**第7章 数组的扩展**

**7.1 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）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。

1. let arrayLike = {

2.     "0": "a",

3.     "1": "b",

4.     "2": "3",

5.     "length": 3

6. };

7.

8. // ES5的写法

9. var arr1 = [].slice.call(arrayLike); // ["a", "b", "c"]

10.

11. // ES6的写法

12. let arr2 = Array.from(arrayLike); // ["a", "b", "c"]

13.

实际应用中，常见的类似数组的对象是DOM操作返回的NodeList集合，以及函数内部的arguments对象。Array.from都可以将它们转为真正的数组。

1. // NodeList对象

2. let ps = document.querySelectorAll("p");

3. Array.from(ps).forEach(function (p) {

4.     console.log(p);

5. });

6.

7. // arguments对象

8. function foo() {

9.     var args = Array.from(arguments);

10.     // ...

11. }

12.

上面的代码中，querySelectorAll方法返回的是一个类似数组的对象，只有将这个对象转为真正的数组，才能使用forEach方法。

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

1. Array.from("hello");

2. // ["h", "e", "l", "l", "o"]

3.

4. let nameSet = new Set(["a", "b"])

5. Array.from(nameSet); // ["a", "b"]

6.

上面的代码中，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会返回一个一摸一样的新数组。

1. Array.from([1, 2, 3]);

2. // [1, 2, 3]

3.

值得提醒的是，扩展运算符（…）也可以将某些数据结构转为数组。

1. // arguments对象

2. function foo() {

3.     var args = [...arguments];

4. }

5.

6. // NodeList对象

7. [...document.querySelectorAll("div")]

8.

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该接口，就无法转换。Array.from方法还支持类似数组的对象。所谓类似数组的对象，本质特征只有一点，即必须有length属性。因此，任何有length属性的对象，都可以通过Array.from方法转为数组，而这种情况扩展运算符无法转换。

1. Array.from({ length: 3 });

2. // [undefined, undefined, undefined]

3.

上面的代码中，Array.from返回了一个具有3个成员的数组，每个位置的值都是undefined。扩展运算符转换不了这个对象。

对于没有部署该方法的浏览器，可以用Array.prototype.slice方法替代。

1. const toArray = (() => {

2.     Array.from ? Array.from : obj => [].slice.call(obj)

3. })();

4.

Array.from还可以接受第二个参数，作用类似于数组的map方法，用来对每个元素进行处理，将处理后的值放入返回的数组。

1. Array.from(arrayLike, x => x \* x);

2. // 等同于

3. Array.from(arrayLike).map(x => x \* x);

4.

5. Array.from([1, 2, 3], (x) => x \* x);

6. // [1, 4, 9]

7.

下面的例子取出了一组DOM节点的文本内容。

1. let spans = document.querySelectorAll("span.name");

2.

3. // map()

4. let names1 = Array.prototype.map.call(spans, s => s.textContent);

5.

6. // Array.from()

7. let names2 = Array.from(spans, s => s.textContent);

8.

下面的例子将数组中布尔值为false的成员转为0。

1. Array.from([1, , 2, , 3], (n) => n || 0);

2. // [1, 0, 2, 0, 3]

3.

另一个例子是返回各种数据的类型。

1. function typesOf() {

2.     return Array.from(arguments, value => typeof value);

3. }

4.

5. typesOf(null, [], NaN);

6. // ["object", "object", "number"]

7.

如果map函数里面用到了this关键字，可以传入Array.from第三个参数，用来绑定this。Array.from()可以将各种值转为真正的数组，并且提供map功能。这实际上意味着，只要有一个原始的数据结构，就可以先对它的值进行处理，然后转成规范的数组结构，进而可以使用数量众多的数组方法。

1. Array.from({ length: 2 }, () => "jack");

2. // ["jack", "jack"]

3.

上面的代码中，Array.from的第一个参数指定了第二个参数运行的次数。这种特性可以让该方法的用法变得非常灵活。

Array.from()的另一个应用，将字符串转为数组，然后返回字符串的长度。因为它能正确处理各种Unicode字符，可以避免JavaScript将大于\uFFFF的Unicdoe字符算作2个字符的bug。

1. function countSymbols(string) {

2.     return Array.from(string).length;

3. }

4.

**7.2 Array.of()**

Array.of方法用于将一组值转换为数组。

1. Array.of(2, 11, 8); // [2, 11, 8]

2. Array.of(3); // [3]

3. Array.of(3).length; // 1

4.

这个方法的主要目的，是弥补数组构造函数Array()的不足。因为参数个数的不同会导致Array()的行为有差异。

1. Array(); // []

2. Array(3); // [,,,]

3. Array(3, 11, 8) // [3, 11, 8]

4.

上面的代码中，Array方法没有参数、有1一个参数或有3个参数时，返回会结果都不一样。只有当参数个数不少于2个时，Array()才会返回由参数组成的新数组。参数个数只有1个时，实际上是指定数组的长度。

Array.of基本上可以用来替代Array()或new Array()，并且不存在由于参数不同而导致的重载。它的行为非常统一。

1. Array.of(); // []

2. Array.of(undefined); // [undefined]

3. Array.of(1); // [1]

4. Array.of(1, 2); // [1, 3]

5.

Array.of总是返回参数值组成的数组。如果没有参数，就返回一个空数组。

Array.of方法可以用下面的代码模拟实现。

1. function ArrayOf() {

2.     return [].slice.call(arguments);

3. }

4.

**7.3 数组实例的copyWithin()**

数组实例的copyWithin方法，在当前数组内部将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组。也就是说，使用这个方法会修改当前数组。

1. Array.prototype.copyWithin(target, start = 0, end = this.length)

2.

它接受3个参数。

* target（必须）：从该位置开始替换数据。
* start（可选）：从该位置开始读取数据，默认为0。如果为负值，表示倒数。
* end（可选）：到该位置前停止读取数据，默认等于数组长度。如果为负值，表示倒数。

这3个参数都应该是数值，如果不是，会自动转为数值。

1. [1,2,3,4,5].copyWithin(0, 3);

2. // [4,5,3,4,5]

3.

上面的代码表示将从3号位置直到数组结束的成员（4和5），复制到从0号位置开始的位置。结果覆盖了原来的1和2。

下面是更多例子。

1. // 将3号位置复制到0号位置

2. [1,2,3,4,5].copyWithin(0, 3, 4);

3. // [4,2,3,4,5]

4.

5. // -2相当于3号位置，-1相当于4号位置

6. [1,2,3,4,5].copyWithin(0, -2, -1)

7. // [4,2,3,4,5]

8.

9. // 将3号位置复制到0号位置

10. [].copyWithin.call({length: 5, 3: 1}, 0, 3)

11. // { '0': 1, '3': 1, length: 5 }

12.

13. // 将2号位置到数组结束，复制到0号位置

14. var i32a = new Int32Array([1,2,3,4,5]);

15. i32a.copyWithin(0, 2);

16. // Int32Array [3,4,5,4,5]

17.

18. // 对于没有部署TypedArray的copyWithin方法的平台

19. // 需要采用下面的写法

20. [].copyWithin.call(new Int32Array([1,2,3,4,5]), 0, 3, 4);

21. // Int32Array [4,2,3,4,5]

22.

**7.4 数组实例的find()和findIndex()**

数组实例的find方法，用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为true的成员，然后返回该成员。如果没有符合条件的成员，则返回undefined。

1. [1, 4, -5, 10].find((n) => n < 0);

2. // -5

3.

上面的代码找出数组中第一个小于0的成员。

1. [1, 5, 10, 15].find(function (value, index, arr) {

2.     return value > 9;

3. }); // 10

4.

上面的代码中，find方法的回调函数可以接受3个参数，依次为当前的值、当前的位置和原数组。

数组实例的findIndex方法的用法与find方法非常类似，返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回-1。

1. [1, 5, 10, 15].findIndex(function (value, index, arr) {

2.     return value > 9;

3. }); // 2

4.

这两个方法都可以接受第二个参数，用来绑定回调函数的this对象。

另外，这两个方法都可以发现NaN，弥补了数组的IndexOf方法的不足。

1. [NaN].indexOf(NaN);

2. // -1

3.

4. [NaN].findIndex(y => Object.is(NaN, y));

5. // 0

6.

上面的代码中，indexOf方法无法识别数组的NaN成员，但是findIndex方法可以借助Object.is方法做到。

**7.5 数组实例的fill()**

fill方法使用给定值填充数组。

1. [1, 2, 3].fill(7);

2. // [7, 7, 7]

3.

4. new Array(3).fill(7);

5. // [7, 7, 7]

6.

上面的代码表明，fill方法用于空数组的初始化非常方便。数组中已有的元素，会被全部抹去。fill方法还可以接受第二个和第三个参数，用于指定填充的起始位置和结束位置。

1. ["a", "b", "c"].fill(7, 1, 2);

2. // ["a", 7, "c"]

3.

上面的代码表示，fill方法从1号位置开始，向原数组填充7，到2号位置之前结束。

**7.6 数组实例的entries()、keys()和values()**

ES6提供了3个新方法---entries()、keys()和values()---用于遍历数组。它们都返回一个遍历器对象（详见第14章），可用for…of循环遍历，唯一的区别是：keys()是对键名的遍历，values()是对键值的遍历，entries()是对键值对的遍历。

1. for (let index of ["a", "b"].keys()) {

2.     console.log(index);

3. }

4. // 0

5. // 1

6.

7. for (let elem of ["a", "b"].values()) {

8.     console.log(elem);

9. }

10. // "a"

11. // "b"

12.

13. for (let [index, elem] of ["a", "b"].entries()) {

14.     console.log(index, elem);

15. }

16. // 0 "a"

17. // 1 "b"

18.

如果不使用for…of循环，可以手动调用遍历器对象的next方法进行遍历。

1. let letter = ["a", "b", "v"];

2. let entries = letter.entries();

3. console.log(entries.next().value); // [ 0, 'a' ]

4. console.log(entries.next().value); // [ 1, 'b' ]

5. console.log(entries.next().value); // [ 2, 'v' ]

6.

**7.7 数组实例的includes()**

Array.prototype.includes方法返回一个布尔值，表示某个数组是否包含给定的值，与字符串的includes方法类似。该方法属于ES7，但Babel转码器已经支持。

1. [1,2,3].includes(2); // true

2. [1,2,3].includes(4); // false

3. [1,2,NaN].includes(NaN); // true

4.

该方法的第二个参数表示搜索的起始位置，默认为0。如果第二个参数为负数，则表示倒数的位置，若此位置超出数组长度（比如第二个参数为-4，但数组长度为3），则会重置从0开始。

1. [1,2,3].includes(3, 3); // false

2. [1,2,3].includes(3, -1); // true

3.

没有该方法时，通常使用数组的indexOf方法检查是否包含某个值。

1. if (arr.indexOf(el) !== -1) {

2.     // ...

3. }

4.

indexOf方法有两个缺点：一是不够语义化，其含义是找到参数值的第一个出现位置，所以要比较是否不等于-1，表达起来不够直观；二是，其内部使用严格相当运算符（===）进行判断，会导致对NaN的误判。

1. [NaN].indexOf(NaN);

2. // -1

3.

includes使用的是不一样的判断算法，没有这个问题。

1. [NaN].includes(NaN);

2. // true

3.

下面的代码用来检查当前环境是否支持该方法，如果不支持，就部署一个简易的替代版本。

1. const contains = (() =>

2.     Array.prototype.includes ? (arr, value) => arr.includes(value)

3.         : (arr, value) => arr.some(el => el === value)

4. )();

5.

6. contains(["foo", "bar"], "baz"); // => false

7.

另外，Map和Set数据结构有一个has方法，需要注意与includes区分。

* Map结构的has方法是用来查找键名的，比如Map.prototype.has(key)、WeakMap.prototype.has(key)、Reflect.has(target, propertyKey)。
* Set结构的has方法是用来查找值的，比如Set.prototype.has(value)、WeakSet.prototype.has(value)。

**7.8 数组的空位**

数组的空位指数组的某一个位置没有任何值。比如，Array构造函数返回的数组都是空位。

1. Array(3); // [,,]

2.

上面的代码中，Array(3)返回一个具有3个空位的数组。

注意，空位不是undefined，一个位置的值等于undefined依然是有值的。空位是没有任何值，in运算符可以说明这一点。

1. 0 in [undefined, undefined, undefined] // true

2. 0 in [,,,] // false

3.

上面的代码说明，第一个数组的0号位置是有值的，第二个数组的0号位置没有值。

ES5对空位的处理很不一致，大多数情况下会忽略空位。

* forEach()、filter()、every()和some()都会跳过空位。
* map()会跳过空位，但会保留这个值。
* join()和toString()会将空位视为undefined，而undefined和null会被处理成空字符串。

1. // forEach方法

2. [,"a"].forEach((x, i) => console.log(i)); // 1

3.

4. // filter方法

5. ["a", , "b"].filter(x => true); // ["a", "b"]

6.

7. // every方法

8. [,"a"].every(x => x === "a"); // true

9.

10. // some方法

11. [,"a"].some(x => x !== "a"); // false

12.

13. // map方法

14. [,"a"].map(x => 1); // [, 1]

15.

16. // join方法

17. [, "a", undefined, null].join("#"); // "#a##"

18.

19. // toString方法

20. [, "a", undefined, null].toString(); // ",a,,"

21.

ES6则是明确将空位转为undefined。

Array.from方法会将数组的空位转为undefined。也就是说，这个方法不会忽略空位。

1. Array.from(["a", "b"]);

2. // ["a", undefined, "b"]

3.

扩展运算符（…）也会将空位转为undefined。

1. [...["a", , "b"]]

2. // ["a", undefined, "b"]

3.

copyWithin()会连空位一起复制。

1. [, "a", "b",,].copyWithin(2, 0); // [ <1 empty item>, 'a', <1 empty item>, 'a' ]

2.

fill()会将空位视为正常的数组位置。

1. new Array(3).fill('a'); // ["a", "a", "a"]

2.

for…of循环也会遍历空位。

1. let arr = [,,];

2. for (let i of arr) {

3.     console.log(1);

4. }

5. // 1

6. // 1

7.

上面的代码中，数组arr有两个空位，for…of并没有忽略它们。如果改成map遍历，是会跳过空位的。

entries()、keys()、values()、find()和findIndex会将空位处理成undefined。

1. // entries()

2. [...[, "a"].entries()]; // [[0, undefined], [1, "a"]]

3.

4. // keys()

5. [...[, "a"].keys()]; // [0, 1]

6.

7. // values()

8. [...[, "a"].values()]; // [undefined, "a"]

9.

10. // find()

11. [...[, "a"].find(x => true)]; // undefined

12.

13. // findIndex()

14. [...[, "a"].findIndex(x => true)]; // 0

15.

由于空位的处理规则非常不统一，所以建议避免出现空位。

**7.9 数组推导**

数组推导（array comprehension）提供了简洁写法，允许直接通过现有数组生成新数组。这项功能本来是要放入ES6的，但是TC39委员会想继续完善它，让其支持所有数据结构（内部调用iterator对象），不像现在只支持数组，所以就把它推迟到了ES7。Babel转码器已经支持这个功能。

1. var a1 = [1,2,3,4];

2. var a2 = [for (i of a1) i \* 2];

3.

4. a2 // [2,4,6,8]

5.

上面的代码表示，数组a2通过for…of结构直接在a1的基础上生成。

注意，数组推导中，for…of结构总是写在最前面，返回的表达式写在最后面。

for…of后面还可以附件if语句，用于设定循环的限制条件。

1. [for (year of years) if (year > 2000) year];

2. // [2006, 2010, 2014]

3.

4. [for (year of years) if (year > 2000) if (year < 2010) year];

5. // [2006]

6.

7. [for (year of years) if (year > 2000 && year < 2010) year];

8. // [2006]

9.

上面的代码表明，if语句要写在for…of与返回的表达式之间，而且可以多个if语句连用。

下面是另一个例子。

1. var customers = [

2.     {

3.         name: "Jack",

4.         age: 25,

5.         city: "New York"

6.     },

7.     {

8.         name: "Peter",

9.         age: 30,

10.         city: "Seattle"

11.     }

12. ];

13.

14. var results = [

15.     for (c of customers)

16.         if (c.city == "Seattle")

17.             { name: c.name, age: c.age}

18. ];

19.

20. results // {name: "Peter", age: 30}

21.

数组推导可以替代map和filter方法。

1. [for (i of [1, 2, 3]) i \* i];

2. // 等价于

3. [1, 2, 3].map(function (i) { return i \* i; });

4.

5. [for (i of [1, 4, 2, 3, -8]) if (i < 3) i];

6. // 等价于

7. [1, 4, 2, 3, -8].filter(function (i) { return i < 3 });

8.

上面的代码说明，模拟map功能只要单纯的for…of循环就行了，模拟filter功能除了for…of循环，还必须加上if语句。

在一个数组推到中，还可以使用多个for…of结构，构成多重循环。

1. var a1 = ["x1", "y1"];

2. var a2 = ["x2", "y2"];

3. var a3 = ["x3", "y3"];

4.

5. [for (s of a1) for (w of a2) for (r of a3) console.log(s + w + r)];

6. // x1x2x3

7. // x1x2y3

8. // x1y2x3

9. // x1y2y3

10. // y1x2x3

11. // y1x2y3

12. // y1y2x3

13. // y1y2y3

14.

上面的代码在一个数组推到中使用了3个for…of结构。

需要注意的是，数组推导的方括号构成了一个单独的作用域，在其中声明的变量类似于使用let语句声明的变量。

1. [for (c of "abcde") if (/[aeiou]/.test(c)) c].join(""); // "ae"

2. [for (c of "abcde") c + "0"].join(""); // a0b0c0d0e0

3.

上面的代码使用了数组推导对字符串进行处理。

数组推导需要注意的是，新数组会立即在内存中生成。这时，如果原数组很大，将非常耗费内存。