ECMAScript 6 入门

作者：阮一峰

目录

其他

- 源码
- 修订历史
- 反馈意见

数值的扩展

---

1. 二进制和八进制表示法

ES6 提供了二进制和八进制数值的新的写法，分别用前缀 `0b` （或 `0B` ）和 `0o` （或 `0O` ）表示。

```
0b111110111 === 503 // true
0o767 === 503 // true
```

从 ES5 开始，在严格模式之中，八进制就不再允许使用前缀 `0` 表示，ES6 进一步明确，要使用前缀 `0o` 表示。

```
// 非严格模式
(function(){
  console.log(0o11 === 011);
})() // true

// 严格模式
(function(){
  'use strict';
  console.log(0o11 === 011);
})() // Uncaught SyntaxError: Octal literals are not allowed in strict mode.
```

如果要将 `0b` 和 `0o` 前缀的字符串数值转为十进制，要使用 `Number` 方法。

```
Number('0b111')  // 7
Number('0o10')  // 8
```

---

2. 数值分隔符

欧美语言中，较长的数值允许每三位添加一个分隔符（通常是一个逗号），增加数值的可读性。比如， `1000` 可以写作 `1,000` 。

ES2021，允许 JavaScript 的数值使用下划线（ `_` ）作为分隔符。

```
let budget = 1_000_000_000_000;
budget === 10 ** 12 // true
```

这个数值分隔符没有指定间隔的位数，也就是说，可以每三位添加一个分隔符，也可以每一位、每两位、每四位添加一个。

```
123_00 === 12_300 // true

12345_00 === 123_4500 // true
12345_00 === 1_234_500 // true
```

小数和科学计数法也可以使用数值分隔符。

```
// 小数
0.000_001

// 科学计数法
1e10_000
```

数值分隔符有几个使用注意点。

- 不能放在数值的最前面（leading）或最后面（trailing）。
- 不能两个或两个以上的分隔符连在一起。
- 小数点的前后不能有分隔符。
- 科学计数法里面，表示指数的 `e` 或 `E` 前后不能有分隔符。

下面的写法都会报错。

```
// 全部报错
3_.141
3._141
1_e12
1e_12
123__456
_1464301
1464301_
```

除了十进制，其他进制的数值也可以使用分隔符。

```
// 二进制
0b1010_0001_1000_0101
// 十六进制
0xA0_B0_C0
```

可以看到，数值分隔符可以按字节顺序分隔数值，这在操作二进制位时，非常有用。

注意，分隔符不能紧跟着进制的前缀 `0b` 、 `0B` 、 `0o` 、 `0O` 、 `0x` 、 `0X` 。

```
// 报错
0_b111111000
0b_111111000
```

数值分隔符只是一种书写便利，对于 JavaScript 内部数值的存储和输出，并没有影响。

```
let num = 12_345;

num // 12345
num.toString() // 12345
```

上面示例中，变量 `num` 的值为 `12_345` ，但是内部存储和输出的时候，都不会有数值分隔符。

下面三个将字符串转成数值的函数，不支持数值分隔符。主要原因是语言的设计者认为，数值分隔符主要是为了编码时书写数值的方便，而不是为了处理外部输入的数据。

- Number()
- parseInt()
- parseFloat()

```
Number('123_456') // NaN
parseInt('123_456') // 123
```

---

3. Number.isFinite(), Number.isNaN()

ES6 在 `Number` 对象上，新提供了 `Number.isFinite()` 和 `Number.isNaN()` 两个方法。

`Number.isFinite()` 用来检查一个数值是否为有限的（finite），即不是 `Infinity`。

```
Number.isFinite(15); // true
Number.isFinite(0.8); // true
Number.isFinite(NaN); // false
Number.isFinite(Infinity); // false
Number.isFinite(-Infinity); // false
Number.isFinite('foo'); // false
Number.isFinite('15'); // false
Number.isFinite(true); // false
```

注意，如果参数类型不是数值，`Number.isFinite` 一律返回 `false`。

`Number.isNaN()` 用来检查一个值是否为 `NaN`。

```
Number.isNaN(NaN) // true
Number.isNaN(15) // false
Number.isNaN('15') // false
Number.isNaN(true) // false
Number.isNaN(9/NaN) // true
Number.isNaN('true' / 0) // true
Number.isNaN('true' / 'true') // true
```

如果参数类型不是 `NaN`，`Number.isNaN` 一律返回 `false`。

它们与传统的全局方法 `isFinite()` 和 `isNaN()` 的区别在于，传统方法先调用 `Number()` 将非数值的值转为数值，再进行判断，而这两个新方法只对数值有效，`Number.isFinite()` 对于非数值一律返回 `false`，`Number.isNaN()` 只有对于 `NaN` 才返回 `true`，非 `NaN` 一律返回 `false`。

```
isFinite(25) // true
isFinite("25") // true
Number.isFinite(25) // true
Number.isFinite("25") // false

isNaN(NaN) // true
isNaN("NaN") // true
Number.isNaN(NaN) // true
Number.isNaN("NaN") // false
Number.isNaN(1) // false
```

---

4. Number.parseInt(), Number.parseFloat()

ES6 将全局方法 `parseInt()` 和 `parseFloat()`，移植到 `Number` 对象上面，行为完全保持不变。

```
// ES5的写法
parseInt('12.34') // 12
```

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

// ES6的写法
Number.parseInt('12.34') // 12
Number.parseFloat('123.45#') // 123.45
```

这样做的目的，是逐步减少全局性方法，使得语言逐步模块化。

```
Number.parseInt === parseInt // true
Number.parseFloat === parseFloat // true
```

---

5. Number.isInteger()

`Number.isInteger()` 用来判断一个数值是否为整数。

```
Number.isInteger(25) // true
Number.isInteger(25.1) // false
```

JavaScript 内部，整数和浮点数采用的是同样的储存方法，所以 25 和 25.0 被视为同一个值。

```
Number.isInteger(25) // true
Number.isInteger(25.0) // true
```

如果参数不是数值， `Number.isInteger` 返回 `false` 。

```
Number.isInteger() // false
Number.isInteger(null) // false
Number.isInteger('15') // false
Number.isInteger(true) // false
```

注意，由于 JavaScript 采用 IEEE 754 标准，数值存储为64位双精度格式，数值精度最多可以达到 53 个二进制位（1 个隐藏位与 52 个有效位）。如果数值的精度超过这个限度，第54位及后面的位就会被丢弃，这种情况下， `Number.isInteger` 可能会误判。

```
Number.isInteger(3.0000000000000002) // true
```

上面代码中， `Number.isInteger` 的参数明明不是整数，但是会返回 `true` 。原因就是这个小数的精度达到了小数点后16个十进制位，转成二进制位超过了53个二进制位，导致最后的那个 `2` 被丢弃了。

类似的情况还有，如果一个数值的绝对值小于 `Number.MIN_VALUE` （5E-324），即小于 JavaScript 能够分辨的最小值，会被自动转为0。这时， `Number.isInteger` 也会误判。

```
Number.isInteger(5E-324) // false
Number.isInteger(5E-325) // true
```

上面代码中， `5E-325` 由于值太小，会被自动转为0，因此返回 `true` 。

总之，如果对数据精度的要求较高，不建议使用 `Number.isInteger()` 判断一个数值是否为整数。

---

6. Number.EPSILON

ES6 在 `Number` 对象上面，新增一个极小的常量 `Number.EPSILON` 。根据规格，它表示 1 与大于 1 的最小浮点数之间的差。

对于 64 位浮点数来说，大于 1 的最小浮点数相当于二进制的 `1.00..001` ，小数点后面有连续 51 个零。这个值减去 1 之后，就等于 2 的 -52 次方。

```
Number.EPSILON === Math.pow(2, -52)
// true
Number.EPSILON
// 2.220446049250313e-16
Number.EPSILON.toFixed(20)
// "0.00000000000000022204"
```

`Number.EPSILON` 实际上是 JavaScript 能够表示的最小精度。误差如果小于这个值，就可以认为已经没有意义了，即不存在误差了。

引入一个这么小的量的目的，在于为浮点数计算，设置一个误差范围。我们知道浮点数计算是不精确的。

```
0.1 + 0.2
// 0.30000000000000004

0.1 + 0.2 - 0.3
// 5.551115123125783e-17

5.551115123125783e-17.toFixed(20)
// '0.00000000000000005551'
```

上面代码解释了，为什么比较 `0.1 + 0.2` 与 `0.3` 得到的结果是 `false` 。

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

`Number.EPSILON` 可以用来设置“能够接受的误差范围”。比如，误差范围设为 2 的-50 次方（即 `Number.EPSILON * Math.pow(2, 2)` ），即如果两个浮点数的差小于这个值，我们就认为这两个浮点数相等。

```
5.551115123125783e-17 < Number.EPSILON * Math.pow(2, 2)
// true
```

因此， `Number.EPSILON` 的实质是一个可以接受的最小误差范围。

```
function withinErrorMargin (left, right) {
  return Math.abs(left - right) < Number.EPSILON * Math.pow(2, 2);
}

0.1 + 0.2 === 0.3 // false
withinErrorMargin(0.1 + 0.2, 0.3) // true

1.1 + 1.3 === 2.4 // false
withinErrorMargin(1.1 + 1.3, 2.4) // true
```

上面的代码为浮点数运算，部署了一个误差检查函数。

---

7. 安全整数和 Number.isSafeInteger()

JavaScript 能够准确表示的整数范围在 `-2^53` 到 `2^53` 之间（不含两个端点），超过这个范围，无法精确表示这个值。

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

9007199254740992  // 9007199254740992
9007199254740993  // 9007199254740992

Math.pow(2, 53) === Math.pow(2, 53) + 1
// true
```

上面代码中，超出 2 的 53 次方之后，一个数就不精确了。

ES6 引入了 `Number.MAX_SAFE_INTEGER` 和 `Number.MIN_SAFE_INTEGER` 这两个常量，用来表示这个范围的上下限。

```
Number.MAX_SAFE_INTEGER === Math.pow(2, 53) - 1
// true
Number.MAX_SAFE_INTEGER === 9007199254740991
// true

Number.MIN_SAFE_INTEGER === -Number.MAX_SAFE_INTEGER
// true
Number.MIN_SAFE_INTEGER === -9007199254740991
// true
```

上面代码中，可以看到 JavaScript 能够精确表示的极限。

`Number.isSafeInteger()` 则是用来判断一个整数是否落在这个范围之内。

```
Number.isSafeInteger('a') // false
Number.isSafeInteger(null) // false
Number.isSafeInteger(NaN) // false
Number.isSafeInteger(Infinity) // false
Number.isSafeInteger(-Infinity) // false

Number.isSafeInteger(3) // true
Number.isSafeInteger(1.2) // false
Number.isSafeInteger(9007199254740990) // true
Number.isSafeInteger(9007199254740992) // false

Number.isSafeInteger(Number.MIN_SAFE_INTEGER - 1) // false
Number.isSafeInteger(Number.MIN_SAFE_INTEGER) // true
Number.isSafeInteger(Number.MAX_SAFE_INTEGER) // true
Number.isSafeInteger(Number.MAX_SAFE_INTEGER + 1) // false
```

这个函数的实现很简单，就是跟安全整数的两个边界值比较一下。

```
Number.isSafeInteger = function (n) {
  return (typeof n === 'number' &&
    Math.round(n) === n &&
    Number.MIN_SAFE_INTEGER <= n &&
    n <= Number.MAX_SAFE_INTEGER);
}
```

实际使用这个函数时，需要注意。验证运算结果是否落在安全整数的范围内，不要只验证运算结果，而要同时验证参与运算的每个值。

```
Number.isSafeInteger(9007199254740993)
// false
Number.isSafeInteger(990)
// true
Number.isSafeInteger(9007199254740993 - 990)
// true
9007199254740993 - 990
```

```
// 返回结果 9007199254740002
// 正确答案应该是 9007199254740003
```

上面代码中，`9007199254740993` 不是一个安全整数，但是 `Number.isSafeInteger` 会返回结果，显示计算结果是安全的。这是因为，这个数超出了精度范围，导致在计算机内部，以 `9007199254740992` 的形式储存。

```
9007199254740993 === 9007199254740992
// true
```

所以，如果只验证运算结果是否为安全整数，很可能得到错误结果。下面的函数可以同时验证两个运算数和运算结果。

```
function trusty (left, right, result) {
  if (
    Number.isSafeInteger(left) &&
    Number.isSafeInteger(right) &&
    Number.isSafeInteger(result)
  ) {
    return result;
  }
  throw new RangeError('Operation cannot be trusted!');
}

trusty(9007199254740993, 990, 9007199254740993 - 990)
// RangeError: Operation cannot be trusted!

trusty(1, 2, 3)
// 3
```

---

8. Math 对象的扩展

ES6 在 Math 对象上新增了 17 个与数学相关的方法。所有这些方法都是静态方法，只能在 Math 对象上调用。

Math.trunc()

`Math.trunc` 方法用于去除一个数的小数部分，返回整数部分。

```
Math.trunc(4.1) // 4
Math.trunc(4.9) // 4
Math.trunc(-4.1) // -4
Math.trunc(-4.9) // -4
Math.trunc(-0.1234) // -0
```

对于非数值，`Math.trunc` 内部使用 `Number` 方法将其先转为数值。

```
Math.trunc('123.456') // 123
Math.trunc(true) //1
Math.trunc(false) // 0
Math.trunc(null) // 0
```

对于空值和无法截取整数的值，返回 `NaN` 。

```
Math.trunc(NaN);      // NaN
Math.trunc('foo');    // NaN
Math.trunc();         // NaN
Math.trunc(undefined) // NaN
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.trunc = Math.trunc || function(x) {
  return x < 0 ? Math.ceil(x) : Math.floor(x);
};
```

Math.sign()

`Math.sign` 方法用来判断一个数到底是正数、负数、还是零。对于非数值，会先将其转换为数值。

它会返回五种值。

- 参数为正数，返回 `+1`；
- 参数为负数，返回 `-1`；
- 参数为 0，返回 `0`；
- 参数为-0，返回 `-0`；
- 其他值，返回 `NaN`。

```
Math.sign(-5) // -1
Math.sign(5) // +1
Math.sign(0) // +0
Math.sign(-0) // -0
Math.sign(NaN) // NaN
```

如果参数是非数值，会自动转为数值。对于那些无法转为数值的值，会返回 `NaN`。

```
Math.sign('')  // 0
Math.sign(true)  // +1
Math.sign(false)  // 0
Math.sign(null)  // 0
Math.sign('9')  // +1
Math.sign('foo')  // NaN
Math.sign()  // NaN
Math.sign(undefined)  // NaN
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.sign = Math.sign || function(x) {
  x = +x; // convert to a number
  if (x === 0 || isNaN(x)) {
    return x;
  }
  return x > 0 ? 1 : -1;
};
```

Math.cbrt()

`Math.cbrt()` 方法用于计算一个数的立方根。

```
Math.cbrt(-1) // -1
Math.cbrt(0)  // 0
Math.cbrt(1)  // 1
Math.cbrt(2)  // 1.2599210498948732
```

对于非数值， `Math.cbrt()` 方法内部也是先使用 `Number()` 方法将其转为数值。

```
Math.cbrt('8') // 2
Math.cbrt('hello') // NaN
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.cbrt = Math.cbrt || function(x) {
  var y = Math.pow(Math.abs(x), 1/3);
  return x < 0 ? -y : y;
};
```

Math.clz32()

`Math.clz32()` 方法将参数转为 32 位无符号整数的形式，然后返回这个 32 位值里面有多少个前导 0。

```
Math.clz32(0) // 32
Math.clz32(1) // 31
Math.clz32(1000) // 22
Math.clz32(0b01000000000000000000000000000000) // 1
Math.clz32(0b00100000000000000000000000000000) // 2
```

上面代码中，0 的二进制形式全为 0，所以有 32 个前导 0；1 的二进制形式是 `0b1` ，只占 1 位，所以 32 位之中有 31 个前导 0；1000 的二进制形式是 `0b1111101000` ，一共有 10 位，所以 32 位之中有 22 个前导 0。

`clz32` 这个函数名就来自”count leading zero bits in 32-bit binary representation of a number“（计算一个数的 32 位二进制形式的前导 0 的个数）的缩写。

左移运算符（ `<<` ）与 `Math.clz32` 方法直接相关。

```
Math.clz32(0) // 32
Math.clz32(1) // 31
Math.clz32(1 << 1) // 30
Math.clz32(1 << 2) // 29
Math.clz32(1 << 29) // 2
```

对于小数， `Math.clz32` 方法只考虑整数部分。

```
Math.clz32(3.2) // 30
Math.clz32(3.9) // 30
```

对于空值或其他类型的值， `Math.clz32` 方法会将它们先转为数值，然后再计算。

```
Math.clz32() // 32
Math.clz32(NaN) // 32
Math.clz32(Infinity) // 32
Math.clz32(null) // 32
Math.clz32('foo') // 32
Math.clz32([]) // 32
Math.clz32({}) // 32
Math.clz32(true) // 31
```

Math.imul()

`Math.imul` 方法返回两个数以 32 位带符号整数形式相乘的结果，返回的也是一个 32 位的带符号整数。

```
Math.imul(2, 4)   // 8
Math.imul(-1, 8)  // -8
Math.imul(-2, -2) // 4
```

如果只考虑最后 32 位，大多数情况下， `Math.imul(a, b)` 与 `a * b` 的结果是相同的，即该方法等同于 `(a * b)|0` 的效果（超过 32 位的部分溢出）。之所以需要部署这个方法，是因为 JavaScript 有精度限制，超过 2 的 53 次方的值无法精确表示。这就是说，对于那些很大的数的乘法，低位数值往往都是不精确的， `Math.imul` 方法可以返回正确的低位数值。

```
(0x7fffffff * 0x7fffffff)|0 // 0
```

上面这个乘法算式，返回结果为 0。但是由于这两个二进制数的最低位都是 1，所以这个结果肯定是不正确的，因为根据二进制乘法，计算结果的二进制最低位应该也是 1。这个错误就是因为它们的乘积超过了 2 的 53 次方，JavaScript 无法保存额外的精度，就把低位的值都变成了 0。 `Math.imul` 方法可以返回正确的值 1。

```
Math.imul(0x7fffffff, 0x7fffffff) // 1
```

Math.fround()

`Math.fround` 方法返回一个数的32位单精度浮点数形式。

对于32位单精度格式来说，数值精度是24个二进制位（1 位隐藏位与 23 位有效位），所以对于 -2^{24} 至 2^{24} 之间的整数（不含两个端点），返回结果与参数本身一致。

```
Math.fround(0)   // 0
Math.fround(1)   // 1
Math.fround(2 ** 24 - 1)   // 16777215
```

如果参数的绝对值大于 2^{24}，返回的结果便开始丢失精度。

```
Math.fround(2 ** 24)       // 16777216
Math.fround(2 ** 24 + 1)   // 16777216
```

`Math.fround` 方法的主要作用，是将64位双精度浮点数转为32位单精度浮点数。如果小数的精度超过24个二进制位，返回值就会不同于原值，否则返回值不变（即与64位双精度值一致）。

```
// 未丢失有效精度
Math.fround(1.125) // 1.125
Math.fround(7.25)  // 7.25

// 丢失精度
Math.fround(0.3)   // 0.30000001192092896
Math.fround(0.7)   // 0.699999988079071
Math.fround(1.0000000123) // 1
```

对于 `NaN` 和 `Infinity`，此方法返回原值。对于其它类型的非数值，`Math.fround` 方法会先将其转为数值，再返回单精度浮点数。

```
Math.fround(NaN)      // NaN
Math.fround(Infinity) // Infinity

Math.fround('5')      // 5
Math.fround(true)     // 1
Math.fround(null)     // 0
Math.fround([])       // 0
Math.fround({})       // NaN
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.fround = Math.fround || function (x) {
  return new Float32Array([x])[0];
};
```

Math.hypot()

`Math.hypot` 方法返回所有参数的平方和的平方根。

```
Math.hypot(3, 4);        // 5
Math.hypot(3, 4, 5);     // 7.0710678118654755
Math.hypot();            // 0
Math.hypot(NaN);         // NaN
Math.hypot(3, 4, 'foo'); // NaN
Math.hypot(3, 4, '5');   // 7.0710678118654755
Math.hypot(-3);          // 3
```

上面代码中，3 的平方加上 4 的平方，等于 5 的平方。

如果参数不是数值，`Math.hypot` 方法会将其转为数值。只要有一个参数无法转为数值，就会返回 NaN。

对数方法

ES6 新增了 4 个对数相关方法。

(1) Math.expm1()

`Math.expm1(x)` 返回 $e^x - 1$，即 `Math.exp(x) - 1`。

```
Math.expm1(-1) // -0.6321205588285577
Math.expm1(0)  // 0
```

```
Math.expm1(1)  // 1.718281828459045
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.expm1 = Math.expm1 || function(x) {
  return Math.exp(x) - 1;
};
```

(2) Math.log1p()

`Math.log1p(x)` 方法返回 `1 + x` 的自然对数，即 `Math.log(1 + x)` 。如果 `x` 小于-1，返回 `NaN` 。

```
Math.log1p(1)  // 0.6931471805599453
Math.log1p(0)  // 0
Math.log1p(-1) // -Infinity
Math.log1p(-2) // NaN
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.log1p = Math.log1p || function(x) {
  return Math.log(1 + x);
};
```

(3) Math.log10()

`Math.log10(x)` 返回以 10 为底的 `x` 的对数。如果 `x` 小于 0，则返回 NaN。

```
Math.log10(2)      // 0.3010299956639812
Math.log10(1)      // 0
Math.log10(0)      // -Infinity
Math.log10(-2)     // NaN
Math.log10(100000) // 5
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.log10 = Math.log10 || function(x) {
  return Math.log(x) / Math.LN10;
};
```

(4) Math.log2()

`Math.log2(x)` 返回以 2 为底的 `x` 的对数。如果 `x` 小于 0，则返回 NaN。

```
Math.log2(3)       // 1.584962500721156
Math.log2(2)       // 1
Math.log2(1)       // 0
Math.log2(0)       // -Infinity
Math.log2(-2)      // NaN
Math.log2(1024)    // 10
Math.log2(1 << 29) // 29
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.log2 = Math.log2 || function(x) {
  return Math.log(x) / Math.LN2;
};
```

双曲函数方法

ES6 新增了 6 个双曲函数方法。

- `Math.sinh(x)` 返回 `x` 的双曲正弦（hyperbolic sine）
- `Math.cosh(x)` 返回 `x` 的双曲余弦（hyperbolic cosine）
- `Math.tanh(x)` 返回 `x` 的双曲正切（hyperbolic tangent）
- `Math.asinh(x)` 返回 `x` 的反双曲正弦（inverse hyperbolic sine）
- `Math.acosh(x)` 返回 `x` 的反双曲余弦（inverse hyperbolic cosine）
- `Math.atanh(x)` 返回 `x` 的反双曲正切（inverse hyperbolic tangent）

---

9. BigInt 数据类型

简介

JavaScript 所有数字都保存成 64 位浮点数，这给数值的表示带来了两大限制。一是数值的精度只能到 53 个二进制位（相当于 16 个十进制位），大于这个范围的整数，JavaScript 是无法精确表示，这使得 JavaScript 不适合进行科学和金融方面的精确计算。二是大于或等于2的1024次方的数值，JavaScript 无法表示，会返回 `Infinity` 。

```
// 超过 53 个二进制位的数值，无法保持精度
Math.pow(2, 53) === Math.pow(2, 53) + 1 // true

// 超过 2 的 1024 次方的数值，无法表示
Math.pow(2, 1024) // Infinity
```

ES2020 引入了一种新的数据类型 BigInt（大整数），来解决这个问题，这是 ECMAScript 的第八种数据类型。BigInt 只用来表示整数，没有位数的限制，任何位数的整数都可以精确表示。

```
const a = 2172141653n;
const b = 15346349309n;

// BigInt 可以保持精度
a * b // 33334444555566667777n

// 普通整数无法保持精度
Number(a) * Number(b) // 33334444555566670000
```

为了与 Number 类型区别，BigInt 类型的数据必须添加后缀 `n` 。

```
1234 // 普通整数
1234n // BigInt

// BigInt 的运算
1n + 2n // 3n
```

BigInt 同样可以使用各种进制表示，都要加上后缀 `n` 。

```
0b1101n // 二进制
0o777n // 八进制
0xFFn // 十六进制
```

BigInt 与普通整数是两种值，它们之间并不相等。

```
42n === 42 // false
```

`typeof` 运算符对于 BigInt 类型的数据返回 `bigint`。

```
typeof 123n // 'bigint'
```

BigInt 可以使用负号（`-`），但是不能使用正号（`+`），因为会与 asm.js 冲突。

```
-42n // 正确
+42n // 报错
```

JavaScript 以前不能计算70的阶乘（即 `70!`），因为超出了可以表示的精度。

```
let p = 1;
for (let i = 1; i <= 70; i++) {
  p *= i;
}
console.log(p); // 1.197857166996989e+100
```

现在支持大整数了，就可以算了，浏览器的开发者工具运行下面代码，就OK。

```
let p = 1n;
for (let i = 1n; i <= 70n; i++) {
  p *= i;
}
console.log(p); // 11978571...00000000n
```

BigInt 函数

JavaScript 原生提供 `BigInt` 函数，可以用它生成 BigInt 类型的数值。转换规则基本与 `Number()` 一致，将其他类型的值转为 BigInt。

```
BigInt(123) // 123n
BigInt('123') // 123n
BigInt(false) // 0n
BigInt(true) // 1n
```

`BigInt()` 函数必须有参数，而且参数必须可以正常转为数值，下面的用法都会报错。

```
new BigInt() // TypeError
BigInt(undefined) //TypeError
BigInt(null) // TypeError
BigInt('123n') // SyntaxError
BigInt('abc') // SyntaxError
```

上面代码中，尤其值得注意字符串 `123n` 无法解析成 Number 类型，所以会报错。

参数如果是小数，也会报错。

```
BigInt(1.5) // RangeError
BigInt('1.5') // SyntaxError
```

BigInt 继承了 Object 对象的两个实例方法。

- `BigInt.prototype.toString()`
- `BigInt.prototype.valueOf()`

它还继承了 Number 对象的一个实例方法。

- `BigInt.prototype.toLocaleString()`

此外，还提供了三个静态方法。

- `BigInt.asUintN(width, BigInt)`： 给定的 BigInt 转为 0 到 $2^{width} - 1$ 之间对应的值。
- `BigInt.asIntN(width, BigInt)`：给定的 BigInt 转为 $-2^{width - 1}$ 到 $2^{width - 1} - 1$ 之间对应的值。
- `BigInt.parseInt(string[, radix])`：近似于 `Number.parseInt()`，将一个字符串转换成指定进制的 BigInt。

```
const max = 2n ** (64n - 1n) - 1n;

BigInt.asIntN(64, max)
// 9223372036854775807n
BigInt.asIntN(64, max + 1n)
// -9223372036854775808n
BigInt.asUintN(64, max + 1n)
// 9223372036854775808n
```

上面代码中，`max` 是64位带符号的 BigInt 所能表示的最大值。如果对这个值加 `1n`，`BigInt.asIntN()` 将会返回一个负值，因为这时新增的一位将被解释为符号位。而 `BigInt.asUintN()` 方法由于不存在符号位，所以可以正确返回结果。

如果 `BigInt.asIntN()` 和 `BigInt.asUintN()` 指定的位数，小于数值本身的位数，那么头部的位将被舍弃。

```
const max = 2n ** (64n - 1n) - 1n;

BigInt.asIntN(32, max) // -1n
BigInt.asUintN(32, max) // 4294967295n
```

上面代码中，`max` 是一个64位的 BigInt，如果转为32位，前面的32位都会被舍弃。

下面是 `BigInt.parseInt()` 的例子。

```
// Number.parseInt() 与 BigInt.parseInt() 的对比
Number.parseInt('9007199254740993', 10)
// 9007199254740992
BigInt.parseInt('9007199254740993', 10)
// 9007199254740993n
```

上面代码中，由于有效数字超出了最大限度，`Number.parseInt` 方法返回的结果是不精确的，而 `BigInt.parseInt` 方法正确返回了对应的 BigInt。

对于二进制数组，BigInt 新增了两个类型 `BigUint64Array` 和 `BigInt64Array`，这两种数据类型返回的都是64位 BigInt。`DataView` 对象的实例方法 `DataView.prototype.getBigInt64()` 和 `DataVie 上一章 .g 下一章 ()`，返回的也是 BigInt。

转换规则

可以使用 `Boolean()`、`Number()` 和 `String()` 这三个方法，将 BigInt 可以转为布尔值、数值和字符串类型。

```
Boolean(0n) // false
Boolean(1n) // true
Number(1n)  // 1
String(1n)  // "1"
```

上面代码中，注意最后一个例子，转为字符串时后缀 `n` 会消失。

另外，取反运算符（`!`）也可以将 BigInt 转为布尔值。

```
!0n // true
!1n // false
```

数学运算

数学运算方面，BigInt 类型的 `+`、`-`、`*` 和 `**` 这四个二元运算符，与 Number 类型的行为一致。除法运算 `/` 会舍去小数部分，返回一个整数。

```
9n / 5n
// 1n
```

几乎所有的数值运算符都可以用在 BigInt，但是有两个例外。

- 不带符号的右移位运算符 `>>>`
- 一元的求正运算符 `+`

上面两个运算符用在 BigInt 会报错。前者是因为 `>>>` 运算符是不带符号的，但是 BigInt 总是带有符号的，导致该运算无意义，完全等同于右移运算符 `>>`。后者是因为一元运算符 `+` 在 asm.js 里面总是返回 Number 类型，为了不破坏 asm.js 就规定 `+1n` 会报错。

BigInt 不能与普通数值进行混合运算。

```
1n + 1.3 // 报错
```

上面代码报错是因为无论返回的是 BigInt 或 Number，都会导致丢失精度信息。比如 `(2n**53n + 1n) + 0.5` 这个表达式，如果返回 BigInt 类型，`0.5` 这个小数部分会丢失；如果返回 Number 类型，有效精度只能保持 53 位，导致精度下降。

同样的原因，如果一个标准库函数的参数预期是 Number 类型，但是得到的是一个 BigInt，就会报错。

```
// 错误的写法
Math.sqrt(4n) // 报错

// 正确的写法
Math.sqrt(Number(4n)) // 2
```

上面代码中，`Math.sqrt` 的参数预期是 Number 类型，如果是 BigInt 就会报错，必须先用 `Number` 方法转一下类型，才能进行计算。

asm.js 里面，`|0` 跟在一个数值的后面会返回一个32位整数。根据不能与 Number 类型混合运算的规则，BigInt 如果与 `|0` 进行运算会报错。

```
1n | 0 // 报错
```

其他运算

BigInt 对应的布尔值，与 Number 类型一致，即 `0n` 会转为 `false`，其他值转为 `true`。

```
if (0n) {
  console.log('if');
} else {
  console.log('else');
}
// else
```

上面代码中，`0n` 对应 `false`，所以会进入 `else` 子句。

比较运算符（比如 `>`）和相等运算符（`==`）允许 BigInt 与其他类型的值混合计算，因为这样做不会损失精度。

```
0n < 1 // true
0n < true // true
0n == 0 // true
0n == false // true
0n === 0 // false
```

BigInt 与字符串混合运算时，会先转为字符串，再进行运算。

```
'' + 123n // "123"
```

留言