



第27课：ES6进阶之数据类型拓展

主讲老师：万章



四知

字符串的模板拓展

字符串的方法拓展

数字格式的方法拓展

解构赋值的注意点辨析



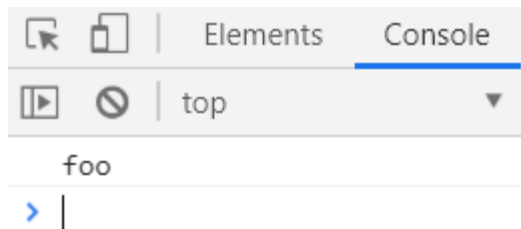
字符串拓展

字符串拓展之字符串遍历

```
for (let codePoint of 'foo') {  
  console.log(codePoint);  
  // "f"  
  // "o"  
  // "o"  
}
```

ES6 为字符串添加了遍历器接口,使得字符串可以被for...of循环遍历。
这个和字符串的方法类似,当字符串被用来遍历的时候,字符串临时变成了一个数组,当完成遍历之后,这个数组就会被销毁

```
let s="foo"  
  
for (let codePoint of s) {  
  codePoint="h";  
}  
  
console.log(s);
```



和其他的封装对象类似,基础类型的封装对象只给我们提供的读取的接口,我们无法通过遍历的方式改变字符串的字符串

字符串拓展之模板字符串

在ajax的时代，我们经常需要把后端传输到前端的JSON数据用js生成HTML标签结构，为了能够快速生成和添加，我们会使用字符串拼接的方式来进行操作

```
obj.innerHTML =  
    'There are <b>' + value1 + '</b> ' +  
    'items in your basket, ' +  
    '<em>' + value2 +  
    '</em> are on sale!';
```

上面这种写法相当繁琐不方便，ES6 引入了模板字符串解决这个问题。

字符串拓展之模板字符串

模板字符串的格式大致如下

``abvsadf${变量}aadasda``

模板字符串 (template string) 是增强版的字符串，用反引号 (```) 标识(这个符号在你的键盘tab键的上面，需要在英文输入法下输入)。它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。

```
let value1=1;
let value2=2;

obj.innerHTML =
`
  There are <b>${value1}</b> items
  in your basket, <em>${value2}</em>
  are on sale!
`
```

```
<div class="demo">
  "
    There are "
    <b>1</b>
    " items
    in your basket, "
    <em>2</em>
    "
    are on sale!
  "
</div>
```

如果使用模板字符串表示多行字符串，所有的空格和缩进都会被保留在输出之中。

abvsadf\${变量}aadasda`

```
let x = 1;
let y = 2;

`${x} + ${y} = ${x + y}`
// "1 + 2 = 3"

`${x} + ${y * 2} = ${x + y * 2}`
// "1 + 4 = 5"

let obj = {
  x: 1,
  y: 2
};

`${obj.x + obj.y}`
// "3"
```

大括号内部可以放入任意的 JavaScript 表达式，可以进行运算，以及引用对象属

```
function fn() {
  return "Hello World";
}

`foo ${fn()} bar`
// foo Hello World bar
```

模板字符串之中还能调用函数

如果大括号中的值不是字符串，将按照一般的规则转为字符串。比如，大括号中是一个对象，将默认调用对象的toString方法。

字符串拓展之模板字符串

`abvsadf\${变量}aadasda`

```
// 变量place没有声明  
let msg = `Hello, ${place}`;  
// 报错
```

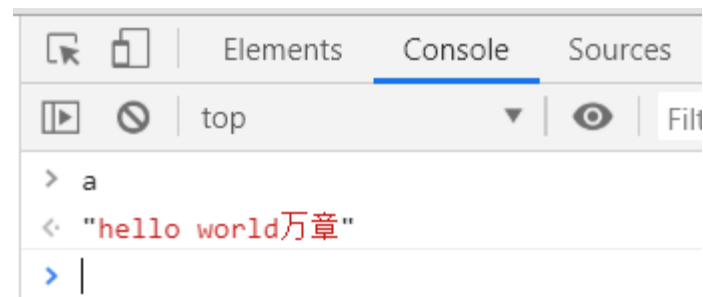
如果模板字符串中的变量没有声明，将报错。

```
`Hello ${'World'}`  
// "Hello World"
```

由于模板字符串的大括号内部，就是执行 JavaScript 代码，因此如果大括号内部是一个字符串，将会原样输出。

字符串拓展之模板字符串

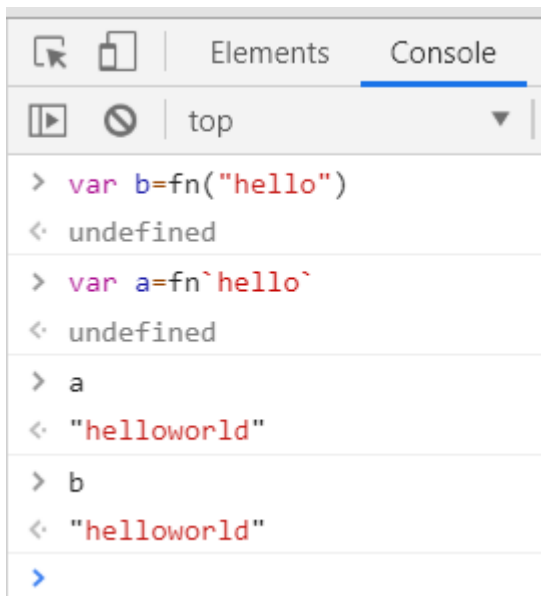
```
function fn(s) {  
    return s;  
}  
  
let a = `${"hello world" + `${fn("万章")}`}`;
```



模板字符串还能嵌套，可以在一个模板字符串中间再嵌套一个模板字符串

字符串拓展之模板字符串

模板字符串的功能，不仅仅是上面这些。它可以紧跟在一个函数名后面，该函数将被调用来处理这个模板字符串。这被称为“标签模板”功能（tagged template）。



```
< Elements Console
top
> var b=fn("hello")
< undefined
> var a=fn`hello`
< undefined
> a
< "helloworld"
> b
< "helloworld"
>
```



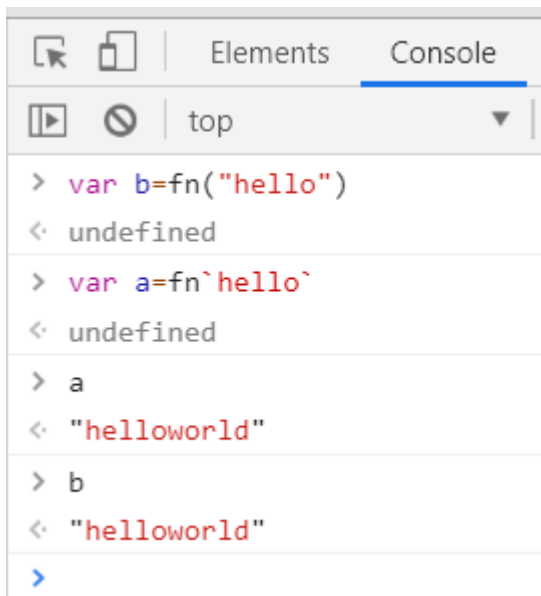
```
function fn(s) {
  return s+"world";
}

fn`hello`; // 返回helloworld
fn("hello"); // 返回helloworld
```

标签模板其实不是模板，而是函数调用的一种特殊形式。“标签”指的就是函数，紧跟在后面的模板字符串就是它的参数

字符串拓展之模板字符串

模板字符串的功能，不仅仅是上面这些。它可以紧跟在一个函数名后面，该函数将被调用来处理这个模板字符串。这被称为“标签模板”功能（tagged template）。



```
< Elements Console
top
> var b=fn("hello")
< undefined
> var a=fn`hello`
< undefined
> a
< "helloworld"
> b
< "helloworld"
>
```



```
function fn(s) {
    return s+"world";
}

fn`hello`; // 返回helloworld
fn("hello"); // 返回helloworld
```

标签模板其实不是模板，而是函数调用的一种特殊形式。“标签”指的就是函数，紧跟在后面的模板字符串就是它的参数

字符串拓展之模板字符串

但是，如果模板字符串里面有变量，就不是简单的调用了，而是会将模板字符串先处理成多个参数，再调用函数。

```
let a = 5;  
let b = 10;  
  
tag`Hello ${ a + b } world ${ a * b }`;  
// 等同于  
tag(['Hello ', ' world ', ''], 15, 50);
```

上面代码中，模板字符串前面有一个标识名tag，它是一个函数。整个表达式的返回值，就是tag函数处理模板字符串后的返回值。

函数tag依次会接收到多个参数。

tag函数的第一个参数是一个数组，该数组的成员是模板字符串中那些没有变量替换的部分；

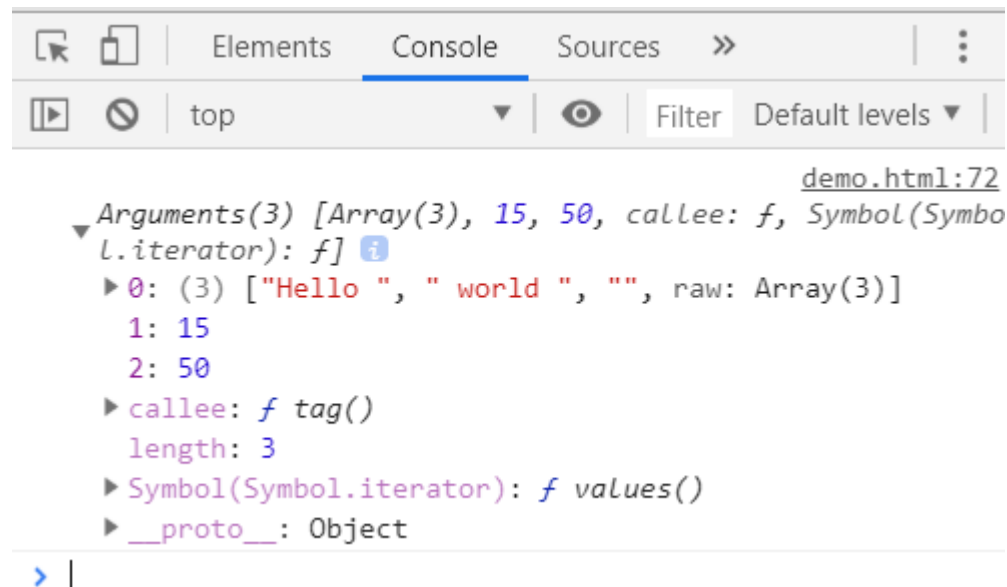
tag函数的其他参数，都是模板字符串各个变量被替换后的值。由于本例中，模板字符串含有两个变量，因此tag会接受到value1和value2两个参数。

字符串拓展之模板字符串

```
let a=5;
let b=10;

function tag(){
  console.log(arguments);
}

tag`Hello ${ a + b } world ${ a * b }`;
```



在提取字符串的时候，不要忘记空格也是字符串所以
“hello ”是单词hello和后面的一个空格组成的
“ world ”是单词world和前后各一个空格组成的

The diagram shows the template string `tag`Hello ${ a + b } world ${ a * b }`;` with red dotted arrows pointing to the corresponding elements in the array `tag(["hello ", " world ", ""], 15, 50)`. The arrows indicate that the string before the first interpolation is "hello ", the string between the two interpolations is " world ", and the string after the second interpolation is an empty string. A red triangle points to the closing backtick of the template string.

```
tag`Hello ${ a + b } world ${ a * b }`;
```

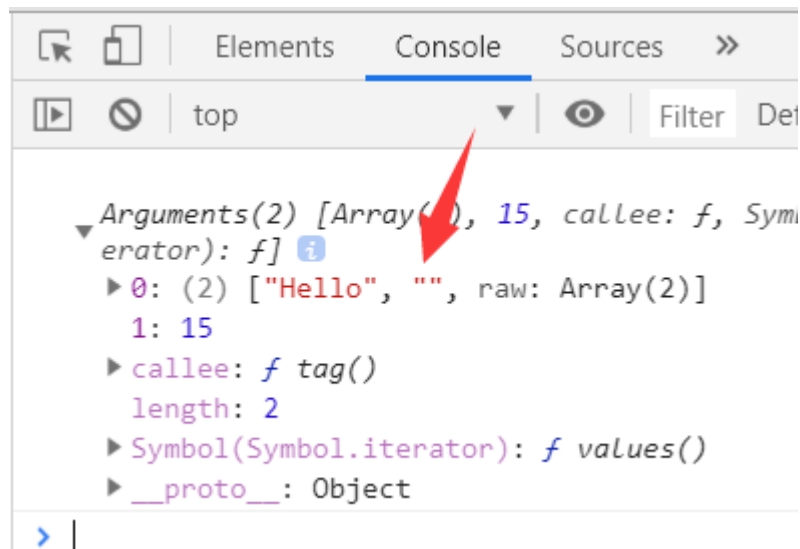
```
tag(["hello ", " world ", ""], 15, 50)
```

当变量是第一个或是最后一个时,会在第一个前方或是最后一个的后方加入一个空字符串

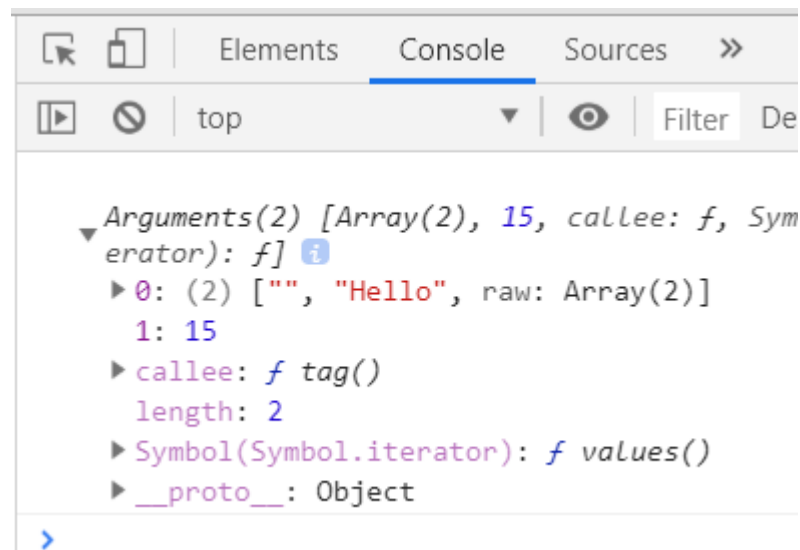
字符串拓展之模板字符串

当变量是第一个或是最后一个时,会在第一个前方或是最后一个的后方加入一个空字符串

```
tag`Hello${ a + b }`;
```



```
tag`${ a + b }Hello`;
```





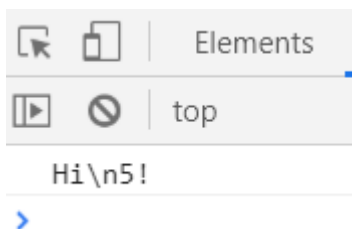
字符串的方法拓展

字符串的方法拓展之raw()

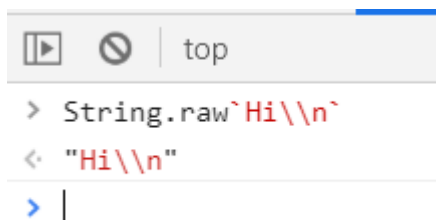
ES6 为原生的 String 对象, 提供了一个raw()方法。该方法返回一个斜杠都被转义 (即斜杠前面再加一个斜杠) 的字符串, 往往用于模板字符串的处理方法

```
> `Hi\n${2+3}!`  
< "Hi  
5!"  
> |
```

```
String.raw`Hi\n${2+3}!`;
```



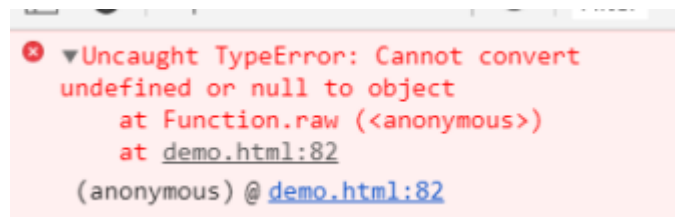
```
Elements  
top  
Hi\n5!  
>
```



```
top  
> String.raw`Hi\\n`  
< "Hi\\n"  
> |
```

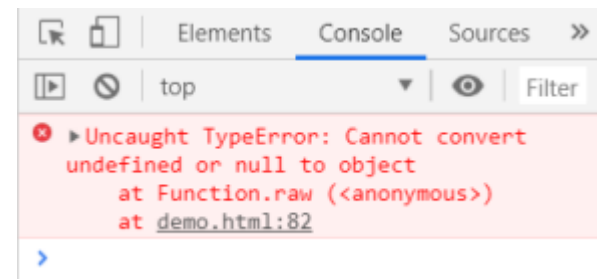
如果原字符串的斜杠已经转义, 那么就
直接输出该模板字符串

```
console.log(String.raw("Hi\n5"));
```



```
Uncaught TypeError: Cannot convert  
undefined or null to object  
at Function.raw (<anonymous>)  
at demo.html:82  
(anonymous) @ demo.html:82
```

```
console.log(String.raw("Hi5"));
```



```
Elements Console Sources  
top  
Uncaught TypeError: Cannot convert  
undefined or null to object  
at Function.raw (<anonymous>)  
at demo.html:82  
>
```

.raw方法不能直接往里面传入普通
字符串的, 否则是会直接报错的

字符串的方法拓展之raw()

String.raw()方法也可以作为正常的函数使用。这时，它的第一个参数，应该是一个具有raw属性的对象，且raw属性的值应该是一个数组。

```
String.raw({ raw: 'test' }, 0, 1, 2); // 't0e1s2t'  
// 等同于  
String.raw({ raw: ['t','e','s','t'] }, 0, 1, 2);
```

字符串的方法拓展之includes(), startsWith(), endsWith()

传统上, JavaScript 只有indexOf方法, 可以用来确定一个字符串是否包含在另一个字符串中。

ES6 又提供了三种新方法。

1. includes(): 返回布尔值, 表示是否找到了参数字符串。
2. startsWith(): 返回布尔值, 表示参数字符串是否在原字符串的头部。
3. endsWith(): 返回布尔值, 表示参数字符串是否在原字符串的尾部。

```
let s = 'Hello world!';  
  
s.startsWith('Hello') // true  
s.endsWith('!') // true  
s.includes('o') // true
```

```
let s = 'Hello world!';  
  
s.startsWith('world', 6) // true  
s.endsWith('Hello', 5) // true  
s.includes('Hello', 6) // false
```

这三个方法都支持第二个参数, 表示开始搜索的位置。

使用第二个参数n时, endsWith的行为与其他两个方法有所不同。它针对前n个字符, 而其他两个方法针对从第n个位置直到字符串结束。

字符串的方法拓展之字符串重复repeat()

```
'x'.repeat(3) // "xxx"  
'hello'.repeat(2) // "hellohello"  
'na'.repeat(0) // ""
```

repeat方法返回一个新字符串，表示将原字符串重复n次。

```
'na'.repeat(-0.9) // ""
```

如果参数是 0 到-1 之间的小数，则等同于 0，这是因为会先进行取整运算。0 到-1 之间的小数，取整以后等于-0，repeat视同为 0。

```
'na'.repeat(2.9) // "nana"
```

```
'na'.repeat(Infinity)  
// RangeError  
'na'.repeat(-1)  
// RangeError
```

参数如果是小数，会被取整(向下取整)。
如果repeat的参数是负数或者Infinity，会报错。

```
'na'.repeat(NaN) // ""
```

```
'na'.repeat('na') // ""  
'na'.repeat('3') // "nanana"
```

参数NaN等同于 0。
如果repeat的参数是字符串，则会先转换成数字

字符串的方法拓展之字符串补全padStart(), padEnd()

ES2017 引入了字符串补全长度的功能。如果某个字符串不够指定长度，会在头部或尾部补全。

```
'x'.padStart(5, 'ab') // 'ababx'  
'x'.padStart(4, 'ab') // 'abax'  
  
'x'.padEnd(5, 'ab') // 'xabab'  
'x'.padEnd(4, 'ab') // 'xaba'
```

padStart()用于头部补全，padEnd()用于尾部补全。

padStart()和padEnd()一共接受两个参数，**第一个参数是字符串补全生效的最大长度，第二个参数是用来补全的字符串。**

```
'xxx'.padStart(2, 'ab') // 'xxx'  
'xxx'.padEnd(2, 'ab') // 'xxx'
```

如果原字符串的长度，等于或大于最大长度，则字符串补全不生效，返回原字符串。

```
'abc'.padStart(10, '0123456789')  
// '0123456abc'
```

如果用来补全的字符串与原字符串，两者的长度之和超过了最大长度，则会截去超出位数的补全字符串。

```
'x'.padStart(4) // '   x'  
'x'.padEnd(4) // 'x   '
```

如果省略第二个参数，默认使用空格补全长度。

字符串的方法拓展之字符串空格消除trimStart(),trimEnd()

ES2019 对字符串实例新增了trimStart()和trimEnd()这两个方法。它们的行为与trim()一致, trimStart()消除字符串头部的空格, trimEnd()消除尾部的空格。

它们返回的都是新字符串, 不会修改原始字符串。

```
const s = '  abc  ';  
  
s.trim() // "abc"  
s.trimStart() // "abc  "  
s.trimEnd() // "  abc"
```

trimStart()只消除头部的空格, 保留尾部的空格。trimEnd()也是类似行为。

除了空格键, 这两个方法对字符串头部 (或尾部) 的 tab 键、换行符等不可见的空白符号也有效。

浏览器还部署了额外的两个方法, trimLeft()是trimStart()的别名, trimRight()是trimEnd()的别名。



数字格式的方法拓展

数字格式的方法拓展

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.isNaN()用来检查一个值是否为NaN。

```
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(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。

数字格式的方法拓展

`Number.isInteger()`用来判断一个数值是否为整数。

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

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

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

如果参数不是数值，`Number.isInteger`返回false。

数字格式的方法拓展

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.isSafeInteger()`则是用来判断一个整数是否落在这个范围之内

```
top
> Number.MAX_SAFE_INTEGER
< 9007199254740991
> Number.MIN_SAFE_INTEGER
< -9007199254740991
>
```

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

数字格式的方法拓展

`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
```

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

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

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

对于空值和无法截取整数的值，返回NaN。

数字格式的方法拓展

`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('') // 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
```

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。

数字格式的高级数学拓展

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

Math.cbrt方法用于计算一个数的立方根。
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。

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

ES2016 新增了一个指数运算符 (**)。
多个指数运算符连用时，是从最右边开始计算的。

```
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
```

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

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

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

指数运算符可以与等号结合
形成一个新的赋值运算符 (**=)。