

ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证



目录

- 0.前言
- 1.ECMAScript 6简介
- 2.let 和 const 命令
- 3.变量的解构赋值
- 4.字符串的扩展
- 5.正则的扩展
- 6.数值的扩展
- 7.函数的扩展
- 8.数组的扩展
- 9.对象的扩展
- 10.对象的新增方法
- 11.Symbol
- 12.Set 和 Map 数据结构
- 13.Proxy
- 14.Reflect
- 15.Promise 对象
- 16.Iterator 和 for...of 循环
- 17.Generator 函数的语法
- 18.Generator 函数的异步应用
- 19.async 函数
- 20.Class 的基本语法
- 21.Class 的继承
- 22.Decorator
- 23.Module 的语法
- 24.Module 的加载实现
- 25.编程风格
- 26.读懂规格
- 27.ArrayBuffer
- 28.最新提案
- 29.参考链接

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
- 2.Number.isFinite(), Number.isNaN()
- 3.Number.parseInt(), Number.parseFloat()
- 4.Number.isInteger()
- 5.Number.EPSILON
- 6.安全整数和 Number.isSafeInteger()
- 7.Math 对象的扩展
- 8.指数运算符

[上一章](#)

[下一章](#)

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），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
```

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位及以后会被舍去，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。

5. 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)
// '0.0000000000000005551'
```

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。

[上一章](#)

[下一章](#)

```
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。

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
```

```
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
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。

它会返回五种植。

- 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)  // 1.2599210498948734
```

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()

`Math.clz32()` 方法将参数转为 32 位无符号整数的形式，然后这个 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 `0b111101000`，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(1.000000123) // 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()

[上一章](#)

[下一章](#)

`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. 指数运算符

ES2016 新增了一个指数运算符（`**`）。

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

这个运算符的一个特点是右结合，而不是常见的左结合。多个指数运算符连用时，是从最右边开始计算的。

```
// 相当于 2 ** (3 ** 2)  
2 ** 3 ** 2  
// 512
```

上面代码中，首先计算的是第二个指数运算符，而不是第一个。


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，V8 引擎的指数运算符与 `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。



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 


Name

 **Heekei Zhuang** • 8 months ago

Math.trunc 的 polyfill:

```
~~4.1 === 4 // true
~~3.9 === 3 // true
~~-3.9 === -3 // true
.....
```

2 ^ | ▾ • Reply • Share ›

 **evantre** • 20 days ago

Math.clz32 这里 JavaScript 的整数使用 32 位二进制形式表示，这个说法不对啊，什么数都是 ieee754 啊，这个是先要转换成 32-bit unsigned integer

^ | ▾ • Reply • Share ›

 **ruanyf** Mod ➔ **evantre** • 16 days ago

谢谢指出，已经改正。


^ | ▾ • Reply • Share ›

 于振楠 • a month ago

“注意，V8 引擎的指数运算符与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。”

这个差异具体产生原因是什么呢？

^ | ▾ • Reply • Share ›

 **YX CHEN** • 2 months ago

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

// RangeError: Operation cannot be trusted!

trusty(1, 2, 3)

// 3

请问下 为什么要这样写是为什么解决 `isSafeInteger`的什么问题呢

^ | v • Reply • Share ›



Thom Liu • a year ago

至于 `3.000000000000000002` 问题，明明 `3.000000000000000002 === 3`（这个与 `NaN` 问题并不相同）。IEEE 754 并不能表示所有十进制小数。

^ | v • Reply • Share ›



郭敬文 • a year ago

`Math.trunc([]) // 0`

`Math.trunc([3]) // 3`

`Math.trunc(['-3']) // -3`

`Math.trunc(['- 3']) // NaN`

`Math.trunc(['- 3', 1]) //NaN`

关于类型转换的例子可以举两个例子

^ | v • Reply • Share ›



Nucky Thompson • a year ago

我的天，真正常用的能有几个..

^ | v • Reply • Share ›



Zoe Zhang • a year ago

那个，`Integer`有几个写成了`Interger`，会被修改嘛。。。。。

^ | v • Reply • Share ›



ruanyf Mod ➔ **Zoe Zhang** • a year ago

谢谢指出，已经更正了。

^ | v • Reply • Share ›



guoxing cao • 3 years ago

Polyfill:

```
Math.hypot = Math.hypot || function() {
```

```
  var y = 0;
```

```
  var length = arguments.length;
```

```
  for (var i = 0; i < length; i++) {
```

```
    if (arguments[i] === Infinity || arguments[i] === -Infinity) {
```

```
      return Infinity;
```

```
    }
```

```
    y += arguments[i] * arguments[i];
```

```
  }
```

```
  return Math.sqrt(y);
```

```
};
```

^ | v • Reply • Share ›

夏虫 • 3 years ago

trunc 和 floor 有什么区别?

^ | v • Reply • Share ›



Youxiang Lu ➔ 夏虫 • 3 years ago

作者不是写了么

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

^ | v • Reply • Share ›



Rocky Ren ➔ 夏虫 • 3 years ago

let a = -4.1;

Math.trunc(a); // => -4

Math.floor(a); // => -5

^ | v • Reply • Share ›



yacent ➔ Rocky Ren • 2 years ago

你没看 $x < 0$ 是 Math.ceil(x) 吗

^ | v • Reply • Share ›



张宇亮 • 3 years ago

Math.trunc()和Number.parseInt()有什么区别呢

^ | v • Reply • Share ›



ruanyf Mod ➔ 张宇亮 • 3 years ago

parseFloat('12.3a')

// 12.3

Math.trunc('12.3a')

// NaN

^ | v • Reply • Share ›



Jun Lang • 3 years ago

mark

^ | v • Reply • Share ›



xcatliu • 3 years ago

Math.fround(1.337) 为什么是 1.3370000123977661 呢?

^ | v • Reply • Share ›



Youxiang Lu ➔ xcatliu • 3 years ago

js的浮点数是按照 IEEE 754 standard来设计的（参考<https://en.wikipedia.org/wi...>

所以通过这种格式编码，对于小数，只能精确表示 0.5 ($\text{Math.pow}(2, -1)$), 0.25 ($\text{Math.pow}(2, -2)$), 0.125 ($\text{Math.pow}(2, -3)$)...

或者 $0.5+0.25$, $0.5+0.125$ 这种类型

对于你说的1.337，其小数部分^{上一章}为0.337，他^{下一章}可以表示为 2^N 次方或者 2^N 次方相加得到。

大致 $0.337 = 0.25 + 0.0625 + 0.015625 + \dots$

如果你大学是读过计算机系，那么问这个问题就很不应该了

^ | v • Reply • Share ›



xcatliu → Youxiang Lu • 3 years ago

感谢指导！身为计算机系毕业的自惭形秽

^ | v • Reply • Share ›



Chenghao Lan • 3 years ago

.....做银行、保险等金融类系统的小伙伴 升级到ES6 估计要好好检查数据接口了.....

^ | v • Reply • Share ›



Youxiang Lu → Chenghao Lan • 3 years ago

之前前端做计算，如果数据不是很大，一般都是把浮点转成整数然后计算。
比如 $0.2 * 0.2$ 转成 $2 * 2$ 计算。

这也是你会经常看到美国支付接口，都是以每分（cent）作为单位，而不是以美元。

1 ^ | v • Reply • Share ›



LeeJunhui → Youxiang Lu • 2 years ago

厉害

^ | v • Reply • Share ›



AaronJin • 4 years ago

Math应该是漏了sinh吧

^ | v • Reply • Share ›



dou4cc • 4 years ago

Math.trunc(n)在n不是很大时等同 $n < 0$ ，用哪个更好一点呢？

^ | v • Reply • Share ›



Thom Liu • a year ago

这个传统的 isNaN 写得不太好啊。我一般使用 $a !== a$ 来判断是不是 NaN。

^ | v • Reply • Share ›



Frank Fang • 5 years ago

Number.isInteger(25.0) // true 将会是一个槽点。

^ | v • Reply • Share ›



Andre → Frank Fang • 3 years ago

这个槽点终将会被淹没于js众多的槽点之中，不值一提

1 ^ | v • Reply • Share ›



杨小 → Frank Fang • 8 months ago

只能说不能怪我们

^ | v • Reply • Share ›



onweer → Frank Fang • 2 years ago

js没有变量类型... 存储方式一样的 ..

^ | v • Reply • Share ›



feipigzi → Frank Fang • 3 years ago

按数学的定义来说，它就是整数！

^ | v • Reply • Share ›



曾坤鹏 → Frank Fang • 4 years ago

不是说了js里面整数和浮点数存储方式一样，那肯定是算是整数啊

^ | v • Reply • Share ›



bumfo → Frank Fang • 4 years ago

25.0 確實是一個整數啊。。。只是在某些語言中被用來作為浮點數的標記。

^ | v • Reply • Share ›



Frank Fang → bumfo • 4 years ago

对于C系语言的程序员来说 25.0 和 25 绝对不一样啊。

^ | v • Reply • Share ›



paladintyrion → Frank Fang • 4 years ago

js一直是这样啊。javascript一直都没有int、float、double之分，只有Number基础类型。能解析成int、float已经算不错了。。。

^ | v • Reply • Share ›

ALSO ON ECMASCRIPT 6 入门

Class 的继承

38 comments • 2 years ago

Francie Who —

最新提案

6 comments • a year ago

hao wangg — 觉得链判断运算符很好用

修饰器

43 comments • 3 years ago

bpceee — 方法的修饰，target 是 Class 的 prototype，并不是实例。这个时候实例还未创建出来。

Promise对象

106 comments • 5 years ago

冯飞林 — 谢谢

