数值的扩展

1. 二进制和八进制表示法

二进制（binary）和八进制（octonary）数值的新的写法，分别用前缀 `0b`（或 `0B`）和 `0o`（或 `0O`）表示。十六进制用前缀 `0x` 表示：

```
0b10011 === 19 // true, (1 * 2 ** 4) + (1 * 2 ** 1) + (1 * 2 ** 0 )
0o23 === 19 // true, (2 * 8 ** 1) + (3 * 8 ** 0)
0b10011 === 0o23 // true

0x13 === 19 // 前缀 0x 表示十六进制
```

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

```
Number('0b111');  // 或 Number('0b111', 10),  7
Number('0o10');  // 或 Number('0o10', 10),  8
Number('0x18'); // 或 Number('0x18', 10), 24
```

如果要将十进制数转换为二进制、八进制、十六进制，需要使用 `toString()` 方法：

```
(19).toString(2)  // '10010'
(19).toString(8)  // '23'
(19).toString(16)  // '13'
```

2. 数值分隔符

数值使用下划线（_）作为分隔符。

```
1_000 === 1000; // true
```

数值分隔符没有指定间隔的位数，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...甚至可以随意位数间隔：

```
1_2_3_4 === 1234; // true, 每一位添加分隔符
12_34_56 === 123456; // true, 每二位添加分隔符
123_456_789 === 123456789; // true, 每三位添加分隔符
1234_5678_9000 === 123456789000; // true, 每四位添加分隔符
1_23_456_789 === 123456789; // true, 随意位数间隔添加分隔符
```

数值分隔符不允许的写法：

> - 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。

```
// 下面写法全部报错
_1464301 // 分隔符不能放在数值的最前面
1464301_ // 分隔符不能放在数值的最后面
123__456 // Only one underscore is allowed as numeric separator
3_.141 // 小数点前面不能有分隔符
3._141 // 小数点后面不能有分隔符
1_e12 // e 前面不能有分隔符
1e_12 // e 后面不能有分隔符
```

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。

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。 下面三个将字符串转成数值的函数，不支持数值分隔符：

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

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

3. 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
Math.pow(2, 53) === Math.pow(2, 53) + 1; // true

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

上面代码中，超出 2 的 53 次方之后，一个数就不精确了。

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

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

`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
```

`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`。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
```