## 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。

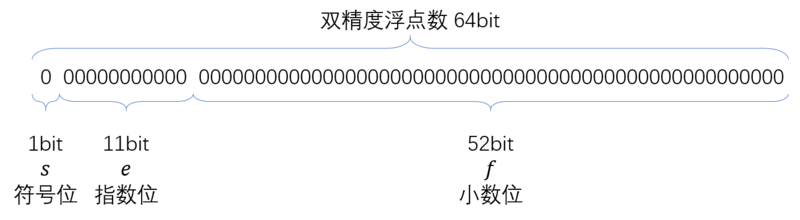
八进制和二进制的原理与十六进制类似，所不同的是：

* 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 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

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 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，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*。

### 位运算