|  |
| --- |
| JavaScript 艺术之旅 |
| The JavaScript Art Tour |

|  |
| --- |
| tanpero  2019/8/20 |

目录

[第三章 4](#_Toc17284761)

[逻辑 4](#_Toc17284762)

[布尔值 4](#_Toc17284763)

[undefined 和 null 5](#_Toc17284764)

[逻辑运算 7](#_Toc17284765)

[条件表达式 12](#_Toc17284766)

[数值 13](#_Toc17284767)

[数值的表示 13](#_Toc17284768)

[数值的精度 15](#_Toc17284769)

[基本运算 17](#_Toc17284770)

[NaN 22](#_Toc17284771)

[parseInt 和 parseFloat 23](#_Toc17284772)

[数学函数 26](#_Toc17284773)

[随机数 27](#_Toc17284774)

[字符串 29](#_Toc17284775)

[字符串的概念 29](#_Toc17284776)

[转义字符 31](#_Toc17284777)

[Unicode 32](#_Toc17284778)

[字符串操作 36](#_Toc17284779)

[转换规则 41](#_Toc17284780)

[模板字符串 42](#_Toc17284781)

[对象 46](#_Toc17284782)

[对象的概念 46](#_Toc17284783)

[成员操作 50](#_Toc17284784)

[枚举 53](#_Toc17284785)

[全局对象 54](#_Toc17284786)

[数组 56](#_Toc17284787)

[初探数组 56](#_Toc17284788)

[Array 构造器 60](#_Toc17284789)

[插入与移除 62](#_Toc17284790)

[排序与查找 65](#_Toc17284791)

[数据类型 71](#_Toc17284792)

[JavaScript 中的数据类型 71](#_Toc17284793)

[判断数据类型的技巧 73](#_Toc17284794)

[解构赋值 75](#_Toc17284795)

[数组解构 75](#_Toc17284796)

[对象解构 78](#_Toc17284797)

[第四章 语句 81](#_Toc17284798)

[if 语句 81](#_Toc17284799)

[switch 语句 87](#_Toc17284800)

[初识 switch 语句 87](#_Toc17284801)

[使用 break 90](#_Toc17284802)

[关联操作 91](#_Toc17284803)

[分组 92](#_Toc17284804)

[使用动态条件 94](#_Toc17284805)

[while 和 do-while 语句 94](#_Toc17284806)

[循环结构 95](#_Toc17284807)

[do-while 语句 100](#_Toc17284808)

[for 语句 101](#_Toc17284809)

[基本概念 102](#_Toc17284810)

[迭代算法 104](#_Toc17284811)

[数组遍历 109](#_Toc17284812)

[for-in 和 for-of 语句 110](#_Toc17284813)

[属性遍历 110](#_Toc17284814)

[可迭代对象 114](#_Toc17284815)

[异常处理 119](#_Toc17284816)

[语法 119](#_Toc17284817)

[try...catch 语句的使用 120](#_Toc17284818)

[异常处理的应用 122](#_Toc17284819)

[异常对象 124](#_Toc17284820)

# 第三章

数据

## 逻辑

### 布尔值

小的时候，我们认为一张白纸只有两面，一个人只有好人坏人的区别，这个世界只有黑白两种颜色，说的话只有真假之分。现在，我们也可以说：逻辑——所有的逻辑，本质上都是一个靠真与假驱动的世界。真，则一往直前；假，则回归起点。JavaScript 提供了 true 来表示所有的“真”，false 来表示所有的“假”。他们便是一个真与假的二元世界。true 和 false 被称为布尔值，用以纪念 19 世纪为逻辑学做出杰出贡献的 George Boole。

在 JavaScript 中，布尔值属于 boolean 类型。我们可以直接使用它们来明确地表示真假、是非，也时常会隐藏在一串逻辑表达式中，作为它背后的力量。

alert(true); // true  
alert(false); // false

### undefined 和 null

先来看四个句子：

杨雨露有个姐姐，她叫杨雨晴。

杨雨露没有姐姐。

杨雨露不知道自己有没有姐姐。

杨雨露有姐姐，但是不知道姐姐在哪。

JavaScript 为第二种情况提供了 null，为第三、四种情况提供了 undefined。

null 和 undefined 是 JavaScript 定义的两个特殊值，分别表示

1. 一个空值。

* 这个可能需要一个值，但是明确地知道”这是空的“，用 null 来表示空值。

1. 未发现需要的值。

* 这个地方不知道有没有值，用 undefined 来表示”未定义“。

null 是 JavaScript 的*关键字*，如果对它进行声明或赋值操作会产生错误。

let null; // SyntaxError: Unexpected token null  
null = 1; // ReferenceError: Invalid left-hand side in assignment

undefined 不是一个明确定义的保留字，如果尝试对它赋值不会产生错误，但它的值也不会改变。

alert(undefined); // undefined  
undefined = 0; // 不会产生错误  
alert(undefined); // undefined

如果对它进行重复声明，则它会被视作一个变量，在声明前会产生一个“变量未定义”的错误，在声明后使用它则会发现它的值已经发生改变。这是一个语言缺陷。

alert(undefined); // ReferenceError: undefined is not defined  
let undefined = 0;  
alert(undefined)； // 0  
undefined = 3;  
alert(undefined); // 3

我们可以使用表达式 void 0 来得到最“纯粹”的 undefined 值。并且我们也推荐这种方法——它写起来更简短！

alert(void 0); // undefined  
undefined = 0;  
alert(undefined); // 0  
alert(void 0); // undefined

如果我们声明了一个变量却没有给它赋予任何值，那么它的默认值就是 undefined ——即“未定义”。

let a;  
alert(a); // undefined

练习 3.1.1

1. 举出生活中可以分别用 null 和 undefined 描述的例子。
2. 尝试了解在 JavaScript 的创造过程中，null 和 undefined 分别是怎样出现的。

### 逻辑运算

很多人觉得逻辑冰冷而机械死板，正是如此。因此，它才有用。人类易被感情左右，但计算机不同。正是因为冰冷且机械死板，计算机才会一直稳定运行，为我们所用。

逻辑的本质是真与假的组合。在 JavaScript 中，以下值都会被视为“假”：

false NaN 0 "" '' null undefined

除了以上的“假”值，其他自然都是“真”值。关于 0 、特殊数值 NaN、字符串的概念将在下文中讲到。

真值都可以被看做 true，假值都可以被看做 false。这两个布尔值是逻辑的基本组成部分，简单的逻辑自然也可以组合成更复杂的逻辑，这个组合的过程我们称为*逻辑运算*。与、或、非是三个基本的逻辑运算，JavaScript 提供了它们的运算符 && || 和 ！。这三个运算得到的值与参与运算的值有关，但是得到的还是参与运算的值本身，而不一定是布尔值。

1. **&& （与）**

* 如果两个条件都为 true，那么得到 true，否则得到 false。

它和我们平时说话时“如果……并且……”是类似的，即判断两个条件是否都*成立*，

alert(true && false); // false  
alert(true && true); // true  
alert(false && false); // false

很容易理解，对吧！但事实上，&& 不一定会得到一个布尔值。它得到的值与用来运算的值有关，如果运算的值不是布尔值，它也不一定得到一个布尔值，而是根据值本身被看做“真”或被看做“假”来决定得到什么值。

它的具体运算方式如下：

* 如果第一个条件被视为 true，而第二个条件被视为 false ，那么得到第二个条件的值。
* 如果两个条件都被视为 true，那么得到第二个条件的值。
* 如果第一个条件被视为 false，那么得到第一个条件的值。

示例：

alert(0 && true); // 0  
alert(true && 0); // 0  
alert(0 && false); // 0  
alert(false && 0); // false  
alert(100 && 0); // 0  
alert("Hello" && "") // ""  
alert(null && undefined) // null  
alert(100 && NaN) // NaN

1. **||（或）**

* 两个条件中只要有一个为 true，那么得到 true，否则为 false。

它和我们平时所说的“如果……或者……”是等价的。

alert(true || false); // true（第一个条件的值）  
alert(true || true); // true（第一个条件的值）  
alert(false || false); // false（第二个条件的值）

和 && 类似，|| 也不一定得到一个布尔值，而是根据它所运算的值被看做“真”还是看做“假”来得到值。

它的具体运算方式如下：

* 如果第一个条件被视为 true，那么直接得到第一个条件的值。
* 如果第一个条件被视为false，那么得到第二个条件的值。

示例：

alert(0 || true); // true  
alert(true || 0); // true  
alert(0 || false); // false  
alert(false || 0); // 0  
alert(100 || 0); // 100  
alert("Hello" || "") // "Hello"  
alert(null || undefined) // undefined  
alert(100 || NaN) // 100

换句话说，如果第一个条件为“真”，那么就符合“或”的条件了，不必再判断下一个。如果第一个条件为假，就需要将第二个条件作为整个运算得到的值。

1. **!（非）**

* 如果值为 true，那么得到 false，否则得到 true。

它确实得到一个布尔值，具体运算方式如下：

* 如果条件被视为 true，那么得到 false。
* 如果条件被视为 false，那么得到 true。

示例：

alert(!true); // false  
alert(!false); // true  
alert(!0); // true  
alert(!100); // false  
alert(!NaN); // true  
alert(!""); // true  
alert(!undefined); // true  
alert(!!0); // false  
alert(!!null); // true

因此，!! 两个非运算重复进行，得到的值就是条件本身的布尔值描述形式，即被看做“真”还是“假”。

**运算符优先级**

当 && || ! 三个运算符同时在一个表达式中，运算过程遵循**操作符优先级**。! 操作符具有最高的优先级，即在一个表达式中它所属的式子总是被最先计算，其次是 &&，|| 的优先级最低。

alert(10 && !5); // false  
alert(!5 && 10); // false  
alert(!5 || 6 && 7); // 7  
alert(5 || 6 && 7 || 8); // 5  
alert(5 && 6 || 7 && 8); // 6  
alert(!5 || !6 || 7 && 8); // 8

在第三行代码中：

1. !5 由于具有最高的优先级，被最先计算。由于它是 false，且是 || 的第一个条件，因此会继续计算位于 || 右侧的第二个条件。
2. 由于 && 的优先级大于 || ，因此会计算 6 && 7，结果为 7，那么 || 的第二个条件就是 7。
3. 因此整个逻辑表达式的结果就是 7。

在第四行代码中：

1. 5 被视作 true, 因此 || 运算符不会查看第二个条件。
2. 结果为 5。

在第五行代码中，首先从左往右运算，5 && 6 的值为 6，则 || 的第一个条件为 6，最后结果为 6。

在第六行代码中：

1. !5 为 false，|| 运算符会查看第二个条件。
2. 由于 ! 具有最高的优先级，!6 会先得到计算，结果为 false。
3. 那么 !5 || !6 的值为 false。
4. 第二个 || 操作符会查看右边的条件。
5. 由于 && 的优先级大于 || ，会先计算 7 && 8 的值，值为 8。
6. 那么右边的条件为 8。
7. false || 8 的值为 8。

在实际应用中，我们可以使用括号 ( ) 来更改默认的运算符优先级。使用了括号的示例如下：

alert((5 || 6) && (7 || 8)); // 8  
alert(5 && (6 || 7) && 8); // 8  
alert(5 && 6 && (7 && 8)); // 8

一个操作符后使用括号括起来的内容是一个整体，会先计算括号中表达式的值，这个值作为该操作符的条件进行下一步计算。因此，在第一行代码中：

1. (5 || 6) 作为一个整体会被计算，值为 6。
2. 6 成为了 && 的第一个条件。&& 会查看第二个条件。
3. 第二个条件是 (7 || 8)，值为 8，所以第二个条件是 8。
4. 6 && 8 结果为 8。

我们可以尝试演算另两行代码的运算过程。

练习 3.1.2

1. 计算如下代码的值：

* (18 || 24) && (15 && 0) || 6 || !12

1. 计算如下代码的值：

* !(15 || 0) && !(12 && !12)

1. 计算如下代码的值：

* 18 && (!(15 || 10) && (15 && 10))

### 条件表达式

条件表达式是一种*三目运算符*，它需要三个操作数。格式如下：

a ? b : c

如果 a 被视作 true，则这个表达式的值为 b ，否则为 c。

示例：

let a = 0;  
alert(a ? "Hello" : "Hi"); // "Hi"

条件表达式的运算符具有最低的优先级。也就是说，如果 a b c 都是其它表达式，那么一定会先计算出 a b c 的值，再得到条件表达式的值。一般来说，如果 a b c 都是表达式，我们推荐给用括号进行包裹以避免混淆。

练习 3.1.3

1. 计算下列条件表达式的值：

* (!(15 || 10) && (15 && 10)) ? "Hello" : "world"

1. 计算下列条件表达式的值：

* (15 || (true && NaN) || !Infinity && (!NaN || 12))) ? "Jim" : "Tom"

## 数值

毫无疑问，在 JavaScript 中，数值是一类非常基本的值。关于它的意义我们无需多言，这里，我们将逐步探索 JavaScript 中有关数值和数值运算的各类细节。

### 数值的表示

JavaScript 中数值的十进制表示法与我们平时所使用的基本相同，如 121，-623，91902.34688，等等。

除了十进制以外，JavaScript 还支持处理二进制、八进制、十六进制数字。这里简要介绍一下进制的概念。

十六进制（Hexadecimal）指逢16进1位的表示方法（通常的十进制是逢10进1），在十个阿拉伯数字之外，它还拓展了 a b c d e f 六个符号。0 ~ f 分别对应十进制的 0 ~ 15，十六进制的 10 就相当于十进制的 16 。十进制与十六进制的关系类似下面这样：

|  |  |  |  |
| --- | --- | --- | --- |
| 十六进制 | 十进制 | 十六进制 | 十进制 |
| 0 | 0 | b | 11 |
| 1 | 1 | c | 12 |
| 2 | 2 | d | 13 |
| 3 | 3 | e | 14 |
| 4 | 4 | f | 15 |
| 5 | 5 | 10 | 16 |
| 6 | 6 | 1f | 31 |
| 7 | 7 | ff | 255 |
| 8 | 8 | abc | 2748 |
| 9 | 9 | feff | 65279 |
| a | 10 | ffffffff | 4294967295 |

在 JavaScript 中使用十六进制数时，为了避免与标识符混淆，我们需要在数字前面加上前缀 0x，如：

0xff 0xfeff 0x1800 0xabcdef 0xffeeffee 0xcccccccc 0x5cf423

我们可以使用 alert 函数来显示它们对应的十进制形式。

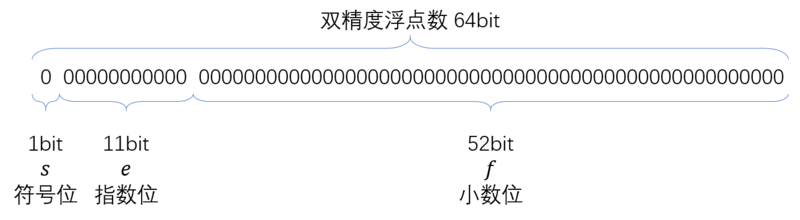
八进制和二进制的原理与十六进制类似，所不同的是：

* 八进制数字前需加上前缀 0 或 0o
* 二进制数字前需加上前缀 0b

### 数值的精度

众所周知，计算机中任何数据都是以二进制形式被存储和识别，数值自然不例外。JavaScript 中的数值类型被称为 number ，它的实现遵循国际标准 IEEE 754，被称为*64位双精度浮点数*。

根据 IEEE 754 的规定，一个64位双精度浮点数的存储方式如下：



由上图可知，JavaScript 中任何数值类型都使用 64 个二进制位来存储和表示。其中左数第一位记录了这个数值的正负符号，其后 11 位代表了这个数值的*指数位*，其后 52 位为小数位。

64 位双精度浮点数可以提供一个相当大的可供使用的数字范围。这个范围的上限约是 1.7976931348623157E+308，若遇到大于这个值的数值会导致*溢出*，即超出了 64 位双精度浮点数的最大可表示范围，JavaScript 中使用一个特殊值 Infinity 来表示溢出了的值，即“无穷大”。可用数值范围的下限自然是 -1.7976931348623157E+308，小于这个值的溢出数字则会表示为 -Infinity 。±Infinity 与除 NaN 之外的任何数值进行运算，结果依然是 ±Infinity 。在实际使用中，我们当然不用直接写出 ±1.7976931348623157E+308 这对极限值，JavaScript 提供了特殊的名称 Number.MAX\_VALUE 来表示最大可表示的正数，-Number.MAX\_VALUE 则是负数的下限。

JavaScript 提供了一个函数 isFinite 来判断给定值是否为 Infinity 。

alert(isFinite(Infinity)); // true  
alert(isFinite(Number.MAX\_VALUE)); // false  
alert(isFinite(123)); // false

同样地，JavaScript 中的数字不能极限逼近 0 。绝对值最小的可表示数值是 5E-324 ，它是能够表示的最接近 0 的数。JavaScript 提供了特殊名称 Number.MIN\_VALUE 来保存这个极限值。如果进一步缩小这个值，会发生溢出，导致舍入为 0。

alert(Infinity); // Infinity  
alert(Infinity + 1); // Infinity  
alert(-Infinity + 1); // -Infinity  
  
alert(Number.MAX\_VALUE); // 1.7976931348623157e+308  
alert(Number.MAX\_VALUE + 1); // 1.7976931348623157e+308  
alert(Number.MAX\_VALUE \* 1000); // Infinity  
alert(Number.MAX\_VALUE / 1000); // 1.7976931348623157e+305  
  
alert(Number.MIN\_VALUE); // 5e-324  
alert(Number.MIN\_VALUE - 1); // -1  
alert(Number.MIN\_VALUE \* 1000); // 4.94e-321  
alert(Number.MIN\_VALUE / 1000); // 0

在 JavaScript 中，我们可以随意使用 2-53+1 到 253-1 之间的所有整数，它们本身都是能够精确表示的。介于 253+1 和 Number.MAX\_VALUE 之间也可以使用科学计数法表示一些整数。越接近 Number.MAX\_VALUE ，能够精确表示的数越稀疏，其它的则会被近似处理。253-1 和 2-53+1 是可精确表示整数的上下限，JavaScript 提供了一对名称 Number.MAX\_SAFE\_INTEGER 和 Number.MIN\_SAFE\_INTEGER 来保存这对上下限，我们将处于这对上下限中的整数称为“安全整数”。一般情况下我们只与安全整数打交道。

除了安全整数以外的数字，我们将在下文中称为*浮点数*。浮点数包括小数和其他不属于安全整数范围内的数字（不包括 NaN 和 ±Infinity。

alert(Number.MAX\_SAFE\_INTEGER); // 9007199254740991  
alert(Number.MIN\_SAFE\_INTEGER); // -9007199254740991

在可精确表示的数值中，小数点后的位数最多可保存 16 位，如果实际位数多于 16 位，则会被舍弃。

alert(1 / 3); // 0.3333333333333333

由于64位双精度浮点数在运算方式上的一些特性，有些小数运算会出现一些看起来有些诡异的误差。

alert(0.1 + 0.2); // 0.30000000000000004  
alert(0.2 + 0.4); // 0.6000000000000001  
alert(0.3 + 0.6); // 0.8999999999999999

这类误差的产生涉及到浮点数的内部处理机制，限于篇幅不展开解释。在使用浮点数的时候注意到这种坑就行了。JavaScript 提供了一个*非常小*的特殊数值来表示这类误差，名为 Number.EPSILON 。如果计算结果减去目标值所得的值（可能为误差）小于 Number.EPSILON，即可确定计算结果与目标值实际上相等。

alert(Number.EPSILON); // 2.220446049250313e-16  
alert(0.1 + 0.2 - 0.3 < Number.EPSILON); // true（表示确认）  
alert(0.2 + 0.4 - 0.6 < Number.EPSILON); // true  
alert(0.3 + 0.6 - 0.9 < Number.EPSILON); // true

### 基本运算

JavaScript 中提供了一批运算符以供进行常见的数值运算。

加法运算： +

减法运算：-

乘法运算：\*

除法运算：/

求余运算：%

乘方运算：\*\*

正、负号：+ -

如果一个值使用运算符进行运算，我们称它为*运算数*。通过运算得到结果的过程称为*求值*。JavaScript 会对任何表达式进行求值。

let a = 10, b = 5;  
alert(a + b); // 15  
alert(a - b); // 5  
alert(a \* b); // 50  
alert(a / b); // 2  
alert(a % b); // 0  
alert(a \*\* b); // 100000

在 JavaScript 中，数值除以 0 会得到 Infinity。

alert(3 / 0); // Infinity  
alert(-3 / 0); // -Infinity

我们可以给一个数值自由地加上正负号，就像在草稿纸上做的那样。

我们可以在一个非数值的值前加上 +，将它转换为一个数值。如果转换失败，就会得到 NaN。NaN 将在下文解释。

let a = "123";  
alert(+a) // 123  
alert(+"3a"); // NaN

但是！有件事情请务必记住：在 JavaScript 中，**我们不能直接将负数当作乘方运算的底数**，否则会得到一个冗长的语法错误，这是为了避免*优先级冲突*，造成歧义。我们可以用括号将负数包裹起来。

alert(-10 \*\* 6); // SyntaxError: Unary operator used immediately before exponentiation expression. Parenthesis must be used to disambiguate operator precedence  
  
// 浏览器认为我们可能是想写出这样的表达式：-(10 \*\* 6)  
alert((-10) \*\* 6); // 1000000  
alert(-(10 \*\* 6)); // -1000000

JavaScript 提供了一批运算符来比较两个数值的关系。它们被称为*关系运算符*。

|  |  |
| --- | --- |
| 操作符 | 名称 |
| === | 严格相等 |
| !== | 严格不相等 |
| > | 大于 |
| < | 小于 |
| >= | 大于等于 |
| <= | 小于等于 |

这些操作会得到一个布尔值，用以决定下一步的逻辑。

alert(10 === 20); // false  
alert(10 !== 20); // true  
alert(10 > 20); // false  
alert(10 < 20); // true  
alert(10 >= 20); // false  
alert(10 <= 20); // true

有了关系运算符，结合条件表达式和逻辑运算，我们可以写出一些判断逻辑。

// 判断两个输入数值的大小   
const a = +prompt("请输入第一个值。");   
const b = +prompt("请输入第二个值。");   
let cond = a > b ? a : b;   
alert(cond + "更大。");

// 判断两个输入数值是否相差 20 以上。  
const a = +prompt("请输入第一个值。");  
const b = +prompt("请输入第二个值。");  
let c = a - b; // 求两数之差。  
let sub = c < 0 ? -c : c; // 如果 c < 0，那么差为 c 的绝对值，否则为 c 。  
let cond = c > 20; // 判断差值是否大于 20。  
alert(cond ? "相差 20 以上。" : "相差 20 以内。");

其中，计算 c 的绝对值这一操作，我们会在后面 Math 对象一节中了解到更好的方法。

Note: === 和 !== 为何会有“严格”二字？

是的，JavaScript 中除了这两个之外还有（非严格的）相等操作符==和不相等操作符 !=，它们在设计上存在缺陷，容易带来陷阱，甚至引起一些令人费解的比较结果。我们不推荐使用它们，当做 JavaScript 中的设计鸡肋即可。请使用严格相等操作符和严格不相等操作符。

假如我们要将一个变量的值进行运算，得到的结果还给这个变量，我们可以这么写：

let a = 10;  
a = a \* 10;  
alert(a); // 100  
a = a - 10;  
alert(a); // 0  
a = a + 10;  
alert(a); // 10  
a = a / 10;  
alert(a); // 1  
a = 50;  
a = a % 40;  
alert(a); // 10

但是 JavaScript 提供了几个特殊的赋值操作符，来更方便地做这些事情：

+= -= \*= /= \*\*=

它们可以用更简便的办法来完成上述代码的任务。

let a = 10;   
a \*= 10;   
alert(a); // 100   
a -= 10;   
alert(a); // 0   
a += 10;   
alert(a); // 10   
a /= 10;   
alert(a); // 1

JavaScript 中还提供了一些只有一个操作数的运算符，它们是*自增运算符*和*自减运算符*。如名称所描述的那样，它们可以改变变量本身的值，将其 +1 或者 -1。如果位于操作数前面，那么它会先改变操作数的值，然后得到这个新的值。如果位于操作数后面，我们会先得到这个操作数本来的值，然后操作数发生改变。

let a = 10;  
alert(a++); // 10，然后 a 变为 11  
alert(++a); // 12  
alert(a--); // 12，然后 a 变为 11  
alert(--a); // 10  
  
let b = 20;  
alert(b++ + ++b); // 42  
let c = 20;  
alert(--c + c--); // 38  
alert(c); // 18

它们存在于 JavaScript 中是因为历史遗留问题。我们建议尽量不要使用它们，以避免不必要的混淆。使用上面提到的赋值运算符吧。

### NaN

在 JavaScript 中有一个特殊的“数值”——NaN，它表示“不是一个数值（Not a Number)”。当我们期望一个值应该是数值，可是却无法采取手段将它转换为数值时，就会得到这个值。NaN 也会在某些没有意义的运算中也会作为结果出现。

let a = "123a";  
alert(+a); // NaN  
alert(Number(a)); // NaN  
alert(Infinity % 0); // NaN

* NaN 与任何值进行运算都是 NaN，包括 Infinty。
* NaN 不等于任何值，**甚至不等于它自身**。换句话说：

alert(NaN === NaN); // false

我们可以用这一特性来判断一个值是否为 NaN（即，是否不等于它自己）。如果我们不仅仅是检查 NaN 这个值，而是要排除掉一切不是数字的值，可以使用 JavaScript 中提供的函数 isNaN 。如果检查的值就是数字或者可以被转换为数字，那么它会得到 false ，反之为 true。

alert(isNaN(NaN)); // true  
alert(isNaN("Hello")) // true  
alert(isNaN(true)); // false  
alert(isNaN("")); // false  
alert(isNaN(100)); // false  
alert(isNaN(Infinity)); // false  
alert(isNaN(Infinity \* 10)); // false  
alert(isNaN(Infinity % 0)); // true  
alert(isNaN(NaN + 10)); // true  
alert(isNaN(+"10")); // false  
alert(isNaN(+"10a")); // true

### parseInt 和 parseFloat

我们已经在前面了解了 + 可以直接对非数字值进行转换，但是如果无法进行转换，就会得到一个令人失望的 NaN 值。所幸 JavaScript 提供了两个函数以供更好、更安全地将字符串转换为数字。

parseInt 从左往右读取字符串，如果读取到的内容可以被解析为安全整数，那么它就会进行转换，否则就会停止读取，并得到转换后的数字。

* 如果读取到的数字是一个整数而不是浮点数，但是太大无法处理，会得到 ±Infinity。
* 如果根本无法读取到整数，会得到 NaN;

let str = "12345.6789abc";  
let number = parseInt(str);  
alert(number); // 12345  
  
let str2 = "0xabcdefghi";  
let number2 = parseInt(str2);  
alert(number2); // 11297375  
  
let str3 = "123E+456ab";  
let number3 = parseInt(str3);  
alert(number3); // 123  
  
let str4 = "abcdef";  
let number4 = parseInt(str4);  
alert(number4); // NaN

* 在第一个示例中，parseInt 遇到了小数点，因此停止读取，将 "12345" 转换为了数字 12345。
* 在第二个示例中，parseInt 首先遇到了 "0x"，知道后面的是十六进制数，读取到 "f" ，由于 "g" 不是十六进制数字，因此停止读取，得到的数字是 0xabcdef，alert 用十进制方式显示就是 11297375。
* 在第三个示例中，由于需要使用 E （即科学计数法）表示的数不是安全整数，只会读取到 "123"。
* 在第四个示例中，由于无法读取到整数，得到 NaN。

parseFloat 从左往右读取字符串：

* 如果读取到的内容可以被解析为整数或浮点数，那么它就会进行转换，否则就会停止读取，并得到转换后的数字。
* 如果读取到的小数位数太多，就会进行四舍五入。
* 如果读取到的整数超出了安全整数的范围，那么会将其处理成浮点数。
* 如果太大或太小，会得到 ±Infinity。
* 如果无法读取到一个数字，会得到 NaN。
* 注意：parseFloat 只能处理十进制数，如果在读取时遇到 0x 0b 0o 这些标记，它只会得到 0。

let str = "12345.6789abc";  
let number = parseFloat(str);  
alert(number); // 12345.6489  
  
let str2 = "0xabcdefghi";  
let number2 = parseFloat(str2);  
alert(number2); // 0  
  
let str3 = "123E+456ab";  
let number3 = parseFloat(str3);  
alert(number3); // Infinity  
  
let str4 = "abcdef";  
let number4 = parseFloat(str4);  
alert(number4); // NaN

* 在第一个示例中，12345.6789 是一个浮点数，因此得到它。
* 在第二个示例中，parseFloat 读取到 0，由于 x 不属于浮点数标记，停止读取，得到 0。
* 在第三个示例中，读取到的浮点数为 123E+456，但是它太大无法处理，得到 Infinity。
* 在第四个示例中，根本无法读取到数字，得到 NaN。

### 数学函数

如果我们要计算一道小学的数学应用题，那么 JavaScript 中所提供的数学运算符是完全够用的。但当我们需要进行更加复杂的数学运算（例如开平方根和三角函数等），就需要求助于 JavaScript 中提供的一系列**数学函数**。

这些数学函数的名称遵循一个统一的形式：

Math.<函数名>

其中 Math 是一个*全局对象*，我们将会在后文中介绍这一概念。我们只需要记住：当我们需要进行加减乘除以外的四则运算时，就要写出它，然后用一个点号 . 分隔，然后写出我们所需的具体函数名称。

常用的数学函数例如：

* Math.floor 得到一个数字向下取整后的结果（小于等于这个数的最大整数）
* Math.ceil 得到一个数字向上取整后的结果（大于等于这个数的最小整数）
* Math.round 得到一个数字四舍五入后的结果
* Math.pow 接受两个值：a 和 n，得到 an（已被 \*\* 运算符代替）
* Math.sqrt 求一个数的平方根
* Math.sin Math.cos Math.tan 等都是三角函数。
* 它们所进行的运算就像它们的名字所提示的那样。
* Math.random 得到一个介于 0 和 1 之间的*随机数*。

数学函数的使用方式大致如下：

let number = 1234.5678;  
alert(Math.floor(number)); // 1234  
alert(Math.ceil(number)); // 1235  
alert(Math.round(number)); // 1235  
alert(Math.pow(10, 5)); // 100000  
alert(Math.sqrt(3)); // 1.7320508075688772  
alert(Math.random()); // 0.6451182481258273

除了数学函数外，Math 对象中还提供一些*数学常量*，例如 ， 等，它存放了这些数学常量能够在 JavaScript 中数字存储的近似值，使我们免于每次使用都需要手动输入它们的值。

alert(Math.PI); // 3.141592653589793  
alert(Math.E); // 2.718281828459045

Note:

JavaScript 中的三角函数得到的是弧度值。可以通过除法（Math.PI / 180）把弧度转换为角度。

### 随机数

随机数在一些诸如幸运开奖，或者掷骰子等地方发挥着重要作用。JavaScript 中的 Math.random 函数会在每次使用时得到一个介于 0 和 1 之间的随机数，它有可能（但是概率非常小）得到 0，而不会出现 1。随机数的大致出现频率遵循*正态分布*。

alert(Math.random()); // 0.7676430146750075  
alert(Math.random()); // 0.14541252190516185  
alert(Math.random()); // 0.5985808933645129  
alert(Math.random()); // 0.20314278019751697  
alert(Math.random()); // 0.26858695307604075  
alert(Math.random()); // 0.7388373263409738  
alert(Math.random()); // 0.3131427029040914  
alert(Math.random()); // 0.3509369624385763

一个介于 0 ~ 1 的随机数能干什么呢？它的范围显然太小，无法满足我们的诸多需要。

我们可以将得到的原始随机数乘上一个值，使它可能存在的范围变大。

let r = Math.random();  
alert(r); // 0.4994890897612041  
alert(r \* 2); // 0.9989781795224082  
alert(r \* 10); // 4.994890897612041  
alert(Math.random() \* 100); // 60.89928523091233  
alert(Math.random() \* 2000); // 528.3919941661499

我们设置了一个随机数生成的范围，但是得到的结果还有一个“又臭又长”的小数位。想想看，我们在掷骰子的时候，得到的随机结果会是一个模棱两可的小数吗？好在我们已经知道了 JavaScript 中的取整函数，可以直接运用到生成的随机数上：

alert(Math.floor(Math.random() \* 6)); // 4  
alert(Math.floor(Math.random() \* 6)); // 2  
alert(Math.floor(Math.random() \* 6)); // 5  
alert(Math.floor(Math.random() \* 6)); // 0

现在我们可以晃动这个随机数骰子，每次得到一个 0 ~ 6 之间的整数。

但是等等！如果我想微调一下这个范围，变为 1 ~ 7 呢？

并没有什么难的。想做什么，去做就行了。Go it now！

alert(Math.floor(Math.random() \* 6) + 1); // 5  
alert(Math.floor(Math.random() \* 6) + 1); // 1  
alert(Math.floor(Math.random() \* 6) + 1); // 3  
alert(Math.floor(Math.random() \* 6) + 1); // 7

如果我们所需的随机数范围不是 0 ~ n，而需要我们自己来制定上限和下限 m ~ n，只需要稍微改变一下写法。

let m = 10, n = 20;  
alert(Math.random() \* (n - m) + m); // 14.453251269589478  
alert(Math.floor(Math.random() \* (n - m) + m)); // 16

事实上，我们也可以使用另两个取整函数：Math.ceil 和 Math.round，但是大量的实践表明，使用 Math.floor 对随机数进行取整可以得到*出现频率更平均的整数*。

## 字符串

### 字符串的概念

字符串即我们通常所说的*文本*。它来自计算机深处，却拥有人类可读的形式。它是在人机之间传递信息的使者，一座字符堆垒起来的桥梁。它的定义形式是一对引号 "" 或 '' ，引号中的内容就是它的全部。我们可以使用 + 号来拼接两个字符串，使用 length 属性来获得它的长度，使用下标 [] 来获得它在某个位置上的字符。

const s = "Hello ";  
alert(s.length); // 6  
alert(s + "world"); // "Hello world"  
alert(s + 'JavaScript'); // "Hello JavaScript"

一个字符串中的每一个字符都具有一个编号，这个编号从 0 开始，最后一个字符的编号就是它的长度减去 1。如果指定编号超出了限度，则会得到一个 undefined。

const s = "Hello world";  
alert(s[0]); // "H";  
alert(s.length); // 11  
alert(s[s.length]); // undefined  
alert(s[s.length - 1]); // "d"  
alert(s[-1]); // undefined

像数值一样，你可以使用 === 和 !== 来比较两个字符串是否完全相等。

let a = "aaa", b = "bbb";  
alert(a === b); // false  
alert(a !== b); // true  
alert(a === "aaa"); // true

练习 3.1.1

1. 编写一个程序，使用 prompt 函数得到用户输入的的名字，用 alert 对这个名字打招呼。
2. 编写一个程序，从用户输入的字符串中获得一个随机位置上的字符。（备注：你可能需要复习“数值”一节。）

### 转义字符

有时我们需要在字符串中使用一些特殊的字符，这些字符无法用通常的方式输入，例如换行符。我们可以使用一类称为*转义字符*的标记来表示这些字符。它们的原理是，通过在普通的字符前加上符号 \ ，来改变这个字符本来的含义或者作用。JavaScript 规定了一些转义字符，如下表。

|  |  |
| --- | --- |
| 转义字符 | 含义 |
| \\ | 反斜杠本身 |
| \n | 换行符 |
| \r | 回车符 |
| \t | 水平制表符 |
| \v | 垂直制表符 |
| \b | 退格符 |
| \f | 换页符 |
| \u | Unicode 码点 |

以下是一些使用示例：

alert("Hello\\world!"); // Hello\world!  
alert("Hello\nworld!"); // Hello  
 // world!  
alert("Hello\rworld!"); // Hello  
 // world!  
alert("Hello\tworld!"); // Hello world!  
// \v \b \f 三个字符在一般的文本编辑器中无法显示  
alert("Hello\vworld!"); // Hello□world!  
alert("Hello\bworld!"); // Hello□world!  
alert("Hello\fworld!"); // Helloworld!

一般情况下我们只需要使用 \\ 来得到 \ 字符本身，或者 \n 作为换行符即可。\u 表示一个 Unicode 码点，关于它的详细内容见下文。

**转义引号**

假如你的字符串里面需要包含单引号和双引号，但是又不能与表示字符串开头结尾的引号冲突，可以用*转义引号*的办法来规避。你可以在字符串中出现的引号前加 \ 以进行转义，如果不会发生冲突，则不需要转义。

const s1 = "He said: \"Hello world!\""; // He said: "Hello world!"  
const s2 = "He said: "Hello world!""; // SyntaxError: Unexpected identifier  
const s3 = "He said: \'Hello world!\'"; // He said: 'Hello world!'  
const s4 = "He said: 'Hello world!'"; // He said: 'Hello world!'

### Unicode

在深入探讨 JavaScript 中的字符串之前，我们首先要了解一下 **Unicode**。

Unicode 是世界上最为通用的字符集，它可以看做一切其他字符集（如 ASCII，GBK 等）的合体，涵盖了目前世界上几乎所有已知的现存书写系统，从欧洲的拉丁和西里尔字母，到远东的汉字、日文、韩文，再到东南亚圆润的字母文字、印第安人的奇特符号，甚至盲文、emoji ，都在 Unicode 这一字符集的涵盖范围内。它的开发与实现遵循 ISO 的国际标准，有许多具体的方式来处理遵循 Unicode 标准的文本内容，包括 UTF-8、UTF-16、UTF-32 等。JavaSciript 中的字符串使用 Unicode 作为处理依据，以便充分融入国际化的 Web 环境中。

Unicode 源于一个很简单的想法：将全世界所有的字符包含在一个集合里，计算机只要支持这一个字符集，就能显示所有的字符，再也不会有乱码了。

——阮一峰（2014年）

Unicode 的核心概念是，从 0 开始，为每一个包含在这个字符集中的字符分配一个独一无二的数字编号，称为“码点（code point）”，并将相应的字形和意义与这个编号一一对应。如：U+0000，U+0FE3，U+CFFF 等。

前缀 U+ 表示紧跟在后面的十六进制数是一个 Unicode 码点，我们一般习惯使用十六进制数来表达 Unicode 码点。

例如，中文“好”的码点是：U+597D。

Unicode 中不同的符号不是一次性全部定义的，而是分成多个区域，每个区域可以存放 216 （65536）个字符，称为一个平面。目前一共有 17 个平面，也就是说，整个 Unicode 字符集的大小现在是 221。

最前面的 65536 个字符位，称为*基本多文种平面*（缩写为 BMP），它的码点范围是从 0 一直到 216-1 ，写成 16 进制就是从U+0000 到 U+FFFF。所有最常见的字符都放在这个平面，这是 Unicode 最先定义和公布的一个平面。剩下的字符都放在辅助平面（缩写为 SMP），码点范围从 U+010000 一直到 U+10FFFF。

**1. UTF-32**

Unicode 只规定了每个字符的码点作为统一标准，而在实际应用中基于 Unicode 标准有多种具体实现方式，它们统称为*编码方法*。最直观的编码方法是，每个码点使用八个十六进制数（即四个字节）表示，字节内容完全对应码点。这种编码方法称为 **UTF-32** 。比如，码点 U+0000 就用四个字节的 0 表示，码点597D就在前面加两个字节的0。

使用 UTF-32 方式编码的“好”和 U+0000（空字符）如下：

U+0000 => 0x0000 0000

U+597D => 0x0000 597D

UTF-32 的优点在于：

1. 转换规则简单直观
2. 查找效率高。

缺点在于：**浪费空间**，同样一份英语文本，用它进行编码所占用的空间是原始的 ASCII 编码的四倍。这是致命的缺点，因此实际上没有人使用这种编码方法，目前的互联网页面标准（HTML 5）就明文规定，网页不能以 UTF-32 方式进行编码。

**2. UTF-8**

人们需要的是一种节省空间的编码方法，于是 **UTF-8** 应运而生。UTF-8 是一种*变长*的编码方法，一个字符的编码长度，从 1 个字节到 4 个字节不等。常用的字符编码较短，而最前面的 128 个字符，只使用 1 个字节表示，与 ASCII 的方式完全相同。

如下表：

|  |  |
| --- | --- |
| 编码范围 | 占用字节数 |
| 0x0000 - 0x007F | 1 |
| 0x0080 - 0x07FF | 2 |
| 0x0800 - 0xFFFF | 3 |
| 0x010000 - 0x10FFFF | 4 |

UTF-8 对存储空间的节省使得它成为互联网上最常用的编码方式。

**3. UTF-16**

**UTF-16** 编码方式介于 UTF-32 与 UTF-8 之间，同时结合了*定长*和*变长*两种编码方法的特点。

它的编码规则很简单：基本多文种平面的字符占用 2 个字节，其余的（不那么常用）的字符占用 4 个字节。也就是说，UTF-16 的编码长度要么是 2 个字节（U+0000 到 U+FFFF），要么是 4 个字节（U+010000 到 U+10FFFF）。根据字符的 Unicode 码点进行相应的 UTF-16 编码的时候，首先区分这是基本多文种平面的字符（码点 U+FFFF 以内），还是辅助平面字符。如果是前者，直接将码点转为对应的十六进制形式，长度为 2 字节。

如： U+597D => 0x597D

而辅助平面字符的编码则根据 Unicode 3.0 标准给出的公式，用 JavaScript 代码编写如下：

// 假设 c 是待编码的字符码点  
const H = Math.floor((c - 0x10000) / 0x400) + 0xD800;  
const L = (c - 0x10000) % 0x400 + 0xDC00;

其中 H 就是最终编码的左 4 位十六进制数，L 就是右 4 位。用一个码点为 U+1D306 的字符演示：

// const c = 0x1D306;  
const H = Math.floor((0x1D306 - 0x10000) / 0x400) + 0xD800;  
const L = (0x1D306 - 0x10000) % 0x400 + 0xDC00;  
// H = 0xD834, L = 0xDF06  
// U+1D306 的 UTF-16 编码即 0xD834 DF06

**4. UCS-2**

JavaScript 所采用的编码方式称为 *UCS-2*，它的出现是基于历史原因，可以看做 UTF-16 的子集。因此，JavaScript 中字符串的每一个字符都至少占用 2 字节空间，BMP 以外的字符则通过两个 2 字节的字符来表示这个字符的编码。这种实现方式称为*代理对*，相关细节可参见本书附录。

2015 年的 ECMAScript 6 标准提供了另一种更加方便的表示非 BMP 字符，可以使用一对大括号将字符码点包裹起来，写在 \u 前缀后面：

alert("\u{1316c}"); // "𓅬"（古埃及象形文字“鹅”）

在 JavaScript 的字符串中，你可以直接打出一个字符并放在字符串中，也可以使用 *Unicode 转义标记*来输入 Unicode 中的其它字符。转义标记写作 \u ，后接字符码点的十六进制表示。如数字表示方式一样，JavaScript 中的十六进制数字不区分大小写。注意：**你至少要写 4 位十六进制数字**，否则会得到一个错误。如下：

alert("\u0041"); // "A"  
alert("\u41"); // SyntaxError: Invalid Unicode escape sequence  
alert("\u597D"); // "好"   
alert("\u4f60\u597d"); // "你好"  
alert("\u3053\u3093\u306b\u3061\u306f"); // "こんにちは"  
alert("\uc5b4\ub5bb\uac8c 지내니"); // "어떻게 지내니"  
alert("\u2600"); // "☀"  
alert("\u2614"); // "☔"  
alert("\u3a3\u222b\u221e"); // "Σ∫∞"  
alert("我是\u5c0f\u53ef\u7231！"); // "我是小可爱"

### 字符串操作

JavaScript 提供了一组实用的字符串操作函数，以便于完成诸多常见的文本操作需求。

**大小写转换**

toUpperCase 和 toLowerCase 两个方法用于转换一个字符串中的大小写，并返回转换后的结果。

const s = "Hello world";  
alert(s.toUpperCase()); // "HELLO WORLD"  
alert(s.toLowerCase()); // "hello world"  
alert("Madam, I'm Adam.".toUpperCase()) // "MADAM, I'M ADAM."

**替换**

trim、trimLeft 和 trimRight 方法分别去除字符串左右的空格、左边的空格、右边的空格。

alert(" 精 罗 震 怒 ".trim()); // "精 罗 震 怒"  
alert(" 精 罗 震 怒 ".trimLeft()); // "精 罗 震 怒 "  
alert(" 精 罗 震 怒 ".trimRight()); // " 精 罗 震 怒"

replace 方法用于在给定字符串中替换第一个匹配的文本。

const s = "Hello world";  
const s2 = s.replace("world", "JavaScript");  
alert(s2); // "Hello JavaScript"  
alert(s.replace("l", "k")); // "Heklo world"

replace 函数的第一个参数也可以是一个*正则表达式*，用于描述更加复杂的模式或者进行全局性的替换。关于正则表达式的概念和细节我们将在第八章讨论。

**字符串检测**

有三种简单的方法可以用于判断字符串中是否包含另一个字符串的内容。

* includes()：返回布尔值，表示是否包含参数的字符串。
* startsWith()：返回布尔值，表示参数字符串是否在原字符串的头部。
* endsWith()：返回布尔值，表示参数字符串是否在原字符串的尾部。

const string = "Hello world";  
alert(string.startsWith("Hello")); // true  
alert(string.endsWith("rld")); // true  
alert(string.includes("o")); // true

这三个方法也可以接受第二个参数，表示开始查找的位置。

const string = "Hello world";   
alert(string.startsWith("world", 6)); // true  
alert(string.endsWith("Hello", 5)); // true  
alert(string.includes("Hello", 6)); // false

如果接受了第二个参数，endsWith 方法的行为与其它两个有所不同。它针对前 n 个字符，而其它两个方法则是从第 n 个字符开始，直到字符串结束。

**提取字符串**

slice 方法从字符串中提取一个片段，第一个参数指定从第几个字符开始提取，如果是负数，则该参数规定的是从字符串的尾部开始算起的位置。也就是说，-1 指字符串的最后一个字符，-2 指倒数第二个字符，以此类推。

而第二个参数是可选的，决定到什么位置结束。如果没有第二个参数，那么截取到字符串末尾。如果它是负数，那么它规定的是从字符串的尾部开始算起的位置。

const string = "君不见黄河之水天上来，奔流到海不复回。"  
alert(string.slice(8)); // "上来，奔流到海不复回。"  
alert(string.slice(4, 19)); // "河之水天上来，奔流到海不复回。"  
alert(string.slice(-4)); // "不复回。"  
alert(string.slice(-9, -5)); // "，奔流到"

substr 方法从指定位置开始，提取字符串中指定数目的字符。它的参数规则与 slice 类似，只不过如果指定第二个参数，则必须为正整数。

const string = "君不见黄河之水天上来，奔流到海不复回。";  
alert(string.substr(5, 10)); // 之水天上来，奔流到海  
alert(string.substr(0, 5)); // 君不见黄河  
alert(string.substr(10)); // ，奔流到海不复回。

另一个截取字符串的方法是 substring。与 slice 不同，substring 一定以两个参数中较小一个作为起始位置，较大的参数作为结束位置。当参数是负数时，slice 会以较大的一个作为起始位置，而 substring 则将负参数都看做 0。

const string = "君不见黄河之水天上来，奔流到海不复回。"  
alert(string.slice(-3)); // 复回。  
alert(string.substring(-3)); // 君不见黄河之水天上来，奔流到海不复回。  
alert(string.slice(3, -4)); // 黄河之水天上来，奔流到海  
alert(string.substring(3, -4)); // 君不见

**字符位置**

charAt 方法都可以得到指定位置的字符，而 charCodeAt 得到指定位置字符的 Unicode 码点。一些字符使用了代理对，它们以多个字符的形式存储，charCodeAt 只会单纯获取指定位置上的字符，它可能是一个代理字符，并不是我们想要的。因此 codePointAt 方法会将使用了代理对的字符视作一个整体，获取其码点。

const string = "Hello world";  
alert(string.charAt(0)); // "H"  
alert(string.charCodeAt(0)); // "72"  
alert(string.codePointAt(0)); // "72"  
  
const string2 = "\u{1f511}\u{1f512}"; // 钥匙和锁的 Emoji 字符  
alert(string2.charAt(0)); // 🔑  
alert(string2.charCodeAt(0)); // 55357（代理字符的码点）  
alert(string2.codePointAt(0)); // 128273（这个 Emoji 字符的实际码点）

**构造字符串**

repeat 方法返回一个新字符串，表示将源字符串重复 次。

* 如果参数是小数，会被向下取整。
* 如果参数是负数、±Infinty 或者超过 JavaScript 中字符串的最大长度限制，会报错。

"QwQ".repeat(2); // "QwQQwQQwQ"  
"QwQ".repeat(0); // ""  
"QwQ".repeat(2.9) // "QwQQwQ"  
"QwQ".repeat(Number.MAX\_SAFE\_INTEGER); // RangeError: Invalid string length  
"QwQ".repeat(-666); // RangeError: Invalid count value

ECMAScript 8 标准引入了字符串补全长度的功能。如果某个字符串的长度没有达到指定长度，会在头部或尾部补全。其中 padStart会将用于补全的字符串填充在源字符串头部，而 padEnd 则会在尾部填充。

* 如果原字符串的长度，等于或大于指定的最小长度，则返回原字符串。
* 如果省略第二个参数，默认使用空格补全长度。

alert("奶茶".padStart(5, "哈嚯")); // 哈嚯哈奶茶  
alert("奶茶".padStart(4, "哈嚯")); // 哈嚯奶茶  
alert("奶茶".padEnd(5, "哈嚯")); // 奶茶哈嚯哈  
alert("奶茶".padEnd(4, "哈嚯")); // 奶茶哈嚯  
alert("奶茶奶茶奶茶".padStart(2, "哈嚯")); // 奶茶奶茶奶茶  
alert("奶茶奶茶奶茶".padEnd(2, "哈嚯")); // 奶茶奶茶奶茶  
alert("哈嚯嚯".padStart(10, "0123456789")); // 0123456哈嚯嚯  
alert("奶茶".padStart(4)); // 奶茶  
alert("奶茶".padEnd(4)); // 奶茶

padStart的常见用途是为数值补全指定位数。

"1".padStart(10, "0") // "0000000001"  
"12".padStart(10, "0") // "0000000012"  
"123456".padStart(10, "0") // "0000123456"

另一种用法是将数据填充进格式字符串里。

"06".padStart(10, "YYYY-MM-DD"); // "YYYY-MM-06"  
"02-06".padStart(10, "YYYY-MM-DD"); // "YYYY-02-06"  
"2005-02-06".padStart(10, "YYYY-MM-DD"); // "2005-02-06"

JavaScript 的字符串方法中，有一些根据正则表达式，对文本进行处理，包括 replace 、search、match 等。正则表达式是一类非常强大也非常复杂的技术，可以使文本处理变得如虎添翼，关于正则表达式的概念和内容我们暂时不需要了解，只需知道它们是一类处理文本的辅助工具。此外，JavaScript 还提供了一些与*本地化*相关的字符串方法，它们为适用于不同地区、文化、语言习惯的应用程序提供帮助。关于正则表达式与本地化的细节我们会在后面的章节中了解到。

### 转换规则

* 数字和字符串相加，会将数字转换为十进制形式，与字符串拼接，得到拼接后的结果。
* alert(1 + "1"); // "11"  
  alert("My number is " + 15 + "2"); // "My number is 152"  
  alert("My number is " + 15 + 2); // "My number is 152"
* 数字与字符串相减，会先尝试将字符串转换为数值，然后进行相减。
* alert(100 - "10"); // 90  
  alert("100" - 10); // 90  
  alert("100" - "10"); // 90
* 布尔值在参与数字运算的时候, true 会被转换为 1，false 会被转换为 0，然后进行运算。
* alert(1 + true); // 2  
  alert(1 + false); // 1
* 布尔值与字符串相加，会将布尔值直接转换为字符串。
* alert("I think it is " + true); // "I think it is true"  
  alert("Oh, it's " + false); // "Oh, it's false"
* 布尔值与字符串相减，则会依据上述转换规则，将布尔值和字符串分别转换为数字，然后进行运算。
* alert(true - "1"); // 0  
  alert(false - "10"); // -10

### 模板字符串

假如我们需要在一个字符串里写长长的一段话：

Lorem ipsum dolor sit amet, consectetur adipiscing elit.  
Sed eleifend vitae massa sed porttitor. Aliquam erat volutpat.  
Maecenas feugiat, urna sit amet feugiat gravida,  
felis ante lobortis tortor, vel dictum enim sem vitae eros.  
Vivamus mi eros, feugiat ut ex laoreet,  
commodo mattis nisi. Praesent mollis augue eu ligula scelerisque,  
et accumsan mauris pretium. Cras efficitur vel elit eu varius.  
Integer luctus facilisis dignissim.  
Duis pretium lorem nec risus posuere euismod.  
Quisque leo erat, suscipit eget metus vitae,  
accumsan accumsan ex. Curabitur mattis non neque at hendrerit.  
Praesent sollicitudin, nibh quis maximus vestibulum,  
risus ipsum tempus leo, nec imperdiet quam purus eget sem.  
Proin lectus nibh, viverra et vestibulum sed, lacinia ut ipsum.

这段话很长，中间夹杂着许多换行。而 JavaScript 中本来的字符串定义方式是不支持直接换行的，连成一段就失去了美感，该怎样解决呢？

一个最直观的办法是这样：

const s = "Lorem ipsum dolor sit amet, consectetur adipiscing elit." +  
"Sed eleifend vitae massa sed porttitor. Aliquam erat volutpat.\n" +  
"Maecenas feugiat, urna sit amet feugiat gravida,\n" +  
"felis ante lobortis tortor, vel dictum enim sem vitae eros.\n" +  
"Vivamus mi eros, feugiat ut ex laoreet,\n" +  
"commodo mattis nisi. Praesent mollis augue eu ligula scelerisque,\n" +  
"et accumsan mauris pretium. Cras efficitur vel elit eu varius,\n" +  
"Integer luctus facilisis dignissim.\n" +  
"Duis pretium lorem nec risus posuere euismod.\n" +  
"Quisque leo erat, suscipit eget metus vitae,\n" +  
"accumsan accumsan ex. Curabitur mattis non neque at hendrerit.\n" +  
"Praesent sollicitudin, nibh quis maximus vestibulum,\n" +  
"risus ipsum tempus leo, nec imperdiet quam purus eget sem.\n" +  
"Proin lectus nibh, viverra et vestibulum sed, lacinia ut ipsum.\n";

将字符串根据换行拆成许多小的字符串，每个字符串末尾使用 \n 标记换行，使用 + 一个一个进行拼接。很长时间里人们用的就是这种办法。

如果不考虑文本呈现出来的模样，只是在代码里美观一些，也有一种简便点的解决办法：

const s = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.\  
Sed eleifend vitae massa sed porttitor. Aliquam erat volutpat.\  
Maecenas feugiat, urna sit amet feugiat gravida,\  
felis ante lobortis tortor, vel dictum enim sem vitae eros.\  
Vivamus mi eros, feugiat ut ex laoreet,\  
commodo mattis nisi. Praesent mollis augue eu ligula scelerisque,\  
et accumsan mauris pretium. Cras efficitur vel elit eu varius.\  
Integer luctus facilisis dignissim.\  
Duis pretium lorem nec risus posuere euismod.\  
Quisque leo erat, suscipit eget metus vitae,\  
accumsan accumsan ex. Curabitur mattis non neque at hendrerit.\  
Praesent sollicitudin, nibh quis maximus vestibulum,\  
risus ipsum tempus leo, nec imperdiet quam purus eget sem.\  
Proin lectus nibh, viverra et vestibulum sed, lacinia ut ipsum.";

通过在每一行末尾加上 \ 来将换行*转义*，使其不被认为是语法上一行的截止，连成一整个字符串。

假如我们需要在字符串中插入其它一些运行时才确定的内容，比如一个表达式的运算结果，我们可以使用字符串拼接，将表达式的值拼接在两个字符串中，像这样：

alert("1 + 13 - 32 + 11 + 53 - 29 的结果是" + (1 + 13 - 32 + 11 + 53 - 29) + "。");

如果要拼接的表达式不多还好，多了写起来可就麻烦了。同时考虑到多行字符串写起来的种种不便，在 ECMAScript 6 的标准中规定了一种新的字符串定义法——**模板字符串**。

模板字符串能够扫除已有的问题，它使用 ` （反引号）来标记字符串的开始和结束。一个模板字符串可以像这样使用：

const s = `Lorem ipsum dolor sit amet, consectetur adipiscing elit.  
Sed eleifend vitae massa sed porttitor. Aliquam erat volutpat.  
Maecenas feugiat, urna sit amet feugiat gravida,  
felis ante lobortis tortor, vel dictum enim sem vitae eros.  
Vivamus mi eros, feugiat ut ex laoreet,  
commodo mattis nisi. Praesent mollis augue eu ligula scelerisque,  
et accumsan mauris pretium. Cras efficitur vel elit eu varius.  
Integer luctus facilisis dignissim.  
Duis pretium lorem nec risus posuere euismod.  
Quisque leo erat, suscipit eget metus vitae,  
accumsan accumsan ex. Curabitur mattis non neque at hendrerit.  
Praesent sollicitudin, nibh quis maximus vestibulum,  
risus ipsum tempus leo, nec imperdiet quam purus eget sem.  
Proin lectus nibh, viverra et vestibulum sed, lacinia ut ipsum.`;

如果你将 s 显示出来，就会发现它完全记录了*原文*。不但字符串中间可以直接换行，换行还可以被直接记录下来，不需要再使用单独的 \n 来标记换行。

但是模板字符串的方便之处不仅仅止于此。请看：

const a = 3, b = 2;  
const s = `I had ${a} apples, and Lily gave me ${b} apples.  
Now I have ${a + b} apples.`  
alert(s); // I had 3 apples, and Lily gave me 2 apples.  
 // Now I have 5 apples.

在模板字符串中，你可以使用 ${} 标记来插入一个表达式，这个表达式的值可以直接被插入最终的字符串中，省去了反复拼接的麻烦。如果你在字符串中需要使用 ${} 这三个字符本身，直接使用\进行转义即可：

const a = 3, b = 2;  
const s = `I had \${a} apples, and Lily gave me \${b} apples.  
Now I have \${a + b} apples.`  
alert(s); // I had ${a} apples, and Lily gave me ${b} apples.  
 // Now I have ${a + b} apples.

一个模板字符串和普通字符串在意义和使用上是基本一致的。

**标签模板**

模板字符串的功能不止于此。它可以紧跟在一个函数名后面，该函数将被调用来处理这个模板字符串。这被称为 “ *标签模板* ” 功能（ tagged template ）。

alert`123` // 123  
// 等同于  
alert(123) // 123

前面说过模板字符串和普通字符串在使用上基本一致，因此普通字符串中的转义字符在模板字符串中依然会被转义。如：

alert`Hello\nworld`; // Hello  
 // world

我们可以使用 String.raw 函数，来使模板字符串真正“如实”记录我们写下的内容。

alert(String.raw`Hello\nworld`); // Hello\nworld

String.raw 虽然是一个函数，但是不能使用括号进行调用，自然也不能用于普通字符串。

alert(String.raw(`Hello\nworld`)); // TypeError: Cannot convert undefined or null to object   
alert(String.raw("Hello\nworld")); // TypeError: Cannot convert undefined or null to object   
alert(String.raw"Hello\nworld"); // SyntaxError: missing ) after argument list

## 对象

### 对象的概念

**构造器与字面量**

*字面量*是一种值的表示法。通过直接写出一个值所包含的内容，来创建这个值，这种**形式**被称为字面量。

如："Hello world" 就是一个字符串字面量，1234.5678 就是一个数字字面量。

字面量与*创建实例*相对。在 JavaScript 中，每种类型的值都有一个对应的*构造器*，原则上我们可以使用构造器来构造一个值，构造器写成函数调用的形式，可以接受一些值，作为创建新值的所需信息，这种创建值的形式称为*创建实例*。如：

const a = new Number(); // a = 0  
const b = new Number(10); // b = 10  
const c = new String(); // c = ""  
const d = new String(100); // c = "100"

当然，JavaScript 中提供了方便的*字面量*方式来让我们创建值，我们不需要为值一个个创建实例。

除了我们已经了解过的布尔值、数值、字符串之外，有一大类值被称为*对象*。对象是一个包含了一些值的结构。

const o = new Object();  
o.name = "John";  
o.age = 12;

现在我们创建了一个对象，并将它赋给常量 o 。我们为 o 提供了两个信息，分别是名称和年龄。这个信息被称作*属性*，我们可以通过属性来访问对象中所存放的具体的值。

const cat = new Object();  
cat.name = "Lily";  
cat.age = 3;  
alert(cat.age); // 3

我们赋予了这个“猫对象”两个属性，一个是它的名字，一个是年龄。每个属性都对应一个值。当我们需要改变或得到某个属性的值时，我们只需要写下 <对象名>.<属性名> 即可。我们可以把它当做一个对象内部的变量。

如果访问了未设置值的属性，我们会得到值 undefined ，——因为“未定义”啊！

和基本的值一样，对象也可以用简便的字面量方式创建，这种形式被称为*对象字面量*。

它的特征是一对大括号，标志着这个对象字面量的起始与结束。在大括号中我们可以写下一个属性和一个值，二者用冒号分隔，值后面写一个逗号，表示这一项*成员*结束了。

const dog = {name: "Peter", age: 4};  
alert(dog.name); // "Peter"

现在我们拥有了一只小狗，并给它起了名字。我们可以利用合适的换行，让对象字面量看起来舒服一些：

const dog = {  
 name: "Peter",  
 age: 4,  
};  
alert(dog.name); // "Peter"

最后一项成员的逗号是可选的，我们可以自由决定是否添加。

我们可以重复定义对象中的一个属性，最后一次定义的值会覆盖掉先前的值。但是我们没必要这样做，如果要改变一个属性的值，随时可以通过访问它的方式来改变。

// 不要这样做  
const object = {  
 a: 1,  
 a: 2  
};  
alert(object.a); // 2  
  
// 我们可以像这样  
const object2 = {  
 a: 1  
};  
object2.a = 2;  
alert(object2.a); // 2

对象字面量中所包含的信息构成了这个对象本身，自身也是值，因此我们可以在对象字面量中将一个属性的值设置为另一个对象字面量。当一个对象成为属性的值时，我们可以通过属性访问这个对象，进一步访问它的属性只需要在它后面接着写 .<属性名>即可。这种访问属性的方式被称为*点号访问法。*

const person = {  
 name: "Venn",  
 age: 18,  
 foods: {  
 apple: 5,  
 pear: 3  
 }  
};  
alert(person.name); // Venn  
alert(person.foods.apple); // 5

当然，我们不能直接查看一个对象，否则我们会得到一个看起来有些奇怪的字符串值：

alert(person); // "[object Object]"  
alert(person.foods); // "[object Object]"

**属性名**

如前文所说，对象的属性名可以看做是一个变量名，一个合法的标识符也可以直接用做属性名。但事实上，我们可以用**任何字符**作为属性名，包括空格等。我们只需要用双引号或单引号将属性名括起：

const person = {  
 "first name": "John",  
 "last name": "Doe",  
};

然后，可以这样访问属性值：

alert(person["first name"]); // John  
alert(person["last name"]); // Doe

这种方法被称为*方括号访问法*。如果属性名是合法的标识符，我们也可以写成完全等价的点号访问法的形式。

const o = {  
 "aaa": "bbb"  
};  
alert(o["aaa"]); // "bbb"  
alert(o.aaa); // "bbb"

练习 3.4.1

1. 编写一个程序，提示用户输入一些个人信息，并将它存储到一个对象里。
2. 在 3.4.1.1 的基础上，从对象中访问某个信息，并告诉用户某个它的值。

### 成员操作

JavaScript 提供了一些方式来操作对象中的属性和值。我们已经讨论了访问和修改成员值的方式，添加一个成员也很简单——就像修改它一样，直接赋值即可。

const cat = {  
 name: "Kitty",  
 age: 4,  
 kind: "unknown"  
};  
  
// 现在我知道它是波斯猫  
cat.kind = "Persian cat";  
alert(cat.kind); // "Persian cat"

我们也可以删除对象中的属性，JavaScript 提供了 delete 操作符。

const cat = {  
 name: "Kitty",  
 age: 4,  
 kind: "unknown"  
};  
  
// 现在我不需要关心它的品种  
delete cat.kind;  
alert(cat.kind); // undefined

这时候我们或许会有疑问：当一个属性被删除之后，再访问它就会得到 undefined。那么这跟直接把属性的值设为 undefined 有什么区别吗？

当然有！当属性值为 undefined 时，它只是一个值为 undefined 的属性，依然存在于对象中。但是，当我们用 delete 删除一个属性的时候，这个对象中就不存在这个属性了，只不过是访问了不存在的属性会得到 undefined 而已。

我们可以使用 in 操作符来查看对象中究竟是否存在某一个属性：

const cat = {  
 name: "Kitty",  
 age: 4,  
 kind: "unknown"  
};  
  
// 现在我不需要关心它的品种  
delete cat.kind;  
alert("kind" in cat); // false  
alert("name" in cat); // true

为什么当我们查看某个属性是否存在时，这个属性名要写成字符串？这是因为在 JavaScript 的语法中，如果直接写成 kind in cat 这样，kind 会被理解为一个标识符，然而我们并没有定义这个标识符，因此会得到一个错误。

事实上，JavaScript 中的对象的属性名都是以字符串方式存储的，只不过在对象字面量中，如果是合法的标识符，就不需要写成字符串的形式，JavaScript 会理解它。如果是点号标记法，由于它**只能用于访问写成合法标识符的属性名称**，因此属性名也不需要写成字符串的形式。但是在中括号标记法中，就应该写成字符串的形式。因为 JavaScript 会**将中括号中的内容当做表达式进行求值**，如果是一个字符串，那么就是按这个字符串对应的属性名进行访问，而如果认为是一个标识符，就会查找这个标识符本身的值所对应的属性名。

下面用一个示例来说明。

let a = "123";  
const object = {  
 a: 456,  
 "123": 123  
};  
alert(object.a); // 456  
alert(object[a]); // 123（标识符 a 的值是 "123"，那么访问了名为 "123" 的属性）  
alert(object["a"]); // 456（访问名为"a"的属性）

练习 3.4.2

1. 在 3.4.1.1 的基础上，提示用户输入一个名称，告诉用户这个对象中是否能找到名称对应的值。

* （提示：使用前面学习的逻辑相关知识）

### 枚举

上文所提供的访问对象成员的方式，都是通过属性名进行的。假如我们不知道某个属性的名称，或者说想一次性访问到所有的值，JavaScript 也提供了一些方便的函数来进行这类操作。

第一个是 Object.values 函数。我们可以从它得到一个列表，列表中会依次存放它所有的值。由于我们可以直接显示数组中的内容，因此它的所有值也就一目了然。

const cat = {  
 name: "Kitty",  
 age: 4,  
 kind: "Cheshire Cat"  
};  
  
const values = Object.values(cat);  
alert(values); // Kitty,4,Cheshire Cat

第二个是 Object.keys 函数，它得到一个对象中所有属性名称的列表。

const cat = {  
 name: "Kitty",  
 age: 4,  
 kind: "Cheshire Cat"  
};  
  
const keys = Object.keys(cat);  
alert(keys); // name,age,kind

此外，我们还可以使用 Object.entries 函数，它同样是得到一个列表，列表中交替保存属性名和值。

const cat = {  
 name: "Kitty",  
 age: 4,  
 kind: "Cheshire Cat"  
};  
  
const entries = Object.entries(cat);  
alert(entries); // name,Kitty,age,4,kind,Cheshire Cat

我们会在下一节中详细了解如何使用列表中的值。

### 全局对象

我们在这里需要明确一个之前没有提及的概念：

在 JavaScript 中，所有变量和常量都是一个*全局对象*的属性。

这个全局对象叫 window 。它在语义上指浏览器的窗口对象，同时也是 JavaScript 运行环境的一个“兜底”对象。当我们访问 window 对象的属性的时候，我们实际上是在访问这样一个变量/常量。

let hello = "Hello world";  
alert(window.hello); // "Hello world"  
alert(window.hello === hello); // true

由于 window 自身也是一个变量，因此它也是它自己的属性。

alert(window === window.window); // true  
alert(window.window === window.window.window); // true

一般情况下我们不需要直接使用 window 对象，不过我们可以使用它来判断一个标识符是否存在于当前环境中——换句话说，只要知道 window 对象是否有这个 A 属性，就可以是否声明过一个叫 A 的标识符。

练习 3.4.3

1. 编写一个程序，判断当前环境中是否存在一个变量或常量。
2. 编写一个程序，在 3.4.3.1 的基础上判断某个标识符是否为变量或常量。

* （提示：window 对象的属性值可改变）

## 数组

### 初探数组

我们已经学习了 JavaScript 中的对象的概念。假如我们想表示教室里一排学生的座次，我们或许可以用对象字面量这样写：

let seating = {  
 "1": "Mason Mills",  
 "2": "Rachel Griffiths",  
 "3": "Melissa Duncan",  
 "4": "Marley Hughes",  
 "5": "Charlie Henderson",  
 "6": "Brandon Sharp",  
 "7": "Morgan Woods",  
 "8": "Maddox Andrews",  
 "9": "Lexie Jefferson",  
 "10": "Estelle Bolton"   
};  
  
alert(seating["5"]); // "Charlie Henderson"  
alert(seating["9"]); // "Estelle Bolton"

然后，我们可以使用中括号访问法访问某个具体的序号上的学生姓名。但是这样显然非常不方便：

* 每增加一个学生，就要手动安排一个序号；
* 如果我们要将一个学生移到别的位置，将会非常糟糕：我们要把从这里到那里所有的学生重新排一遍序号。
* 我们不能方便地获取到已经加入到这个列表中的学生数量。

所幸，JavaScript 为我们提供了另外一种结构可以避免上面说的问题，它就是**数组**。

我们只需在数组中写下学生姓名，而无需被序号和获取学生数量等问题所困扰，数组为我们搞定了这一切。你可以这样构造一个数组：

let seating = ["Mason Mills", "Rachel Griffiths", "Melissa Duncan", "Marley Hughes", "Charlie Henderson", "Brandon Sharp", "Morgan Woods", "Maddox Andrews", "Lexie Jefferson", "Estelle Bolton"]；

你猜对了！这种构造形式被称为*数组字面量*。一对中括号 [] 标志着数组字面量的开始与结束，中括号之间的每一项都是数组的一个*成员*，或者叫*元素*。这两种称呼是等价的，本书中会交替出现这两个名词。

我们依然可以通过中括号访问法访问数组元素，不同的是，这里不用写成字符串的形式，我们只需要使用数字来访问进行访问，这种访问方式称为*下标访问法*，每个元素的唯一序号称为*下标*。下标是从 0 开始的，换句话说，第一个元素下标为 0，第二个为 1 ，第三个为 2……以此类推，第 n 个元素下标为 n-1 。如果数组中找不到指定下标的元素，就会得到一个 undefined。

// 不要写成这样;  
let student = seating["3"];  
alert(student); // "Melissa Duncan"  
  
// 只需要这样：  
let student = seating[2];  
alert(student); // "Melissa Duncan"  
  
// 找不到指定下标的元素的例子：  
alert(seating[100]); // undefined

如果要移动某个元素的位置，你只需要在构造时将它写在你需要的位置即可。你还可以使用 length 属性来获取数组的长度（即所包含元素的数量）。由于第 n 个元素的下标是 n-1，因此数组中最后一个元素的下标是 length 值减一。

// const seating = ["Mason Mills", "Rachel Griffiths", "Melissa Duncan", "Marley Hughes", "Charlie Henderson", "Brandon Sharp", "Morgan Woods", "Maddox Andrews", "Lexie Jefferson", "Estelle Bolton"]；  
// Morgan Woods 上课会跟 Maddox Andrews 说话，把他挪到第二排吧！  
let seating = ["Mason Mills", "Morgan Woods", "Rachel Griffiths", "Melissa Duncan", "Marley Hughes", "Charlie Henderson", "Brandon Sharp", "Maddox Andrews", "Lexie Jefferson", "Estelle Bolton"]；  
  
alert(seating.length); // 10  
alert(seating[0]); // "Mason Mills"  
alert(seating[1]); // "Morgan Woods"  
alert(seating[seating.length - 1]); // "Estelle Bolton"  
alert(seating[seating.length]); // undefined （超出了数组边界）

就像我们更改一个对象的属性一样，我们也可以用类似的方式更改数组的元素值，只要知道它的下标即可：

alert(seating[0]); // Mason Mills 被点名了  
seating[0] = "Anchimolios"; // 这个位置坐了一个古希腊人！  
alert(seating[0]); // 现在报上名字的是 Anchimolios

可怜的 Mason Mills！他的位置被一个我们都不认识的古希腊人给占了。不过我们不需要为他感到难过，我们会在下文学到如何在不剥夺任何一个学生座位的情况下安排新的学生进入这里。

现在，我们知道这一列有十个学生，但是今天很巧，坐在第四、第五位的 "Melissa Duncan、Marley Hughes 都没来上课，我们可以给他们留两个空位：

let seating = ["Anchimolios", "Morgan Woods", "Rachel Griffiths", , , "Charlie Henderson", "Brandon Sharp", "Maddox Andrews", "Lexie Jefferson", "Estelle Bolton"]；

老师开始点人数了！坐在第一个的是 Anchimolios……咦，是谁坐在第四、第五个来着？老师没看到人，就想不起来他们的名字，只能用 undefined 来表示自己的无奈。

alert(seating[2]); // "Rachel Griffiths"  
alert(seating[3]); // undefined  
alert(seating[4]); // undefined

几分钟后，Melissa Duncan 和 Marley Hughes 气喘吁吁地赶回了他们的座位上，老师终于想起了他们的名字。

seating[3] = "Marley Hughes";  
seating[4] = "Melissa Duncan";

等等……好像有点不对！这两人的位置是不是坐反了？我们记得 Melissa Duncan 明明坐在 Marley Hughes 的前面！老师发现了这个问题，现在要求他们互换座位。于是 Marley Hughes 先腾出了座位，等 Melissa Duncan 从原本属于他的位置上起来，好坐回去。

let temp = seating[3];  
seating[3] = seating[4];  
seating[4] = temp;

Marley Hughes 站起来，站到了一个临时（temp）位置，于是 Melissa Duncan 一步到位，坐到了第四桌，Marley Hughes 终于也回到了他的位置上。但是，这两个小伙子不仅迟到，还搞出了这么些名堂，于是，**他们被老师记住了！**（默哀一秒钟）

现在终于开始正式上课了，同学们开始聚精会神地听讲，只见老师在黑板上写下了 “Array 构造器“几个大字，同学们似乎想起了什么……

note：还记得在上一节中提到的 Object.values、 Object.keys 和 Object.entries 吗？它们都会得到数组。如果不记得的话，可以回去看一看。

### Array 构造器

在 JavaScript 中，每种类型的值都具有一个*构造器*，数组也不例外。它是一类称为 array 的值。我们已经见到了 number 类型（数字）的构造器 Number，string 类型（字符串）的构造器 String，还有 object 类型（对象）的构造器 Object。那么数组——这类称为 array 类型的值，它的构造器自然叫 Array。虽然我们总是用字面量方式创建这些值，但是了解它们的构造器也是很有必要的。

我们可以创建一个构造器的**实例**，在前文中这种形式就叫做创建实例，但是现在我们要冠以一个更准确的术语：实例化。例如，当我们写下 const o = new Object(); 的时候，我们实例化了 Object 构造器，得到了一个对象。我们也可以如法炮制，实例化 Array 构造器：

let a = new Array();

它创建了一个*空数组*，与这种字面量形式是等价的：

let a = [];

如果我们知道数组应该有多少元素（例如：我们知道有多少学生坐在同一列），可以在实例化 Array 构造器时附加一个数据，以表示创建一个包含这么多元素的数组：

let a = new Array(10);  
alert(a.length); // 10  
  
// 相当于  
let a = [ , , , , , , , , , ]; // 嘿，怎么搞的，今天一个人都没回来？

趁着老师发出怒吼之前，我们赶紧往下看！

我们也可以把需要包含的元素写在 Array 构造器的括号里，括号内的东西有个正式名称，叫*实参*。

let a = new Array("Anchimolios", "Morgan Woods", "Rachel Griffiths", "Charlie Henderson");  
alert(a.length); // 4  
  
// 相当于  
let a = ["Anchimolios", "Morgan Woods", "Rachel Griffiths", "Charlie Henderson" ]; // 好吧，至少回来了四个

这时有同学提问：

假如我们要用这种方式创建一个只包含一个数字成员的数组，不就会与创建“包含这个数量的数组”混淆了吗？

这是 JavaScript 设计时一个没考虑周到的地方，我们在使用时应当加以注意。当然，我们完全可以直接使用方便的数组字面量来创建数组，也可以使用另一种方式——利用字符串的 split 方法，将一段字符串按照特定分隔符，分解成数组。当我们需要构造比较长的字符串数组时，可以使用这种方式，它比完整的数组字面量更为简便。split 方法更多情况下适用于直接根据用户输入建立列表。

const source = "Dark night gave me black eyes, but I use it to look for light";  
const array = source.split(" "); // 根据空格进行拆分  
alert(array); // "Dark,night,gave,me,black,eyes,,but,I,use,it,to,look,for,light"  
const source2 = "黑夜给了我黑色的眼睛，我却用它来寻找光明";  
const array2 = source2.split(""); // 直接将每个字符都作为数组元素  
alert(array2); // "黑,夜,给,了,我,黑,色,的,眼,睛,，,我,却,用,它,来,寻,找,光,明"  
  
// 数组的 join 方法可以将数组元素以特定文本相连，组合成字符串  
alert(array2.join("。")); // "黑。夜。给。了。我。黑。色。的。眼。睛。，。我。却。用。它。来。寻。找。光。明"

### 插入与移除

我们的数组不会是一成不变的。如果我们有一列学生，我们有可能会在前面、后面或中间安插新学生的座位，如果有学生要离开这一列，我们也需要将他的座位移走。JavaScript 中提供了一些方便的操作，来在数组中实施以上现实中的需求。

JavaScript 提供了三个*数组方法*。方法是一些函数，它以属性的形式被访问，并按规定的方式使用。“方法”一词反映了它们的本质：通过一些固定方式来完成指定的需求。

push 方法用于在数组末尾加入一个元素，并得到数组的新长度。

let list = ["red", "orange", "yellow", "green"];  
alert(list.length); // 4  
list.push("blue"); // 加入 "blue" 这个字符串  
alert(list.length); // 5  
alert(list); // red,orange,yellow,green,blue

我们可以直接查看数组内部所包含的元素，如运行结果所显示的那样，list 数组末尾添加了 "blue" 这个字符串。我们也可以一次多添加几个元素：

alert(list.push("purple", "pink")); // 添加两个元素并显示数组的新长度 7  
alert(list.length); // 7  
alert(list); // red,orange,yellow,green,blue,purple,pink

unshift 方法将一个或多个元素添加到数组的开头，并得到数组的新长度。

let flowers = ["Calendula", "Chrysanthemum", "Cosmos", "Dianthus"];  
alert(flowers.length); // 4  
alert(flowers.unshift("Dahlia")); // 把大丽花加进来，现在长度是 5  
alert(flowers.length); // 是的！就是5！

和 push 类似，它也可以一次添加多个元素。

此外还有一个 concat 方法，它用于将两个数组拼在一起，并得到合并后的新数组。

// 现在有两支来自日本的观光团，他们要进行合并以统一行程。  
let visitors\_1 = ["Jissoji Goro", "Kagawa Nui", "Okamura Nariakira",  
 "Ehara Kiyumi", "Okita Miu", "Tanji Chigusa"];  
let visitors\_2 = ["Abo Akasuki", "Kirishima Kayoko", "Kuwahara Toichi",  
 "Amachi Hisato, Nakayama Norihisa"];  
alert(visitors\_1.length); // 现在第一支观光团有 6 人。  
alert(visitors\_2.length); // 第二支观光团有 4 人。  
let visitors = visitors\_1.concat(visitors\_2); // 合并！  
alert(visitors.length); // 现在新的观光团有十个人。  
alert(visitors); // Jissoji Goro,Kagawa Nui,  
 // Okamura Nariakira,Ehara Kiyumi,  
 // Okita Miu,Tanji Chigusa,...

提供了加入元素的方法，自然也提供了移除元素的途径。

pop 方法与 push 相对，用于移除数组的最后一个元素，并得到这个移除的元素值。

let list = [1, 2, 3, 4, 5];  
alert(list.pop()); // 5  
alert(list); // 1,2,3,4

shift 方法与 unshift 方法相对，用于移除数组的第一个元素，并得到移除的元素值。

let list = [1, 2, 3, 4, 5];  
alert(list.shift()); // 1  
alert(list); // 2,3,4,5

如果操作的数组是什么元素也没有的空数组，那么执行方法时只能得到一个 undefined 值。

let empty = [];  
alert(empty.pop()); // undefined

我们可以利用刚刚了解到的几个数组方法来模拟一个排队的场景。

let queue = []; // 现在还没人开始排队。  
queue.push("Delphium Vitulus");  
queue.push("Palinurus Catullus");  
queue.push("Amatia Laevinus"); // 到现在为止，队伍里有三个人。  
alert("现在队伍情况：" + queue);  
queue.push(prompt("请输入新来者的名字：")); // 新来者的名字由你决定  
alert(queue.shift() + "离开了，大家向前一位。");  
alert("又离开了一位：" + queue.shift());  
queue.push("Talmudia Zosimus");  
queue.push(queue.shift())); // 队伍第一位又重新开始排队  
alert(queue.pop() + "等不住了，走了");  
alert("现在队伍里共有" + queue.length + “人。"");

上述数组方法都是在起始或结束位置添加元素，约束性太强。我们经常需要从数组的任意位置删除元素。由于数组也是一类对象，也可以使用 delete 操作符，那么它会产生怎样的效果呢？

const language = ["C++", "Java", "Python", "Ruby", "Swift"];  
delete language[2];  
// 咦，长度怎么没有改变？  
alert(language.length); // 5  
alert(language); // "C++,Java,undefined,Ruby,Swift"

显然，这种删除数组元素的方法是无效的，它只是清除了某一位置上元素的值，但依然保留了元素所占的位置。数组仍然有五个元素，我们还要删除 undefined。

类似地，如果要在数组任意位置插入元素，应该怎么做呢？JavaScript 的所有数组都拥有 splice 方法。给出一个索引，splice 方法可以完成删除和插入元素的操作。

const language = ["C++", "Java", "Python", "Ruby", "Swift"];  
let removedItem = language.splice(2, 1); // 从第二个索引，也就是第三个元素开始，删除一个元素  
 // splice 方法返回被删除的元素组成的数组  
alert(removedItem); // "Python"  
alert(removedItem.length); // 1  
// language 中不再有 "Python"，后面的元素自动向前移动  
alert(language.length); // 现在只剩四个元素了  
  
removedItem = language.splice(1, 2, "Haskell", "Erlang", "Perl");  
 // 在 splice 方法中添加参数，可以实现在指定位置插入新元素  
alert(removedItem.length); // 3  
alert(language.length); // 5  
alert(language); // "C++,Haskell,Erlang,Perl,Swift"

在这个例子中，splice 至少需要两个参数，分别是起始索引和需要移除的元素个数。如果没有指定元素个数也可以工作，splice 会删除从起始索引开始直到末尾的所有元素。然后，splice 将所有删除掉的元素组合成一个新数组并返回。如果 splice 接收到了三个或更多参数，那么移除元素后，会将剩余的参数都作为数组元素，从起始索引开始插入数组。这里，splice 删除了 "Java" 和 "Ruby" 两个元素，然后从 "Java" 原来的位置开始插入 "Haskell"，接下来是 "Erlang" 和 "Perl"。插入完毕后，再调整被移除元素后面原来元素的位置。

### 排序与查找

在计算机科学中，对数据进行排序是一个常见需求。生活中也常有这样的例子，比如教科书的扉页上按照笔画顺序排列的编者姓名、按照分数排列的成绩单等。JavaScript 的数组作为存储数据的载体，提供了一个叫 sort 的方法用于数组排序。它会将数组中的元素按照字典顺序排列，如果遇到的元素不是字符串，就会将它转换为字符串来看待。

sort 方法直接对数组进行操作，同时返回修改后的数组，与下文中的 reverse 方法一致。

let array = ["Since", "then", "no", "torch", "such", "as", "fire", "I", "will", "be", "the", "only", "light"]; // 鲁迅先生名言：此后如竟没有炬火，我便是那唯一的光  
alert(array.sort()); // "I,Since,as,be,fire,light,no,only,such,the,then,torch,will"  
  
// 大写字母总会被排列在小写字母前面，数字也是按照字典顺序排列  
array = [1, 11, 21, 2, 0, "Apple", "apollo", "Beta"];   
alert(array.sort()); // "0,1,11,2,21,Apple,Beta,apollo"

这样的排序规则比较简单，我们将会在第五章接触到更为高级和灵活的排序技巧。

Note：

数组的 sort 方法实际上是根据每项元素转换为字符串后的 Unicode 码点进行排列的。·

另一种可能遇到的排序需求是，将数组中所有元素的顺序反过来，也就是*数组倒序*（reverse），这时不会关心数组的具体内容，一把梭，敢教日月换新天！

let array = ["Since", "then", "no", "torch", "such", "as", "fire", "I", "will", "be", "the", "only", "light"];  
alert(array.reverse()); // "light,only,the,be,will,I,fire,as,such,torch,no,then,Since"

reverse 方法可以配合 sort 方法来使用：sort 将数组元素按照字典顺序排列，reverse 再进行倒序，也就是按照倒字典序（Z-A）进行排列。如果需要将字符串中每个字符的顺序翻转，也可以结合字符串的 split 方法。

let array = ["Since", "then", "no", "torch", "such", "as", "fire", "I", "will", "be", "the", "only", "light"];  
alert(array.sort().reverse()); // "will,torch,then,the,such,only,no,light,fire,be,as,Since,I"  
let array2 = "鸿雁长飞光不度，鱼龙潜跃水成文".split(""); // 将每个字拆分成数组元素  
alert(array2.reverse().join("")); // "文成水跃潜龙鱼，度不光飞长雁鸿"

计算机科学界的高峰、高德纳先生的著作《计算机程序设计艺术》中有一整卷书专门讲排序与查找两大数据处理方式，可见查找操作在数据处理领域占据着与排序相当的重要地位。我们在使用文本处理软件时，查找功能为我们带来了莫大方便。JavaScript 中的数组提供了 indexOf 和 lastIndexOf 两个方法，用于查找指定元素。

const weather = ["sunshine", "rain", "snow", "dew", "rainbow", "wind"];  
alert(weather.indexOf("rain")); // 1  
alert(weather.lastIndexOf("rain")); // 5

indexOf 方法接受要查找的元素作为参数，在数组中找到第一个匹配的元素，并返回索引。如果找不到，那么返回 -1。

weather.indexOf("snow"); // 2  
weather.indexOf("snowman"); // -1

而查找某个元素最后一次出现时的索引，可以用 lastIndexOf方法。

weather.lastIndexOf("snow"); // 4  
weather.lastIndexOf("snowman"); // -1

此外，还有一个简单的 includes 方法，用于确定一个元素是否在数组中存在，返回 true 或 false。

weather.includes("snow"); // true  
weather.includes("snowman"); // false

数组的查找方法可以应用在实际生活中。例如下面这个小程序模拟了文本处理软件的查找功能。

// 查找小程序  
const source = prompt("请输入一段英文文本，用空格分隔").split(" ");  
const target = prompt("请输入要查找的词：");  
let index = source.indexOf(target);  
let lastIndex = source.lastIndexOf(target);  
alert(`在给定文本中搜索“${target}”，结果如下：  
 第一次出现的位置：${index}  
 最后一次出现的位置：${lastIndex}`);

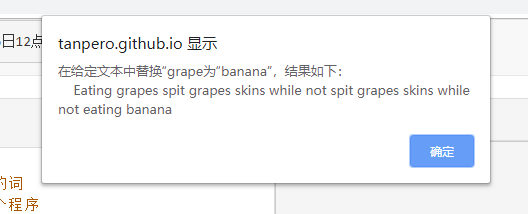
假设第一次输入“ Eating grapes spit grapes skins while not spit grapes skins while not eating grapes”，第二次输入“grapes”，那么会得到这样的搜索结果。



我们刚才已经了解了用于删除和替换元素的 splice 方法，那么也可以考虑将它应用到这里来，升级成替换功能：

// 替换小程序  
// 目前只能替换第一个（和最后一个）匹配的词  
// 在后面我们将学到更高级的内容来完善这个程序  
const source = prompt("请输入一段英文文本，用空格分隔").split(" ");  
const target = prompt("要替换的词：");  
const destination = prompt("替换成：");  
let index = source.indexOf(target);  
source.splice(index, 1, destination);  
alert(`在给定文本中替换“${target}为“${destination}”，结果如下：  
 ${source.join(" ")}`);

这次我们试试把“grape”替换成“banana”（有点奇怪），得到了这样的结果：



咦？我们不是要替换掉第一个出现的“grape”吗？怎么会这样？

关键的地方在于这里：

let index = source.indexOf(target);  
source.splice(index, 1, destination);

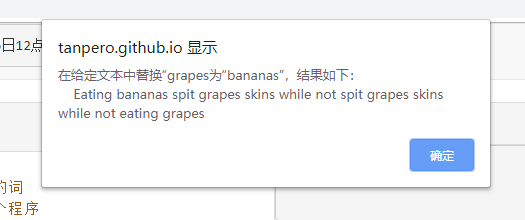
indexOf 方法只有在找到与 target 的内容完全一致的元素时才会返回索引，否则得到 -1。因为 -1不是一个合法的数组索引，它表示“未找到”。这里，我们用“grape”进行查找，但是数组中只有单独的"grapes"，而没有"grape"，因此index 会等于 -1。

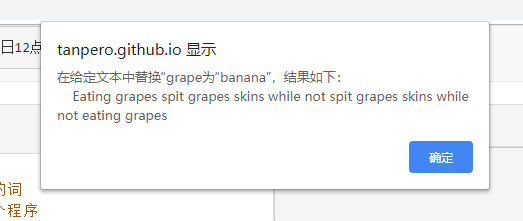
但是 splice 并不知道这一点。根据 splice 的规则，如果第一个表示索引的参数是一个负数（），那么 splice 会理解为移除倒数第 个元素，并在其位置上插入新元素（如果指定了的话）。因此，这里 splice 得到了 index，由于它是 -1，所以将 source 的倒数第一个匹配元素替换为 destination（也就是 "banana"），最终呈现为我们看到的结果。

为了解决这个问题，我们可以用 includes 方法判断一下 source 中是否包含指定的 target。当然，includes 方法需要重新查看整个数组，可能会造成效率问题，因此可以直接用 index 做简单判断，再决定是否进行替换。

// 替换小程序  
// 目前只能替换第一个（和最后一个）匹配的词  
// 在后面我们将学到更高级的内容来完善这个程序  
const source = prompt("请输入一段英文文本，用空格分隔").split(" ");  
const target = prompt("要替换的词：");  
const destination = prompt("替换成：");  
let index = source.indexOf(target);  
index !== -1 ? source.splice(index, 1, destination) : null;  
alert(`在给定文本中替换“${target}”为“${destination}”，结果如下：  
 ${source.join(" ")}`);

第八行代码使用了一个三目表达式来做选择，表达式的值会被丢弃，其作用体现在 source 上。现在再来测试一波代码，可以看到我们实现了预期结果。





## 数据类型

### JavaScript 中的数据类型

JavaScript 所有的值（数据）都有着自己的**数据类型**。目前 JavaScript 中共有以下的数据类型：

* number（数值）
* bitInt （任意精度整数）
* string（字符串）
* object（对象）
* function（函数）
* undefined
* regexp（正则表达式）
* symbol（符号）

我们已经接触了 number 用于数值计算，string 用于处理文本内容，object 作为存放其他数据的结构，还有 undefined，它自身也是一种独立的数据类型。我们可能会为看不见 array 而感到奇怪，事实上，数组是一种特殊的对象，它被精心设计以用于处理数据，但是我们仍需将它归类为 object。至于另外三种数据类型，我们会在后面逐渐接触到。

JavaScript 提供了 typeof 操作符来确定一个值的数据类型，它得到类型的字符串名称，即以上七种之一。

alert(typeof 123); // "number"  
alert(typeof NaN); // "number  
alert(typeof Infinity); // "number"  
alert(typeof "Hello world"); // "string"  
alert(typeof {}); // "object"  
alert(typeof [1, 2, 3]); // "object"  
alert(typeof null); // "object"  
alert(typeof alert); // "function"  
alert(typeof undefined); // "undefined"

Note: 为什么 typeof null === "object" ？

在 JavaScript 第一个版本的实现中，每个值在内部存储时都会用一个标记来记录它的数据类型。由于 object 是 JavaScript 中的“第一等类型”，它的类型标记为 0，而 null 表示为 NULL *指针*，在大多数平台上， NULL 指针的实际值是 0x00，那么 null 的数据类型标记实际上与 object 相同。因此 typeof 运算符在获取 null 的类型标记的时候，得到的是 0 ，便会将它判断为 "object"。

### 判断数据类型的技巧

typeof 的首要作用自然是判断数据类型，以便了解到的数据类型做出可能的操作。它具有明显的局限性：它对数据类型的判断仅限于以上七种；无法区分 array 和一般的 object；当我们明确地需要一个**对象**时它却会将 null 混为一谈。我们需要在 typeof 的判断的基础上使用一些辅助方法，以便在各类数据类型之间游刃有余。

**区分数组与对象**

typeof 会将数组判定为 "object"，没关系，JavaScript 提供了 Array.isArray 函数来判断一个值是不是数组。那么我们只需要：

alert(Array.isArray([1, 2, 3])); // true  
alert(Array.isArray([])); // true  
alert(Array.isArray(new Array())); // true  
alert(Array.isArray({})); // false

**精确地判断对象**

我们在第一节（逻辑）中了解过，null 会被当做一个假值，因此对它进行非运算会得到 true。

我们可以采取这样的策略：

* 使用typeof 进行判断，如果它不是 "object"，得到 false。
* 如果它是 "object"，判断它是否为数组，若是，得到 false。
* 如果它是 null，得到 false，否则为 true。可以对它进行 !!操作，若为 null ，我们就会得到 false，否则我们就会得到 true。

示例：

let value = "Hello world";  
alert( // 对 "Hello" 进行判断  
 typeof value !== "object" ? false :  
 Array.isArray(value) ? false :  
 !!value  
); // false  
// value = true -> false  
// value = 123 -> false  
// value = null -> false  
// value = [1, 2, 3] -> false  
// value = {a: 1, b: 2} -> true  
// value = window -> true

**精确判断数字**

NaN 和 ±Infinity 都属于 number 类型，但是我们在进行数学运算的时候并不希望将它们参与到运算中。我们只需要进行一些附加判断即可将它们与真正的数字区分开来。

let n = 100;  
alert(  
 typeof n !== "number" ? false :  
 isNaN(n) ? false : isFinite(n)  
); // true  
// n = "123" -> false  
// n = NaN -> false  
// n = Infinity -> false  
// n = Number.MAX\_VALUE -> true  
// n = Number.MAX\_VALUE \* 10 -> false

## 解构赋值

JavaScript 提供了一类语法以简化赋值操作，可以根据一组数据，一次定义一组变量，像这样：

let [a, b] = [1, 2];  
alert(a); // 1  
alert(b); // 2

等号右边是一个通常的对象/数组字面量，而等号左边就写成与右边相匹配的格式，即一个*模式*，以满足我们的需要。

它实际上是如下方式的替代：

let arr = [1, 2];  
let a = arr[0];  
let b = arr[1];

这类语法被称为*解构赋值*，按字面意义上来理解，就是“解析一个结构，并进行赋值”。

### 数组解构

我们可以将一组变量在声明时同时进行赋值操作。

let arr = ["one", "two", "three"];  
let [one, two, three] = arr;  
alert(one); // "one"  
alert(two); // "two"  
alert(three); // "three"

也可以将声明与赋值分离。

let a, b;  
[a, b] = [1, 2];  
alert(a); // 1  
alert(b); // 2

如果一个模式没有找到相匹配的值，相应的变量就会得到 undefined，因为不知道到该给它赋什么值，只能说明它是“未定义”的。

let [a, b, c] = [1, 2];  
alert(a); // 1  
alert(b); // 2  
alert(c); // undefined

为了避免发生这种情况，可以在模式中为变量设置一个默认值。如果找不到匹配值，就使用变量的默认值。

let [a = 3, b = 4, c = 5] = [1, 2];  
alert(a); // 1  
alert(b); // 2  
alert(c); // 5

在一个解构赋值的表达式中我们还可以交换两个已有变量的值，以避免使用*临时变量*。

let a = 3, b = 5;  
[a, b] = [b, a];  
alert(a); // 5  
alert(b); // 3  
  
// 不使用解构赋值  
let temp = a;  
a = b;  
b = temp;  
alert(a); // 5  
alert(b); // 3

我们也可以忽略我们不感兴趣的值：

let [a, , b] = [1, 2, 3];  
alert(a); // 1  
alert(b); // 3  
  
// 甚至忽略所有值，但是没必要这样  
[, ,] = [1, 2, 3];

当解构一个数组时，可以使用*剩余模式*，将数组的“剩余部分”交给一个变量。

let [a, ...b] = [1, 2, 3];  
alert(a); // 1  
alert(b); // [2, 3]

注意：如果剩余模式右侧有逗号，会得到一个语法错误，因为剩余元素必须是数组的最后一个元素。

let [a, ...b,] = [1, 2, 3]; // SyntaxError: rest element may not have a trailing comma

同时，使用剩余模式赋值的变量不能被设置默认值。

let [a, ...b = 2] = [3, 4] // SyntaxError: Invalid destructuring assignment target  
alert("a = " + a); // 不会执行  
alert("b = " + b); // 不会执行

用于实现剩余模式的三个点号称为*Spread 操作符*。

### 对象解构

除了数组以外，我们还可以对一个对象进行解构赋值。在数组解构中，变量被赋予的值取决于它的排列顺序，而在对象解构中，变量根据对象中对应的属性名被赋予值。

let o = {p: 42, q: true};  
let {p, q} = o;  
alert(p); // 42  
alert(q); // true

通过解构，无需声明即可赋值一个变量。

let a, b;  
({a, b} = {a: 1, b: 2});  
alert(a); // 1  
alert(b); // 2

赋值语句周围的 ( ... ) 是使用对象字面解构赋值时不需要声明的语法。{a, b} = {a: 1, b: 2} 不是有效的独立语法，因为左边的 {a, b} 被认为是一个**块语句**而不是对象字面量。然而，({a, b} = {a: 1, b: 2}) 是有效的，相当于let {a, b} = {a: 1, b: 2}

Note：

我们的( ... ) 表达式需要一个分号在它前面，否则它也许会被当成上一行中的函数来执行。

我们也可以根据属性，将值赋给我们需要的变量名，而不一定与属性名重名。

let obj = {p: 42, q: true};  
let {p: a, q: b} = obj;   
// 相当于  
// a = obj.p;  
// b = obj.q   
alert(a); // 42   
alert(b); // true

变量也可以先赋予默认值。当要提取的对象没有对应的属性，变量的值就是它的默认值。

let {a = 10, b = 5} = {a: 3};  
alert(a); // 3  
alert(b); // 5

可以将上述方法结合起来。

let {a: first = 10, b: second = 5} = {a: 3};  
alert(first); // 3  
alert(second); // 5

模式中可以像普通对象字面量那样使用计算属性名，来确定要从对象中匹配的属性值。

let key = "name";  
let { [key]: theName } = { name: "Pluto" };  
alert(theName); // "Pluto"

事实上，我们还可以像数组解构那样，对对象解构应用剩余模式：

let {a, b, ...others} = {a: 10, b: 20, c: 30, d: 40}  
alert(a); // 10   
alert(b); // 20  
// others = { c: 30, d: 40 }  
alert(others.c); // 30  
alert(others.d); // 40

显然，如果一个属性不能作为合法的标识符名称，就需要用一个等价的方式来代替。

const obj = { "hello-world": "Oh!" };  
const { "hello-world": helloWorld } = obj;  
  
alert(helloWorld) // "Oh!"

# 第四章

语句

## if 语句

我们已经在第三章中了解了 JavaScript 中数据的使用，掌握到的预备知识是我们具备了随心所欲操控数据的能力，在这一章中我们将学习 JavaScript 中的*语句*，以编写出“真正的”程序，并逐步将我们头脑中的逻辑，转化为程序真切的运行过程。

在前面的运行示例中，我们接触到的都是顺序结构，即程序从第一条语句，逐行执行到结尾，而在这一章中我们将了解到*分支结构*和*循环结构*。

我们首先要学习的是 if *语句*。它用于实现分支结构。

想象一下，当我们要去买芹菜的时候，我们会执行这样的过程：

1. 出门，找到菜市场
2. 挑选芹菜
3. 付钱，将菜带回家。

这就是我们所说的顺序结构。但实际上，我们买菜可能会经历这样的过程：

1. 出门，找到菜市场
2. 寻找卖芹菜的摊位

* 如果有卖芹菜的
  1. 买一斤芹菜
  2. 称重，付款，回家

如果没有卖芹菜的

1. 看看有没有其他适合的蔬菜
2. 称重，付款，回家

以上过程的结构称为*分支结构*，我们可以用 if 语句来在程序中实现分支结构，它的*语法*是这样的：

if (条件) {  
 执行语句  
}

一条 if 语句以关键字 if 开头，其后跟随一对括号，括号中是一个逻辑表达式，用于表示执行语句的条件。

例如：

let a = 4;  
if (a > 3) {  
 alert("a 大于 3");  
}

这段代码中的 if 语句的条件是 a > 3，所执行的语句是 alert("a 大于 3"); 。显然，a 大于 3，因此条件成立（逻辑表达式得到 true），那么执行大括号中的语句——显示出 "a 大于 3" 这行内容。

if 语句只管理它大括号中的内容，其他部分不受条件影响。

倘若我们要考虑两种情况（如示例中的“有卖芹菜的摊位”与“没有卖芹菜的摊位”），那么我们可以使用 if 语句的另一种形式。

if (条件) {  
 语句1  
} else {  
 语句2   
}

这种形式的作用是：如果条件成立，那么执行代码1中的内容，否则，执行代码2。

else 关键字所*引导*的内容称为 *else 子句*，它用于说明“条件不成立”这一情况。

示例：

let a = 4;  
if (a > 5) {  
 alert("a 大于 5");  
} else {  
 alert("a 小于等于 5");  
}

在示例中，由于 a 的值为 4，if 语句的判断条件不成立（得到 false），所以不会显示 "a 大于 5"，而是执行 else 子句中的语句，显示 "a 小于等于 5"。如果 a 的值大于 5，符合判断条件的话，那么就会显示 "a 大于 5"。

我们似乎已经能够处理条件成立和不成立两种情况了。但现实生活比这复杂的多，我们可能会遇到许多其他情况，需要对每种情况一一加以判断并分别作出相应决定。在 JavaScript 中，我们可以使用 if 语句的高级形式，像这样：

let score = 85;  
if (score >= 90) {  
 alert("优秀");  
} else if (score >= 80) {  
 alert("良好");  
} else if (score >= 60) {  
 alert("及格");  
} else {  
 alert("不及格");  
}

上述代码模拟了一种常见情景：将成绩分为不同层次，并作出对应通知。

它的执行过程会先从第一个条件开始，如果条件成立，那么执行其后的语句；如果条件不成立，那么尝试判断第二个条件，若成立则执行相应语句，否则判断第三个条件……直到最后一条 else 子句，即当所有可能条件都不成立时，执行其中的语句。在上述代码中，分数大于等于 80，我们便得到了“良好”。

我们在第三章已经阐述过，JavaScript 会将一些值视为“真”，另一些视为“假”，以此进行逻辑运算，而不仅仅是局限于布尔值。同样，在 if 语句的条件并不要求得到一个布尔值，只要得到的值被视作“真”，就会执行相应语句。

因此，假如我们有如下代码：

// 判断一个数字是奇数还是偶数。  
// 如果它除以 2 的余数为 0，即能被 2 整除，那么是偶数。  
// 否则为奇数。  
let number = 100;  
if (number % 2 === 0) {  
 alert("偶数");  
} else {  
 alert("奇数")  
}

当条件的值为 0 时，它会被当做假，那么我们可以用更方便的形式书写条件：

let number = 100;  
if (number % 2) { // 余数不为 0，会当做“真”  
 alert("奇数");  
} else { // 余数为 0，会当做“假”  
 alert("偶数")  
}

if 语句是我们接触到的第一个 JavaScript 控制语句。我们可以用它来描述真实世界中的各类选择。

一条简洁明快的 if 语句，将一切分成了真假两个世界，中间隔着逻辑这条天河，它向何处流动，取决于你的思考。逻辑的伟力在于泾渭分明，逻辑的生命在于它所带来的不容逾越的秩序。

练习 4.1.1

1. 写一个幸运转盘的程序，每次根据一个随机数字的范围来决定颁发什么奖品。
2. 写一个程序，让用户输入出生年份，判断用户的生肖属相。如果不是一个合理的年份，就显示一个错误。

我们可以用 if 语句来处理一些生活中常见的、需要进行繁琐的分类讨论的问题，例如个人所得税的计算。

假设我们的父母（成年人也可以假设为自己）是工薪阶层，每个月都可能需要根据月收入缴纳一笔个人所得税，数额会被划分为不同的层级，面临的税率和所要缴纳的相应税款也不一样。根据中国法律规定，个人所得税的起征点是 3500 元。

在实际工作中，对于某些个人所得收入采用税后收入的概念，比如支付税后多少多少金额。这时，需要将税后的收入按一定公式换算为应税所得，然后再按照一般方法计算应交的税款。否则，将导致税款的少征。这里，在换算为应税所得过程中需要适用的税率及速算扣除数不能按照含税级距的税率表来套用，必须使用不含税级距的税率表。这就是不含税级距税率表产生的原因。这里的不含税级距指的是“税后收入”级距。

截止到 2011 年，中国的个人所得税税率如下表所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 级数 | 含税级距 | 不含税级距 | 税率(%) | 速算扣除数 |
| 1 | 不超过1500元 | 不超过1455元的 | 3 | 0 |
| 2 | 超过1500元至4,500元的部分 | 超过1455元至4,155元的部分 | 10 | 105 |
| 3 | 超过4,500元至9,000元的部分 | 超过4,155元至7,755元的部分 | 20 | 555 |
| 4 | 超过9,000元至35,000元的部分 | 超过7,755元至27,255元的部分 | 25 | 1,005 |
| 5 | 超过35,000元至55,000元的部分 | 超过27,255元至41,255元的部分 | 30 | 2,755 |
| 6 | 超过55,000元至80,000元的部分 | 超过31,375元至45,375元的部分 | 35 | 5,505 |
| 7 | 超过80,000元的部分 | 超过57,505的部分 | 45 | 13,505 |

备注：

* 本表含税级距指以每月收入额减除费用 **3500 元**后的余额或者减除附加减除费用后的余额。
* 含税级距适用于由纳税人负担税款的工资、薪金所得。
* 不含税级距适用于由他人（单位）代付税款的工资、薪金所得。

我们可以根据税率的级数进行分类讨论，有点像中学数学课上学过的分段函数。

let income = parseFloat(prompt("请输入原始收入"));  
  
let basic = 3500; // 个人所得税起征点  
let gap = income - basic; // 税前收入与起征点之差  
let tax = 0; // 应付税额  
if (gap <= 0) {  
 tax = 0;  
} else if (gap > 0 && gap <= 1500) {  
 tax = gap \* 0.03;   
} else if (gap > 1500 && gap <= 4500) {  
 tax = gap \* 0.1 - 105;  
} else if (gap > 4500 && gap <= 9000) {  
 tax = gap \* 0.2 - 555;   
} else if (gap > 9000 && gap <= 35000) {  
 tax = gap \* 0.25 - 1005;  
} else if (gap > 35000 && gap <= 55000) {  
 tax = gap \* 0.3 - 2775;  
} else if (gap > 55000 && gap <= 80000) {  
 tax = gap \* 0.35 - 5505;  
} else {  
 tax = gap \* 0.45 - 13505;  
}  
  
alert(Math.floor(tax));

这个例子具有相当的实用性，要不要考虑将自己或家人的收入通过这个程序计算一下，看看是否与实际情况吻合呢？

## switch 语句

### 初识 switch 语句

在前文中，我们已经学习并尝试了**分支结构**。我们将在几个示例中进一步了解它的使用。

设想一下，我们有一个有奖猜数活动，参与者可以随机输入一个 1 和 3 之间的整数，我们告诉参与者是否猜中，或与答案相差多少。利用 if 语句，我们可以这样写：

let number = +prompt("请输入 1 和 3 之间的整数");  
  
if (number === 1) {  
 alert("太小了！");  
} else if (number === 2) {  
 alert("刚刚好！");  
} else if (number === 3) {  
 alert("太大了！");  
} else {  
 alert("哦，数字超出范围！");  
}

我们共需依次比较三种情况，根据每种情况作出相应回应。而不在我们考虑范围内的“其他情况”则放入 else 子句中处理。在需考虑的情况较少时，使用 if 语句以此判断尚可应付需求，但当我们需要考虑到情况变得多，使用 if 语句就会显得力不从心。此外，如果我们要用相同的方式分别处理不同的情况，使用 if 语句会相当麻烦。

为此，JavaScript 中提供了另一种实现分支结构的语句：switch *语句*。

switch 语句的一般格式如下：

switch (表达式) {  
 case 情况1: 处理语句1; break;  
 case 情况2: 处理语句2; break;  
 case 情况3: 处理语句3; break;  
 //...  
 default: 默认处理语句;  
}

switch 语句遵循这样的执行顺序：

* 每个 case 关键字都用于标明一个情况。
* JavaScript 依次查看每种情况，然后对表达式求值，查看表达式的值是否能够与这种情况匹配。
* 如果匹配（严格相等），就执行冒号后的处理语句。如果以 break; 结尾，那么终止 switch 语句。
* 否则，继续查看下一个情况，以此类推。
* 当所有列举的情况都查看完之后，如果有一个 default 标志，就执行它的处理语句，作为默认情况。

前面的例子可以使用 switch 语句改写如下：

let number = +prompt("请输入 1 和 3 之间的整数");  
  
switch (number) {  
 case 1:  
 alert("太小了！");  
 break;  
 case 2:  
 alert("刚刚好！");  
 break;  
 case 3:  
 alert("太大了！");  
 break;  
 default: alert("哦，数字超出范围！");  
}

练习 4.2.1

1. 找到一个可以使用 switch 进行处理的生活中的例子，并编写程序实现它。

### 使用 break

让我们回忆一下 if 语句的机制：

对每个条件进行查看，如果成立，就执行相应处理语句，然后结束 if 语句。

当我们使用 switch 语句的时候，我们并不一定希望在找到匹配情况后，仍然继续匹配其余情况。但是，switch 语句有个特点：**当它匹配到一种情况之后，会继续执行之后其他情况的处理语句，**甚至包括 default 。

为了模仿 if 语句“干完事就走人”，不拖泥带水，我们需要在每一个 case 的处理语句后添加一行 break; 。

let fruit = prompt("请输入一种水果的名字：");  
  
switch (fruit) {  
 case "橙子":  
 alert("橙子卖 0.59 美元。");  
 break;  
 case "苹果":  
 alert("苹果卖 0.32 美元。");  
 break;  
 case "香蕉":  
 alert("香蕉卖 0.48 美元。");  
 break;  
 case "车厘子":  
 alert("车厘子卖 3 美元");  
 break;  
 case "芒果":  
 case "桑葚":  
 alert("芒果和桑葚卖 2.79 美元。");  
 break;  
 default:  
 alert("抱歉，本店没有水果" + fruit + "。");  
}

switch 语句每当遇到匹配的情况，执行相应处理语句后，就会终止。

如果 break; 后面还有处理语句，那么它不会被执行，因为 break; 已经起到了终止 switch 语句的作用。

我们使用 switch 语句时一般都会添加 break; ，这是一种良好习惯。没有添加 break; 以至于所有 case 都会被查看一遍的 switch 语句被称为 *switch 穿越*，可能会引发一些问题，我们将在下文中看到 switch 穿越所展现出的效果。。

### 关联操作

这个例子阐述了利用 switch 语句进行的关联操作。如前文所述，当我们输入一个数字并匹配之后，它会执行其后，一直第一个到 break; 之前的所有处理语句。

let value = 1;  
let output = "输出： "  
switch (value) {  
 case 10:  
 output += "所以";  
 case 1:  
 output += "你的";  
 output += "名字";  
 case 2:  
 output += "是";  
 case 3:  
 output += "什么";  
 case 4:  
 output += "？";  
 alert(output)  
 break;  
 case 5:  
 output += "！";  
 alert(output);  
 break;  
 default:  
 alert("请选择一个 1 到 6 之间的数字！");  
}

尝试改变 value 的值，相信你会对 switch 的机制有更深的体会。如果你不是特意为了制造出这种效果，请记得：务必在每种情况的相应处理语句末尾添加 break;。

练习 4.2.3

1. 使用 switch 语句的穿越特性，写一个程序，模拟一个会根据指令来问好的机器人。

### 分组

生活中有什么事情会像在 1 ~ 3 之间猜一个数字这样简单而乏味呢——让我们将目光投向更“实际”的问题。

现在我们摇身一变成了动物保护专家，向好奇的小朋友普及动物保护的知识，告诉他们哪些动物已经灭绝而湮没在历史中，哪些动物在悬崖边上苦苦挣扎，哪些动物暂时毫无危险。

let animal = prompt("请输入一种动物的名字：");  
switch (animal) {  
 case "猫":  
 case "金鱼":  
 case "鸵鸟":  
 case "企鹅":  
 case "火鸡":  
 case "马":  
 alert(animal + "没有危险！");  
 break;  
 case "大象":  
 case "熊猫":  
 case "江豚":  
 alert(animal + "处于危险之中，我们要一起保护它们。");  
 break;  
 case "渡渡鸟":  
 case "恐龙":  
 case "象鸟":  
 alert(animal + "已经灭绝。");  
 break;  
 default:  
 alert("我没听过这种动物的名字。");  
}

我们对小朋友提出的问题进行了分类，只用一行处理代码，统一回应相同性质的提问。

练习 4.2.4

1. 思考生活中有哪些实际问题可以使用 switch 进行分组处理。

### 使用动态条件

我们用一个小示例来结束本节。

let i = Math.floor(Math.random() \* 7)   
switch (i) {   
 case ((i >= 0 && i <= 5) ? i : -1): // 如果 0 ≤ i ≤ 5，那么待匹配的值为 i ，否则为 -1。   
 alert("0 ~ 5");   
 break;   
 case 6:   
 alert("6");   
 break;   
}

如上所示，你可以在 switch 语句的 case 中进行一些运算，以此呈现出“动态”的匹配效果。

## while 和 do-while 语句

### 循环结构

我们将开始接触 JavaScript 中的**循环结构**。顾名思义，循环结构能够在一定条件下循环执行同一段代码，这个过程称之为*迭代*。我们可以使用 while *语句*来实现循环结构。

while 语句看起来像这样：

while (条件) {  
 执行语句  
}

像 if 语句的条件一样， while 语句的条件是一个表达式，称为*条件*，写在一对小括号中。大括号连同其中的执行语句被称为*循环体*。如果执行语句只有一行，也可以省略包裹它的大括号。

* 对条件进行求值。
* 如果得到的值被看做 true（条件成立），那么执行一次循环。否则不执行。
* 循环体执行完毕后，再次对条件进行求值，如果成立则再次循环执行，否则停止。

用一个简单的示例来演示 while 语句的用法：

let n = 0;  
let x = 0;  
  
while (n < 3) {  
 n += 1;  
 x += n;  
}  
  
alert(n); // 3  
alert(x); // 6

在每次循环中，n 都会自增 1，然后再把 n 加到 x 上。因此，在每轮循环结束后，x 和 n 的值分别是：

* 第一轮后：n = 1，x = 1
* 第二轮后：n = 2，x = 3
* 第三轮后：n = 3，x = 6

当完成第三轮循环后，条件 n< 3 的值不再为真，因此循环终止。

现在我们要考虑用 while 语句进行一些实际应用。我们首先回忆一下高斯小时候那个著名的故事——你一定耳熟能详，对吧？

高斯上小学时，有一天老师出了一道算术难题：计算 1＋2＋3＋……＋100 。这下可难倒了刚学数学的小朋友们，他们按照题目的要求，正把数字一个一个地相加．可这时，却传来了高斯的声音：“老师，我已经算好了！”老师很吃惊，高斯解释道：因为1＋100＝101，2＋99＝101，3＋98＝101，……，49＋52＝101，50＋51＝101，而像这样的等于101的组合一共有50组，所以答案很快就可以求出：101×50＝5050

哦，我们不是高斯，所以我们可以使用 while 语句来进行循环计算，这样就解决了其他小朋友们的痛点。我们要干的事情就像任何一个不懂计算技巧的普通的小朋友那样：

1. 使用一个变量 n，存放初始值 0 。现在什么也没有，计算还没开始。
2. 使用另一个变量 i，用来计算每次应该被加上的值。i 每增加 1，n 就加上 i，直到 i 等于 100。
3. 现在 n 就包含了我们累加的值。

累加的过程使用 while 语句来处理，写成这样：

let n = 0;  
let i = 0;  
while (i < 100) {  
 i += 1;  
 n += i;  
}  
  
alert(n); // 5050

上述代码演示了 while 语句的基本应用。使用循环结构，可以解决我们手工计算的一些常见痛点，避免带来冗余。

实际上，你也可以将这样简单有效的运算推广到更大的范围，例如从 1 加到 10000，或 i 增加的值换成其他。当然，计算量越大，等待计算完成的时间也就越长，如果数字大小超过了 Number.MAX\_VALUE，就会溢出。（还记得第三章中的相关知识吗？:-D）

一个更常见的需求是*阶乘*。在数学上，n 的阶乘是指 1 × 2 × 3 × ... × n 这样的计算过程，其符号是 n! 。

我们已经有了从 1 累加到 100 的经验，而实现阶乘，看起来只是把加法运算改为乘法运算。

let n = 1;  
let i = 1;  
while (i < 5) {  
 i += 1;  
 n \*= i;  
}  
  
alert(n); // 120

这段代码实现了 5 的阶乘。i 作为*计数器*，依然是每次加上 1 ，而作为结果的 n 每次循环中乘以 i ，并作为新值。我们可以将它的步骤展开，以更清楚地观察运行过程：

1. n = 1, i = 1
2. i = 2, n = n \* i = 1 \* 2 = 2
3. i = 3, n = n \* i = 2 \* 3 = 6
4. i = 4, n = n \* i = 6 \* 4 = 24
5. i = 5, n = n \* i = 24 \* 5 = 120

我们可以从用户那里得到一个数字，并求出它的阶乘值。

let n = 1;  
let i = 1;  
let value = parseInt(prompt("请输入一个整数："));  
while (i < value) {  
 i += 1;  
 n \*= i;  
}  
  
if (!isFinite(n)) {  
 alert("数字太大了！");  
} else {  
 alert(value + "的阶乘是：" + n);  
}

我们可以输入一些数字来进行测试。如果我们输入的数字的阶乘值太大，超出了 JavaScript 的表示范围（得到 Infinity），那么我们会得到一个贴心的提示。或者，得到这个阶乘值（或其约数）。看起来一切正常。

但是！当我们输入负数呢？如果我们输入的内容无法解析为整数以至于得到 NaN 呢？我们会得出错误的结果。

value = 0; // 1，正确  
value = -1; // 1，错误  
value = -2; // 1，错误  
value = "Hello"; // 1，错误

关于“负数是否具有阶乘”等数学概念不在这里讨论范围内，我们应当设立明确的界限，对得到的值进行检查。如果它不符合要求，就通知用户，并不进行后续计算。

let n = 1;  
let i = 1;  
let value = parseInt(prompt("请输入一个正整数："));  
  
if (isNaN(value) || value < 0) {  
 alert("无法进行计算！")  
} else {  
   
 while (i < value) {  
 i += 1;  
 n \*= i;  
 }  
  
 if (!isFinite(n)) {  
 alert("数字太大了！");  
 } else {  
 alert(value + "的阶乘是：" + n);  
 }  
}

这样，我们可以保证：只有当得到一个正确的值的时候，才会进行计算。

我们还可以开动脑筋，将这个程序赋予更多创意：

let n = 1;  
let i = 1;  
let value = parseInt(prompt("请输入一个正整数："));  
  
while (!isNaN(value) && value >= 0) {   
 while (i < value) {  
 i += 1;  
 n \*= i;  
 }  
  
 if (!isFinite(n)) {  
 alert("数字太大了！");  
 } else {  
 alert(value + "的阶乘是：" + n);  
 }  
 value = parseInt(prompt("请输入一个正整数："));  
}

这个程序将我们已经学习的诸多概念融合在了一起，如果你一时没看明白这个程序究竟在做什么，可以多花点时间仔细看一看。这里解释一下这个程序所干的事情：

1. 得到一个用户输入的值（value）。
2. 如果这个值符合我们的要求，就开始执行后续过程。
3. 进行常规阶乘计算。
4. 通知用户关于阶乘计算的情况。
5. 再次要求用户输入一个值。
6. 回到步骤 2。

这个程序演示了 while 语句作为控制结构的一般应用。它控制了整个程序的运行流程，这类流程被称为*控制流*。我们将在本章后续了解到如何进一步完善控制流。在这个程序中，当你输入一个不符合要求的值时，控制流便会终止。当我们学习*异常处理*的概念之后，我们可以使用更加优雅的方式来处理异常和终止的情况。

### do-while 语句

while 语句有一种变体，称为 do-while *语句*。它的形式如下：

do {  
 执行语句  
} while (条件)

与通常的 while 语句不同的是，它的循环体写在了 do 关键字后面，而循环条件则放在了循环体的后面。

在第一次查看条件之前，do-while 语句无论如何都会先执行循环体，然后再查看条件，判断是否进行下一次循环。除此以外与通常的 while 语句是完全等价的。

let n = 1;  
let i = 1;  
do {  
 i += 1;  
 n \*= i;  
} while (n < 5)  
  
alert(n); // 120

do-while 语句适用于需要先执行一遍，再进行条件判断的情况。

## for 语句

### 基本概念

while 和 do-while 语句可以用于实现循环结构，除此以外，JavaScript 中提供了另一种更加灵活快捷的方式来进行循环（或者叫迭代）：for *语句*。

一个 for 语句看起来像这样：

for (初始化表达式; 条件; 增量表达式) {  
 执行语句  
}

等等，for 语句的小括号中包含了三个东西！它们都是什么？

首先是*初始化表达式*。它用于说明哪些值会被用在循环中。例如，在上一节的阶乘示例中，我们在进行计算之前要先设置变量 i 和 n 的值为 1，这类操作就是*初始化*。你可以将初始化的操作直接放在 for 语句里，称为初始化表达式。

初始化表达式的分号后面是*条件*，它表示控制循环进行的条件，与 while 语句是一致的。

而*增量表达式*用于在每次循环后，改变控制循环的变量的值，以此达到控制循环次数的作用。

事不宜迟，用一个简单的示例来看一下 for 语句的使用：

for (let i = 0; i < 5; i += 1) {  
 alert(i);  
}  
// 0  
// 1  
// 2  
// 3  
// 4

1. 首先设置控制循环的变量 i 的值为 0。
2. 查看 i 的值是否满足条件。
3. 满足，那么执行循环体的语句，显示出 i 的值。
4. 循环体结束，根据增量表达式，改变 i 的值，为下一次循环做准备。
5. 回到步骤 2。

如果 i 的值一开始就不满足条件，那么循环体一次也不会被执行，像 while 语句一样。

上一节中的计算 1 + 2 + 3 + ... + 100 的示例，使用 for 语句可以改写如下：

let n = 0;  
for (let i = 1; i < 100; i += 1) {  
 n += i;  
}  
alert(n); // 5050

我们使用 i 作为控制循环的变量，每次循环后它的值便会 +1 ，同时 i 也起到了从 1 增长到 100 ，用于使 n 进行累加的作用。

事实上，为了充分利用 for 语句，我们还可以更进一步：

for (let i = 1, n = 0; i < 100; i += 1, n += i) {  
  
}  
alert(n); // 5050

我们可以在初始化表达式的位置上写几个用于进行初始化的表达式，只需使用逗号隔开。

同样，增量表达式也可以对不同的变量进行增量处理。它的求值顺序是从左到右的，也就是说，先计算了 i += 1 ，然后再处理 n += 1。

如果我们的循环体没有语句，我们根本就不用写大括号，直接省略就好了！。

for (let i = 1, n = 0; i < 100; i += 1, n += i)  
  
alert(n);

但是运行这段代码，就会发现，alert 不会在循环结束后执行，而是每进行一次循环都会显示一次当前 n 的值。因此，你一共要点一百次“确认”，直到循环结束。

现在，刷新页面，一切恢复正常。

为什么会这样？因为 for 语句和 while 语句一样，如果没有写大括号，会将循环头部（即一对小括号包裹的内容）的后面遇到的第一个语句当做是循环体，因此 alert(n) 被循环执行了。解决这个问题的办法是在循环头部后面写一个分号 ; ，用一个*空语句*来代替循环体。

for (let i = 1, n = 0; i < 100; i += 1, n += i) ;  
// ; （更推荐写在第二行）  
alert(n); // 5050

练习 4.4.1

1. 创建一个 1 ~ 100 的循环，当数字 n 是奇数时，打印 “ n是奇数”，否则打印“n是偶数”。

* 提示：使用运行器提供的 document.write 函数来进行“打印”操作。

1. 显示一个九九乘法表，使用 document.writeln 函数来打印每一行。

* 提示：你需要将一个 for 语句写在另一个内部，使用它们的控制变量来输出因数。

### 迭代算法

我们在这里第一次接触*算法*一词。Wikipedia 对此的定义如下：

在数学和计算机科学中，**算法**是一个明确的、关于如何解决一类问题的规范。算法可以执行计算，数据处理、自动推理和其他任务。算法可以在有限的空间和时间内表达。从初始状态和初始输入开始，描述了一系列计算，当执行时，通过有限个明确定义的连续状态，最终产生“输出” ，并终止于最终结束状态。

当我们要实现某种功能时，我们将会用精确的语言，描述我们所要实施的步骤。例如在上文中对于循环过程的描述就叫算法。迭代算法则是用迭代等方式实现的算法，通俗来讲，使用循环和条件控制来进行一系列运算，以得到我们需要的结果。上文中的累加和阶乘等运算就属于迭代算法。

我们将会在本节深入了解迭代算法，一个常见的实际应用就是检测一个数是否为质数。

如果你忘记了什么叫质数，我们可以先复习一下小学数学书上对于质数的描述：

如果一个大于 1 的整数只有 1 和它本身两个因数，那么它就是一个质数，否则就是合数。

2 是最小的质数，1 既不是质数也不是合数。

换句话说，一个整数如果大于 1，并且不能整除除 1 以外的所有比它小的整数，那么它就是一个质数。

我们很快就可以得到判断一个数 n 是否为质数的思路：

1. 如果这个数不大于 1 或不为整数，那么它必定不是质数。
2. 从 2 开始，列举从 2 到 n-1 的所有整数，用 n 除以列举的数。
3. 如果得到的余数为 0，说明 n 可以整除它，那么 n 不是质数。
4. 如果列举完后也没有找到一个数可以被 n 整除，那么 n 是质数。

我们可以根据这个思路尝试写出代码，列举数字的工作自然就交给 for 语句。

let n = 100;  
let isPrime = true; // 假设它是一个质数  
for (let i = 2; i < n; i += 1) {  
 if (n % i === 0) {  
 isPrime = false;  
 }  
}  
alert(isPrime ? "质数" : "合数"); // "合数"

通过列举它可能的因数，尝试进行整除，这种策略称为*试除法*。

这个代码可以正常工作，但我们很快就发现了它的问题：当我们发现 n 不是一个质数的时候，应该终止计算并告知结果。但是这里，即使我们发现了 n 是合数，for 语句也不会停下来，又白白将剩下的循环运行完。

因此，我们应当采取策略：当我们知道它不是一个质数的时候，我们就不再进行试除了，而是报告结果。

let n = 100;  
let isPrime = true; // 假设它是一个质数  
`for` (let i = 2; i < n; i += 1) {  
 if (n % i === 0) {  
 isPrime = false;  
 break;  
 }  
}  
alert(isPrime ? "质数" : "合数"); // "合数"

这里再次出现了 break; 语句。它在这里的作用是**直接终止循环**。

我们知道，一个合数最大的因数不会超过它的平方根，例如 100 最大的因数就是 10 ，判断一个整数是否为质数，只需列举到它的平方根进行试除就足够了：

let n = 100;  
let isPrime = true; // 假设它是一个质数  
for (let i = 2, last = Math.sqrt(n); i <= last; i += 1) {  
 if (n % i === 0) {  
 isPrime = false;  
 break;  
 }  
}  
alert(isPrime ? "质数" : "合数"); // "合数"

由于除了 2 以外的所有质数都是奇数，因此我们可以进行一个简单的判断：

* 如果输入的数字是 2 ，那么它是一个质数。
* 如果输入的数字不是 2，但能被 2 整除，那么它是一个合数。
* 从 3 开始列举它的因数，每次 +2，使列举的值始终是奇数。

let n = 100;  
let isPrime = (n === 2) || (n % 2 !== 0);  
for (let i = 3, last = Math.sqrt(n); i <= last; i += 2) {  
 if (n % i === 0) {  
 isPrime = false;  
 break;  
 }  
}  
alert(isPrime ? "质数" : "合数"); // "合数"

我们的程序可以用于处理用户输入并得到结果了，不过务必记得进行输入检查。

let n = parseInt(prompt("请输入一个大于 1 的正整数。"));  
while (true) { // 循环接受输入。  
 if (isNaN(n) || !isFinite(n) || n <= 1) {  
 alert("输入不符合要求，程序停止");  
 break; // 如果输入不符合要求，就停止循环接受输入。  
 }  
   
 let isPrime = (n === 2) || (n % 2 !== 0);  
 for (let i = 3, last = Math.sqrt(n); i <= last; i += 2) {  
 if (n % i === 0) {  
 isPrime = false;  
 break; // 这个 break 语句只退出当前所在的循环。  
 }  
 }  
 alert(`${n}是一个${isPrime ? "质数" : "合数"}`); // 使用模板字符串来拼凑信息  
 n = parseInt(prompt("请输入一个大于 1 的正整数。"));  
}

我们的质数判断程序遵循的基本流程是“输入-处理-输出”。为了避免每次输出后都要重新运行才能开始新的流程，我们可以用一个循环将流程包进去。由于我们已经认识了 break 语句，因此可以自由决定 while 循环何时终止。循环头的条件用 true 来表示“条件始终成立”，表示它不再管条件判断，只需不断进行循环以重复相同的流程。这样的程序称为“Read-Eval-Print Loop”（输入-处理-输出循环），缩写为 **REPL**。

练习 4.4.2

1. 本节提供了一个完整的用于判断质数的 REPL 示例程序，请在此基础上对它进行修改：如果 n 是一个合数，那么同时显示发现的第一个因数。
2. 对本节的示例程序进行扩充，接受一个用户输入的整数 n，查找 2 ~ n 范围内的所有质数并显示。

* （提示：将找到的质数放在数组中）

### 数组遍历

假如我们有一列排列整齐的课桌，每张课桌上都写着使用它的学生的姓名，现在我们要依次浏览并记录每张课桌上的姓名，最直观的的办法显然是：从第一张课桌开始，记录课桌上的信息，然后走到下一个课桌，以此类推。

这个过程用精确的语言描述一下：

1. 走到第一张课桌的位置，记录信息
2. 走到下一张课桌的位置，如果这里确实还有课桌，就继续记录。
3. 如果没有课桌了，就停止这个过程。
4. 否则，回到步骤 2。

我们应该怎样用 JavaScript 来实现这个过程呢？相信答案已经呼之欲出了——循环！使用循环来解决这个问题。

let queue = ["Sonam", "Susanna Kliment", "Unnr Radmila", "Davide", "Rebekah "];   
   
for (let i = 0; i < queue.length; i += 1) {   
 alert(`第${i}个学生的姓名是：${queue[i]}`);   
}   
// Sonam   
// Susanna Kliment   
// Unnr Radmila   
// Davide   
// Rebekah

我们使用 for 语句依次访问了数组中的每一个元素，变量 i 表示数组元素的*索引*，它是一个约定俗成的名称。如果这个索引值小于数组长度，说明还没有到数组尽头，那么就继续进行处理，否则就停止循环。依次访问数组每一个元素的过程称为*遍历*。

## for-in 和 for-of 语句

JavaScript 提供了 for 语句的两种变体用于更加灵活地实现遍历，不但能遍历数组的索引和值，也能遍历对象的成员，并将得到的值赋给一个变量。

### 属性遍历

我们可以使用 for-in *语句*来遍历一个对象中的所有属性名称。for-in 语句的形式如下：

for (let 变量 in 对象) {  
 处理语句  
}

for-in 语句会依次访问对象中的每个属性的名称，并将得到的字符串存放在指定的变量中，这个过程会对每一个*可遍历*的属性都执行一次。

const person = {  
 name: "Jason",  
 age: 30,  
 sex: "male",  
 job: "teacher"  
};  
  
for (let i in person) {  
 alert(i);  
}  
// "name"  
// "age"  
// "sex"  
// "job"

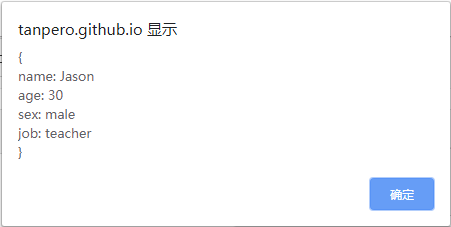
有了属性名，我们也就能同时得到它的值：

for (let i in person) {  
 alert(person[i]);  
}  
// "Jason"  
// 30  
// "male"  
// "teacher"

于是，我们可以简单地打印出一个对象的内容：

let s = "{\n";  
for (let i in person) {  
 s += `${i}: ${person[i]}\n`;  
}  
s += "}";  
alert(s);

结果如图所示：



for-in 语句提供了对对象内容进行操作的快捷方式。在这个遍历的过程中，我们可以干许多事情。比如——给每个属性都重新起一个名字，抛弃原来的：

for (let i in person) {  
 person["属性-" + i] = person[i];  
 delete person[i];  
}

这时再用之前的方式查看对象内容，就会看到每个属性的名字都被改变了。

{  
属性-name: Jason  
属性-age: 30  
属性-sex: male  
属性-job: teacher  
}

我们还可以更进一步：为每个属性都进行编号，毕竟， for-in 语句的本质还是循环，可以做一些适合循环做的事情。

let count = 1;  
for (let i in person) {  
 person[`第${count}个属性-${i}`] = person[i];  
 delete person[i];  
 count += 1;  
}  
// {  
// 第1个属性-name: Jason  
// 第2个属性-age: 30  
// 第3个属性-sex: male  
// 第4个属性-job: teacher  
// }

要知道 for-in 语句赋予了我们随意与属性和值打交道的权力——是的，我们甚至可以*交换*属性的名称与值的位置。**当然，如果原本的值就不是一个*基本类型*，我们还是不要这样做，否则会发生奇怪的事情。**

const object = {  
 name: "Andy",  
 checked: true,  
 anotherObj: {  
 a: 1,  
 b: 2  
 }  
};  
  
for (let i in object) {  
 let newName = object[i]; // 将原本的值存放起来  
 if (typeof newName !== "object") {   
 // 它不是一个对象，目测是基本类型  
   
 object[newName] = i; // 值的内容来命名一个新的属性，它的值就是原本的属性名  
 delete object[i]; // 原来的属性还在，但我们不需要它了  
 }   
}  
// 再用之前的方式查看一下对象里的情况  
// {  
// anotherObj: [object Object]  
// Andy: name  
// true: checked  
// }

又是 "[object Object]" ！恐怕你已经猜测到了我们所要避免的问题所在了。这个奇怪的东西我们将会在第七章详细讨论到，现在我们只需简单了解这一情况。

使用 for-in 语句，我们可以自由地查看、操作一个对象的内容。它是否使我们与对象更亲近了？

Note：我们只能遍历一个*可迭代对象*中的*可枚举属性*，我们将在下文了解这些概念。

### 可迭代对象

不单单是对象，我们也可以使用 for-in 语句来遍历数组，它提供了比前一节所介绍的更为简便的方法。在第三章中我们已经知道，**数组也是一种特殊的对象，它的索引都是属性，元素就是属性的值**，我们可以使用类似的方式来遍历它。

let array = ["aa", "bb", "cc", "dd", "ee", "ff"];  
for (let i in array) {  
 alert(`${i}: ${array[i]}`);  
}  
// 0: aa  
// 1: bb  
// 2: cc  
// 3: dd  
// 4: ee  
// 5: ff

数组与我们通常所写的对象的本质区别在于，它是*可迭代的*，也就是说每个成员的排列方式都遵循固定的顺序，我们可以通过固定的方式来依次访问每个成员。还有什么东西也是这样的呢？字符串！

let s = "Hello world";  
for (let i in s) {  
 alert(`第${i}个字符是 "${s[i]}"`);  
}  
// 第0个字符是 "H"  
// 第1个字符是 "e"  
// 第2个字符是 "l"  
// 第3个字符是 "l"  
// 第4个字符是 "o"  
// 第5个字符是 " "  
// 第6个字符是 "w"  
// 第7个字符是 "o"  
// 第8个字符是 "r"  
// 第9个字符是 "l"  
// 第10个字符是 "d"

字符串可以看做“字符的数组”，也就可以通过通常的方式遍历其中包含的每一个字符。

Note:

每一个可迭代对象都包含一个**迭代器**。迭代器涉及 JavaScript 中一些非常高级的概念，我们将在第七章中详细了解。

for 语句的另一种变体—— for-of *语句*更关注对值的操作，当我们只需要遍历一些值时，我们就可以使用它。

for-of 语句的形式与 for-in 语句类似：

for (let 变量 in 对象) {  
 执行操作  
}

用 for-of 语句来遍历数组中的每个值会格外方便。

const arr = ["aa", "bb", "cc", "dd", "ee", "ff"];  
for (let i of arr) {  
 alert(i);  
}  
  
// "aa"  
// "bb"  
// "cc"  
// "dd"  
// "ee"  
// "ff"

与 for-in 语句的显著不同之处在于，for-of 语句只能对**可遍历对象**进行遍历。如果你对一个普通对象使用 for-of 语句，会得到一个错误。

// person 对象就是先前的那个   
for (let i of person) {   
 alert(i);   
} // TypeError: person is not iterable

但是不必就此打住：还记得第三章中见到的 Object.keys Object.values Object.entries 三个函数吗？它们得到的是数组！换句话说，我们可以借助于它们来迭代普通对象！

for (let [name, value] of Object.entries(person)) {  
 alert(`${name}: ${value}`);  
};  
// name: Jason  
// age: 30  
// sex: male  
// job: teacher

或者使用 Object.values 作为跳板，直接对值进行遍历。

for (let value of Object.values(person)) {   
 alert(value);   
}   
// "Jason"   
// 30   
// "male"   
// "teacher"

假如我们有一个数组，里面的元素都是对象，利用前一章中了解到的解构赋值，我们可以根据对象的属性来进行处理。

const peoples = [   
 {   
 name: 'Mike Smith',   
 family: {   
 mother: 'Jane Smith',   
 father: 'Harry Smith',   
 sister: 'Samantha Smith'   
 },   
 age: 35   
 },   
 {   
 name: 'Tom Jones',   
 family: {   
 mother: 'Norah Jones',   
 father: 'Richard Jones',   
 brother: 'Howard Jones'   
 },   
 age: 25   
 }   
];   
   
for (let {name: n, family: {father: f}} of peoples) {   
 alert('Name: ' + n + ', Father: ' + f);   
}   
// "Name: Mike Smith, Father: Harry Smith"   
// "Name: Tom Jones, Father: Richard Jones"

## 异常处理

无论我们多么精通编程，有时我们的程序仍会不可避免的遭遇到一些错误，可能单纯是我们的程序编写出错，或是接收到了与我们预期不符的用户输入，或者是其它什么原因。通常，一段代码会在出错的时候停止执行，如果只是一个用于练习的小程序可能及时排查出问题倒没什么，但如果实在一个监测生命健康，或是多人网络游戏中，程序一旦因为遇到异常而停止执行，会造成难以预料的不良后果。JavaScript 提供了一种try...catch *语句*，它会在捕捉到异常的同时不会使代码停止执行，还能根据得到的异常信息做一些更为合理的操作。

### 语法

try...catch 结构由两部分组成：try 和 catch：

try {  
 // 代码...  
} catch (e) {  
 // 处理异常  
}

它按照以下步骤执行：

* 首先，执行 try 子句中包含的代码。
* 如果执行过程中没有异常，那么忽略 catch 子句里面的代码，try 子句执行完之后离开这个 try...catch 语句去做其他事情。
* 如果执行过程中发生异常，控制流就转移到了 catch 子句的开头。变量e是一个包含了异常信息的对象，也可以取其它名称，但是 e 可以看做 error（错误）或exception（异常）的缩写。

图片来源：https://mmbiz.qpic.cn/

所以，发生在 try子句的异常不会使代码停止执行：我们可以在 catch 子句里处理异常。

### try...catch 语句的使用

让我们来看更多的例子。

没有异常的例子：

try {  
 alert("开始运行 try 子句");  
 // ...这里没有异常  
 alert("try 子句运行完毕");  
} catch (err) {  
 alert("没有任何异常，catch 子句被忽略了");  
}  
alert("现在继续执行");

包含异常的例子：

try {  
 alert("开始执行 try 子句");  
 lalala; // 异常，变量未定义！  
 alert("try 子句执行完了"); // (2)  
} catch(err) {  
 alert("捕捉一只异常！");  
}  
alert("现在继续执行");

要使得 try...catch能工作，代码必须是可执行的，换句话说，它必须是有效的 JavaScript 代码。

如果代码包含语法错误，那么try...catch 不能正常工作，例如含有未闭合的大括号：

try {  
 {{{{{{{{{{{{  
} catch(e) {  
 alert("这不是合法的代码，这段 catch 子句也不会被执行");  
}

浏览器读取然后执行代码，发生在读取代码阶段的异常被称为*解析时错误*（parse-time），try...catch 也对它们无可奈何，因为这样的代码浏览器就读不懂，也就无法理解 try...catch 语句。try...catch 只能处理有效代码之中的异常。这类异常被称为*运行时错误*（runtime errors），有时候也称为 “exceptions”。

当一个异常发生之后，JavaScript 生成一个包含异常细节的对象。这个对象会作为一个参数传递给 catch：

try {  
 // ...  
} catch(e) {  
 // “异常对象”，可以用其他参数名代替  
 // ...  
}

对于所有内置的异常，catch 子句捕捉到的相应的异常的对象都有两个属性：

name ：异常名称，对于一个未定义的变量，名称是 “ReferenceError”

message ：关于异常的文字描述。

还有很多非标准的属性在绝大多数环境中可用。其中使用最广泛并且被广泛支持的是：

stack ：当前的调用栈。它是用于调试的，一个包含引发异常的嵌套调用序列的字符串。

例如：

try {  
 lalala; // 异常，变量未定义！  
} catch(err) {  
 alert(err.name); // ReferenceError  
 alert(err.message); // lalala 未定义  
 alert(err.stack); // 异常捕获过程的细节  
 alert(err); // ReferenceError: lalala 未定义  
}

### 异常处理的应用

让我们一起探究一下真实使用场景中 try...catch 的使用。

在前面的章节中，我们了解过质数判断算法，并编写了一个循环接收用户输入并给出判断结果的程序。我们对用户输入进行了检查，如果符合要求，那么往后执行；否则的话，会给出一个错误信息并终止程序。由于是在一个 while 语句的循环体内，直接使用 break 语句就能达到终止程序的目的。但是break语句的本意是“停止循环“，而非“中止程序”，如果不是在循环内运行，就不能使用 break 语句，此外，在 break 语句之前还需要告知用户遇到的问题。

遇到不合法输入的问题本质是“处理异常”，而非“结束程序”，因此我们的程序应该拥有一个处理异常的机制，同时将“遇到异常，暂停程序”和“告知用户遇到的异常”优雅地结合在一起。前一个需求我们已经有了 try...catch 语句，而对于后一个，另一种语句可以做到：throw *语句*。它可以*标记*一个异常信息，称为*抛出异常*。它的语法如下所示：

throw 表达式;

当遇到 throw 语句的时候，程序暂停执行后面的内容，带着表达式的值一层一层地退出控制流，直到遇到外层的 try 子句。如果没有 try 子句包裹可能会抛出异常的语句，那么异常信息就会被直接告知浏览器，整个程序也就真正停止运行了。我们用一个示例来观察一下 throw 语句与 try..catch 语句的搭配使用。

try {  
 alert("一二三四五，上山打老虎");  
 throw "老虎来了";  
 alert("老虎没打到，打到小松鼠");  
} catch (e) {  
 alert(e);  
}

这段代码在输出 "一二三四五，上山打老虎"之后，遇到了 throw 语句，就暂停执行后面的内容。程序带着 "老虎来了" 的信息逃离现场，遇到 try 子句，就相当于吃了一记定心丸，带着强大的武器去捕捉老虎，便开始执行 catch 语句，同时捕获了"老虎来了"的异常信息，并输出它。如果 throw 语句的处于其它语句内部，也会使程序执行到这里就带着异常信息撤退，倘若遇到了 try 语句，就说明这个异常被捕获了，异常信息作为异常对象被传递给 catch 子句的括号里绑定的变量。

有了 throw 语句和 try...catch 语句搭配使用，程序便有了强大的异常处理机制，即便遇到“未知的危险”也可临危不惧，异常已经被抓在了 catch 子句的手心里，正常执行程序时也不会因可能遇到的异常而手忙脚乱。利用异常处理机制，我们来改写一下前面的质数判断程序。

let n = parseInt(prompt("请输入一个大于 1 的正整数。"));  
while (true) { // 循环接受输入。  
 try {  
 if (isNaN(n) || !isFinite(n) || n <= 1) {  
 throw "输入不符合要求，程序停止";  
 }   
 let isPrime = (n === 2) || (n % 2 !== 0);  
 for (let i = 3, last = Math.sqrt(n); i <= last; i += 2) {  
 if (n % i === 0) {  
 isPrime = false;  
 break; // 这个 break 语句只退出当前所在的循环。  
 }  
 }  
 alert(`${n}是一个${isPrime ? "质数" : "合数"}`); // 使用模板字符串来拼凑信息  
 n = parseInt(prompt("请输入一个大于 1 的正整数。"));  
 } catch (e) {  
 alert(e);  
 }  
}

原先的版本中，程序一旦接受到了不符合要求的用户输入，就会退出循环，也就停止了程序；而这里用异常处理机制改写之后，即使遇到异常，程序只会跳过这一轮的正常处理，直接告知用户，程序依然保持运行，同时“抛出”——“捕获”异常的语义性也远比原先的“输出信息”“跳出循环”要清晰得多。充分利用异常处理机制，我们能够写出更加优雅和*健壮*的代码。对于不可预知的异常输入而仍然能保持正常运行，并将信息及时展现给用户，这种性质被称为程序的*鲁棒性*。

### 异常对象

技术上讲，我们可以使用任何东西来作为一个异常对象。甚至可以是基础类型，比如数字或者字符串。但是更好的方式是用对象，尤其是有 name 和 message 属性的对象（某种程度上和内置的异常有可比性）。

JavaScript 有很多内置的标准异常构造器，我们也可以用它们来构造标准的异常对象。

|  |  |
| --- | --- |
| JavaScript 标准异常构造器 | 描述 |
| Error | 默认的错误。 |
| EvalError | 调用 eval 函数时出现错误。 |
| InternalError | JavaScript 引擎遇到的内部错误，如：“递归嵌套太多”。 |
| RangeError | 数值变量或参数超出其有效范围。 |
| ReferenceError | 无效的引用、求值过程。 |
| SyntaxError | JavaScript 引擎在解析代码时遇到的语法错误。 |
| TypeError | 变量或参数不属于有效类型。 |
| URIError | 给 encodeURI 或 decodeURI 传递的参数无效。 |

使用异常构造器的方式如下：

let error = new Error(message);  
// 或者  
let error = new SyntaxError(message);  
let error = new ReferenceError(message);  
// ...

对于内置的异常对象（不是对于其他的对象，而是对于异常对象），name 属性刚好是构造器的名字。message 则来自于参数所提供的异常信息。例如：

let error = new Error("不知道发生了什么 (O\_o)??");  
alert(error.name); // "Error"  
alert(error.message); // "不知道发生了什么 (O\_o)??"

我们可以使用任何东西来作为一个异常对象。甚至可以是基础类型，比如数字或者字符串。但是更好的方式是用对象，尤其是有 name 和 message 属性的对象。而内置的异常构造器同时为我们设定好了异常所属的类型，因此尽量使用具体的异常构造器。如果异常不是特定的，那么可以直接用 Error 构造器。

异常构造器可以通过 new 运算符建立新的异常对象，包含下列属性：

* message —— 我们能阅读的异常提示信息。
* name —— 异常名称（异常对象的构造函数的名称）。
* stack —— 异常发生时的调用栈。