数组的扩展

1. 扩展运算符

扩展运算符（spread）是三个点（...）。好比 rest 的逆运算，**将一个数组转为用逗号分隔的参数序列。**

```
console.log(...[1, 2, 3]);             // 1 2 3
console.log(1, ...[2, 3, 4], 5);       // 1 2 3 4 5
[...document.querySelectorAll('div')]; // [<div>, <div>, <div>]
```

该运算符主要用于函数调用。

```
function push(array, ...items) {
  array.push(...items); // 数组 push 方法可以一次添加多个值，用逗号隔开
}

function add(x, y) {
  return x + y;
}
const numbers = [4, 38];
add(...numbers); // 42, 相当于: add(4, 38)
```

扩展运算符与正常的函数参数可以结合使用：

```
function f(v, w, x, y, z) { }
const args = [0, 1];
f(-1, ...args, 2, ...[3]); // 相当于: f(-1, 0, 1, 2, 3)
```

如果扩展运算符后面是一个空数组，则不产生任何效果。

```
[...[], 1]; // [1]
```

只有函数调用时，扩展运算符才可以放在圆括号中，否则会报错。

```
(...[1, 2]);              // Uncaught SyntaxError: Unexpected token '...'
console.log((...[1, 2])); // Uncaught SyntaxError: Unexpected token '...'
console.log(...[1, 2]);   // 1 2
```

1.1. 替代函数的 apply() 方法

由于扩展运算符可以展开数组，所以不再需要 `apply()` 方法将数组转为函数的参数了。

```
// ES5 的写法
function f(x, y, z) { /**/ }
var args = [0, 1, 2];
f.apply(null, args);

// ES6 的写法
function f(x, y, z) { /**/ }
let args = [0, 1, 2];
f(...args);
```

应用 `Math.max()` 方法，简化求出一个数组最大元素：

```
Math.max.apply(null, [14, 3, 77]) // ES5 的写法
Math.max(...[14, 3, 77])  // ES6 的写法
// 等同于
Math.max(14, 3, 77); // Math.max() 不能直接求数组的最大值，只能求一组序列数字的最大值
```

将一个数组添加到另一个数组的尾部：

```
// ES5 的写法
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
Array.prototype.push.apply(arr1, arr2);

// ES6 的写法
let arr1 = [0, 1, 2];
let arr2 = [3, 4, 5];
arr1.push(...arr2);
```

1.2. 扩展运算符的应用

1.2.1. 复制数组

数组是复合的数据类型，直接复制的话，只是复制了指向底层数据结构的指针，而不是克隆一个全新的数组。

```
const a1 = [1, 2];
const a2 = a1;

a2[0] = 2;
a1 // [2, 2]
```

`a2` 和 `a1` 指向同一份数据的另一个指针。修改 `a2`，会直接导致 `a1` 的变化。

ES5 只能用变通方法来复制数组。

```
const a1 = [1, 2];
const a2 = a1.concat(); // concat 返回一个新数组，a2 是一个新数组，和 a1 无关

a2[0] = 2;
a1 // [1, 2]
```

a1 会返回原数组的克隆，再修改 a2 就不会对 a1 产生影响。

扩展运算符提供了复制数组的简便写法。

```
const a1 = [1, 2];
// 下面的两种写法，`a2` 都是 `a1` 的克隆。
const a2 = [...a1]; // 写法一
const [...a2] = a1; // 写法二
```

1.2.2. 合并数组

扩展运算符提供了数组合并的新写法。

```
const arr1 = ['a', 'b'];
const arr2 = ['c'];
const arr3 = ['d', 'e'];

// ES5 的合并数组
arr1.concat(arr2, arr3); // [ 'a', 'b', 'c', 'd', 'e' ]

// ES6 的合并数组
[...arr1, ...arr2, ...arr3]; // [ 'a', 'b', 'c', 'd', 'e' ]
```

1.2.3. 与解构赋值结合

扩展运算符可以与解构赋值结合起来，用于生成数组。

```
a = list[0], rest = list.slice(1) // ES5

[a, ...rest] = list  // ES6
```

1.2.4. 字符串

扩展运算符还可以将字符串转为真正的数组。

```
[...'hello'] // [ "h", "e", "l", "l", "o" ]
```

1.2.5. 实现了 Iterator 接口的对象

任何定义了遍历器（Iterator）接口的对象，都可以用扩展运算符转为真正的数组。

```
let nodeList = document.querySelectorAll('div');
let array = [...nodeList];
```

`querySelectorAll()` 方法返回的是一个 `NodeList` 对象，是一个类似数组的对象。扩展运算符可以将其转为真正的数组，原因就在于NodeList对象实现了 Iterator。

对于那些没有部署 Iterator 接口的类似数组的对象，扩展运算符就无法将其转为真正的数组。

```
let arrayLike = {
  '0': 'a',
  '1': 'b',
  '2': 'c',
  length: 3
};
let arr = [...arrayLike]; // arrayLike is not iterable
```

arrayLike 是一个类似数组的对象，但是没有部署 Iterator 接口，扩展运算符就会报错。这时，可以改为使用 `Array.from` 方法将 `arrayLike` 转为真正的数组。

1.2.6. Map 和 Set 结构

扩展运算符内部调用的是数据结构的 Iterator 接口，因此只要具有 Iterator 接口的对象，都可以使用扩展运算符，比如 Map 结构。

let map = new Map([[1, 'one'], [2, 'two'], [3, 'three'],]);

如果对没有 Iterator 接口的对象，使用扩展运算符，将会报错。

```
const obj = {a: 1, b: 2};
let arr = [...obj]; // TypeError: Cannot spread non-iterable
```

2. Array.from()

`Array.from()` 方法用于将两类对象转为真正的数组：类似数组的对象（array-like object）和可遍历（iterable）的对象（包括 ES6 新增的数据结构 Set 和 Map）。

下面是一个类似数组的对象，Array.from()将它转为真正的数组。

```
let arrayLike = {
    '0': 'a',
    '1': 'b',
    '2': 'c',
    length: 3
};

// ES5 的写法
var arr1 = [].slice.call(arrayLike); // ['a', 'b', 'c']
var arr1 = Array.prototype.slice.call(arrayLike); // ['a', 'b', 'c']
Array.prototype === [].__proto__ // true, 类的显示原型指向实例的隐士原型

// ES6 的写法
let arr2 = Array.from(arrayLike); // ['a', 'b', 'c']
```

字符串、NodeList、arguments、定义了 `length` 的对象，Array.from()都可以将它们转为真正的数组。

只要是部署了 Iterator 接口的数据结构，`Array.from()` 都能将其转为数组。

```
Array.from('hello') // ['h', 'e', 'l', 'l', 'o']

let namesSet = new Set(['a', 'b'])
Array.from(namesSet) // ['a', 'b']
```

字符串和 Set 结构都具有 Iterator 接口，因此可以被Array.from()转为真正的数组。

如果参数是一个真正的数组，Array.from()会返回一个一模一样的新数组。

```
Array.from([1, 2, 3]) // [1, 2, 3]
```

扩展运算符背后调用的是遍历器接口（Symbol.iterator），如果一个对象没有部署这个接口，就无法转换。Array.from() 方法还支持类似数组的对象。