数值

概述

整数和浮点数

JavaScript 内部，所有数字都是以64位浮点数形式储存，即使整数也是如此。所以，`1`与`1.0`是相同的，是同一个数。

```javascript
1 === 1.0 // true
```

这就是说，JavaScript 语言的底层根本没有整数，所有数字都是小数（64位浮点数）。容易造成混淆的是，某些运算只有整数才能完成，此时 JavaScript 会自动把64位浮点数，转成32位整数，然后再进行运算，参见《运算符》一章的“位运算”部分。

由于浮点数不是精确的值，所以涉及小数的比较和运算要特别小心。

```javascript
0.1 + 0.2 === 0.3
// false

0.3 / 0.1
// 2.9999999999999996

(0.3 - 0.2) === (0.2 - 0.1)
// false
```

数值精度

根据国际标准 IEEE 754，JavaScript 浮点数的64个二进制位，从最左边开始，是这样组成的。

- 第1位：符号位，`0`表示正数，`1`表示负数
- 第2位到第12位（共11位）：指数部分
- 第13位到第64位（共52位）：小数部分（即有效数字）

符号位决定了一个数的正负，指数部分决定了数值的大小，小数部分决定了数值的精度。

指数部分一共有11个二进制位，因此大小范围就是0到2047。IEEE 754 规定，如果指数部分的值在0到2047之间（不含两个端点），那么有效数字的第一位默认总是1，不保存在64位浮点数之中。也就是说，有效数字这时总是`1.xx...xx`的形式，其中`xx..xx`的部分保存在64位浮点数之中，最长可能为52位。因此，JavaScript 提供的有效数字最长为53个二进制位。

```
```

(-1)^符号位 * 1.xx...xx * 2^指数部分
```

上面公式是正常情况下（指数部分在0到2047之间），一个数在 JavaScript 内部实际的表示形式。

精度最多只能到53个二进制位，这意味着，绝对值小于2的53次方的整数，即-2<sup>53</sup>到2<sup>53</sup>，都可以精确表示。

```javascript
Math.pow(2, 53)
// 9007199254740992

Math.pow(2, 53) + 1
// 9007199254740992

Math.pow(2, 53) + 2
// 9007199254740994

Math.pow(2, 53) + 3
// 9007199254740996

Math.pow(2, 53) + 4
// 9007199254740996
```

上面代码中，大于2的53次方以后，整数运算的结果开始出现错误。所以，大于2的53次方的数值，都无法保持精度。由于2的53次方是一个16位的十进制数值，所以简单的法则就是，JavaScript 对15位的十进制数都可以精确处理。

```javascript
Math.pow(2, 53)
// 9007199254740992

// 多出的三个有效数字，将无法保存
9007199254740992111
// 9007199254740992000
```

上面示例表明，大于2的53次方以后，多出来的有效数字（最后三位的`111`）都会无法保存，变成0。

数值范围

根据标准，64位浮点数的指数部分的长度是11个二进制位，意味着指数部分的最大值是2047（2的11次方减1）。也就是说，64位浮点数的指数部分的值最大为2047，分出一半表示负数，则 JavaScript 能够表示的数值范围为2¹⁰²⁴到2⁻¹⁰²³（开区间），超出这个范围的数无法表示。
```

如果一个数大于等于2的1024次方，那么就会发生“正向溢出”，即 JavaScript 无法表示这么大的数，这时就会返回`Infinity`。

```javascript
Math.pow(2, 1024) // Infinity
```

如果一个数小于等于2的-1075次方（指数部分最小值-1023，再加上小数部分的52位），那么就会发生为“负向溢出”，即 JavaScript 无法表示这么小的数，这时会直接返回0。

```javascript
Math.pow(2, -1075) // 0
```

下面是一个实际的例子。

```javascript
var x = 0.5;

for(var i = 0; i < 25; i++) {
  x = x * x;
}

x // 0
```

上面代码中，对`0.5`连续做25次平方，由于最后结果太接近0，超出了可表示的范围，JavaScript 就直接将其转为0。

JavaScript 提供`Number`对象的`MAX_VALUE`和`MIN_VALUE`属性，返回可以表示的具体的最大值和最小值。

```javascript
Number.MAX_VALUE // 1.7976931348623157e+308
Number.MIN_VALUE // 5e-324
```

数值的表示法

JavaScript 的数值有多种表示方法，可以用字面形式直接表示，比如`35`（十进制）和`0xFF`（十六进制）。

数值也可以采用科学计数法表示，下面是几个科学计数法的例子。

```javascript
123e3 // 123000
123e-3 // 0.123
-3.1E+12
```

.1e-23
```

科学计数法允许字母`e`或`E`的后面，跟着一个整数，表示这个数值的指数部分。

以下两种情况，JavaScript 会自动将数值转为科学计数法表示，其他情况都采用字面形式直接表示。

**（1）小数点前的数字多于21位。**

```javascript
123456789012345678912
// 1.2345678901234568e+21

12345678901234567901
// 12345678901234568000
```

（2）小数点后的零多于5个。

```javascript
// 小数点后紧跟5个以上的零，
// 就自动转为科学计数法
0.0000003 // 3e-7

// 否则，就保持原来的字面形式
0.000003 // 0.000003
```

数值的进制

使用字面量（literal）直接表示一个数值时，JavaScript 对整数提供四种进制的表示方法：十进制、十六进制、八进制、二进制。

- 十进制：没有前导0的数值。
- 八进制：有前缀`0o`或`0O`的数值，或者有前导0、且只用到0-7的八个阿拉伯数字的数值。
- 十六进制：有前缀`0x`或`0X`的数值。
- 二进制：有前缀`0b`或`0B`的数值。

默认情况下，JavaScript 内部会自动将八进制、十六进制、二进制转为十进制。下面是一些例子。

```javascript
0xff // 255
0o377 // 255
0b11 // 3
```
```

如果八进制、十六进制、二进制的数值里面，出现不属于该进制的数字，就会报错。

```javascript
0xzz // 报错
0o88 // 报错
0b22 // 报错
```

上面代码中，十六进制出现了字母`z`、八进制出现数字`8`、二进制出现数字`2`，因此报错。

通常来说，有前导0的数值会被视为八进制，但是如果前导0后面有数字`8`和`9`，则该数值被视为十进制。

```javascript
0888 // 888
0777 // 511
```

前导0表示八进制，处理时很容易造成混乱。ES5 的严格模式和 ES6，已经废除了这种表示法，但是浏览器为了兼容以前的代码，目前还继续支持这种表示法。

特殊数值

JavaScript 提供了几个特殊的数值。

正零和负零

前面说过，JavaScript 的64位浮点数之中，有一个二进制位是符号位。这意味着，任何一个数都有一个对应的负值，就连`0`也不例外。

JavaScript 内部实际上存在2个`0`：一个是`+0`，一个是`-0`，区别就是64位浮点数表示法的符号位不同。它们是等价的。

```javascript
-0 === +0 // true
0 === -0 // true
0 === +0 // true
```

几乎所有场合，正零和负零都会被当作正常的`0`。

```javascript
+0 // 0
-0 // 0
(-0).toString() // '0'
(+0).toString() // '0'
```

唯一有区别的场合是，`+0`或`-0`当作分母，返回的值是不相等的。

```javascript
(1 / +0) === (1 / -0) // false
```

上面的代码之所以出现这样结果，是因为除以正零得到`+Infinity`，除以负零得到`-Infinity`，这两者是不相等的（关于`Infinity`详见下文）。

NaN

（1）含义

`NaN`是 JavaScript 的特殊值，表示“非数字”（Not a Number），主要出现在将字符串解析成数字出错的场合。

```javascript
5 - 'x' // NaN
```

上面代码运行时，会自动将字符串`x`转为数值，但是由于`x`不是数值，所以最后得到结果为`NaN`，表示它是“非数字”（`NaN`）。

另外，一些数学函数的运算结果会出现`NaN`。

```javascript
Math.acos(2) // NaN
Math.log(-1) // NaN
Math.sqrt(-1) // NaN
```

`0`除以`0`也会得到`NaN`。

```javascript
0 / 0 // NaN
```

需要注意的是，`NaN`不是独立的数据类型，而是一个特殊数值，它的数据类型依然属于`Number`，使用`typeof`运算符可以看得很清楚。

```javascript
typeof NaN // 'number'
```

（2）运算规则

`NaN`不等于任何值，包括它本身。

```javascript
NaN === NaN // false
```

数组的`indexOf`方法内部使用的是严格相等运算符，所以该方法对`NaN`不成立。

```javascript
[NaN].indexOf(NaN) // -1
```

`NaN`在布尔运算时被当作`false`。

```javascript
Boolean(NaN) // false
```

`NaN`与任何数（包括它自己）的运算，得到的都是`NaN`。

```javascript
NaN + 32 // NaN
NaN - 32 // NaN
NaN * 32 // NaN
NaN / 32 // NaN
```

Infinity

（1）含义

`Infinity`表示“无穷”，用来表示两种场景。一种是一个正的数值太大，或一个负的数值太小，无法表示；另一种是非0数值除以0，得到`Infinity`。

```javascript
// 场景一
Math.pow(2, 1024)
// Infinity

// 场景二
0 / 0 // NaN
1 / 0 // Infinity
```

上面代码中，第一个场景是一个表达式的计算结果太大，超出了能够表示的范围，因此返回`Infinity`。第二个场景是`0`除以`0`会得到`NaN`，而非0数值除以`0`，会返回`Infinity`。

`Infinity`有正负之分，`Infinity`表示正的无穷，`-Infinity`表示负的无穷。

```javascript
Infinity === -Infinity // false
```

```
1 / -0 // -Infinity
-1 / -0 // Infinity
```

上面代码中，非零正数除以`-0`，会得到`-Infinity`，负数除以`-0`，会得到`Infinity`。

由于数值正向溢出（overflow）、负向溢出（underflow）和被`0`除，JavaScript 都不报错，所以单纯的数学运算几乎没有可能抛出错误。

`Infinity`大于一切数值（除了`NaN`），`-Infinity`小于一切数值（除了`NaN`）。

```javascript
Infinity > 1000 // true
-Infinity < -1000 // true
```

`Infinity`与`NaN`比较，总是返回`false`。

```javascript
Infinity > NaN // false
-Infinity > NaN // false

Infinity < NaN // false
-Infinity < NaN // false
```

（2）运算规则

`Infinity`的四则运算，符合无穷的数学计算规则。

```javascript
5 * Infinity // Infinity
5 - Infinity // -Infinity
Infinity / 5 // Infinity
5 / Infinity // 0
```

0乘以`Infinity`，返回`NaN`；0除以`Infinity`，返回`0`；`Infinity`除以0，返回`Infinity`。

```javascript
0 * Infinity // NaN
0 / Infinity // 0
Infinity / 0 // Infinity
```

`Infinity`加上或乘以`Infinity`，返回的还是`Infinity`。

```javascript
Infinity + Infinity // Infinity
Infinity * Infinity // Infinity
```

```

`Infinity`减去或除以`Infinity`，得到`NaN`。

```javascript
Infinity - Infinity // NaN
Infinity / Infinity // NaN
```

`Infinity`与`null`计算时，`null`会转成0，等同于与`0`的计算。

```javascript
null * Infinity // NaN
null / Infinity // 0
Infinity / null // Infinity
```

`Infinity`与`undefined`计算，返回的都是`NaN`。

```javascript
undefined + Infinity // NaN
undefined - Infinity // NaN
undefined * Infinity // NaN
undefined / Infinity // NaN
Infinity / undefined // NaN
```

与数值相关的全局方法

parseInt()

（1）基本用法

`parseInt`方法用于将字符串转为整数。

```javascript
parseInt('123') // 123
```

如果字符串头部有空格，空格会被自动去除。

```javascript
parseInt('  81') // 81
```

如果`parseInt`的参数不是字符串，则会先转为字符串再转换。

```javascript
parseInt(1.23) // 1
// 等同于
```
```

```
parseInt('1.23') // 1
```

字符串转为整数的时候，是一个个字符依次转换，如果遇到不能转为数字的字符，就不再进行下去，返回已经转好的部分。

```javascript
parseInt('8a') // 8
parseInt('12**') // 12
parseInt('12.34') // 12
parseInt('15e2') // 15
parseInt('15px') // 15
```

上面代码中，`parseInt`的参数都是字符串，结果只返回字符串头部可以转为数字的部分。

如果字符串的第一个字符不能转化为数字（后面跟着数字的正负号除外），返回`NaN`。

```javascript
parseInt('abc') // NaN
parseInt('.3') // NaN
parseInt('') // NaN
parseInt('+') // NaN
parseInt('+1') // 1
```

所以，`parseInt`的返回值只有两种可能，要么是一个十进制整数，要么是`NaN`。

如果字符串以`0x`或`0X`开头，`parseInt`会将其按照十六进制数解析。

```javascript
parseInt('0x10') // 16
```

如果字符串以`0`开头，将其按照10进制解析。

```javascript
parseInt('011') // 11
```

对于那些会自动转为科学计数法的数字，`parseInt`会将科学计数法的表示方法视为字符串，因此导致一些奇怪的结果。

```javascript
parseInt(1000000000000000000000.5) // 1
// 等同于
parseInt('1e+21') // 1

parseInt(0.0000008) // 8
// 等同于
```

```
parseInt('8e-7') // 8
```

（2）进制转换

`parseInt`方法还可以接受第二个参数（2到36之间），表示被解析的值的进制，返回该值对应的十进制数。默认情况下，`parseInt`的第二个参数为10，即默认是十进制转十进制。

```javascript
parseInt('1000') // 1000
// 等同于
parseInt('1000', 10) // 1000
```

下面是转换指定进制的数的例子。

```javascript
parseInt('1000', 2) // 8
parseInt('1000', 6) // 216
parseInt('1000', 8) // 512
```

上面代码中，二进制、六进制、八进制的`1000`，分别等于十进制的8、216和512。这意味着，可以用`parseInt`方法进行进制的转换。

如果第二个参数不是数值，会被自动转为一个整数。这个整数只有在2到36之间，才能得到有意义的结果，超出这个范围，则返回`NaN`。如果第二个参数是`0`、`undefined`和`null`，则直接忽略。

```javascript
parseInt('10', 37) // NaN
parseInt('10', 1) // NaN
parseInt('10', 0) // 10
parseInt('10', null) // 10
parseInt('10', undefined) // 10
```

如果字符串包含对于指定进制无意义的字符，则从最高位开始，只返回可以转换的数值。如果最高位无法转换，则直接返回`NaN`。

```javascript
parseInt('1546', 2) // 1
parseInt('546', 2) // NaN
```

上面代码中，对于二进制来说，`1`是有意义的字符，`5`、`4`、`6`都是无意义的字符，所以第一行返回1，第二行返回`NaN`。

前面说过，如果`parseInt`的第一个参数不是字符串，会被先转为字符串。这会导致一些令人意外的结果。

```javascript
parseInt(0x11, 36) // 43
parseInt(0x11, 2) // 1

// 等同于
parseInt(String(0x11), 36)
parseInt(String(0x11), 2)

// 等同于
parseInt('17', 36)
parseInt('17', 2)
```

上面代码中，十六进制的`0x11`会被先转为十进制的17，再转为字符串。然后，再用36进制或二进制解读字符串`17`，最后返回结果`43`和`1`。

这种处理方式，对于八进制的前缀0，尤其需要注意。

```javascript
parseInt(011, 2) // NaN

// 等同于
parseInt(String(011), 2)

// 等同于
parseInt(String(9), 2)
```

上面代码中，第一行的`011`会被先转为字符串`9`，因为`9`不是二进制的有效字符，所以返回`NaN`。如果直接计算`parseInt('011', 2)`，`011`则是会被当作二进制处理，返回3。

JavaScript 不再允许将带有前缀0的数字视为八进制数，而是要求忽略这个`0`。但是，为了保证兼容性，大部分浏览器并没有部署这一条规定。

parseFloat()

`parseFloat`方法用于将一个字符串转为浮点数。

```javascript
parseFloat('3.14') // 3.14
```

如果字符串符合科学计数法，则会进行相应的转换。

```javascript
```

```
parseFloat('314e-2') // 3.14
parseFloat('0.0314E+2') // 3.14
```

如果字符串包含不能转为浮点数的字符，则不再进行往后转换，返回已经转好的部分。

```javascript
parseFloat('3.14more non-digit characters') // 3.14
```

`parseFloat`方法会自动过滤字符串前导的空格。

```javascript
parseFloat('\t\v\r12.34\n ') // 12.34
```

如果参数不是字符串，或者字符串的第一个字符不能转化为浮点数，则返回`NaN`。

```javascript
parseFloat([]) // NaN
parseFloat('FF2') // NaN
parseFloat('') // NaN
```

上面代码中，尤其值得注意，`parseFloat`会将空字符串转为`NaN`。

这些特点使得`parseFloat`的转换结果不同于`Number`函数。

```javascript
parseFloat(true)  // NaN
Number(true) // 1

parseFloat(null) // NaN
Number(null) // 0

parseFloat('') // NaN
Number('') // 0

parseFloat('123.45#') // 123.45
Number('123.45#') // NaN
```

isNaN()

`isNaN`方法可以用来判断一个值是否为`NaN`。

```javascript
isNaN(NaN) // true
isNaN(123) // false
```

但是，`isNaN`只对数值有效，如果传入其他值，会被先转成数值。比如，传入字符串的时候，字符串会被先转成`NaN`，所以最后返回`true`，这一点要特别引起注意。也就是说，`isNaN`为`true`的值，有可能不是`NaN`，而是一个字符串。

```javascript
isNaN('Hello') // true
// 相当于
isNaN(Number('Hello')) // true
```

出于同样的原因，对于对象和数组，`isNaN`也返回`true`。

```javascript
isNaN({}) // true
// 等同于
isNaN(Number({})) // true

isNaN(['xzy']) // true
// 等同于
isNaN(Number(['xzy'])) // true
```

但是，对于空数组和只有一个数值成员的数组，`isNaN`返回`false`。

```javascript
isNaN([]) // false
isNaN([123]) // false
isNaN(['123']) // false
```

上面代码之所以返回`false`，原因是这些数组能被`Number`函数转成数值，请参见《数据类型转换》一章。

因此，使用`isNaN`之前，最好判断一下数据类型。

```javascript
function myIsNaN(value) {
  return typeof value === 'number' && isNaN(value);
}
```

判断`NaN`更可靠的方法是，利用`NaN`为唯一不等于自身的值的这个特点，进行判断。

```javascript
function myIsNaN(value) {
  return value !== value;
}
```

isFinite()

`isFinite`方法返回一个布尔值，表示某个值是否为正常的数值。

```javascript
isFinite(Infinity) // false
isFinite(-Infinity) // false
isFinite(NaN) // false
isFinite(undefined) // false
isFinite(null) // true
isFinite(-1) // true
```

除了`Infinity`、`-Infinity`、`NaN`和`undefined`这几个值会返回`false`，`isFinite`对于其他的数值都会返回`true`。

参考链接

- Dr. Axel Rauschmayer, [How numbers are encoded in JavaScript](http://www.2ality.com/2012/04/number-encoding.html)
- Humphry, [JavaScript 中 Number 的一些表示上/下限](http://blog.segmentfault.com/humphry/1190000000407658)