JavaScript 不支持这种形式，因为乘法 * 只对数字有定义，因此 "foo" 被强制转换成了数字 NaN。

然而，ES6 定义了一个字符串原型方法 repeat(..) 来完成这个任务：

```
"foo".repeat( 3 );                  // "foofoofoo"
```

6.5.4 字符串检查函数

除了 ES6 之前的 String#indexOf(..) 和 String#lastIndexOf(..)，又新增了 3 个用于搜索 / 检查的新方法：startsWith(..)、endsWidth(..) 和 includes(..)。

```
var palindrome = "step on no pets";

palindrome.startsWith( "step on" );     // true
palindrome.startsWith( "on", 5 );       // true

palindrome.endsWith( "no pets" );       // true
palindrome.endsWith( "no", 10 );        // true

palindrome.includes( "on" );            // true
palindrome.includes( "on", 6 );         // false
```

对于所有的字符串搜索 / 检查方法，如果寻找空字符串 ""，总是会在字符串的开头或结尾找到。

默认情况下，这些方法不会接受正则表达式用于字符串搜索。参见 7.3.5 节中关于关闭对第一个参数执行的 isRegExp 检查的部分。

6.6 小结

ES6 为各种内置原生对象新增了许多额外的辅助 API。

- Array 新增了静态函数 of(..) 和 from(..)，以及像 copyWithin(..) 和 fill(..) 这样的原型函数。
- Object 新增了静态函数 is(..) 和 assign(..)。
- Math 新增了静态函数 acosh(..) 和 clz32(..)。
- Number 新增了静态属性 Number.EPSILON，以及静态函数 Number.isFinite(..)。
- String 新增了静态函数 String.fromCodePoint(..) 和 String.raw(..)，以及原型函数 repeat(..) 和 includes(..)。

这些新增内容多数都可以 polyfill（参见 ES6 Shim），其思路来自于常用的 JavaScript 库和框架。

第 7 章

元编程

元编程是指操作目标是程序本身的行为特性的编程。换句话说，它是对程序的编程的编程。有点拗口，是吧？

举例来说，如果想要查看对象 `a` 和另外一个对象 `b` 的关系是否是 `[[Prototype]]` 链接的，可以使用 `a.isProto type(b)`，这是一种元编程形式，通常称为内省（introspection）。另外一个明显的元编程例子是宏（在 JavaScript 中还不支持）——代码在编译时修改自身。用 `for..in` 循环枚举对象的键，或者检查一个对象是否是某个“类构造器”的**实例**，也都是常见的元编程例子。

元编程关注以下一点或几点：代码查看自身、代码修改自身、代码修改默认语言特性，以此影响其他代码。

元编程的目标是利用语言自身的内省能力使代码的其余部分更具描述性、表达性和灵活性。因为元编程的**元**（meta）本质，我们有点难以给出比上面提到的更精确的定义。要理解元编程，最好的方法是通过实例来展示。

ES6 在 JavaScript 现有的基础上为元编程新增了一些形式 / 特性。

7.1 函数名称

你的代码在有些情况下可能想要了解自身，想要知道某个函数的名称是什么。如何得知一个函数的名称，答案出人意料地有些模棱两可。 考虑：

```
function daz() {
    // ..
}

var obj = {
    foo: function() {
         // ..
    },
    bar: function baz() {
        // ..
    },
    bam: daz,
    zim() {
        // ..
    }
};
```

在前面的代码中，“obj.foo() 的名称是什么”略显微妙。是 "foo"、""，还是 undefined？obj.bar() 呢？它的名称是 "bar" 还是 "baz"？ obj.bam() 的名称是 "bam" 还是 "daz"？ obj.zim() 呢？

此外，作为回调传递的函数又是怎样的呢？比如：

```
function foo(cb) {
    // 这里cb()的名称是什么?
}

foo( function(){
    // 我是匿名的!
} );
```

程序中有多种方式可以表达一个函数，函数的“名称”应该是什么并非总是清晰无疑的。

更重要的是，我们需要确定函数的“名称”是否就是它的 name 属性（是的，函数有一个名为 name 的属性），或者它是否指向其词法绑定名称，比如 function bar() {..} 中的 bar。

词法绑定名称是用于像递归这样的任务：

```
function foo(i) {
    if (i < 10) return foo( i * 2 );
    return i;
}
```

name 属性是用于元编程目的的，所以它是这里讨论的关注点。

这里比较混乱，因为默认情况下函数的词法名称（如果有的话）也会被设为它的 name 属性。实际上，ES5（和之前的）规范对这一行为并没有正式要求。name 属性的设定是非标准的，但还是比较可靠的。而在 ES6 中这一点已经得到了标准化。

如果函数设定了 name 值，那么这个值通常也就是开发者工具中栈踪迹使用的名称。

推导

对于没有词法名称的函数，name 属性是怎样的呢？

在 ES6 中，现在已经有了一组推导规则可以合理地为函数的 name 属性赋值，即使这个函数并没有词法名称可用。

考虑：

```
var abc = function() {
    // ..
};

abc.name;               // "abc"
```

如果给了这个函数一个词法名称，比如 abc = function def() { .. }，那么 name 属性当然就是 "def"。而如果没有词法名称的话，直觉上看似乎名称为 "abc" 比较合理。

下面是 ES6 中名称推导（或者没有名称）的其他几种形式：

```
(function(){ .. });                     // name:
(function*(){ .. });                    // name:
window.foo = function(){ .. };          // name:

class Awesome {
    constructor() { .. }                // name: Awesome
    funny() { .. }                      // name: funny
}

var c = class Awesome { .. };           // name: Awesome

var o = {
    foo() { .. },                       // name: foo
    *bar() { .. },                      // name: bar
    baz: () => { .. },                  // name: baz
    bam: function(){ .. },              // name: bam
    get qux() { .. },                   // name: get qux
    set fuz() { .. },                   // name: set fuz
    ["b" + "iz"]:
        function(){ .. },               // name: biz
    [Symbol( "buz" )]:
        function(){ .. }                // name: [buz]
};

var x = o.foo.bind( o );                // name: bound foo
(function(){ .. }).bind( o );           // name: bound
```

```
export default function() { .. }     // name: default

var y = new Function();              // name: anonymous
var GeneratorFunction =
    function*(){}.__proto__.constructor;
var z = new GeneratorFunction();     // name: anonymous
```

默认情况下，name 属性不可写，但可配置，也就是说如果需要的话，可使用 Object.defineProperty(..) 来手动修改。

7.2 元属性

在 3.4.3 节中，我们介绍了 ES6 中的一个 JavaScript 新概念：元属性。正如其名称所暗示的，元属性以属性访问的形式提供特殊的其他方法无法获取的元信息。

以 new.target 为例，关键字 new 用作属性访问的上下文。显然，new 本身并不是一个对象，因此这个功能很特殊。而在构造器调用（通过 new 触发的函数 / 方法）内部使用 new.target 时，new 成了一个虚拟上下文，使得 new.target 能够指向调用 new 的目标构造器。

这个是元编程操作的一个明显示例，因为它的目的是从构造器调用内部确定最初 new 的目标是什么，通用地说就是用于内省（检查类型 / 结构）或者静态属性访问。

举例来说，你可能需要在构造器内部根据是直接调用还是通过子类调用采取不同的动作：

```
class Parent {
    constructor() {
        if (new.target === Parent) {
            console.log( "Parent instantiated" );
        }
        else {
            console.log( "A child instantiated" );
        }
    }
}

class Child extends Parent {}

var a = new Parent();
// Parent instantiated

var b = new Child();
// A child instantiated
```

这里有点微妙，Parent 类定义内部的 constructor() 实际上被给定了类的词法名称（Parent），即使语法暗示这个类是与构造器分立的实体。

对于所有的元编程技术都要小心，不要编写过于机灵的代码，让未来的你或者其他代码维护者难以理解。要小心使用这些技巧。

7.3 公开符号

在 2.13 节中，我们介绍了 ES6 新原生类型 `symbol`。除了在自己的程序中定义符号之外，JavaScript 预先定义了一些内置符号，称为**公开符号**（Well-Known Symbol，WKS）。

定义这些符号主要是为了提供专门的元属性，以便把这些元属性暴露给 JavaScript 程序以获取对 JavaScript 行为更多的控制。

接下来我们会简单介绍一下每个符号和它们的作用。

7.3.1 `Symbol.iterator`

在第 2 章和第 3 章中，我们介绍并使用了符号 `@@iterator`，它是由 `...` 展开和 `for..of` 循环自动使用的。在第 5 章中，我们还看到了 ES6 新特性集合中的 `@@iterator`。

`Symbol.iterator` 表示任意对象上的一个专门位置（属性），语言机制自动在这个位置上寻找一个方法，这个方法构造一个迭代器来消耗这个对象的值。很多对象定义有这个符号的默认值。

然而，也可以通过定义 `Symbol.iterator` 属性为任意对象值定义自己的迭代器逻辑，即使这会覆盖默认的迭代器。这里的元编程特性在于我们定义了一个行为特性，供 JavaScript 其他部分（也就是运算符和循环结构）在处理定义的对象时使用。

考虑：

```
var arr = [4,5,6,7,8,9];

for (var v of arr) {
    console.log( v );
}
// 4 5 6 7 8 9

// 定义一个只在奇数索引值产生值的迭代器
arr[Symbol.iterator] = function*() {
    var idx = 1;
    do {
        yield this[idx];
    } while ((idx += 2) < this.length);
};

for (var v of arr) {
```

```
    console.log( v );
}
// 5 7 9
```

7.3.2 Symbol.toStringTag 与 Symbol.hasInstance

最常见的一个元编程任务，就是在一个值上进行内省来找出它是什么种类，这通常是为了确定其上适合执行何种运算。对于对象来说，最常用的内省技术是 `toString()` 和 `instanceof`。

考虑：

```
function Foo() {}

var a = new Foo();

a.toString();               // [object Object]
a instanceof Foo;           // true
```

在 ES6 中，可以控制这些操作的行为特性：

```
function Foo(greeting) {
    this.greeting = greeting;
}

Foo.prototype[Symbol.toStringTag] = "Foo";

Object.defineProperty( Foo, Symbol.hasInstance, {
    value: function(inst) {
        return inst.greeting == "hello";
    }
} );

var a = new Foo( "hello" ),
    b = new Foo( "world" );

b[Symbol.toStringTag] = "cool";

a.toString();        // [object Foo]
String( b );         // [object cool]

a instanceof Foo;    // true
b instanceof Foo;    // false
```

原型（或实例本身）的 `@@toStringTag` 符号指定了在 `[object ___]` 字符串化时使用的字符串值。

`@@hasInstance` 符号是在构造器函数上的一个方法，接受实例对象值，通过返回 `true` 或 `false` 来指示这个值是否可以被认为是一个实例。

要在一个函数上设定 `@@hasInstance`，必须使用 `Object.defineProperty(..)`，因为 `Function.prototype` 上默认的那一个是 `writable: false`（不可写的）。参见本系列《你不知道的 JavaScript（上卷）》第二部分获取更多信息。

7.3.3 Symbol.species

在 3.4 节中，我们介绍了符号 `@@species`，这个符号控制要生成新实例时，类的内置方法使用哪一个构造器。

最常见的例子是，在创建 `Array` 的子类并想要定义继承的方法（比如 `slice(..)`）时使用哪一个构造器（是 `Array(..)` 还是自定义的子类）。默认情况下，调用 `Array` 子类实例上的 `slice(..)` 会创建这个子类的新实例，坦白说这很可能就是你想要的。

但是，你可以通过覆盖一个类的默认 `@@species` 定义来进行元编程：

```
class Cool {
    // 把@@species推迟到子类
    static get [Symbol.species]() { return this; }

    again() {
        return new this.constructor[Symbol.species]();
    }
}

class Fun extends Cool {}

class Awesome extends Cool {
    // 强制指定@@species为父构造器
    static get [Symbol.species]() { return Cool; }
}

var a = new Fun(),
    b = new Awesome(),
    c = a.again(),
    d = b.again();

c instanceof Fun;       // true
d instanceof Awesome;   // false
d instanceof Cool;      // true
```

就像前面代码中 `Cool` 的定义那样，内置原生构造器上 `Symbol.species` 的默认行为是 `return this`。在用户类上没有默认值，但是就像展示的那样，这个行为特性很容易模拟。

如果需要定义生成新实例的方法，使用 `new this.constructor[Symbol.species](..)` 模式元编程，而不要硬编码 `new this.constructor(..)` 或 `new XYZ(..)`。然后继承类就能够自定义 `Symbol.species` 来控制由哪个构造器产生这些实例。

7.3.4 Symbol.toPrimitive

在本系列《你不知道的 JavaScript（中卷）》第一部分中，我们讨论了抽象类型转换运算 ToPrimitive，它用在对象为了某个操作（比如比较 == 或者相加 +）必须被强制转换为一个原生类型值的时候。在 ES6 之前，没有办法控制这一行为。

而在 ES6 中，在任意对象值上作为属性的符号 @@toPrimitivesymbol 都可以通过指定一个方法来定制这个 ToPrimitive 强制转换。

考虑：

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

arr + 10;               // 1,2,3,4,510

arr[Symbol.toPrimitive] = function(hint) {
    if (hint == "default" || hint == "number") {
        // 求所有数字之和
        return this.reduce( function(acc,curr){
            return acc + curr;
        }, 0 );
    }
};

arr + 10;               // 25
```

Symbol.toPrimitive 方法根据调用 ToPrimitive 的运算期望的类型，会提供一个提示（hint）指定 "string"、"number" 或 "default"（这应该被解释为 "number"）。在前面的代码中，加法 + 运算没有提示（传入 "default"）。而乘法 * 运算提示为 "number"，String(arr) 提示为 "string"。

如果一个对象与另一个非对象值比较，== 运算符调用这个对象上的 ToPrimitive 方法时不指定提示——如果有 @@toPrimitive 方法的话，调用时提示为 "default"。但是，如果比较的两个值都是对象，== 的行为和 === 一样，也就是直接比较其引用。这种情况下完全不会调用 @@toPrimitive。参见本系列《你不知道的 JavaScript（中卷）》第一部分获取关于类型转换和抽象运算的更多信息。

7.3.5 正则表达式符号

对于正则表达式对象，有 4 个公开符号可以被覆盖，它们控制着这些正则表达式在 4 个对应的同名 String.prototype 函数中如何被使用。

- @@match：正则表达式的 Symbol.match 值是一个用于利用给定的正则表达式匹配一个字符串值的部分或全部内容的方法。如果传给 String.prototype.match(..) 一个正则表达式，那么用它来进行模式匹配。

ES6 规范的 21.2.5.6 节中给出了默认匹配算法（https://people.mozilla.org/~jorendorff/es6-draft.html#sec-regexp.prototype-@@match）。你可以覆盖这个默认算法并提供额外的正则匹配特性，比如向后断言（look-behind assertion）。

抽象运算 `isRegExp`（参见 6.5.4 节的“警示”内容）也使用了 `Symbol.match`，来确定对象是否会被用作正则表达式。要强制使得某个对象上的这个检查失败，使它不被当作正则表达式，可以把 `Symbol.match` 值设置为 `false`（或者任何为假的东西）。`* @@replace`：正则表达式的 `Symbol.replace` 值是一个方法，`String.prototype.replace(..)` 用它来替换一个字符串内匹配给定的正则表达式模式的一个或多个字符序列。

ES6 规范的 21.2.5.8 节中给出了替换的默认算法（https://people.mozilla.org/~jorendorff/es6-draft.html#sec-regexp.prototype-@@replace）。

覆盖默认算法的一个很棒的应用是提供额外的 `replacer` 参数选项，比如通过消耗 iterable 来产生连续替换值，支持 `"abaca".replace(/a/g,[1,2,3])` 产生 `"1b2c3"` 这样的形式。`* @@search`：正则表达式的 `Symbol.search` 值是一个方法，`String.prototype.search(..)` 用它来在另一个字符串中搜索一个匹配给定正则表达式的子串。

ES6 规范的 21.2.5.9 节中给出了搜索的默认算法（https://people.mozilla.org/~jorendorff/es6-draft.html#sec-regexp.prototype-@@search）。`* @@split`：正则表达式的 `Symbol.split` 值是一个方法，`String.prototype.split(..)` 用它把字符串在匹配给定正则表达式的分隔符处分割为子串。

ES6 规范的 21.2.5.11 节中给出了默认的分割算法（https://people.mozilla.org/~jorendorff/es6-draft.html#sec-regexp.prototype-@@split）。

如果你不够艺高人胆大的话，就不要覆盖内置正则表达式算法了！ JavaScript 的正则表达式引擎经过高度优化，所以你自己的用户代码很可能会慢上许多。这类元编程简洁强大，但是只应该在确实需要或能带来收益的时候才使用。

7.3.6 Symbol.isConcatSpreadable

符号 `@@isConcatSpreadable` 可以被定义为任意对象（比如数组或其他可迭代对象）的布尔型属性（`Symbol.isConcatSpreadable`），用来指示如果把它传给一个数组的 `concat(..)` 是否应该将其**展开**。

考虑：

```
var a = [1,2,3],
    b = [4,5,6];

b[Symbol.isConcatSpreadable] = false;

[].concat( a, b );      // [1,2,3,[4,5,6]]
```