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。你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。

2. 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
```

2. **|| （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。

3. **！（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。

```
1 alert(10 && !5);           // false
2 alert(!5 && 10);           // false
3 alert(!5 || 6 && 7);       // 7
4 alert(5 || 6 && 7 || 8);   // 5
5 alert(5 && 6 || 7 && 8);   // 6
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：

```
1 alert((5 || 6) && (7 || 8));   // 8
2 alert(5 && (6 || 7) && 8);     // 8
3 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。

你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`

2. 计算如下代码的值：

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

3. 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"`

2. 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

你可以使用 `alert` 函数来显示它们对应的十进制形式。

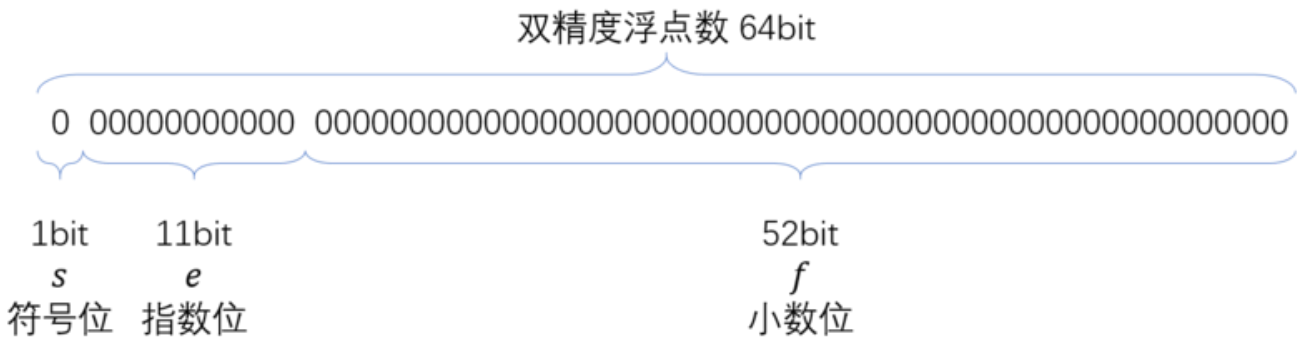
八进制和二进制的原理与十六进制类似，所不同的是：

- 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，你当然不用直接写出 ±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，你可以随意使用 $2^{-53}+1$ 到 $2^{53}-1$ 之间的所有整数，它们本身都是能够精确表示的。介于 $2^{53}+1$ 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。$2^{53}-1$ 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
```

你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。

你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，**你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。你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

// 浏览器认为你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 a^n（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 π，e 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，你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 2^{16} （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 2^{21}。

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 2^{16}-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{ed306}");  // "□"
```

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。

toUpperCase 和 toLowerCase

这两个函数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."
```

replace

这个函数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。

indexOf 和 lastIndexOf

`indexOf` 函数用于在一个字符串中从左往右搜索第一处匹配的位置，以 0 开始计数。如果没有找到，得到的值为 -1。可以接受第二个参数，用于查找指定出现次数的出现位置。

```
const s = "Hello world";
alert(s.indexOf("H"));       // 0
alert(s.indexOf("l"));       // 2
alert(s.indexOf("llo"));     // 2
alert(s.indexOf("l", 3));    // 3
alert(s.indexOf("666"));     // -1
```

`lastIndexOf` 函数与 `indexOf` 类似，但是搜索方向相反，从后往前查找。

```
const s = "Hello world!";
alert(s.lastIndexOf("llo"));     //2
alert(s.lastIndexOf("l"));       //9
alert(s.lastIndexOf("l", 4));    //3
```

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*，原则上你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。你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，你会得到值 `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。在大括号中你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，你可以自由决定是否添加。

你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

// 你可以像这样
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，因此你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，你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，你不能直接查看一个对象，否则你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，你可以用**任何字符**作为属性名，包括空格等。你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，你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"
```

你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，当你用 `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。

你可以使用 `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。

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（$-x$），那么 `splice` 会理解为移除倒数第 x 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。你可能会为看不见 `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

你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
```

你也可以忽略你不感兴趣的值：

```
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，你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:

你的`( ... )` 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!"
```