在循环体的底部更新 counter 变量。

很多循环都采用这种模式，JavaScript 和其他类似语言也都提供了更为简洁和易于理解的用法。

```
for (var number = 0; number <= 12; number = number + 2)
  print(number);
```

该程序恰好和前面的输出偶数数字示例是相同的。唯一的变化是，所有与循环的“状态”相关的语句都放在了一行。for 后面的括号里包含两个分号，第一部分通常可以通过**声明**一个变量来初始化循环；第二部分用于**检测**循环是否必须继续；第三部分用于**更新**循环的状态。在大多数情况下，for 语句比 while 语句更加简短和清晰。

下列代码用来计算 2^{10}，用 for 代替 while：

```
var result = 1;
for (var counter = 0; counter < 10; counter = counter + 1)
  result = result * 2;
result;
→ 1024
```

即便代码块没有以大括号“{”开头，循环里的语句前面仍要使用两个空格进行缩进，以便清晰地表示该语句“属于”该循环。

1.4.5 跳出循环

如果一个循环并不总是需要运行到结束，可以使用 break 关键字。break 语句用于立即退出当前循环，继续执行后面的程序。下列程序可用于查找第一个大于 20 并且能被 7 整除的数字。

```
for (var current = 20; ; current++) {
  if (current % 7 == 0)
    break;
}
current;
→ 21
```

百分比（%）操作符是判断一个数字是否可以被另外一个数字整除的最简单的方式：如果可以整除，则余数应该为 0。

本例的 for 结构没有检测是否结束循环的代码，这意味着是否退出循环取决于内部的 break 语句。另外，同样功能的循环代码可以简单编写成如下形式：

```
for (var current = 20; current % 7 != 0; current++)
  ; // 什么都不做
```

在此例中，循环体为空，单独一个分号可以用于产生一个空语句。

1.4.6 更新变量简便法

程序通常需要在原值的基础上为变量更新一个新值，尤其是在循环发生的时候。例如 counter = counter + 1，JavaScript 提供了一种简便方法：counter += 1。其他操作符也可以使用该方法，例如 result *= 2 是计算 result 的 2 倍，而 counter−= 1 则是将 counter 减 1。

对于 counter += 1 和 counter−= 1，还有更简洁的版本：counter++ 和 counter−−。

此时，代码会变得更短：

```
var result = 1;
for (var counter = 0; counter < 10; counter++)
  result *= 2;
```

1.4.7 使用 switch 进行调度

下面的代码非常普遍：

```
if (variable == "value1") action1();
else if (variable == "value2") action2();
else if (variable == "value3") action3();
else defaultAction();
```

switch 结构能够以更直接的方式解决这样的“调度”问题，不过，JavaScript 为解决该问题使用的语法（继承自 C 语言和 Java 编程语句）看起来有点笨拙，因此，有时认为 if 语句链仍是较好的选择。下面是一个示例。

```
switch(prompt("What is the weather like?")) {
  case "rainy":
    print("Remember to bring an umbrella.");
    break;
  case "sunny":
    print("Dress lightly.");
  case "cloudy":
    print("Go outside.");
    break;
  default:
    print("Unknown weather type!");
    break;
}
```

代码块内部使用了 switch，可以使用任意数量的 case 标签。程序会根据 switch 给定的值跳转到相应的标签处，如果没有匹配到标签则跳转到默认标签，然后开始执行标签处的语句，**紧接着**执行其他标签，一直到遇见 break 语句为止。在某些情况下，比如该例中的 sunny 这个 case，可用于在多个 case 里共享代码（无论是 sunny 还是 cloudy 都推荐退出）。但要注意，因为很容易忘记声明 break，所以往往会导致程序执行一些不想执行的代码。

1.4.8 大小写

之所以在声明变量名时采用独特的大小写形式，是因为变量声明不能包含空格。计算机会将带有空格的变量识别成两个单独的变量，如果一个变量名由多个单词组成，只可选用如下几种形式：fuzzylittleturtle、fuzzy_little_turtle、FuzzyLittleTurtle 或者 fuzzyLittleTurtle。第一种形式很难读懂，我个人比较喜欢使用下划线（尽管输入有点麻烦）。标准的 JavaScript 函数和大多数 JavaScript 开发人员都采用最后一种写法，习惯这种小用法不是很难，因此普遍使用第一个单词后面的每个单词的首字母大写。

在少数情况下（比如 Number 函数），变量的第一个字母也是大写。这主要是为了将该函数标记为构造函数。第 6 章将详细讲解什么是构造函数，但现在最重要的是不要被这种明显缺乏一致性的书写形式所牵绊。

1.4.9 注释

在一个程序示例中写了 // Do nothing 这个部分，大家可能会有点疑惑。在程序里添加额外的文本通常是非常有用的，最常见的用法就是给程序添加注释。

```
// The variable counter, which is about to be defined, is going
// to start with a value of 0, which is zero.
var counter = 0;
// Next, we loop. Hold on to your hat.
while (counter < 100 /* counter is less than one hundred */)
```

```
/* Every time we loop, we INCREMENT the value of counter,
   You could say we just add one to it. */
  counter++;
// And here, we are done.
```

这种文本叫做**注释**，其规则如下：/* 表示注释的开始，*/ 表示注释的结束；// 表示另一种注释的开始，该行的结尾就是注释的结束。

正如我们所看到的，添加大量注释后，即使是最简单的程序看起来也很繁冗、复杂、不美观。另外，如果一段程序代码的确很复杂或者很难理解添加注释则有很大帮助，能够解释代码的目的和作用。

1.5 进一步认识类型

前面介绍的内容可以教会大家编写和理解简单的 JavaScript 程序了，但在本章节结束之前，还要再澄清几个细节。

1.5.1 Undefined 值

声明变量时允许使用 var something; 而无需赋值，那么如果为此类变量赋值会发生什么情况呢？

```
var mysteryVariable;
mysteryVariable;
→ undefined
```

如果将变量视为触手，则该变量最终是扑空的，不能捕捉到任何值。当获取这样的变量值的时候，得到的是一个特殊的值，叫做 undefined。如果一个函数没有特别指定返回值，而只是内部调用诸如 print 和 alert 语句的时候，它的返回值也是 undefined 值。

还有一个类似的值——null，其含义是“该值已经被定义，但是没有任何值”。undefined 与 null 之间主要是理论上的区别，不会引起太多关注。在实际应用中，通常需要检测某些对象是否有值。在这种情况下，可能会用到 something == undefine 表达式——即使这些值并不相同，表达式 null == undefined 也会返回 true。

1.5.2 自动类型转换

前面的内容引入了另外一个棘手的课题。思考下面的表达式以及它们产生的布尔值。

```
false == 0;
→ true
"" == 0;
→ true
"5" == 5;
→ true
```

对不同类型的值进行比较后得知，JavaScript 使用的规则非常复杂难懂。这里不会解释得那

么精确，不过在多数情况下，它都只是将一个类型的值转化成另外一个类型的值。但是当遇到 null 或者 undefined 时，只有在两边都是 null 或 undefined 的情况下，才返回 true。

如何判断一个变量是否引用的是 false 值？规则是将字符串或者数字转化成布尔值，0、NaN 和空字符串转化成 false，其他所有的值都会转化成 true。因此，如果变量引用的是 0 或者 ""，则表达式 variable == false 返回 true。正因为如此，如果不想发生任何自动类型转化，可以使用两个额外的操作符：=== 和 !==。第一个操作符（三个等号）用于判断两个值是否精确相等（包括类型），第二个则判断是否不精确相等。在上例中，如果使用 ===，则返回 false。

```
null === undefined;
→ false
false === 0;
→ false
"" === 0;
→ false
"5" === 5;
→ false
```

if、while 和 for 语句条件里的值并不一定是布尔值，比较之前，它们会自动转换成布尔类型。这就是说数字 0、字符串 ""、null、undefined 和 false 都会转换成 false。

事实上，如果所有其他值都转换成 true，则在很多情况下省去了显式比较。如果一个变量里面包含的是字符或者 null，那就很容易判断了。

```
var maybeNull = null;
// 神秘的代码可能会给 maybeNull 赋值一个字符串
if (maybeNull)
  print("maybeNull has a value");
```

这段代码除了在神秘代码给 maybeNull 赋值为 "" 的情况外都有效，空字符串返回的是 false，因此不会输出任何内容。代码有可能是错的。为了避免一些细节错误，在类似情况下我们可以添加明确的 === null 或 ===false 进行判断。同样，数字为 0 的时候也存在这个问题。

1.5.3 自动类型转换的风险

还有其他情况可以导致发生自动类型转换。如果添加非字符串值到字符串上，在连接之前，该值会自动转换成字符串。如果数字和字符串相乘，JavaScript 将尝试将字符串转换为数字。

```
"Apollo" + 5;
→ "Apollo5"
null + "ify";
→ "nullify"
"5" * 5;
→ 25
"strawberry" * 5;
→ NaN
```

上例中的 NaN 指的是：strawberry 不是数字。对 NaN 的所有算术操作结果都是 NaN，如上例所示，NaN 和 5 相乘以后结果还是 NaN。有时让人疑惑的是：NaN==NaN 返回 false。检查一个值是否为 NaN，可以使用之前介绍的 isNaN 函数。

自动类型转换可能非常容易，但也很容易出现异常和错误。尽管“+”和“*”都是运算符，但在该例中它们的用法完全不同。在写代码时，可以经常在非字符串上使用“+”，但要注意，在字符串上不要使用“*”和其他数字运算符。将数字转换成字符串是可以实现的，并且也很简单，但将字符串转换成数字有可能行不通（如上例最后一行所示）。可以使用 Number 函数将字符串转换为数字，并要注意有可能得到的是 NaN 值。

```
Number("5") * 5;
→ 25
```

1.5.4 进一步了解 && 和 ||

前面讨论布尔操作符 && 和 || 的时候，只说过这些操作符产生布尔值，其实是有点过于简单了。如果比较的是布尔值，它们返回的就是布尔值。但它们也适用于其他类型的值，并会返回其中一个参数。

|| 的真正作用是：首先检查左边的值，如果能将该值转化成布尔类型，并且值为 true，就返回左边的值；否则返回右边的值。如果两边参数都是布尔值的话，要自己检查一下它是否运行正常。为什么会有这样的作用？事实证明它非常实用，思考下面这个示例。

```
var input = prompt("What is your name?", "Kilgore Trout");
print("Well hello " + (input || "dear"));
```

如果用户没有输入名字，而是单击了取消或是关闭了 prompt 对话框，变量 input 将会得到 null 或者 ""。这两个值在转化布尔值的过程中都会返回 false，表达式 input || "dear" 可表示变量 input 的值或者字符串 "dear"。这是一种能够提供可靠值的简单方法。

&& 操作符的作用与 || 相似，但却是相反的形式，当左边的值转换的布尔值是 false 时，返回该值；否则返回右边的值。

这两个操作符另外一个非常重要的属性是：只在必要时才执行表达式右边的值。例如 true || X，不管 X 是什么，结果都是 true，因此 X 从来没被执行过，如果内部有执行代码，也不会被执行。对于 false && X 也是如此。运行下列代码只会弹出一个 alert 窗口。

```
false || alert("I'm happening!");
false && alert("Not me.");
```

第 2 章

函　数

第 1 章我们已经使用了 alert 和 print 等几个函数来命令计算机执行特定的操作。本章将开始创建我们自己的函数，扩展现有的词汇。从某种程度上说，这就像是在我们写的故事里定义自己的词汇，以增强表达能力。虽然在写散文时这样做被认为很糟糕，但在编程中是必不可少的。

2.1　剖析函数定义

函数定义的最基本形式是这样的：

```
function square(x) {
  return x * x;
}

square(12);
→ 144
```

此处，square 是函数的名称，x 是函数的参数（第一个，也是唯一一个），return x * x; 则是函数体。

关键字 function 通常用于创建新函数。如果 function 后面是变量名，新函数就会用该变量名保存，变量名后面是一组参数名称，最后是函数体。与 while 循环和 if 语句不同，此处函数体外面的大括号是必须有的。

关键字 return 后面是表达式，用于确定函数返回的值。当代码遇到 return 语句时，立即跳出当前的函数，将返回值传递给调用该函数的代码。如果 return 后面没有表达式，函数将返回 undefined。

函数体可以包含多条语句。下面的函数用于计算次方（输入正数和次方指数）。

```
function power(base, exponent) {
  var result = 1;
  for (var count = 0; count < exponent; count++)
    result *= base;
  return result;
}

power(2, 10);
→ 1024
```

函数参数的作用类似于变量，但其值是由函数的**调用者**而不是函数自身传递的。但函数也可以就如正常变量一样任意为这些参数赋予新值。

2.1.1 定义顺序

尽管函数的定义以语句的形式出现在其余程序中，但是它们的时间轴是不一样的。在下面示例中，第一个语句可以正常调用 future 函数，尽管 future 函数的定义是在后面出现的。

```
print("The future says: ", future());

function future() {
  return "We STILL have no flying cars.";
}
```

原理是这样的：计算机在开始执行语句之前，会先查找所有的 function 定义，然后保存相关的 function。我们不必去思考多个函数定义的顺序，使用函数时允许它们之间互相调用，而不用考虑哪个函数在第一个位置定义。

2.1.2 局部变量

函数一个非常重要的特性就是：内部创建的变量是函数的**局部**变量。这就是说，power 示例中的 result 变量每次在调用该函数的时候都会创建一次，退出时则不再存在。事实上，如果 power 函数调用自身的话，该调用会创建一个独特的新 result 变量，以便内部调用时使用，回到外部调用的时候，该变量则会消失。

局部变量只适用于函数的参数和函数内部以 var 关键字定义的变量。如果没有定义同名的局部变量，函数内部则可能访问全局（非局部）变量。

下面的代码展示了该功能，它定义（并调用）可以修改变量 x 值的两个函数：第一个函数没有将变量定义为局部变量，因此修改的是程序开始时定义的**全局**变量 x 的值；而第二个函数内部定义了变量 x，因此它只改变了局部变量。

```
var x = "A";

function setVarToB() {
  x = "B";
}
setVarToB();
x;
→ "B";

function setVarToC() {
  var x;
  x = "C";
}
setVarToC();
x;
→ "B";
```

此外，这两个函数都没有包含 return 语句，它们之所以被调用是为了实现副作用功能，而不是创建值。这类函数的实际返回值是 undefined。

2.1.3　嵌套作用域

在 JavaScript 中，仅区分**全局**变量和**局部**变量还不够。实际上，变量作用域可以有任意层级（或嵌套）。其他函数内部定义的函数可以调用父函数的局部变量，而内部函数里定义的函数则不仅可以调用父函数的局部变量，还可以调用祖父函数的局部变量，以此类推。

请看下面的示例，它定义一个函数接收正数 number，然后将其和参数 factor 相乘。

```
function multiplyAbsolute(number, factor) {
  function multiply(number) {
    return number * factor;
  }
  if (number < 0)
    return multiply(-number);
  else
    return multiply(number);
}
```

我有意复杂化了该示例，目的是演示一个小技巧——函数包含两个完全独立的 number 变量。当函数体 multiply 运行时，虽然它使用的是和外部函数相同的 factor 变量，但是它有自己的 number 变量（根据内部函数 multiply 的参数名 number 创建）。因此，它将通过 factor 多次与自身参数相乘的结果返回给 multiplyAbsolute 函数。

归根结底，函数内部的变量集是否可见，取决于函数在程序中的位置，在函数“上面”定义的所有变量都是可见的，也就是存在于函数体内并包含函数定义的以及位于程序顶级的变量。这种方式的变量可访问性称为**词法作用域**。

有其他编程语言经验的人可能期望代码块（大括号之间）里也可以产生新的局部环境。但 JavaScript 没有这个功能，函数是唯一能创建新作用域的地方。可以使用任何独立的代码块：

```
var something = 1;
{
  var something = 2;
  // 这里可以变量定义
}
// 重新回到外部
```

不过，代码块内部的 something 变量和外部的变量引用的是同样的值[⊖]。实际上，尽管程序允许这样的代码块存在，但它们只能用于 if 语句分组或者循环。（很多人认为这是 JavaScript 设计者的小失误，新版本将增加能够定义存在于代码块内部的变量的功能。）

⊖　最新的值都是 2。——译者注

2.1.4 栈

为了了解函数是如何调用以及如何返回的，要先了解**栈**的概念。当函数调用时，程序控制权就交给了这个函数体，当函数体返回以后，调用该函数的代码则重新开始。因此，当函数体运行时，计算机必须记住调用该函数的上下文，以便知道之后从哪里继续。存放上下文的地方就是栈。

栈之所以存在，与一个函数体可以再次调用另外一个函数有关。每次一个函数被调用的时候，另外一个上下文就要被存储起来。可以把它想象成一个上下文的栈：每次一个函数被调用的时候，当前上下文就会压入栈的顶部；函数返回时，在顶部的上下文就会被弹出栈被重新获得。

栈需要存储于计算机的内存空间里，如果栈增长得太大，计算机就会抛出这样的信息：“堆栈溢出”或“太多递归”。下面的代码描述了这种情况：代码向计算机出了一个难题，导致两个函数之间出现了一个无穷的往返循环。更确切地说，如果栈是无穷大的，代码也将是无限大的。事实上，它会导致堆栈溢出或栈泄漏。

```
function chicken() {
  return egg();
}
function egg() {
  return chicken();
}
print(chicken() + " came first.");
```

2.1.5 函数值

正如在前一章中提到的，JavaScript 里的**所有东西**都是值，包括 function 函数。这就是说定义的函数名称可以像普通的变量一样使用，而且其内容可以传递给表达式并用于更大的表达式。在下面的示例中，如果 a 不是 false 值，程序会调用变量 a 里的函数；而如果 a 为 false 值（如 null），代码则调用 b 函数。

```
var a = null;
function b() {return "B";}
(a || b)();
→ "B"
```

(a || b)() 这个看起来有点怪异的表达式将“调用时不传参数”的操作符 () 应用到 (a || b) 上，如果该表达式产生的不是函数，则调用就会产生错误。一旦产生的是函数，就像该例中那样，结果值被调用了，则调用一切都正常。

如果仅需要一个未命名的函数值，function 关键字可以用作一个表达式，就像这样：

```
var a = null;
(a || function(){return "B";})();
→ "B"
```

这段代码产生的效果和上例一样（除了没有定义函数名 b）。无名（或匿名）函数表达式

function(){return "B";} 仅创建了一个函数值，也可以在这种定义里定义参数或者多条语句的函数体。

在第 5 章中，我们将进一步探讨函数**第一型**的特性（通常是“函数都是值”这一概念的术语），并利用其特性编写一些灵活的代码。

2.1.6　闭包

函数栈的特性及其将函数用作值的能力带来了一个有趣的问题：如果创建局部变量的函数调用不在栈上了，那局部变量会发生什么变化？下面的代码展示了这种情况：

```
function createFunction() {
  var local = 100;
  return function(){return local;};
}
```

一旦调用了 createFunction 函数，它就会创建一个局部变量，并且返回一个函数（该函数又返回该局部变量）。如何处理这一情况就是向上函数变元问题 (upwards Funarg problem)，很多旧编程语言只是禁用了这种形式。幸运的是，JavaScript 是从一种能够解决这个问题的语言演变而来的，只要这个局部变量是可达的，就会尽力保存局部变量。执行 createFunction()（创建函数并且执行），返回值是 100，这正是我们所希望的那样。

这种特性称为**闭包**，包裹一些局部变量的一个函数叫做**一个闭包**。该行为不仅让我们不用担心变量是否依然存在，而且还可以创造性地使用函数值。

例如，下面的函数可以动态创建函数值：将函数的参数加上指定的数字。

```
function makeAdder(amount) {
  return function(number) {
    return number + amount;
  };
}

var addTwo = makeAdder(2);
addTwo(3);
→ 5
```

2.1.7　可选参数

事实证明，我们可以执行下面的代码：

```
alert("Hello", "Good Evening", "How do you do?", "Good-bye");
```

函数 alert 只接收一个参数。当如此调用该函数时，它也不会出错，只是忽略其他的参数，将 Hello 显示出来。

JavaScript 不会限制传入函数的参数数目。如果传入的参数过多，多余的参数则被忽略掉；

如果传入的参数过少，缺失的参数则默认为 undefined。这样很可能导致意外传入错误的参数数目，而且不会得到提醒。

这样做的好处是，它可使函数接收“可选参数”。例如，调用下面版本的 power 函数时可以只传一个参数，此时它的行为就是平方：

```
function power(base, exponent) {
  var result = 1;
  if (exponent === undefined)
    exponent = 2;
  for (var count = 0; count < exponent; count++)
    result *= base;
  return result;
}
```

在下一章中，我们将了解一种可以获取额外传入参数列表的方式。这种方式非常有用，可以使一个函数接收任意数量的参数。print 语句使用了这种方式——下面的代码打印结果是 R2D2：

```
print("R", 2, "D", 2);
```

2.2 技巧

现在我们已经很好地了解了什么是 JavaScript 函数以及函数是如何工作的，接下来了解一些设计和编写代码时的注意事项。

2.2.1 避免重复

发明函数的原因就是为了代码复用。程序通常都要多次执行相同的操作（例如取幂），如果每一次操作都重新编写所有的代码，那么程序就会变得很长。

程序不仅会变长，读起来也会更加困难，并且很可能有错误。例如，定义 power 函数在传入负数的情况下不能正确执行。如果仍需要这些代码，那就不得不更新所有出现这段代码的地方，然后解决这些问题。如果定义的是函数，则只需要在函数里修改这个错误就可以了，所有调用它的代码就都能够正确执行。

如果发现需要不止一次地使用同一段代码，并决定将代码移至函数，就需要确定应放入函数的代码数量以及函数的接口应该是什么样的。例如，编写一个 0 填充数字[⊖] 的程序，如下所示：

```
var number = 5;
if (number < 10)
  print("0", number);
else
  print(number);
```

⊖ 在小于 10 的数字前加 0 以便对齐，比如月份 0 2。——译者注

实际工作时，我们需要在其他地方也打印这样的数字，现在可以做几项选择：

1）我们是否需要定义一个函数？这段代码有可能在不同的项目中使用，共享这个函数，让它发挥更多作用。通常答案都是“yes”。

2）函数包括 print 行为吗？还是只需要产生一个 0 填充字符串？最佳的函数是只处理单个简单行为的函数，它们很容易命名（也因此容易理解）并且可以用于各种场景。因此，应该编写一个 zeroPad 函数，而不是 printZeroPadded 函数，毕竟 print(zeroPad(5)) 不会比 printZeroPadded(5) 更难输入。

3）函数应该具有怎样的灵活性和多用性？可以编写一个极其简单的“向数字填充一个 0”的函数，也可以编写支持分数、取整、表布局的字符格式化输出系统。一个重要的原则是：在明确自己需要该函数之前，不要自作聪明。这是在引诱我们掉进为每个小功能编写复杂框架的陷阱，并且这些功能没有任何实际用处。在这种情况下，添加能够定义字符串宽度的第二个参数是一种简单操作，而且非常有用。

```
function zeroPad(number, width) {
  var string = String(Math.round(number));
  while (string.length < width)
    string = "0" + string;
  return string;
}
```

Math.round 是一个对数字取整的函数，String 是一个将参数转化为字符串的函数。

2.2.2　纯函数

用 Purity 描述函数的时候，不是指它是否不洁净或者行为不纯，而是指它是否存在副作用。**纯函数**在数理上是指：当使用函数的时候，同样的参数总是返回同样的值，而没有副作用。

纯函数和非纯函数的区别主要是在代码设计和思维层面上。如果一个函数是纯函数，调用它的时候，可以用结果直接替换而不需要改变代码的意义。当不确定它是否正常运行的时候，只需调用它进行测试，看看它在此上下文中是否正常运行——它应该在任何上下文都能运行。非纯函数因各种因素可能返回不同的值并产生副作用，我们可能很难测试和理解这些副作用。

由于纯函数是自足自给的，所以它可能比非纯函数的作用更大，适用范围更广。以前面编写的 zeroPad 函数为例，如果定义的是 printZeroPadded 函数，那么该函数只能在已经定义 print 功能并且愿意直接输出填充数字的特定场景下才能使用。如果声明一个将数字转化为字符串的纯函数，那么该函数对上下文的依赖度就会更低，并且应用范围更广。

当然，zeroPad 解决的问题与 print 不同，非纯函数能够解决 print 的问题，因为它需要一个副作用。在很多情况下，我们需要的是非纯函数。在少数情况下，纯函数也可以解决某个问题，但使用非纯函数会更方便、更有效。通常来说，如果我们想写的东西可以很自然地使用纯函数来表达，那就编写纯函数。后面我们会庆幸自己这样做，如果没有，也不要觉得编写非纯函数很低级。

2.2.3 递归

如前所述，函数可以调用自身。一个函数调用自身叫做**递归**。递归有一些有趣的函数定义。看一下 power 函数的另一种实现方式：

```
function power(base, exponent) {
  if (exponent == 0)
    return 1;
  else
    return base * power(base, exponent - 1);
}
```

这种方式更接近数学的求幂计算，概念上比前面的方式好得多。它是一种循环，但没有使用 while、for，甚至连一个局部的副作用都没有看到。该函数通过调用自身产生了和 for 循环一样的效果。

这里有一个重要的问题：在大多数 JavaScript 实现里，第二个版本会比第一个版本慢 10 倍。在 JavaScript 里，运行简单的循环比多次调用一个函数方便得多。另外，如果在该函数上使用一个足够大的幂数，将会导致栈溢出。

速度和美观两者不可兼得的难题是我们比较关注的，这并不局限于关于递归的争论。在很多情况下，一个美观、直观而又短小的方案会被一个复杂但非常快的方案所代替。

对于前面版本中的 power 函数，不美观的函数依然很简单并且易读。用递归替换它也没有多大意义。但更多情况下，如果程序处理的概念非常复杂，为了让程序更简单明了，损失一些效率就成为一个明智的选择。

很多开发人员反复重复的基本规则是，只有在证明程序运行太慢时才去关注效率问题。一旦出现这种情况，找出占用最多时间的代码，然后将这些美观的代码改成高效的代码。

当然，以前的规则不是指应该完全忽略效率问题。大多数情况下，像 power 这样的函数，采用了“美观”的方式就会失去简单性；而在其他情况下，有经验的开发人员能够立即判断出采用简单的方式永远都不够快。

着重强调这一点的原因是，我很奇怪为什么这么多开发人员都过分狂热地追求效率，甚至非常小的细节。结果导致程序变得更臃肿、更复杂、错误更多，而将其改写为简单的程序又导致了周期过长，却通常也只是让程序快了那么一丁点儿。

递归并不总是循环低效率的代名词，有些问题用递归解决比循环要简单得多。大部分都是一些需要探测或者处理几个分支情况（每个分支又可能生成更多分支）的问题。

思考这样一个难题：从数字 1 开始，重复执行加 5 或者乘 3 这个步骤，会产生无穷多个数字。如何编写函数，使其能够找出恰当的加法和乘法运算序列，以产生指定数字呢？

例如，数字 13 可以通过 1 乘 3，然后加两次 5 计算出来，而数字 15 却没办法实现。

解决方案如下所示：

```
function findSequence(goal) {
  function find(start, history) {
    if (start == goal)
      return history;
    else if (start > goal)
      return null;
    else
      return find(start + 5, "(" + history + " + 5)") ||
             find(start * 3, "(" + history + " * 3)");
  }
  return find(1, "1");
}

findSequence(24);
→ (((1 * 3) + 5) * 3)
```

请注意，该方案找出的不一定是**最短**的运算序列，不过只要找出任一序列就足够了。

它是如何实现的呢？通过两个不同的路径调用自身，内部 find 函数能够检测当前数字加 5 或者乘 3 的可能性，如果找到了该数字，就返回 history 字符串（该字符串用于记录所有尝试获取该数字的操作）。同时，它也判断当前数字是否大于 goal，如果大于，就应该停止继续检测该分支，因为继续下去也根本不可能得到需要的数字。

示例中使用的 || 操作符可以理解为：返回 start 加 5 的值，如果是 false，就返回 start 乘 3 这个方式。其意思相当于（但代码更冗长）如下代码：

```
else {
  var found = find(start + 5, "(" + history + " + 5)");
  if (found == null)
    found = find(start * 3, "(" + history + " * 3)");
  return found;
}
```

HZ BOOKS
华章图书

第 ③ 章

数据结构：对象与数组

在本章中，我们将解决从文本中提取数据的编程问题。在此过程中，可以了解什么是对象和数组，以及如何使用它们。

3.1 问题：Emily 姨妈家的猫

想象一下下面这种情况：据说你那个疯狂的姨妈 Emily 在家里养了 50 多只猫（从来没有数过），她定期给你发邮件叙述最新发生的事情。邮件内容通常是这样的：

亲爱的外甥：

你妈告诉我你从事跳伞运动了，是真的吗？小心点儿，小子！你还记得发生在我丈夫身上的事吧，那还只是从二楼跳下来。

不管怎样，我这里发生的事情非常有趣。这一周，我一直在想方设法吸引 Drake 先生的注意，他是位刚搬到隔壁的绅士，但我想他挺害怕猫的，或者是对猫过敏。

下次见到他我就把胖 Igor 放到他肩上，我非常想知道会发生什么。

还有，我上次告诉你的那个技巧，效果比我期望得还好。我已经收到五笔付款了，只有一个投诉，但我开始感觉有点糟了，你说得对，可能在某种形式上它是非法的。

……

爱你的

Emily 姨妈

died 27/04/2006: Black Leclère

born 05/04/2006 (mother Lady Penelope): Red Lion, DoctorHobbles the 3rd, Little Iroquois

为了让姨妈高兴，你要记住她养的猫的家族情况，因此，可以添加这样的内容：“附：祝老二 Doctor Hobbles 这周六生日快乐！”或者“Penelope Lady 还好吗？她现在 5 岁了，对吧？”——最好不要唐突地询问有关猫死亡的事情。你有大量从姨妈那里收到的旧邮件，幸运的是，每次邮件的格式都是一模一样的：在邮件底部记录了猫的出生和死亡情况。

当然，你不愿意也很难自己一封一封地去翻阅这些邮件。而我们恰巧也需要这样一个问题范例。试着编写一个能够解决该问题的程序。首先，编写一个程序，它可以根据最新的邮件，用列表来记录仍然活着的猫。

（在通信一开始的时候，Emily 姨妈只有一只名为 Spot 的猫，那时她仍然是很“正常”的。）

在编写程序之前，通常要了解程序的目的，并努力找出其线索。下面是编写计划：

1）开始定义一个猫的集合，里面只有一只猫 Spot。

2）按时间前后顺序，遍历所有的归档邮件。

3）查找 born 和 died 开头的段落。

4）将以 born 开头的段落里的猫的名字全部添加到定义的猫集合里。

5）将以 died 开头的段落里的猫的名字从猫集合里删除。

从段落中找出名字，步骤如下：

1）找出段落里的冒号；

2）获取冒号后面的内容；

3）将找到的名字通过逗号进行分割。

我们可能还需要确定 Emily 姨妈是否总是使用这种格式的邮件，以及有没有拼错或者忘记某个名字——这就是你姨妈的特点。

3.2 基本数据结构

在编写程序之前，需要重温一下该语言的一些新特性。

3.2.1 属性

有些 JavaScript 值有其他一些与其相关联的值，这些相关联的值称为**属性**。例如，每个字符串都有一个 length 属性，它引用的是一个数字，用来表示该字符串的字符数量。

可以采用两种方式访问属性：使用中括号或者使用点标记法。

```
var text = "purple haze";
text["length"];
→ 11
text.length;
→ 11
```

第二种方式是第一种的速记法，只有在属性名称是一个合法的变量名称的时候才能使用，也就是属性不包含任何空格或符号并且不以数字开头。

从值 null 和 undefined 中读取属性会产生错误。数字和布尔值确实都有属性，但是没有任何意义，或者说我们没有必要在此讨论。

3.2.2 对象值

对于大多数的值类型，如果它们有属性，这些属性就是固定属性，不能进行修改（例如，字符串的 length 属性都是相同的）。但是，有一个值类型——**对象**，它的属性可以自由添加、删除及修改。对象所扮演的主要角色，实际上就是一个属性集合。

可以这样编写一个对象：

```
var cat = {color: "gray", name: "Spot", size: 46};
cat.size = 47;
cat.size;
→ 47
delete cat.size;
cat.size;
→ undefined
```

像变量一样，对象的每个属性都是用一个名称来标记的。属性名称可以是任意字符串，而不仅仅是合法的变量名。第一行语句创建的对象包含了一个 size 属性，值为 46，第二行语句以修改变量的方式，给这个 size 属性赋了一个新值。

读取一个不存在的属性会得到一个 undefined 值，关键字 delete 用于删除属性。

如果用 = 操作符设置一个不存在的属性，将会给对象添加一个新属性，如下面的示例：

```
var empty = {};
empty.notReally = 1000;
empty;
→ {notReally: 1000}
```

属性名称如果不是一个合法的变量名称，则不可以用点标记法访问，只能使用中括号的形式访问。创建对象时，除数字外，这些属性名称都需要用引号引住。

```
var thing = {"gabba gabba": "hey", 5: 10};
thing["5"];
→ 10
thing[2 + 3];
→ 10
delete thing["gabba gabba"];
```

中括号内的部分可以为任意表达式，中括号会将表达式转化为字符串来判断是否有该属性的名称。因此，也可以把变量名称当成属性名称。

```
var propertyName = "length";
var text = "coco";
text[propertyName];
→ 4
```

操作符 in 可以用来判断一个对象是否有某个属性，它产生的是布尔值。

```
var chineseBox = {};
chineseBox.content = chineseBox;
"content" in chineseBox;
```

```
→ true
"content" in chineseBox.content;
→ true
```

3.2.3 对象即集合

有关猫问题的方案里讨论的是名字集合。集合是指一组值，这组值中的每个值都只出现一次。如果名字是字符串，可以想想利用什么方式来让一个对象描述这些名字。

当然是利用名字作为属性名。要将一个名字添加到集合里，需要在对象里设置这个属性，同时设置属性的值（可以为任意值）。删除属性可以将名字从集合里删除。in 操作符用于判断某个名字是否存在于集合内。（这样使用 in 操作符有些小问题，我们将在第 6 章进行讨论，到目前为止它还是很好用的。）

在此创建一个仅包含 Spot 的集合值，添加 WhiteFang 到集合里，然后再次删除 Spot，测试 Asoka 是否在集合里。

```
var set = {"Spot": true};
set["White Fang"] = true;
delete set["Spot"];
"Asoka" in set;
→ false
```

3.2.4 易变性

对象值看起来是可以修改的。但在第 1 章讨论的所有的值类型都**不能改变**——不可能去改变这些类型的现有值。可以将这些类型混合在一起，从中派生新的值，但如果声明了字符串值，内部的字符文本是不能修改的。另一方面，对象的内容**可以**通过修改其属性来进行更改。

如果有两个数字 120 和 120，不管它们是否引用相同的物理位，都被认为是完全相同的数字。使用对象时，相同对象的两个引用和包含相同属性的两个不同对象是有区别的。思考下面的代码：

```
var object1 = {value: 10};
var object2 = object1;
var object3 = {value: 10};

object1 == object2;
→ true
object1 == object3;
→ false

object1.value = 15;
object2.value;
→ 15
object3.value;
→ 10
```

object1 和 object2 是两个变量，抓取的是**相同**的值，这里只有一个实际对象，因此修改了 object1 的值，同时也改变了 object2 的值。变量 object3 指向的是另外一个对象，默认和 object1 有相同的属性，但各自单独运行。

比较对象时，JavaScript 中的 == 操作符只有在赋予的两个值都是完全相同时才能返回 true。比较两个内容相同的不同的对象将返回 false。这在有些情况下非常有用，但有些时候则不实用，而只能编写单独的函数来比较对象的内容。

3.2.5 对象即集合：数组

对象值可以扮演很多不同的角色。集合只是其中一个角色，本章我们将了解对象值的其他一些角色，第 6 章将讲述使用对象的另外一种重要方式。

猫问题的计划（实际上，我们称它为**算法**，而非计划）是讲述在存档里查看所有电子邮件。什么样的值能够代表这种存档呢？

这种存档应该能够容纳很多电子邮件。为了达到目的，一个邮件可以仅是一个字符串。我们需要将多个字符串收集到一个值里。集合就是使用的对象。作为初稿，一个集合对象可以这样编写：

```
var mailArchive = {"the first email": "Dear nephew, ...",
                   "the second email": "..."
                   /* 等等 ...*/};
```

但我们很难从头到尾地查看所有的邮件——那么程序又将如何猜测这些属性的名称？可以使用容易理解的属性名称来解决这个问题。

```
var mailArchive = {0: "Dear nephew, ... (mail number 1)",
                   1: "(mail number 2)",
                   2: "(mail number 3)"};

for (var current = 0; current in mailArchive; current++)
  print("Processing email #", current, ": ", mailArchive[current]);
```

幸运的是，有一种特殊的对象能够用于这种特殊用途，我们称之为**数组**。数组提供了一些便利，比如，length 属性能够告诉我们数组里有多少值。

新数组可以使用中括号（[]）来创建：

```
var mailArchive = ["mail one", "mail two", "mail three"];

for (var current = 0; current < mailArchive.length; current++)
  print("Processing email #", current, ": ", mailArchive[current]);
```

本示例中的代码并没有明确声明元素对应的数字，而是第一个元素自动获取数字 0，第二个元素获取数字 1，以此类推。

计数一般都是从 1 开始，此处为什么从 0 开始？这样看起来缺乏直观性：集合里的元素对应的数字从 0 开始是大多数编程语言过去采用的处理方式。暂时先采用这样的用法，我们会渐渐习

惯这种用法。

从元素 0 开始意味着，如果一个集合有 X 个元素，那么最后一个元素可以在 X-1 的位置找到。这也就是示例中的 for 循环要判断 current <mailArchive.length 的原因。在集合 mailArchive.length 的位置上没有元素，因此，一旦 current 达到该值，就停止循环。

编写一个 range 函数作为一次练习：接收一个正整数作为参数，返回一个数组，数组包含从 0 到给定正整数之间所有的数字。

仅输入 [] 就可以创建一个空数组。和对象一样，可以通过赋值属性给数组添加元素。因为元素属性都是数字，所以必须使用 [和] 来引用它们而不能用点（.）。（请注意，添加或删除元素时，数组的 length 属性会自动更新——该值是数组元素最大的索引数字 +1。）

```
function range(upto) {
  var result = [];
  for (var i = 0; i <= upto; i++)
    result[i] = i;
  return result;
}
range(4);
→ [0, 1, 2, 3, 4]
```

现在采用 i 代替之前一直使用的循环变量 counter 或 current。使用 i、j、k 等单个字母作为循环变量是程序员普遍采用的方法。这样做的根源主要是懒惰：我们宁愿敲 1 个字符，而不是 7 个，而且像 counter 或 current 这样的名称并没有十分明确地阐明变量的意思。

如果一个程序使用过多的单字母变量，就会变得难以理解。我自己写程序时，只是在一些常见的场景中使用单字母变量，小的 loop 循环就是一个例子。如果一个循环包含了另一个同样使用 i 变量的循环，那么内部循环就会破坏外部循环的值，所有运行都会出错。可以在内部循环里使用 j 变量，但一般来说，如果一个循环体非常庞大，我们应该为计数变量取一个含义明确的变量名。

3.2.6 方法

除了 length 属性以外，string 和 array 还包含很多个引用到函数值的属性。

```
var doh = "Doh";
typeof doh.toUpperCase;
→ "function"
doh.toUpperCase();
→ "DOH"
```

每个字符串都有一个 toUpperCase 属性，调用时，它返回字符串的拷贝并且所有的字符都是大写的。另外还有 toLowerCase 属性，我们可以猜到它的作用。

请注意，尽管调用 toUpperCase 的时候没有传入任何参数，该函数却能访问字符串“Doh”，并且它的值是一个属性，我们将在第 6 章讲解它是如何实现的。

包含函数的属性通常称为**方法**。toUpperCase 是字符串对象的一个方法。下面的示例展示了数组对象的一些方法：

```
var mack = [];
mack.push("Mack");
mack.push("the");
mack.push("Knife");
mack;
→ ["Mack", "the", "Knife"]
mack.join(" ");
→ "Mack the Knife"
mack.pop();
→ "Knife"
mack;
→ ["Mack", "the"]
```

与数组有关的方法 push，可用于在数组末尾插入新值。它也可用于 range 函数，用 result.push(i) 替换 result[i] = i。还有一个方法 pop，与 push 相反，它是将数组的最后一个值删除并且返回该值。方法 join 是将字符串数组转化为一个单字符串，传入的参数用于将数组的各个值连接起来。

3.3 解决关于 Emily 姨妈家猫的问题

回到前面提到的猫的问题，现在已经知道用数组保存归档邮件是一种很好的方式。如果假设变量 ARCHIVE 内有一个邮件字符串数组，那么现在查看所有的邮件就变得简单了：

```
for (var i = 0; i < ARCHIVE.length; i++) {
  var email = ARCHIVE[i];
  // 对这封邮件进行处理
}
```

我们已经确定了描述活着猫的集合的方法。下一个问题就是在邮件里找出以“born”和“died”开头的段落。

3.3.1 分离段落

我们遇到的第一个问题是：什么是段落？在本例中，字符串值本身并没有多少用处：JavaScript 对文本的概念并没有深入到字符序列的概念，所以必须用这些术语来定义段落。

在第 1 章中，我们了解了换行字符，换行字符通常用来分离段落。然后要将段落视为一封邮件的一部分，它开始于一个新行或者文本开始处，并且结束于下一个新行之前或文本的末尾。

不需要编写一个让字符串分隔成多个段落的算法，字符串有个很好用的 split 方法。与数组的 join 方法基本相反，它将一个字符串分解成一个数组，以给定的字符串作为参数来确定在什么位置分解字符串。例如：

```
var words = "Cities of the Interior";
words.split(" ");
→ ["Cities", "of", "the", "Interior"]
```

因此，分隔新行（email.split("\n")）可以将一封邮件分隔成多个段落。

3.3.2 找出相关段落

程序可以忽略不以“born”和“died”开头的段落。如何测试一个段落是否以某个特定单词开头呢？ charAt 方法用于从某个字符串获取指定的字符，x.charAt(0) 给出第一个字符，而 x.charAt(1) 则给出第二个字符，以此类推。因此，下列代码可以判断一个字符串是否以“born”开头。

```
var paragraph = "born 15-11-2003 (mother Spot): White Fang";
paragraph.charAt(0) == "b" && paragraph.charAt(1) == "o" &&
  paragraph.charAt(2) == "r" && paragraph.charAt(3) == "n";
→ true
```

但这种方式有些笨拙，想象一下如果要判断 10 个字符的字符串会怎样。这里需要了解一点：如果一行语句太长，可以分成多行，并缩进第二行以便识别第二行与上一行是一体的，更易于阅读结果。

字符串还有一个方法叫 slice，用于从第一个参数所在的位置开始到第二个参数所在的位置结束拷贝出一部分字符串。该方法可以用更少的代码判断字符串：

```
paragraph.slice(0, 4) == "born";
→ true
```

我们可以将这种方法封装成一个 startsWith 函数，它接收两个参数，均为字符串。如果第一个参数以第二个参数字符串开头就返回 true，否则返回 false。

```
function startsWith(string, pattern) {
  return string.slice(0, pattern.length) == pattern;
}

startsWith("rotation", "rot");
→ true
```

如果 charAt 或 slice 获取不存在的字符串会发生什么？如果 pattern 参数比 string 参数长，startsWith 函数是否依然能正常运行？

```
"Pip".charAt(250);
→ ""
"Nop".slice(1, 10);
→ "op"
```

如果指定的位置没有字符，charAt 将返回空字符 ""，而 slice 则只是将不存在的内容忽略掉。

所以，这意味着，调用 startsWith("Idiots", "Most honored colleagues") 时，startsWith 是有效

的。如果 pattern 比 string 长，slice 返回的永远是比 pattern 短的字符串（因为 string 没有足够的字符），因此，用 == 比较，结果返回 false 是正确的。

它能够帮助程序在遇到异常（但有效）的输入时总是花一点时间对此进行判断。这些情况通常称为**特殊用例**，能在所有“正常”输入情况下完美运行的程序在特殊用例中失效，这是很常见的。

3.3.3 提取猫的名字

猫问题中唯一还没解决的部分就是从段落里提取猫的名字。之前提到的算法是：

1）找出段落里的冒号；

2）获取冒号后面的内容；

3）将找到的内容用逗号分隔出多个名字。

以“died”和“born”开头的段落都可以使用这种方式，因此，最好是编写一个统一的函数，以便两个不同位置的段落都可以使用这个函数。

字符串有一个 indexOf 方法，可以找出字符第一次出现的位置或者截取字符串中的子串。另外，如果 slice 只有一个参数，它将返回从指定位置一直到字符串结束位置之间的字符串。

因此，可以这样提取猫的名字：

```
function catNames(paragraph) {
  var colon = paragraph.indexOf(":");
  return paragraph.slice(colon + 2).split(", ");
}

catNames("born 20/09/2004 (mother Yellow Bess): Doctor Hobbles the 2nd, Noog");
→ ["Doctor Hobbles the 2nd", "Noog"]
```

一般所说的算法忽略了冒号（：）和逗号（，）之后的空格，截取字符串时使用 +2 是为了省略冒号以及冒号后面的空格。split 的参数包含了逗号和空格，因为它们才是真正用于分离名字的字符，而不仅仅是逗号。

函数没有做任何问题检查，在这里假设输入一直都是正确的。

3.3.4 完整算法

剩余的工作就是将这些代码段拼在一起。其中一个方法如下所示：

```
var livingCats = {"Spot": true};

for (var mail = 0; mail < ARCHIVE.length; mail++) {
  var paragraphs = ARCHIVE[mail].split("\n");
  for (var i = 0; i < paragraphs.length; i++) {
    var paragraph = paragraphs[i];
    if (startsWith(paragraph, "born")) {
      var names = catNames(paragraph);
      for (var name = 0; name < names.length; name++)
```

```
        livingCats[names[name]] = true;
    }
    else if (startsWith(paragraph, "died")) {
      var names = catNames(paragraph);
      for (var name = 0; name < names.length; name++)
        delete livingCats[names[name]];
    }
  }
}
```

这段代码非常庞大，要设法在短时间内它看起来更加美观。但首先看一下结果，我们知道如何判断某只猫是否还活着。

```
if ("Spot" in livingCats)
  print("Spot lives!");
else
  print("Good old Spot, may she rest in peace.");
```

如何列出所有活着的猫？ in 关键字与 for 一起使用时有不同的含义：

```
for (var cat in livingCats)
  print(cat);
```

这样的循环能够查看一个对象的所有属性名称，并允许在集合里列举所有的名字。

3.3.5 清理代码

有些代码看起来像杂乱的丛林，猫问题的示例方案也有这样的问题。仅仅添加一些空行可以使代码看起来更清晰。但这只是让代码看起来更美观，并没有真正解决这个问题。

这里需要做的是分解这些代码，我们已经编写了两个辅助函数：startsWith 和 catNames，它们都能够解决小部分容易理解的问题。继续清理代码如下：

```
function addToSet(set, values) {
  for (var i = 0; i < values.length; i++)
    set[values[i]] = true;
}

function removeFromSet(set, values) {
  for (var i = 0; i < values.length; i++)
    delete set[values[i]];
}
```

这两个函数用于向集合添加或从中删除名字，这样就已经将方案里的两个内部循环移植到这两个函数里了。

```
var livingCats = {Spot: true};

for (var mail = 0; mail < ARCHIVE.length; mail++) {
```

```
  var paragraphs = ARCHIVE[mail].split("\n");
  for (var i = 0; i < paragraphs.length; i++) {
    var paragraph = paragraphs[i];
    if (startsWith(paragraph, "born"))
      addToSet(livingCats, catNames(paragraph));
    else if (startsWith(paragraph, "died"))
      removeFromSet(livingCats, catNames(paragraph));
  }
}
```

要我说的话，这段代码确实进步了很多。

addToSet 和 removeFromSet 为何要接收 set 集合作为参数呢？它们可以直接用 livingCats 变量。这样就不会被我们现在遇到的问题困扰，如果 addToSet 直接修改了 livingCats，就可以称它为 addCatsToCatSet 或者其他类似的名称。通过采用这种方式，它就是一个更加通用的工具。

即使永远不再用这些函数做任何事情（事实上这是很有可能的），这样编写也是很有用的。因为它们是“自足自给的”——不需要了解 livingCats 外部变量的情况，就可以被程序读取和理解。

这两个函数不是纯函数，它们将传入的对象更改为 set 参数。这使它们比真正的纯函数更难处理，不过，比起失去控制并能随意更改任意值或变量的函数，它们仍然比较容易理解。

继续将整体算法拆解成不同的部分：

```
function findLivingCats() {
  var livingCats = {"Spot": true};

  function handleParagraph(paragraph) {
    if (startsWith(paragraph, "born"))
      addToSet(livingCats, catNames(paragraph));
    else if (startsWith(paragraph, "died"))
      removeFromSet(livingCats, catNames(paragraph));
  }

  for (var mail = 0; mail < ARCHIVE.length; mail++) {
    var paragraphs = ARCHIVE[mail].split("\n");
    for (var i = 0; i < paragraphs.length; i++)
      handleParagraph(paragraphs[i]);
  }
  return livingCats;
}
```

现在整个算法被封装成一个函数，这意味着它运行之后不会导致混乱：livingCats 在函数中是局部变量，而不是顶级变量（也就是全局变量），因此该变量只有在函数运行的时候才存在。需要该集合的代码可以调用 findLivingCats 并使用它的返回值。

让 handleParagraph 成为单独的函数也可以起到清理代码的作用，但是该函数与猫问题算法

联系紧密，在其他情况下就没有任何意义。另外，它需要访问 livingCats 变量，因此，最好的选择就是将其作为一个函数内部的函数。当它位于 findLivingCats 内部时，很显然它只与确定的位置相关，并且能够访问父函数里的变量。

该方案实际上比前面的方案更**庞大**，但它看起来更加整齐、更易于阅读，希望大家也赞同这一点。

```
var howMany = 0;
for (var cat in findLivingCats())
  howMany++;
print("There are currently ", howMany, " cats alive.");
```

3.3.6 日期表示

该程序还忽略了很多邮件里的信息，邮件中有猫的出生日期、死亡日期和猫母亲的名字。

我们将从处理日期开始。怎样存储日期才是最好的方式？可以使用三个属性——年 (year)、月 (month)、日 (day)，来生成一个对象并在这些属性中存储数字。

```
var when = {year: 1980, month: 2, day: 1};
```

JavaScript 已经为此提供了一种对象，这种对象可以使用关键字 new 来创建。

```
var when = new Date(1980, 1, 1);
```

就像之前讲过的花括号和分号（{};）法，new 也是创建对象值的一种方式。它不是指定所有的属性名称和值，而是用函数来创建对象。这样就可以定义一种创建对象的标准过程。这种函数称为**构造函数**，我们将在第 6 章中了解如何编写这种函数。

可以按照不同的方式使用时间构造函数：

```
// 声明一个对象表示当前时间
new Date();
 // 1980 年 2 月 1 日！
new Date(1980, 1, 1);
// 2007 年 3 月 30 日，8 点 20 分 30 秒
new Date(2007, 2, 30, 8, 20, 30);
```

这些对象可以存储一天中的某个时间，也可以存储日期。如果不传入任何参数，就创建表示当前时间和日期的对象。参数用于请求特定的时间和日期，顺序是：年、月、日、小时、分钟、秒和毫秒。最后 4 个是可选的，不特别指定时默认为 0。

这些对象用到的月份数字是从 0 到 11，这点可能有点难理解，尤其是日数字是从 1 开始的。

可以通过一系列的 get 方法来检查日期对象：

```
var today = new Date();
print("Year: ", today.getFullYear(), ", month: ",
      today.getMonth(), ", day: ", today.getDate());
print("Hour: ", today.getHours(), ", minutes: ",
      today.getMinutes(), ", seconds: ", today.getSeconds());
```

```
print("Day of week: ", today.getDay());
```

除了 getDay[⊖] 外，这些 get 方法都有对应的 set 方法，用于改变日期对象的各项值。

在对象内部，日期被描绘成从 1970 年 1 月 1 日到当前时间的毫秒数。这是个非常大的数字。

```
var today = new Date();
today.getTime();
→ 1266587282246
```

正确处理日期的方法是比较这些日期：

```
var wende = new Date(1989, 10, 9);
var gulfWarOne = new Date(1990, 6, 2);
wende < gulfWarOne;
→ true
wende == wende;
→ true
// 但是小心 ...
wende == new Date(1989, 10, 9);
→ false
```

使用 <、>、<= 和 >= 比较日期比较理想。如果用 == 将一个日期对象与其自身进行比较，结果是 true，这样做也是可行的；但如果使用 == 将日期对象和另一个日期相同的对象进行比较，得到的结果却是 false。

如前所述，即使这两个对象包含相同的属性，使用 == 比较两个不同的对象的时候，也返回 false。这样做有点笨拙并且容易出错，即便我们原本以为 >= 和 == 的作用差不太多。测试两个日期对象是否相等可以使用如下代码：

```
var wende1 = new Date(1989, 10, 9),
    wende2 = new Date(1989, 10, 9);
wende1.getTime() == wende2.getTime();
→ true
```

除了日期和时间以外，日期对象还包含时区信息。阿姆斯特丹下午 1 点的时候，根据当年的时间，它可以是伦敦的中午 12 点，也可以是纽约的早上 7 点。只有将各自的时区考虑在内，才能对这些时间进行比较。日期对象的 getTimezoneOffset 函数用于找出当前时间和格林威治时间 (GMT) 相差多少分钟。如果是在柏林，将得到如下结果：

```
new Date().getTimezoneOffset();
→ -60
```

3.3.7 日期提取

日期部分总是在一个段落的相同位置，这样很方便！

```
"born 02/04/2001 (mother Clementine): Bugeye, Wolverine"
"died 27/04/2006: Black Leclère"
```

⊖ getDay 是获取当天是当前周的第几天，而 getDate 和 setDate 才是获取和设置年月日里的日的方法。——译者注

可以编写一个函数 extractDate，传入一个段落，返回一个日期对象：

```
function extractDate(paragraph) {
  function numberAt(start, length) {
    return Number(paragraph.slice(start, start + length));
  }
  return new Date(numberAt(11, 4), numberAt(8, 2) - 1, numberAt(5, 2));
}
```

不用 Number 这段代码，程序也可以正常运行，但正如在第 1 章中提到的，我倾向于使用数字而非字符串。引入内部函数是为了防止重复执行三次 Number 和 slice。

请注意，月份数字要减去 1。和大多数人一样，Emily 姨妈数月份的时候都是从 1 开始，所以在传入日期构造函数之前，需要调整这个值。（日数字不会遇到这个问题，因为日期对象是按照平常的方式计算的。）

在第 8 章中，我们将了解一种更实用更强大的从字符串中提取信息的方式。

3.3.8 收集更多信息

从现在开始存储猫的方式就不同了。现在不是仅存储一个 true 到集合里而是存储一个关于猫信息的对象。当某只猫去世的时候，我们不是从集合里删除它，而是仅仅给这只猫的对象添加一个 death 属性来存储死亡日期。

这意味着 addToSet 函数和 removeFromSet 函数已经没有用处了，我们需要使用类似的函数，但它必须不仅能够存储出生日期，还可以存储猫母亲的名字。

```
function catRecord(name, birthdate, mother) {
  return {name: name, birth: birthdate, mother: mother};
}

function addCats(set, names, birthdate, mother) {
  for (var i = 0; i < names.length; i++)
    set[names[i]] = catRecord(names[i], birthdate, mother);
}
function deadCats(set, names, deathdate) {
  for (var i = 0; i < names.length; i++)
    set[names[i]].death = deathdate;
}
```

catRecord 是一个单独的函数，用于创建这些存储对象。在其他情况下它可能也有用处，例如为 Spot 猫创建对象。这种对象使用的术语通常是 Record，用于聚合一定数量的值。

我们试着从“born 15/11/2003 (mother Spot) ....”段落里提取猫母亲的名字：

```
function extractMother(paragraph) {
  var start = paragraph.indexOf("(mother ") + "(mother ".length;
  var end = paragraph.indexOf(")");
  return paragraph.slice(start, end);
```

```
}

extractMother("born 15/11/2003 (mother Spot): White Fang");
→ "Spot"
```

请注意开始的位置是如何调整 "(mother " 的长度的——indexOf 返回的位置是该 "(mother " 字符串的开始位置，而不是它结束的位置。

新扩展的猫算法如下所示：

```
function findCats() {
  var cats = {"Spot": catRecord("Spot", new Date(1997, 2, 5), "unknown")};
  function handleParagraph(paragraph) {
    if (startsWith(paragraph, "born"))
      addCats(cats, catNames(paragraph), extractDate(paragraph),
              extractMother(paragraph));
    else if (startsWith(paragraph, "died"))
      deadCats(cats, catNames(paragraph), extractDate(paragraph));
  }

  for (var mail = 0; mail < ARCHIVE.length; mail++) {
    var paragraphs = ARCHIVE[mail].split("\n");
    for (var i = 0; i < paragraphs.length; i++)
      handleParagraph(paragraphs[i]);
  }
  return cats;
}
```

每只猫出生的时候，我们向 cats 对象添加一条新记录；猫去世的时候，标记一下该记录。因此，findCats 的返回值是一个对象，该对象的每个属性都是一个已命名的猫、并保存了该猫相关信息的记录。

3.3.9 数据表示

有了这些额外的数据，我们就可以根据 Emily 姨妈讲述的内容找到猫的信息线索。下面的函数可能是有用的：

```
function formatDate(date) {
  return date.getDate() + "/" + (date.getMonth() + 1) + "/" +
    date.getFullYear();
}

function catInfo(data, name) {
  var cat = data[name];
  if (cat == undefined)
    return "No cat by the name of " + name + " is known.";
```

```
  var message = name + ", born " + formatDate(cat.birth) +
                " from mother " + cat.mother;
  if ("death" in cat)
    message += ", died " + formatDate(cat.death);
  return message + ".";
}

//例如…
catInfo(catData, "Fat Igor");
→ "Fat Igor, born 1/6/2004 from mother Miss Bushtail."
```

catInfo 里的第一个 return 语句是个特例返回，如果找不到给定猫的信息，该函数的余下语句就没有意义了，因此要立即返回一个值，防止余下语句继续执行。

过去，一些程序员认为函数内部包含多个返回语句是错误的。原因是这样做很难知道哪些代码已被执行以及哪些代码未被执行。在第 4 章中讨论的一些其他技术基本能够反驳这种思想，但偶尔还会遇到有些人批判这些简便的 return 语句。

接下来，编写一个 oldestCat 函数，传入一个猫集合对象作为参数，返回活着的最高龄的猫。

```
function oldestCat(data) {
  var oldest = null;

  for (var name in data) {
    var cat = data[name];
    if (!("death" in cat) && (oldest == null || oldest.birth > cat.birth))
      oldest = cat;
  }

  if (oldest == null)
    return null;
  else
    return oldest.name;
}
```

if 语句里的条件看起来有点难以理解，可以理解为“如果这只猫没有去世，并且 oldest 为 null 或者是出生日期晚于当前猫的猫，那么将这只猫存储到 oldest 变量中。”

3.4 更多理论

现在已经了解数组和对象的概念了，接下来澄清一些之前未具体解释的问题。

3.4.1 arguments 对象

每当调用函数的时候，一个有“魔力”的特殊变量 arguments 就会添加到允许函数体运行的环境（上下文）中。该变量引用的对象类似于一个数组，它使用属性 0 对应第一个函数参数，属性 1

对应第二个参数，以此类推，所有传入到该函数的参数都有记录。另外它也有一个 length 属性。

不过，arguments 对象并不是一个真正的数组——它没有像 push 这样的方法；当向该变量添加属性的时候，它也不会自动更新 length 属性。这是 JavaScript 语言无规则发展造成的弊端。

```
function argumentCounter() {
  return "You gave me " + arguments.length + " arguments.";
}
argumentCounter("Straw man", "Tautology", "Ad hominem");
→ "You gave me 3 arguments."
```

有些函数可以像 print 函数一样接收任意数量的参数。这些函数通常会遍历 arguments 对象中的每个值，然后再处理这些值。其他函数只可以接收可选参数，如果不设置，就指定一些可用的默认值。

```
function add(number, howmuch) {
  if (arguments.length < 2)
    howmuch = 1;
  return number + howmuch;
}

add(6);
→ 7
add(6, 4);
→ 10
```

我们也可以扩展前面编写的 range 函数，提供第 2 个可选参数。如果只传 1 个参数，产生一个从 0 到该给定数字之间的数组；如果传入了两个参数，第 1 个参数便是 range 的开始值，而第 2 个则是结束值。

```
function range(start, end) {
  if (arguments.length < 2) {
    end = start;
    start = 0;
  }
  var result = [];
  for (var i = start; i <= end; i++)
    result.push(i);
  return result;
}

range(4);
→ [0, 1, 2, 3, 4]
range(2, 4);
→ [2, 3, 4]
```

可选参数的运行和上例不太一样，如果没有传入两个参数，则第一个参数作为结束值使用，而开始值则变为 0。

3.4.2 完成扫尾工作

你还记得前言中介绍的这个代码行吗？

```
print(sum(range(1, 10)));
```

我们已经定义了 range 操作符（即函数），现在只需编写 sum 函数让代码运行起来，该函数接收一个数字数组然后返回它们的和。此时，可以很容易地编写这些代码。

```
function sum(numbers) {
  var total = 0;
  for (var i = 0; i < numbers.length; i++)
    total += numbers[i];
  return total;
}

sum(range(1, 10));
→ 55
```

3.4.3 Math 对象

前一章讲述了 Math.max 和 Math.min 函数，会注意到这些函数就是存储在 Math 下的对象的 max 和 min 属性。这是对象扮演的另外一个角色：一个位置拥有若干相关的值。

在 Math 内部还有很多其他值，如果都直接将它们放在全局环境里，就会“污染”该环境。名称使用的越多，就越可能会意外覆盖其他变量的值。例如，如果想在程序里命名 max，并不是不可能的事。

如果一个变量的名称已经存在，大多数语言都会制止重复定义该名称，或者至少是警示不要这样做。但 JavaScript 不会，所以应特别注意。

在任何情况下，都可以使用 Math 内部的常量和数学函数。所有的三角函数都包括在内——Math.cos、sin、tan、acos、asin 和 atan。π 和 e 也包含在内，都要写成大写字母（Math.PI 和 Math.E），由于某些历史原因，这是一种表明某值是常数值的通用方法。Math.pow 完美替代了前面编写的 power 函数——它可以接受负数和分指数；Math.sqrt 返回平方根；Math.max 和 Math.min 返回两个值中的最大和最小的值；Math.round、Math.floor 和 Math.ceil 将分别返回离给定数字最近、小于给定数字的最大值、大于给定数字并最小的整数值。

3.4.4 可枚举属性

可能大家已经想出如何找出 Math 对象中的内容了：

```
for (var name in Math)
  print(name);
```

但是，什么都没有发生。同样，设想一下它遍历一个数组属性的结果。

```
for (var name in ["Huey", "Dewey", "Louie"])
  print(name);
```

结果只看到 0、1 和 2，而不是 length、push 或 join。对象的一些属性似乎在循环的时候隐藏了，或者是正式调用了，总之有很多理由。一个合理的理由是：所有的对象都拥有一些方法（例如 toString），将对象转化为某些形式的相关字符串，但在查找存放在对象里的猫或进行其他操作时，不想看到这些属性。

所有在程序里添加到对象上的属性都是可见的，没有办法隐藏它们，在第 6 章中会明白这是为什么。这种方式通常适用于将方法添加到对象上，而不是在 for 和 in 循环里显示方法。

Eloquent JavaScript
A Modern Introduction to Programming

编程原理与运用规则的简练、完美融合。我喜欢游戏式的程序开发教程，本书再次点燃了我学习编程的热情。对了，是JavaScript！

—— **Brendan Eich** JavaScript之父

因为这本书，我成为了更棒的架构师、作家、咨询师和开发者。

—— **Angus Croll** Twitter开发者

如果你决定只买一本有关JavaScript的书，那么就应是Marijn Haverbeke的这本书。

—— **Joeyde Villa** GlobalNerdy

本书不仅是学习JavaScript最棒的教材之一，而且是通过学习JavaScript进而学习现代编程的优秀图书。当有人问我如何学好JavaScript时，我会推荐这本书。

—— **Chris Williams** 美国JSConf的组织者

我读过的最棒的JavaScript书籍之一。

—— **Rey Bango** 微软Client-Web社区项目经理和jQuery团队成员

这本书不仅是一本非常不错的JavaScript指导书，而且是一本很棒的编程指导书。

—— **Ben Nadel** Epicenter咨询公司首席软件工程师

这本书对编程基本原理的详述以及对栈和环境等概念的解释非常到位。注重细节使本书从其他的JavaScript书中脱颖而出。

—— **Designorati**

上架指导：计算机/程序设计/Web开发
ISBN 978-7-111-39665-9
9 787111 396659
定价：49.00元

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379525
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com
网上购书：www.china-pub.com

no starch press