

其次，`console`。这一部分是 `log(..)` 函数所在的对象引用。我们将在第 2 章中深入介绍对象及其属性。

创建可见输出的另外一个方法是运行 `alert(..)` 语句。如下所示：

```
alert( b );
```

如果运行这个语句，那么你就会发现输出并没有打印到终端中，而是弹出一个“OK”对话框，变量 `b` 的内容会呈现在对话框中。然而，在终端中学习和运行程序时，使用 `console.log(..)` 通常比使用 `alert(..)` 更加方便，因为这样无需与浏览器界面交互就可以一次输出多个变量。

我们在本部分中使用 `console.log(..)` 作为输出方法。

## 1.3.2 输入

在讨论输出的同时，你可能也会好奇如何实现输入（即如何接收用户的信息）。

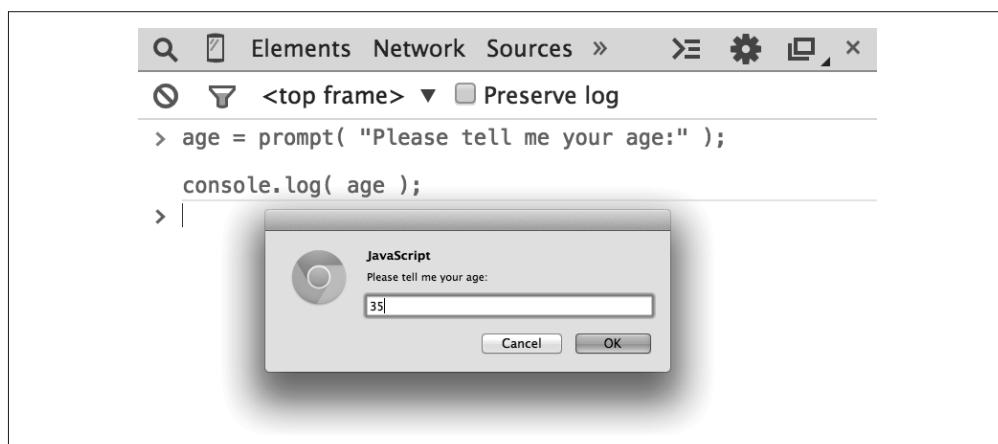
最常用的方法是，通过 HTML 页面向用户显示表单元素（如文本框）用于输入，然后通过 JavaScript 将这些值读取到程序变量中。

还有另一种更为简单的获取输入的方法，用于简单的学习和展示，这也是我们将会使用的方法，即 `prompt(..)` 函数：

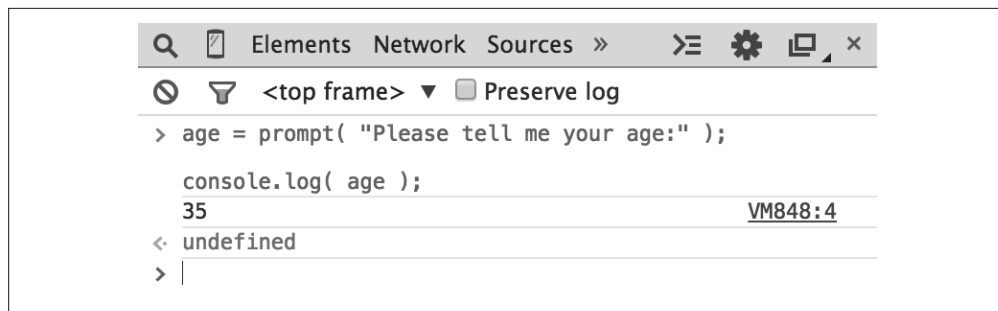
```
age = prompt( "Please tell me your age:" );  
  
console.log( age );
```

你可能已经猜到了，传给 `prompt(..)` 的消息会打印到弹出窗口中，本例中是 "Please tell me your age:"。

如下图所示：



输入文本并点击“OK”后，你就可以看到输入的值会保存到变量 `age` 中，接着通过 `console.log(..)` 输出：



```
Q Elements Network Sources » ⌵ ⚙ 📄 ✕
🚫 🔍 <top frame> ▼ ☐ Preserve log
> age = prompt( "Please tell me your age:" );
    console.log( age );
    35 VM848:4
< undefined
> |
```

为了简化难度，在学习基本编程概念时，本部分中使用的示例都不需要输入。但既然你已经学习了如何使用 `prompt(..)`，如果想要挑战自我，你可以在自己的示例中使用输入。

## 1.4 运算符

使用运算符，我们可以对变量和值执行操作。我们已经在前文中看到了 `=` 和 `*` 这两个 JavaScript 运算符。

运算符 `*` 执行算术乘法。很简单，对吧？

等号运算符 `=` 用于**赋值**——我们先计算 `=` 右边（源值）的值，然后将它存入**左边**（目标变量）指定的变量中。



这种赋值方法的顺序看起来是反的，有点奇怪。有些人可能更习惯将顺序调过来，将源值放在左边，目标变量放在右边，不用 `a = 42` 这种形式，而是 `42 -> a`（这不是合法的 JavaScript）。但问题是，`a = 42` 这种顺序以及类似的变体在现代编程语言中是非常流行的。如果感觉不太习惯的话，那么你就需要花点时间来习惯它，并将它植入到你的思维中。

考虑：

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

在上述示例中，我们将值 `2` 赋给变量 `a`。然后，我们取得变量 `a` 的值（仍然是 `2`），加上 `1`，得到结果 `3`，再将这个值保存在变量 `b` 中。

虽然 `var` 严格意义上说并不是一个运算符，但每个程序都会用到这个关键词，因为它是**声明**（也就是**创建**）变量（参见 1.7 节）的基本方法。

在使用变量前总是应该先声明变量。一个变量在每个作用域（参见 1.11 节）中只需要声明一次；声明之后可以按照需要多次使用。如下所示：

```
var a = 20;

a = a + 1;
a = a * 2;
console.log( a ); // 42
```

以下是 JavaScript 中最常用的一些运算符。

- 赋值  
=，如 `a = 2` 就表示将值 2 保存在变量 `a` 中。
- 算术  
+（加）、-（减）、\*（乘）、/（除），如 `a * 3`。
- 复合赋值  
+=、-=、\*= 和 /= 是复合运算符，可以将算术运算符与赋值组合起来，比如，`a += 2` 等同于 `a = a + 2`。
- 递增 / 递减  
++ 表示递增，-- 表示递减，比如 `a++` 就类似于 `a = a + 1`。
- 对象属性访问  
如 `console.log()` 中的 `.`。  
  
对象是在名为属性的位置中持有其他值的值。`obj.a` 指的是一个名为 `obj` 的对象值，并伴有一个属性名为 `a` 的属性。也可以通过 `obj["a"]` 这种形式访问属性。参见第 2 章。
- 相等  
==（粗略相等）、===（严格相等）、!=（粗略不等）和 !==（严格不等），如 `a == b`。  
  
参见 2.1 节。
- 比较  
<（小于）、>（大于）、<=（小于或粗略等于）和 >=（大于或粗略等于），如 `a <= b`。  
  
参见 2.1 节。
- 逻辑  
&&（与）和 ||（或），如 `a || b` 就表示 `a` 或者 `b`。  
  
这些运算符用于表示复合条件（参见 1.9 节），比如 `a` 或 `b` 为真。



有关运算符的更多细节以及这里没有覆盖到的更多介绍，参见 Mozilla 开发者网络的“表达式与运算符”([https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions\\_and\\_Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators))。

## 1.5 值与类型

如果你向手机店的店员咨询某款手机的价格，他会回答说“99.99”（也就是 99.99 美元），那么他给你的是一个实际的美元数，表示购买手机需要付的金额（加上税）。如果你想要买两部这款手机，那么可以很容易地算出价格为 199.98 美元。

如果这个店员拿起另一个类似的手机，声称它是“免费的”（很可能要签约），这时他并没有给出一个具体的数字，而是价格（\$0.00）的另外一种表示方法——“免费”。

接着，如果你询问手机是否附带充电器，那么你得到的答案只会是“是”或“否”。

同样，当在程序中表达某些值时，根据将对这些值进行的操作，你可以为这些值选择不同的表示方法。

在编程术语中，对值的不同表示方法称为**类型**。JavaScript 为以下这些**基本值**提供了内置类型。

- 在计算时，你需要的是一个数字（**number**）。
- 在屏幕上打印一个值时，你需要的是一个字符串（**string**，一个或多个字符、单词或句子）。
- 在程序中作出决策时，你需要的是一个布尔值（**boolean**，**true** 或者 **false**）。

直接包含在源码中的值被称为**字面值**。字符串字面值由双引号（"..."）或单引号（'...'）围住，二者的唯一区别只是风格不同。数字和布尔型字面值可以直接表示（如 42、true 等）。

考虑：

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

42;

true;
false;
```

除了字符串 / 数字 / 布尔值类型，编程语言通常还会提供**数组**、**对象**、**函数**等。我们将在本章和下一章中更深入地介绍值和类型。

## 类型转换

如果需要在屏幕上打印出一个数字，那么就需要将这个值转化为字符串，在 JavaScript 中，

这种转化称为“类型转换”。类似地，如果向电子商务网页的表单中输入一系列数字字符，那么这都是字符串，但如果需要使用这些值进行数学计算，则需要将其转换为数字。

JavaScript 为类型间的强制转换提供了几种不同的机制。举例来说：

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

如上所示，`Number(...)`（一个内置函数）的使用是一种显式的类型转换，可以将任意类型转换为数字类型。这应该是很直观的。

但是，如果需要进行比较的是不同类型的两个值，那么会怎么样呢？这就是一个很有争议的问题了，需要隐式的类型转换。

如果要比较字符串 "99.99" 和数字 99.99，多数人会认为它们是相等的，但它们其实并不完全相同，难道不是吗？它们是两种表示方法下的同一个值，属于两种不同的类型。你可以说它们是“粗略相等”，是这样吗？

为了帮助你处理这些常见的情形，JavaScript 有时会隐式地将值转换到匹配的类型。

因此，如果你使用 `==` 粗略相等运算符来判断 `"99.99" == 99.99` 是否成立，JavaScript 会将左边的 "99.99" 转换为等价的数字类型 99.99。这时比较就变成了 `99.99 == 99.99`，当然为 `true` 了。

尽管隐式类型转换的设计意图是为了提供便利，但如果你没有花时间学习其行为方式的规则的话，它也可能会产生误导。而多数 JavaScript 开发者从来没有花时间来学习这一知识，所以他们普遍感觉隐式类型转换令人迷惑，并且会让程序产生出乎意料的 bug。他们认为应该尽量避免隐式类型转换。隐式类型转换甚至被称为是语言设计中的缺陷。

然而，隐式类型转换是可以学习的机制，任何想要严肃对待 JavaScript 编程的人都应该学习。不仅仅是因为一旦掌握了其规则，就不会再被它迷惑，实际上这也可以提高你的程序质量！所以为此付出努力是十分值得的。



有关类型转换的更多信息，参见下一章和本系列《你不知道的 JavaScript（中卷）》<sup>2</sup> 第一部分中的第 4 章。

---

注 2：此书已由人民邮电出版社出版。——编者注

## 1.6 代码注释

手机商店的店员可能会草草记下新发布手机的特性或者其公司提供的新套餐。这些笔记只是给店员看的，而不是给顾客阅读的。然而，通过记录要向顾客提供的信息以及提供方式，这些笔记可以帮助店员提高自己的工作质量。

这里可以学到的最重要的一点是，编写代码并不只是为了给计算机看。在给计算机看的同时，代码同样要给开发者阅读。

计算机只关心机器码，也就是那些编译之后得到的二进制 0 和 1 序列。要想得到同样的 0 和 1 序列，几乎有无数种程序写法。而你选择的程序编写方案很重要，这不只是对你个人来说，对小组的其他成员，甚至对未来的你也同样很重要。

编写程序时不仅应该努力做到让程序能够正确执行，而且应该做到使代码阅读起来也是容易理解的。你可能需要花费很多精力为变量（参见 1.7 节）和函数（参见 1.11 节）选择一个好的名字。

另一个非常重要的部分是代码注释。这是程序中的文本，将其插入程序只是为了向人类解释说明代码的执行。解释器 / 编译器会忽略这些注释。

有关如何编写注释良好的代码有很多观点；我们确实无法定义绝对的普遍标准。但是以下这些观察结论和指导原则是很有用的。

- 没有注释的代码不是最优的。
- 过多注释（比如每行一个）可能是拙劣代码的征兆。
- 代码应该解释为什么，而非是什么。如果编写的代码特别容易令人迷惑的话，那么注释也可以解释一下实现原理。

JavaScript 中的注释有两种类型：单行注释和多行注释。

考虑：

```
// 这是一个单行注释

/* 而这是
   一个多行
   注释。
  */
```

如果想要将注释放到单个语句上方或者一行的末尾，那么可以使用单行注释 `//`。这一行中位于 `//` 之后直到行尾的所有内容都会被当作注释（因此会被编译器忽略）。单行注释的内容没有限制。

考虑：

```
var a = 42;      // 42是生命的意义
```

多行注释 `/* .. */` 适用于在注释中需要多行解释的情况。

以下是多行注释常用的一个场景：

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

多行注释也可以出现在行中的任意位置，甚至可以出现在行中间，因为 `*/` 会结束注释。如下所示：

```
var a = /* 任意值 */ 42;  
  
console.log( a );    // 42
```

唯一不能出现在多行注释中的是 `*/`，因为这会被解释为注释的结束。

开始学习编程时一定要养成注释代码的习惯。在本章后面的内容中，你会看到我使用注释来进行解释，所以你在自己的练习中也应该这么做。相信我，如果你这么做的话，每个阅读你代码的人都会感谢你的！

## 1.7 变量

大多数的实用程序都需要跟踪值的变化，因为程序在执行任务时会对值进行各种操作，值会不断发生变化。

在程序中实现这一点的最简单方法是将值赋给一个符号容器，这个符号容器称为**变量**，使用这个名字是因为这个容器中的值是可以变化的。

在某些编程语言中，你需要声明一个变量（容器）用于存放指定类型的值（如数字或字符串）。通过避免不想要的值转换，人们认为这种**静态类型**（也称为**类型强制**）提高了程序的正确性。

其他语言强调的是值的类型而不是变量的类型。**弱类型**（也称为**动态类型**）允许一个变量在任意时刻存放任意类型的值。这种方式允许一个变量在程序的逻辑流中的任意时刻代表任意类型的值，人们认为这样可以提高程序的灵活性。

JavaScript 采用了后一种机制——**动态类型**，这也就是说，变量可以持有任意类型值而不存在类型强制。

前面提到过，我们使用 `var` 语句声明一个变量。注意，声明中没有额外的类型信息。考虑以下这个简单的程序：

```
var amount = 99.99;

amount = amount * 2;

console.log( amount );      // 199.98

// 将amount转化为一个字符串，并在开头添加 "$"
amount = "$" + String( amount );

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

变量 `amount` 开始时持有值 `99.99`，然后持有 `amount * 2` 的结果值，也就是 `199.98`。

第一个 `console.log(..)` 命令需要隐式地转换类型，将数字值转换为字符串用于输出。

然后语句 `amount = "$" + String(amount)` 显式地将值 `199.98` 转换为字符串，并在开头加上字符 `"$"`。此时，`amount` 持有字符串值 `"$199.98"`，所以第二个 `console.log(..)` 语句打印输出时就不需要转换类型了。

JavaScript 开发者应该注意到变量 `amount` 表示值 `99.99`、`199.98` 和 `"$199.98"` 的这种灵活性。静态类型的狂热支持者可能会单独使用一个变量，例如，使用 `amountStr` 来保存最后的 `"$199.98"`，因为这是一个不同的类型。

无论是哪一种方式，你都会注意到 `amount` 保存的值会随着程序运行而有所变化，这展示了变量的主要用途：管理程序状态。

换句话说，状态跟踪了值随着程序运行的变化。

变量的另一个常见用法是集中设置值。更常见的说法是常量，即声明一个变量，赋予一个特定值，然后这个值在程序执行过程中保持不变。

这些常量的声明通常放在程序的开头，所以如果需要改变这些值的话，那么就会有一个很方便的集中位置。通常来说，在 JavaScript 中作为常量的变量用大写表示，多个单词之间用下划线 `_` 分隔。

以下是一个简单的示例：

```
var TAX_RATE = 0.08; // 8%的营业税

var amount = 99.99;

amount = amount * 2;

amount = amount + (amount * TAX_RATE);
```



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



`console.log(..)` 是作为 `console` 值的一个对象属性的函数 `log(..)`，与此类似，`toFixed(..)` 是一个可以通过数字值访问的函数。JavaScript 的数字不会自动格式化为美元表示法，因为引擎无法了解你的意图，也没有适合现金的类型。`toFixed(..)` 可以帮助我们指定保留数字小数点后的几位，并按照期望生成字符串值。

变量 `TAX_RATE` 是依靠惯例而定的一个常量，程序中没有任何特殊实现可以防止它被修改。而如果这个城市的营业税提高到 9%，那么我们可以很容易地修改程序，只需要在唯一一处修改 `TAX_RATE` 值为 `0.09`，而不是在程序中搜索多个 `0.08`，然后修改所有的值。

在编写本部分时，最新版本的 JavaScript（一般被称为“ES6”）提供了一个新的常量声明方法，使用 `const` 代替了 `var`：

```
// 自ES6起：
const TAX_RATE = 0.08;

var amount = 99.99;

// ..
```

常量和值不变的变量一样有用，而且常量还可以防止值在最初设定后被无意修改。如果要在初始声明后给 `TAX_RATE` 赋其他值，那么程序会拒绝这个修改（严格模式下会失败退出，参见 2.4 节）。

另外，这种防止出错的“保护”措施与静态类型相似，所以你应该可以理解其他语言中的静态类型是多么具有吸引力了！



有关如何在程序中使用变量中的不同值，参见本系列《你不知道的 JavaScript（中卷）》第一部分中的前两章。

## 1.8 块

当你选购好新手机而结账时，手机商店的店员必须完成一系列的步骤。

与此类似，我们常常需要将在代码中的一系列语句组织到一起，这些语句通常被称为块。在 JavaScript 中，使用一对大括号 `{ .. }` 在一个或多个语句外来表示块。考虑：

```
var amount = 99.99;
```

```
//一个通用的块
{
    amount = amount * 2;
    console.log( amount );    // 199.98
}
```

这种独立的 { .. } 块是合法的，但在 JavaScript 程序中比较少见。通常来说，块会与其他某个控制语句组合在一起，比如 if 语句（参见 1.9 节）或循环（参见 1.10 节）。举例来说：

```
var amount = 99.99;

// amount是否足够大呢？
if (amount > 10) {           // <-- 块与if组合
    amount = amount * 2;
    console.log( amount );    // 199.98
}
```

我们将在下一节中介绍 if 语句，但正如你可以看到的，包含两个语句的 { .. } 块与 if (amount > 10) 结合在一起了；块内的语句只有在条件判断成立时才会运行。



与 console.log( amount ); 这样的大多数其他语句不同，块语句不需要以分号 (;) 结尾。

## 1.9 条件判断

“您想要再加一个价值 \$9.99 的屏幕保护膜吗？”手机商店的店员这么问就是在请你作出一个决定。你可能会先看看钱包或银行账号的当前状态再回答这个问题。但显然，这只是一个简单的“是”或“否”的问题。

程序中有很多方法可以用于表示条件判断（也就是决策）。

最常用的是 if 语句。本质上就是在表达“如果这个条件是真的，那么进行后续这些……”。举例来说：

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

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

if 语句要求在括号 ( ) 中放一个表达式，这个表达式要么是 true，要么是 false。在这个程序中，我们提供的表达式是 amount < bank\_balance，根据 bank\_balance 变量中的数量，其求