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

points.findIndex( function matcher(point) {
    return (
        point.x % 3 == 0 &&
        point.y % 4 == 0
    );
} );                                // 2

points.findIndex( function matcher(point) {
    return (
        point.x % 6 == 0 &&
        point.y % 7 == 0
    );
} );                                // -1
```

不要使用 `findIndex(..) != -1`（这是 `indexOf(..)` 的惯用法）从搜索中得到布尔值，因为 `some(..)` 已经 yield 出你想要的 `true/false`。也不要用 `a[ a.findIndex(..) ]` 来得到匹配值，因为这是 `find(..)` 所做的事。最后，如果需要严格匹配的索引值，那么使用 `indexOf(..)`；如果需要自定义匹配的索引值，那么使用 `findIndex(..)`。

就像其他接收回调的数组方法一样，`findIndex(..)` 接收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`。

### 6.1.8 原型方法 entries()、values()、keys()

在第 3 章中，我们展示了各种数据结构如何通过迭代器提供一个依次枚举其值的模式。然后在第 5 章探索新的 ES6 集合（Map、Set 等）如何提供了几种方法以产生不同迭代的时候，详细展示了这种方法。

因为 `Array` 对于 ES6 来说已经不是新的了，所以从传统角度来说，它可能不会被看作是“集合”，但是它提供了同样的迭代器方法 `entries()`、`values()` 和 `keys()`，从这个意义上说，它是一个集合。考虑：

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

[...a.values()];                    // [1,2,3]
[...a.keys()];                      // [0,1,2]
[...a.entries()];                   // [ [0,1], [1,2], [2,3] ]

[...a[Symbol.iterator]()];          // [1,2,3]
```

就像 Set 一样，默认的 Array 迭代器和 values() 返回的值一样。

在后面的 6.5.4 节中，我们将展示 Array.from(..) 如何把数组中的空槽位看作值为 undefined 的槽位。这实际上是因为在底层数组迭代器是这样工作的：

```
var a = [];
a.length = 3;
a[1] = 2;

[...a.values()];        // [undefined,2,undefined]
[...a.keys()];          // [0,1,2]
[...a.entries()];       // [ [0,undefined], [1,2], [2,undefined] ]
```

# 6.2 Object

Object 也新增了几个静态辅助函数。传统上认为，这一类函数的关注点在对象值的行为方式 / 功能上。

但是，从 ES6 开始，Object 静态方法也开始用于那些还没有更自然的有另外归属的（比如 Array.from(..)）通用全局 API。

## 6.2.1 静态函数 Object.is(..)

静态函数 Object.is(..) 执行比 === 比较更严格的值比较。

Object.is(..) 调用底层 SameValue 算法（ES6 规范，7.2.9 节）。SameValue 算法基本上和 === 严格相等比较算法一样（ES6 规范，7.2.13 节），但有两个重要的区别。

考虑：

```
var x = NaN, y = 0, z = -0;

x === x;                    // false
y === z;                    // true

Object.is( x, x );          // true
Object.is( y, z );          // false
```

你应该继续使用 === 进行严格相等比较；不应该把 Object.is(..) 当作这个运算符的替代。但是，如果需要严格识别 NaN 或者 -0 值，那么应该选择 Object.is(..)。

ES6 还新增了一个 Number.isNaN(..) 工具（本章后面会介绍），这个工具可能是更方便的检查工具；与 Object.is(x,NaN) 相比，你可能更喜欢 Number.isNaN(x)。你可以使用 x == 0 && 1 / x === -Infinity 这种笨拙的方式精确判断 -0 值，但这种情况下使用 Object.is(x,-0) 会好很多。

### 6.2.2 静态函数 Object.getOwnPropertySymbols(..)

我们在 2.13 节讨论过 ES6 中新增的基本值类型 Symbol。

Symbol 很可能会成为对象最常用的特殊（元）属性。所以引入了工具 `Object.getOwnPropertySymbols(..)`，它直接从对象上取得所有的符号属性：

```
var o = {
    foo: 42,
    [ Symbol( "bar" ) ]: "hello world",
    baz: true
};

Object.getOwnPropertySymbols( o );  // [ Symbol(bar) ]
```

### 6.2.3 静态函数 Object.setPrototypeOf(..)

还是在第 2 章中，我们提到工具 `Object.setPrototypeOf(..)`，这个工具（不出人意料地）设置对象的 `[[Prototype]]` 用于**行为委托**（参见本系列《你不知道的 JavaScript（上卷）》第二部分）。考虑：

```
var o1 = {
    foo() { console.log( "foo" ); }
};
var o2 = {
    // .. o2的定义 ..
};

Object.setPrototypeOf( o2, o1 );

// 委托给o1.foo()
o2.foo();                           // foo
```

也可以：

```
var o1 = {
    foo() { console.log( "foo" ); }
};

var o2 = Object.setPrototypeOf( {
    // .. o2的定义 ..
}, o1 );

// 委托给o1.foo()
o2.foo();                           // foo
```

前面两段代码中，`o2` 和 `o1` 的关系都出现在 `o2` 定义的结尾处。更通俗地说，`o2` 和 `o1` 的关系在 `o2` 的定义上指定，就像类一样，也和字面值对象中的 `__proto__` 一样（参见 2.6.4 节）。

如前所示，紧接对象创建之后设定 [[Prototype]] 是合理的。但是在很久之后才修改它不是一个好主意，因为这通常会产生令人迷惑而非清晰的代码。

### 6.2.4 静态函数 Object.assign(..)

很多 JavaScript 库 / 框架提供了用于把一个对象的属性复制 / 混合到另一个对象中的工具（比如，JQuery 的 extent(..)）。这些不同的工具之间有各种细微的区别，比如是否忽略值为 undefined 的属性。

ES6 新增了 Object.assign(..)，这是这些算法的简化版本。第一个参数是 target，其他传入的参数都是源，它们将按照列出的顺序依次被处理。对于每个源来说，它的可枚举和自己拥有的（也就是不是“继承来的”）键值，包括符号都会通过简单 = 赋值被复制。Object.assign(..) 返回目标对象。

考虑这个对象设定：

```
var target = {},
    o1 = { a: 1 }, o2 = { b: 2 },
    o3 = { c: 3 }, o4 = { d: 4 };

// 设定只读属性
Object.defineProperty( o3, "e", {
    value: 5,
    enumerable: true,
    writable: false,
    configurable: false
} );

// 设定不可枚举属性
Object.defineProperty( o3, "f", {
    value: 6,
    enumerable: false
} );

o3[ Symbol( "g" ) ] = 7;

// 设定不可枚举符号
Object.defineProperty( o3, Symbol( "h" ), {
    value: 8,
    enumerable: false
} );

Object.setPrototypeOf( o3, o4 );
```

只有属性 a、b、c、e 以及 Symbol("g") 会被复制到 target 中：

```
Object.assign( target, o1, o2, o3 );

target.a;                           // 1
target.b;                           // 2
target.c;                           // 3

Object.getOwnPropertyDescriptor( target, "e" );
// { value: 5, writable: true, enumerable: true,
//   configurable: true }

Object.getOwnPropertySymbols( target );
// [Symbol("g")]
```

复制过程会忽略属性 d、f 和 Symbol("h")；不可枚举的属性和非自有的属性都被排除在赋值过程之外。另外，e 作为一个普通属性赋值被复制，而不是作为只读属性复制。

在前面一节中，我们展示了使用 setPrototypeOf(..) 设定对象 o2 和 o1 之间的 [[Prototype]] 关系。还有另外一种应用了 Object.assign(..) 的形式：

```
var o1 = {
    foo() { console.log( "foo" ); }
};

var o2 = Object.assign(
    Object.create( o1 ),
    {
        // .. o2的定义 ..
    }
);

// 委托给o1.foo()
o2.foo();                // foo
```

Object.create(..) 是 ES5 工具，创建一个 [[Prototype]] 链接的空对象。参见本系列《你不知道的 JavaScript（上卷）》第二部分获取更多信息。

## 6.3 Math

ES6 增加了几个新的数学工具，填补了常用计算方面的空白。这些工具都可以手动计算，但是其中多数现在有了原生定义，所以某些情况下 JavaScript 引擎可以更优化地执行这些计算，或者执行比手动版本精度更高的计算。

这些工具的使用者更可能是 asm.js/transpile 的 JavaScript 代码（参见本系列《你不知道的 JavaScript（中卷）》第二部分），而非直接开发者。

三角函数：

`cosh(..)`
双曲余弦函数

`acosh(..)`
双曲反余弦函数

`sinh(..)`
双曲正弦函数

`asinh(..)`
双曲反正弦函数

`tanh(..)`
双曲正切函数

`atanh(..)`
双曲反正切函数

`hypot(..)`
平方和的平方根（也即：广义勾股定理）

算术：

`cbrt(..)`
立方根

`clz32(..)`
计算 32 位二进制表示的前导 0 个数

`expm1(..)`
等价于 `exp(x) - 1`

`log2(..)`
二进制对数（以 2 为底的对数）

`log10(..)`
以 10 为底的对数

`log1p(..)`
等价于 `log(x + 1)`

`imul(..)`
两个数字的 32 位整数乘法

元工具：

`sign(..)`
: 返回数字符号

`trunc(..)`
: 返回数字的整数部分

`fround(..)`
: 向最接近的 32 位（单精度）浮点值取整

## 6.4 Number

更重要的是，要让程序正常工作，必须精确处理数字。ES6 新增了额外的属性和函数来提供常用数字运算。

对 `Number` 的两个新增内容就是指向已有的全局函数的引用：`Number.parseInt(..)` 和 `Number.parseFloat(..)`。

### 6.4.1 静态属性

ES6 新增了一些作为静态属性的辅助数字常量：

`Number.EPSILON`
: 任意两个值之间的最小差：`2^-52`（参见本系列《你不知道的 JavaScript（中卷）》第一部分的第 2 章，其使用了这个值作为浮点数算法的精度误差值）

`Number.MAX_SAFE_INTEGER`
: JavaScript 可以用数字值无歧义“安全”表达的最大整数：`2^53 - 1`

`Number.MIN_SAFE_INTEGER`
: JavaScript 可以用数字值无歧义“安全”表达的最小整数：`-(2^53 - 1)` 或 `(-2)^53 + 1`

关于“安全”整型的更多信息，参见本系列《你不知道的 JavaScript（中卷）》第一部分的第 2 章。

### 6.4.2 静态函数 Number.isNaN(..)

标准全局工具 `isNaN(..)` 自出现以来就是有缺陷的，它对非数字的东西都会返回 `true`，而不是只对真实的 `NaN` 值返回 `true`，因为它把参数强制转换为数字类型（可能会错误地导致

NaN）。ES6 增加了一个修正工具 Number.isNaN(..)，可以按照期望工作：

```
var a = NaN, b = "NaN", c = 42;

isNaN( a );                     // true
isNaN( b );                     // true--oops!
isNaN( c );                     // false

Number.isNaN( a );              // true
Number.isNaN( b );              // false--修正了!
Number.isNaN( c );              // false
```

### 6.4.3 静态函数 Number.isFinite(..)

看到像 isFinite(..) 这样的函数名，我们常常认为它的意思就是“非无限的”。但是这并不完全正确。这个 ES6 新工具有一些微妙之处。考虑：

```
var a = NaN, b = Infinity, c = 42;

Number.isFinite( a );           // false
Number.isFinite( b );           // false

Number.isFinite( c );           // true
```

标准的全局 isFinite(..) 会对参数进行强制类型转换，但是 Number.isFinite(..) 会略去这种强制行为：

```
var a = "42";

isFinite( a );                  // true
Number.isFinite( a );           // false
```

可能你更需要类型转换，这种情况下全局 isFinite(..) 是一个有效的选择。另外，也许是更合理的，可以使用 Number.isFinite(+x)，它会在传入之前显式地把 x 强制转换为数字（参见本系列《你不知道的 JavaScript（中卷）》第一部分的第 4 章）。

### 6.4.4 整型相关静态函数

JavaScript 的数字值永远都是浮点数（IEE-754）。所以确定数字是否为“整型”的概念并不是检查其类型，因为 JavaScript 并没有这样区分。

相反，你需要检查这个值的小数部分是否非 0。最简单的实现方法通常是：

```
x === Math.floor( x );
```

ES6 新增了一个辅助工具 Number.isInteger(..)，这个工具可能会更有效地确定这个性质：

```
Number.isInteger( 4 );          // true
Number.isInteger( 4.2 );        // false
```

在 JavaScript 中，4、4.、4.0 或者 4.0000 之间并没有区别。所有这些都会被当作“整型”并且从 Number.isInteger(..) 中返回 true。

另外，Number.isInteger(..) 会过滤掉 x === Math.floor(x) 可能会搞混的明显非整数值：

```
Number.isInteger( NaN );            // false
Number.isInteger( Infinity );       // false
```

使用“整数”工作有时候是一个重要的信息，因为这可能会简化某些类型的算法。JavaScript 代码本身不会因为只使用整数而运行得更快，但是，只有在使用整数时，引擎才可以采用优化技术（比如 asm.js）。

基于 Number.isInteger(..) 对 NaN 和 Infinity 值的处理方式，要定义一个 isFloat(..) 工具并不像 !Number.isInteger(..) 那么简单。可能需要做类似于下面这样的事情：

```
function isFloat(x) {
    return Number.isFinite( x ) && !Number.isInteger( x );
}

isFloat( 4.2 );             // true
isFloat( 4 );               // false

isFloat( NaN );             // false
isFloat( Infinity );        // false
```

看起来可能有点奇怪，但是 Infinity 既不应该被当作整型又不应该被当作浮点型。

ES6 还定义了一个工具 Number.isSafeInteger(..)，这个工具检查一个值以确保其为整数并且在 Number.MIN_SAFE_INTEGER-Number.MAX_SAFE_INTEGER 范围之内（全包含）：

```
var x = Math.pow( 2, 53 ),
    y = Math.pow( -2, 53 );

Number.isSafeInteger( x - 1 );          // true
Number.isSafeInteger( y + 1 );          // true

Number.isSafeInteger( x );              // false
Number.isSafeInteger( y );              // false
```

# 6.5 字符串

在 ES6 之前已经有了很多字符串辅助函数，现在又增加了一些。

### 6.5.1 Unicode 函数

我们在 2.12.1 节详细介绍过 `String.fromCodePoint(..)`、`String#codePointAt(..)` 和 `String#normalize(..)`。新增这些函数是为了提高 JavaScript 字符串值对 Unicode 的支持：

```
String.fromCodePoint( 0x1d49e );            // "𝒞"
"ab𝒞d.codePointAt( 2 ).toString( 16 );      // "1d49e"
```

字符串原型方法 `normalize(..)` 用于执行 Unicode 规范化，或者把字符用“合并符”连接起来或者把合并的字符解开。

一般来说，规范化不会对字符串的内容造成可见的效果，但是会改变字符串的内容，这可能会影响像 `length` 属性的结果，以及通过位置访问字符的方式：

```
var s1 = "e\u0301";
s1.length;                              // 2

var s2 = s1.normalize();
s2.length;                              // 1
s2 === "\xE9";                          // true
```

`normalize(..)` 接受一个可选的参数，来指定要使用的规范化形式。这个参数必须是这几个值之一：`"NFC"`（默认）、`"NFD"`、`"NFKC"` 或者 `"NFKD"`。

规范化形式及其对字符串产生的影响超出了本部分的讨论范围。参见“Unicode Normalization Forms”（http://www.unicode.org/reports/tr15/）获取更多信息。

### 6.5.2 静态函数 `String.raw(..)`

`String.raw(..)` 工具作为内置标签函数提供，与模板字符串字面值（参见第 2 章）一起使用，用于获得不应用任何转义序列的原始字符串。

这个函数基本上不会被手动调用，而是与标签模板字面值一起使用：

```
var str = "bc";

String.raw`\ta${str}d\xE9`;
// "\tabcd\xE9"，而不是"  abcdé"
```

在结果字符串中，`\` 和 `t` 是独立的原始字符，而不是转义字符 `\t`。对于 Unicode 转义序列也是一样。

### 6.5.3 原型函数 `repeat(..)`

像 Python 和 Ruby 这样的语言中，可以这样重复字符串：

```
"foo" * 3;                              // "foofoofoo"
```