这就很清楚地表明了，属性符号实际上并不是隐藏或不可访问的，因为总可以通过 `Object.getOwnPropertySymbols(..)` 列表看到它。

内置符号

ES6 支持若干预先定义好的内置符号，它们可以暴露 JavaScript 对象值的各种元特性。但是，这些符号**并不是**像一般设想的那样注册在全局符号表里。

相反，它们作为 `Symbol` 函数对象的属性保存。比如 2.9 节中介绍的 `Symbol.iterator` 值：

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

a[Symbol.iterator];          // 原生函数
```

规范使用 `@@` 前缀记法来指代内置符号，最常用的一些是：`@@iterator`、`@@toStringTag`、`@@toPrimitive`。规范还定义了一些其他符号，但是可能没那么常用。

关于如何在元编程中应用这些内置符号，参见 7.3 节。

2.14 小结

ES6 为 JavaScript 增加了很多新的语法形式，所以要学的太多了！

这些新语法形式中大多数的设计目的都是消除常见编程技巧中的痛点，比如为函数参数设定默认值以及把参数的“其余”部分收集到数组中。解构是一个强有力的工具，用于更精确地表达从数组和嵌套对象中赋值。

而像 `=>` 箭头函数这样的特性看起来似乎是为了使代码更简洁的语法，但实际上它有非常特别的行为特性，应该只在适当的时候使用。

扩展 Unicode 支持、新的正则表达式技巧，甚至新的基本类型 `symbol` 都使 ES6 的语法发展的更加完善。

第 3 章

代码组织

编写 JavaScript 代码是一回事，而合理组织代码则是另一回事。利用通用模式来组织和复用代码显著提高了代码的可读性和可理解性。记住：对于代码来说，和其他开发者交流与提供计算机指令同等重要。

ES6 提供了几个重要的特性，显著改进了以下模式，包括迭代器、生成器、模块和类。

3.1 迭代器

迭代器（iterator）是一个结构化的模式，用于从源以一次一个的方式提取数据。这个模式在编程中已经使用相当长的一段时间了。从很久之前开始，JavaScript 开发者就已经在 JavaScript 程序中自发地设计和实现迭代器，所以这不是一个全新的主题。

ES6 实现的是为迭代器引入一个隐式的标准化接口。JavaScript 很多内建的数据结构现在都提供了实现这个标准的迭代器。为了达到最大化的互操作性，也可以自己构建符合这个标准的迭代器。

迭代器是一种有序的、连续的、基于拉取的用于消耗数据的组织方式。

例如，你可以实现一个工具，在每次请求的时候产生一个新的唯一标识符。也可以在一个固定列表上以轮询的方式产生一个无限值序列。或者也可以把迭代器附着在一个数据库查询结果上，每次迭代拉出一个新行。

虽然在 JavaScript 中这样的用法并不常见，但是迭代器也可以用于以一次一步的方式控制

行为。将生成器（参见 3.2 节）考虑在内，可以把这一点展示的非常清晰，尽管不用生成器也完全可以实现这一功能。

3.1.1 接口

在编写本部分的时候，ES6 第 25.1.1.2 节（https:// people.mozilla.org/~jorendorff/es6-draft.html#sec-iterator-interface）详细解释了 `Iterator` 接口，包括如下要求：

```
Iterator [required]
    next() {method}: 取得下一个IteratorResult
```

有些迭代器还扩展支持两个可选成员：

```
Iterator [optional]
    return() {method}: 停止迭代器并返回IteratorResult
    throw() {method}: 报错并返回IteratorResult
```

`IteratorResult` 接口指定如下：

```
IteratorResult
    value {property}: 当前迭代值或者最终返回值（如果undefined为可选的）
    done {property}: 布尔值，指示完成状态
```

我把这些接口称为隐式的，并不是因为它们没有在规范中显式说明——它们有说明！——而是因为它们没有暴露为代码可以访问的直接对象。在 ES6 中，JavaScript 不支持任何“接口”的概念，所以你自己的代码符合规范只是单纯的惯用法。然而，在 JavaScript 期望迭代器的位置（比如 `for..of` 循环）所提供的东西必须符合这些接口，否则代码会失败。

还有一个 `Iterable` 接口，用来表述必需能够提供生成器的对象：

```
Iterable
    @@iterator() {method}: 产生一个 Iterator
```

回忆一下 2.13.2 节，你会了解到 `@@iterator` 是一个特殊的内置符号，表示可以为这个对象产生迭代器的方法。

`IteratorResult`

`IteratorResult` 接口指定了从任何迭代器操作返回的值必须是下面这种形式的对象：

```
{ value: .. , done: true / false }
```

内置迭代器总是返回这种形式的值，当然如果需要的话，返回值还可以有更多的属性。

例如，自定义迭代器可能在结果对象上增加额外的元数据（比如数据的来源、获取数据的

时间长度、缓存过期时长、下次请求的适当频率，等等）。

严格来说，如果不提供 value 可以被当作是不存在或者未设置，就像值 undefined，那么 value 是可选的。因为访问 res.value 的时候，不管它存在且值为 undefined，还是根本不存在，都会产生 undefined，这个属性的存在 / 缺席更多的是一个实现细节或者优化技术（或者二者兼有），而非功能问题。

3.1.2 next() 迭代

我们来观察一个数组，这是一个 iterable，它产生的迭代器可以消耗其自身值：

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

var it = arr[Symbol.iterator]();

it.next();      // { value: 1, done: false }
it.next();      // { value: 2, done: false }
it.next();      // { value: 3, done: false }

it.next();      // { value: undefined, done: true }
```

每次在这个 arr 值上调用位于 Symbol.iterator（参见第 2 章和第 7 章）的方法时，都会产生一个全新的迭代器。多数结构都是这么实现的，包括所有 JavaScript 内置数据结构。

但像事件队列消费者这样的结构可能只产生一个迭代器（单例模式）。或者某个结构可能在同一时刻只允许唯一的迭代器，要求当前的迭代器完成才能创建下一个迭代器。

前面代码中在提取值 3 的时候，迭代器 it 不会报告 done: true。必须得再次调用 next()，越过数组结尾的值，才能得到完成信号 done: true。虽然直到本小节的后面才会介绍原因，但这样的设计决策通常被认为是最佳实践。

基本字符串值默认也可以迭代：

```
var greeting = "hello world";

var it = greeting[Symbol.iterator]();

it.next();      // { value: "h", done: false }
it.next();      // { value: "e", done: false }
..
```

严格来说，基本值本身不是 iterable，但是感谢“封箱”技术，"hello world" 被强制转换 / 变换为 String 对象封装形式，而这是一个 iterable。参见本系列《你不知道的 JavaScript（中卷）》第一部分可以获取更多细节。

ES6 中还包括几个新的称为集合（参见第 5 章）的数据结构。这些集合不仅本身是 iterable，还提供了 API 方法来产生迭代器，比如：

```
var m = new Map();
m.set( "foo", 42 );
m.set( { cool: true }, "hello world" );

var it1 = m[Symbol.iterator]();
var it2 = m.entries();

it1.next();     // { value: [ "foo", 42 ], done: false }
it2.next();     // { value: [ "foo", 42 ], done: false }
..
```

迭代器的 `next(..)` 方法可以接受一个或多个可选参数。绝大多数内置迭代器没有利用这个功能，尽管生成器的迭代器肯定有（参见 3.2 节）。

通用的惯例是，包括所有内置迭代器，在已经消耗完毕的迭代器上调用 `next(..)` 不会出错，而只是简单地继续返回结果 `{ value: undefined, done: true }`。

3.1.3 可选的 return(..) 和 throw(..)

多数内置迭代器都没有实现可选的迭代器接口——`return(..)` 和 `throw(..)`。然而，在生成器的上下文中它们肯定是有意义的，参见 3.2 节获取更多信息。

`return(..)` 被定义为向迭代器发送一个信号，表明消费者代码已经完毕，不会再从其中提取任何值。这个信号可以用于通知生产者（响应 `next(..)` 调用的迭代器）执行可能需要的清理工作，比如释放 / 关闭网络、数据库或者文件句柄资源。

如果迭代器存在 `return(..)`，并且出现了任何可以自动被解释为异常或者对迭代器消耗的提前终止的条件，就会自动调用 `return(..)`。你也可以手动调用 `return(..)`。

`return(..)` 就像 `next(..)` 一样会返回一个 `IteratorResult` 对象。一般来说，发送给 `return(..)` 的可选值将会在这个 `IteratorResult` 中作为 `value` 返回，但在一些微妙的情况下并非如此。

`throw(..)` 用于向迭代器报告一个异常 / 错误，迭代器针对这个信号的反应可能不同于针对 `return(..)` 意味着的完成信号。和对于 `return(..)` 的反应不一样，它并不一定意味着迭代器的完全停止。

例如，通过生成器迭代器，`throw(..)` 实际上向生成器的停滞执行上下文中插入了一个抛出的异常，这个异常可以用 `try..catch` 捕获。未捕获的 `throw(..)` 异常最终会异常终止生成器迭代器。

通用的惯例是，迭代器不应该在调用 `return(..)` 或者 `thrown(..)` 之后再产生任何值。

3.1.4 迭代器循环

我们在 2.9 节已经介绍过，ES6 的 `for..of` 循环直接消耗一个符合规范的 iterable。

如果一个迭代器也是一个 iterable，那么它可以直接用于 `for..of` 循环。你可以通过为迭代器提供一个 `Symbol.iterator` 方法简单返回这个迭代器本身使它成为 iterable：

```
var it = {
    // 使迭代器it成为iterable
    [Symbol.iterator]() { return this; },

    next() { .. },
    ..
};

it[Symbol.iterator]() === it;     // true
```

现在可以用 `for..of` 循环消耗这个 `it` 迭代器：

```
for (var v of it) {
    console.log( v );
}
```

要彻底理解这样的循环如何工作，可以回顾一下第 2 章 `for..of` 循环的等价 `for` 形式：

```
for (var v, res; (res = it.next()) && !res.done; ) {
    v = res.value;
    console.log( v );
}
```

如果认真观察的话，可以看到每次迭代之前都调用了 `it.next()`，然后查看一下 `res.done`。如果 `res.done` 为 `true`，表达式求值为 `false`，迭代就不会发生。

回忆一下，前面我们建议迭代器一般不应与最终预期的值一起返回 `done: true`。现在能明白其中的原因了吧。

如果迭代器返回 `{ done: true, value: 42 }`，`for..of` 循环会完全丢弃值 `42`，那么这个值就被丢失了。因为这个原因，假定你的迭代器可能会通过 `for..of` 循环或者手动的等价 `for` 形式模式消耗，那么你应该等返回所有的相关迭代值之后，再返回 `done: true` 来标明迭代完毕。

当然，可以有意地把迭代器设计为在返回 done: true 的同时返回一些相关值。但除非已经写了文档表明这一点，以此迫使迭代器的消费者使用不同的迭代模式，而不是像 for..of 或者其等价手动 for 形式暗示的那样，否则不要这么做。

3.1.5 自定义迭代器

除了标准的内置迭代器，你也可以构造自己的迭代器！要使得它们能够与 ES6 的消费者工具（比如，for..of 循环以及 ... 运算符）互操作，所需要做的就是使其遵循适当的接口。

让我们试着构造一个迭代器来产生一个无限斐波纳契序列：

```
var Fib = {
    [Symbol.iterator]() {
        var n1 = 1, n2 = 1;

        return {
            // 使迭代器成为iterable
            [Symbol.iterator]() { return this; },

            next() {
                var current = n2;
                n2 = n1;
                n1 = n1 + current;
                return { value: current, done: false };
            },

            return(v) {
                console.log(
                    "Fibonacci sequence abandoned."
                );
                return { value: v, done: true };
            }
        };
    }
};

for (var v of Fib) {
    console.log( v );

    if (v > 50) break;
}
// 1 1 2 3 5 8 13 21 34 55
// Fibonacci sequence abandoned.
```

如果我们没有插入 break 条件的话，这个 for..of 循环就会无限循环下去，这可能不是你想要的结果。

调用 Fib[Symbol.iterator]() 方法的时候，会返回带有 next() 和 return(..) 方法的迭代器对象。通过放在闭包里的变量 n1 和 n2 维护状态。

接下来考虑一个迭代器，它的设计意图是用来在一系列（也就是一个队列）动作上运行，一次一个条目：

```
var tasks = {
    [Symbol.iterator]() {
        var steps = this.actions.slice();

        return {
            // 使迭代器成为iterable
            [Symbol.iterator]() { return this; },

            next(...args) {
                if (steps.length > 0) {
                   let res = steps.shift()( ...args );
                   return { value: res, done: false };
                }
                else {
                    return { done: true }
                }
            },

            return(v) {
                steps.length = 0;
                return { value: v, done: true };
            }
        };
    },
    actions: []
};
```

tasks 上的迭代器走过 actions 数组属性中找到的函数（如果有的话），然后一次一个执行这些函数，把传入 next(..) 的所有参数传入，将其返回值在标准 IteratorResult 对象中返回。

下面是这个 tasks 队列的一种使用方式：

```
tasks.actions.push(
    function step1(x){
        console.log( "step 1:", x );
        return x * 2;
    },
    function step2(x,y){
        console.log( "step 2:", x, y );
        return x + (y * 2);
    },
    function step3(x,y,z){
        console.log( "step 3:", x, y, z );
        return (x * y) + z;
```

```
    }
);

var it = tasks[Symbol.iterator]();

it.next( 10 );          // step 1: 10
                        // { value: 20, done: false }

it.next( 20, 50 );      // step 2: 20 50
                        // { value:  120, done: false }

it.next( 20, 50, 120 ); // step 3: 20 50 120
                        // { value: 1120, done: false }

it.next();              // { done: true }
```

这种特定的用法强调了迭代器可以作为一个模式来组织功能，而不仅仅是数据。下一小节我们介绍生成器时也可以回顾一下这里。

你甚至可以创造性地定义一个迭代器来表示单个数据上的元操作。举例来说，我们可以为数字定义一个迭代器，默认范围是从 0 到（或者对于负数来说，向下到）关注的数字。

考虑：

```
if (!Number.prototype[Symbol.iterator]) {
    Object.defineProperty(
        Number.prototype,
        Symbol.iterator,
        {

            writable: true,
            configurable: true,
            enumerable: false,
            value: function iterator(){
                var i, inc, done = false, top = +this;

                // 正向还是反向迭代?
                inc = 1 * (top < 0 ? -1 : 1);

                return {
                    // 使得迭代器本身成为iterable!
                    [Symbol.iterator](){ return this; },

                    next() {
                        if (!done) {
                            // 初始迭代总是0
                            if (i == null){
                                i = 0;
                            }
                            // 正向迭代
                            else if (top >= 0) {
                                i = Math.min(top,i + inc);
                            }
```

```
                        // 反向迭代
                        else {
                            i = Math.max(top,i + inc);
                        }

                        // 本次迭代后结束?
                        if (i == top) done = true;

                        return { value: i, done: false };
                    }
                    else {
                        return { done: true };
                    }
                }
            };
        }
    }
);
}
```

那么这种创造性提供了哪些技巧呢?

```
for (var i of 3) {
    console.log( i );
}
// 0 1 2 3

[...-3];                // [0,-1,-2,-3]
```

这是一些有趣的技巧，尽管其实际功效值得商榷。但又一次地，有人可能会奇怪为什么ES6 不把这个微小但可爱的功能作为复活节彩蛋提供呢!

这一点我不能疏于提醒，像我在前面代码中那样扩展原生原型需要格外小心和清醒，以避免隐患。

在这个例子中，与其他代码甚至是未来的 JavaScript 功能冲突的可能性是微乎其微的。但要清醒意识到这么一丝的可能性。还要编写文档为后来者解释你的所作所为。

如果你想了解更多细节，在这篇博客文章（http://blog.getify.com/iterating-es6-numbers/）里我已经解释过这种特殊技术。还有这篇评论（http://blog.getify.com/iterating-es6-numbers/comment-page-1/#comment-535294）甚至建议为字符范围构造类似的技巧。

3.1.6 迭代器消耗

前面已经展示了如何通过 `for..of` 循环一个接一个地消耗迭代器项目，但是还有其他 ES6 结构可以用来消耗迭代器。