ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证

目录

其他

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. 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）。

```
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
```

ES5 可以通过下面的代码，部署 `Number.isFinite` 方法。

```
(function (global) {
  var global_isFinite = global.isFinite;

  Object.defineProperty(Number, 'isFinite', {
    value: function isFinite(value) {
      return typeof value === 'number' && global_isFinite(value);
    },
    configurable: true,
    enumerable: false,
    writable: true
  });
})(this);
```

`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
```

ES5 通过下面的代码，部署 `Number.isNaN()` 。

```
(function (global) {
  var global_isNaN = global.isNaN;

  Object.defineProperty(Number, 'isNaN', {
    value: function isNaN(value) {
      return typeof value === 'number' && global_isNaN(value);
    },
    configurable: true,
    enumerable: false,
    writable: true
  });
})(this);
```

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
```

---

3. 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
```

---

4. Number.isInteger()

`Number.isInteger()` 用来判断一个值是否为整数。需要注意的是，在 JavaScript 内部，整数和浮点数是同样的储存方法，所以3和3.0被视为同一个值。

```
Number.isInteger(25) // true
Number.isInteger(25.0) // true
Number.isInteger(25.1) // false
Number.isInteger("15") // false
Number.isInteger(true) // false
```

ES5 可以通过下面的代码，部署 `Number.isInteger()` 。

```
(function (global) {
  var floor = Math.floor,
    isFinite = global.isFinite;

  Object.defineProperty(Number, 'isInteger', {
    value: function isInteger(value) {
      return typeof value === 'number' &&
        isFinite(value) &&
        floor(value) === value;
    },
    configurable: true,
    enumerable: false,
    writable: true
  });
})(this);
```

---

5. Number.EPSILON

ES6在Number对象上面，新增一个极小的常量`Number.EPSILON`。

```
Number.EPSILON
// 2.220446049250313e-16
Number.EPSILON.toFixed(20)
// '0.00000000000000022204'
```

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'
```

但是如果这个误差能够小于`Number.EPSILON`，我们就可以认为得到了正确结果。

```
5.551115123125783e-17 < Number.EPSILON
// true
```

因此，`Number.EPSILON`的实质是一个可以接受的误差范围。

```
function withinErrorMargin (left, right) {
  return Math.abs(left - right) < Number.EPSILON;
}
withinErrorMargin(0.1 + 0.2, 0.3)
// true
withinErrorMargin(0.2 + 0.2, 0.3)
// false
```

上面的代码为浮点数运算，部署了一个误差检查函数。

---

6. 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
```

---

7. 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
```

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
```

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
Math.sign('9'); // +1
Math.sign('foo'); // NaN
Math.sign();      // 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)  // 1.2599210498948734
```

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()

JavaScript的整数使用32位二进制形式表示， `Math.clz32` 方法返回一个数的32位无符号整数形式有多少个前导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 b… …es … …of a number"（计算32位整数的前导0）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方法返回一个数的单精度浮点数形式。

```
Math.fround(0)     // 0
Math.fround(1)     // 1
Math.fround(1.337) // 1.3370000123977661
Math.fround(1.5)   // 1.5
Math.fround(NaN)   // NaN
```

对于整数来说，`Math.fround`方法返回结果不会有任何不同，区别主要是那些无法用64个二进制位精确表示的小数。这时，`Math.fround`方法会返回最接近这个小数的单精度浮点数。

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）

---

8. Math.signbit()

`Math.sign()` 用来判断一个值的正负，但是如果参数是 `-0`，它会返回 `-0`。

```
Math.sign(-0) // -0
```

这导致对于判断符号位的正负，`Math.sign()` 不是很有用。JavaScript 内部使用64位浮点数（国际标准IEEE 754）表示数值，IEEE 754规定第一位是符号位，`0` 表示正数，`1` 表示负数。所以会有两种零，`+0` 是符号位为 `0` 时的零值，`-0` 是符号位为 `1` 时的零值。实际编程中，判断一个值是 `+0` 还是 `-0` 非常麻烦，因为它们是相等的。

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

目前，有一个提案，引入了 `Math.signbit()` 方法判断一个数的符号位是否设置了。

```
Math.signbit(2) //false
Math.signbit(-2) //true
Math.signbit(0) //false
Math.signbit(-0) //true
```

可以看到，该方法正确返回了`-0`的符号位是设置了的。

该方法的算法如下。

- 如果参数是`NaN`，返回`false`
- 如果参数是`-0`，返回`true`
- 如果参数是负值，返回`true`
- 其他情况返回`false`

---

9. 指数运算符

ES2016 新增了一个指数运算符（`**`）。

```
2 ** 2 // 4
2 ** 3 // 8
```

指数运算符可以与等号结合，形成一个新的赋值运算符（`**=`）。

```
let a = 1.5;
a **= 2;
// 等同于 a = a * a;

let b = 4;
b **= 3;
// 等同于 b = b * b * b;
```

注意，在 V8 引擎中，指数运算符与`Math.pow`的实现不相同，对于特别大的运算结果，两者会有细微的差异。

```
Math.pow(99, 99)
// 3.697296376497263e+197

99 ** 99
// 3.697296376497268e+197
```

上面代码中，两个运算结果的最后一位有效数字是有差异的。

---

10. Integer 数据类型

简介

JavaScript 所有数字都保存成64位浮点数，这决定了整数的精确程度只能到53个二进制位。大于这个范围的整数，JavaScript 是无法精确表示的，这使得 JavaScript 不适合进行科学和金融方面的精确计算。

现在有一个提案，引入了新的数据类型 Integer（整数），来解决这个问题。整数类型的数据只用来表示整数，没有位数的限制，任何位数的整数都可以精确表示。

为了与 Number 类型区别，Integer 类型的数据必须使用后缀`n`表示。

```
1n + 2n // 3n
```

二进制、八进制、十六进制的表示法，都要加上后缀 `n`。

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

`typeof` 运算符对于 Integer 类型的数据返回 `integer`。

```
typeof 123n
// 'integer'
```

JavaScript 原生提供 `Integer` 对象，用来生成 Integer 类型的数值。转换规则基本与 `Number()` 一致。

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

以下的用法会报错。

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

运算

在数学运算方面，Integer 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 Number 运算符都可以用在 Integer，但是有两个除外：不带符号的右移位运算符 `>>>` 和一元的求正运算符 `+`，使用时会报错。前者是因为 `>>>` 要求最高位补0，但是 Integer 类型没有最高位，导致这个运算符无意义。后者是因为一元运算符 `+` 在 asm.js 里面总是返回 Number 类型或者报错。

Integer 类型不能与 Number 类型进行混合运算。

```
1n + 1
// 报错
```

这是因为无论返回的是 Integer 或 Number，都会导致丢失信息。比如 `(2n**53n + 1n) + 0.5` 这个表达式，如果返回 Integer 类型，`0.5` 这个小数部分会丢失；如果返回 Number 类型，会超过 53 位精确数字，精度下降。

相等运算符（`==`）会改变数据类型，也是不允许混合使用。

```
0n == 0
// 报错 TypeError

0n == false
// 报错 TypeError
```

精确相等运算符（`===`）不会改变数据类型，因此可以混合使用。

```
0n === 0
// false
```

留言