```
var s = new Set();

var x = { id: 1 },
    y = { id: 2 };

s.add( x ).add( y );

var keys = [ ...s.keys() ],
    vals = [ ...s.values() ],
    entries = [ ...s.entries() ];

keys[0] === x;
keys[1] === y;

vals[0] === x;
vals[1] === y;

entries[0][0] === x;
entries[0][1] === x;
entries[1][0] === y;
entries[1][1] === y;
```

`keys()` 和 `values()` 迭代器都从 set 中 yield 出一列不重复的值。`entries()` 迭代器 yield 出一列项目数组，其中的数组的两个项目都是唯一 set 值。set 默认的迭代器是它的 `values()` 迭代器。

set 固有的唯一性是它最有用的特性。举例来说：

```
var s = new Set( [1,2,3,4,"1",2,4,"5"] ),
    uniques = [ ...s ];

uniques;                        // [1,2,3,4,"1","5"]
```

set 的唯一性不允许强制转换，所以 `1` 和 `"1"` 被认为是不同的值。

5.5 WeakSet

就像 WeakMap 弱持有它的键（对其值是强持有的）一样，WeakSet 对其值也是弱持有的（这里并没有键）：

```
var s = new WeakSet();

var x = { id: 1 },
    y = { id: 2 };

s.add( x );
s.add( y );

x = null;                       // x可GC
y = null;                       // y可GC
```

WeakSet 的值必须是对象，而并不像 set 一样可以是原生类型值。

5.6 小结

ES6 定义了几个有用的集合，这使得对数据的访问更结构化且更高效。

TypedArray 提供了对二进制数据 buffer 的各种整型类型“视图”，比如 8 位无符号整型和 32 位浮点型。对二进制数据的数组访问使得运算更容易表达和维护，从而可以更容易操纵视频、音频、canvas 数据等这样的复杂数据。

Map 是键 – 值对，其中的键不只是字符串 / 原生类型，也可以是对象。Set 是成员值（任意类型）唯一的列表。

WeakMap 也是 map，其中的键（对象）是弱持有的，因此当它是对这个对象的最后一个引用的时候，GC（垃圾回收）可以回收这个项目。WeakSet 也是 set，其中的值是弱持有的，也就是说如果其中的项目是对这个对象最后一个引用的时候，GC 可以移除它。

第 6 章

新增 API

从值的转换到数学计算，ES6 为各种内置原生类型和对象新增了很多静态属性和方法，用来辅助完成一些常见的任务。另外，某些原生类型的实例通过新的原型方法有了新的功能。

这些特性中大多数都可以忠实 polyfill。这里我们不讨论这样的细节，你可以在“ES6 Shim”项目（https://github.com/paul millr/es6-shim/）中找到符合标准的 shim / polyfill。

6.1 Array

各种 JavaScript 用户库扩展最多的特性之一就是数组（Array）类型。所以 ES6 为 Array 增加了一些静态函数和原型（实例）方法辅助函数也在意料之中。

6.1.1 静态函数 `Array.of(..)`

`Array(..)` 构造器有一个众所周知的陷阱，就是如果只传入一个参数，并且这个参数是数字的话，那么不会构造一个值为这个数字的单个元素的数组，而是构造一个空数组，其 `length` 属性为这个数字。这个动作会产生不幸又诡异的“空槽”行为，这是 JavaScript 数组广为人所诟病的一点。

`Array.of(..)` 取代了 `Array(..)` 成为数组的推荐函数形式构造器，因为 `Array.of(..)` 并没有这个特殊的单个数字参数的问题。考虑：

```
var a = Array( 3 );
a.length;                       // 3
a[0];                           // undefined

var b = Array.of( 3 );
b.length;                       // 1
b[0];                           // 3

var c = Array.of( 1, 2, 3 );
c.length;                       // 3
c;                              // [1,2,3]
```

什么情况下你会需要使用 `Array.of(..)` 而不是只用 `c = [1,2,3]` 这样的字面值语法创建一个数组呢？有两种可能的情况。

如果你有一个回调函数需要传入的参数封装为数组，`Array.of(..)` 可以完美解决这个需求。这样的用法不是很常见，但是可能恰好“解了你的痒”。

另外一种情况是，如果你构建 `Array` 的子类（参见 3.4 节），并且想要在你的子类实例中创建和初始化元素，比如：

```
class MyCoolArray extends Array {
    sum() {
        return this.reduce( function reducer(acc,curr){
            return acc + curr;
        }, 0 );
    }
}

var x = new MyCoolArray( 3 );
x.length;                       // 3--oops!
x.sum();                        // 0--oops!

var y = [3];                    // Array，而不是MyCoolArray
y.length;                       // 1
y.sum();                        // sum不是一个函数

var z = MyCoolArray.of( 3 );
z.length;                       // 1
z.sum();                        // 3
```

你不能（简单地）只是为 `MyCoolArray` 创建一个构造器来覆盖 `Array` 父构造器的行为，因为那个构造器对于实际构造一个行为符合规范的数组值（初始化 `this`）是必要的。`MyCoolArray` 子类“继承来的”静态 `of(..)` 方法提供了很好的解决方案。

6.1.2 静态函数 Array.from(..)

JavaScript 中的“类（似）数组对象”是指一个有 `length` 属性，具体说是大于等于 0 的整数值的对象。

这样的值在使用 JavaScript 工作的过程中是非常令人沮丧的；普遍的需求就是把它们转换为真正的数组，这样就可以应用各种 Array.prototype 方法（map(..)、indexOf(..) 等）了。这个过程通常类似于：

```
// 类数组对象
var arrLike = {
    length: 3,
    0: "foo",
    1: "bar"
};

var arr = Array.prototype.slice.call( arrLike );
```

另外一个常见的任务是使用 slice(..) 来复制产生一个真正的数组：

```
var arr2 = arr.slice();
```

两种情况下，新的 ES6 Array.from(..) 方法都是更好理解、更优雅、更简洁的替代方法：

```
var arr = Array.from( arrLike );

var arrCopy = Array.from( arr );
```

Array.from(..) 检查第一个参数是否为 iterable（参见 3.1 节），如果是的话，就使用迭代器来产生值并“复制”进入返回的数组。因为真正的数组有一个这些值之上的迭代器，所以会自动使用这个迭代器。

而如果你把类数组对象作为第一个参数传给 Array.from(..)，它的行为方式和 slice()（没有参数）或者 apply(..) 是一样的，就是简单地按照数字命名的属性从 0 开始直到 length 值在这些值上循环。

考虑：

```
var arrLike = {
    length: 4,
    2: "foo"
};

Array.from( arrLike );
// [ undefined, undefined, "foo", undefined ]
```

因为位置 0、1 和 3 在 arrLike 上并不存在，所以在这些位置上是 undefined 值。

你也可以这样产生类似的结果：

```
var emptySlotsArr = [];
emptySlotsArr.length = 4;
emptySlotsArr[2] = "foo";
```

```
Array.from( emptySlotsArr );
// [ undefined, undefined, "foo", undefined ]
```

1. 避免空槽位

前面代码中的 `emptySlotArr` 和 `Array.from(..)` 调用的结果有一个微妙但重要的区别。也就是 `Array.from(..)` 永远不会产生空槽位。

在 ES6 之前，如果你想要产生一个初始化为某个长度，在每个槽位上都是真正的 `undefined` 值（不是空槽位！）的数组，不得不做额外的工作：

```
var a = Array( 4 );
// 4个空槽位!

var b = Array.apply( null, { length: 4 } );
// 4个undefined值
```

而现在 `Array.from(..)` 使其简单了很多：

```
var c = Array.from( { length: 4 } );
// 4个undefined值
```

像前面代码中的 `a` 那样使用空槽位数组能在某些数组函数上工作，但是另外一些会忽略空槽位（比如 `map(..)` 等）。永远不要故意利用空槽位工作，因为它几乎肯定会导致程序出现诡异 / 意料之外的行为。

2. 映射

`Array.from(..)` 工具还有另外一个有用的技巧。如果提供了的话，第二个参数是一个映射回调（和一般的 `Array#map(..)` 所期望的几乎一样），这个函数会被调用，来把来自于源的每个值映射 / 转换到返回值。考虑：

```
var arrLike = {
    length: 4,
    2: "foo"
};

Array.from( arrLike, function mapper(val,idx){
    if (typeof val == "string") {
        return val.toUpperCase();
    }
    else {
        return idx;
    }
} );
// [ 0, 1, "FOO", 3 ]
```

和其他接收回调的数组方法一样，Array.from(..) 接收一个可选的第三个参数，如果设置了的话，这个参数为作为第二个参数传入的回调指定 this 绑定。否则，this 将会是 undefined。

参见 5.1 节，其中给出了使用 Array.from(..) 把 8 位值数组转换为 16 位值数组的例子。

6.1.3 创建数组和子类型

前面几小节中，我们已经讨论了 Array.of(..) 和 Array.from(..)，二者都以与构造器类似的方式创建一个新数组，而在子类型方面它们又是怎样的呢？它们会创建基类 Array 的实例还是继承子类型的实例呢？

```
class MyCoolArray extends Array {
    ..
}

MyCoolArray.from( [1, 2] ) instanceof MyCoolArray;  // true

Array.from(
    MyCoolArray.from( [1, 2] )
) instanceof MyCoolArray;                           // false
```

of(..) 和 from(..) 都使用访问它们的构造器来构造数组。所以如果使用基类 Array.of(..)，那么得到的就是 Array 实例；如果使用 MyCoolArray.of(..)，那么得到的就是 MyCoolArray 实例。

在 3.4 节中，我们介绍了 @@species 设置，所有的内置类（比如 Array）都有定义，任何创建新实例的原型方法都会使用它。slice(..) 是一个很好的例子：

```
var x = new MyCoolArray( 1, 2, 3 );

x.slice( 1 ) instanceof MyCoolArray;                // true
```

一般来说，默认的行为方式很可能就是需要的，但就像我们在第 3 章中介绍的，必要的话也可以覆盖它：

```
class MyCoolArray extends Array {
    // 强制species为父构造器
    static get [Symbol.species]() { return Array; }
}

var x = new MyCoolArray( 1, 2, 3 );

x.slice( 1 ) instanceof MyCoolArray;          // false
x.slice( 1 ) instanceof Array;                // true
```

需要注意的是，@@species 设置只用于像 slice(..) 这样的原型方法。of(..) 和 from(..)

不会使用它；它们都只使用 this 绑定（由使用的构造器来构造其引用）。考虑：

```
class MyCoolArray extends Array {
    // 强制species为父构造器
    static get [Symbol.species]() { return Array; }
}

var x = new MyCoolArray( 1, 2, 3 );

MyCoolArray.from( x ) instanceof MyCoolArray;       // true
MyCoolArray.of( [2, 3] ) instanceof MyCoolArray;    // true
```

6.1.4 原型方法 copyWithin(..)

Array#copyWithin(..) 是一个新的修改器方法，（包括带类型的数组在内的，参见第 5 章）所有数组都支持。copyWithin(..) 从一个数组中复制一部分到同一个数组的另一个位置，覆盖这个位置所有原来的值。

参数是 target（要复制到的索引）、start（开始复制的源索引，包括在内）以及可选的 end（复制结束的不包含索引）。如果任何一个参数是负数，就被当作是相对于数组结束的相对值。

考虑：

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

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

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

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

就像前面代码片段展示的，copyWithin(..) 方法不会增加数组的长度。到达数组结尾复制就会停止。

与你想象的正相反，复制并非总是从左到右（索引递增）进行的。如果源范围和目标范围重叠的话，可能会出现重复复制已经复制的值，而这可能并非你想要的结果。

所以，内部算法通过反向复制避免了这种情况。考虑：

```
[1,2,3,4,5].copyWithin( 2, 1 );     // ???
```

如果算法严格按照从左到右来移动，那么 2 应该被复制来覆盖 3，然后这个被复制的 2 应该被复制来覆盖 4，然后这个被复制的 2 应该被复制来覆盖 5，而你最终会得到 [1,2,2,2,2]。

而实际上，复制算法会反向进行，复制 4 来覆盖 5，然后复制 3 来覆盖 4，然后复制 2 来覆盖 3，最后的结果是 [1,2,2,3,4]。根据期望来说，这可能是更“正确”的结果，但如果只

考虑简单的从左到右方式的复制算法，你可能会觉得很迷惑。

6.1.5 原型方法 fill(..)

可以通过 ES6 原生支持的方法 Array#fill(..) 用指定值完全（或部分）填充已存在的数组：

```
var a = Array( 4 ).fill( undefined );
a;
// [undefined,undefined,undefined,undefined]
```

fill(..) 可选地接收参数 start 和 end，它们指定了数组要填充的子集位置，比如：

```
var a = [ null, null, null, null ].fill( 42, 1, 3 );

a;                                  // [null,42,42,null]
```

6.1.6 原型方法 find(..)

一般来说，在数组中搜索一个值的最常用方法一直是 indexOf(..) 方法，这个方法返回找到值的索引，如果没有找到就返回 -1：

```
var a = [1,2,3,4,5];

(a.indexOf( 3 ) != -1);             // true
(a.indexOf( 7 ) != -1);             // false

(a.indexOf( "2" ) != -1);           // false
```

相比之下，indexOf(..) 需要严格匹配 ===，所以搜索 "2" 不会找到值 2，反之也是如此。indexOf(..) 的匹配算法无法覆盖，而且要手动与值 -1 进行比较也很麻烦 / 笨拙。

本系列《你不知道的 JavaScript（中卷）》第一部分中给出了一个有趣（也充满争议地难以理解）的技术，用 ~ 运算符来绕过丑陋的返回值 -1 的问题。

从 ES5 以来，控制匹配逻辑的最常用变通技术是使用 some(..) 方法。它的实现是通过为每个元素调用一个函数回调，直到某次调用返回 true / 真值时才会停止。因为你可以定义这个回调函数，也就有了对匹配方式的完全控制：

```
var a = [1,2,3,4,5];

a.some( function matcher(v){
    return v == "2";
} );                    // true
```

```
a.some( function matcher(v){
    return v == 7;
} );                    // false
```

但这种方式的缺点是如果找到匹配的值的时候，只能得到匹配的 true/false 指示，而无法得到真正的匹配值本身。

ES6 的 find(..) 解决了这个问题。基本上它和 some(..) 的工作方式一样，除了一旦回调返回 true/ 真值，会返回实际的数组值：

```
var a = [1,2,3,4,5];

a.find( function matcher(v){
    return v == "2";
} );                                // 2

a.find( function matcher(v){
    return v == 7;                  // undefined
});
```

通过自定义 matcher(..) 函数也可以支持比较像对象这样的复杂值：

```
var points = [
    { x: 10, y: 20 },
    { x: 20, y: 30 },
    { x: 30, y: 40 },
    { x: 40, y: 50 },
    { x: 50, y: 60 }
];

points.find( function matcher(point) {
    return (
        point.x % 3 == 0 &&
        point.y % 4 == 0
    );
} );                                // { x: 30, y: 40 }
```

就像其他接受回调的数组方法一样，find(..) 接受一个可选的第二个参数，如果设定这个参数就绑定到第一个参数回调的 this。否则，this 就是 undefined。

6.1.7 原型方法 findIndex(..)

前面一小节展示了 some(..) 如何 yield 出一个布尔型结果用于在数组中搜索，以及 find(..) 如何从数组搜索 yield 出匹配的值本身，另外，还需要找到匹配值的位置索引。

indexOf(..) 会提供这些，但是无法控制匹配逻辑；它总是使用 === 严格相等。所以 ES6 的 findIndex(..) 才是解决方案：