目　录

致谢

阅前必读

序

第一章：ES？现在与未来

- 版本
- 转译
- 复习

第二章：语法

- 块儿作用域声明
- 扩散/剩余
- 默认参数值
- 解构
- 对象字面量扩展
- 模板字面量
- 箭头函数
- for..of 循环
- 正则表达式扩展
- 数字字面量扩展
- Unicode
- Symbol
- 复习

第三章：组织

- 迭代器
- Generators
- 模块
- 类
- 复习

第四章：异步流程控制

- Promises
- Generators + Promises
- 复习

第五章：集合

- 类型化数组（TypedArrays）

Maps
WeakMaps
Sets
WeakSets
复习
第六章：新增 API
Array
Object
Math
Number
String
复习
第七章：元编程
函数名
元属性
通用Symbol
代理
Reflect API
特性测试
尾部调用优化（TCO）
复习
第八章：ES6 以后
async function
Object.observe(..)
指数操作符
对象属性与 ...
Array#includes(..)
SIMD
WebAssembly (WASM)
复习
附录A：鸣谢

致谢

当前文档 《你不懂JS：ES6与未来（You Dont Know JS）》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-02-08。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/You-Dont-Know-JS-es6-beyond

书栈官网：http://www.bookstack.cn

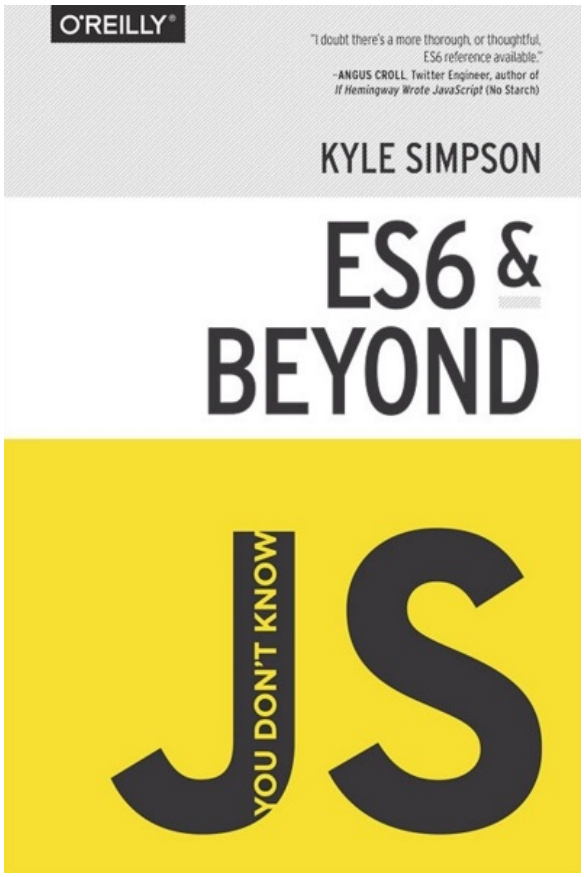
书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

阅前必读

- 你不懂JS：ES6与未来

你不懂JS：ES6与未来



从 **O'Reilly** 购买数字/印刷版

- 第一章：ES？现在与未来
- 第二章：语法
- 第三章：组织
- 第四章：异步流程控制
- 第五章：集合
- 第六章：新增 API
- 第七章：元编程
- 第八章：ES6 之后
- 附录A：鸣谢

序

- 你不懂JS：ES6 与未来
- 序

你不懂JS：ES6 与未来

序

Kyle Simpson 是一个严谨的实用主义者。

我想不出比这更高的赞美。对我来说，这是一个软件开发者必须具备的两个最重要的素质。是的：必须，不是 应当。将 JavaScript 编程语言层层梳理，并将它们用易懂而且有意的部分表现出来，Kyle 的这种敏锐的能力无人能出其右。

对于 你不懂*JS* 系列的读者来说 *ES6* 与未来 使人感到十分熟悉：他们将深深地沉浸在从明显到非常微妙的每一件事中 —— 揭示那些要么被想当然地接受、要么甚至从未被考虑过的语义。至今为止，你不懂*JS* 系列丛书已经向它的读者们讲解了他们至少在某种程度上熟悉的内容。他们不是见过就是听说过那些主题很重要；也许他们甚至曾经有过相关的经验。而这一卷讲解了只有在很少一部分的 JavaScript 开发者社区中才曝光过的内容：在 ECMAScript 2015 语言规范中给这门语言引入的革命性改变。

在过去的几年中，我目睹了 Kyle 不知疲倦地努力学习这些内容，直到只有少数专业人士才能与之媲美的专家级水准。这真是一个壮举，试想就在他撰写的时候，语言规范的文档还没有正式发布哩！但我说的是真的，而且我读了 Kyle 为这本书写的每一个字。我追随着每一次修改，而且每一次它的内容只会变得更好，并提供更深一层的理解。

这本书会将你暴露在新的与未知的事物中来震撼你理解的感官。它意在通过赐予你新的力量来使你的知识更上一个台阶。它存在的目的是为了给你自信，去完全地拥抱 JavaScript 编程的下一个新纪元。

Rick Waldron

@rwaldron

Bocoup 的开放 Web 工程师

Ecma/TC39 jQuery 代表

第一章：ES？现在与未来

- 第一章：ES？现在与未来

第一章：ES？现在与未来

在你一头扎进这本书之前，你应当可以熟练地使用（在本书写作时）最近版本的JavaScript，也就是通常所说的 *ES5*（技术上讲是ES 5.1）。这里，我们打算好好谈谈即将到来的 *ES6*，同时放眼未来去看看JS将会如何继续进化。

如果你还在JavaScript上寻找信心，我强烈推荐你首先读一读本系列的其他书目：

- 入门与进阶：你是编程和JS的新手吗？这就是你在开启学习的旅程前需要查看的路线图。
- 作用域与闭包：你知道JS的词法作用域是基于编译器（不是解释器！）语义的吗？你能解释闭包是如何成为词法作用域和函数作为值的直接结果的吗？
- *this*与对象原型：你能复述`this`绑定的四个简单规则吗？你有没有曾经在JS中对付着去山寨“类”，而不是采取更简单的“行为委托”设计模式？你听说过 链接到其他对象的对象（OOLO）吗？
- 类型与文法：你知道JS中的内建类型吗？更重要的是，你知道如何在类型之间正确且安全地使用强制转换吗？你对JS文法/语法的微妙之处感到有多习惯？
- 异步与性能：你还在使用回调管理你的异步处理吗？你能解释promise是为什么/如何解决了“回调地狱”的吗？你知道如何使用generator来改进异步代码的易读性吗？到底是什么构成了JS程序和独立操作的成熟优化？

如果你已经读过了这些书目而且对它们涵盖的内容感到十分轻松，那么现在是时候让我们深入JS的进化过程来探索所有即将到来的以及未来会发生的改变了。

与ES5不同，ES6不仅仅是向语言添加的一组不算太多的新API。它包含大量的新的语法形式，其中的一些你可能会花上相当一段时间才能适应。还有几种新的组织形式和为各种数据类型添加的新API。

对这门语言来说ES6十分激进。就算你认为你懂得ES5的JS，ES6也满是 你还不懂的 新东西，所以做好准备！这本书探索所有你需要迅速掌握的ES6主要主题，并且窥见一下那些你应当注意的正在步入正轨的未来特性。

警告： 这本书中的所有代码都假定运行在ES6+的环境中。在写作本书时，浏览器和JS环境（比如Node.js）对ES6的支持相当不同，因此你的感觉可能将会不同。

- 版本
- 转译
- 复习

版本

JavaScript标准在官方上被称为“ECMAScript”（缩写为“ES”），而且直到最近才刚刚完全采用顺序数字来标记版本（例如，“5”代表“第五版”）。

最早的版本，ES1和ES2，并不广为人知也没有大范围地被实现。ES3是JavaScript第一次广泛传播的基准线，并且构成了像IE6-8和更早的Android 2.x移动浏览器的JavaScript标准。由于一些超出我们讨论范围的政治原因，命运多舛的ES4从未问世。

在2009年，ES5正式定稿（在2011年出现了ES5.1），它在浏览器的现代革新和爆发性增长（比如Firefox，Chrome，Opera，Safari，和其他许多）中广泛传播，并作为JS标准稳定下来。

预计下一个版本的JS（从2013年到2014年和之后的2015年中的内容），在人们的讨论中显然地经常被称为ES6。

然而，在ES6规范的晚些时候，有建议提及未来的版本号也许会切换到编年制，比如用ES2016（也叫ES7）来指代在2016年末之前被定稿的任何版本。有些人对此持否定意见，但是相对于后来的ES2015来说，ES6将很可能继续维持它占统治地位的影响力。可是，ES2016事实上可能标志了新的编年制。

还可以看到，JS进化的频度即使与一年一度的定版相比都要快得多。只要一个想法开始标准化讨论的进程，浏览器就开始为这种特性建造原型，而且早期的采用者就开始在代码中进行实验。

通常在一个特性被盖上官方承认的印章以前，由于这些早期的引擎/工具的原型它实际上已经被标准化了。所以也可以认为未来的JS版本将是一个特性一个特性的更新，而非一组主要特性的随意集合的更新（就像现在），也不是一年一年的更新（就像可能将变成的那样）。

简而言之，版本号不再那么重要了，JavaScript开始变得更像一个常青的，活的标准。应对它的最佳方法是，举例来说，不再将你的代码库认为是“基于ES6”的，而是考虑它支持的一个个特性。

转译

- 填补（Shims/Polyfills）

由于特性的快速进化，给开发者们造成了一个糟糕的问题，他们强烈地渴望立即使用新特性，而同时被被现实打脸 —— 他们的网站/app需要支持那些不支持这些特性的老版本浏览器。

在整个行业中ES5的方式似乎已经无力回天了，它典型的思维模式是，代码库等待几乎所有的前ES5环境从它们的支持谱系中除名之后才开始采用ES5。结果呢，许多人最近（在本书写作时）才开始采用 `strict` 模式这样的东西，而它早在五年前就在ES5中定稿了。

对于JS生态系统的未来来说，等待和落后于语言规范那么多年被广泛地认为是一种有害的方式。所有负责推动语言演进的人都渴望这样的事情；只要新的特性和模式以规范的形式稳定下来，并且浏览器有机会实现它们，开发者就开始基于这些新的特性和模式进行编码。

那么我们如何解决这个看起来似乎矛盾的问题？答案是工具，特别是一种称为 转译（*transpiling*） 的技术（转换+编译）。大致上，它的想法是使用一种特殊的工具将你的ES6代码转换为可以在ES5环境中工作的等价物（或近似物！）。

例如，考虑属性定义缩写（见第二章的“对象字面扩展”）。这是ES6的形式：

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

var obj = {
    foo         // 意思是 `foo: foo`
};

obj.foo;    // [1,2,3]
```

这（大致）是它如何被转译：

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

var obj = {
    foo: foo
};

obj.foo;    // [1,2,3]
```

这是一个微小但令人高兴的转换，它让我们在一个对象字面声明中将 `foo: foo` 缩写为 `foo`，如果名称相同的话。

转译器为你实施这些变形，这个过程通常是构建工作流的一个步骤 —— 与你进行linting，压缩，和其他类似操作相似。

填补（Shims/Polyfills）

不是所有的ES6新特性都需要转译器。填补（也叫shims）是一种模式，在可能的情况下，它为一个新环境的行为定义一个可以在旧环境中运行的等价行为。语法是不能填补的，但是API经常是可以的。

例如，`Object.is(..)` 是一个用来检查两个值严格等价性的新工具，它不带有 `===` 对于 `NaN` 和 `-0` 值的那种微妙的例外。`Object.is(..)` 的填补相当简单：

```
if (!Object.is) {
    Object.is = function(v1, v2) {
        // 测试 `-0`
        if (v1 === 0 && v2 === 0) {
            return 1 / v1 === 1 / v2;
        }
        // 测试 `NaN`
        if (v1 !== v1) {
            return v2 !== v2;
        }
        // 其他的一切情况
        return v1 === v2;
    };
}
```

提示：注意外部的 `if` 语句守护性地包围着填补的内容。这是一个重要的细节，它意味着这个代码段仅仅是为这个API还未定义的老环境而定义的后备行为；你想要覆盖既存API的情况是非常少见的。

有一个被称为“ES6 Shim”（ https://github.com/paulmillr/es6-shim/ ）的了不起的ES6填补集合，你绝对应该将它采纳为任何新JS项目的标准组成部分！

看起来JS将会继续一往无前的进化下去，同时浏览器也会持续地小步迭代以支持新特性，而不是大块大块地更新。所以跟上时代的最佳策略就是在你的代码库中引入填补，并在你的构建流程中引入一个转译器步骤，现在就开始习惯新的现实。

如果你决定维持现状，等待不支持新特性的所有浏览器都消失才开始使用新特性，那么你将总是落后于时代。你将可悲地错过所有新发明的设计 —— 而它们使编写JavaScript更有效，更高效，而且更健壮。

复习

第二章：语法

- 第二章：语法

第二章：语法

如果你曾经或多或少地写过JS，那么你很可能对它的语法感到十分熟悉。当然有一些奇怪之处，但是总体来讲这是一种与其他语言有很多相似之处的，相当合理而且直接的语法。

然而，ES6增加了好几种需要费些功夫才能习惯的新语法形式。在这一章中，我们将遍历它们来看看葫芦里到底卖的什么药。

提示： 在写作本书时，这本书中所讨论的特性中的一些已经被各种浏览器（Firefox，Chrome，等等）实现了，但是有一些仅仅被实现了一部分，而另一些根本就没实现。如果直接尝试这些例子，你的体验可能会夹杂着三种情况。如果是这样，就使用转译器尝试吧，这些特性中的大多数都被那些工具涵盖了。ES6Fiddle（http://www.es6fiddle.net/）是一个了不起的尝试ES6的游乐场，简单易用，它是一个Babel转译器的在线REPL（http://babeljs.io/repl/ ）。

- 块儿作用域声明
- 扩散/剩余
- 默认参数值
- 解构
- 对象字面量扩展
- 模板字面量
- 箭头函数
- for..of 循环
- 正则表达式扩展
- 数字字面量扩展
- Unicode
- Symbol
- 复习

块儿作用域声明

- 块儿作用域声明
  - `let`声明
    - `let` + `for`
  - `const`声明
    - `const`用还是不用
  - 块儿作用域的函数

块儿作用域声明

你可能知道在JavaScript中变量作用域的基本单位总是 `function` 。如果你需要创建一个作用域的块儿，除了普通的函数声明以外最流行的方法就是使用立即被调用的函数表达式（IIFE）。例如：

```
var a = 2;

(function IIFE(){
    var a = 3;
    console.log( a );    // 3
})();

console.log( a );        // 2
```

`let`声明

但是，现在我们可以创建绑定到任意的块儿上的声明了，它（勿庸置疑地）称为 块儿作用域。这意味着一对 `{ .. }` 就是我们用来创建一个作用域所需要的全部。 `var` 总是声明附着在外围函数（或者全局，如果在顶层的话）上的变量，取而代之的是，使用 `let` ：

```
var a = 2;

{
    let a = 3;
    console.log( a );    // 3
}

console.log( a );        // 2
```

迄今为止，在JS中使用独立的 `{ .. }` 块儿不是很常见，也不是惯用模式，但它总是合法的。而且那些来自拥有 块儿作用域 的语言的开发者将很容易认出这种模式。

我相信使用一个专门的 `{ .. }` 块儿是创建块儿作用域变量的最佳方法。但是，你应该总是将 `let` 声明放在块儿的最顶端。如果你有多于一个的声明，我推荐只使用一个 `let` 。

从文体上说，我甚至喜欢将 `let` 放在与开放的 `{` 的同一行中，以便更清楚地表示这个块儿的目的仅仅是为了这些变量声明作用域。

```
{    let a = 2, b, c;
    // ..
}
```

它现在看起来很奇怪，而且不大可能与其他大多数ES6文献中推荐的文法吻合。但我的疯狂是有原因的。

这是另一种实验性的（不是标准化的） `let` 声明形式，称为 `let` 块儿，看起来就像这样：

```
let (a = 2, b, c) {
    // ..
}
```

我称这种形式为 明确的 块儿作用域，而与 `var` 相似的 `let` 声明形式更像是 隐含的，因为它在某种意义上劫持了它所处的 `{ .. }` 。一般来说开发者们认为 明确的 机制要比 隐含的 机制更好一些，我主张这种情况就是这样的情况之一。

如果你比较前面两个形式的代码段，它们非常相似，而且我个人认为两种形式都有资格在文体上称为明确的 块儿作用域。不幸的是，两者中最 明确的 `let (..) { .. }` 形式没有被ES6所采用。它可能会在后ES6时代被重新提起，但我想目前为止前者是我们的最佳选择。

为了增强对 `let ..` 声明的 隐含 性质的理解，考虑一下这些用法：

```
let a = 2;

if (a > 1) {
    let b = a * 3;
    console.log( b );        // 6

    for (let i = a; i <= b; i++) {
        let j = i + 10;
        console.log( j );
    }
    // 12 13 14 15 16

    let c = a + b;
    console.log( c );        // 8
}
```

不要回头去看这个代码段，小测验：哪些变量仅存在于 `if` 语句内部？哪些变量仅存在于 `for` 循环内部？

答案： `if` 语句包含块儿作用域变量 `b` 和 `c` ，而 `for` 循环包含块儿作用域变量 `i` 和 `j` 。

你有任何迟疑吗？ `i` 没有被加入外围的 `if` 语句的作用域让你惊讶吗？思维上的停顿和疑问 —— 我称之为“思维税” —— 不仅源自于 `let` 机制对我们来说是新东西，还因为它是 隐含的。

还有一个灾难是 `let c = ..` 声明出现在作用域中太过靠下的地方。传统的被 `var` 声明的变量，无论它们出现在何处，都会被附着在整个外围的函数作用域中；与此不同的是， `let` 声明附着在块儿作用域，而且在它们出现在块儿中之前是不会被初始化的。

在一个 `let ..` 声明/初始化之前访问一个用 `let` 声明的变量会导致一个错误，而对于 `var` 声明来说这个顺序无关紧要（除了文体上的区别）。

考虑如下代码：

```
{
    console.log( a );    // undefined
    console.log( b );    // ReferenceError!

    var a;
    let b;
}
```

警告： 这个由于过早访问被 `let` 声明的引用而引起的 `ReferenceError` 在技术上称为一个 临时死区（*Temporal Dead Zone —— TDZ*） 错误 —— 你在访问一个已经被声明但还没被初始化的变量。这将不是我们唯一能够见到TDZ错误的地方 —— 在ES6中它们会在几种地方意外地发生。另外，注意“初始化”并不要求在你的代码中明确地赋一个值，比如 `let b;` 是完全合法的。一个在声明时没有被赋值的变量被认为已经被赋予了 `undefined` 值，所以 `let b;` 和 `let b = undefined;` 是一样的。无论是否明确赋值，在 `let b` 语句运行之前你都不能访问 `b` 。

最后一个坑：对于TDZ变量和未声明的（或声明的！）变量， `typeof` 的行为是不同的。例如：

```
{
    // `a` 没有被声明
    if (typeof a === "undefined") {
        console.log( "cool" );
    }

    // `b` 被声明了，但位于它的TDZ中
    if (typeof b === "undefined") {        // ReferenceError!
        // ..
    }

    // ..
```

```
13.
14.     let b;
15. }
```

`a` 没有被声明，所以 `typeof` 是检查它是否存在的唯一安全的方法。但是 `typeof b` 抛出了TDZ错误，因为在代码下面很远的地方偶然出现了一个 `let b` 声明。噢。

现在你应当清楚为什么我坚持认为所有的 `let` 声明都应该位于它们作用域的顶部了。这完全避免了偶然过早访问的错误。当你观察一个块儿，或任何块儿的开始部分时，它还更 明确 地指出这个块儿中含有什么变量。

你的块儿（ `if` 语句， `while` 循环，等等）不一定要与作用域行为共享它们原有的行为。

这种明确性要由你负责，由你用毅力来维护，它将为你省去许多重构时的头疼和后续的麻烦。

注意： 更多关于 `let` 和块儿作用域的信息，参见本系列的 作用域与闭包 的第三章。

`let` + `for`

我偏好 明确 形式的 `let` 声明块儿，但对此的唯一例外是出现在 `for` 循环头部的 `let` 。这里的原因看起来很微妙，但我相信它是更重要的ES6特性中的一个。

考虑如下代码：

```
1. var funcs = [];
2.
3. for (let i = 0; i < 5; i++) {
4.     funcs.push( function(){
5.         console.log( i );
6.     } );
7. }
8.
9. funcs[3]();        // 3
```

在 `for` 头部中的 `let i` 不仅是为 `for` 循环本身声明了一个 `i` ，而且它为循环的每一次迭代都重新声明了一个新的 `i` 。这意味着在循环迭代内部创建的闭包都分别引用着那些在每次迭代中创建的变量，正如你期望的那样。

如果你尝试在这段相同代码的 `for` 循环头部使用 `var i` ，那么你会得到 `5` 而不是 `3` ，因为在被引用的外部作用域中只有一个 `i` ，而不是为每次迭代的函数都有一个 `i` 被引用。

你也可以稍稍繁冗地实现相同的东西：

```
1. var funcs = [];
2.
```

```
for (var i = 0; i < 5; i++) {
    let j = i;
    funcs.push( function(){
        console.log( j );
    } );
}

funcs[3]();        // 3
```

在这里，我们强制地为每次迭代都创建一个新的 `j` ，然后闭包以相同的方式工作。我喜欢前一种形式；那种额外的特殊能力正是我支持 `for(let .. ) ..` 形式的原因。可能有人会争论说它有点儿 隐晦，但是对我的口味来说，它足够 明确 了，也足够有用。

`let` 在 `for..in` 和 `for..of` （参见“ `for..of` 循环”）循环中也以形同的方式工作。

`const` 声明

还有另一种需要考虑的块儿作用域声明： `const` ，它创建 常量。

到底什么是一个常量？它是一个在初始值被设定后就成为只读的变量。考虑如下代码：

```
{
    const a = 2;
    console.log( a );    // 2

    a = 3;                // TypeError!
}
```

变量持有的值一旦在声明时被设定就不允许你改变了。一个 `const` 声明必须拥有一个明确的初始化。如果想要一个持有 `undefined` 值的 常量，你必须声明 `const a = undefined` 来得到它。

常量不是一个作用于值本身的制约，而是作用于变量对这个值的赋值。换句话说，值不会因为 `const` 而冻结或不可变，只是它的赋值被冻结了。如果这个值是一个复杂值，比如对象或数组，那么这个值的内容仍然是可以被修改的：

```
{
    const a = [1,2,3];
    a.push( 4 );
    console.log( a );        // [1,2,3,4]

    a = 42;                    // TypeError!
}
```

变量 `a` 实际上没有持有一个恒定的数组；而是持有一个指向数组的恒定的引用。数组本身可以自由

变化。

警告： 将一个对象或数组作为常量赋值意味着这个值在常量的词法作用域消失以前是不能够被垃圾回收的，因为指向这个值的引用是永远不能解除的。这可能是你期望的，但如果不是你就要小心！

实质上， `const` 声明强制实行了我们许多年来在代码中用文体来表明的东西：我们声明一个名称全由大写字母组成的变量并赋予它某些字面值，我们小心照看它以使它永不改变。 `var` 赋值没有强制性，但是现在 `const` 赋值上有了，它可以帮你发现不经意的改变。

`const` 可以 被用于 `for` ， `for..in` ，和 `for..of` 循环（参见“ `for..of` 循环”）的变量声明。然而，如果有任何重新赋值的企图，一个错误就会被抛出，例如在 `for` 循环中常见的 `i++` 子句。

`const` 用还是不用

有些流传的猜测认为在特定的场景下，与 `let` 或 `var` 相比一个 `const` 可能会被JS引擎进行更多的优化。理论上，引擎可以更容易地知道变量的值/类型将永远不会改变，所以它可以免除一些可能的追踪工作。

无论 `const` 在这方面是否真的有帮助，还是这仅仅是我们的幻想和直觉，你要做的更重要的决定是你是否打算使用常量的行为。记住：源代码扮演的一个最重要的角色是为了明确地交流你的意图是什么，不仅是与你自己，而且还是与未来的你和其他的代码协作者。

一些开发者喜欢在一开始将每个变量都声明为一个 `const` ，然后当它的值在代码中有必要发生变化的时候将声明放松至一个 `let` 。这是一个有趣的角度，但是不清楚这是否真正能够改善代码的可读性或可推理性。

就像许多人认为的那样，它不是一种真正的 保护，因为任何后来的想要改变一个 `const` 值的开发者都可以盲目地将声明从 `const` 改为 `let` 。它至多是防止意外的改变。但是同样地，除了我们的直觉和感觉以外，似乎没有客观和明确的标准可以衡量什么构成了“意外”或预防措施。这与类型强制上的思维模式类似。

我的建议：为了避免潜在的令人糊涂的代码，仅将 `const` 用于那些你有意地并且明显地标识为不会改变的变量。换言之，不要为了代码行为而 依靠 `const` ，而是在为了意图可以被清楚地表明时，将它作为一个表明意图的工具。

块儿作用域的函数

从ES6开始，发生在块儿内部的函数声明现在被明确规定属于那个块儿的作用域。在ES6之前，语言规范没有要求这一点，但是许多实现不管怎样都是这么做的。所以现在语言规范和现实吻合了。

考虑如下代码：

```
1. {
```

```
    foo();                      // 好用！

    function foo() {
        // ..
    }
}

foo();                          // ReferenceError
```

函数 `foo()` 是在 `{ .. }` 块儿内部被声明的，由于ES6的原因它是属于那里的块儿作用域的。所以在那个块儿的外部是不可用的。但是还要注意它在块儿里面被“提升”了，这与早先提到的遭受TDZ错误陷阱的 `let` 声明是相反的。

如果你以前曾经写过这样的代码，并依赖于老旧的非块儿作用域行为的话，那么函数声明的块儿作用域可能是一个问题：

```
if (something) {
    function foo() {
        console.log( "1" );
    }
}
else {
    function foo() {
        console.log( "2" );
    }
}

foo();        // ??
```

在前ES6环境下，无论 `something` 的值是什么 `foo()` 都将会打印 `"2"` ，因为两个函数声明被提升到了块儿的顶端，而且总是第二个有效。

在ES6中，最后一行将抛出一个 `ReferenceError` 。

扩散/剩余

默认参数值

解构

- 解构
  - 对象属性赋值模式
  - 不仅是声明
  - 重复赋值
    - 解构赋值表达式
  - 太多，太少，正合适
  - 默认值赋值
  - 嵌套解构
  - 参数解构
    - 解构默认值 + 参数默认值
    - 嵌套默认值：解构与重构

解构

ES6引入了一个称为 解构 的新语法特性，如果你将它考虑为 结构化赋值 那么它令人困惑的程度可能会小一些。为了理解它的含义，考虑如下代码：

```
function foo() {
    return [1,2,3];
}

var tmp = foo(),
    a = tmp[0], b = tmp[1], c = tmp[2];

console.log( a, b, c );                // 1 2 3
```

如你所见，我们创建了一个手动赋值：从 `foo()` 返回的数组中的值到个别的变量 `a` ， `b` ，和 `c` ，而且这么做我们就（不幸地）需要 `tmp` 变量。

相似地，我们也可以用对象这么做：

```
function bar() {
    return {
        x: 4,
        y: 5,
        z: 6
    };
}

var tmp = bar(),
```

```
    x = tmp.x, y = tmp.y, z = tmp.z;

console.log( x, y, z );               // 4 5 6
```

属性值 `tmp.x` 被赋值给变量 `x` ， `tmp.y` 到 `y` 和 `tmp.z` 到 `z` 也一样。

从一个数组中取得索引的值，或从一个对象中取得属性并手动赋值可以被认为是 结构化赋值。ES6为解构 增加了一种专门的语法，具体地称为 数组解构 和 对象结构。这种语法消灭了前一个代码段中对变量 `tmp` 的需要，使它们更加干净。考虑如下代码：

```
var [ a, b, c ] = foo();
var { x: x, y: y, z: z } = bar();

console.log( a, b, c );               // 1 2 3
console.log( x, y, z );               // 4 5 6
```

你很可能更加习惯于看到像 `[a,b,c]` 这样的东西出现在一个 `=` 赋值的右手边的语法，即作为要被赋予的值。

解构对称地翻转了这个模式，所以在 `=` 赋值左手边的 `[a,b,c]` 被看作是为了将右手边的数组拆解为分离的变量赋值的某种“模式”。

类似地， `{ x: x, y: y, z: z }` 指明了一种“模式”把来自于 `bar()` 的对象拆解为分离的变量赋值。

对象属性赋值模式

让我们深入前一个代码段中的 `{ x: x, .. }` 语法。如果属性名与你想要声明的变量名一致，你实际上可以缩写这个语法：

```
var { x, y, z } = bar();

console.log( x, y, z );               // 4 5 6
```

很酷，对吧？

但 `{ x, .. }` 是省略了 `x:` 部分还是省略了 `: x` 部分？当我们使用这种缩写语法时，我们实际上省略了 `x:` 部分。这看起来可能不是一个重要的细节，但是一会儿你就会了解它的重要性。

如果你能写缩写形式，那为什么你还要写出更长的形式呢？因为更长的形式事实上允许你将一个属性赋值给一个不同的变量名称，这有时很有用：

```
var { x: bam, y: baz, z: bap } = bar();

console.log( bam, baz, bap );       // 4 5 6
```

```
4. console.log( x, y, z );                // ReferenceError
```

关于这种对象结构形式有一个微妙但超级重要的怪异之处需要理解。为了展示为什么它可能是一个你需要注意的坑，让我们考虑一下普通对象字面量的“模式”是如何被指定的：

```
1. var X = 10, Y = 20;
2.
3. var o = { a: X, b: Y };
4.
5. console.log( o.a, o.b );           // 10 20
```

在 `{ a: X, b: Y }` 中，我们知道 `a` 是对象属性，而 `X` 是被赋值给它的源值。换句话说，它的语义模式是 `目标: 源` ，或者更明显地， `属性别名: 值` 。我们能直观地明白这一点，因为它和 `=` 赋值是一样的，而它的模式就是 `目标 = 源` 。

然而，当你使用对象解构赋值时 —— 也就是，将看起来像是对象字面量的 `{ .. }` 语法放在 `=` 操作符的左手边 —— 你反转了这个 `目标: 源` 的模式。

回想一下：

```
1. var { x: bam, y: baz, z: bap } = bar();
```

这里面对称的模式是 `源: 目标` （或者 `值: 属性别名` ）。 `x: bam` 意味着属性 `x` 是源值而 `bam` 是被赋值的目标变量。换句话说，对象字面量是 `target <-- source` ，而对象解构赋值是 `source --> target` 。看到它是如何反转的了吗？

有另外一种考虑这种语法的方式，可能有助于缓和这种困惑。考虑如下代码：

```
1. var aa = 10, bb = 20;
2.
3. var o = { x: aa, y: bb };
4. var     { x: AA, y: BB } = o;
5.
6. console.log( AA, BB );             // 10 20
```

在 `{ x: aa, y: bb }` 这一行中， `x` 和 `y` 代表对象属性。在 `{ x: AA, y: BB }` 这一行， `x` 和 `y` 也 代表对象属性。

还记得刚才我是如何断言 `{ x, .. }` 省去了 `x:` 部分的吗？在这两行中，如果你在代码段中擦掉 `x:` 和 `y:` 部分，仅留下 `aa, bb` 和 `AA, BB` ，它的效果 —— 从概念上讲，实际上不能 —— 将是从 `aa` 赋值到 `AA` 和从 `bb` 赋值到 `BB` 。

所以，这种平行性也许有助于解释为什么对于这种ES6特性，语法模式被故意地反转了。

注意： 对于解构赋值来说我更喜欢它的语法是 `{ AA: x , BB: y }` ，因为那样的话可以在两种用法中一致地使用我们更熟悉的 `target: source` 模式。唉，我已经被迫训练自己的大脑去习惯这种反转了，就像一些读者也不得不去做的那样。

不仅是声明

至此，我们一直将解构赋值与 `var` 声明（当然，它们也可以使用 `let` 和 `const` ）一起使用，但是解构是一种一般意义上的赋值操作，不仅是一种声明。

考虑如下代码：

```
var a, b, c, x, y, z;

[a,b,c] = foo();
( { x, y, z } = bar() );

console.log( a, b, c );               // 1 2 3
console.log( x, y, z );               // 4 5 6
```

变量可以是已经被定义好的，然后解构仅仅负责赋值，正如我们已经看到的那样。

注意： 特别对于对象解构形式来说，当我们省略了 `var` / `let` / `const` 声明符时，就必须将整个赋值表达式包含在 `()` 中，因为如果不这样做的话左手边作为语句第一个元素的 `{ .. }` 将被视为一个语句块儿而不是一个对象。

事实上，变量表达式（ `a` ， `y` ，等等）不必是一个变量标识符。任何合法的赋值表达式都是允许的。例如：

```
var o = {};

[o.a, o.b, o.c] = foo();
( { x: o.x, y: o.y, z: o.z } = bar() );

console.log( o.a, o.b, o.c );         // 1 2 3
console.log( o.x, o.y, o.z );         // 4 5 6
```

你甚至可以在解构中使用计算型属性名。考虑如下代码：

```
var which = "x",
    o = {};

( { [which]: o[which] } = bar() );

console.log( o.x );                   // 4
```

`[which]:` 的部分是计算型属性名，它的结果是 `x` —— 将从当前的对象中拆解出来作为赋值的源头的属性。 `o[which]` 的部分只是一个普通的对象键引用，作为赋值的目标来说它与 `o.x` 是等价的。

你可以使用普通的赋值来创建对象映射/变形，例如：

```
var o1 = { a: 1, b: 2, c: 3 },
    o2 = {};

( { a: o2.x, b: o2.y, c: o2.z } = o1 );

console.log( o2.x, o2.y, o2.z );    // 1 2 3
```

或者你可以将对象映射进一个数组，例如：

```
var o1 = { a: 1, b: 2, c: 3 },
    a2 = [];

( { a: a2[0], b: a2[1], c: a2[2] } = o1 );

console.log( a2 );                  // [1,2,3]
```

或者从另一个方向：

```
var a1 = [ 1, 2, 3 ],
    o2 = {};

[ o2.a, o2.b, o2.c ] = a1;

console.log( o2.a, o2.b, o2.c );    // 1 2 3
```

或者你可以将一个数组重排到另一个数组中：

```
var a1 = [ 1, 2, 3 ],
    a2 = [];

[ a2[2], a2[0], a2[1] ] = a1;

console.log( a2 );                  // [2,3,1]
```

你甚至可以不使用临时变量来解决传统的“交换两个变量”的问题：

```
var x = 10, y = 20;

```

```
[ y, x ] = [ x, y ];

console.log( x, y );                // 20 10
```

警告： 小心：你不应该将声明和赋值混在一起，除非你想要所有的赋值表达式 也 被视为声明。否则，你会得到一个语法错误。这就是为什么在刚才的例子中我必须将 `var a2 = []` 与 `[ a2[0], .. ] = ..` 解构赋值分开做。尝试 `var [ a2[0], .. ] = ..` 没有任何意义，因为 `a2[0]` 不是一个合法的声明标识符；很显然它也不能隐含地创建一个 `var a2 = []` 声明来使用。

重复赋值

对象解构形式允许源属性（持有任意值的类型）被罗列多次。例如：

```
var { a: X, a: Y } = { a: 1 };

X;    // 1
Y;    // 1
```

这意味着你既可以解构一个子对象/数组属性，也可以捕获这个子对象/数组的值本身。考虑如下代码：

```
var { a: { x: X, x: Y }, a } = { a: { x: 1 } };

X;    // 1
Y;    // 1
a;    // { x: 1 }

( { a: X, a: Y, a: [ Z ] } = { a: [ 1 ] } );

X.push( 2 );
Y[0] = 10;

X;    // [10,2]
Y;    // [10,2]
Z;    // 1
```

关于解构有一句话要提醒：像我们到目前为止的讨论中做的那样，将所有的解构赋值都罗列在单独一行中的方式可能很诱人。然而，一个好得多的主意是使用恰当的缩进将解构赋值的模式分散在多行中 —— 和你在JSON或对象字面量中做的事非常相似 —— 为了可读性。

```
// 很难读懂：
var { a: { b: [ c, d ], e: { f } }, g } = obj;

// 好一些：
```

```
var {
    a: {
        b: [ c, d ],
        e: { f }
    },
    g
} = obj;
```

记住：解构的目的不仅是为了少打些字，更多是为了声明可读性

解构赋值表达式

带有对象或数组解构的赋值表达式的完成值是右手边完整的对象/数组值。考虑如下代码：

```
var o = { a:1, b:2, c:3 },
    a, b, c, p;

p = { a, b, c } = o;

console.log( a, b, c );             // 1 2 3
p === o;                        // true
```

在前面的代码段中， `p` 被赋值为对象 `o` 的引用，而不是 `a` ， `b` ，或 `c` 的值。数组解构也是一样：

```
var o = [1,2,3],
    a, b, c, p;

p = [ a, b, c ] = o;

console.log( a, b, c );             // 1 2 3
p === o;                        // true
```

通过将这个对象/数组作为完成值传递下去，你可将解构赋值表达式链接在一起：

```
var o = { a:1, b:2, c:3 },
    p = [4,5,6],
    a, b, c, x, y, z;

( {a} = {b,c} = o );
[x,y] = [z] = p;

console.log( a, b, c );             // 1 2 3
console.log( x, y, z );             // 4 5 4
```

太多，太少，正合适

对于数组解构赋值和对象解构赋值两者来说，你不必分配所有出现的值。例如：

```
1. var [,b] = foo();
2. var { x, z } = bar();
3.
4. console.log( b, x, z );              // 2 4 6
```

从 `foo()` 返回的值 `1` 和 `3` 被丢弃了，从 `bar()` 返回的值 `5` 也是。

相似地，如果你试着分配比你正在解构/拆解的值要多的值时，它们会如你所想的那样安静地退回到 `undefined`：

```
1. var [,,c,d] = foo();
2. var { w, z } = bar();
3.
4. console.log( c, z );                 // 3 6
5. console.log( d, w );                 // undefined undefined
```

这种行为平行地遵循早先提到的“ `undefined` 意味着缺失”原则。

我们在本章早先检视了 `...` 操作符，并看到了它有时可以用于将一个数组值扩散为它的分离值，而有时它可以被用于相反的操作：将一组值收集进一个数组。

除了在函数声明中的收集/剩余用法以外， `...` 可以在解构赋值中实施相同的行为。为了展示这一点，让我们回想一下本章早先的一个代码段：

```
1. var a = [2,3,4];
2. var b = [ 1, ...a, 5 ];
3.
4. console.log( b );                    // [1,2,3,4,5]
```

我们在这里看到因为 `...a` 出现在数组 `[ .. ]` 中值的位置，所以它将 `a` 扩散开。如果 `...a` 出现一个数组解构的位置，它会实施收集行为：

```
1. var a = [2,3,4];
2. var [ b, ...c ] = a;
3.
4. console.log( b, c );                 // 2 [3,4]
```

解构赋值 `var [ .. ] = a` 为了将 `a` 赋值给在 `[ .. ]` 中描述的模式而将它扩散开。第一部分的名称 `b` 对应 `a` 中的第一个值(`2`)。然后 `...c` 将剩余的值 (`3` 和 `4`) 收集到一个称

为 `c` 的数组中。

注意： 我们已经看到 `...` 是如何与数组一起工作的，但是对象呢？那不是一个ES6特性，但是参看第八章中关于一种可能的“ES6之后”的特性的讨论，它可以让 `...` 扩散或者收集对象。

默认值赋值

两种形式的解构都可以为赋值提供默认值选项，它使用和早先讨论过的默认函数参数值相似的 `=` 语法。

考虑如下代码：

```
var [ a = 3, b = 6, c = 9, d = 12 ] = foo();
var { x = 5, y = 10, z = 15, w = 20 } = bar();

console.log( a, b, c, d );          // 1 2 3 12
console.log( x, y, z, w );          // 4 5 6 20
```

你可以将默认值赋值与前面讲过的赋值表达式语法组合在一起。例如：

```
var { x, y, z, w: WW = 20 } = bar();

console.log( x, y, z, WW );          // 4 5 6 20
```

如果你在一个解构中使用一个对象或者数组作为默认值，那么要小心不要把自己（或者读你的代码的其他开发者）搞糊涂了。你可能会创建一些非常难理解的代码：

```
var x = 200, y = 300, z = 100;
var o1 = { x: { y: 42 }, z: { y: z } };

( { y: x = { y: y } } = o1 );
( { z: y = { y: z } } = o1 );
( { x: z = { y: x } } = o1 );
```

你能从这个代码段中看出 `x`， `y` 和 `z` 最终是什么值吗？花点儿时间好好考虑一下，我能想象你的样子。我会终结这个悬念：

```
console.log( x.y, y.y, z.y );       // 300 100 42
```

这里的要点是：解构很棒也可以很有用，但是如果使用得不明智，它也是一把可以伤人（某人的大脑）的利剑。

嵌套解构

如果你正在解构的值拥有嵌套的对象或数组，你也可以解构这些嵌套的值：

```
var a1 = [ 1, [2, 3, 4], 5 ];
var o1 = { x: { y: { z: 6 } } };

var [ a, [ b, c, d ], e ] = a1;
var { x: { y: { z: w } } } = o1;

console.log( a, b, c, d, e );        // 1 2 3 4 5
console.log( w );                    // 6
```

嵌套的解构可以是一种将对象名称空间扁平化的简单方法。例如：

```
var App = {
    model: {
        User: function(){ .. }
    }
};

// 取代：
// var User = App.model.User;

var { model: { User } } = App;
```

参数解构

你能在下面的代码段中发现赋值吗？

```
function foo(x) {
    console.log( x );
}

foo( 42 );
```

其中的赋值有点儿被隐藏的感觉：当 `foo(42)` 被执行时 `42` （参数值）被赋值给 `x` （参数）。如果参数/参数值对是一种赋值，那么按常理说它是一个可以被解构的赋值，对吧？当然！

考虑参数的数组解构：

```
function foo( [ x, y ] ) {
    console.log( x, y );
}
```

```
4.
5. foo( [ 1, 2 ] );                    // 1 2
6. foo( [ 1 ] );                        // 1 undefined
7. foo( [] );                           // undefined undefined
```

参数也可以进行对象解构：

```
1. function foo( { x, y } ) {
2.     console.log( x, y );
3. }
4.
5. foo( { y: 1, x: 2 } );               // 2 1
6. foo( { y: 42 } );                    // undefined 42
7. foo( {} );                           // undefined undefined
```

这种技术是命名参数值（一个长期以来被渴求的JS特性！）的一种近似解法：对象上的属性映射到被解构的同名参数上。这也意味着我们免费地（在任何位置）得到了可选参数，如你所见，省去“参数” `x` 可以如我们期望的那样工作。

当然，先前讨论过的所有解构的种类对于参数解构来说都是可用的，包括嵌套解构，默认值，和其他。解构也可以和其他ES6函数参数功能很好地混合在一起，比如默认参数值和剩余/收集参数。

考虑这些快速的示例（当然这没有穷尽所有可能的种类）：

```
1. function f1([ x=2, y=3, z ]) { .. }
2. function f2([ x, y, ...z], w) { .. }
3. function f3([ x, y, ...z], ...w) { .. }
4.
5. function f4({ x: X, y }) { .. }
6. function f5({ x: X = 10, y = 20 }) { .. }
7. function f6({ x = 10 } = {}, { y } = { y: 10 }) { .. }
```

为了展示一下，让我们从这个代码段中取一个例子来检视：

```
1. function f3([ x, y, ...z], ...w) {
2.     console.log( x, y, z, w );
3. }
4.
5. f3( [] );                            // undefined undefined [] []
6. f3( [1,2,3,4], 5, 6 );               // 1 2 [3,4] [5,6]
```

这里使用了两个 `...` 操作符，他们都是将值收集到数组中（ `z` 和 `w` ），虽然 `...z` 是从第一个数组参数值的剩余值中收集，而 `...w` 是从第一个之后的剩余主参数值中收集的。

解构默认值 + 参数默认值

有一个微妙的地方你应当注意要特别小心 —— 解构默认值与函数参数默认值的行为之间的不同。例如：

```
function f6({ x = 10 } = {}, { y } = { y: 10 }) {
    console.log( x, y );
}

f6();                               // 10 10
```

首先，看起来我们用两种不同的方法为参数 `x` 和 `y` 都声明了默认值 `10` 。然而，这两种不同的方式会在特定的情况下表现出不同的行为，而且这种区别极其微妙。

考虑如下代码：

```
f6( {}, {} );                       // 10 undefined
```

等等，为什么会这样？十分清楚，如果在第一个参数值的对象中没有一个同名属性被传递，那么命名参数 `x` 将默认为 `10` 。

但 `y` 是 `undefined` 是怎么回事儿？值 `{ y: 10 }` 是一个作为函数参数默认值的对象，不是结构默认值。因此，它仅在第二个参数根本没有被传递，或者 `undefined` 被传递时生效，

在前面的代码段中，我们传递了第二个参数（ `{}` ），所以默认值 `{ y: 10 }` 不被使用，而解构 `{ y }` 会针对被传入的空对象值 `{}` 发生。

现在，将 `{ y } = { y: 10 }` 与 `{ x = 10 } = {}` 比较一下。

对于 `x` 的使用形式来说，如果第一个函数参数值被省略或者是 `undefined` ，会默认地使用空对象 `{}` 。然后，不管在第一个参数值的位置上是什么值 —— 要么是默认的 `{}` ，要么是你传入的 —— 都会被 `{ x = 10 }` 解构，它会检查属性 `x` 是否被找到，如果没有找到（或者是 `undefined` ），默认值 `10` 会被设置到命名参数 `x` 上。

深呼吸。回过头去把最后几段多读几遍。让我们用代码复习一下：

```
function f6({ x = 10 } = {}, { y } = { y: 10 }) {
    console.log( x, y );
}

f6();                               // 10 10
f6( undefined, undefined );           // 10 10
f6( {}, undefined );                // 10 10

f6( {}, {} );                       // 10 undefined
```

```
f6( undefined, {} );                // 10 undefined

f6( { x: 2 }, { y: 3 } );           // 2 3
```

一般来说，与参数 `y` 的默认行为比起来，参数 `x` 的默认行为可能看起来更可取也更合理。因此，理解 `{ x = 10 } = {}` 形式与 `{ y } = { y: 10 }` 形式为何与如何不同是很重要的。

如果这仍然有点儿模糊，回头再把它读一遍，并亲自把它玩弄一番。未来的你将会感谢你花了时间把这种非常微妙的，晦涩的细节的坑搞明白。

嵌套默认值：解构与重构

虽然一开始可能很难掌握，但是为一个嵌套的对象的属性设置默认值产生了一种有趣的惯用法：将对象解构与一种我称为 重构 的东西一起使用。

考虑在一个嵌套的对象结构中的一组默认值，就像下面这样：

```
// 摘自：http://es-discourse.com/t/partial-default-arguments/120/7

var defaults = {
    options: {
        remove: true,
        enable: false,
        instance: {}
    },
    log: {
        warn: true,
        error: true
    }
};
```

现在，我们假定你有一个称为 `config` 的对象，它有一些这其中的值，但也许不全有，而且你想要将所有的默认值设置到这个对象的缺失点上，但不覆盖已经存在的特定设置：

```
var config = {
    options: {
        remove: false,
        instance: null
    }
};
```

你当然可以手动这样做，就像你可能曾经做过的那样：

```
config.options = config.options || {};
config.options.remove = (config.options.remove !== undefined) ?
```

```
    config.options.remove : defaults.options.remove;
config.options.enable = (config.options.enable !== undefined) ?
    config.options.enable : defaults.options.enable;
...
```

讨厌。

另一些人可能喜欢用覆盖赋值的方式来完成这个任务。你可能会被ES6的 `Object.assign(..)` 工具（见第六章）所吸引，来首先克隆 `defaults` 中的属性然后使用从 `config` 中克隆的属性覆盖它，像这样：

```
config = Object.assign( {}, defaults, config );
```

这看起来好多了，是吧？但是这里有一个重大问题！ `Object.assign(..)` 是浅拷贝，这意味着当它拷贝 `defaults.options` 时，它仅仅拷贝这个对象的引用，而不是深度克隆这个对象的属性到一个 `config.options` 对象。 `Object.assign(..)` 需要在你的对象树的每一层中实施才能得到你期望的深度克隆。

注意： 许多JS工具库/框架都为对象的深度克隆提供它们自己的选项，但是那些方式和它们的坑超出了我们在这里的讨论范围。

那么让我们检视一下ES6的带有默认值的对象解构能否帮到我们：

```
config.options = config.options || {};
config.log = config.log || {};
({
    options: {
        remove: config.options.remove = defaults.options.remove,
        enable: config.options.enable = defaults.options.enable,
        instance: config.options.instance = defaults.options.instance
    } = {},
    log: {
        warn: config.log.warn = defaults.log.warn,
        error: config.log.error = defaults.log.error
    } = {}
} = config);
```

不像 `Object.assign(..)` 的虚假诺言（因为它只是浅拷贝）那么好，但是我想它要比手动的方式强多了。虽然它仍然很不幸地带有冗余和重复。

前面的代码段的方式可以工作，因为我黑进了结构和默认机制来为我做属性的 `=== undefined` 检查和赋值的决定。这里的技巧是，我解构了 `config` （看看在代码段末尾的 `= config` ），但是我将所有解构出来的值又立即赋值回 `config` ，带着 `config.options.enable` 赋值引用。

但还是太多了。让我们看看能否做得更好。

下面的技巧在你知道你正在解构的所有属性的名称都是唯一的情况下工作得最好。但即使不是这样的情况你也仍然可以使用它，只是没有那么好 —— 你将不得不分阶段解构，或者创建独一无二的本地变量作为临时的别名。

如果我们将所有的属性完全解构为顶层变量，那么我们就可以立即重构来重组原本的嵌套对象解构。

但是所有那些游荡在外的临时变量将会污染作用域。所以，让我们通过一个普通的 `{ }` 包围块儿来使用块儿作用域（参见本章早先的“块儿作用域声明”）。

```
// 将`defaults`混入`config`
{
    // 解构（使用默认值赋值）
    let {
        options: {
            remove = defaults.options.remove,
            enable = defaults.options.enable,
            instance = defaults.options.instance
        } = {},
        log: {
            warn = defaults.log.warn,
            error = defaults.log.error
        } = {}
    } = config;

    // 重构
    config = {
        options: { remove, enable, instance },
        log: { warn, error }
    };
}
```

这看起来好多了，是吧？

注意： 你也可以使用箭头IIFE来代替一般的 `{ }` 块儿和 `let` 声明来达到圈占作用域的目的。你的解构赋值/默认值将位于参数列表中，而你的重构将位于函数体的 `return` 语句中。

在重构部分的 `{ warn, error }` 语法可能是你初次见到；它称为“简约属性”，我们将在下一节讲解它！

对象字面量扩展

- 对象字面量扩展
  - 简约属性
  - 简约方法
    - 简约匿名
    - ES5 Getter/Setter
  - 计算型属性名
  - 设置 `[[Prototype]]`
  - 对象 `super`

对象字面量扩展

ES6给不起眼儿的 `{ .. }` 对象字面量增加了几个重要的便利扩展。

简约属性

你一定很熟悉用这种形式的对象字面量声明：

```
var x = 2, y = 3,
    o = {
        x: x,
        y: y
    };
```

如果到处说 `x: x` 总是让你感到繁冗，那么有个好消息。如果你需要定义一个名称和词法标识符一致的属性，你可以将它从 `x: x` 缩写为 `x` 。考虑如下代码：

```
var x = 2, y = 3,
    o = {
        x,
        y
    };
```

简约方法

本着与我们刚刚检视的简约属性相同的精神，添附在对象字面量属性上的函数也有一种便利简约形式。

以前的方式：

```
var o = {
    x: function(){
        // ..
    },
    y: function(){
        // ..
    }
}
```

而在ES6中：

```
var o = {
    x() {
        // ..
    },
    y() {
        // ..
    }
}
```

警告： 虽然 `x() { .. }` 看起来只是 `x: function(){ .. }` 的缩写，但是简约方法有一种特殊行为，是它们对应的老方式所不具有的；确切地说，是允许 `super` （参见本章稍后的“对象 `super` ”）的使用。

Generator（见第四章）也有一种简约方法形式：

```
var o = {
    *foo() { .. }
};
```

简约匿名

虽然这种便利缩写十分诱人，但是这其中有一个微妙的坑要小心。为了展示这一点，让我们检视一下如下的前ES6代码，你可能会试着使用简约方法来重构它：

```
function runSomething(o) {
    var x = Math.random(),
        y = Math.random();

    return o.something( x, y );
}

runSomething( {
    something: function something(x,y) {
        if (x > y) {
```

```
            // 使用相互对调的`x`和`y`来递归地调用
            return something( y, x );
        }

        return y - x;
    }
} );
```

这段蠢代码只是生成两个随机数，然后用大的减去小的。但这里重要的不是它做的是什么，而是它是如何被定义的。让我把焦点放在对象字面量和函数定义上，就像我们在这里看到的：

```
runSomething( {
    something: function something(x,y) {
        // ..
    }
} );
```

为什么我们同时说 `something:` 和 `function something` ？这不是冗余吗？实际上，不是，它们俩被用于不同的目的。属性 `something` 让我们能够调用 `o.something(..)` ，有点儿像它的公有名称。但是第二个 `something` 是一个词法名称，使这个函数可以为了递归而从内部引用它自己。

你能看出来为什么 `return something(y,x)` 这一行需要名称 `something` 来引用这个函数吗？因为这里没有对象的词法名称，要是有的话我们就可以说 `return o.something(y,x)` 或者其他类似的东西。

当一个对象字面量的确拥有一个标识符名称时，这其实是一个很常见的做法，比如：

```
var controller = {
    makeRequest: function(..){
        // ..
        controller.makeRequest(..);
    }
};
```

这是个好主意吗？也许是，也许不是。你在假设名称 `controller` 将总是指向目标对象。但它也很可能不是 —— 函数 `makeRequest(..)` 不能控制外部的代码，因此不能强制你的假设一定成立。这可能会回过头来咬到你。

另一些人喜欢使用 `this` 定义这样的东西：

```
var controller = {
    makeRequest: function(..){
        // ..
        this.makeRequest(..);
    }
};
```

这看起来不错，而且如果你总是用 `controller.makeRequest(..)` 来调用方法的话它就应该能工作。但现在你有一个 `this` 绑定的坑，如果你做这样的事情的话：

```
btn.addEventListener( "click", controller.makeRequest, false );
```

当然，你可以通过传递 `controller.makeRequest.bind(controller)` 作为绑定到事件上的处理器引用来解决这个问题。但是这很讨厌 —— 它不是很吸引人。

或者要是你的内部 `this.makeRequest(..)` 调用需要从一个嵌套的函数内发起呢？你会有另一个 `this` 绑定灾难，人们经常使用 `var self = this` 这种用黑科技解决，就像：

```
var controller = {
    makeRequest: function(..){
        var self = this;

        btn.addEventListener( "click", function(){
            // ..
            self.makeRequest(..);
        }, false );
    }
};
```

更讨厌。

注意： 更多关于 `this` 绑定规则和陷阱的信息，参见本系列的 *this*与对象原型 的第一到二章。

好了，这些与简约方法有什么关系？回想一下我们的 `something(..)` 方法定义：

```
runSomething( {
    something: function something(x,y) {
        // ..
    }
} );
```

在这里的第二个 `something` 提供了一个超级便利的词法标识符，它总是指向函数自己，给了我们一个可用于递归，事件绑定/解除等等的完美引用 —— 不用乱搞 `this` 或者使用不可靠的对象引用。

太好了！

那么，现在我们试着将函数引用重构为这种ES6解约方法的形式：

```
runSomething( {
    something(x,y) {
        if (x > y) {
```

```
            return something( y, x );
        }

        return y - x;
    }
} );
```

第一眼看上去不错，除了这个代码将会坏掉。 `return something(..)` 调用经不会找到 `something` 标识符，所以你会得到一个 `ReferenceError` 。噢，但为什么？

上面的ES6代码段将会被翻译为：

```
runSomething( {
    something: function(x,y){
        if (x > y) {
            return something( y, x );
        }

        return y - x;
    }
} );
```

仔细看。你看出问题了吗？简约方法定义暗指 `something: function(x,y)` 。看到我们依靠的第二个 `something` 是如何被省略的了吗？换句话说，简约方法暗指匿名函数表达式。

对，讨厌。

注意： 你可能认为在这里 `=>` 箭头函数是一个好的解决方案。但是它们也同样不够，因为它们也是匿名函数表达式。我们将在本章稍后的“箭头函数”中讲解它们。

一个部分地补偿了这一点的消息是，我们的简约函数 `something(x,y)` 将不会是完全匿名的。参见第七章的“函数名”来了解ES6函数名称的推断规则。这不会在递归中帮到我们，但是它至少在调试时有用处。

那么我们怎样总结简约方法？它们简短又甜蜜，而且很方便。但是你应当仅在你永远不需要将它们用于递归或事件绑定/解除时使用它们。否则，就坚持使用你的老式 `something: function something(..)` 方法定义。

你的很多方法都将可能从简约方法定义中受益，这是个非常好的消息！只要小心几处未命名的灾难就好。

ES5 Getter/Setter

技术上讲，ES5定义了getter/setter字面形式，但是看起来它们没有被太多地使用，这主要是由于缺乏转译器来处理这种新的语法（其实，它是ES5中加入的唯一的主要新语法）。所以虽然它不是一

个ES6的新特性，我们也将简单地复习一下这种形式，因为它可能会随着ES6的向前发展而变得有用得多。

考虑如下代码：

```
var o = {
    __id: 10,
    get id() { return this.__id++; },
    set id(v) { this.__id = v; }
}

o.id;            // 10
o.id;            // 11
o.id = 20;
o.id;            // 20

// 而：
o.__id;            // 21
o.__id;            // 还是 —— 21！
```

这些getter和setter字面形式也可以出现在类中；参见第三章。

警告： 可能不太明显，但是setter字面量必须恰好有一个被声明的参数；省略它或罗列其他的参数都是不合法的语法。这个单独的必须参数 可以 使用解构和默认值（例如， `set id({ id: v = 0 }) { .. }` ），但是收集/剩余 `...` 是不允许的（ `set id(...v) { .. }` ）。

计算型属性名

你可能曾经遇到过像下面的代码段那样的情况，你的一个或多个属性名来自于某种表达式，因此你不能将它们放在对象字面量中：

```
var prefix = "user_";

var o = {
    baz: function(..){ .. }
};

o[ prefix + "foo" ] = function(..){ .. };
o[ prefix + "bar" ] = function(..){ .. };
..
```

ES6为对象字面定义增加了一种语法，它允许你指定一个应当被计算的表达式，其结果就是被赋值属性名。考虑如下代码：

```
var prefix = "user_";
```

```
var o = {
    baz: function(..){ .. },
    [ prefix + "foo" ]: function(..){ .. },
    [ prefix + "bar" ]: function(..){ .. }
    ..
};
```

任何合法的表达式都可以出现在位于对象字面定义的属性名位置的 `[ .. ]` 内部。

很有可能，计算型属性名最经常与 `Symbol` （我们将在本章稍后的“Symbol”中讲解）一起使用，比如：

```
var o = {
    [Symbol.toStringTag]: "really cool thing",
    ..
};
```

`Symbol.toStringTag` 是一个特殊的内建值，我们使用 `[ .. ]` 语法求值得到，所以我们可以将值 `"really cool thing"` 赋值给这个特殊的属性名。

计算型属性名还可以作为简约方法或简约generator的名称出现：

```
var o = {
    ["f" + "oo"]() { .. }    // 计算型简约方法
    *["b" + "ar"]() { .. }    // 计算型简约generator
};
```

设置 `[[Prototype]]`

我们不会在这里讲解原型的细节，所以关于它的更多信息，参见本系列的 *this与对象原型*。

有时候在你声明对象字面量的同时给它的 `[[Prototype]]` 赋值很有用。下面的代码在一段时期内曾经是许多JS引擎的一种非标准扩展，但是在ES6中得到了标准化：

```
var o1 = {
    // ..
};

var o2 = {
    __proto__: o1,
    // ..
};
```

`o2` 是用一个对象字面量声明的，但它也被 `[[Prototype]]` 链接到了 `o1` 。这里的 `__proto__` 属性名还可以是一个字符串 `"__proto__"` ，但是要注意它 不能 是一个计算型属性名的结果（参见前一节）。

客气点儿说， `__proto__` 是有争议的。在ES6中，它看起来是一个最终被很勉强地标准化了的，几十年前的自主扩展功能。实际上，它属于ES6的“Annex B”，这一部分罗列了JS感觉它仅仅为了兼容性的原因，而不得不标准化的东西。

警告： 虽然我勉强赞同在一个对象字面定义中将 `__proto__` 作为一个键，但我绝对不赞同在对象属性形式中使用它，就像 `o.__proto__` 。这种形式既是一个getter也是一个setter（同样也是为了兼容性的原因），但绝对存在更好的选择。更多信息参见本系列的 *this*与对象原型。

对于给一个既存的对象设置 `[[Prototype]]` ，你可以使用ES6的工具 `Object.setPrototypeOf(..)` 。考虑如下代码：

```
var o1 = {
    // ..
};

var o2 = {
    // ..
};

Object.setPrototypeOf( o2, o1 );
```

注意： 我们将在第六章中再次讨论 `Object` 。“ `Object.setPrototypeOf(..)` 静态函数”提供了关于 `Object.setPrototypeOf(..)` 的额外细节。另外参见“ `Object.assign(..)` 静态函数”来了解另一种将 `o2` 原型关联到 `o1` 的形式。

对象 `super`

`super` 通常被认为是仅与类有关。然而，由于JS对象仅有原型而没有类的性质， `super` 是同样有效的，而且在普通对象的简约方法中行为几乎一样。

考虑如下代码：

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

var o2 = {
    foo() {
        super.foo();
```

```
        console.log( "o2:foo" );
    }
};

Object.setPrototypeOf( o2, o1 );

o2.foo();        // o1:foo
                 // o2:foo
```

警告： `super` 仅在简约方法中允许使用，而不允许在普通的函数表达式属性中。而且它还仅允许使用 `super.XXX` 形式（属性/方法访问），而不是 `super()` 形式。

在方法 `o2.foo()` 中的 `super` 引用被静态地锁定在了 `o2` ，而且明确地说是 `o2` 的 `[[Prototype]]` 。这里的 `super` 基本上是 `Object.getPrototypeOf(o2)` —— 显然被解析为 `o1` —— 这就是他如何找到并调用 `o1.foo()` 的。

关于 `super` 的完整细节，参见第三章的“类”。

模板字面量

- 模板字面量
  - 插值表达式
    - 表达式作用域
  - 标签型模板字面量
    - 原始字符串

模板字面量

在这一节的最开始，我将不得不呼唤这个ES6特性的极其……误导人的名称，这要看在你的经验中 模板（*template*） 一词的含义是什么。

许多开发者认为模板是一段可复用的，可重绘的文本，就像大多数模板引擎（Mustache，Handlebars，等等）提供的能力那样。ES6中使用的 模板 一词暗示着相似的东西，就像一种声明可以被重绘的内联模板字面量的方法。然而，这根本不是考虑这个特性的正确方式。

所以，在我们继续之前，我把它重命名为它本应被称呼的名字：插值型字符串字面量（或者略称为 插值型字面量）。

你已经十分清楚地知道了如何使用 `"` 或 `'` 分隔符来声明字符串字面量，而且你还知道它们不是（像有些语言中拥有的）内容将被解析为插值表达式的 智能字符串。

但是，ES6引入了一种新型的字符串字面量，使用反引号 `` ` `` 作为分隔符。这些字符串字面量允许嵌入基本的字符串插值表达式，之后这些表达式自动地被解析和求值。

这是老式的前ES6方式：

```
var name = "Kyle";

var greeting = "Hello " + name + "!";

console.log( greeting );            // "Hello Kyle!"
console.log( typeof greeting );        // "string"
```

现在，考虑这种新的ES6方式：

```
var name = "Kyle";

var greeting = `Hello ${name}!`;

console.log( greeting );            // "Hello Kyle!"
console.log( typeof greeting );        // "string"
```

如你所见，我们在一系列被翻译为字符串字面量的字符周围使用了 `..` ，但是 ${..} 形式中的任何表达式都将立即内联地被解析和求值。称呼这样的解析和求值的高大上名词就是 插值（*interpolation*）（比模板要准确多了）。

被插值的字符串字面量表达式的结果只是一个老式的普通字符串，赋值给变量 greeting 。

警告： typeof greeting == "string" 展示了为什么不将这些实体考虑为特殊的模板值很重要，因为你不能将这种字面量的未求值形式赋值给某些东西并复用它。 `..` 字符串字面量在某种意义上更像是IIFE，因为它自动内联地被求值。 `..` 字符串字面量的结果只不过是一个简单的字符串。

插值型字符串字面量的一个真正的好处是他们允许被分割为多行：

```
var text =
`Now is the time for all good men
to come to the aid of their
country!`;

console.log( text );
// Now is the time for all good men
// to come to the aid of their
// country!
```

在插值型字符串字面量中的换行将会被保留在字符串值中。

除非在字面量值中作为明确的转义序列出现，回车字符 \r （编码点 U+000D ）的值或者回车+换行序列 \r\n （编码点 U+000D 和 U+000A ）的值都会被泛化为一个换行字符 \n （编码点 U+000A ）。但不要担心；这种泛化很少见而且很可能仅会在你将文本拷贝粘贴到JS文件中时才会发生。

插值表达式

在一个插值型字符串字面量中，任何合法的表达式都被允许出现在 ${..} 内部，包括函数调用，内联函数表达式调用，甚至是另一个插值型字符串字面量！

考虑如下代码：

```
function upper(s) {
    return s.toUpperCase();
}

var who = "reader";

var text =
`A very ${upper( "warm" )} welcome
```

```
to all of you ${upper( `${who}s` )}!`;

console.log( text );
// A very WARM welcome
// to all of you READERS!
```

当我们组合变量 `who` 与字符串 `s` 时， 相对于 `who + "s"` ，这里的内部插值型字符串字面量 `` `${who}s` `` 更方便一些。有些情况下嵌套的插值型字符串字面量是有用的，但是如果你发现自己做这样的事情太频繁，或者发现你自己嵌套了好几层时，你就要小心一些。

如果确实有这样情况，你的字符串你值生产过程很可能可以从某些抽象中获益。

警告： 作为一个忠告，使用这样的新发现的力量时要非常小心你代码的可读性。就像默认值表达式和解构赋值表达式一样，仅仅因为你 能 做某些事情，并不意味着你 应该 做这些事情。在使用新的ES6技巧时千万不要做过了头，使你的代码比你或者你的其他队友聪明。

表达式作用域

关于作用域的一个快速提醒是它用于解析表达式中的变量时。我早先提到过一个插值型字符串字面量与IIFE有些相像，事实上这也可以考虑为作用域行为的一种解释。

考虑如下代码：

```
function foo(str) {
    var name = "foo";
    console.log( str );
}

function bar() {
    var name = "bar";
    foo( `Hello from ${name}!` );
}

var name = "global";

bar();                  // "Hello from bar!"
```

在函数 `bar()` 内部，字符串字面量 `` `..` `` 被表达的那一刻，可供它查找的作用域发现变量的 `name` 的值为 `"bar"` 。既不是全局的 `name` 也不是 `foo(..)` 的 `name` 。换句话说，一个插值型字符串字面量在它出现的地方是词法作用域的，而不是任何方式的动态作用域。

标签型模板字面量

再次为了合理性而重命名这个特性：标签型字符串字面量。

老实说，这是一个ES6提供的更酷的特性。它可能看起来有点儿奇怪，而且也许一开始看起来一般不那么实用。但一旦你花些时间在它上面，标签型字符串字面量的用处可能会令你惊讶。

例如：

```
function foo(strings, ...values) {
    console.log( strings );
    console.log( values );
}

var desc = "awesome";

foo`Everything is ${desc}!`;
// [ "Everything is ", "!"]
// [ "awesome" ]
```

让我们花点儿时间考虑一下前面的代码段中发生了什么。首先，跳出来的最刺眼的东西就是 `foo`Everything...`;` 。它看起来不像是任何我们曾经见过的东西。不是吗？

它实质上是一种不需要 `( .. )` 的特殊函数调用。标签 —— 在字符串字面量 `` `..` `` 之前的 `foo` 部分 —— 是一个应当被调用的函数的值。实际上，它可以是返回函数的任何表达式，甚至是一个返回另一个函数的函数调用，就像：

```
function bar() {
    return function foo(strings, ...values) {
        console.log( strings );
        console.log( values );
    }
}

var desc = "awesome";

bar()`Everything is ${desc}!`;
// [ "Everything is ", "!"]
// [ "awesome" ]
```

但是当作为一个字符串字面量的标签时，函数 `foo(..)` 被传入了什么？

第一个参数值 —— 我们称它为 `strings` —— 是一个所有普通字符串的数组（所有被插值的表达式之间的东西）。我们在 `strings` 数组中得到两个值： `"Everything is "` 和 `"!"` 。

之后为了我们示例的方便，我们使用 `...` 收集/剩余操作符（见本章早先的“扩散/剩余”部分）将所有后续的参数值收集到一个称为 `values` 的数组中，虽说你本来当然可以把它们留作参数 `strings` 后面单独的命名参数。

被收集进我们的 `values` 数组中的参数值，就是在字符串字面量中发现的，已经被求过值的插值表达式的结果。所以在我们的例子中 `values` 里唯一的元素显然就是 `awesome` 。

你可以将这两个数组考虑为：在 `values` 中的值原本是你拼接在 `stings` 的值之间的分隔符，而且如果你将所有的东西连接在一起，你就会得到完整的插值字符串值。

一个标签型字符串字面量像是一个在插值表达式被求值之后，但是在最终的字符串被编译之前的处理步骤，允许你在从字面量中产生字符串的过程中进行更多的控制。

一般来说，一个字符串字面量标签函数（在前面的代码段中是 `foo(..)` ）应当计算一个恰当的字符串值并返回它，所以你可以使用标签型字符串字面量作为一个未打标签的字符串字面量来使用：

```
function tag(strings, ...values) {
    return strings.reduce( function(s,v,idx){
        return s + (idx > 0 ? values[idx-1] : "") + v;
    }, "" );
}

var desc = "awesome";

var text = tag`Everything is ${desc}!`;

console.log( text );            // Everything is awesome!
```

在这个代码段中， `tag(..)` 是一个直通操作，因为它不实施任何特殊的修改，而只是使用 `reduce(..)` 来循环遍历，并像一个未打标签的字符串字面量一样，将 `strings` 和 `values` 拼接/穿插在一起。

那么实际的用法是什么？有许多高级的用法超出了我们要在这里讨论的范围。但这里有一个格式化美元数字的简单想法（有些像基本的本地化）：

```
function dollabillsyall(strings, ...values) {
    return strings.reduce( function(s,v,idx){
        if (idx > 0) {
            if (typeof values[idx-1] == "number") {
                // 看，也使用插值性字符串字面量！
                s += `$${values[idx-1].toFixed( 2 )}`;
            }
            else {
                s += values[idx-1];
            }
        }

        return s + v;
    }, "" );
}
```

```
16.
17. var amt1 = 11.99,
18.     amt2 = amt1 * 1.08,
19.     name = "Kyle";
20.
21. var text = dollabillsyall
22. `Thanks for your purchase, ${name}! Your
23. product cost was ${amt1}, which with tax
24. comes out to ${amt2}.`
25.
26. console.log( text );
27. // Thanks for your purchase, Kyle! Your
28. // product cost was $11.99, which with tax
29. // comes out to $12.95.
```

如果在 `values` 数组中遇到一个 `number` 值，我们就在它前面放一个 `"$"` 并用 `toFixed(2)` 将它格式化为小数点后两位有效。否则，我们就不碰这个值而让它直通过去。

原始字符串

在前一个代码段中，我们的标签函数接受的第一个参数值称为 `strings` ，是一个数组。但是有一点儿额外的数据被包含了进来：所有字符串的原始未处理版本。你可以使用 `.raw` 属性访问这些原始字符串值，就像这样：

```
1. function showraw(strings, ...values) {
2.     console.log( strings );
3.     console.log( strings.raw );
4. }
5.
6. showraw`Hello\nWorld`;
7. // [ "Hello
8. // World" ]
9. // [ "Hello\nWorld" ]
```

原始版本的值保留了原始的转义序列 `\n` （ `\` 和 `n` 是两个分离的字符），但处理过的版本认为它是一个单独的换行符。但是，早先提到的行终结符泛化操作，是对两个值都实施的。

ES6带来了一个内建函数，它可以用做字符串字面量的标签： `String.raw(..)` 。它简单地直通 `strings` 值的原始版本：

```
1. console.log( `Hello\nWorld` );
2. // Hello
3. // World
4.
5. console.log( String.raw`Hello\nWorld` );
```

```
// Hello\nWorld

String.raw`Hello\nWorld`.length;
// 12
```

字符串字面量标签的其他用法包括国际化，本地化，和许多其他的特殊处理。

箭头函数

- 箭头函数
  - 不只是简短的语法，而是`this`

箭头函数

我们在本章早先接触了函数中 `this` 绑定的复杂性，而且在本系列的 *this与对象原型* 中也以相当的篇幅讲解过。理解普通函数中基于 `this` 的编程带来的挫折是很重要的，因为这是ES6的新 `=>` 箭头函数的主要动机。

作为与普通函数的比较，我们首先来展示一下箭头函数看起来什么样：

```
function foo(x,y) {
    return x + y;
}

// 对比

var foo = (x,y) => x + y;
```

箭头函数的定义由一个参数列表（零个或多个参数，如果参数不是只有一个，需要有一个 `( .. )` 包围这些参数）组成，紧跟着是一个 `=>` 符号，然后是一个函数体。

所以，在前面的代码段中，箭头函数只是 `(x,y) => x + y` 这一部分，而这个函数的引用刚好被赋值给了变量 `foo` 。

函数体仅在含有多于一个表达式，或者由一个非表达式语句组成时才需要用 `{ .. }` 括起来。如果仅含有一个表达式，而且你省略了外围的 `{ .. }` ，那么在这个表达式前面就会有一个隐含的 `return` ，就像前面的代码段中展示的那样。

这里是一些其他种类的箭头函数：

```
var f1 = () => 12;
var f2 = x => x * 2;
var f3 = (x,y) => {
    var z = x * 2 + y;
    y++;
    x *= 3;
    return (x + y + z) / 2;
};
```

箭头函数 总是 函数表达式；不存在箭头函数声明。而且很明显它们都是匿名函数表达式 —— 它们没

有可以用于递归或者事件绑定/解除的命名引用 —— 但在第七章的“函数名”中将会讲解为了调试的目的而存在的ES6函数名接口规则。

注意： 普通函数参数的所有功能对于箭头函数都是可用的，包括默认值，解构，剩余参数，等等。

箭头函数拥有漂亮，简短的语法，这使得它们在表面上看起来对于编写简洁代码很有吸引力。确实，几乎所有关于ES6的文献（除了这个系列中的书目）看起来都立即将箭头函数仅仅认作“新函数”。

这说明在关于箭头函数的讨论中，几乎所有的例子都是简短的单语句工具，比如那些作为回调传递给各种工具的箭头函数。例如：

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

a = a.map( v => v * 2 );

console.log( a );               // [2,4,6,8,10]
```

在这些情况下，你的内联函数表达式很适合这种在一个单独语句中快速计算并返回结果的模式，对于更繁冗的 `function` 关键字和语法来说箭头函数确实看起来是一个很吸人，而且轻量的替代品。

大多数人看着这样简洁的例子都倾向于发出“哦……！啊……！”的感叹，就像我想象中你刚刚做的那样！

然而我要警示你的是，在我看来，使用箭头函数的语法代替普通的，多语句函数，特别是那些可以被自然地表达为函数声明的函数，是某种误用。

回忆本章早前的字符串字面量标签函数 `dollabillsyall(..)` —— 让我们将它改为使用 `=>` 语法：

```
var dollabillsyall = (strings, ...values) =>
    strings.reduce( (s,v,idx) => {
        if (idx > 0) {
            if (typeof values[idx-1] == "number") {
                // look, also using interpolated
                // string literals!
                s += `$${values[idx-1].toFixed( 2 )}`;
            }
            else {
                s += values[idx-1];
            }
        }

        return s + v;
    }, "" );
```

在这个例子中，我做的唯一修改是删除了 `function` ， `return` ，和一些 `{ .. }` ，然后插入了 `=>` 和一个 `var` 。这是对代码可读性的重大改进吗？呵呵。

实际上我会争论，缺少 `return` 和外部的 `{ .. }` 在某种程度上模糊了这样的事实： `reduce(..)` 调用是函数 `dollabillsyall(..)` 中唯一的语句，而且它的结果是这个调用的预期结果。另外，那些受过训练而习惯于在代码中搜索 `function` 关键字来寻找作用域边界的眼睛，现在需要搜索 `=>` 标志，在密集的代码中这绝对会更加困难。

虽然不是一个硬性规则，但是我要说从 `=>` 箭头函数转换得来的可读性，与被转换的函数长度成反比。函数越长， `=>` 能帮的忙越少；函数越短， `=>` 的闪光之处就越多。

我觉得这样做更明智也更合理：在你需要短的内联函数表达式的地方采用 `=>` ，但保持你的一般长度的主函数原封不动。

不只是简短的语法，而是 `this`

曾经集中在 `=>` 上的大多数注意力都是它通过在你的代码中除去 `function` ， `return` ，和 `{ .. }` 来节省那些宝贵的击键。

但是至此我们一直忽略了一个重要的细节。我在这一节最开始的时候说过， `=>` 函数与 `this` 绑定行为密切相关。事实上， `=>` 箭头函数 主要的设计目的 就是以一种特定的方式改变 `this` 的行为，解决在 `this` 敏感的编码中的一个痛点。

节省击键是掩人耳目的东西，至多是一个误导人的配角。

让我们重温本章早前的另一个例子：

```
var controller = {
    makeRequest: function(..){
        var self = this;

        btn.addEventListener( "click", function(){
            // ..
            self.makeRequest(..);
        }, false );
    }
};
```

我们使用了黑科技 `var self = this` ，然后引用了 `self.makeRequest(..)` ，因为在我们传递给 `addEventListener(..)` 的回调函数内部， `this` 绑定将与 `makeRequest(..)` 本身中的 `this` 绑定不同。换句话说，因为 `this` 绑定是动态的，我们通过 `self` 变量退回到了可预测的词法作用域。

在这其中我们终于可以看到 `=>` 箭头函数主要的设计特性了。在箭头函数内部， `this` 绑定不是动态的，而是词法的。在前一个代码段中，如果我们在回调里使用一个箭头函数， `this` 将会不出所料地成为我们希望它成为的东西。

考虑如下代码：

```
var controller = {
    makeRequest: function(..){
        btn.addEventListener( "click", () => {
            // ..
            this.makeRequest(..);
        }, false );
    }
};
```

前面代码段的箭头函数中的词法 `this` 现在指向的值与外围的 `makeRequest(..)` 函数相同。换句话说， `=>` 是 `var self = this` 的语法上的替代品。

在 `var self = this` （或者，另一种选择是， `.bind(this)` 调用）通常可以帮忙的情况下， `=>` 箭头函数是一个基于相同原则的很好的替代操作。听起来很棒，是吧？

没那么简单。

如果 `=>` 取代 `var self = this` 或 `.bind(this)` 可以工作，那么猜猜 `=>` 用于一个 不需要 `var self = this` 就能工作的 `this` 敏感的函数会发生么？你可能会猜到它将会把事情搞砸。没错。

考虑如下代码：

```
var controller = {
    makeRequest: (..) => {
        // ..
        this.helper(..);
    },
    helper: (..) => {
        // ..
    }
};

controller.makeRequest(..);
```

虽然我们以 `controller.makeRequest(..)` 的方式进行了调用，但是 `this.helper` 引用失败了，因为这里的 `this` 没有像平常那样指向 `controller` 。那么它指向哪里？它通过词法继承了外围的作用域中的 `this` 。在前面的代码段中，它是全局作用域， `this` 指向了全局作用域。呃。

除了词法的 `this` 以外，箭头函数还拥有词法的 `arguments` —— 它们没有自己的 `arguments` 数组，而是从它们的上层继承下来 —— 同样还有词法的 `super` 和 `new.target` （参见第三章的“类”）。

所以，关于 `=>` 在什么情况下合适或不合适，我们现在可以推论出一组更加微妙的规则：

- 如果你有一个简短的，单语句内联函数表达式，它唯一的语句是某个计算后的值的 `return` 语句，

并且 这个函数没有在它内部制造一个`this`引用，并且 没有自引用（递归，事件绑定/解除），并且 你合理地预期这个函数绝不会变得需要`this`引用或自引用，那么你就可能安全地将它重构为一个`=>`箭头函数。

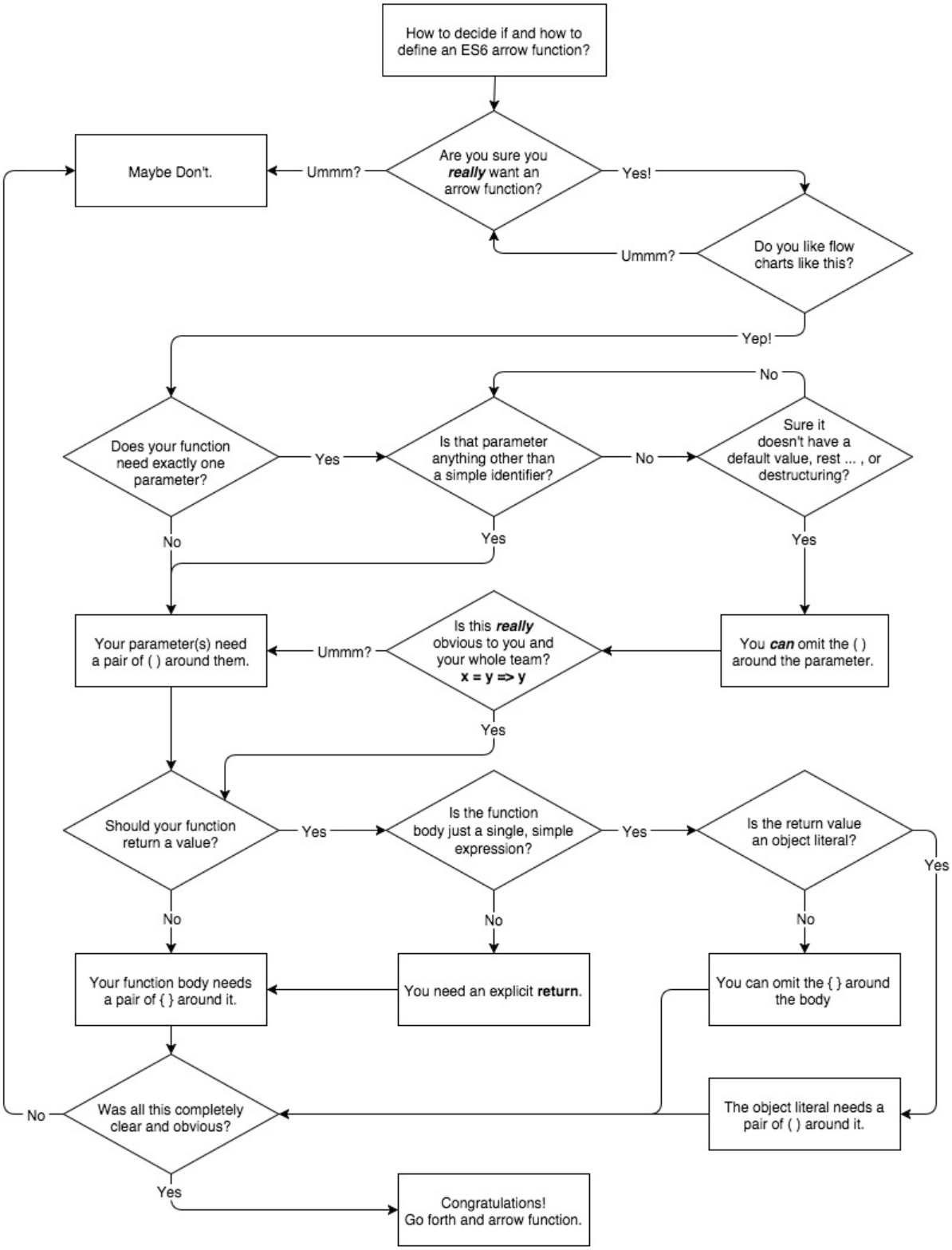
- 如果你有一个内部函数表达式，它依赖于外围函数的`var self = this`黑科技或者`.bind(this)`调用来确保正确的`this`绑定，那么这个内部函数表达式就可能安全地变为一个`=>`箭头函数。
- 如果你有一个内部函数表达式，它依赖于外围函数的类似于`var args = Array.prototype.slice.call(arguments)`这样的东西来制造一个`arguments`的词法拷贝，那么这个内部函数就可能安全地变为一个`=>`箭头函数。
- 对于其他的所有东西 —— 普通函数声明，较长的多语句函数表达式，需要词法名称标识符进行自引用（递归等）的函数，和任何其他不符合前述性质的函数 —— 你就可能应当避免`=>`函数语法。

底线：`=>`与`this`，`arguments`，和`super`的词法绑定有关。它们是ES6为了修正一些常见的问题而被有意设计的特性，而不是为了修正bug，怪异的代码，或者错误。

不要相信任何说`=>`主要是，或者几乎是，为了减少几下击键的炒作。无论你是省下还是浪费了这几下击键，你都应当确切地知道你打入的每个字母是为了做什么。

提示： 如果你有一个函数，由于上述各种清楚的原因而不适合成为一个`=>`箭头函数，但同时它又被声明为一个对象字面量的一部分，那么回想一下本章早先的“简约方法”，它有简短函数语法的另一种选择。

对于如何/为何选用一个箭头函数，如果你喜欢一个可视化的决策图的话：


How to decide if and how to define an ES6 arrow function?
Are you sure you really want an arrow function?
Ummm?
Maybe Don't.
Yes!
Ummm?
Do you like flow charts like this?
Yep!
Does your function need exactly one parameter?
Yes
Is that parameter anything other than a simple identifier?
No
Sure it doesn't have a default value, rest ... , or destructuring?
No
Yes
Yes
Your parameter(s) need a pair of ( ) around them.
Ummm?
Is this really obvious to you and your whole team? x = y => y
You can omit the ( ) around the parameter.
Yes
Should your function return a value?
Yes
Is the function body just a single, simple expression?
Yes
Is the return value an object literal?
Yes
No
No
No
Your function body needs a pair of { } around it.
You need an explicit return.
You can omit the { } around the body
Was all this completely clear and obvious?
The object literal needs a pair of ( ) around it.
No
Yes
Congratulations! Go forth and arrow function.

for..of 循环

正则表达式扩展

- 正则表达式扩展
  - Unicode标志
  - 粘性标志
    - 粘性定位
    - 粘性对比全局
    - 锚定粘性
  - 正则表达式 `flags`

正则表达式扩展

让我们承认吧：长久以来在JS中正则表达式都没怎么改变过。所以一件很棒的事情是，在ES6中它们终于学会了一些新招数。我们将在这里简要地讲解一下新增的功能，但是正则表达式整体的话题是如此厚重，以至于如果你需要复习一下的话你需要找一些关于它的专门章节/书籍（有许多！）。

Unicode标志

我们将在本章稍后的“Unicode”一节中讲解关于Unicode的更多细节。在此，我们将仅仅简要地看一下ES6+正则表达式的新 `u` 标志，它使这个正则表达式的Unicode匹配成为可能。

JavaScript字符串通常被解释为16位字符的序列，它们对应于 基本多文种平面（*Basic Multilingual Plane (BMP)*）(http://en.wikipedia.org/wiki/Plane_%28Unicode%29)中的字符。但是有许多UTF-16字符在这个范围以外，而且字符串可能含有这些多字节字符。

在ES6之前，正则表达式只能基于BMP字符进行匹配，这意味着在匹配时那些扩展字符被看作是两个分离的字符。这通常不理想。

所以，在ES6中， `u` 标志告诉正则表达式使用Unicode（UTF-16）字符的解释方式来处理字符串，这样一来一个扩展的字符将作为一个单独的实体被匹配。

警告： 尽管名字的暗示是这样，但是“UTF-16”并不严格地意味着16位。现代的Unicode使用21位，而且像UTF-8和UTF-16这样的标准大体上是指有多少位用于表示一个字符。

一个例子（直接从ES6语言规范中拿来的）： 𝄞 （G大调音乐符号）是Unicode代码点U+1D11E（0x1D11E）。

如果这个字符出现在一个正则表达式范例中（比如 `/𝄞/` ），标准的BMP解释方式将认为它是需要被匹配的两个字符（0xD834和0xDD1E）。但是ES6新的Unicode敏感模式意味着 `/𝄞/u` （或者Unicode的转义形式 `/\u{1D11E}/u` ）将会把 `"𝄞"` 作为一个单独的字符在一个字符串中进行匹配。

你可能想知道为什么这很重要。在非Unicode的BMP模式下，这个正则表达式范例被看作两个分离的字符，但它仍然可以在一个含有 `"?"` 字符的字符串中找到匹配，如果你试一下就会看到：

```
/?/.test( "?-clef" );           // true
```

重要的是匹配的长度。例如：

```
/^.-clef/ .test( "?-clef" );        // false
/^.-clef/u.test( "?-clef" );        // true
```

这个范例中的 `^.-clef` 说要在普通的 `"-clef"` 文本前面只匹配一个单独的字符。在标准的BMP模式下，这个匹配会失败（因为是两个字符），但是在Unicode模式标志位 `u` 打开的情况下，这个匹配会成功（一个字符）。

另外一个重要的注意点是， `u` 使像 `+` 和 `*` 这样的量词实施于作为一个单独字符的整个Unicode代码点，而不仅仅是字符的 低端替代符（也就是符号最右边的一半）。对于出现在字符类中的Unicode字符也是一样，比如 `/[?-?]/u` 。

注意： 还有许多关于 `u` 在正则表达式中行为的细节，对此Mathias Bynens(https://twitter.com/mathias)撰写了大量的作品(https://mathiasbynens.be/notes/es6-unicode-regex)。

粘性标志

另一个加入ES6正则表达式的模式标志是 `y` ，它经常被称为“粘性模式（sticky mode）”。粘性实质上意味着正则表达式在它开始时有一个虚拟的锚点，这个锚点使正则表达式仅以自己的 `lastIndex` 属性所指示的位置为起点进行匹配。

为了展示一下，让我们考虑两个正则表达式，第一个没有使用粘性模式而第二个有：

```
var re1 = /foo/,
    str = "++foo++";

re1.lastIndex;            // 0
re1.test( str );        // true
re1.lastIndex;            // 0 —— 没有更新

re1.lastIndex = 4;
re1.test( str );        // true —— `lastIndex`被忽略了
re1.lastIndex;            // 4 —— 没有更新
```

关于这个代码段可以观察到三件事：

- `test(..)` 根本不在意 `lastIndex` 的值，而总是从输入字符串的开始实施它的匹配。
- 因为我们的模式没有输入的起始锚点 `^`，所以对 `"foo"` 的搜索可以在整个字符串上自由向前移动。
- `lastIndex` 没有被 `test(..)` 更新。

现在，让我们试一下粘性模式的正则表达式：

```
var re2 = /foo/y,        // <-- 注意粘性标志`y`
    str = "++foo++";

re2.lastIndex;           // 0
re2.test( str );        // false —— 在`0`没有找到“foo”
re2.lastIndex;           // 0

re2.lastIndex = 2;
re2.test( str );        // true
re2.lastIndex;           // 5 —— 在前一次匹配后更新了

re2.test( str );        // false
re2.lastIndex;           // 0 —— 在前一次匹配失败后重置
```

于是关于粘性模式我们可以观察到一些新的事实：

- `test(..)` 在 `str` 中使用 `lastIndex` 作为唯一精确的位置来进行匹配。在寻找匹配时不会发生向前的移动 —— 匹配要么出现在 `lastIndex` 的位置，要么就不存在。
- 如果发生了一个匹配，`test(..)` 就更新 `lastIndex` 使它指向紧随匹配之后的那个字符。如果匹配失败，`test(..)` 就将 `lastIndex` 重置为 `0`。

没有使用 `^` 固定在输入起点的普通非粘性范例可以自由地在字符串中向前移动来搜索匹配。但是粘性模式制约这个范例仅在 `lastIndex` 的位置进行匹配。

正如我在这一节开始时提到过的，另一种考虑的方式是，`y` 暗示着一个虚拟的锚点，它位于正好相对于（也就是制约着匹配的起始位置）`lastIndex` 位置的范例的开头。

警告：在关于这个话题的以前的文献中，这种行为曾经被声称为 `y` 像是在范例中暗示着一个 `^`（输入的起始）锚点。这是不准确的。我们将在稍后的“锚定粘性”中讲解更多细节。

粘性定位

对反复匹配使用 `y` 可能看起来是一种奇怪的限制，因为匹配没有向前移动的能力，你不得不手动保证 `lastIndex` 恰好位于正确的位置上。

这是一种可能的场景：如果你知道你关心的匹配总是会出现在一个数字（例如，`0`，`10`，`20`，等等）倍数的位置。那么你就可以只构建一个受限的范例来匹配你关心的东西，然后在每次匹配那些固定位置之前手动设置 `lastIndex`。

考虑如下代码：

```
var re = /f../y,
    str = "foo       far       fad";

str.match( re );        // ["foo"]

re.lastIndex = 10;
str.match( re );        // ["far"]

re.lastIndex = 20;
str.match( re );        // ["fad"]
```

然而，如果你正在解析一个没有像这样被格式化为固定位置的字符串，在每次匹配之前搞清楚为 `lastIndex` 设置什么东西的做法可能会难以维系。

这里有一个微妙之处要考虑。 `y` 要求 `lastIndex` 位于发生匹配的准确位置。但它不严格要求 你来手动设置 `lastIndex` 。

取而代之的是，你可以用这样的方式构建你的正则表达式：它们在每次主匹配中都捕获你所关心的东西的前后所有内容，直到你想要进行下一次匹配的东西为止。

因为 `lastIndex` 将被设置为一个匹配末尾之后的下一个字符，所以如果你已经匹配了到那个位置的所有东西， `lastIndex` 将总是位于下次 `y` 范例开始的正确位置。

警告： 如果你不能像这样足够范例化地预知输入字符串的结构，这种技术可能不合适，而且你可能不应使用 `y` 。

拥有结构化的字符串输入，可能是 `y` 能够在一个字符串上由始至终地进行反复匹配的最实际场景。考虑如下代码：

```
var re = /\d+\.\s(.*?)(?:\s|$)/y
    str = "1. foo 2. bar 3. baz";

str.match( re );        // [ "1. foo ", "foo" ]

re.lastIndex;            // 7 —— 正确位置！
str.match( re );        // [ "2. bar ", "bar" ]

re.lastIndex;            // 14 —— 正确位置！
str.match( re );        // ["3. baz", "baz"]
```

这能够工作是因为我事先知道输入字符串的结构：总是有一个像 `"1. "` 这样的数字的前缀出现在期望的匹配（ `"foo"` ，等等）之前，而且它后面要么是一个空格，要么就是字符串的末尾（ `$` 锚点）。所以我构建的正则表达式在每次主匹配中捕获了所有这一切，然后我使用一个匹配分组 `(`

`)` 使我真正关心的东西被方便地分离出来。

在第一次匹配（ `"1. foo "` ）之后， `lastIndex` 是 `7` ，它已经是开始下一次匹配 `"2. bar "` 所需的位置了，如此类推。

如果你要使用粘性模式 `y` 进行反复匹配，那么你就可能想要像我们刚刚展示的那样寻找一个机会自动地定位 `lastIndex` 。

粘性对比全局

一些读者可能意识到，你可以使用全局匹配标志位 `g` 和 `exec(..)` 方法来模拟某些像 `lastIndex` 相对匹配的东西，就像这样：

```
var re = /o+./g,        // <-- 看, `g`!
    str = "foot book more";

re.exec( str );           // ["oot"]
re.lastIndex;           // 4

re.exec( str );           // ["ook"]
re.lastIndex;           // 9

re.exec( str );           // ["or"]
re.lastIndex;           // 13

re.exec( str );           // null —— 没有更多的匹配了！
re.lastIndex;           // 0 —— 现在重新开始！
```

虽然使用 `exec(..)` 的 `g` 范例确实从 `lastIndex` 的当前值开始它们的匹配，而且也在每次匹配（或失败）之后更新 `lastIndex` ，但这与 `y` 的行为不是相同的东西。

注意前面代码段中被第二个 `exec(..)` 调用匹配并找到的 `"ook"` ，被定位在位置 `6` ，即便在这个时候 `lastIndex` 是 `4` （前一次匹配的末尾）。为什么？因为正如我们前面讲过的，非粘性匹配可以在它们的匹配过程中自由地向前移动。一个粘性模式表达式在这里将会失败，因为它不允许向前移动。

除了也许不被期望的向前移动的匹配行为以外，使用 `g` 代替 `y` 的另一个缺点是， `g` 改变了一些匹配方法的行为，比如 `str.match(re)` 。

考虑如下代码：

```
var re = /o+./g,        // <-- 看, `g`!
    str = "foot book more";

str.match( re );        // ["oot","ook","or"]
```

看到所有的匹配是如何一次性地被返回的吗？有时这没问题，但有时这不是你想要的。

与 `test(..)` 和 `match(..)` 这样的工具一起使用，粘性标志位 `y` 将给你一次一个的推进式的匹配。只要保证每次匹配时 `lastIndex` 总是在正确的位置上就行！

锚定粘性

正如我们早先被警告过的，将粘性模式认为是暗含着一个以 `^` 开头的范例是不准确的。在正则表达式中锚点 `^` 拥有独特的含义，它 没有 被粘性模式改变。 `^` 总是 一个指向输入起点的锚点，而且 不 以任何方式相对于 `lastIndex` 。

在这个问题上，除了糟糕/不准确的文档，一个在Firefox中进行的老旧的前ES6粘性模式实验不幸地加深了这种困惑，它确实 曾经 使 `^` 相对于 `lastIndex` ，所以这种行为曾经存在了许多年。

ES6选择不这么做。 `^` 在一个范例中绝对且唯一地意味着输入的起点。

这样的后果是，一个像 `/^foo/y` 这样的范例将总是仅在一个字符串的开头找到 `"foo"` 匹配，如果它被允许在那里匹配的话。如果 `lastIndex` 不是 `0` ，匹配就会失败。考虑如下代码：

```
var re = /^foo/y,
    str = "foo";

re.test( str );           // true
re.test( str );           // false
re.lastIndex;            // 0 — 失败之后被重置

re.lastIndex = 1;
re.test( str );           // false — 由于定位而失败
re.lastIndex;            // 0 — 失败之后被重置
```

底线： `y` 加 `^` 加 `lastIndex > 0` 是一种不兼容的组合，它将总是导致失败的匹配。

注意： 虽然 `y` 不会以任何方式改变 `^` 的含义，但是多行模式 `m` 会，这样 `^` 就意味着输入的起点 或者 一个换行之后的文本的起点。所以，如果你在一个范例中组合使用 `y` 和 `m` ，你会在一个字符串中发现多个开始于 `^` 的匹配。但是要记住：因为它的粘性 `y` ，将不得不在后续的每次匹配时确保 `lastIndex` 被置于正确的换行的位置（可能是通过匹配到行的末尾），否者后续的匹配将不会执行。

正则表达式 `flags`

在ES6之前，如果你想要检查一个正则表达式来看看它被施用了什么标志位，你需要将它们 — 讽刺的是，可能是使用另一个正则表达式 — 从 `source` 属性的内容中解析出来，就像这样：

```
var re = /foo/ig;

```

```
re.toString();              // "/foo/ig"

var flags = re.toString().match( /\/([gim]*)$/ )[1];

flags;                      // "ig"
```

在ES6中，你现在可以直接得到这些值，使用新的 `flags` 属性：

```
var re = /foo/ig;

re.flags;                  // "gi"
```

虽然是个细小的地方，但是ES6规范要求表达式的标志位以 `"gimuy"` 的顺序罗列，无论原本的范例中是以什么顺序指定的。这就是出现 `/ig` 和 `"gi"` 的区别的原因。

是的，标志位被指定和罗列的顺序无所谓。

ES6的另一个调整是，如果你向构造器 `RegExp(..)` 传递一个既存的正则表达式，它现在是 `flags` 敏感的：

```
var re1 = /foo*/y;
re1.source;                             // "foo*"
re1.flags;                             // "y"

var re2 = new RegExp( re1 );
re2.source;                             // "foo*"
re2.flags;                             // "y"

var re3 = new RegExp( re1, "ig" );
re3.source;                             // "foo*"
re3.flags;                             // "gi"
```

在ES6之前，构造 `re3` 将抛出一个错误，但是在ES6中你可以在复制时覆盖标志位。

数字字面量扩展

- 数字字面量扩展

数字字面量扩展

在ES5之前，数字字面量看起来就像下面的东西 —— 八进制形式没有被官方指定，唯一被允许的是各种浏览器已经实质上达成一致的一种扩展：

```
var dec = 42,
    oct = 052,
    hex = 0x2a;
```

注意： 虽然你用不同的进制来指定一个数字，但是数字的数学值才是被存储的东西，而且默认的输出解释方式总是10进制的。前面代码段中的三个变量都在它们当中存储了值 `42` 。

为了进一步说明 `052` 是一种非标准形式扩展，考虑如下代码：

```
Number( "42" );             // 42
Number( "052" );           // 52
Number( "0x2a" );           // 42
```

ES5继续允许这种浏览器扩展的八进制形式（包括这样的不一致性），除了在strict模式下，八进制字面量（ `052` ）是不允许的。做出这种限制的主要原因是，许多开发者似乎习惯于下意识地为了将代码对齐而在十进制的数字前面前缀 `0` ，然后遭遇他们完全改变了数字的值的意外！

ES6延续了除十进制数字之外的数字字面量可以被表示的遗留的改变/种类。现在有了一种官方的八进制形式，一种改进了的十六进制形式，和一种全新的二进制形式。由于Web兼容性的原因，在非strict模式下老式的八进制形式 `052` 将继续是合法的，但其实应当永远不再被使用了。

这些是新的ES6数字字面形式：

```
var dec = 42,
    oct = 0o52,           // or `0O52` :(
    hex = 0x2a,           // or `0X2a` :/
    bin = 0b101010;       // or `0B101010` :/
```

唯一允许的小数形式是十进制的。八进制，十六进制，和二进制都是整数形式。

而且所有这些形式的字符串表达形式都是可以被强制转换/变换为它们的数字等价物的：

```
Number( "42" );           // 42
```

```
Number( "0o52" );        // 42
Number( "0x2a" );        // 42
Number( "0b101010" );    // 42
```

虽然严格来说不是ES6新增的，但一个鲜为人知的事实是你其实可以做反方向的转换（好吧，某种意义上的）：

```
var a = 42;

a.toString();            // "42" —— 也可使用`a.toString( 10 )`
a.toString( 8 );        // "52"
a.toString( 16 );        // "2a"
a.toString( 2 );        // "101010"
```

事实上，以这种方你可以用从 `2` 到 `36` 的任何进制表达一个数字，虽然你会使用标准进制 —— 2，8，10，和16 ——之外的情况非常少见。

Unicode

- Unicode
  - Unicode敏感的字符串操作
  - 字符定位
  - Unicode标识符名称

Unicode

我只能说这一节不是一个穷尽了“关于Unicode你想知道的一切”的资料。我想讲解的是，你需要知道在ES6中对Unicode改变了什么，但是我们不会比这深入太多。Mathias Bynens (http://twitter.com/mathias) 大量且出色地撰写/讲解了关于JS和Unicode（参见 https://mathiasbynens.be/notes/javascript-unicode 和 http://fluentconf.com/javascript-html-2015/public/content/2015/02/18-javascript-loves-unicode)。

从 `0x0000` 到 `0xFFFF` 范围内的Unicode字符包含了所有的标准印刷字符（以各种语言），它们都是你可能看到过和互动过的。这组字符被称为 基本多文种平面（*Basic Multilingual Plane (BMP)*）。BMP甚至包含像这个酷雪人一样的有趣字符：☃ (U+2603)。

在这个BMP集合之外还有许多扩展的Unicode字符，它们的范围一直到 `0x10FFFF`。这些符号经常被称为 星形（*astral*） 符号，这正是BMP之外的字符的16组 平面 （也就是，分层/分组）的名称。星形符号的例子包括? （U+1D11E）和? （U+1F4A9）。

在ES6之前，JavaScript字符串可以使用Unicode转义来指定Unicode字符，例如：

```
var snowman = "\u2603";
console.log( snowman );            // "☃"
```

然而， `\uXXXX` Unicode转义仅支持四个十六进制字符，所以用这种方式表示你只能表示BMP集合中的字符。要在ES6以前使用Unicode转义表示一个星形字符，你需要使用一个 代理对（*surrogate pair*） —— 基本上是两个经特殊计算的Unicode转义字符放在一起，被JS解释为一个单独星形字符：

```
var gclef = "\uD834\uDD1E";
console.log( gclef );            // "?"
```

在ES6中，我们现在有了一种Unicode转义的新形式（在字符串和正则表达式中），称为Unicode代码点转义：

```
var gclef = "\u{1D11E}";
```

```
2. console.log( gclef );            // "?"
```

如你所见，它的区别是出现在转义序列中的 `{ }` ，它允许转义序列中包含任意数量的十六进制字符。因为你只需要六个就可以表示在Unicode中可能的最高代码点（也就是，0x10FFFF），所以这是足够的。

Unicode敏感的字符串操作

在默认情况下，JavaScript字符串操作和方法对字符串值中的星形符号是不敏感的。所以，它们独立地处理每个BMP字符，即便是可以组成一个单独字符的两半代理。考虑如下代码：

```
1. var snowman = "☃";
2. snowman.length;                    // 1
3.
4. var gclef = "?";
5. gclef.length;                     // 2
```

那么，我们如何才能正确地计算这样的字符串的长度呢？在这种场景下，下面的技巧可以工作：

```
1. var gclef = "?";
2.
3. [...gclef].length;                // 1
4. Array.from( gclef ).length;        // 1
```

回想一下本章早先的“ `for..of` 循环”一节，ES6字符串拥有内建的迭代器。这个迭代器恰好是Unicode敏感的，这意味着它将自动地把一个星形符号作为一个单独的值输出。我们在一个数组字面量上使用扩散操作符 `...` ，利用它创建了一个字符串符号的数组。然后我们只需检查这个结果数组的长度。ES6的 `Array.from(..)` 基本上与 `[...XYZ]` 做的事情相同，不过我们将在第六章中讲解这个工具的细节。

警告： 应当注意的是，相对地讲，与理论上经过优化的原生工具/属性将做的事情比起来，仅仅为了得到一个字符串的长度就构建并耗尽一个迭代器在性能上的代价是高昂的。

不幸的是，完整的答案并不简单或直接。除了代理对（字符串迭代器可以搞定的），一些特殊的Unicode代码点有其他特殊的行为，解释起来非常困难。例如，有一组代码点可以修改前一个相邻的字符，称为 组合变音符号（*Combining Diacritical Marks*）

考虑这两个数组的输出：

```
1. console.log( s1 );                // "é"
2. console.log( s2 );                // "é"
```

它们看起来一样，但它们不是！这是我们如何创建 `s1` 和 `s2` 的：

```
var s1 = "\xE9",
    s2 = "e\u0301";
```

你可能猜到了，我们前面的 `length` 技巧对 `s2` 不管用：

```
[...s1].length;                  // 1
[...s2].length;                  // 2
```

那么我们能做什么？在这种情况下，我们可以使用ES6的 `String#normalize(..)` 工具，在查询这个值的长度前对它实施一个 *Unicode*正规化操作：

```
var s1 = "\xE9",
    s2 = "e\u0301";

s1.normalize().length;           // 1
s2.normalize().length;           // 1

s1 === s2;                       // false
s1 === s2.normalize();           // true
```

实质上， `normalize(..)` 接受一个 `"e\u0301"` 这样的序列，并把它正规化为 `\xE9` 。正规化甚至可以组合多个相邻的组合符号，如果存在适合他们组合的Unicode字符的话：

```
var s1 = "o\u0302\u0300",
    s2 = s1.normalize(),
    s3 = "ồ";

s1.length;                       // 3
s2.length;                       // 1
s3.length;                       // 1

s2 === s3;                       // true
```

不幸的是，这里的正规化也不完美。如果你有多个组合符号在修改一个字符，你可能不会得到你所期望的长度计数，因为一个被独立定义的，可以表示所有这些符号组合的正规化字符可能不存在。例如：

```
var s1 = "e\u0301\u0330";

console.log( s1 );               // "ḛ́"

s1.normalize().length;           // 2
```

你越深入这个兔子洞，你就越能理解要得到一个“长度”的精确定义是很困难的。我们在视觉上看到的作为一个单独字符绘制的东西 —— 更精确地说，它称为一个 字形 —— 在程序处理的意义上不总是严格地关联到一个单独的“字符”上。

提示： 如果你就是想看看这个兔子洞有多深，看看“字形群集边界（Grapheme Cluster Boundaries）”算法(http://www.Unicode.org/reports/tr29/#Grapheme_Cluster_Boundaries)。

字符定位

与长度的复杂性相似，“在位置2上的字符是什么？”，这么问的意思究竟是什么？前ES6的原生答案来自 `charAt(..)` ，它不会遵守一个星形字符的原子性，也不会考虑组合符号。

考虑如下代码：

```
var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

console.log( s1 );              // "abćd"
console.log( s2 );              // "abćd"
console.log( s3 );              // "ab?d"

s1.charAt( 2 );                  // "c"
s2.charAt( 2 );                  // "ć"
s3.charAt( 2 );                  // "" <-- 不可打印的代理字符
s3.charAt( 3 );                  // "" <-- 不可打印的代理字符
```

那么，ES6会给我们Unicode敏感版本的 `charAt(..)` 吗？不幸的是，不。在本书写作时，在后ES6的考虑之中有一个这样的工具的提案。

但是使用我们在前一节探索的东西（当然也带着它的限制！），我们可以黑一个ES6的答案：

```
var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

[...s1.normalize()][2];          // "ć"
[...s2.normalize()][2];          // "ć"
[...s3.normalize()][2];          // "?"
```

警告： 提醒一个早先的警告：在每次你想得到一个单独的字符时构建并耗尽一个迭代器……在性能上不是很理想。对此，希望我们很快能在后ES6时代得到一个内建的，优化过的工具。

那么 `charCodeAt(..)` 工具的Unicode敏感版本呢？ES6给了我们 `codePointAt(..)` ：

```
var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

s1.normalize().codePointAt( 2 ).toString( 16 );
// "107"

s2.normalize().codePointAt( 2 ).toString( 16 );
// "107"

s3.normalize().codePointAt( 2 ).toString( 16 );
// "1d49e"
```

那么从另一个方向呢？ `String.fromCharCode(..)` 的Unicode敏感版本是ES6的 `String.fromCodePoint(..)` ：

```
String.fromCodePoint( 0x107 );       // "ć"

String.fromCodePoint( 0x1d49e );    // "?"
```

那么等一下，我们能组合 `String.fromCodePoint(..)` 与 `codePointAt(..)` 来得到一个刚才的Unicode敏感 `charAt(..)` 的更好版本吗？是的！

```
var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

String.fromCodePoint( s1.normalize().codePointAt( 2 ) );
// "ć"

String.fromCodePoint( s2.normalize().codePointAt( 2 ) );
// "ć"

String.fromCodePoint( s3.normalize().codePointAt( 2 ) );
// "?"
```

还有好几个字符串方法我们没有在这里讲解，包括 `toUpperCase()` ， `toLowerCase()` ， `substring(..)` ， `indexOf(..)` ， `slice(..)` ，以及其他十几个。它们中没有任何一个为了完全支持Unicode而被改变或增强过，所以在处理含有星形符号的字符串是，你应当非常小心 —— 可能干脆回避它们！

还有几个字符串方法为了它们的行为而使用正则表达式，比如 `replace(..)` 和 `match(..)` 。值得庆幸的是，ES6为正则表达式带来了Unicode支持，正如我们在本章早前的“Unicode标志”中讲解过的那样。

好了，就是这些！有了我们刚刚讲过的各种附加功能，JavaScript的Unicode字符串支持要比前ES6时代好太多了（虽然还不完美）。

Unicode标识符名称

Unicode还可以被用于标识符名称（变量，属性，等等）。在ES6之前，你可以通过Unicode转义这么做，比如：

```
var \u03A9 = 42;

// 等同于：var Ω = 42;
```

在ES6中，你还可以使用前面讲过的代码点转义语法：

```
var \u{2B400} = 42;

// 等同于：var ? = 42;
```

关于究竟哪些Unicode字符被允许使用，有一组复杂的规则。另外，有些字符只要不是标识符名称的第一个字符就允许使用。

注意： 关于所有这些细节，Mathias Bynens写了一篇了不起的文章(https://mathiasbynens.be/notes/javascript-identifiers-es6)。

很少有理由，或者是为了学术上的目的，才会在标识符名称中使用这样不寻常的字符。你通常不会因为依靠这些深奥的功能编写代码而感到舒服。

Symbol

- Symbol
  - Symbol注册表
  - Symbols作为对象属性
    - 内建Symbols

Symbol

在ES6中，长久以来首次，有一个新的基本类型被加入到了JavaScript: `symbol` 。但是，与其他的基本类型不同，symbol没有字面形式。

这是你如何创建一个symbol:

```
var sym = Symbol( "some optional description" );

typeof sym;        // "symbol"
```

一些要注意的事情是:

- 你不能也不应该将`new`与`Symbol(..)`一起使用。它不是一个构造器，你也不是在产生一个对象。
- 被传入`Symbol(..)`的参数是可选的。如果传入的话，它应当是一个字符串，为symbol的目的给出一个友好的描述。
- `typeof`的输出是一个新的值（`"symbol"`），这是识别一个symbol的主要方法。

如果描述被提供的话，它仅仅用于symbol的字符串化表示:

```
sym.toString();        // "Symbol(some optional description)"
```

与基本字符串值如何不是`String`的实例的原理很相似，symbol也不是`Symbol`的实例。如果，由于某些原因，你想要为一个symbol值构建一个封箱的包装器对像，你可以做如下的事情:

```
sym instanceof Symbol;        // false

var symObj = Object( sym );
symObj instanceof Symbol;    // true

symObj.valueOf() === sym;    // true
```

注意： 在这个代码段中的`symObj`和`sym`是可以互换使用的；两种形式可以在symbol被用到的地

方使用。没有太多的理由要使用封箱的包装对象形式（ `symObj` ），而不用基本类型形式（ `sym` ）。和其他基本类型的建议相似，使用 `sym` 而非 `symObj` 可能是最好的。

一个symbol本身的内部值 —— 称为它的 `name` —— 被隐藏在代码之外而不能取得。你可以认为这个symbol的值是一个自动生成的，（在你的应用程序中）独一无二的字符串值。

但如果这个值是隐藏且不可取得的，那么拥有一个symbol还有什么意义？

一个symbol的主要意义是创建一个不会和其他任何值冲突的类字符串值。所以，举例来说，可以考虑将一个symbol用做表示一个事件的名称的值：

```
const EVT_LOGIN = Symbol( "event.login" );
```

然后你可以在一个使用像 `"event.login"` 这样的一般字符串字面量的地方使用 `EVT_LOGIN` ：

```
evthub.listen( EVT_LOGIN, function(data){
    // ..
} );
```

其中的好处是， `EVT_LOGIN` 持有一个不能被其他任何值所（有意或无意地）重复的值，所以在哪个事件被分发或处理的问题上不可能存在任何含糊。

注意： 在前面的代码段的幕后，几乎可以肯定地认为 `evthub` 工具使用了 `EVT_LOGIN` 参数值的symbol值作为某个跟踪事件处理器的内部对象的属性/键。如果 `evthub` 需要将symbol值作为一个真实的字符串使用，那么它将需要使用 `String(..)` 或者 `toString(..)` 进行明确强制转换，因为symbol的隐含字符串强制转换是不允许的。

你可能会将一个symbol直接用做一个对象中的属性名/键，如此作为一个你想将之用于隐藏或元属性的特殊属性。重要的是，要知道虽然你试图这样对待它，但是它 实际上 并不是隐藏或不可接触的属性。

考虑这个实现了 单例 模式行为的模块 —— 也就是，它仅允许自己被创建一次：

```
const INSTANCE = Symbol( "instance" );

function HappyFace() {
    if (HappyFace[INSTANCE]) return HappyFace[INSTANCE];

    function smile() { .. }

    return HappyFace[INSTANCE] = {
        smile: smile
    };
}

```

```
var me = HappyFace(),
    you = HappyFace();

me === you;           // true
```

这里的symbol值 `INSTANCE` 是一个被静态地存储在 `HappyFace()` 函数对象上的特殊的，几乎是隐藏的，类元属性。

替代性地，它本可以是一个像 `__instance` 这样的普通属性，而且其行为将会是一模一样的。symbol的使用仅仅增强了程序元编程的风格，将这个 `INSTANCE` 属性与其他普通的属性间保持隔离。

Symbol注册表

在前面几个例子中使用symbol的一个微小的缺点是，变量 `EVT_LOGIN` 和 `INSTANCE` 不得不存储在外部作用域中（甚至也许是全局作用域），或者用某种方法存储在一个可用的公共位置，这样代码所有需要使用这些symbol的部分都可以访问它们。

为了辅助组织访问这些symbol的代码，你可以使用 全局*symbol*注册表 来创建symbol。例如：

```
const EVT_LOGIN = Symbol.for( "event.login" );

console.log( EVT_LOGIN );       // Symbol(event.login)
```

和：

```
function HappyFace() {
    const INSTANCE = Symbol.for( "instance" );

    if (HappyFace[INSTANCE]) return HappyFace[INSTANCE];

    // ..

    return HappyFace[INSTANCE] = { .. };
}
```

`Symbol.for(..)` 查询全局symbol注册表来查看一个symbol是否已经使用被提供的说明文本存储过了，如果有就返回它。如果没有，就创建一个并返回。换句话说，全局symbol注册表通过描述文本将symbol值看作它们本身的单例。

但这也意味着只要使用匹配的描述名，你的应用程序的任何部分都可以使用 `Symbol.for(..)` 从注册表中取得symbol。

讽刺的是，基本上symbol的本意是在你的应用程序中取代 魔法字符串 的使用（被赋予了特殊意义的随意的字符串值）。但是你正是在全局symbol注册表中使用 魔法 描述字符串值来唯一识别/定位

它们的！

为了避免意外的冲突，你可能想使你的symbol描述十分独特。这么做的一个简单的方法是在它们之中包含前缀/环境/名称空间的信息。

例如，考虑一个像下面这样的工具：

```
function extractValues(str) {
    var key = Symbol.for( "extractValues.parse" ),
        re = extractValues[key] ||
            /[^=&]+?=([^&]+?)(?=&|$)/g,
        values = [], match;

    while (match = re.exec( str )) {
        values.push( match[1] );
    }

    return values;
}
```

我们使用魔法字符串值 `"extractValues.parse"` ，因为在注册表中的其他任何symbol都不太可能与这个描述相冲突。

如果这个工具的一个用户想要覆盖这个解析用的正则表达式，他们也可以使用symbol注册表：

```
extractValues[Symbol.for( "extractValues.parse" )] =
    /..some pattern../g;

extractValues( "..some string.." );
```

除了symbol注册表在全局地存储这些值上提供的协助以外，我们在这里看到的一切其实都可以通过将魔法字符串 `"extractValues.parse"` 作为一个键，而不是一个symbol，来做到。这其中在元编程的层次上的改进要多于在函数层次上的改进。

你可能偶然会使用一个已经被存储在注册表中的symbol值来查询它底层存储了什么描述文本（键）。例如，因为你无法传递symbol值本身，你可能需要通知你的应用程序的另一个部分如何在注册表中定位一个symbol。

你可以使用 `Symbol.keyFor(..)` 取得一个被注册的symbol描述文本（键）：

```
var s = Symbol.for( "something cool" );

var desc = Symbol.keyFor( s );
console.log( desc );            // "something cool"

```

```
// 再次从注册表取得symbol
var s2 = Symbol.for( desc );

s2 === s;                       // true
```

Symbols作为对象属性

如果一个symbol被用作一个对象的属性/键，它会被以一种特殊的方式存储，以至这个属性不会出现在这个对象属性的普通枚举中：

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

Object.getOwnPropertyNames( o );    // [ "foo","baz" ]
```

要取得对象的symbol属性：

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

这表明一个属性symbol实际上不是隐藏的或不可访问的，因为你总是可以在 `Object.getOwnPropertySymbols(..)` 的列表中看到它。

内建Symbols

ES6带来了好几种预定义的内建symbol，它们暴露了在JavaScript对象值上的各种元行为。然而，正如人们所预料的那样，这些symbol 没有 没被注册到全局symbol注册表中。

取而代之的是，它们作为属性被存储到了 `Symbol` 函数对象中。例如，在本章早先的“ `for..of` ”一节中，我们介绍了值 `Symbol.iterator` ：

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

a[Symbol.iterator];           // native function
```

语言规范使用 `@@` 前缀注释指代内建的symbol，最常见的几个是： `@@iterator` ， `@@toStringTag` ， `@@toPrimitive` 。还定义了几个其他的symbol，虽然他们可能不那么频繁地被使用。

注意： 关于这些内建symbol如何被用于元编程的详细信息，参见第七章的“通用Symbol”。

复习

- 复习

复习

ES6给JavaScript增加了一堆新的语法形式，有好多东西要学！

这些东西中的大多数都是为了缓解常见编程惯用法中的痛点而设计的，比如为函数参数设置默认值和将“剩余”的参数收集到一个数组中。解构是一个强大的工具，用来更简约地表达从数组或嵌套对象的赋值。

虽然像箭头函数 `=>` 这样的特性看起来也都是关于更简短更好看的语法，但是它们实际上拥有非常特殊的行为，你应当在恰当的情况下有意地使用它们。

扩展的Unicode支持，新的正则表达式技巧，和新的 `symbol` 基本类型充实了ES6语法的发展演变。

第三章：组织

- 第三章：组织

第三章：组织

编写JS代码是一回事儿，而合理地组织它是另一回事儿。利用常见的组织和重用模式在很大程度上改善了你代码的可读性和可理解性。记住：代码在与其他开发者交流上起的作用，与在给计算机喂指令上起的作用同样重要。

ES6拥有几种重要的特性可以显著改善这些模式，包括：迭代器，generator，模块，和类。

- 迭代器
- Generators
- 模块
- 类
- 复习

迭代器

- 迭代器
  - 接口
    - IteratorResult
  - `next()`迭代
  - 可选的`return(..)`和`throw(..)`
  - 迭代器循环
  - 自定义迭代器
  - 消费迭代器

迭代器

迭代器（*iterator*） 是一种结构化的模式，用于从一个信息源中以一次一个的方式抽取信息。这种模式在程序设计中存在很久了。而且不可否认的是，不知从什么时候起JS开发者们就已经特别地设计并实现了迭代器，所以它根本不是什么新的话题。

ES6所做的是，为迭代器引入了一个隐含的标准化接口。许多在JavaScript中内建的数据结构现在都会暴露一个实现了这个标准的迭代器。而且你也可以构建自己的遵循同样标准的迭代器，来使互用性最大化。

迭代器是一种消费数据的方法，它是组织有顺序的，相继的，基于抽取的。

举个例子，你可能实现一个工具，它在每次被请求时产生一个新的唯一的标识符。或者你可能循环一个固定的列表以轮流的方式产生一系列无限多的值。或者你可以在一个数据库查询的结果上添加一个迭代器来一次抽取一行结果。

虽然在JS中它们不经常以这样的方式被使用，但是迭代器还可以认为是每次控制行为中的一个步骤。这会在考虑generator时得到相当清楚的展示（参见本章稍后的“Generator”），虽然你当然可以不使用generator而做同样的事。

接口

在本书写作的时候，ES6的25.1.1.2部分（https://people.mozilla.org/~jorendorff/es6-draft.html#sec-iterator-interface）详述了 `Iterator` 接口，它有如下的要求：

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

有两个可选成员，有些迭代器用它们进行了扩展：

```
Iterator [可选]
    return() {method}: 停止迭代并返回IteratorResult
    throw() {method}: 通知错误并返回IteratorResult
```

接口 `IteratorResult` 被规定为：

```
IteratorResult
    value {property}: 当前的迭代值或最终的返回值
        （如果它的值为`undefined`，是可选的）
    done {property}: 布尔值，指示完成的状态
```

注意： 我称这些接口是隐含的，不是因为它们没有在语言规范中被明确地被说出来 —— 它们被说出来了！—— 而是因为它们没有作为可以直接访问的对象暴露给代码。在ES6中，JavaScript不支持任何“接口”的概念，所以在你自己的代码中遵循它们纯粹是惯例上的。但是，不论JS在何处需要一个迭代器 —— 例如在一个 `for..of` 循环中 —— 你提供的东西必须遵循这些接口，否则代码就会失败。

还有一个 `Iterable` 接口，它描述了一定能够产生迭代器的对象：

```
Iterable
    @@iterator() {method}: 产生一个迭代器
```

如果你回忆一下第二章的“内建Symbol”， `@@iterator` 是一种特殊的内建symbol，表示可以为对象产生迭代器的方法。

IteratorResult

`IteratorResult` 接口规定从任何迭代器操作的返回值都是这样形式的对象：

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

内建迭代器将总是返回这种形式的值，当然，更多的属性也允许出现在这个返回值中，如果有必要的话。

例如，一个自定义的迭代器可能会在结果对象中加入额外的元数据（比如，数据是从哪里来的，取得它花了多久，缓存过期的时间长度，下次请求的恰当频率，等等）。

注意： 从技术上讲，在值为 `undefined` 的情况下， `value` 是可选的，它将会被认为是不存在或者是没有被设置。因为不管它是表示的就是这个值还是完全不存在，访问 `res.value` 都将会产生 `undefined` ，所以这个属性的存在/不存在更大程度上是一个实现或者优化（或两者）的细节，而非一个功能上的问题。

`next()` 迭代

让我们来看一个数组，它是一个可迭代对象，可以生成一个迭代器来消费它的值：

```
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 }
```

每一次定位在 `Symbol.iterator` 上的方法在值 `arr` 上被调用时，它都将生成一个全新的迭代器。大多数的数据结构都会这么做，包括所有内建在JS中的数据结构。

然而，像事件队列这样的结构也许只能生成一个单独的迭代器（单例模式）。或者某种结构可能在同一时间内只允许存在一个唯一的迭代器，要求当前的迭代器必须完成，才能创建一个新的。

前一个代码段中的 `it` 迭代器不会再你得到值 `3` 时报告 `done: true` 。你必须再次调用 `next()` ，实质上越过数组末尾的值，才能得到完成信号 `done: true` 。在这一节稍后会清楚地讲解这种设计方式的原因，但是它通常被认为是一种最佳实践。

基本类型的字符串值也默认地是可迭代对象：

```
var greeting = "hello world";

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

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

注意： 从技术上讲，这个基本类型值本身不是可迭代对象，但多亏了“封箱”， `"hello world"` 被强制转换为它的 `String` 对象包装形式，它 才是一个可迭代对象。更多信息参见本系列的 类型与文法。

ES6还包括几种新的数据结构，称为集合（参见第五章）。这些集合不仅本身就是可迭代对象，而且它们还提供API方法来生成一个迭代器，例如：

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

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

```
7.
8. it1.next();        // { value: [ "foo", 42 ], done: false }
9. it2.next();        // { value: [ "foo", 42 ], done: false }
10. ..
```

一个迭代器的 `next(..)` 方法能够可选地接受一个或多个参数。大多数内建的迭代器不会实施这种能力，虽然一个generator的迭代器绝对会这么做（参见本章稍后的“Generator”）。

根据一般的惯例，包括所有的内建迭代器，在一个已经被耗尽的迭代器上调用 `next(..)` 不是一个错误，而是简单地持续返回结果 `{ value: undefined, done: true }` 。

可选的 `return(..)` 和 `throw(..)`

在迭代器接口上的可选方法 —— `return(..)` 和 `throw(..)` —— 在大多数内建的迭代器上都没有被实现。但是，它们在generator的上下文环境中绝对有某些含义，所以更具体的信息可以参看“Generator”。

`return(..)` 被定义为向一个迭代器发送一个信号，告知它消费者代码已经完成而且不会再从它那里抽取更多的值。这个信号可以用于通知生产者（应答 `next(..)` 调用的迭代器）去实施一些可能的清理作业，比如释放/关闭网络，数据库，或者文件引用资源。

如果一个迭代器拥有 `return(..)` ，而且发生了可以自动被解释为非正常或者提前终止消费迭代器的任何情况， `return(..)` 就将会被自动调用。你也可以手动调用 `return(..)` 。

`return(..)` 将会像 `next(..)` 一样返回一个 `IteratorResult` 对象。一般来说，你向 `return(..)` 发送的可选值将会在这个 `IteratorResult` 中作为 `value` 发送回来，虽然在一些微妙的情况下这可能不成立。

`throw(..)` 被用于向一个迭代器发送一个异常/错误信号，与 `return(..)` 隐含的完成信号相比，它可能会被迭代器用于不同的目的。它不一定像 `return(..)` 一样暗示着迭代器的完全停止。

例如，在generator迭代器中， `throw(..)` 实际上会将一个被抛出的异常注射到generator暂停的执行环境中，这个异常可以用 `try..catch` 捕获。一个未捕获的 `throw(..)` 异常将会导致generator的迭代器异常中止。

注意： 根据一般的惯例，在 `return(..)` 或 `throw(..)` 被调用之后，一个迭代器就不应该在产生任何结果了。

迭代器循环

正如我们在第二章的“ `for..of` ”一节中讲解的，ES6的 `for..of` 循环可以直接消费一个规范的可迭代对象。

如果一个迭代器也是一个可迭代对象，那么它就可以直接与 `for..of` 循环一起使用。通过给予迭代器

一个简单地返回它自身的 `Symbol.iterator` 方法，你就可以使它成为一个可迭代对象：

```
var it = {
    // 使迭代器`it`成为一个可迭代对象
    [Symbol.iterator]() { return this; },

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

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

现在我们就可以用一个 `for..of` 循环来消费迭代器 `it` 了：

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

为了完全理解这样的循环如何工作，回忆下第二章中的 `for..of` 循环的 `for` 等价物：

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

如果你仔细观察，你会发现 `it.next()` 是在每次迭代之前被调用的，然后 `res.done` 才被查询。如果 `res.done` 是 `true` ，那么这个表达式将会求值为 `false` 于是这次迭代不会发生。

回忆一下之前我们建议说，迭代器一般不应与最终预期的值一起返回 `done: true` 。现在你知道为什么了。

如果一个迭代器返回了 `{ done: true, value: 42 }` ， `for..of` 循环将完全扔掉值 `42` 。因此，假定你的迭代器可能会被 `for..of` 循环或它的 `for` 等价物这样的模式消费的话，你可能应当等到你已经返回了所有相关的迭代值之后才返回 `done: true` 来表示完成。

警告： 当然，你可以有意地将你的迭代器设计为将某些相关的 `value` 与 `done: true` 同时返回。但除非你将此情况在文档中记录下来，否则不要这么做，因为这样会隐含地强制你的迭代器消费者使用一种，与我们刚才描述的 `for..of` 或它的手动等价物不同的模式来进行迭代。

自定义迭代器

除了标准的内建迭代器，你还可以制造你自己的迭代器！所有使它们可以与ES6消费设施（例如， `for..of` 循环和 `...` 操作符）进行互动的代价就是遵循恰当的接口。

让我们试着构建一个迭代器，它能够以斐波那契（Fibonacci）数列的形式产生无限多的数字序列：

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

        return {
            // 使迭代器成为一个可迭代对象
            [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.
```

警告： 如果我们没有插入 `break` 条件，这个 `for..of` 循环将会永远运行下去，这回破坏你的程序，因此可能不是我们想要的！

方法 `Fib[Symbol.iterator]()` 在被调用时返回带有 `next()` 和 `return(..)` 方法的迭代器对象。它的状态通过变量 `n1` 和 `n2` 维护在闭包中。

接下来让我们考虑一个迭代器，它被设计为执行一系列（也叫队列）动作，一次一个：

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

```

```
        return {
            // 使迭代器成为一个可迭代对象
            [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: []
};
```

在 `tasks` 上的迭代器步过在数组属性 `actions` 中找到的函数，并每次执行它们中的一个，并传入你传递给 `next(..)` 的任何参数值，并在标准的 `IteratorResult` 对象中向你返回任何它返回的东西。

这是我们如何使用这个 `tasks` 队列：

```
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 }
```

```
20.
21. it.next( 20, 50 );        // step 2: 20 50
22.                           // { value:  120, done: false }
23.
24. it.next( 20, 50, 120 );    // step 3: 20 50 120
25.                           // { value: 1120, done: false }
26.
27. it.next();                // { done: true }
```

这种特别的用法证实了迭代器可以是一种具有组织功能的模式，不仅仅是数据。这也联系着我们在下一节关于generator将要看到的东西。

你甚至可以更有创意一些，在一块数据上定义一个表示元操作的迭代器。例如，我们可以为默认从0开始递增至（或递减至，对于负数来说）指定数字的一组数字定义一个迭代器。

考虑如下代码：

```
1. if (!Number.prototype[Symbol.iterator]) {
2.     Object.defineProperty(
3.         Number.prototype,
4.         Symbol.iterator,
5.         {
6.             writable: true,
7.             configurable: true,
8.             enumerable: false,
9.             value: function iterator(){
10.                 var i, inc, done = false, top = +this;
11.
12.                 // 正向迭代还是负向迭代？
13.                 inc = 1 * (top < 0 ? -1 : 1);
14.
15.                 return {
16.                     // 使迭代器本身成为一个可迭代对象！
17.                     [Symbol.iterator](){ return this; },
18.
19.                     next() {
20.                         if (!done) {
21.                             // 最初的迭代总是0
22.                             if (i == null) {
23.                                 i = 0;
24.                             }
25.                             // 正向迭代
26.                             else if (top >= 0) {
27.                                 i = Math.min(top,i + inc);
28.                             }
29.                             // 负向迭代
30.                             else {
```

```
                                i = Math.max(top,i + inc);
                            }

                            // 这次迭代之后就完了？
                            if (i == top) done = true;

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

现在，这种创意给了我们什么技巧？

```
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，虽然其实际用途有些值得商榷。但是再一次，有人可能想知道为什么ES6没有提供如此微小但讨喜的特性呢？

如果我连这样的提醒都没给过你，那就是我的疏忽：像我在前面的代码段中做的那样扩展原生原型，是一件你需要小心并了解潜在的危害后才应该做的事情。

在这样的情况下，你与其他代码或者未来的JS特性发生冲突的可能性非常低。但是要小心微小的可能性。并在文档中为后人详细记录下你在做什么。

注意： 如果你想知道更多细节，我在这篇文章(http://blog.getify.com/iterating-es6-numbers/) 中详细论述了这种特别的技术。而且这段评论(http://blog.getify.com/iterating-es6-numbers/comment-page-1/#comment-535294)甚至为制造一个字符串字符范围提出了一个相似的技巧。

消费迭代器

我们已经看到了使用 `for..of` 循环来一个元素一个元素地消费一个迭代器。但是还有一些其他的ES6结构可以消费迭代器。

让我们考虑一下附着这个数组上的迭代器（虽然任何我们选择的迭代器都将拥有如下的行为）：

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

扩散操作符 `...` 将完全耗尽一个迭代器。考虑如下代码：

```
function foo(x,y,z,w,p) {
    console.log( x + y + z + w + p );
}

foo( ...a );            // 15
```

`...` 还可以在一个数组内部扩散一个迭代器：

```
var b = [ 0, ...a, 6 ];
b;                      // [0,1,2,3,4,5,6]
```

数组解构（参见第二章的“解构”）可以部分地或者完全地（如果与一个 `...` 剩余/收集操作符一起使用）消费一个迭代器：

```
var it = a[Symbol.iterator]();

var [x,y] = it;            // 仅从`it`中取前两个元素
var [z, ...w] = it;        // 取第三个，然后一次取得剩下所有的

// `it`被完全耗尽了吗？是的
it.next();                 // { value: undefined, done: true }

x;                         // 1
y;                         // 2
z;                         // 3
w;                         // [4,5]
```

Generators

- Generator
  - 语法
    - 执行一个Generator
    - `yield`
    - `yield *`
  - 迭代器控制
  - 提前完成
    - 提前中止
  - 错误处理
  - 转译一个Generator
  - Generator的使用

Generator

所有的函数都会运行至完成，对吧？换句话说，一旦一个函数开始运行，在它完成之前没有任何东西能够打断它。

至少对于到目前为止的JavaScript的整个历史来说是这样的。在ES6中，引入了一个有些异乎寻常的新形式的函数，称为generator。一个generator可以在运行期间暂停它自己，还可以立即或者稍后继续运行。所以显然它没有普通函数那样的运行至完成的保证。

另外，在运行期间的每次暂停/继续轮回都是一个双向消息传递的好机会，generator可以在这里返回一个值，而使它继续的控制端代码可以发回一个值。

就像前一节中的迭代器一样，有种方式可以考虑generator是什么，或者说它对什么最有用。对此没有一个正确的答案，但我们将试着从几个角度考虑。

注意： 关于generator的更多信息参见本系列的 异步与性能，还可以参见本书的第四章。

语法

generator函数使用这种新语法声明：

```
function *foo() {
    // ..
}
```

`*` 的位置在功能上无关紧要。同样的声明还可以写做以下的任意一种：

```
function *foo()  { .. }
function* foo()  { .. }
function * foo() { .. }
function*foo()   { .. }
..
```

这里 唯一 的区别就是风格的偏好。大多数其他的文献似乎喜欢 `function* foo(..) { .. }` 。我喜欢 `function *foo(..) { .. }` ，所以这就是我将在本书剩余部分中表示它们的方法。

我这样做的理由实质上纯粹是为了教学。在这本书中，当我引用一个generator函数时，我将使用 `*foo(..)` ，与普通函数的 `foo(..)` 相对。我发现 `*foo(..)` 与 `function *foo(..) { .. }` 中 `*` 的位置更加吻合。

另外，就像我们在第二章的简约方法中看到的，在对象字面量中有一种简约generator形式：

```
var a = {
    *foo() { .. }
};
```

我要说在简约generator中， `*foo() { .. }` 要比 `* foo() { .. }` 更自然。这进一步表明了为何使用 `*foo()` 匹配一致性。

一致性使理解与学习更轻松。

执行一个Generator

虽然一个generator使用 `*` 进行声明，但是你依然可以像一个普通函数那样执行它：

```
foo();
```

你依然可以传给它参数值，就像：

```
function *foo(x,y) {
    // ..
}

foo( 5, 10 );
```

主要区别在于，执行一个generator，比如 `foo(5,10)` ，并不实际运行generator中的代码。取而代之的是，它生成一个迭代器来控制generator执行它的代码。

我们将在稍后的“迭代器控制”中回到这个话题，但是简要地说：

```
function *foo() {
    // ..
}

var it = foo();

// 要开始/推进`*foo()`，调用
// `it.next(..)`
```

`yield`

Generator还有一个你可以在它们内部使用的新关键字，用来表示暂停点： `yield` 。考虑如下代码：

```
function *foo() {
    var x = 10;
    var y = 20;

    yield;

    var z = x + y;
}
```

在这个 `*foo()` generator中，前两行的操作将会在开始时运行，然后 `yield` 将会暂停这个generator。如果这个generator被继续， `*foo()` 的最后一行将运行。在一个generator中 `yield` 可以出现任意多次（或者，在技术上讲，根本不出现！）。

你甚至可以在一个循环内部放置 `yield` ，它可以表示一个重复的暂停点。事实上，一个永不完成的循环就意味着一个永不完成的generator，这是完全合法的，而且有时候完全是你需要的。

`yield` 不只是一个暂停点。它是在暂停generator时发送出一个值的表达式。这里是一个位于generator中的 `while..true` 循环，它每次迭代时 `yield` 出一个新的随机数：

```
function *foo() {
    while (true) {
        yield Math.random();
    }
}
```

`yield ..` 表达式不仅发送一个值 —— 不带值的 `yield` 与 `yield undefined` 相同 —— 它还接收（也就是，被替换为）最终的继续值。考虑如下代码：

```
function *foo() {
    var x = yield 10;
    console.log( x );
```

```
4. }
```

这个generator在暂停它自己时将首先 `yield` 出值 `10` 。当你继续这个generator时 —— 使用我们先前提到的 `it.next(..)` —— 无论你使用什么值继续它，这个值都将替换/完成整个表达式 `yield 10` ，这意味着这个值将被赋值给变量 `x`

一个 `yield..` 表达式可以出现在任意普通表达式可能出现的地方。例如：

```
1. function *foo() {
2.     var arr = [ yield 1, yield 2, yield 3 ];
3.     console.log( arr, yield 4 );
4. }
```

这里的 `*foo()` 有四个 `yield ..` 表达式。其中每个 `yield` 都会导致generator暂停以等待一个继续值，这个继续值稍后被用于各个表达式环境中。

`yield` 在技术上讲不是一个操作符，虽然像 `yield 1` 这样使用时看起来确实很像。因为 `yield` 可以像 `var x = yield` 这样完全通过自己被使用，所以将它认为是一个操作符有时令人困惑。

从技术上讲， `yield ..` 与 `a = 3` 这样的赋值表达式拥有相同的“表达式优先级” —— 概念上和操作符优先级很相似。这意味着 `yield ..` 基本上可以出现在任何 `a = 3` 可以合法出现的地方。

让我们展示一下这种对称性：

```
1. var a, b;
2.
3. a = 3;                   // 合法
4. b = 2 + a = 3;           // 不合法
5. b = 2 + (a = 3);         // 合法
6.
7. yield 3;                 // 合法
8. a = 2 + yield 3;         // 不合法
9. a = 2 + (yield 3);       // 合法
```

注意： 如果你好好考虑一下，认为一个 `yield ..` 表达式与一个赋值表达式的行为相似在概念上有些道理。当一个被暂停的generator被继续时，它就以一种与被这个继续值“赋值”区别不大的方式，被这个值完成/替换。

要点：如果你需要 `yield ..` 出现在 `a = 3` 这样的赋值本不被允许出现的位置，那么它就需要被包在一个 `( )` 中。

因为 `yield` 关键字的优先级很低，几乎任何出现在 `yield ..` 之后的表达式都会在被 `yield` 发送之前首先被计算。只有扩散操作符 `...` 和逗号操作符 `,` 拥有更低的优先级，这意味着他们会在 `yield` 已经被求值之后才会被处理。

所以正如带有多个操作符的普通语句一样，存在另一个可能需要 `( )` 来覆盖（提升） `yield` 的低优先级的情况，就像这些表达式之间的区别：

```
yield 2 + 3;            // 与`yield (2 + 3)`相同

(yield 2) + 3;            // 首先`yield 2`, 然后`+ 3`
```

和 `=` 赋值一样， `yield` 也是“右结合性”的，这意味着多个接连出现的 `yield` 表达式被视为从右到左被 `( .. )` 分组。所以， `yield yield yield 3` 将被视为 `yield (yield (yield 3))` 。像 `((yield) yield) yield 3` 这样的“左结合性”解释没有意义。

和其他操作符一样， `yield` 与其他操作符或 `yield` 组合时为了使你的意图没有歧义，使用 `( .. )` 分组是一个好主意，即使这不是严格要求的。

注意： 更多关于操作符优先级和结合性的信息，参见本系列的 类型与文法。

`yield *`

与 `*` 使一个 `function` 声明成为一个 `function *` generator声明的方式一样，一个 `*` 使 `yield` 成为一个机制非常不同的 `yield *` ，称为 *yield*委托。从文法上讲， `yield *..` 的行为与 `yield ..` 相同，就像在前一节讨论过的那样。

`yield * ..` 需要一个可迭代对象；然后它调用这个可迭代对象的迭代器，并将它自己的宿主generator的控制权委托给那个迭代器，直到它被耗尽。考虑如下代码：

```
function *foo() {
    yield *[1,2,3];
}
```

注意： 与generator声明中 `*` 的位置（早先讨论过）一样，在 `yield *` 表达式中的 `*` 的位置在风格上由你来决定。大多数其他文献偏好 `yield* ..` ，但是我喜欢 `yield *..` ，理由和我们已经讨论过的相同。

值 `[1,2,3]` 产生一个将会步过它的值的迭代器，所以generator `*foo()` 将会在被消费时产生这些值。另一种说明这种行为的方式是，yield委托到了另一个generator：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}

function *bar() {
    yield *foo();
}
```

当 `*bar()` 调用 `*foo()` 产生的迭代器通过 `yield *` 受到委托，意味着无论 `*foo()` 产生什么值都会被 `*bar()` 产生。

在 `yield ..` 中表达式的完成值来自于使用 `it.next(..)` 继续generator，而 `yield *..` 表达式的完成值来自于受到委托的迭代器的返回值（如果有的话）。

内建的迭代器一般没有返回值，正如我们在本章早先的“迭代器循环”一节的末尾讲过的。但是如果你定义你自己的迭代器（或者generator），你就可以将它设计为 `return` 一个值， `yield *..` 将会捕获它：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
    return 4;
}

function *bar() {
    var x = yield *foo();
    console.log( "x:", x );
}

for (var v of bar()) {
    console.log( v );
}
// 1 2 3
// x: 4
```

虽然值 `1` ， `2` ，和 `3` 从 `*foo()` 中被 `yield` 出来，然后从 `*bar()` 中被 `yield` 出来，但是从 `*foo()` 中返回的值 `4` 是表达式 `yield *foo()` 的完成值，然后它被赋值给 `x` 。

因为 `yield *` 可以调用另一个generator（通过委托到它的迭代器的方式），它还可以通过调用自己来实施某种generator递归：

```
function *foo(x) {
    if (x < 3) {
        x = yield *foo( x + 1 );
    }
    return x * 2;
}

foo( 1 );
```

取得 `foo(1)` 的结果并调用迭代器的 `next()` 来使它运行它的递归步骤，结果将是 `24` 。第一

次 `*foo()` 运行时 `x` 拥有值 `1` ，它是 `x < 3` 。 `x + 1` 被递归地传递到 `*foo(..)` ，所以之后的 `x` 是 `2` 。再一次递归调用导致 `x` 为 `3` 。

现在，因为 `x < 3` 失败了，递归停止，而且 `return 3 * 2` 将 `6` 给回前一个调用的 `yeild *..` 表达式，它被赋值给 `x` 。另一个 `return 6 * 2` 返回 `12` 给前一个调用的 `x` 。最终 `12 * 2` ，即 `24` ，从generator `*foo(..)` 运行的完成中被返回。

迭代器控制

早先，我们简要地介绍了generator是由迭代器控制的概念。现在让我们完整地深入这个话题。

回忆一下前一节的递归 `*for(..)` 。这是我们如何运行它：

```
function *foo(x) {
    if (x < 3) {
        x = yield *foo( x + 1 );
    }
    return x * 2;
}

var it = foo( 1 );
it.next();                // { value: 24, done: true }
```

在这种情况下，generator并没有真正暂停过，因为这里没有 `yield ..` 表达式。而 `yield *` 只是通过递归调用保持当前的迭代步骤继续运行下去。所以，仅仅对迭代器的 `next()` 函数进行一次调用就完全地运行了generator。

现在让我们考虑一个有多个步骤并且因此有多个产生值的generator：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}
```

我们已经知道我们可以是使用一个 `for..of` 循环来消费一个迭代器，即便它是一个附着在 `*foo()` 这样的generator上：

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

注意： `for..of` 循环需要一个可迭代对象。一个generator函数引用（比如 `foo` ）本身不是一

个可迭代对象；你必须使用 `foo()` 来执行它以得到迭代器（它也是一个可迭代对象，正如我们在本章早先讲解过的）。理论上你可以使用一个实质上仅仅执行 `return this()` 的 `Symbol.iterator` 函数来扩展 `GeneratorPrototype` （所有generator函数的原型）。这将使 `foo` 引用本身成为一个可迭代对象，也就意味着 `for (var v of foo) { .. }` （注意在 `foo` 上没有 `()` ）将可以工作。

让我们手动迭代这个generator：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}

var it = foo();

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

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

如果你仔细观察，这里有三个 `yield` 语句和四个 `next()` 调用。这可能看起来像是一个奇怪的不匹配。事实上，假定所有的东西都被求值并且generator完全运行至完成的话， `next()` 调用将总是比 `yield` 表达式多一个。

但是如果你相反的角度观察（从里向外而不是从外向里）， `yield` 和 `next()` 之间的匹配就显得更有道理。

回忆一下， `yield ..` 表达式将被你用于继续generator的值完成。这意味着你传递给 `next(..)` 的参数值将完成任何当前暂停中等待完成的 `yield ..` 表达式。

让我们这样展示一下这种视角：

```
function *foo() {
    var x = yield 1;
    var y = yield 2;
    var z = yield 3;
    console.log( x, y, z );
}
```

在这个代码段中，每个 `yield ..` 都送出一个值（ `1` ， `2` ， `3` ），但更直接的是，它暂停了generator来等待一个值。换句话说，它就像在问这样一个问题，“我应当在这里用什么值？我会在这里等你告诉我。”

现在，这是我们如何控制 `*foo()` 来启动它：

```
var it = foo();

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

这第一个 `next()` 调用从generator初始的暂停状态启动了它，并运行至第一个 `yield` 。在你调用第一个 `next()` 的那一刻，并没有 `yield ..` 表达式等待完成。如果你给第一个 `next()` 调用传递一个值，目前它会被扔掉，因为没有 `yield` 等着接受这样的一个值。

注意： 一个“ES6之后”时间表中的早期提案 将 允许你在generator内部通过一个分离的元属性（见第七章）来访问一个被传入初始 `next(..)` 调用的值。

现在，让我们回答那个未解的问题，“我应当给 `x` 赋什么值？” 我们将通过给 下一个 `next(..)` 调用发送一个值来回答：

```
it.next( "foo" );        // { value: 2, done: false }
```

现在， `x` 将拥有值 `"foo"` ，但我们也问了一个新的问题，“我应当给 `y` 赋什么值？”

```
it.next( "bar" );        // { value: 3, done: false }
```

答案给出了，另一个问题被提出了。最终答案：

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

现在，每一个 `yield ..` 的“问题”是如何被 下一个 `next(..)` 调用回答的，所以我们观察到的那个“额外的” `next()` 调用总是使一切开始的那一个。

让我们把这些步骤放在一起：

```
var it = foo();

// 启动generator
it.next();               // { value: 1, done: false }

// 回答第一个问题
it.next( "foo" );        // { value: 2, done: false }

// 回答第二个问题
it.next( "bar" );        // { value: 3, done: false }

// 回答第三个问题
it.next( "baz" );        // "foo" "bar" "baz"
                        // { value: undefined, done: true }
```

在生成器的每次迭代都简单地为消费者生成一个值的情况下，你可认为一个generator是一个值的生成器。

但是在更一般的意义上，也许将generator认为是一个受控制的，累进的代码执行过程更恰当，与早先“自定义迭代器”一节中的 `tasks` 队列的例子非常相像。

注意： 这种视角正是我们将如何在第四章中重温generator的动力。特别是， `next(..)` 没有理由一定要在前一个 `next(..)` 完成之后立即被调用。虽然generator的内部执行环境被暂停了，程序的其他部分仍然没有被阻塞，这包括控制generator什么时候被继续的异步动作能力。

提前完成

正如我们在本章早先讲过的，连接到一个generator的迭代器支持可选的 `return(..)` 和 `throw(..)` 方法。它们俩都有立即中止一个暂停的的generator的效果。

考虑如下代码：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}

var it = foo();

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

it.return( 42 );        // { value: 42, done: true }

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

`return(x)` 有点像强制一个 `return x` 就在那个时刻被处理，这样你就立即得到这个指定的值。一旦一个generator完成，无论是正常地还是像展示的那样提前地，它就不再处理任何代码或返回任何值了。

`return(..)` 除了可以手动调用，它还在迭代的最后被任何ES6中消费迭代器的结构自动调用，比如 `for..of` 循环和 `...` 扩散操作符。

这种能力的目的是，在控制端的代码不再继续迭代generator时它可以收到通知，这样它就可能做一些清理工作（释放资源，复位状态，等等）。与普通函数的清理模式完全相同，达成这个目的的主要方法是使用一个 `finally` 子句：

```
function *foo() {
    try {
```

```
        yield 1;
        yield 2;
        yield 3;
    }
    finally {
        console.log( "cleanup!" );
    }
}

for (var v of foo()) {
    console.log( v );
}
// 1 2 3
// cleanup!

var it = foo();

it.next();                // { value: 1, done: false }
it.return( 42 );        // cleanup!
                        // { value: 42, done: true }
```

警告： 不要把 `yield` 语句放在 `finally` 子句内部！它是有效和合法的，但这确实是一个可怕的主意。它在某种意义上推迟了 `return(..)` 调用的完成，因为在 `finally` 子句中的任何 `yield ..` 表达式都被遵循来暂停和发送消息；你不会像期望的那样立即得到一个完成的generator。基本上没有任何好的理由去选择这种疯狂的 坏的部分，所以避免这么做！

前一个代码段除了展示 `return(..)` 如何在中止generator的同时触发 `finally` 子句，它还展示了一个generator在每次被调用时都产生一个全新的迭代器。事实上，你可以并发地使用连接到相同generator的多个迭代器：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}

var it1 = foo();
it1.next();                // { value: 1, done: false }
it1.next();                // { value: 2, done: false }

var it2 = foo();
it2.next();                // { value: 1, done: false }

it1.next();                // { value: 3, done: false }

it2.next();                // { value: 2, done: false }
it2.next();                // { value: 3, done: false }
```

```
18.
19. it2.next();               // { value: undefined, done: true }
20. it1.next();               // { value: undefined, done: true }
```

提前中止

你可以调用 `throw(..)` 来代替 `return(..)` 调用。就像 `return(x)` 实质上在generator当前的暂停点上注入了一个 `return x` 一样，调用 `throw(x)` 实质上就像在暂停点上注入了一个 `throw x` 。

除了处理异常的行为（我们在下一节讲解这对 `try` 子句意味着什么），`throw(..)` 产生相同的提前完成 —— 在generator当前的暂停点中止它的运行。例如：

```
1. function *foo() {
2.     yield 1;
3.     yield 2;
4.     yield 3;
5. }
6.
7. var it = foo();
8.
9. it.next();               // { value: 1, done: false }
10.
11. try {
12.     it.throw( "Oops!" );
13. }
14. catch (err) {
15.     console.log( err );    // Exception: Oops!
16. }
17.
18. it.next();               // { value: undefined, done: true }
```

因为 `throw(..)` 基本上注入了一个 `throw ..` 来替换generator的 `yield 1` 这一行，而且没有东西处理这个异常，它立即传播回外面的调用端代码，调用端代码使用了一个 `try..catch` 来处理了它。

与 `return(..)` 不同的是，迭代器的 `throw(..)` 方法绝不会被自动调用。

当然，虽然没有在前面的代码段中展示，但如果当你调用 `throw(..)` 时有一个 `try..finally` 子句等在generator内部的话，这个 `finally` 子句将会在异常被传播回调用端代码之前有机会运行。

错误处理

正如我们已经得到的提示，generator中的错误处理可以使用 `try..catch` 表达，它在上行和下行两个方向都可以工作。

```
function *foo() {
    try {
        yield 1;
    }
    catch (err) {
        console.log( err );
    }

    yield 2;

    throw "Hello!";
}

var it = foo();

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

try {
    it.throw( "Hi!" );    // Hi!
                          // { value: 2, done: false }
    it.next();

    console.log( "never gets here" );
}
catch (err) {
    console.log( err );    // Hello!
}
```

错误也可以通过 `yield *` 委托在两个方向上传播：

```
function *foo() {
    try {
        yield 1;
    }
    catch (err) {
        console.log( err );
    }

    yield 2;

    throw "foo: e2";
}

function *bar() {
    try {
        yield *foo();

```

```
        console.log( "never gets here" );
    }
    catch (err) {
        console.log( err );
    }
}

var it = bar();

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

    it.throw( "e1" );    // e1
                         // { value: 2, done: false }

    it.next();            // foo: e2
                         // { value: undefined, done: true }
}
catch (err) {
    console.log( "never gets here" );
}

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

当 `*foo()` 调用 `yield 1` 时，值 `1` 原封不动地穿过了 `*bar()` ，就像我们已经看到过的那样。

但这个代码段最有趣的部分是，当 `*foo()` 调用 `throw "foo: e2"` 时，这个错误传播到了 `*bar()` 并立即被 `*bar()` 的 `try..catch` 块儿捕获。错误没有像值 `1` 那样穿过 `*bar()` 。

然后 `*bar()` 的 `catch` 将 `err` 普通地输出（ `"foo: e2"` ）之后 `*bar()` 就正常结束了，这就是为什么迭代器结果 `{ value: undefined, done: true }` 从 `it.next()` 中返回。

如果 `*bar()` 没有用 `try..catch` 环绕着 `yield *..` 表达式，那么错误将理所当然地一直传播出来，而且在它传播的路径上依然会完成（中止） `*bar()` 。

转译一个Generator

有可能在ES6之前的环境中表达generator的能力吗？事实上是可以的，而且有好几种了不起的工具在这么做，包括最著名的Facebook的Regenerator工具(https://facebook.github.io/regenerator/)。

但为了更好地理解generator，让我们试着手动转换一下。基本上讲，我们将制造一个简单的基于闭包的状态机。

我们将使原本的generator非常简单：

```
function *foo() {
    var x = yield 42;
    console.log( x );
}
```

开始之前，我们将需要一个我们能够执行的称为 `foo()` 的函数，它需要返回一个迭代器：

```
function foo() {
    // ..

    return {
        next: function(v) {
            // ..
        }

        // 我们将省略`return(..)`和`throw(..)`
    };
}
```

现在，我们需要一些内部变量来持续跟踪我们的“generator”的逻辑走到了哪一个步骤。我们称它为 `state` 。我们将有三种状态：起始状态的 `0` ，等待完成 `yield` 表达式的 `1` ，和generator完成的 `2` 。

每次 `next(..)` 被调用时，我们需要处理下一个步骤，然后递增 `state` 。为了方便，我们将每个步骤放在一个 `switch` 语句的 `case` 子句中，并且我们将它放在一个 `next(..)` 可以调用的称为 `nextState(..)` 的内部函数中。另外，因为 `x` 是一个横跨整个“generator”作用域的变量，所以它需要存活在 `nextState(..)` 函数的外部。

这是将它们放在一起（很明显，为了使概念的展示更清晰，它经过了某些简化）：

```
function foo() {
    function nextState(v) {
        switch (state) {
            case 0:
                state++;

                // `yield`表达式
                return 42;
            case 1:
                state++;

                // `yield`表达式完成了
                x = v;
                console.log( x );

```

```
                // 隐含的`return`
                return undefined;

            // 无需处理状态`2`
        }
    }

    var state = 0, x;

    return {
        next: function(v) {
            var ret = nextState( v );

            return { value: ret, done: (state == 2) };
        }

        // 我们将省略`return(..)`和`throw(..)`
    };
}
```

最后，让我们测试一下我们的前ES6“generator”：

```
var it = foo();

it.next();                // { value: 42, done: false }

it.next( 10 );            // 10
                        // { value: undefined, done: true }
```

不赖吧？希望这个练习能在你的脑中巩固这个概念：generator实际上只是状态机逻辑的简单语法。这使它们可以广泛地应用。

Generator的使用

我们现在非常深入地理解了generator如何工作，那么，它们在什么地方有用？

我们已经看过了两种主要模式：

- 生产一系列值： 这种用法可以很简单（例如，随机字符串或者递增的数字），或者它也可以表达更加结构化的数据访问（例如，迭代一个数据库查询结果的所有行）。

  这两种方式中，我们使用迭代器来控制generator，这样就可以为每次 `next(..)` 调用执行一些逻辑。在数据解构上的普通迭代器只不过生成值而没有任何控制逻辑。

- 串行执行的任务队列： 这种用法经常用来表达一个算法中步骤的流程控制，其中每一步都要求从某些外部数据源取得数据。对每块儿数据的请求可能会立即满足，或者可能会异步延迟地满足。

> 从generator内部代码的角度来看，在 `yield` 的地方，同步或异步的细节是完全不透明的。另外，这些细节被有意地抽象出去，如此就不会让这样的实现细节把各个步骤间自然的，顺序的表达搞得模糊不清。抽象还意味着实现可以被替换/重构，而根本不用碰generator中的代码。

当根据这些用法观察generator时，它们的含义要比仅仅是手动状态机的一种不同或更好的语法多多了。它们是一种用于组织和控制有序地生产与消费数据的强大工具。

模块

- 模块
  - 过去的方式
  - 向前迈进
    - CommonJS
  - 新的方式
    - `export` API成员
    - `import` API成员
  - 模块循环依赖
  - 模块加载
    - 加载模块之外的模块
    - 自定义加载

模块

我觉得这样说并不夸张：在所有的JavaScript代码组织模式中最重要的就是，而且一直是，模块。对于我自己来说，而且我认为对广大典型的技术社区来说，模块模式驱动着绝大多数代码。

过去的方式

传统的模块模式基于一个外部函数，它带有内部变量和函数，以及一个被返回的“公有API”。这个“公有API”带有对内部变量和功能拥有闭包的方法。它经常这样表达：

```
function Hello(name) {
    function greeting() {
        console.log( "Hello " + name + "!" );
    }

    // 公有API
    return {
        greeting: greeting
    };
}

var me = Hello( "Kyle" );
me.greeting();            // Hello Kyle!
```

这个 `Hello(..)` 模块通过被后续调用可以产生多个实例。有时，一个模块为了作为一个单例（也就是，只需要一个实例）而只被调用一次，这样的情况下常见的是一种前面代码段的变种，使用IIFE：

```
var me = (function Hello(name){
    function greeting() {
        console.log( "Hello " + name + "!" );
    }

    // 公有API
    return {
        greeting: greeting
    };
})( "Kyle" );

me.greeting();            // Hello Kyle!
```

这种模式是经受过检验的。它也足够灵活，以至于在许多不同的场景下可以有大量的各种变化。

其中一种最常见的是异步模块定义（AMD），另一种是统一模块定义（UMD）。我们不会在这里涵盖这些特定的模式和技术，但是它们在网上的许多地方有大量的讲解。

向前迈进

在ES6中，我们不再需要依赖外围函数和闭包来为我们提供模块支持了。ES6模块拥有头等语法上和功能上的支持。

在我们接触这些具体语法之前，重要的是要理解ES6模块与你以前曾经用过的模块比较起来，在概念上的一些相当显著的不同之处：

- ES6使用基于文件的模块，这意味着一个模块一个文件。目前，没有标准的方法将多个模块组合到一个文件中。

  这意味着如果你要直接把ES6模块加载到一个浏览器web应用中的话，你将个别地加载它们，不是像常见的那样为了性能优化而作为一个单独文件中的一个巨大的包加载。

  预计同时期到来的HTTP/2将会大幅缓和这种性能上的顾虑，因为它工作在一个持续的套接字连接上，因而可以用并行的，互相交错的方式非常高效地加载许多小文件。

- 一个ES6模块的API是静态的。这就是说，你在模块的公有API上静态地定义所有被导出的顶层内容，而这些内容导出之后不能被修改。

  有些用法习惯于能够提供动态API定义，它的方法可以根据运行时的条件被增加/删除/替换。这些用法要么必须改变以适应ES6静态API，要么它们就不得不将属性/方法的动态修改限制在一个内层对象中。

- ES6模块都是单例。也就是，模块只有一个维持它状态的实例。每次你将这个模块导入到另一个模块时，你得到的都是一个指向中央实例的引用。如果你想要能够产生多个模块实例，你的模块将需要提供某种工厂来这么做。

- 你在模块的公有API上暴露的属性和方法不是值和引用的普通赋值。它们是在你内部模块定义中的标识符的实际绑定（几乎就是指针）。

  在前ES6的模块中，如果你将一个持有像数字或者字符串这样基本类型的属性放在你的共有API中，那么这个属性是通过值拷贝赋值的，任何对相应内部变量的更新都将是分离的，不会影响在API对象上的共有拷贝。

  在ES6中，导出一个本地私有变量，即便它当前持有一个基本类型的字符串/数字/等等，导出的都是这个变量的一个绑定。如果这个模块改变了这个变量的值，外部导入的绑定就会解析为那个新的值。

- 导入一个模块和静态地请求它被加载是同一件事情（如果它还没被加载的话）。如果你在浏览器中，这意味着通过网络的阻塞加载。如果你在服务器中，它是一个通过文件系统的阻塞加载。

  但是，不要对它在性能的影响上惊慌。因为ES6模块是静态定义的，导入的请求可以被静态地扫描，并提前加载，甚至是在你使用这个模块之前。

  ES6并没有实际规定或操纵这些加载请求如何工作的机制。有一个模块加载器的分离概念，它让每一个宿主环境（浏览器，Node.js，等等）为该环境提供合适的默认加载器。一个模块的导入使用一个字符串值来表示从哪里去取得模块（URL，文件路径，等等），但是这个值在你的程序中是不透明的，它仅对加载器自身有意义。

  如果你想要比默认加载器提供的更细致的控制能力，你可以定义你自己的加载器 —— 默认加载器基本上不提供任何控制，它对于你的程序代码是完全隐藏的。

如你所见，ES6模块将通过封装，控制共有API，以及应用依赖导入来服务于所有的代码组织需求。但是它们用一种非常特别的方式来这样做，这可能与你已经使用多年的模块方式十分接近，也肯能差得很远。

CommonJS

有一种相似，但不是完全兼容的模块语法，称为CommonJS，那些使用Node.js生态系统的人很熟悉它。

不太委婉地说，从长久看来，ES6模块实质上将要取代所有先前的模块格式与标准，即便是CommonJS，因为它们是建立在语言的语法支持上的。如果除了普遍性以外没有其他原因，迟早ES6将不可避免地作为更好的方式胜出。

但是，要达到那一天我们还有相当长的路要走。在服务器端的JavaScript世界中差不多有成百上千的CommonJS风格模块，而在浏览器的世界里各种格式标准的模块（UMD，AMD，临时性的模块方案）数量还要多十倍。这要花许多年过渡才能取得任何显著的进展。

在这个过渡期间，模块转译器/转换器将是绝对必要的。你可能刚刚适应了这种新的现实。不论你是使用正规的模块，AMD，UMD，CommonJS，或者ES6，这些工具都不得不解析并转换为适合你代码运行环境的格式。

对于Node.js，这可能意味着（目前）转换的目标是CommonJS。对于浏览器来说，可能是UMD或者AMD。除了在接下来的几年中随着这些工具的成熟和最佳实践的出现而发生的许多变化。

从现在起，我能对模块的提出的最佳建议是：不管你曾经由于强烈的爱好而虔诚地追随哪一种格式，都要培养对理解ES6模块的欣赏能力，并让你对其他模块模式的倾向性渐渐消失掉。它们就是JS中模块的未来，即便现实有些偏差。

新的方式

使用ES6模块的两个主要的新关键字是 `import` 和 `export` 。在语法上有许多微妙的地方，那么让我们深入地看看。

警告： 一个容易忽视的重要细节： `import` 和 `export` 都必须总是出现在它们分别被使用之处的顶层作用域。例如，你不能把 `import` 或 `export` 放在一个 `if` 条件内部；它们必须出现在所有块儿和函数的外部。

`export` API成员

`export` 关键字要么放在一个声明的前面，要么就与一组特殊的要被导出的绑定一起用作一个操作符。考虑如下代码：

```
export function foo() {
    // ..
}

export var awesome = 42;

var bar = [1,2,3];
export { bar };
```

表达相同导出的另一种方法：

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

var awesome = 42;
var bar = [1,2,3];

export { foo, awesome, bar };
```

这些都称为 命名导出，因为你实际上导出的是变量/函数/等等其他的名称绑定。

任何你没有使用 `export` 标记 的东西将在模块作用域的内部保持私有。也就是说，虽然有些像 `var`

`bar = ..` 的东西看起来像是在顶层全局作用域中声明的，但是这个顶层作用域实际上是模块本身；在模块中没有全局作用域。

注意： 模块确实依然可以访问挂在它外面的 `window` 和所有的“全局”，只是不作为顶层词法作用域而已。但是，你真的应该在你的模块中尽可能地远离全局。

你还可以在命名导出期间“重命名”（也叫别名）一个模块成员：

```
function foo() { .. }

export { foo as bar };
```

当这个模块被导入时，只有成员名称 `bar` 可以用于导入； `foo` 在模块内部保持隐藏。

模块导出不像你习以为常的 `=` 赋值操作符那样，仅仅是值或引用的普通赋值。实际上，当你导出某些东西时，你导出了一个对那个东西（变量等）的一个绑定（有些像指针）。

在你的模块内部，如果你改变一个你已经被导出绑定的变量的值，即使它已经被导入了（见下一节），这个被导入的绑定也将解析为当前的（更新后的）值。

考虑如下代码：

```
var awesome = 42;
export { awesome };

// 稍后
awesome = 100;
```

当这个模块被导入时，无论它是在 `awesome = 100` 设定的之前还是之后，一旦这个赋值发生，被导入的绑定都将被解析为值 `100` ，不是 `42` 。

这是因为，这个绑定实质上是一个指向变量 `awesome` 本身的一个引用，或指针，而不是它的值的一个拷贝。ES6模块绑定引入了一个对于JS来说几乎是史无前例的概念。

虽然你显然可以在一个模块定义的内部多次使用 `export` ，但是ES6绝对偏向于一个模块只有一个单独导出的方式，这称为 默认导出。用TC39协会的一些成员的话说，如果你遵循这个模式你就可以“获得更简单的 `import` 语法作为奖励”，如果你不遵循你就会反过来得到更繁冗的语法作为“惩罚”。

一个默认导出将一个特定的导出绑定设置为在这个模块被导入时的默认绑定。这个绑定的名称是字面上的 `default` 。正如你即将看到的，在导入模块绑定时你还可以重命名它们，你经常会对默认导出这么做。

每个模块定义只能有一个 `default` 。我们将在下一节中讲解 `import` ，你将看到如果模块拥有默认导入时 `import` 语法如何变得更简洁。

默认导出语法有一个微妙的细节你应当多加注意。比较这两个代码段：

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

export default foo;
```

和这一个：

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

export { foo as default };
```

在第一个代码段中，你导出的是那一个函数表达式在那一刻的值的绑定，不是 标识符 `foo` 的绑定。换句话说， `export default ..` 接收一个表达式。如果你稍后在你的模块内部赋给 `foo` 一个不同的值，这个模块导入将依然表示原本被导出的函数，而不是那个新的值。

顺带一提，第一个代码段还可以写做：

```
export default function foo(..) {
    // ..
}
```

警告： 虽然技术上讲这里的 `function foo..` 部分是一个函数表达式，但是对于模块内部作用域来说，它被视为一个函数声明，因为名称 `foo` 被绑定在模块的顶层作用域（经常称为“提升”）。对 `export default var foo = ..` 也是如此。然而，虽然你 可以 `export var foo = ..` ，但是一个令人沮丧的不一致是，你目前还不能 `export default bar foo = ..` （或者 `let` 和 `const` ）。在写作本书时，为了保持一致性，已经开始了在后ES6不久的时期增加这种能力的讨论。

再次回想一下第二个代码段：

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

export { foo as default };
```

这种版本的模块导出中，默认导出的绑定实际上是标识符 `foo` 而不是它的值，所以你会得到先前描述过的绑定行为（也就是，如果你稍后改变 `foo` 的值，在导入一端看到的值也会被更新）。

要非常小心这种默认导出语法的微妙区别，特别是在你的逻辑需要导出的值要被更新时。如果你永远

不打算更新一个默认导出的值，`export default ..` 就没问题。如果你确实打算更新这个值，你必须使用 `export { .. as default }` 。无论哪种情况，都要确保注释你的代码以解释你的意图！

因为一个模块只能有一个 `default` ，这可能会诱使你将你的模块设计为默认导出一个带有你所有API方法的对象，就像这样：

```
export default {
    foo() { .. },
    bar() { .. },
    ..
};
```

这种模式看起来十分接近于许多开发者构建它们的前ES6模块时曾经用过的模式，所以它看起来像是一种十分自然的方式。不幸的是，它有一些缺陷并且不为官方所鼓励使用。

特别是，JS引擎不能静态地分析一个普通对象的内容，这意味着它不能为静态 `import` 性能进行一些优化。使每个成员独立地并明确地导出的好处是，引擎 可以 进行静态分析和性能优化。

如果你的API已经有多于一个的成员，这些原则 —— 一个模块一个默认导出，和所有API成员作为被命名的导出 —— 看起来是冲突的，不是吗？但是你 可以 有一个单独的默认导出并且有其他的被命名导出；它们不是互相排斥的。

所以，取代这种（不被鼓励使用的）模式：

```
export default function foo() { .. }

foo.bar = function() { .. };
foo.baz = function() { .. };
```

你可以这样做：

```
export default function foo() { .. }

export function bar() { .. }
export function baz() { .. }
```

注意： 在前面这个代码段中，我为标记为 `default` 的函数使用了名称 `foo` 。但是，这个名称 `foo` 为了导出的目的而被忽略掉了 —— `default` 才是实际上被导出的名称。当你导入这个默认绑定时，你可以叫它任何你想用的名字，就像你将在下一节中看到的。

或者，一些人喜欢：

```
function foo() { .. }
function bar() { .. }
```

```
function baz() { .. }

export { foo as default, bar, baz, .. };
```

混合默认和被命名导出的效果将在稍后我们讲解 `import` 时更加清晰。但它实质上意味着最简洁的默认导入形式将仅仅取回 `foo()` 函数。用户可以额外地手动罗列 `bar` 和 `baz` 作为命名导入，如果他们想用它们的话。

你可能能够想象，如果你的模块有许多命名导出绑定，那么对于模块的消费者来说将有多么乏味。有一个通配符导入形式，你可以在一个名称空间对象中导入一个模块的所有导出，但是没有办法用通配符导入到顶层绑定。

要重申的是，ES6模块机制被有意设计为不鼓励带有许多导出的模块；相对而言，它被期望成为一种更困难一些的，作为某种社会工程的方式，以鼓励对大型/复杂模块设计有利的简单模块设计。

我将可能推荐你不要将默认导出与命名导出混在一起，特别是当你有一个大型API，并且将它重构为分离的模块是不现实或不希望的时候。在这种情况下，就都使用命名导出，并在文档中记录你的模块的消费者可能应当使用 `import * as ..` （名称空间导入，在下一节中讨论）方式来将整个API一次性地带到一个单独的名称空间中。

我们早先提到过这一点，但让我们回过头来更详细地讨论一下。除了导出一个表达式的值的绑定的 `export default ...` 形式，所有其他的导出形式都导出本地标识符的绑定。对于这些绑定，如果你在导出之后改变一个模块内部变量的值，外部被导入的绑定将可以访问这个被更新的值：

```
var foo = 42;
export { foo as default };

export var bar = "hello world";

foo = 10;
bar = "cool";
```

当你导出这个模块时， `default` 和 `bar` 导出将会绑定到本地变量 `foo` 和 `bar` ，这意味着它们将反映被更新的值 `10` 和 `"cool"` 。在被导出时的值是无关紧要的。在被导入时的值是无关紧要的。这些绑定是实时的链接，所以唯一重要的是当你访问这个绑定时它当前的值是什么。

警告： 双向绑定是不允许的。如果你从一个模块中导入一个 `foo` ，并试图改变你导入的变量 `foo` 的值，一个错误就会被抛出！我们将在下一节重新回到这个问题。

你还可以重新导出另一个模块的导出，比如：

```
export { foo, bar } from "baz";
export { foo as FOO, bar as BAR } from "baz";
export * from "baz";
```

这些形式都与首先从 `"baz"` 模块导入然后为了从你的模块中到处而明确地罗列它的成员相似。然而，在这些形式中，模块 `"baz"` 的成员从没有被导入到你的模块的本地作用域；某种程度上，它们原封不动地穿了过去。

`import`API成员

要导入一个模块，你将不出意料地使用 `import` 语句。就像 `export` 有几种微妙的变化一样， `import` 也有，所以你要花相当多的时间来考虑下面的问题，并试验你的选择。

如果你想要导入一个模块的API中的特定命名成员到你的顶层作用域，使用这种语法：

```
1. import { foo, bar, baz } from "foo";
```

警告： 这里的 `{ .. }` 语法可能看起来像一个对象字面量，甚至是像一个对象解构语法。但是，它的形式仅对模块而言是特殊的，所以不要将它与其他地方的 `{ .. }` 模式搞混了。

字符串 `"foo"` 称为一个 模块指示符。因为它的全部目的在于可以静态分析的语法，所以模块指示符必须是一个字符串字面量；它不能是一个持有字符串值的变量。

从你的ES6代码和JS引擎本身的角度来看，这个字符串字面量的内容是完全不透明和没有意义的。模块加载器将会把这个字符串翻译为一个在何处寻找被期望的模块的指令，不是作为一个URL路径就是一个本地文件系统路径。

被罗列的标识符 `foo` ， `bar` 和 `baz` 必须匹配在模块的API上的命名导出（这里将会发生静态分析和错误断言）。它们在你当前的作用域中被绑定为顶层标识符。

```
1. import { foo } from "foo";
2.
3. foo();
```

你可以重命名被导入的绑定标识符，就像：

```
1. import { foo as theFooFunc } from "foo";
2.
3. theFooFunc();
```

如果这个模块仅有一个你想要导入并绑定到一个标识符的默认导出，你可以为这个绑定选择性地跳过外围的 `{ .. }` 语法。在这种首选情况下 `import` 会得到最好的最简洁的 `import` 语法形式：

```
1. import foo from "foo";
2.
3. // 或者：
4. import { default as foo } from "foo";
```

注意： 正如我们在前一节中讲解过的，一个模块的 `export` 中的 `default` 关键字指定了一个名称实际上为 `default` 的命名导出，正如在第二个更加繁冗的语法中展示的那样。在这个例子中，从 `default` 到 `foo` 的重命名在后者的语法中是明确的，并且与前者隐含地重命名是完全相同的。

如果模块有这样的定义，你还可以与其他的命名导出一起导入一个默认导出。回忆一下先前的这个模块定义：

```
export default function foo() { .. }

export function bar() { .. }
export function baz() { .. }
```

要引入这个模块的默认导出和它的两个命名导出：

```
import FOOFN, { bar, baz as BAZ } from "foo";

FOOFN();
bar();
BAZ();
```

ES6的模块哲学强烈推荐的方式是，你只从一个模块中导入你需要的特定的绑定。如果一个模块提供10个API方法，但是你只需它们中的两个，有些人认为带入整套API绑定是一种浪费。

一个好处是，除了代码变得更加明确，收窄导入使得静态分析和错误检测（例如，不小心使用了错误的绑定名称）变得更加健壮。

当然，这只是受ES6设计哲学影响的标准观点；没有什么东西要求我们坚持这种方式。

许多开发者可能很快指出这样的方式更令人厌烦，每次你发现自己需要一个模块中的其他某些东西时，它要求你经常地重新找到并更新你的 `import` 语句。它的代价是牺牲便利性。

以这种观点看，首选方式可能是将模块中的所有东西都导入到一个单独的名称空间中，而不是将每个个别的成员直接导入到作用域中。幸运的是， `import` 语句拥有一个变种语法可以支持这种风格的模块使用，它被称为 名称空间导入。

考虑一个被这样导出的 `"foo"` 模块：

```
export function bar() { .. }
export var x = 42;
export function baz() { .. }
```

你可以将整个API导入到一个单独的模块名称空间绑定中：

```
import * as foo from "foo";
```

```
2.
3. foo.bar();
4. foo.x;           // 42
5. foo.baz();
```

注意： `* as ..` 子句要求使用 `*` 通配符。换句话说，你不能做像 `import { bar, x } as foo from "foo"` 这样的事情来将API的一部分绑定到 `foo` 名称空间。我会很喜欢这样的东西，但是对ES6的名称空间导入来说，要么全有要么全无。

如果你正在使用 `* as ..` 导入的模块拥有一个默认导出，它会在指定的名称空间中被命名为 `default` 。你可以在这个名称空间绑定的外面，作为一个顶层标识符额外地命名这个默认导出。考虑一个被这样导出的 `"world"` 模块：

```
1. export default function foo() { .. }
2. export function bar() { .. }
3. export function baz() { .. }
```

和这个 `import` ：

```
1. import foofn, * as hello from "world";
2.
3. foofn();
4. hello.default();
5. hello.bar();
6. hello.baz();
```

虽然这个语法是合法的，但是它可能令人困惑：这个模块的一个方法（那个默认导出）被绑定到你作用域的顶层，然而其他的命名导出（而且之中之一称为 `default` ）作为一个不同名称（ `hello` ）的标识符名称空间的属性被绑定。

正如我早先提到的，我的建议是避免这样设计你的模块导出，以降低你模块的用户受困于这些奇异之处的可能性。

所有被导入的绑定都是不可变和/或只读的。考虑前面的导入；所有这些后续的赋值尝试都将抛出 `TypeError` ：

```
1. import foofn, * as hello from "world";
2.
3. foofn = 42;            // (运行时) TypeError!
4. hello.default = 42;    // (运行时) TypeError!
5. hello.bar = 42;        // (运行时) TypeError!
6. hello.baz = 42;        // (运行时) TypeError!
```

回忆早先在“ `export` API成员”一节中，我们谈到 `bar` 和 `baz` 绑定是如何被绑定

到 `"world"` 模块内部的实际标识符上的。它意味着如果模块改变那些值， `hello.bar` 和 `hello.baz` 将引用更新后的值。

但是你的本地导入绑定的不可变/只读的性质强制你不能从被导入的绑定一方改变他们，不然就会发生 `TypeError` 。这很重要，因为如果没有这种保护，你的修改将会最终影响所有其他该模块的消费者（记住：单例），这可能会产生一些非常令人吃惊的副作用！

另外，虽然一个模块 可以 从内部改变它的API成员，但你应当对有意地以这种风格设计你的模块非常谨慎。ES6模块 被预计 是静态的，所以背离这个原则应当是不常见的，而且应当在文档中被非常小心和详细地记录下来。

警告： 存在一些这样的模块设计思想，你实际上打算允许一个消费者改变你的API上的一个属性的值，或者模块的API被设计为可以通过向API的名称空间中添加“插件”来“扩展”。但正如我们刚刚断言的，ES6模块API应当被认为并设计为静态的和不可变的，这强烈地约束和不鼓励那些其他的模块设计模式。你可以通过导出一个普通对象 —— 它理所当然是可以随意改变的 —— 来绕过这些限制。但是在选择这条路之前要三思而后行。

作为一个 `import` 的结果发生的声明将被“提升”（参见本系列的 作用域与闭包）。考虑如下代码：

```
1. foo();
2.
3. import { foo } from "foo";
```

`foo()` 可以运行是因为 `import ..` 语句的静态解析不仅在编译时搞清了 `foo` 是什么，它还将这个声明“提升”到模块作用域的顶部，如此使它在模块中通篇都是可用的。

最后，最基本的 `import` 形式看起来像这样：

```
1. import "foo";
```

这种形式实际上不会将模块的任何绑定导入到你的作用域中。它加载（如果还没被加载过），编译（如果还没被编译过），并对 `"foo"` 模块求值（如果还没被运行过）。

一般来说，这种导入可能不会特别有用。可能会有一些模块的定义拥有副作用（比如向 `window` /全局对象赋值）的特殊情况。你还可以将 `import "foo"` 用作稍后可能需要的模块的预加载。

模块循环依赖

A导入B。B导入A。这将如何工作？

我要立即声明，一般来说我会避免使用刻意的循环依赖来设计系统。话虽如此，我也认识到人们这么做是有原因的，而且它可以解决一些艰难的设计问题。

让我们考虑一下ES6如何处理这种情况。首先，模块 `"A"` ：

```
import bar from "B";

export default function foo(x) {
    if (x > 10) return bar( x - 1 );
    return x * 2;
}
```

现在，是模块 `"B"` ：

```
import foo from "A";

export default function bar(y) {
    if (y > 5) return foo( y / 2 );
    return y * 3;
}
```

这两个函数， `foo(..)` 和 `bar(..)` ，如果它们在相同的作用域中就会像标准的函数声明那样工作，因为声明被“提升”至整个作用域，而因此与它们的编写顺序无关，它们互相是可用的。

在模块中，你的声明在完全不同的作用域中，所以ES6必须做一些额外的工作以使这些循环引用工作起来。

在大致的概念上，这就是循环的 `import` 依赖如何被验证和解析的：

- 如果模块 `"A"` 被首先加载，第一步将是扫描文件并分析所有的导出，这样就可以为导入注册所有可用的绑定。然后它处理 `import .. from "B"` ，这指示它需要去取得 `"B"` 。
- 一旦引擎加载了 `"B"` ，它会做同样的导出绑定分析。当它看到 `import .. from "A"` 时，它知道 `"A"` 的API已经准备好了，所以它可以验证这个 `import` 为合法的。现在它知道了 `"B"` 的API，它也可以验证在模块 `"A"` 中等待的 `import .. from "B"` 了。

实质上，这种相互导入，连同对两个 `import` 语句合法性的静态验证，虚拟地组合了两个分离的模块作用域（通过绑定），因此 `foo(..)` 可以调用 `bar(..)` 或相反。这与我们在相同的作用域中声明是对称的。

现在让我们试着一起使用这两个模块。首先，我们将试用 `foo(..)` ：

```
import foo from "foo";
foo( 25 );              // 11
```

或者我们可以试用 `bar(..)` ：

```
import bar from "bar";
bar( 25 );              // 11.5
```

在 `foo(25)` 调用 `bar(25)` 被执行的时刻，所有模块的所有分析/编译都已经完成了。这意味着 `foo(..)` 内部地直接知道 `bar(..)`，而且 `bar(..)` 内部地直接知道 `foo(..)`。

如果所有我们需要的仅是与 `foo(..)` 互动，那么我们只需要导入 `"foo"` 模块。`bar(..)` 和 `"bar"` 模块也同理。

当然，如果我们想，我们 可以 导入并使用它们两个：

```
import foo from "foo";
import bar from "bar";

foo( 25 );              // 11
bar( 25 );              // 11.5
```

`import` 语句的静态加载语义意味着通过 `import` 互相依赖对方的 `"foo"` 和 `"bar"` 将确保在它们运行前被加载，解析，和编译。所以它们的循环依赖是被静态地解析的，而且将会如你所愿地工作。

模块加载

我们在“模块”这一节的最开始声称， `import` 语句使用了一个由宿主环境（浏览器，Node.js，等等）提供的分离的机制，来实际地将模块指示符字符串解析为一些对寻找和加载所期望模块的有用的指令。这种机制就是系统 模块加载器。

由环境提供的默认模块加载器，如果是在浏览器中将会把模块指示符解释为一个URL，如果是在服务器端（一般地）将会解释为一个本地文件系统路径，比如Node.js。它的默认行为是假定被加载的文件是以ES6标准的模块格式编写的。

另外，与当下脚本程序被加载的方式相似，你将可以通过一个HTML标签将一个模块加载到浏览器中。在本书写作时，这个标签将会是 `<script type="module">` 还是 `<module>` 还不完全清楚。ES6没有控制这个决定，但是在相应的标准化机构中的讨论早已随着ES6开始了。

无论这个标签看起来什么样，你可以确信它的内部将会使用默认加载器（或者一个你预先指定好的加载器，就像我们将在下一节中讨论的）。

就像你将在标记中使用的标签一样，ES6没有规定模块加载器本身。它是一个分离的，目前由WHATWG浏览器标准化小组控制的平行的标准。(http://whatwg.github.io/loader/)

在本书写作时，接下来的讨论反映了它的API设计的一个早期版本，和一些可能将要改变的东西。

加载模块之外的模块

一个与模块加载器直接交互的用法，是当一个非模块需要加载一个模块时。考虑如下代码：

```
// 在浏览器中通过`<script>`加载的普通script,
```

```
// `import`在这里是不合法的

Reflect.Loader.import( "foo" ) // 返回一个`"foo"`的promise
.then( function(foo){
    foo.bar();
} );
```

工具 `Reflect.Loader.import(..)` 将整个模块导入到命名参数中（作为一个名称空间），就像我们早先讨论过的 `import * as foo ..` 名称空间导入。

注意： `Reflect.Loader.import(..)` 返回一个promise，它在模块准备好时被完成。要导入多个模块的话，你可以使用 `Promise.all([ .. ])` 将多个 `Reflect.Loader.import(..)` 的promise组合起来。有关Promise的更多信息，参见第四章的“Promise”。

你还可以在一个真正的模块中使用 `Reflect.Loader.import(..)` 来动态地/条件性地加载一个模块，这是 `import` 自身无法做到的。例如，你可能在一个特性测试表明某个ES7+特性没有被当前的引擎所定义的情况下，选择性地加载一个含有此特性的填补的模块。

由于性能的原因，你将想要尽量避免动态加载，因为它阻碍了JS引擎从它的静态分析中提前获取的能力。

自定义加载

直接与模块加载器交互的另外一种用法是，你想要通过配置或者甚至是重定义来定制它的行为。

在本书写作时，有一个被开发好的模块加载器API的填补(https://github.com/ModuleLoader/es6-module-loader)。虽然关于它的细节非常匮乏，而且很可能改变，但是我们可以通过它来探索最终可能固定下来的东西是什么。

`Reflect.Loader.import(..)` 调用可能会支持第二个参数，它指定各种选项来定制导入/加载任务。例如：

```
Reflect.Loader.import( "foo", { address: "/path/to/foo.js" } )
.then( function(foo){
    // ..
} )
```

还有一种预期是，会为一个自定义内容提供某种机制来将之挂钩到模块加载的处理过程中，就在翻译/转译可能发生的加载之后，但是在引擎编译这个模块之前。

例如，你可能会加载某些还不是ES6兼容的模块格式的东西（例如，CoffeeScript，TypeScript，CommonJS，AMD）。你的翻译步骤可能会为了后面的引擎处理而将它转换为ES6兼容的模块。

类

- 类
  - `class`
  - `extends` 和 `super`
    - `super`的坑
    - 子类构造器
    - `extend`原生类型
  - `new.target`
  - `static`
    - `Symbol.species`构造器Getter

类

几乎从JavaScript的最开始的那时候起，语法和开发模式都曾努力（读作：挣扎地）地戴上一个支持面向类的开发的假面具。伴随着 `new` 和 `instanceof` 和一个 `.constructor` 属性，谁能不认为JS在它的原型系统的某个地方藏着类机制呢？

当然，JS的“类”与经典的类完全不同。其区别有很好的文档记录，所以在此我不会在这一点上花更多力气。

注意： 要学习更多关于在JS中假冒“类”的模式，以及另一种称为“委托”的原型的视角，参见本系列的 *this*与对象原型 的后半部分。

`class`

虽然JS的原型机制与传统的类的工作方式不同，但是这并不能阻挡一种强烈的潮流 —— 要求这门语言扩展它的语法糖以便将“类”表达得更像真正的类。让我们进入ES6 `class` 关键字和它相关的机制。

这个特性是一个具有高度争议、旷日持久的争论的结果，而且代表了几种对关于如何处理JS类的强烈反对意见的妥协的一小部分。大多数希望JS拥有完整的类机制的开发者将会发现新语法的一些部分十分吸引人，但是也会发现一些重要的部分仍然缺失了。但不要担心，TC39已经致力于另外的特性，以求在后ES6时代中增强类机制。

新的ES6类机制的核心是 `class` 关键字，它标识了一个 块，其内容定义了一个函数的原型的成员。考虑如下代码：

```
class Foo {
    constructor(a,b) {
        this.x = a;
        this.y = b;
    }

```

```
    gimmeXY() {
        return this.x * this.y;
    }
}
```

一些要注意的事情：

- `class Foo` 暗示着创建一个（特殊的）名为`Foo`的函数，与你在前ES6中所做的非常相似。
- `constructor(..)`表示了这个`Foo(..)`函数的签名，和它的函数体内容。
- 类方法同样使用对象字面量中可以使用的“简约方法”语法，正如在第二章中讨论过的。这也包括在本章早先讨论过的简约generator，以及ES5的getter/setter语法。但是，类方法是不可枚举的而对象方法默认是可枚举的。
- 与对象字面量不同的是，在一个`class`内容的部分没有逗号分隔各个成员！事实上，这甚至是不允许的。

前一个代码段的 `class` 语法定义可以大致认为和这个前ES6等价物相同，对于那些以前做过原型风格代码的人来说可能十分熟悉它：

```
function Foo(a,b) {
    this.x = a;
    this.y = b;
}

Foo.prototype.gimmeXY = function() {
    return this.x * this.y;
}
```

不管是前ES6形式还是新的ES6 `class` 形式，这个“类”现在可以被实例化并如你所想地使用了：

```
var f = new Foo( 5, 15 );

f.x;                        // 5
f.y;                        // 15
f.gimmeXY();                // 75
```

注意！虽然 `class Foo` 看起来很像 `function Foo()` ，但是有一些重要的区别：

- `class Foo`的一个`Foo(..)`调用 必须 与`new`一起使用，因为前ES6的`Foo.call( obj )`方式 不能 工作。
- 虽然`function Foo`会被“提升”（参见本系列的 作用域与闭包），但是`class Foo`不会；`extends ..`指定的表达式不能被“提升”。所以，在你能够实例化一个`class`之前必须先声明它。
- 在顶层全局作用域中的`class Foo`在这个作用域中创建了一个词法标识符`Foo`，但与此不同的是`function Foo`不会创建一个同名的全局对象属性。

已经建立的 `instanceof` 操作仍然可以与ES6的类一起工作，因为 `class` 只是创建了一个同名的构造器函数。然而，ES6引入了一个定制 `instanceof` 如何工作的方法，使用 `Symbol.hasInstance` （参见第七章的“通用Symbol”）。

我发现另一种更方便地考虑 `class` 的方法是，将它作为一个用来自动填充 `proptotype` 对象的 宏。可选的是，如果使用 `extends` （参见下一节）的话它还能连接 `[[Prototype]]` 关系。

其实一个ES6 `class` 本身不是一个实体，而是一个元概念，它包裹在其他具体实体上，例如函数和属性，并将它们绑在一起。

提示： 除了这种声明的形式，一个 `class` 还可以是一个表达式，就像： `var x = class Y { .. }` 。这主要用于将类的定义（技术上说，是构造器本身）作为函数参数值传递，或者将它赋值给一个对象属性。

`extends` 和 `super`

ES6的类还有一种语法糖，用于在两个函数原型之间建立 `[[Prototype]]` 委托链 —— 通常被错误地标记为“继承”或者令人困惑地标记为“原型继承” —— 使用我们熟悉的面向类的术语 `extends` ：

```
class Bar extends Foo {
    constructor(a,b,c) {
        super( a, b );
        this.z = c;
    }

    gimmeXYZ() {
        return super.gimmeXY() * this.z;
    }
}

var b = new Bar( 5, 15, 25 );

b.x;                        // 5
b.y;                        // 15
b.z;                        // 25
b.gimmeXYZ();                // 1875
```

一个有重要意义的新增物是 `super` ，它实际上在前ES6中不是直接可能的东西（不付出一些不幸的黑科技的代价的话）。在构造器中， `super` 自动指向“父构造器”，这在前一个例子中是 `Foo(..)` 。在方法中，它指向“父对象”，如此你就可以访问它上面的属性/方法，比如 `super.gimmeXY()` 。

`Bar extends Foo` 理所当然地意味着将 `Bar.prototype` 的 `[[Prototype]]` 链接到 `Foo.prototype` 。所以，在 `gimmeXYZ()` 这样的方法中的 `super` 特被地意味着 `Foo.prototype` ，而当 `super` 用在 `Bar` 构造器中时意味着 `Foo` 。

注意： `super` 不仅限于 `class` 声明。它也可以在对象字面量中工作，其方式在很大程度上与我们在此讨论的相同。更多信息参见第二章中的“对象 `super` ”。

`super` 的坑

注意到 `super` 的行为根据它出现的位置不同而不同是很重要的。公平地说，大多数时候这不是一个问题。但是如果你背离一个狭窄的规范，令人诧异的事情就会等着你。

可能会有这样的情况，你想在构造器中引用 `Foo.prototype` ，比如直接访问它的属性/方法之一。然而，在构造器中的 `super` 不能这样被使用； `super.prototype` 将不会工作。 `super(..)` 大致上意味着调用 `new Foo(..)` ，但它实际上不是一个可用的对 `Foo` 本身的引用。

与此对称的是，你可能想要在一个非构造器方法中引用 `Foo(..)` 函数。 `super.constructor` 将会指向 `Foo(..)` 函数，但是要小心这个函数 只能 与 `new` 一起被调用。 `new super.constructor(..)` 将是合法的，但是在大多数情况下它都不是很有用， 因为你不能使这个调用使用或引用当前的 `this` 对象环境，而这很可能是你想要的。

另外， `super` 看起来可能就像 `this` 一样是被函数的环境所驱动的 —— 也就是说，它们都是被动态绑定的。但是， `super` 不像 `this` 那样是动态的。当声明时一个构造器或者方法在它内部使用一个 `super` 引用时（在 `class` 的内容部分），这个 `super` 是被静态地绑定到这个指定的类阶层中的，而且不能被覆盖（至少是在ES6中）。

这意味着什么？这意味着如果你习惯于从一个“类”中拿来一个方法并通过覆盖它的 `this` ，比如使用 `call(..)` 或者 `apply(..)` ，来为另一个类而“借用”它的话，那么当你借用的方法中有一个 `super` 时，将很有可能发生令你诧异的事情。考虑这个类阶层：

```
1. class ParentA {
2.     constructor() { this.id = "a"; }
3.     foo() { console.log( "ParentA:", this.id ); }
4. }
5.
6. class ParentB {
7.     constructor() { this.id = "b"; }
8.     foo() { console.log( "ParentB:", this.id ); }
9. }
10.
11. class ChildA extends ParentA {
12.     foo() {
13.         super.foo();
14.         console.log( "ChildA:", this.id );
15.     }
16. }
17.
18. class ChildB extends ParentB {
19.     foo() {
20.         super.foo();
```

```
        console.log( "ChildB:", this.id );
    }
}

var a = new ChildA();
a.foo();                    // ParentA: a
                            // ChildA: a
var b = new ChildB();        // ParentB: b
b.foo();                    // ChildB: b
```

在前面这个代码段中一切看起来都相当自然和在意料之中。但是，如果你试着借来 `b.foo()` 并在 `a` 的上下文中使用它的话 —— 通过动态 `this` 绑定的力量，这样的借用十分常见而且以许多不同的方式被使用，包括最明显的mixin —— 你可能会发现这个结果出奇地难看：

```
// 在`a`的上下文环境中借用`b.foo()`
b.foo.call( a );            // ParentB: a
                            // ChildB: a
```

如你所见，引用 `this.id` 被动态地重绑定所以在两种情况下都报告 `: a` 而不是 `: b` 。但是 `b.foo()` 的 `super.foo()` 引用没有被动态重绑定，所以它依然报告 `ParentB` 而不是期望的 `ParentA` 。

因为 `b.foo()` 引用 `super` ，所以它被静态地绑定到了 `ChildB` / `ParentB` 阶层而不能被用于 `ChildA` / `ParentA` 阶层。在ES6中没有办法解决这个限制。

如果你有一个不带移花接木的静态类阶层，那么 `super` 的工作方式看起来很直观。但公平地说，实施带有 `this` 的编码的一个主要好处正是这种灵活性。简单地说， `class` + `super` 要求你避免使用这样的技术。

你能在对象设计上作出的选择归结为两个：使用这些静态的阶层 —— `class` ， `extends` ，和 `super` 将十分不错 —— 要么放弃所有“山寨”类的企图，而接受动态且灵活的，没有类的对象和 `[[Prototype]]` 委托（参见本系列的 *this*与对象原型）。

子类构造器

对类或子类来说构造器不是必需的；如果构造器被省略，这两种情况下都会有一个默认构造器顶替上来。但是，对于一个直接的类和一个被扩展的类来说，顶替上来的默认构造器是不同的。

特别地，默认的子类构造器自动地调用父构造器，并且传递所有参数值。换句话说，你可以认为默认的子类构造器有些像这样：

```
constructor(...args) {
    super(...args);
}
```

这是一个需要注意的重要细节。不是所有支持类的语言的子类构造器都会自动地调用父构造器。C++会，但Java不会。更重要的是，在前ES6的类中，这样的自动“父构造器”调用不会发生。如果你曾经依赖于这样的调用 不会 发生，按么当你将代码转换为ES6 `class` 时就要小心。

ES6子类构造器的另一个也许令人吃惊的偏差/限制是：在一个子类的构造器中，在 `super(..)` 被调用之前你不能访问 `this` 。其中的原因十分微妙和复杂，但是可以归结为是父构造器在实际上创建/初始化你的实例的 `this` 。前ES6中，它相反地工作； `this` 对象被“子类构造器”创建，然后你使用这个“子类”的 `this` 上下文环境调用“父构造器”。

让我们展示一下。这是前ES6版本：

```
function Foo() {
    this.a = 1;
}

function Bar() {
    this.b = 2;
    Foo.call( this );
}

// `Bar` “扩展” `Foo`
Bar.prototype = Object.create( Foo.prototype );
```

但是这个ES6等价物不允许：

```
class Foo {
    constructor() { this.a = 1; }
}

class Bar extends Foo {
    constructor() {
        this.b = 2;             // 在`super()`之前不允许
        super();            // 可以通过调换这两个语句修正
    }
}
```

在这种情况下，修改很简单。只要在子类 `Bar` 的构造器中调换两个语句的位置就行了。但是，如果你曾经依赖于前ES6可以跳过“父构造器”调用的话，就要小心这不再被允许了。

`extend` 原生类型

新的 `class` 和 `extend` 设计中最值得被欢呼的好处之一，就是（终于！）能够为内建原生类型，比如 `Array` ，创建子类。考虑如下代码：

```
class MyCoolArray extends Array {
```

```
    first() { return this[0]; }
    last() { return this[this.length - 1]; }
}

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

a.length;                    // 3
a;                           // [1,2,3]

a.first();                   // 1
a.last();                    // 3
```

在ES6之前，可以使用手动的对象创建并将它链接到 `Array.prototype` 来制造一个 `Array` 的“子类”的山寨版，但它仅能部分地工作。它缺失了一个真正数组的特殊行为，比如自动地更新 `length` 属性。ES6子类应该可以如我们盼望的那样使用“继承”与增强的行为来完整地工作！

另一个常见的前ES6“子类”的限制与 `Error` 对象有关，在创建自定义的错误“子类”时。当纯粹的 `Error` 被创建时，它们自动地捕获特殊的 `stack` 信息，包括错误被创建的行号和文件。前ES6的自定义错误“子类”没有这样的特殊行为，这严重地限制了它们的用处。

ES6前来拯救：

```
class Oops extends Error {
    constructor(reason) {
        super(reason);
        this.oops = reason;
    }
}

// 稍后:
var ouch = new Oops( "I messed up!" );
throw ouch;
```

前面代码段的 `ouch` 自定义错误对象将会向任何其他的纯粹错误对象那样动作，包括捕获 `stack`。这是一个巨大的改进！

`new.target`

ES6引入了一个称为 元属性 的新概念（见第七章），用 `new.target` 的形式表示。

如果这看起来很奇怪，是的；将一个带有 `.` 的关键字与一个属性名配成一对，对JS来说绝对是不同寻常的模式。

`new.target` 是一个在所有函数中可用的“魔法”值，虽然在普通的函数中它总是 `undefined`。在任意的构造器中， `new.target` 总是指向 `new` 实际直接调用的构造器，即便这个构造器是在一个父类中，而且是通过一个在子构造器中的 `super(..)` 调用被委托的。

```
class Foo {
    constructor() {
        console.log( "Foo: ", new.target.name );
    }
}

class Bar extends Foo {
    constructor() {
        super();
        console.log( "Bar: ", new.target.name );
    }
    baz() {
        console.log( "baz: ", new.target );
    }
}

var a = new Foo();
// Foo: Foo

var b = new Bar();
// Foo: Bar   <-- 遵照`new`的调用点
// Bar: Bar

b.baz();
// baz: undefined
```

`new.target` 元属性在类构造器中没有太多作用，除了访问一个静态属性/方法（见下一节）。

如果 `new.target` 是 `undefined`，那么你就知道这个函数不是用 `new` 调用的。然后你就可以强制一个 `new` 调用，如果有必要的话。

`static`

当一个子类 `Bar` 扩展一个父类 `Foo` 时，我们已经观察到 `Bar.prototype` 被 `[[Prototype]]` 链接到 `Foo.prototype`。但是额外地，`Bar()` 被 `[[Prototype]]` 链接到 `Foo()`。这部分可能就没有那么明显了。

但是，在你为一个类声明 `static` 方法（不只是属性）时它就十分有用，因为这些静态方法被直接添加到这个类的函数对象上，不是函数对象的 `prototype` 对象上。考虑如下代码：

```
class Foo {
    static cool() { console.log( "cool" ); }
    wow() { console.log( "wow" ); }
}

class Bar extends Foo {
```

```
    static awesome() {
        super.cool();
        console.log( "awesome" );
    }
    neat() {
        super.wow();
        console.log( "neat" );
    }
}

Foo.cool();                    // "cool"
Bar.cool();                    // "cool"
Bar.awesome();                // "cool"
                              // "awesome"

var b = new Bar();
b.neat();                     // "wow"
                              // "neat"

b.awesome;                    // undefined
b.cool;                       // undefined
```

小心不要被搞糊涂，认为 `static` 成员是在类的原型链上的。它们实际上存在与函数构造器中间的一个双重/平行链条上。

`Symbol.species` 构造器Getter

一个 `static` 可以十分有用的地方是为一个衍生（子）类设置 `Symbol.species` getter（在语言规范内部称为 `@@species` ）。这种能力允许一个子类通知一个父类应当使用什么样的构造器 —— 当不打算使用子类的构造器本身时 —— 如果有任何父类方法需要产生新的实例的话。

举个例子，在 `Array` 上的许多方法都创建并返回一个新的 `Array` 实例。如果你从 `Array` 定义一个衍生的类，但你想让这些方法实际上继续产生 `Array` 实例，而非从你的衍生类中产生实例，那么这就可以工作：

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

var a = new MyCoolArray( 1, 2, 3 ),
    b = a.map( function(v){ return v * 2; } );

b instanceof MyCoolArray;    // false
b instanceof Array;          // true
```

为了展示一个父类方法如何可以有些像 `Array#map(..)` 所做的那样，使用一个子类型声明，考虑如下代码：

```
class Foo {
    // 将`species`推迟到衍生的构造器中
    static get [Symbol.species]() { return this; }
    spawn() {
        return new this.constructor[Symbol.species]();
    }
}

class Bar extends Foo {
    // 强制`species`为父类构造器
    static get [Symbol.species]() { return Foo; }
}

var a = new Foo();
var b = a.spawn();
b instanceof Foo;                   // true

var x = new Bar();
var y = x.spawn();
y instanceof Bar;                   // false
y instanceof Foo;                   // true
```

父类的 `Symbol.species` 使用 `return this` 来推迟到任意的衍生类，就像你通常期望的那样。然后 `Bar` 手动地声明 `Foo` 被用于这样的实例创建。当然，一个衍生的类依然可以使用 `new this.constructor(..)` 生成它本身的实例。

复习

- 复习

复习

ES6引入了几个在代码组织上提供帮助的新特性：

- 迭代器提供了对数据和操作的序列化访问。它们可以被`for..of`和`...`这样的新语言特性消费。
- Generator是由一个迭代器控制的能够在本地暂停/继续的函数。它们可以被用于程序化地（并且是互动地，通过`yield`/`next(..)`消息传递） 生成 通过迭代器被消费的值。
- 模块允许实现的细节的私有封装带有一个公开导出的API。模块定义是基于文件的，单例的实例，并且在编译时静态地解析。
- 类为基于原型的编码提供了更干净的语法。`super`的到来也解决了在`[[Prototype]]`链中进行相对引用的刁钻问题。

在你考虑通过采纳ES6来改进你的JS项目体系结构时，这些新工具应当是你的第一站。

第四章：异步流程控制

- 第四章：异步流程控制

第四章：异步流程控制

如果你写过任何数量相当的JavaScript，这就不是什么秘密：异步编程是一种必须的技能。管理异步的主要机制曾经是函数回调。

然而，ES6增加了一种新特性：*Promise*，来帮助你解决仅使用回调来管理异步的重大缺陷。另外，我们可以重温generator（前一章中提到的）来看看一种将两者组合的模式，它是JavaScript中异步流程控制编程向前迈出的重要一步。

- Promises
- Generators + Promises
- 复习

Promises

- Promises
  - 创建与使用 Promises
  - Thenables
  - `Promise` API

Promises

让我们辨明一些误解：Promise不是回调的替代品。Promise提供了一种可信的中介机制 —— 也就是，在你的调用代码和将要执行任务的异步代码之间 —— 来管理回调。

另一种考虑Promise的方式是作为一种事件监听器，你可以在它上面注册监听一个通知你任务何时完成的事件。它是一个仅被触发一次的事件，但不管怎样可以被看作是一个事件。

Promise可以被链接在一起，它们可以是一系列顺序的、异步完成的步骤。与 `all(..)` 方法（用经典的术语将，叫“门”）和 `race(..)` 方法（用经典的术语将，叫“闩”）这样的高级抽象一起，promise链可以提供一种异步流程控制的机制。

还有另外一种概念化Promise的方式是，将它看作一个 未来值，一个与时间无关的值的容器。无论底层的值是否是最终值，这种容器都可以被同样地推理。观测一个Promise的解析会在这个值准备好的时候将它抽取出来。换言之，一个Promise被认为是一个同步函数返回值的异步版本。

一个Promise只可能拥有两种解析结果：完成或拒绝，并带有一个可选的信号值。如果一个Promise被完成，这个最终值称为一个完成值。如果它被拒绝，这个最终值称为理由（也就是“拒绝的理由”）。Promise只可能被解析（完成或拒绝）一次。任何其他的完成或拒绝的尝试都会被简单地忽略，一旦一个Promise被解析，它就成为一个不可被改变的值（immutable）。

显然，有几种不同的方式可以来考虑一个Promise是什么。没有一个角度就它自身来说是完全充分的，但是每一个角度都提供了整体的一个方面。这其中的要点是，它们为仅使用回调的异步提供了一个重大的改进，也就是它们提供了顺序、可预测性、以及可信性。

创建与使用 Promises

要构建一个promise实例，可以使用 `Promise(..)` 构造器：

```
var p = new Promise( function pr(resolve,reject){
    // ..
} );
```

`Promise(..)` 构造器接收一个单独的函数（ `pr(..)` ），它被立即调用并以参数值的形式收到两个控制函数，通常被命名为 `resolve(..)` 和 `reject(..)` 。它们被这样使用：

- 如果你调用`reject(..)`，promise就会被拒绝，而且如果有任何值被传入`reject(..)`，它就会被设置为拒绝的理由。
- 如果你不使用参数值，或任何非promise值调用`resolve(..)`，promise就会被完成。
- 如果你调用`resolve(..)`并传入另一个promise，这个promise就会简单地采用 —— 要么立即要么最终地 —— 这个被传入的promise的状态（不是完成就是拒绝）。

这里是你通常如何使用一个promise来重构一个依赖于回调的函数调用。假定你始于使用一个`ajax(..)`工具，它期预期要调用一个错误优先风格的回调：

```
function ajax(url,cb) {
    // 发起请求，最终调用 `cb(..)`
}

// ..

ajax( "http://some.url.1", function handler(err,contents){
    if (err) {
        // 处理ajax错误
    }
    else {
        // 处理成功的`contents`
    }
} );
```

你可以将它转换为：

```
function ajax(url) {
    return new Promise( function pr(resolve,reject){
        // 发起请求，最终不是调用 `resolve(..)` 就是调用 `reject(..)`
    } );
}

// ..

ajax( "http://some.url.1" )
.then(
    function fulfilled(contents){
        // 处理成功的 `contents`
    },
    function rejected(reason){
        // 处理ajax的错误reason
    }
);
```

Promise拥有一个方法`then(..)`，它接收一个或两个回调函数。第一个函数（如果存在的话）被看

作是promise被成功地完成时要调用的处理器。第二个函数（如果存在的话）被看作是promise被明确拒绝时，或者任何错误/异常在解析的过程中被捕捉到时要调用的处理器。

如果这两个参数值之一被省略或者不是一个合法的函数 —— 通常你会用 `null` 来代替 —— 那么一个占位用的默认等价物就会被使用。默认的成功回调将传递它的完成值，而默认的错误回调将传播它的拒绝理由。

调用 `then(null,handleRejection)` 的缩写是 `catch(handleRejection)` 。

`then(..)` 和 `catch(..)` 两者都自动地构建并返回另一个promise实例，它被链接在原本的promise上，接收原本的promise的解析结果 —— （实际被调用的）完成或拒绝处理器返回的任何值。考虑如下代码：

```
ajax( "http://some.url.1" )
.then(
    function fulfilled(contents){
        return contents.toUpperCase();
    },
    function rejected(reason){
        return "DEFAULT VALUE";
    }
)
.then( function fulfilled(data){
    // 处理来自于原本的promise的处理器中的数据
} );
```

在这个代码段中，我们要么从 `fulfilled(..)` 返回一个立即值，要么从 `rejected(..)` 返回一个立即值，然后在下一个事件周期中这个立即值被第二个 `then(..)` 的 `fulfilled(..)` 接收。如果我们返回一个新的promise，那么这个新promise就会作为解析结果被纳入与采用：

```
ajax( "http://some.url.1" )
.then(
    function fulfilled(contents){
        return ajax(
            "http://some.url.2?v=" + contents
        );
    },
    function rejected(reason){
        return ajax(
            "http://backup.url.3?err=" + reason
        );
    }
)
.then( function fulfilled(contents){
    // `contents` 来自于任意一个后续的 `ajax(..)` 调用
} );
```

要注意的是，在第一个 `fulfilled(..)` 中的一个异常（或者promise拒绝）将 不会 导致第一个 `rejected(..)` 被调用，因为这个处理仅会应答第一个原始的promise的解析。取代它的是，第二个 `then(..)` 调用所针对的第二个promise，将会收到这个拒绝。

在上面的代码段中，我们没有监听这个拒绝，这意味着它会为了未来的观察而被静静地保持下来。如果你永远不通过调用 `then(..)` 或 `catch(..)` 来观察它，那么它将会成为未处理的。有些浏览器的开发者控制台可能会探测到这些未处理的拒绝并报告它们，但是这不是有可靠保证的；你应当总是观察promise拒绝。

注意： 这只是Promise理论和行为的简要概览。要进行更加深入的探索，参见本系列的 异步与性能 的第三章。

Thenables

Promise是 `Promise(..)` 构造器的纯粹实例。然而，还存在称为 *thenable* 的类promise对象，它通常可以与Promise机制协作。

任何带有 `then(..)` 函数的对象（或函数）都被认为是一个thenable。任何Promise机制可以接受与采用一个纯粹的promise的状态的地方，都可以处理一个thenable。

Thenable基本上是一个一般化的标签，标识着任何由除了 `Promise(..)` 构造器之外的其他系统创建的类promise值。从这个角度上讲，一个thenable没有一个纯粹的Promise那么可信。例如，考虑这个行为异常的thenable：

```
var th = {
    then: function thener( fulfilled ) {
        // 永远会每100ms调用一次`fulfilled(..)`
        setInterval( fulfilled, 100 );
    }
};
```

如果你收到这个thenable并使用 `th.then(..)` 将它链接，你可能会惊讶地发现你的完成处理器被反复地调用，而普通的Promise本应该仅仅被解析一次。

一般来说，如果你从某些其他系统收到一个声称是promise或thenable的东西，你不应当盲目地相信它。在下一节中，我们将会看到一个ES6 Promise的工具，它可以帮助解决信任的问题。

但是为了进一步理解这个问题的危险，让我们考虑一下，在 任何 一段代码中的 任何 对象，只要曾经被定义为拥有一个称为 `then(..)` 的方法就都潜在地会被误认为是一个thenable —— 当然，如果和Promise一起使用的话 —— 无论这个东西是否有意与Promise风格的异步编码有一丝关联。

在ES6之前，对于称为 `then(..)` 的方法从来没有任何特别的保留措施，正如你能想象的那样，在Promise出现在雷达屏幕上之前就至少有那么几种情况，它已经被选择为方法的名称了。最有可能用

错thenable的情况就是使用 `then(..)` 的异步库不是严格兼容Promise的 —— 在市面上有好几种。

这份重担将由你来肩负：防止那些将被误认为一个thenable的值被直接用于Promise机制。

`Promise` API

`Promise` API还为处理Promise提供了一些静态方法。

`Promise.resolve(..)` 创建一个被解析为传入的值的promise。让我们将它的工作方式与更手动的方法比较一下：

```
var p1 = Promise.resolve( 42 );

var p2 = new Promise( function pr(resolve){
    resolve( 42 );
} );
```

`p1` 和 `p2` 将拥有完全相同的行为。使用一个promise进行解析也一样：

```
var theP = ajax( .. );

var p1 = Promise.resolve( theP );

var p2 = new Promise( function pr(resolve){
    resolve( theP );
} );
```

提示： `Promise.resolve(..)` 就是前一节提出的thenable信任问题的解决方案。任何你还不确定是一个可信promise的值 —— 它甚至可能是一个立即值 —— 都可以通过传入 `Promise.resolve(..)` 来进行规范化。如果这个值已经是一个可识别的promise或thenable，它的状态/解析结果将简单地被采用，将错误行为与你隔绝开。如果相反它是一个立即值，那么它将会被“包装”进一个纯粹的promise，以此将它的行为规范化为异步的。

`Promise.reject(..)` 创建一个立即被拒绝的promise，与它的 `Promise(..)` 构造器对等品一样：

```
var p1 = Promise.reject( "Oops" );

var p2 = new Promise( function pr(resolve,reject){
    reject( "Oops" );
} );
```

虽然 `resolve(..)` 和 `Promise.resolve(..)` 可以接收一个promise并采用它的状态/解析结果，但是 `reject(..)` 和 `Promise.reject(..)` 不会区分它们收到什么样的值。所以，如果你使用一个promise或thenable进行拒绝，这个promise/thenable本身将会被设置为拒绝的理由，而不是它

底层的值。

`Promise.all([ .. ])` 接收一个或多个值（例如，立即值，promise，thenable）的数组。它返回一个promise，这个promise会在所有的值完成时完成，或者在这些值中第一个被拒绝的值出现时被立即拒绝。

使用这些值/promises：

```
var p1 = Promise.resolve( 42 );
var p2 = new Promise( function pr(resolve){
    setTimeout( function(){
        resolve( 43 );
    }, 100 );
} );
var v3 = 44;
var p4 = new Promise( function pr(resolve,reject){
    setTimeout( function(){
        reject( "Oops" );
    }, 10 );
} );
```

让我们考虑一下使用这些值的组合，`Promise.all([ .. ])` 如何工作：

```
Promise.all( [p1,p2,v3] )
.then( function fulfilled(vals){
    console.log( vals );            // [42,43,44]
} );

Promise.all( [p1,p2,v3,p4] )
.then(
    function fulfilled(vals){
        // 永远不会跑到这里
    },
    function rejected(reason){
        console.log( reason );        // Oops
    }
);
```

`Promise.all([ .. ])` 等待所有的值完成（或第一个拒绝），而 `Promise.race([ .. ])` 仅会等待第一个完成或拒绝。考虑如下代码：

```
// 注意：为了避免时间的问题误导你，
// 重建所有的测试值！

Promise.race( [p2,p1,v3] )
.then( function fulfilled(val){
```

```
    console.log( val );                 // 42
} );

Promise.race( [p2,p4] )
.then(
    function fulfilled(val){
        // 永远不会跑到这里
    },
    function rejected(reason){
        console.log( reason );          // Oops
    }
);
```

警告： 虽然 `Promise.all([])` 将会立即完成（没有任何值），但是 `Promise.race([])` 将会被永远挂起。这是一个奇怪的不一致，我建议你应当永远不要使用空数组调用这些方法。

Generators + Promises

Generators + Promises

将一系列promise在一个链条中表达来代表你程序的异步流程控制是 可能 的。考虑如如下代码：

```
step1()
.then(
    step2,
    step1Failed
)
.then(
    function step3(msg) {
        return Promise.all( [
            step3a( msg ),
            step3b( msg ),
            step3c( msg )
        ] )
    }
)
.then(step4);
```

但是对于表达异步流程控制来说有更好的选项，而且在代码风格上可能比长长的promise链更理想。我们可以使用在第三章中学到的generator来表达我们的异步流程控制。

要识别一个重要的模式：一个generator可以yield出一个promise，然后这个promise可以使用它的完成值来推进generator。

考虑前一个代码段，使用generator来表达：

```
function *main() {

    try {
        var ret = yield step1();
    }
    catch (err) {
        ret = yield step1Failed( err );
    }

    ret = yield step2( ret );

    // step 3
    ret = yield Promise.all( [
```

```
        step3a( ret ),
        step3b( ret ),
        step3c( ret )
    ] );

    yield step4( ret );
}
```

从表面上看，这个代码段要比前一个promise链等价物要更繁冗。但是它提供了更加吸引人的 —— 而且重要的是，更加容易理解和阅读的 —— 看起来同步的代码风格（“return”值的 `=` 赋值操作，等等），对于 `try..catch` 错误处理可以跨越那些隐藏的异步边界使用来说就更是这样。

为什么我们要与generator一起使用Promise？不用Promise进行异步generator编码当然是可能的。

Promise是一个可信的系统，它将普通的回调和thunk中发生的控制倒转（参见本系列的 异步与性能）反转回来。所以组合Promise的可信性与generator中代码的同步性有效地解决了回调的主要缺陷。另外，像 `Promise.all([ .. ])` 这样的工具是一个非常美好、干净的方式 —— 在一个generator的一个 `yield` 步骤中表达并发。

那么这种魔法是如何工作的？我们需要一个可以运行我们generator的 运行器（*runner*），接收一个被 `yield` 出来的promise并连接它，让它要么使用成功的完成推进generator，要么使用拒绝的理由向generator抛出异常。

许多具备异步能力的工具/库都有这样的“运行器”；例如， `Q.spawn(..)` 和我的asynquence中的 `runner(..)` 插件。这里有一个独立的运行器来展示这种处理如何工作：

```
function run(gen) {
    var args = [].slice.call( arguments, 1), it;

    it = gen.apply( this, args );

    return Promise.resolve()
        .then( function handleNext(value){
            var next = it.next( value );

            return (function handleResult(next){
                if (next.done) {
                    return next.value;
                }
                else {
                    return Promise.resolve( next.value )
                        .then(
                            handleNext,
                            function handleErr(err) {
                                return Promise.resolve(
```

```
                                    it.throw( err )
                                )
                                .then( handleResult );
                            }
                        );
                }
            })( next );
        } );
}
```

注意： 这个工具的更丰富注释的版本，参见本系列的 异步与性能。另外，由各种异步库提供的这种运行工具通常要比我们在这里展示的东西更强大。例如，asynquence的 `runner(..)` 可以处理被 `yield` 的promise、序列、thunk、以及（非promise的）间接值，给你终极的灵活性。

于是现在运行早先代码段中的 `*main()` 就像这样容易：

```
run( main )
.then(
    function fulfilled(){
        // `*main()` 成功地完成了
    },
    function rejected(reason){
        // 噢，什么东西搞错了
    }
);
```

实质上，在你程序中的任何拥有多于两个异步步骤的流程控制逻辑的地方，你就可以 而且应当 使用一个由运行工具驱动的promise-yielding generator来以一种同步的风格表达流程控制。这样做将产生更易于理解和维护的代码。

这种“让出一个promise推进generator”的模式将会如此常见和如此强大，以至于ES6之后的下一个版本的JavaScript几乎可以确定将会引入一中新的函数类型，它无需运行工具就可以自动地执行。我们将在第八章中讲解 `async function` （正如它们期望被称呼的那样）。

复习

复习

随着JavaScript在它被广泛采用过程中的日益成熟与成长，异步编程越发地成为关注的中心。对于这些异步任务来说回调并不完全够用，而且在更精巧的需求面前全面崩塌了。

可喜的是，ES6增加了Promise来解决回调的主要缺陷之一：在可预测的行为上缺乏可信性。Promise代表一个潜在异步任务的未来完成值，跨越同步和异步的边界将行为进行了规范化。

但是，Promise与generator的组合才完全揭示了这样做的好处：将我们的异步流程控制代码重新安排，将难看的回调浆糊（也叫“地狱”）弱化并抽象出去。

目前，我们可以在各种异步库的运行器的帮助下管理这些交互，但是JavaScript最终将会使用一种专门的独立语法来支持这种交互模式！

第五章：集合

- 第五章：集合

第五章：集合

结构化的集合与数据访问对于任何JS程序来说都是一个关键组成部分。从这门语言的最开始到现在，数组和对象一直都是我们创建数据结构的主要机制。当然，许多更高级的数据结构作为用户方的库都曾建立在这些之上。

到了ES6，最有用（而且优化性能的！）的数据结构抽象中的一些已经作为这门语言的原生组件被加入了进来。

我们将通过检视 类型化数组（*TypedArrays*） 来开始这一章，技术上讲它与几年前的ES5是同一时期的产物，但是仅仅作为WebGL的同伴被标准化了，而不是作为JavaScript自身的一部分。到了ES6，这些东西已经被语言规范直接采纳了，这给予了它们头等的地位。

Map就像对象（键/值对），但是与仅能使用一个字符串作为键不同的是，你可以使用任何值 —— 即使是另一个对象或map！Set与数组很相似（值的列表），但是这些值都是唯一的；如果你添加一个重复的值，它会被忽略。还有与之相对应的weak结构（与内存/垃圾回收有关联）：WeakMap和WeakSet。

- 类型化数组（TypedArrays）
- Maps
- WeakMaps
- Sets
- WeakSets
- 复习

类型化数组（TypedArrays）

- 类型化数组（TypedArrays）
  - 字节顺序
  - 多视图
  - 类型化数组构造器

类型化数组（TypedArrays）

正如我们在本系列的 类型与文法 中讲到过的，JS确实拥有一组内建类型，比如 `number` 和 `string`。看到一个称为“类型化的数组”的特性，可能会诱使你推测它意味着一个特定类型的值的数组，比如一个仅含字符串的数组。

然而，类型化数组其实更多的是关于使用类似数组的语义（索引访问，等等）提供对二进制数据的结构化访问。名称中的“类型”指的是在大量二进制位（比特桶）的类型之上覆盖的“视图”，它实质上是一个映射，控制着这些二进制位是否应当被看作8位有符号整数的数组，还是被看作16位有符号整数的数组，等等。

你怎样才能构建这样的比特桶呢？它被称为一个“缓冲（buffer）”，而你可以用 `ArrayBuffer(..)` 构造器直接地构建它：

```
var buf = new ArrayBuffer( 32 );
buf.byteLength;                         // 32
```

现在 `buf` 是一个长度为32字节（256比特）的二进制缓冲，它被预初始化为全 `0`。除了检查它的 `byteLength` 属性，一个缓冲本身不会允许你进行任何操作。

提示： 有几种web平台特性都使用或返回缓冲，比如 `FileReader#readAsArrayBuffer(..)`， `XMLHttpRequest#send(..)`，和 `ImageData`（canvas数据）。

但是在这个数组缓冲的上面，你可以平铺一层“视图”，它就是用类型化数组的形式表现的。考虑如下代码：

```
var arr = new Uint16Array( buf );
arr.length;                             // 16
```

`arr` 是一个256位的 `buf` 缓冲在16位无符号整数的类型化数组的映射，意味着你得到16个元素。

字节顺序

明白一个事实非常重要： `arr` 是使用JS所运行的平台的字节顺序设定（大端法或小端法）被映射的。如果二进制数据是由一种字节顺序创建，但是在一个拥有相反字节数序的平台被解释时，这就可能是个问题。

字节顺序指的是一个多字节数字的低位字节（8个比特位的集合） —— 比如我们在早先的代码段中创建的16位无符号整数 —— 是在这个数字的字节序列的左边还是右边。

举个例子，让我们想象一下用16位来表示的10进制的数字 `3085` 。如果你只有一个16位数字的容器，无论字节顺序怎样它都将以二进制表示为 `0000110000001101` （十六进制的 `0c0d` ）。

但是如果 `3085` 使用两个8位数字来表示的话，字节顺序就像会极大地影响它在内存中的存储：

- `0000110000001101` / `0c0d` （大端法）
- `0000110100001100` / `0d0c` （小端法）

如果你从一个小端法系统中收到表示为 `0000110100001100` 的 `3085` ，但是在一个大端法系统中为它上面铺一层视图，那么你将会看到值 `3340` （10进制）和 `0d0c` （16进制）。

如今在web上最常见的表现形式是小端法，但是绝对存在一些与此不同的浏览器。你明白一块二进制数据的生产者和消费者的字节顺序是十分重要的。

在MDN上有一种快速的方法测试你的JavaScript的字节顺序：

```
1. var littleEndian = (function() {
2.     var buffer = new ArrayBuffer( 2 );
3.     new DataView( buffer ).setInt16( 0, 256, true );
4.     return new Int16Array( buffer )[0] === 256;
5. })();
```

`littleEndian` 将是 `true` 或 `false` ；对大多数浏览器来说，它应当返回 `true` 。这个测试使用 `DataView(..)` ，它允许更底层，更精细地控制如何从你平铺在缓冲上的视图中访问二进制位。前面代码段中的 `setInt16(..)` 方法的第三个参数告诉 `DataView` ，对于这个操作你想使用什么字节顺序。

警告： 不要将一个数组缓冲中底层的二进制存储的字节顺序与一个数字在JS程序中被暴露时如何被表示搞混。举例来说， `(3085).toString(2)` 返回 `"110000001101"` ，它被假定前面有四个 `"0"` 因而是大端法表现形式。事实上，这个表现形式是基于一个单独的16位视图的，而不是两个8位字节的视图。上面的 `DataView` 测试是确定你的JS环境的字节顺序的最佳方法。

多视图

一个单独的缓冲可以连接多个视图，例如：

```
1. var buf = new ArrayBuffer( 2 );
2.
```

```
var view8 = new Uint8Array( buf );
var view16 = new Uint16Array( buf );

view16[0] = 3085;
view8[0];                          // 13
view8[1];                          // 12

view8[0].toString( 16 );         // "d"
view8[1].toString( 16 );         // "c"

// 调换（好像字节顺序一样！）
var tmp = view8[0];
view8[0] = view8[1];
view8[1] = tmp;

view16[0];                         // 3340
```

类型化数组的构造器拥有多种签名。目前我们展示过的只是向它们传递一个既存的缓冲。然而，这种形式还接受两个额外的参数： `byteOffset` 和 `length` 。换句话讲，你可以从 `0` 以外的位置开始类型化数组视图，也可以使它的长度小于整个缓冲的长度。

如果二进制数据的缓冲包含规格不一的大小/位置，这种技术可能十分有用。

例如，考虑一个这样的二进制缓冲：在开头拥有一个2字节数字（也叫做“字”），紧跟着两个1字节数字，然后跟着一个32位浮点数。这是你如何在同一个缓冲，偏移量，和长度上使用多视图来访问数据：

```
var first = new Uint16Array( buf, 0, 2 )[0],
    second = new Uint8Array( buf, 2, 1 )[0],
    third = new Uint8Array( buf, 3, 1 )[0],
    fourth = new Float32Array( buf, 4, 4 )[0];
```

类型化数组构造器

除了前一节我们检视的 `(buffer,[offset, [length]])` 形式之外，类型化数组的构造器还支持这些形式：

- [constructor] `(length)` ：在一个长度为 `length` 字节的缓冲上创建一个新视图
- [constructor] `(typedArr)` ：创建一个新视图和缓冲，并拷贝 `typedArr` 视图中的内容
- [constructor] `(obj)` ：创建一个新视图和缓冲，并迭代类数组或对象 `obj` 来拷贝它的内容

在ES6中可以使用下面的类型化数组构造器：

- `Int8Array` （8位有符号整数）， `Uint8Array` （8位无符号整数）
  - `Uint8ClampedArray` （8位无符号整数，每个值都被卡在 `0 - 255` 范围内）

- `Int16Array`（16位有符号整数），`Uint16Array`（16位无符号整数）
- `Int32Array`（32位有符号整数），`Uint32Array`（32位无符号整数）
- `Float32Array`（32位浮点数，IEEE-754）
- `Float64Array`（64位浮点数，IEEE-754）

类型化数组构造器的实例基本上和原生的普通数组是一样的。一些区别包括它有一个固定的长度并且值都是同种“类型”。

但是，它们共享绝大多数相同的 `prototype` 方法。这样一来，你很可能将会像普通数组那样使用它们而不必进行转换。

例如：

```
var a = new Int32Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

a.map( function(v){
    console.log( v );
} );
// 10 20 30

a.join( "-" );
// "10-20-30"
```

警告： 你不能对类型化数组使用没有意义的特定 `Array.prototype` 方法，比如修改器（`splice(..)`，`push(..)`，等等）和 `concat(..)`。

要小心，在类型化数组中的元素被限制在它被声明的位长度中。如果你有一个 `Uint8Array` 并试着向它的一个元素赋予某些大于8为的值，那么这个值将被截断以保持在相应的位长度中。

这可能造成一些问题，例如，如果你试着对一个类型化数组中的所有值求平方。考虑如下代码：

```
var a = new Uint8Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

var b = a.map( function(v){
    return v * v;
} );

b;              // [100, 144, 132]
```

在被平方后，值 `20` 和 `30` 的结果会位溢出。要绕过这样的限制，你可以使用 `TypedArray#from(..)` 函数：

```
var a = new Uint8Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

var b = Uint16Array.from( a, function(v){
    return v * v;
} );

b;                // [100, 400, 900]
```

关于被类型化数组所共享的 `Array.from(..)` 函数的更多信息，参见第六章的“ `Array.from(..)` 静态方法”一节。特别地，“映射”一节讲解了作为第二个参数值被接受的映射函数。

一个值得考虑的有趣行为是，类型化数组像普通数组一样有一个 `sort(..)` 方法，但是这个方法默认是数字排序比较而不是将值强制转换为字符串进行字典顺序比较。例如：

```
var a = [ 10, 1, 2, ];
a.sort();                               // [1,10,2]

var b = new Uint8Array( [ 10, 1, 2 ] );
b.sort();                               // [1,2,10]
```

就像 `Array#sort(..)` 一样， `TypedArray#sort(..)` 接收一个可选的比较函数作为参数值，它们的工作方式完全一样。

Maps

- Maps
  - Map 值
  - Map 键

Maps

如果你对JS经验丰富，那么你一定知道对象是创建无序键/值对数据结构的主要机制，这也被称为map。然而，将对象作为map的主要缺陷是不能使用一个非字符串值作为键。

例如，考虑如下代码：

```
var m = {};

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

m[x] = "foo";
m[y] = "bar";

m[x];                           // "bar"
m[y];                           // "bar"
```

这里发生了什么？ `x` 和 `y` 这两个对象都被字符串化为 `"[object Object]"` ，所以只有这一个键被设置为 `m` 。

一些人通过在一个值的数组旁边同时维护一个平行的非字符串键的数组实现了山寨的map，比如：

```
var keys = [], vals = [];

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

keys.push( x );
vals.push( "foo" );

keys.push( y );
vals.push( "bar" );

keys[0] === x;                  // true
vals[0];                        // "foo"

keys[1] === y;                  // true
```

```
vals[1];                    // "bar"
```

当然，你不会想亲自管理这些平行数组，所以你可能会定义一个数据解构，使它内部带有自动管理的方法。除了你不得不自己做这些工作，主要的缺陷是访问的时间复杂度不再是O(1)，而是O(n)。

但在ES6中，不再需要这么做了！使用 `Map(..)` 就好：

```
var m = new Map();

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

m.set( x, "foo" );
m.set( y, "bar" );

m.get( x );                 // "foo"
m.get( y );                 // "bar"
```

唯一的缺点是你不能使用 `[]` 方括号访问语法来设置或取得值。但是 `get(..)` 和 `set(..)` 可以完美地取代这种语法。

要从一个map中删除一个元素，不要使用 `delete` 操作符，而是使用 `delete(..)` 方法：

```
m.set( x, "foo" );
m.set( y, "bar" );

m.delete( y );
```

使用 `clear()` 你可清空整个map的内容。要得到map的长度（也就是，键的数量），使用 `size` 属性（不是 `length` ）。

```
m.set( x, "foo" );
m.set( y, "bar" );
m.size;                     // 2

m.clear();
m.size;                     // 0
```

`Map(..)` 的构造器还可以接受一个可迭代对象（参见第三章的“迭代器”），它必须产生一个数组的列表，每个数组的第一个元素是键，第二元素是值。这种用于迭代的格式与 `entries()` 方法产生的格式是一样的， `entries()` 方法将在下一节中讲解。这使得制造一个map的拷贝十分简单：

```
var m2 = new Map( m.entries() );

```

```
// 等同于：
var m2 = new Map( m );
```

因为一个map实例是一个可迭代对象，而且它的默认迭代器与 `entries()` 相同，第二种稍短的形式更理想。

当然，你可以在 `Map(..)` 构造器形式中手动指定一个 *entries* 列表：

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

var m = new Map( [
    [ x, "foo" ],
    [ y, "bar" ]
] );

m.get( x );                         // "foo"
m.get( y );                         // "bar"
```

Map 值

要从一个map得到值的列表，使用 `values(..)` ，它返回一个迭代器。在第二和第三章，我们讲解了几种序列化（像一个数组那样）处理一个迭代器的方法，比如 `...` 扩散操作符和 `for..of` 循环。另外，第六章的“Arrays”将会详细讲解 `Array.from(..)` 方法。考虑如下代码：

```
var m = new Map();

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

m.set( x, "foo" );
m.set( y, "bar" );

var vals = [ ...m.values() ];

vals;                               // ["foo","bar"]
Array.from( m.values() );           // ["foo","bar"]
```

就像在前一节中讨论过的，你可以使用 `entries()` （或者默认的map迭代器）迭代一个map的记录。考虑如下代码：

```
var m = new Map();

var x = { id: 1 },
```

```
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var vals = [ ...m.entries() ];

vals[0][0] === x;               // true
vals[0][1];                       // "foo"

vals[1][0] === y;               // true
vals[1][1];                       // "bar"
```

Map 键

要得到键的列表，使用 `keys()` ，它返回一个map中键的迭代器：

```
var m = new Map();

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

m.set( x, "foo" );
m.set( y, "bar" );

var keys = [ ...m.keys() ];

keys[0] === x;                   // true
keys[1] === y;                   // true
```

要判定一个map中是否拥有一个给定的键，使用 `has(..)` ：

```
var m = new Map();

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

m.set( x, "foo" );

m.has( x );                       // true
m.has( y );                       // false
```

实质上map让你将一些额外的信息（值）与一个对象（键）相关联，而不用实际上将这些信息放在对象本身中。

虽然在一个map中你可以使用任意种类的值作为键，但是你经常使用的将是对象，就像字符串和其他在普通对象中可以合法地作为键的基本类型。换句话说，你可能将想要继续使用普通对象，除非一些或全部的键需要是对象，在那种情况下map更合适。

警告： 如果你使用一个对象作为一个map键，而且这个对象稍后为了能够被垃圾回收器（GC）回收它占用的内存而被丢弃（解除所有的引用），那么map本身将依然持有它的记录。你需要从map中移除这个记录来使它能够被垃圾回收。在下一节中，我们将看到对于作为对象键和GC来说更好的选择 —— WeakMaps。

WeakMaps

- WeakMaps

WeakMaps

WeakMap是map的一个变种，它们的大多数外部行为是相同的，而在底层内存分配（明确地说是它的GC）如何工作上有区别。

WeakMap（仅）接收对象作为键。这些对象被 弱 持有，这意味着如果对象本身被垃圾回收掉了，那么在WeakMap中的记录也会被移除。这是观察不到的，因为一个对象可以被垃圾回收的唯一方法是不再有指向它的引用 —— 一旦不再有指向它的引用，你就没有对象引用可以用来检查它是否存在于这个WeakMap中。

除此以外，WeakMap的API是相似的，虽然限制更多：

```
var m = new WeakMap();

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

m.set( x, "foo" );

m.has( x );                         // true
m.has( y );                         // false
```

WeakMap没有 `size` 属性和 `clear()` 方法，它们也不对它们的键，值和记录暴露任何迭代器。所以即便你解除了 `x` 引用，它将会因GC从 `m` 中移除它的记录，也没有办法确定这一事实。你只能相信JavaScript会这么做！

就像map一样，WeakMap让你将信息与一个对象软关联。如果你不能完全控制这个对象，比如DOM元素，它们就特别有用。如果你用做map键的对象可以被删除并且应当在被删除时成为GC的回收对象，那么一个WeakMap就是更合适的选项。

要注意的是WeakMap只弱持有它的 键，而不是它的值。考虑如下代码：

```
var m = new WeakMap();

var x = { id: 1 },
    y = { id: 2 },
    z = { id: 3 },
    w = { id: 4 };

```

```
8.  m.set( x, y );
9.
10. x = null;                          // { id: 1 } 是可以GC的
11. y = null;                          // 由于 { id: 1 } 是可以GC的，因此 { id: 2 } 也可以
12.
13. m.set( z, w );
14.
15. w = null;                          // { id: 4 } 不可以GC
```

因此，我认为WeakMap被命名为“WeakKeyMap”更好。

Sets

- Sets
  - Set 迭代器

Sets

一个set是一个集合，其中的值都是唯一的（重复的会被忽略）。

set的API与map很相似。 `add(..)` 方法（有点讽刺地）取代了 `set(..)` ，而且没有 `get(..)` 方法。

考虑如下代码：

```
var s = new Set();

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

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

s.size;                             // 2

s.delete( y );
s.size;                             // 1

s.clear();
s.size;                             // 0
```

`Set(..)` 构造器形式与 `Map(..)` 相似，它可以接收一个可迭代对象，比如另一个set或者一个值的数组。但是，与 `Map(..)` 期待一个 记录 的列表（键/值数组的数组）不同的是， `Set(..)` 期待一个 值 的列表（值的数组）：

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

var s = new Set( [x,y] );
```

一个set不需要 `get(..)` ，因为你不会从一个set中取得值，而是使用 `has(..)` 测试一个值是否存在：

```
var s = new Set();

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

s.add( x );

s.has( x );                     // true
s.has( y );                     // false
```

注意： `has(..)` 中的比较算法与 `Object.is(..)` （见第六章）几乎完全相同，除了 `-0` 和 `0` 被视为相同而非不同。

Set 迭代器

set和map一样拥有相同的迭代器方法。set的行为有所不同，但是与map的迭代器的行为是对称的。考虑如下代码：

```
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()` 迭代器都会给出set中唯一值的列表。 `entries()` 迭代器给出记录数组的列表，记录数组中的两个元素都是唯一的set值。一个set的默认迭代器是它的 `values()` 迭代器。

一个set天生的唯一性是它最有用的性质。例如：

```
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。

WeakSets

- WeakSets

WeakSets

一个WeakMap弱持有它的键（但强持有它的值），而一个WeakSet弱持有它的值（不存在真正的键）。

```
var s = new WeakSet();

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

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

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

警告： WeakSet的值必须是对象，在set中被允许的基本类型值是不行的。

复习

复习

ES6定义了几种有用的集合，它们使得处理解构化的数据更加高效和有效。

类型化数组提供了二进制数据缓冲的“视图”，它使用各种整数类型对齐，比如8位无符号整数和32位浮点数。二进制数据的数组访问使得操作更加容易表达和维护，它可以让你更简单地处理如视频，音频，canvas数据等复杂的数组。

Map是键-值对集合，它的键可以是对象而非只可以是字符串/基本类型。Set是（任何类型的）唯一值的列表。

WeakMap是键（对象）被弱持有的map，所以如果它是最后一个指向这个对象的引用，GC就可以自由地回收这个记录。WeakSet是值被弱持有的set，所以同样地，如果它是最后一个指向这个对象的引用，GC就可以移除这个记录。

第六章：新增 API

- 第六章：新增API

第六章：新增API

从值的转换到数学计算，ES6给各种内建原生类型和对象增加了许多静态属性和方法来辅助这些常见任务。另外，一些原生类型的实例通过各种新的原型方法获得了新的能力。

注意： 大多数这些特性都可以被忠实地填补。我们不会在这里深入这样的细节，但是关于兼容标准的shim/填补，你可以看一下“ES6 Shim”(https://github.com/paulmillr/es6-shim/)。

- Array
- Object
- Math
- Number
- String
- 复习

Array

- `Array`
  - `Array.of(..)` 静态函数
  - `Array.from(..)` 静态函数
    - 避免空值槽
    - 映射
  - 创建 Arrays 和子类型
  - `copyWithin(..)` 原型方法
  - `fill(..)` 原型方法
  - `find(..)` 原型方法
  - `findIndex(..)` 原型方法
  - `entries()`, `values()`, `keys()` 原型方法

`Array`

在JS中被各种用户库扩展得最多的特性之一就是数组类型。ES6在数组上增加许多静态的和原型（实例）的帮助功能应当并不令人惊讶。

`Array.of(..)` 静态函数

`Array(..)` 的构造器有一个尽人皆知的坑：如果仅有一个参数值被传递，而且这个参数值是一个数字的话，它并不会制造一个含有一个带有该数值元素的数组，而是构建一个长度等于这个数字的空数组。这种操作造成了不幸的和怪异的“空值槽”行为，而这正是JS数组为人诟病的地方。

`Array.of(..)` 作为数组首选的函数型构造器取代了 `Array(..)`，因为 `Array.of(..)` 没有那种单数字参数值的情况。考虑如下代码：

```
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]
```

在什么样的环境下，你才会想要是使用 `Array.of(..)` 来创建一个数组，而不是使用像 `c = [1,2,3]` 这样的字面语法呢？有两种可能的情况。

如果你有一个回调，传递给它的参数值本应当被包装在一个数组中时，`Array.of(..)` 就完美地符合条件。这可能不是那么常见，但是它可以为你的痒处挠上一把。

另一种场景是如果你扩展 `Array` 构成它的子类，而且希望能够在一个你的子类的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 -- 噢！
x.sum();                        // 0 -- 噢！

var y = [3];                    // Array, 不是 MyCoolArray
y.length;                       // 1
y.sum();                        // `sum` is not a function

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

你不能（简单地）只创建一个 `MyCoolArray` 的构造器，让它覆盖 `Array` 父构造器的行为，因为这个父构造器对于实际创建一个规范的数组值（初始化 `this` ）是必要的。在 `MyCoolArray` 子类上“被继承”的静态 `of(..)` 方法提供了一个不错的解决方案。

`Array.from(..)` 静态函数

在JavaScript中一个“类数组对象”是一个拥有 `length` 属性的对象，这个属性明确地带有0或更高的整数值。

在JS中处理这些值出了名地让人沮丧；将它们变形为真正的数组曾经是十分常见的做法，这样各种 `Array.property` 方法（ `map(..)` ， `indexOf(..)` 等等）才能与它一起使用。这种处理通常看起来像：

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

```
7.
8. var arr = Array.prototype.slice.call( arrLike );
```

另一种 `slice(..)` 经常被使用的常见任务是，复制一个真正的数组：

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

在这两种情况下，新的ES6 `Array.from(..)` 方法是一种更易懂而且更优雅的方式 —— 也不那么冗长：

```
1. var arr = Array.from( arrLike );
2.
3. var arrCopy = Array.from( arr );
```

`Array.from(..)` 会查看第一个参数值是否是一个可迭代对象（参见第三章的“迭代器”），如果是，它就使用迭代器来产生值，并将这些值“拷贝”到将要被返回的数组中。因为真正的数组拥有一个可以产生这些值的迭代器，所以这个迭代器会被自动地使用。

但是如果你传递一个类数组对象作为 `Array.from(..)` 的第一个参数值，它的行为基本上是和 `slice()` （不带参数值的！）或 `apply()` 相同的，它简单地循环所有的值，访问从 `0` 开始到 `length` 值的由数字命名的属性。

考虑如下代码：

```
1. var arrLike = {
2.     length: 4,
3.     2: "foo"
4. };
5.
6. Array.from( arrLike );
7. // [ undefined, undefined, "foo", undefined ]
```

因为在 `arrLike` 上不存在位置 `0` ， `1` ，和 `3` ，所以对这些值槽中的每一个，结果都是 `undefined` 值。

你也可以这样产生类似的结果：

```
1. var emptySlotsArr = [];
2. emptySlotsArr.length = 4;
3. emptySlotsArr[2] = "foo";
4.
5. Array.from( emptySlotsArr );
6. // [ undefined, undefined, "foo", undefined ]
```

避免空值槽

前面的代码段中，在 `emptySlotsArr` 和 `Array.from(..)` 调用的结果有一个微妙但重要的不同。 `Array.from(..)` 从不产生空值槽。

在ES6之前，如果你想要制造一个被初始化为在每个值槽中使用实际 `undefined` 值（不是空值槽！）的特定长数组，你不得不做一些额外的工作：

```
var a = Array( 4 );                             // 四个空值槽！

var b = Array.apply( null, { length: 4 } );     // 四个 `undefined` 值
```

但现在 `Array.from(..)` 使这件事简单了些：

```
var c = Array.from( { length: 4 } );            // 四个 `undefined` 值
```

警告： 使用一个像前面代码段中的 `a` 那样的空值槽数组可以与一些数组函数工作，但是另一些函数会忽略空值槽（比如 `map(..)` 等）。你永远不应该刻意地使用空值槽，因为它几乎肯定会在你的程序中导致奇怪/不可预料的行为。

映射

`Array.from(..)` 工具还有另外一个绝技。第二个参数值，如果被提供的话，是一个映射函数（和普通的 `Array#map(..)` 几乎相同），它在将每个源值映射/变形为返回的目标值时调用。考虑如下代码：

```
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 ]
```

注意： 就像其他接收回调的数组方法一样， `Array.from(..)` 接收可选的第三个参数值，它将被指定为作为第二个参数传递的回调的 `this` 绑定。否则， `this` 将是 `undefined` 。

一个使用 `Array.from(..)` 将一个8位值数组翻译为16位值数组的例子，参见第五章的“类型化数组”。

创建 Arrays 和子类型

在前面几节中，我们讨论了 `Array.of(..)` 和 `Array.from(..)` ，它们都用与构造器相似的方法创建一个新数组。但是在子类中它们会怎么做？它们是创建基本 `Array` 的实例，还是创建衍生的子类的实例？

```
class MyCoolArray extends Array {
    ..
}

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

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

`of(..)` 和 `from(..)` 都使用它们被访问时的构造器来构建数组。所以如果你使用基本的 `Array.of(..)` 你将得到 `Array` 实例，但如果你使用 `MyCoolArray.of(..)` ，你将得到一个 `MyCoolArray` 实例。

在第三章的“类”中，我们讲解了在所有内建类（比如 `Array` ）中定义好的 `@@species` 设定，它被用于任何创建新实例的原型方法。 `slice(..)` 是一个很棒的例子：

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

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

一般来说，这种默认行为将可能是你想要的，但是正如我们在第三章中讨论过的，如果你想的话你 可以 覆盖它：

```
class MyCoolArray extends Array {
    // 强制 `species` 为父类构造器
    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
```

要注意的是， `@@species` 设定仅适用于原型方法，比如 `slice(..)` 。 `of(..)` 和 `from(..)` 不使用它；它们俩都只使用 `this` 绑定（哪个构造器被用于发起引用）。考虑如下代码：

```
class MyCoolArray extends Array {
```

```
    // 强制 `species` 为父类构造器
    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
```

`copyWithin(..)` 原型方法

`Array#copyWithin(..)` 是一个对所有数组可用的新修改器方法（包括类型化数组；参加第五章）。 `copyWithin(..)` 将数组的一部分拷贝到同一个数组的其他位置，覆盖之前存在在那里的任何东西。

它的参数值是 目标（要被拷贝到的索引位置），开始（拷贝开始的索引位置（含）），和可选的 结束（拷贝结束的索引位置（不含））。如果这些参数值中存在任何负数，那么它们就被认为是相对于数组的末尾。

考虑如下代码：

```
[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]
```

`copyWithin(..)` 方法不会扩张数组的长度，就像前面代码段中的第一个例子展示的。当到达数组的末尾时拷贝就会停止。

与你可能想象的不同，拷贝的顺序并不总是从左到右的。如果起始位置与目标为重叠的话，它有可能造成已经被拷贝过的值被重复拷贝，这大概不是你期望的行为。

所以在这种情况下，算法内部通过相反的拷贝顺序来避免这个坑。考虑如下代码：

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

如果算法是严格的从左到右，那么 `2` 应当被拷贝来覆盖 `3` ，然后这个被拷贝的 `2` 应当被拷贝来覆盖 `4` ，然后这个被拷贝的 `2` 应当被拷贝来覆盖 `5` ，而你最终会得到 `[1,2,2,2,2]` 。

与此不同的是，拷贝算法把方向反转过来，拷贝 `4` 来覆盖 `5` ，然后拷贝 `3` 来覆盖 `4` ，然后拷贝 `2` 来覆盖 `3` ，而最后的结果是 `[1,2,2,3,4]` 。就期待的结果而言这可能更“正确”，但是如

果你仅以单纯的从左到右的方式考虑拷贝算法的话，它就可能让人糊涂。

`fill(..)` 原型方法

ES6中的 `Array#fill(..)` 方法原生地支持使用一个指定的值来完全地（或部分地）填充一个既存的数组：

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

`fill(..)` 可选地接收 开始 与 结束 参数，它们指示要被填充的数组的一部分，比如：

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

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

`find(..)` 原型方法

一般来说，在一个数组中搜索一个值的最常见方法曾经是 `indexOf(..)` 方法，如果值被找到的话它返回值的位置索引，没有找到的话返回 `-1` ：

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

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

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

`indexOf(..)` 比较要求一个严格 `===` 匹配，所以搜索 `"2"` 找不到值 `2` ，反之亦然。没有办法覆盖 `indexOf(..)` 的匹配算法。不得不手动与值 `-1` 进行比较也很不幸/不优雅。

提示： 一个使用 `~` 操作符来绕过难看的 `-1` 的有趣（而且争议性地令人糊涂）技术，参见本系列的 类型与文法。

从ES5开始，控制匹配逻辑的最常见的迂回方法是 `some(..)` 。它的工作方式是为每一个元素调用一个回调函数，直到这些调用中的一个返回 `true` /truthy值，然后它就会停止。因为是由你来定义这个回调函数，所以你就拥有了如何做出匹配的完全控制权：

```
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
```

但这种方式的缺陷是你只能使用 `true` / `false` 来指示是否找到了合适的匹配值，而不是实际被匹配的值。

ES6的 `find(..)` 解决了这个问题。它的工作方式基本上与 `some(..)` 相同，除了一旦回调返回一个 `true` /truthy值，实际的数组值就会被返回：

```
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
});
```

使用一个自定义的 `matcher(..)` 函数还允许你与对象这样的复杂值进行匹配：

```
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 }
```

注意： 和其他接收回调的数组方法一样， `find(..)` 接收一个可选的第二参数。如果它被设置了的话，就将被指定为作为第一个参数传递的回调的 `this` 绑定。否则， `this` 将是 `undefined` 。

`findIndex(..)` 原型方法

虽然前一节展示了 `some(..)` 如何在一个数组检索给出一个Boolean结果，和 `find(..)` 如何从数组

检索中给出匹配的值，但是还有一种需求是寻找匹配的值的位置索引。

`indexOf(..)` 可以完成这个任务，但是没有办法控制它的匹配逻辑；它总是使用 `===` 严格等价。所以ES6的 `findIndex(..)` 才是答案：

```
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` （在 `indexOf(..)` 中经常这么干）来从检索中取得一个boolean，因为 `some(..)` 已经给出了你想要的 `true` / `false` 了。而且也不要用 `a[ a.findIndex(..) ]` 来取得一个匹配的值，因为这是 `find(..)` 完成的任务。最后，如果你需要严格匹配的索引，就使用 `indexOf(..)` ，如果你需要一个更加定制化的匹配，就使用 `findIndex(..)` 。

注意： 和其他接收回调的数组方法一样， `findIndex(..)` 接收一个可选的第二参数。如果它被设置了的话，就将被指定为作为第一个参数传递的回调的 `this` 绑定。否则， `this` 将是 `undefined` 。

`entries()` , `values()` , `keys()` 原型方法

在第三章中，我们展示了数据结构如何通过一个迭代器来提供一种模拟逐个值的迭代。然后我们在第五章探索新的ES6集合（Map，Set，等）如何为了产生不同种类的迭代器而提供几种方法时阐述了这种方式。

因为 `Array` 并不是ES6的新东西，所以它可能不被认为是一个传统意义上的“集合”，但是在它提供了相同的迭代器方法： `entries()` ， `values()` ，和 `keys()` 的意义上，它是的。考虑如下代码：

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

```
2.
3. [...a.values()];                  // [1,2,3]
4. [...a.keys()];                      // [0,1,2]
5. [...a.entries()];                  // [ [0,1], [1,2], [2,3] ]
6.
7. [...a[Symbol.iterator]()];          // [1,2,3]
```

就像 `Set` 一样，默认的 `Array` 迭代器与 `values()` 放回的东西相同。

在本章早先的“避免空值槽”一节中，我们展示了 `Array.from(..)` 如何将一个数组中的空值槽看作带有 `undefined` 的存在值槽。其实际的原因是，在底层数组迭代器就是以这种方式动作的：

```
1. var a = [];
2. a.length = 3;
3. a[1] = 2;
4.
5. [...a.values()];        // [undefined,2,undefined]
6. [...a.keys()];            // [0,1,2]
7. [...a.entries()];        // [ [0,undefined], [1,2], [2,undefined] ]
```

Object

- `Object`
  - `Object.is(..)` 静态函数
  - `Object.getOwnPropertySymbols(..)` 静态函数
  - `Object.setPrototypeOf(..)` 静态函数
  - `Object.assign(..)` 静态函数

`Object`

几个额外的静态帮助方法已经被加入 `Object` 。从传统意义上讲，这种种类的函数是关注于对象值的行为/能力的。

但是，从ES6开始， `Object` 静态函数还用于任意种类的通用全局API —— 那些还没有更自然地存在于其他的某些位置的API（例如， `Array.from(..)` ）。

`Object.is(..)` 静态函数

`Object.is(..)` 静态函数进行值的比较，它的风格甚至要比 `===` 比较还要严格。

`Object(..)` 调用底层的 `SameValue` 算法（ES6语言规范，第7.2.9节）。 `SameValue` 算法基本上与 `===` 严格等价比较算法相同（ES6语言规范，第7.2.13节），但是带有两个重要的例外。

考虑如下代码：

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

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

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

你应当为严格等价性比较继续使用 `===` ； `Object.is(..)` 不应当被认为是这个操作符的替代品。但是，在你想要严格地识别 `NaN` 或 `-0` 值的情况下， `Object.is(..)` 是现在的首选方式。

注意： ES6还增加了一个 `Number.isNaN(..)` 工具（在本章稍后讨论），它可能是一个稍稍方便一些的测试；比起 `Object.is(x, NaN)` 你可能更偏好 `Number.isNaN(x)` 。你 可以 使用笨拙的 `x == 0 && 1 / x === -Infinity` 来准确地测试 `-0` ，但在这种情况下 `Object.is(x,-0)` 要好得多。

`Object.getOwnPropertySymbols(..)` 静态函数

第二章中的“Symbol”一节讨论了ES6中的新Symbol基本值类型。

Symbol可能将是在对象上最经常被使用的特殊（元）属性。所以引入了 `Object.getOwnPropertySymbols(..)` ，它仅取回直接存在于对象上的symbol属性：

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

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

`Object.setPrototypeOf(..)` 静态函数

还是在第二章中，我们提到了 `Object.setPrototypeOf(..)` 工具，它为了 行为委托 的目的（意料之中地）设置一个对象的 `[[Prototype]]` （参见本系列的 *this*与对象原型）。考虑如下代码：

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

Object.setPrototypeOf( o2, o1 );

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

另一种方式：

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

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

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

在前面两个代码段中， `o2` 和 `o1` 之间的关系都出现在 `o2` 定义的末尾。更常见的是， `o2` 和 `o1` 之间的关系在 `o2` 定义的上面被指定，就像在类中，而且在对象字面量的 `__proto__` 中也是这样（参见第二章的“设置 `[[Prototype]]` ”）。

警告： 正如展示的那样，在对象创建之后立即设置 `[[Prototype]]` 是合理的。但是在很久之后才改变它一般不是一个好主意，而且经常会导致困惑而非清晰。

`Object.assign(..)` 静态函数

许多JavaScript库/框架都提供将一个对象的属性拷贝/混合到另一个对象中的工具（例如，jQuery的 `extend(..)` ）。在这些不同的工具中存在着各种微妙的区别，比如一个拥有 `undefined` 值的属性是否被忽略。

ES6增加了 `Object.assign(..)` ，它是这些算法的一个简化版本。第一个参数是 目标对象 而所有其他的参数是 源对象，它们会按照罗列的顺序被处理。对每一个源对象，它自己的（也就是，不是“继承的”）可枚举键，包括symbol，将会好像通过普通 `=` 赋值那样拷贝。 `Object.assign(..)` 返回目标对象。

考虑这种对象构成：

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

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

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

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

// 设置不可枚举 symbol
Object.defineProperty( o3, Symbol( "h" ), {
    value: 8,
    enumerable: false
} );

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

仅有属性 `a` ， `b` ， `c` ， `e` ，和 `Symbol("g")` 将被拷贝到 `target` ：

```
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")]
```

属性 `d` ， `f` ，和 `Symbol("h")` 在拷贝中被忽略了；非枚举属性和非自身属性将会被排除在赋值之外。另外， `e` 作为一个普通属性赋值被拷贝，而不是作为一个只读属性被复制。

在早先一节中，我们展示了使用 `setPrototypeOf(..)` 来在对象 `o2` 和 `o1` 之间建立一个 `[[Prototype]]` 关系。这是利用 `Object.assign(..)` 的另外一种形式：

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

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

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

注意： `Object.create(..)` 是一个ES5标准工具，它创建一个 `[[Prototype]]` 链接好的空对象。更多信息参见本系列的 *this*与对象原型。

Math

`Math`

ES6增加了几种新的数学工具，它们协助或填补了常见操作的空白。所有这些操作都可以被手动计算，但是它们中的大多数现在都被原生地定义，这样JS引擎就可以优化计算的性能，或者进行与手动计算比起来小数精度更高的计算。

与直接的开发者相比，asm.js/转译的JS代码（参见本系列的 异步与性能）更可能是这些工具的使用者。

三角函数：

- `cosh(..)` - 双曲余弦
- `acosh(..)` - 双曲反余弦
- `sinh(..)` - 双曲正弦
- `asinh(..)` - 双曲反正弦
- `tanh(..)` - 双曲正切
- `atanh(..)` - 双曲反正切
- `hypot(..)` - 平方和的平方根（也就是，广义勾股定理）

算数函数：

- `cbrt(..)` - 立方根
- `clz32(..)` - 计数32位二进制表达中前缀的零
- `expm1(..)` - 与`exp(x) - 1`相同
- `log2(..)` - 二进制对数（以2为底的对数）
- `log10(..)` - 以10为底的对数
- `log1p(..)` - 与`log(x + 1)`相同
- `imul(..)` - 两个数字的32为整数乘法

元函数：

- `sign(..)` - 返回数字的符号
- `trunc(..)` - 仅返回一个数字的整数部分
- `fround(..)` - 舍入到最接近的32位（单精度）浮点数值

Number

- `Number`
  - 静态属性
  - `Number.isNaN(..)` 静态函数
  - `Number.isFinite(..)` 静态函数
  - 整数相关的静态函数

`Number`

重要的是，为了你的程序能够正常工作，它必须准确地处理数字。ES6增加了一些额外的属性和函数来辅助常见的数字操作。

两个在 `Number` 上新增的功能只是既存全局函数的引用： `Number.parseInt(..)` 和 `Number.parseFloat(..)` 。

静态属性

ES6以静态属性的形式增加了一些有用的数字常数：

- `Number.EPSILON` - 在任意两个数字之间的最小值： `2^-52` （关于为了应对浮点算数运算不精确的问题而将这个值用做容差的讲解，参见本系列的 类型与文法 的第二章）
- `Number.MAX_SAFE_INTEGER` - 可以用一个JS数字值明确且“安全地”表示的最大整数： `2^53 - 1`
- `Number.MIN_SAFE_INTEGER` - 可以用一个JS数字值明确且“安全地”表示的最小整数： `-(2^53 - 1)` 或 `(-2)^53 + 1` .

注意： 关于“安全”整数的更多信息，参见本系列的 类型与文法 的第二章。

`Number.isNaN(..)` 静态函数

标准的全局 `isNaN(..)` 工具从一开始就坏掉了，因为不仅对实际的 `NaN` 值返回 `true` ，而且对不是数字的东西也返回 `true` 。其原因是它会将参数值强制转换为数字类型（这可能失败而导致一个 NaN ）。ES6增加了一个修复过的工具 `Number.isNaN(..)` ，它可以正确工作：

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

isNaN( a );                         // true
isNaN( b );                         // true —— 噢！
isNaN( c );                         // false

Number.isNaN( a );                  // true
Number.isNaN( b );                  // false —— 修好了！
Number.isNaN( c );                  // false
```

`Number.isFinite(..)` 静态函数

看到像 `isFinite(..)` 这样的函数名会诱使人们认为它单纯地意味着“不是无限”。但这不十分正确。这个新的ES6工具有更多的微妙之处。考虑如下代码：

```
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(..)` 会强制转换它收到的参数值，但是 `Number.isFinite(..)` 会省略强制转换的行为：

```
var a = "42";

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

你可能依然偏好强制转换，这时使用全局 `isFinite(..)` 是一个合法的选择。或者，并且可能是更明智的选择，你可以使用 `Number.isFinite(+x)` ，它在将 `x` 传递前明确地将它强制转换为数字（参见本系列的 类型与文法 的第四章）。

整数相关的静态函数

JavaScript数字值总是浮点数（IEEE-754）。所以判定一个数字是否是“整数”的概念与检查它的类型无关，因为JS没有这样的区分。

取而代之的是，你需要检查这个值是否拥有非零的小数部分。这样做的最简单的方法通常是：

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

ES6增加了一个 `Number.isInteger(..)` 帮助工具，它可以潜在地判定这种性质，而且效率稍微高一些：

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

注意： 在JavaScript中， `4` ， `4.` ， `4.0` ，或 `4.0000` 之间没有区别。它们都将被认为是一个“整数”，因此都会从 `Number.isInteger(..)` 中给出 `true` 。

另外， `Number.isInteger(..)` 过滤了一些明显的非整数值，它们在 `x === Math.floor(x)` 中可能会被混淆：

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

有时候处理“整数”是信息的重点，它可以简化特定的算法。由于为了仅留下整数而进行过滤，JS代码本身不会运行得更快，但是当仅有整数被使用时引擎可以采取几种优化技术（例如，asm.js）。

因为 `Number.isInteger(..)` 对 `Nan` 和 `Infinity` 值的处理，定义一个 `isFloat(..)` 工具并不像 `!Number.isInteger(..)` 一样简单。你需要这么做：

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

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

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

注意： 这看起来可能很奇怪，但是无穷即不应当被认为是整数也不应当被认为是浮点数。

ES6还定义了一个 `Number.isSafeInteger(..)` 工具，它检查一个值以确保它是一个整数并且在 `Number.MIN_SAFE_INTEGER` - `Number.MAX_SAFE_INTEGER` 的范围内（包含两端）。

```
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
```

String

- `String`
  - Unicode 函数
  - `String.raw(..)` 静态函数
  - `repeat(..)` 原型函数
  - 字符串检验函数

`String`

在ES6之前字符串就已经拥有好几种帮助函数了，但是有更多的内容被加入了进来。

Unicode 函数

在第二章的“Unicode敏感的字符串操作”中详细讨论了 `String.fromCodePoint(..)` ， `String#codePointAt(..)` ， `String#normalize(..)` 。它们被用来改进JS字符串值对Unicode的支持。

```
String.fromCodePoint( 0x1d49e );            // "?"

"ab?d".codePointAt( 2 ).toString( 16 );        // "1d49e"
```

`normalize(..)` 字符串原型方法用来进行Unicode规范化，它将字符与相邻的“组合标志”进行组合，或者将组合好的字符拆开。

一般来说，规范化不会对字符串的内容产生视觉上的影响，但是会改变字符串的内容，这可能会影响 `length` 属性报告的结果，以及用位置访问字符的行为：

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

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

`normalize(..)` 接受一个可选参数值，它用于指定使用的规范化形式。这个参数值必须是下面四个值中的一个： `"NFC"` （默认）， `"NFD"` ， `"NFKC"` ，或者 `"NFKD"` 。

注意： 规范化形式和它们在字符串上的效果超出了我们要在这里讨论的范围。更多细节参见“Unicode规范化形式”(http://www.unicode.org/reports/tr15/)。

`String.raw(..)` 静态函数

`String.raw(..)` 工具被作为一个内建的标签函数来与字符串字面模板（参见第二章）一起使用，取得不带有任何转译序列处理的未加工的字符串值。

这个函数几乎永远不会被手动调用，但是将与被标记的模板字面量一起使用：

```
var str = "bc";

String.raw`\ta${str}d\xE9`;
// "\tabcd\xE9", not "    abcdé"
```

在结果字符串中，`\` 和 `t` 是分离的未被加工过的字符，而不是一个转译字符序列 `\t` 。这对Unicode转译序列也是一样。

`repeat(..)` 原型函数

在Python和Ruby那样的语言中，你可以这样重复一个字符串：

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

在JS中这不能工作，因为 `*` 乘法是仅对数字定义的，因此 `"foo"` 会被强制转换为 `NaN` 数字。

但是，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"
```

字符串检验函数

作为对ES6以前的 `String#indexOf(..)` 和 `String#lastIndexOf(..)` 的补充，增加了三个新的搜索/检验函数： `startsWith(..)` ， `endsWith(..)` ，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
```

对于所有这些字符串搜索/检验方法，如果你查询一个空字符串 `""` ，那么它将要么在字符串的开头被找到，要么就在字符串的末尾被找到。

警告： 这些方法默认不接受正则表达式作为检索字符串。关于关闭实施在第一个参数值上的 `isRegExp` 检查的信息，参见第七章的“正则表达式Symbol”。

复习

复习

ES6在各种内建原生对象上增加了许多额外的API帮助函数：

- `Array`增加了`of(..)`和`from(..)`之类的静态函数，以及`copyWithin(..)`和`fill(..)`之类的原型函数。
- `Object`增加了`is(..)`和`assign(..)`之类的静态函数。
- `Math`增加了`acosh(..)`和`clz32(..)`之类的静态函数。
- `Number`增加了`Number.EPSILON`之类的静态属性，以及`Number.isFinite(..)`之类的静态函数。
- `String`增加了`String.fromCodePoint(..)`和`String.raw(..)`之类的静态函数，以及`repeat(..)`和`includes(..)`之类的原型函数。

这些新增函数中的绝大多数都可以被填补（参见ES6 Shim），它们都是受常见的JS库/框架中的工具启发的。

第七章：元编程

- 第七章：元编程

第七章：元编程

元编程是针对程序本身的行为进行操作的编程。换句话说，它是为你程序的编程而进行的编程。是的，很拗口，对吧？

例如，如果你为了调查对象 `a` 和另一个对象 `b` 之间的关系 —— 它们是被 `[[Prototype]]` 链接的吗？ —— 而使用 `a.isPrototypeOf(b)` ，这通常称为自省，就是一种形式的元编程。宏（JS中还没有） —— 代码在编译时修改自己 —— 是元编程的另一个明显的例子。使用 `for..in` 循环枚举一个对象的键，或者检查一个对象是否是一个“类构造器”的 实例，是另一些常见的元编程任务。

元编程关注以下的一点或几点：代码检视自己，代码修改自己，或者代码修改默认的语言行为而使其他代码受影响。

元编程的目标是利用语言自身的内在能力使你其他部分的代码更具描述性，表现力，和/或灵活性。由于元编程的 元 的性质，要给它一个更精确的定义有些困难。理解元编程的最佳方法是通过代码来观察它。

ES6在JS已经拥有的东西上，增加了几种新的元编程形式/特性。

- 函数名
- 元属性
- 通用Symbol
- 代理
- Reflect API
- 特性测试
- 尾部调用优化（TCO）
- 复习

函数名

- 函数名
  - 推断

函数名

有一些情况，你的代码想要检视自己并询问某个函数的名称是什么。如果你询问一个函数的名称，答案会有些令人诧异地模糊。考虑如下代码：

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

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

在这前一个代码段中，“ `obj.foo()` 的名字是什么？”有些微妙。是 `"foo"` ， `""` ，还是 `undefined` ？那么 `obj.bar()` 呢 —— 是 `"bar"` 还是 `"baz"` ？ `obj.bam()` 称为 `"bam"` 还是 `"daz"` ？ `obj.zim()` 呢？

另外，作为回调被传递的函数呢？就像：

```
function foo(cb) {
    // 这里的 `cb()` 的名字是什么？
}

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

在程序中函数可以被好几种方法所表达，而函数的“名字”应当是什么并不总是那么清晰和明确。

更重要的是，我们需要区别函数的“名字”是指它的 `name` 属性 —— 是的，函数有一个叫做 `name` 的属性 —— 还是指它词法绑定的名称，比如在 `function bar() { .. }` 中的 `bar` 。

词法绑定名称是你将在递归之类的东西中所使用的：

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

`name` 属性是你为了元编程而使用的，所以它才是我们在这里的讨论中所关注的。

产生这种用困惑是因为，在默认情况下一个函数的词法名称（如果有的话）也会被设置为它的 `name` 属性。实际上，ES5（和以前的）语言规范中并没有官方要求这种行为。 `name` 属性的设置是一种非标准，但依然相当可靠的行为。在ES6中，它已经被标准化。

提示： 如果一个函数的 `name` 被赋值，它通常是在开发者工具的栈轨迹中使用的名称。

推断

但如果函数没有词法名称， `name` 属性会怎么样呢？

现在在ES6中，有一个推断规则可以判定一个合理的 `name` 属性值来赋予一个函数，即使它没有词法名称可用。

考虑如下代码：

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

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

如果我们给了这个函数一个词法名称，比如 `abc = function def() { .. }` ，那么 `name` 属性将理所当然地是 `"def"` 。但是由于缺少词法名称，直观上名称 `"abc"` 看起来很合适。

这里是在ES6中将会（或不会）进行名称推断的其他形式：

```
(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
```

`name` 属性默认是不可写的，但它是可配置的，这意味着如果有需要，你可以使用 `Object.defineProperty(..)` 来手动改变它。

元属性

元属性

在第三章的“ `new.target` ”一节中，我们引入了一个ES6的新概念：元属性。正如这个名称所暗示的，元属性意在以一种属性访问的形式提供特殊的元信息，而这在以前是不可能的。

在 `new.target` 的情况下，关键字 `new` 作为一个属性访问的上下文环境。显然 `new` 本身不是一个对象，这使得这种能力很特殊。然而，当 `new.target` 被用于一个构造器调用（一个使用 `new` 调用的函数/方法）内部时， `new` 变成了一个虚拟上下文环境，如此 `new.target` 就可以指代这个 `new` 调用的目标构造器。

这是一个元编程操作的典型例子，因为它的意图是从一个构造器调用内部判定原来的 `new` 的目标是什么，这一般是为了自省（检查类型/结构）或者静态属性访问。

举例来说，你可能想根据一个构造器是被直接调用，还是通过一个子类进行调用，来使它有不同的行为：

```
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
```

这里有一个微妙的地方，在 `Parent` 类定义内部的 `constructor()` 实际上被给予了这个类的词法名称（ `Parent` ），即便语法暗示着这个类是一个与构造器分离的不同实体。

警告： 与所有的元编程技术一样，要小心不要创建太过聪明的代码，而使未来的你或其他维护你代码的人很难理解。小心使用这些技巧。

通用Symbol

- 通用 Symbol
  - `Symbol.iterator`
  - `Symbol.toStringTag` 和 `Symbol.hasInstance`
  - `Symbol.species`
  - `Symbol.toPrimitive`
  - 正则表达式 Symbols
  - `Symbol.isConcatSpreadable`
  - `Symbol.unscopables`

通用 Symbol

在第二章中的“Symbol”一节中，我们讲解了新的ES6基本类型 `symbol` 。除了你可以在你自己的程序中定义的symbol以外，JS预定义了几种内建symbol，被称为 通用（*Well Known*）*Symbols*（WKS）。

定义这些symbol值主要是为了向你的JS程序暴露特殊的元属性来给你更多JS行为的控制权。

我们将简要介绍每一个symbol并讨论它们的目的。

`Symbol.iterator`

在第二和第三章中，我们介绍并使用了 `@@iterator` symbol，它被自动地用于 `...` 扩散和 `for..of` 循环。我们还在第五章中看到了在新的ES6集合中定义的 `@@iterator` 。

`Symbol.iterator` 表示在任意一个对象上的特殊位置（属性），语言机制自动地在这里寻找一个方法，这个方法将构建一个用于消费对象值的迭代器对象。许多对象都带有一个默认的 `Symbol.iterator` 。

然而，我们可以通过设置 `Symbol.iterator` 属性来为任意对象定义我们自己的迭代器逻辑，即便它是覆盖默认迭代器的。这里的元编程观点是，我们在定义JS的其他部分（明确地说，是操作符和循环结构）在处理我们所定义的对象值时所使用的行为。

考虑如下代码：

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

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

```

```
// 定义一个仅在奇数索引处产生值的迭代器
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
```

`Symbol.toStringTag` 和 `Symbol.hasInstance`

最常见的元编程任务之一，就是在一个值上进行自省来找出它是什么 种类 的，者经常用来决定它们上面适于实施什么操作。对于对象，最常见的两个自省技术是 `toString()` 和 `instanceof` 。

考虑如下代码：

```
function Foo() {}

var a = new Foo();

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

在ES6中，你可以控制这些操作的行为：

```
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
```

在原型（或实例本身）上的 `@@toStringTag` symbol指定一个用于 `[object ___]` 字符串化的字符串值。

`@@hasInstance` symbol是一个在构造器函数上的方法，它接收一个实例对象值并让你通过放回 `true` 或 `false` 来决定这个值是否应当被认为是一个实例。

注意： 要在一个函数上设置 `@@hasInstance` ，你必须使用 `Object.defineProperty(..)` ，因为在 `Function.prototype` 上默认的那一个是 `writable: false` 。更多信息参见本系列的 *this*与对象原型。

`Symbol.species`

在第三章的“类”中，我们介绍了 `@@species` symbol，它控制一个类内建的生成新实例的方法使用哪一个构造器。

最常见的例子是，在子类化 `Array` 并且想要定义 `slice(..)` 之类被继承的方法应当使用哪一个构造器时。默认地，在一个 `Array` 的子类实例上调用的 `slice(..)` 将产生这个子类的实例，坦白地说这正是你经常希望的。

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 {
    // 将 `@@species` 倒推至被衍生的构造器
    static get [Symbol.species]() { return this; }

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

class Fun extends Cool {}

class Awesome extends Cool {
    // 将 `@@species` 强制为父类构造器
    static get [Symbol.species]() { return Cool; }
}

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

```
21.
22. c instanceof Fun;            // true
23. d instanceof Awesome;        // false
24. d instanceof Cool;            // true
```

就像在前面的代码段中的 `Cool` 的定义展示的那样，在内建的原生构造器上的 `Symbol.species` 设定默认为 `return this` 。它在用户自己的类上没有默认值，但也像展示的那样，这种行为很容易模拟。

如果你需要定义生成新实例的方法，使用 `new this.constructor[Symbol.species](..)` 的元编程模式，而不要用手写的 `new this.constructor(..)` 或者 `new XYZ(..)` 。如此衍生的类就能够自定义 `Symbol.species` 来控制哪一个构造器来制造这些实例。

`Symbol.toPrimitive`

在本系列的 类型与文法 一书中，我们讨论了 `ToPrimitive` 抽象强制转换操作，它在对象为了某些操作（例如 `==` 比较或者 `+` 加法）而必须被强制转换为一个基本类型值时被使用。在ES6以前，没有办法控制这个行为。

在ES6中，在任意对象值上作为属性的 `@@toPrimitive` symbol都可以通过指定一个方法来自定义这个 `ToPrimitive` 强制转换。

考虑如下代码：

```
1. var arr = [1,2,3,4,5];
2.
3. arr + 10;                // 1,2,3,4,510
4.
5. arr[Symbol.toPrimitive] = function(hint) {
6.     if (hint == "default" || hint == "number") {
7.         // 所有数字的和
8.         return this.reduce( function(acc,curr){
9.             return acc + curr;
10.         }, 0 );
11.     }
12. };
13.
14. arr + 10;                // 25
```

`Symbol.toPrimitive` 方法将根据调用 `ToPrimitive` 的操作期望何种类型，而被提供一个值为 `"string"` ， `"number"` ，或 `"default"` （这应当被解释为 `"number"` ）的 提示（*hint*）。在前一个代码段中， `+` 加法操作没有提示（ `"default"` 将被传递）。一个 `*` 乘法操作将提示 `"number"` ，而一个 `String(arr)` 将提示 `"string"` 。

警告： `==` 操作符将在一个对象上不使用任何提来示调用 `ToPrimitive` 操作 —— 如果存在 `@@toPrimitive` 方法的话，将使用 `"default"` 被调用 —— 如果另一个被比较的值不是一个对象。

但是，如果两个被比较的值都是对象， `==` 的行为与 `===` 是完全相同的，也就是引用本身将被直接比较。这种情况下， `@@toPrimitive` 根本不会被调用。关于强制转换和抽象操作的更多信息，参见本系列的 类型与文法。

正则表达式 Symbols

对于正则表达式对象，有四种通用 symbols 可以被覆盖，它们控制着这些正则表达式在四个相应的同名 `String.prototype` 函数中如何被使用：

- `@@match` ：一个正则表达式的 `Symbol.match` 值是使用被给定的正则表达式来匹配一个字符串值的全部或部分的方法。如果你为 `String.prototype.match(..)` 传递一个正则表达式做范例匹配，它就会被使用。

  匹配的默认算法写在ES6语言规范的第21.2.5.6部分(@match)。你可以覆盖这个默认算法并提供额外的正则表达式特性，比如后顾断言。"">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表达式特性，比如后顾断言。

  `Symbol.match` 还被用于 `isRegExp` 抽象操作（参见第六章的“字符串检测函数”中的注意部分）来判定一个对象是否意在被用作正则表达式。为了使一个这样的对象不被看作是正则表达式，可以将 `Symbol.match` 的值设置为 `false` （或falsy的东西）强制这个检查失败。

- `@@replace` ：一个正则表达式的 `Symbol.replace` 值是被 `String.prototype.replace(..)` 使用的方法，来替换一个字符串里面出现的一个或所有字符序列，这些字符序列匹配给出的正则表达式范例。

  替换的默认算法写在ES6语言规范的第21.2.5.8部分(@replace)。"">https://people.mozilla.org/~jorendorff/es6-draft.html#sec-regexp.prototype-@@replace)。

  一个覆盖默认算法的很酷的用法是提供额外的 `replacer` 可选参数值，比如通过用连续的替换值消费可迭代对象来支持 `"abaca".replace(/a/g,[1,2,3])` 产生 `"1b2c3"` 。

- `@@search` ：一个正则表达式的 `Symbol.search` 值是被 `String.prototype.search(..)` 使用的方法，来在一个字符串中检索一个匹配给定正则表达式的子字符串。

  检索的默认算法写在ES6语言规范的第21.2.5.9部分(@search)。"">https://people.mozilla.org/~jorendorff/es6-draft.html#sec-regexp.prototype-@@search)。

- `@@split` ：一个正则表达式的 `Symbol.split` 值是被 `String.prototype.split(..)` 使用的方法，来将一个字符串在分隔符匹配给定正则表达式的位置分割为子字符串。

  分割的默认算法写在ES6语言规范的第21.2.5.11部分

(@split)。"">https://people.mozilla.org/~jorendorff/es6-draft.html#sec-regexp.prototype-@@split)。

覆盖内建的正则表达式算法不是为心脏脆弱的人准备的！JS带有高度优化的正则表达式引擎，所以你自己的用户代码将很可能慢得多。这种类型的元编程很精巧和强大，但是应当仅用于确实必要或有好处的情况下。

`Symbol.isConcatSpreadable`

`@@isConcatSpreadable` symbol可以作为一个布尔属性（ `Symbol.isConcatSpreadable` ）在任意对象上（比如一个数组或其他的可迭代对象）定义，来指示当它被传递给一个数组 `concat(..)` 时是否应当被 扩散。

考虑如下代码：

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

b[Symbol.isConcatSpreadable] = false;

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

`Symbol.unscopables`

`@@unscopables` symbol可以作为一个对象属性（ `Symbol.unscopables` ）在任意对象上定义，来指示在一个 `with` 语句中哪一个属性可以和不可以作为此法变量被暴露。

考虑如下代码：

```
var o = { a:1, b:2, c:3 },
    a = 10, b = 20, c = 30;

o[Symbol.unscopables] = {
    a: false,
    b: true,
    c: false
};

with (o) {
    console.log( a, b, c );        // 1 20 3
}
```

一个在 `@@unscopables` 对象中的 `true` 指示这个属性应当是 非作用域（*unscopable*） 的，因此会从此法作用域变量中被过滤掉。 `false` 意味着它可以被包含在此法作用域变量中。

警告： `with` 语句在 `strict` 模式下是完全禁用的，而且因此应当被认为是在语言中被废弃的。不

要使用它。更多信息参见本系列的 作用域与闭包。因为应当避免 `with` ，所以这个 `@@unscopables` symbol也是无意义的。

代理

- 代理
  - 代理的限制
  - 可撤销的代理
  - 使用代理
    - 代理前置，代理后置
    - “No Such Property/Method”
    - 代理黑入 `[[Prototype]]` 链

代理

在ES6中被加入的最明显的元编程特性之一就是 `proxy` 特性。

一个代理是一种由你创建的特殊的对象，它“包”着另一个普通的对象 —— 或者说挡在这个普通对象的前面。你可以在代理对象上注册特殊的处理器（也叫 机关（*traps*）），当对这个代理实施各种操作时被调用。这些处理器除了将操作 传送 到原本的目标/被包装的对象上之外，还有机会运行额外的逻辑。

一个这样的 机关 处理器的例子是，你可以在一个代理上定义一个拦截 `[[Get]]` 操作的 `get` —— 它在当你试图访问一个对象上的属性时运行。考虑如下代码：

```
var obj = { a: 1 },
    handlers = {
        get(target,key,context) {
            // 注意：target === obj,
            // context === pobj
            console.log( "accessing: ", key );
            return Reflect.get(
                target, key, context
            );
        }
    },
    pobj = new Proxy( obj, handlers );

obj.a;
// 1

pobj.a;
// accessing: a
// 1
```

我们将一个 `get(..)` 处理器作为 处理器 对象的命名方法声明（ `Proxy(..)` 的第二个参数值），它

接收一个指向 目标 对象的引用（ `obj` ），属性的 键 名称（ `"a"` ），和 `self` /接受者/代理本身（ `pobj` ）。

在追踪语句 `console.log(..)` 之后，我们通过 `Reflect.get(..)` 将操作“转送”到 `obj` 。我们将在下一节详细讲解 `Reflect` API，但要注意的是每个可用的代理机关都有一个相应的同名 `Reflect` 函数。

这些映射是故意对称的。每个代理处理器在各自的元编程任务实施时进行拦截，而每个 `Reflect` 工具将各自的元编程任务在一个对象上实施。每个代理处理器都有一个自动调用相应 `Reflect` 工具的默认定义。几乎可以肯定你将总是一前一后地使用 `Proxy` 和 `Reflect` 。

这里的列表是你可以在一个代理上为一个 目标 对象/函数定义的处理器，以及它们如何/何时被触发：

- `get(..)` ：通过 `[[Get]]` ，在代理上访问一个属性（ `Reflect.get(..)` ， `.` 属性操作符或 `[ .. ]` 属性操作符）
- `set(..)` ：通过 `[[Set]]` ，在代理对象上设置一个属性（ `Reflect.set(..)` ， `=` 赋值操作符，或者解构赋值 —— 如果目标是一个对象属性的话）
- `deleteProperty(..)` ：通过 `[[Delete]]` ，在代理对象上删除一个属性（ `Reflect.deleteProperty(..)` 或 `delete` ）
- `apply(..)` （如果 目标 是一个函数）：通过 `[[Call]]` ，代理作为一个普通函数/方法被调用（ `Reflect.apply(..)` ， `call(..)` ， `apply(..)` ，或者 `(..)` 调用操作符）
- `construct(..)` （如果 目标 是一个构造函数）：通过 `[[Construct]]` 代理作为一个构造器函数被调用（ `Reflect.construct(..)` 或 `new` ）
- `getOwnPropertyDescriptor(..)` ：通过 `[[GetOwnProperty]]` ，从代理取得一个属性的描述符（ `Object.getOwnPropertyDescriptor(..)` 或 `Reflect.getOwnPropertyDescriptor(..)` ）
- `defineProperty(..)` ：通过 `[[DefineOwnProperty]]` ，在代理上设置一个属性描述符（ `Object.defineProperty(..)` 或 `Reflect.defineProperty(..)` ）
- `getPrototypeOf(..)` ：通过 `[[GetPrototypeOf]]` ，取得代理的 `[[Prototype]]` （ `Object.getPrototypeOf(..)` ， `Reflect.getPrototypeOf(..)` ， `__proto__` ， `Object#isPrototypeOf(..)` ，或 `instanceof` ）
- `setPrototypeOf(..)` ：通过 `[[SetPrototypeOf]]` ，设置代理的 `[[Prototype]]` （ `Object.setPrototypeOf(..)` ， `Reflect.setPrototypeOf(..)` ，或 `__proto__` ）
- `preventExtensions(..)` ：通过 `[[PreventExtensions]]` 使代理成为不可扩展的（ `Object.preventExtensions(..)` 或 `Reflect.preventExtensions(..)` ）
- `isExtensible(..)` ：通过 `[[IsExtensible]]` ，检测代理的可扩展性（ `Object.isExtensible(..)` 或 `Reflect.isExtensible(..)` ）
- `ownKeys(..)` ：通过 `[[OwnPropertyKeys]]` ，取得一组代理的直属属性和/或直属symbol属性（ `Object.keys(..)` ， `Object.getOwnPropertyNames(..)` ， `Object.getOwnSymbolProperties(..)` ， `Reflect.ownKeys(..)` ，或 `JSON.stringify(..)` ）
- `enumerate(..)` ：通过 `[[Enumerate]]` ，为代理的可枚举直属属性及“继承”属性请求一个迭代器（ `Reflect.enumerate(..)` 或 `for..in` ）
- `has(..)` ：通过 `[[HasProperty]]` ，检测代理是否拥有一个直属属性或“继承”属性

（`Reflect.has(..)`，`Object#hasOwnProperty(..)`，或`"prop" in obj`）

提示： 关于每个这些元编程任务的更多信息，参见本章稍后的“`Reflect` API”一节。

关于将会触发各种机关的动作，除了在前面列表中记载的以外，一些机关还会由另一个机关的默认动作间接地触发。举例来说：

```
var handlers = {
        getOwnPropertyDescriptor(target,prop) {
            console.log(
                "getOwnPropertyDescriptor"
            );
            return Object.getOwnPropertyDescriptor(
                target, prop
            );
        },
        defineProperty(target,prop,desc){
            console.log( "defineProperty" );
            return Object.defineProperty(
                target, prop, desc
            );
        }
    },
    proxy = new Proxy( {}, handlers );

proxy.a = 2;
// getOwnPropertyDescriptor
// defineProperty
```

在设置一个属性值时（不管是新添加还是更新），`getOwnPropertyDescriptor(..)`和`defineProperty(..)`处理器被默认的`set(..)`处理器触发。如果你还定义了你自己的`set(..)`处理器，你或许对`context`（不是`target`！）进行了将会触发这些代理机关的相应调用。

代理的限制

这些元编程处理器拦截了你可以对一个对象进行的范围很广泛的一组基础操作。但是，有一些操作不能（至少是还不能）被用于拦截。

例如，从`pobj`代理到`obj`目标，这些操作全都没有被拦截和转送：

```
var obj = { a:1, b:2 },
    handlers = { .. },
    pobj = new Proxy( obj, handlers );

typeof obj;
```

```
String( obj );
obj + "";
obj == pobj;
obj === pobj
```

也许在未来，更多这些语言中的底层基础操作都将是可拦截的，那将给我们更多力量来从JavaScript自身扩展它。

警告： 对于代理处理器的使用来说存在某些 不变量 —— 它们的行为不能被覆盖。例如， `isExtensible(..)` 处理器的结果总是被强制转换为一个 `boolean` 。这些不变量限制了一些你可以使用代理来自定义行为的能力，但是它们这样做只是为了防止你创建奇怪和不寻常（或不合逻辑）的行为。这些不变量的条件十分复杂，所以我们就不再这里全面阐述了，但是这篇博文(http://www.2ality.com/2014/12/es6-proxies.html#invariants)很好地讲解了它们。

可撤销的代理

一个一般的代理总是包装着目标对象，而且在创建之后就不能修改了 —— 只要保持着一个指向这个代理的引用，代理的机制就将维持下去。但是，可能会有一些情况你想要创建一个这样的代理：在你想要停止它作为代理时可以被停用。解决方案就是创建一个 可撤销代理：

```
var obj = { a: 1 },
    handlers = {
        get(target,key,context) {
            // 注意：target === obj,
            // context === pobj
            console.log( "accessing: ", key );
            return target[key];
        }
    },
    { proxy: pobj, revoke: prevoke } =
        Proxy.revocable( obj, handlers );

pobj.a;
// accessing: a
// 1

// 稍后：
prevoke();

pobj.a;
// TypeError
```

一个可撤销代理是由 `Proxy.revocable(..)` 创建的，它是一个普通的函数，不是一个像 `Proxy(..)` 那样的构造器。此外，它接收同样的两个参数值：目标 和 处理器。

与 `new Proxy(..)` 不同的是， `Proxy.revocable(..)` 的返回值不是代理本身。取而代之的是，它返回一个带有 *proxy* 和 *revoke* 两个属性的对象 —— 我们使用了对象解构（参见第二章的“解构”）来将这些属性分别赋值给变量 `pobj` 和 `prevoke` 。

一旦可撤销代理被撤销，任何访问它的企图（触发它的任何机关）都将抛出 `TypeError` 。

一个使用可撤销代理的例子可能是，将一个代理交给另一个存在于你应用中、并管理你模型中的数据的团体，而不是给它们一个指向正式模型对象本身的引用。如果你的模型对象改变了或者被替换掉了，你希望废除这个你交出去的代理，以便于其他的团体能够（通过错误！）知道要请求一个更新过的模型引用。

使用代理

这些代理处理器带来的元编程的好处应当是显而易见的。我们可以全面地拦截（而因此覆盖）对象的行为，这意味着我们可以用一些非常强大的方式将对象行为扩展至JS核心之外。我们将看几个模式的例子来探索这些可能性。

代理前置，代理后置

正如我们早先提到过的，你通常将一个代理考虑为一个目标对象的“包装”。在这种意义上，代理就变成了代码接口所针对的主要对象，而实际的目标对象则保持被隐藏/被保护的状态。

你可能这么做是因为你希望将对象传递到某个你不能完全“信任”的地方去，如此你需要在它的访问权上强制实施一些特殊的规则，而不是传递这个对象本身。

考虑如下代码：

```
var messages = [],
    handlers = {
        get(target,key) {
            // 是字符串值吗？
            if (typeof target[key] == "string") {
                // 过滤掉标点符号
                return target[key]
                    .replace( /[^\w]/g, "" );
            }

            // 让其余的东西通过
            return target[key];
        },
        set(target,key,val) {
            // 仅设置唯一的小写字符串
            if (typeof val == "string") {
                val = val.toLowerCase();
                if (target.indexOf( val ) == -1) {
```

```
                    target.push(val);
                }
            }
            return true;
        }
    },
    messages_proxy =
        new Proxy( messages, handlers );

// 在别处：
messages_proxy.push(
    "heLLo...", 42, "wOrlD!!", "WoRld!!"
);

messages_proxy.forEach( function(val){
    console.log(val);
} );
// hello world

messages.forEach( function(val){
    console.log(val);
} );
// hello... world!!
```

我称此为 代理前置 设计，因为我们首先（主要、完全地）与代理进行互动。

我们在与 `messages_proxy` 的互动上强制实施了一些特殊规则，这些规则不会强制实施在 `messages` 本身上。我们仅在值是一个不重复的字符串时才将它添加为元素；我们还将这个值变为小写。当从 `messages_proxy` 取得值时，我们过滤掉字符串中所有的标点符号。

另一种方式是，我们可以完全反转这个模式，让目标与代理交互而不是让代理与目标交互。这样，代码其实只与主对象交互。达成这种后备方案的最简单的方法是，让代理对象存在于主对象的 `[[Prototype]]` 链中。

考虑如下代码：

```
var handlers = {
        get(target,key,context) {
            return function() {
                context.speak(key + "!");
            };
        }
    },
    catchall = new Proxy( {}, handlers ),
    greeter = {
        speak(who = "someone") {
            console.log( "hello", who );
```

```
        }
    };

// 让 `catchall` 成为 `greeter` 的后备方法
Object.setPrototypeOf( greeter, catchall );

greeter.speak();                // hello someone
greeter.speak( "world" );        // hello world

greeter.everyone();                // hello everyone!
```

我们直接与 `greeter` 而非 `catchall` 进行交互。当我们调用 `speak(..)` 时，它在 `greeter` 上被找到并直接使用。但当我们试图访问 `everyone()` 这样的方法时，这个函数并不存在于 `greeter` 。

默认的对象属性行为是向上检查 `[[Prototype]]` 链（参见本系列的 *this*与对象原型），所以 `catchall` 被询问有没有一个 `everyone` 属性。然后代理的 `get()` 处理器被调用并返回一个函数，这个函数使用被访问的属性名（ `"everyone"` ）调用 `speak(..)` 。

我称这种模式为 代理后置，因为代理仅被用作最后一道防线。

“No Such Property/Method”

一个关于JS的常见的抱怨是，在你试着访问或设置一个对象上还不存在的属性时，默认情况下对象不是非常具有防御性。你可能希望为一个对象预定义所有这些属性/方法，而且在后续使用不存在的属性名时抛出一个错误。

我们可以使用一个代理来达成这种想法，既可以使用 代理前置 也可以 代理后置 设计。我们将两者都考虑一下。

```
var obj = {
        a: 1,
        foo() {
            console.log( "a:", this.a );
        }
    },
    handlers = {
        get(target,key,context) {
            if (Reflect.has( target, key )) {
                return Reflect.get(
                    target, key, context
                );
            }
            else {
                throw "No such property/method!";
            }
        },
```

```
        set(target,key,val,context) {
            if (Reflect.has( target, key )) {
                return Reflect.set(
                    target, key, val, context
                );
            }
            else {
                throw "No such property/method!";
            }
        }
    },
    pobj = new Proxy( obj, handlers );

pobj.a = 3;
pobj.foo();             // a: 3

pobj.b = 4;             // Error: No such property/method!
pobj.bar();             // Error: No such property/method!
```

对于 `get(..)` 和 `set(..)` 两者，我们仅在目标对象的属性已经存在时才转送操作；否则抛出错误。代理对象应当是进行交互的主对象，因为它拦截这些操作来提供保护。

现在，让我们考虑一下反过来的 代理后置 设计：

```
var handlers = {
        get() {
            throw "No such property/method!";
        },
        set() {
            throw "No such property/method!";
        }
    },
    pobj = new Proxy( {}, handlers ),
    obj = {
        a: 1,
        foo() {
            console.log( "a:", this.a );
        }
    };

// 让 `pobj` 称为 `obj` 的后备
Object.setPrototypeOf( obj, pobj );

obj.a = 3;
obj.foo();              // a: 3

obj.b = 4;              // Error: No such property/method!
```

```
obj.bar();          // Error: No such property/method!
```

在处理器如何定义的角度上，这里的 代理后置 设计相当简单。与拦截 `[[Get]]` 和 `[[Set]]` 操作并仅在目标属性存在时转送它们不同，我们依赖于这样一个事实：不管 `[[Get]]` 还是 `[[Set]]` 到达了我们的 `pobj` 后备对象，这个动作已经遍历了整个 `[[Prototype]]` 链并且没有找到匹配的属性。在这时我们可以自由地、无条件地抛出错误。很酷，对吧？

代理黑入 `[[Prototype]]` 链

`[[Get]]` 操作是 `[[Prototype]]` 机制被调用的主要渠道。当一个属性不能在直接对象上找到时， `[[Get]]` 会自动将操作交给 `[[Prototype]]` 对象。

这意味着你可以使用一个代理的 `get(..)` 机关来模拟或扩展这个 `[[Prototype]]` 机制的概念。

我们将考虑的第一种黑科技是创建两个通过 `[[Prototype]]` 循环链接的对象（或者说，至少看起来是这样！）。你不能实际创建一个真正循环的 `[[Prototype]]` 链，因为引擎将会抛出一个错误。但是代理可以假冒它！

考虑如下代码：

```
var handlers = {
        get(target,key,context) {
            if (Reflect.has( target, key )) {
                return Reflect.get(
                    target, key, context
                );
            }
            // 假冒循环的 `[[Prototype]]`
            else {
                return Reflect.get(
                    target[
                        Symbol.for( "[[Prototype]]" )
                    ],
                    key,
                    context
                );
            }
        }
    },
    obj1 = new Proxy(
        {
            name: "obj-1",
            foo() {
                console.log( "foo:", this.name );
            }
        },
```

```
        handlers
    ),
    obj2 = Object.assign(
        Object.create( obj1 ),
        {
            name: "obj-2",
            bar() {
                console.log( "bar:", this.name );
                this.foo();
            }
        }
    );

// 假冒循环的 `[[Prototype]]` 链
obj1[ Symbol.for( "[[Prototype]]" ) ] = obj2;

obj1.bar();
// bar: obj-1 <-- 通过代理假冒 [[Prototype]]
// foo: obj-1 <-- `this` 上下文环境依然被保留

obj2.foo();
// foo: obj-2 <-- 通过 [[Prototype]]
```

注意： 为了让事情简单一些，在这个例子中我们没有代理/转送 `[[Set]]` 。要完整地模拟 `[[Prototype]]` 兼容，你会想要实现一个 `set(..)` 处理器，它在 `[[Prototype]]` 链上检索一个匹配得属性并遵循它的描述符的行为（例如，set，可写性）。参见本系列的 *this*与对象原型。

在前面的代码段中， `obj2` 凭借 `Object.create(..)` 语句 `[[Prototype]]` 链接到 `obj1` 。但是要创建反向（循环）的链接，我们在 `obj1` 的symbol位置 `Symbol.for("[[Prototype]]")` （参见第二章的“Symbol”）上创建了一个属性。这个symbol可能看起来有些特别/魔幻，但它不是的。它只是允许我使用一个被方便地命名的属性，这个属性在语义上看来是与我进行的任务有关联的。

然后，代理的 `get(..)` 处理器首先检查一个被请求的 `key` 是否存在于代理上。如果每个有，操作就被手动地交给存储在 `target` 的 `Symbol.for("[[Prototype]]")` 位置中的对象引用。

这种模式的一个重要优点是，在 `obj1` 和 `obj2` 之间建立循环关系几乎没有入侵它们的定义。虽然前面的代码段为了简短而将所有的步骤交织在一起，但是如果你仔细观察，代理处理器的逻辑完全是范用的（不具体地知道 `obj1` 或 `obj2` ）。所以，这段逻辑可以抽出到一个简单的将它们连在一起的帮助函数中，例如 `setCircularPrototypeOf(..)` 。我们将此作为一个练习留给读者。

现在我们看到了如何使用 `get(..)` 来模拟一个 `[[Prototype]]` 链接，但让我们将这种黑科技推动的远一些。与其制造一个循环 `[[Prototype]]` ，搞一个多重 `[[Prototype]]` 链接（也就是“多重继承”）怎么样？这看起来相当直白：

```
var obj1 = {
        name: "obj-1",
```

```
        foo() {
            console.log( "obj1.foo:", this.name );
        },
    },
    obj2 = {
        name: "obj-2",
        foo() {
            console.log( "obj2.foo:", this.name );
        },
        bar() {
            console.log( "obj2.bar:", this.name );
        }
    },
    handlers = {
        get(target,key,context) {
            if (Reflect.has( target, key )) {
                return Reflect.get(
                    target, key, context
                );
            }
            // 假冒多重 `[[Prototype]]`
            else {
                for (var P of target[
                    Symbol.for( "[[Prototype]]" )
                ]) {
                    if (Reflect.has( P, key )) {
                        return Reflect.get(
                            P, key, context
                        );
                    }
                }
            }
        }
    },
    obj3 = new Proxy(
        {
            name: "obj-3",
            baz() {
                this.foo();
                this.bar();
            }
        },
        handlers
    );

// 假冒多重 `[[Prototype]]` 链接
obj3[ Symbol.for( "[[Prototype]]" ) ] = [
    obj1, obj2
```

```
51. ];
52.
53. obj3.baz();
54. // obj1.foo: obj-3
55. // obj2.bar: obj-3
```

注意： 正如在前面的循环 `[[Prototype]]` 例子后的注意中提到的，我们没有实现 `set(..)` 处理器，但对于一个将 `[[Set]]` 模拟为普通 `[[Prototype]]` 行为的解决方案来说，它将是必要的。

`obj3` 被设置为多重委托到 `obj1` 和 `obj2` 。在 `obj2.baz()` 中， `this.foo()` 调用最终成为从 `obj1` 中抽出 `foo()` （先到先得，虽然还有一个在 `obj2` 上的 `foo()` ）。如果我们将连接重新排列为 `obj2, obj1` ，那么 `obj2.foo()` 将被找到并使用。

同理， `this.bar()` 调用没有在 `obj1` 上找到 `bar()` ，所以它退而检查 `obj2` ，这里找到了一个匹配。

`obj1` 和 `obj2` 代表 `obj3` 的两个平行的 `[[Prototype]]` 链。 `obj1` 和/或 `obj2` 自身可以拥有委托至其他对象的普通 `[[Prototype]]` ，或者自身也可以是多重委托的代理（就像 `obj3` 一样）。

正如先前的循环 `[[Prototype]]` 的例子一样， `obj1` ， `obj2` 和 `obj3` 的定义几乎完全与处理多重委托的范用代理逻辑相分离。定义一个 `setPrototypesOf(..)` （注意那个“s”！）这样的工具将是小菜一碟，它接收一个主对象和一组模拟多重 `[[Prototype]]` 链接用的对象。同样，我们将此作为练习留给读者。

希望在这种种例子之后代理的力量现在变得明朗了。代理使得许多强大的元编程任务成为可能。

Reflect API

- `Reflect` API
  - 属性顺序

`Reflect` API

`Reflect` 对象是一个普通对象（就像 `Math` ），不是其他内建原生类型那样的函数/构造器。

它持有对应于你可以控制的各种元编程任务的静态函数。这些函数与代理可以定义的处理器方法（机关）一一对应。

这些函数中的一些看起来与在 `Object` 上的同名函数很相似：

- `Reflect.getOwnPropertyDescriptor(..)`
- `Reflect.defineProperty(..)`
- `Reflect.getPrototypeOf(..)`
- `Reflect.setPrototypeOf(..)`
- `Reflect.preventExtensions(..)`
- `Reflect.isExtensible(..)`

这些工具一般与它们的 `Object.*` 对等物的行为相同。但一个区别是， `Object.*` 对等物在它们的第一个参数值（目标对象）还不是对象的情况下，试图将它强制转换为一个对象。 `Reflect.*` 方法在同样的情况下仅简单地抛出一个错误。

一个对象的键可以使用这些工具访问/检测：

- `Reflect.ownKeys(..)`：返回一个所有直属（不是“继承的”）键的列表，正如被 `Object.getOwnPropertyNames(..)` 和 `Object.getOwnPropertySymbols(..)` 返回的那样。关于键的顺序问题，参见“属性枚举顺序”一节。
- `Reflect.enumerate(..)`：返回一个产生所有（直属和“继承的”）非symbol、可枚举的键的迭代器（参见本系列的 *this与对象原型*）。 实质上，这组键与在 `for..in` 循环中被处理的那一组键是相同的。关于键的顺序问题，参见“属性枚举顺序”一节。
- `Reflect.has(..)`：实质上与用于检查一个属性是否存在于一个对象或它的 `[[Prototype]]` 链上的 `in` 操作符相同。例如， `Reflect.has(o,"foo")` 实质上实施 `"foo" in o` 。

函数调用和构造器调用可以使用这些工具手动地实施，与普通的语法（例如， `(..)` 和 `new` ）分开：

- `Reflect.apply(..)`：例如， `Reflect.apply(foo,thisObj,[42,"bar"])` 使用 `thisObj` 作为 `foo(..)` 函数的 `this` 来调用它，并传入参数值 `42` 和 `"bar"` 。
- `Reflect.construct(..)`：例如， `Reflect.construct(foo,[42,"bar"])` 实质上调用 `new foo(42,"bar")` 。

对象属性访问，设置，和删除可以使用这些工具手动实施：

- `Reflect.get(..)`：例如，`Reflect.get(o,"foo")`会取得`o.foo`。
- `Reflect.set(..)`：例如，`Reflect.set(o,"foo",42)`实质上实施`o.foo = 42`。
- `Reflect.deleteProperty(..)`：例如，`Reflect.deleteProperty(o,"foo")`实质上实施`delete o.foo`。

`Reflect`的元编程能力给了你可以模拟各种语法特性的程序化等价物，暴露以前隐藏着的抽象操作。例如，你可以使用这些能力来扩展 领域特定语言（DSL）的特性和API。

属性顺序

在ES6之前，罗列一个对象的键/属性的顺序没有在语言规范中定义，而是依赖于具体实现的。一般来说，大多数引擎会以创建的顺序来罗列它们，虽然开发者们已经被强烈建议永远不要依仗这种顺序。

在ES6中，罗列直属属性的属性是由`[[OwnPropertyKeys]]`算法定义的（ES6语言规范，9.1.12部分），它产生所有直属属性（字符串或symbol），不论其可枚举性。这种顺序仅对`Reflect.ownKeys(..)`有保证（）。

这个顺序是：

1. 首先，以数字上升的顺序，枚举所有数字索引的直属属性。
2. 然后，以创建顺序枚举剩下的直属字符串属性名。
3. 最后，以创建顺序枚举直属symbol属性。

考虑如下代码：

```
var o = {};

o[Symbol("c")] = "yay";
o[2] = true;
o[1] = true;
o.b = "awesome";
o.a = "cool";

Reflect.ownKeys( o );               // [1,2,"b","a",Symbol(c)]
Object.getOwnPropertyNames( o );    // [1,2,"b","a"]
Object.getOwnPropertySymbols( o );    // [Symbol(c)]
```

另一方面，`[[Enumeration]]`算法（ES6语言规范，9.1.11部分）从目标对象和它的`[[Prototype]]`链中仅产生可枚举属性。它被用于`Reflect.enumerate(..)`和`for..in`。可观察到的顺序是依赖于具体实现的，语言规范没有控制它。

相比之下，`Object.keys(..)`调用`[[OwnPropertyKeys]]`算法来得到一个所有直属属性的列表。但是，它过滤掉了不可枚举属性，然后特别为了`JSON.stringify(..)`和`for..in`而将这个列表重排，

以匹配遗留的、依赖于具体实现的行为。所以通过扩展，这个顺序 也 与 `Reflect.enumerate(..)` 的顺序像吻合。

换言之，所有四种机制（ `Reflect.enumerate(..)` ， `Object.keys(..)` ， `for..in` ，和 `JSON.stringify(..)` ）都同样将与依赖于具体实现的顺序像吻合，虽然技术上它们是以不同的方式达到的同样的效果。

具体实现可以将这四种机制与 `[[OwnPropertyKeys]]` 的顺序相吻合，但不是必须的。无论如何，你将很可能从它们的行为中观察到以下的排序：

```
var o = { a: 1, b: 2 };
var p = Object.create( o );
p.c = 3;
p.d = 4;

for (var prop of Reflect.enumerate( p )) {
    console.log( prop );
}
// c d a b

for (var prop in p) {
    console.log( prop );
}
// c d a b

JSON.stringify( p );
// {"c":3,"d":4}

Object.keys( p );
// ["c","d"]
```

这一切可以归纳为：在ES6中，根据语言规范 `Reflect.ownKeys(..)` ， `Object.getOwnPropertyNames(..)` ，和 `Object.getOwnPropertySymbols(..)` 保证都有可预见和可靠的顺序。所以依赖于这种顺序来建造代码是安全的。

`Reflect.enumerate(..)` ， `Object.keys(..)` ，和 `for..in` （扩展一下的话还有 `JSON.stringify(..)` ）继续互相共享一个可观察的顺序，就像它们往常一样。但这个顺序不一定与 `Reflect.ownKeys(..)` 的相同。在使用它们依赖于具体实现的顺序时依然应当小心。

特性测试

- 特性测试
  - FeatureTests.io

特性测试

什么是特性测试？它是一种由你运行来判定一个特性是否可用的测试。有些时候，这种测试不仅是为了判定存在性，还是为判定对特定行为的适应性 —— 特性可能存在但有bug。

这是一种元编程技术 —— 测试你程序将要运行的环境然后判定你的程序应当如何动作。

在JS中特性测试最常见的用法是检测一个API的存在性，而且如果它不存在，就定义一个填补（见第一章）。例如：

```
if (!Number.isNaN) {
    Number.isNaN = function(x) {
        return x !== x;
    };
}
```

在这个代码段中的 `if` 语句就是一个元编程：我们探测我们的程序和它的运行时环境，来判定我们是否和如何进行后续处理。

但是如何测试一个涉及新语法的特性呢？

你可能会尝试这样的东西：

```
try {
    a = () => {};
    ARROW_FUNCS_ENABLED = true;
}
catch (err) {
    ARROW_FUNCS_ENABLED = false;
}
```

不幸的是，这不能工作，因为我们的JS程序是要被编译的。因此，如果引擎还没有支持ES6箭头函数的话，它就会在 `() => {}` 语法的地方熄火。你程序中的语法错误会阻止它的运行，进而阻止你程序根据特性是否被支持而进行后续的不同相应。

为了围绕语法相关的特性进行特性测试的元编程，我们需要一个方法将测试与我们程序将要通过的初始编译步骤隔离开。举例来说，如果我们能够将进行测试的代码存储在一个字符串中，之后JS引擎默

认地将不会尝试编译这个字符串中的内容，直到我们要求它这么做。

你的思路是不是跳到了使用 `eval(..)` ？

别这么着急。看看本系列的 作用域与闭包 来了解一下为什么 `eval(..)` 是一个坏主意。但是有另外一个缺陷较少的选项： `Function(..)` 构造器。

考虑如下代码：

```
try {
    new Function( "( () => {} )" );
    ARROW_FUNCS_ENABLED = true;
}
catch (err) {
    ARROW_FUNCS_ENABLED = false;
}
```

好了，现在我们判定一个像箭头函数这样的特性是否 能 被当前的引擎所编译来进行元编程。你可能会想知道，我们要用这种信息做什么？

检查API的存在性，并定义后备的API填补，对于特性检测成功或失败来说都是一条明确的道路。但是对于从 `ARROW_FUNCS_ENABLED` 是 `true` 还是 `false` 中得到的信息来说，我们能对它做什么呢？

因为如果引擎不支持一种特性，它的语法就不能出现在一个文件中，所以你不能在这个文件中定义使用这种语法的函数。

你所能做的是，使用测试来判定你应当加载哪一组JS文件。例如，如果在你的JS应用程序中的启动装置中有一组这样的特性测试，那么它就可以测试环境来判定你的ES6代码是否可以直接加载运行，或者你是否需要加载一个代码的转译版本（参见第一章）。

这种技术称为 分割投递。

事实表明，你使用ES6编写的JS程序有时可以在ES6+浏览器中完全“原生地”运行，但是另一些时候需要在前ES6浏览器中运行转译版本。如果你总是加载并使用转译代码，即便是在新的ES6兼容环境中，至少是有些情况下你运行的也是次优的代码。这并不理想。

分割投递更加复杂和精巧，但对于你编写的代码和你的程序所必须在其中运行的浏览器支持的特性之间，它代表一种更加成熟和健壮的桥接方式。

FeatureTests.io

为所有的ES6+语法以及语义行为定义特性测试，是一项你可能不想自己解决的艰巨任务。因为这些测试要求动态编译（ `new Function(..)` ），这会产生不幸的性能损耗。

另外，在每次你的应用运行时都执行这些测试可能是一种浪费，因为平均来说一个用户的浏览器在几

周之内至多只会更新一次，而即使是这样，新特性也不一定会在每次更新中都出现。

最终，管理一个对你特定代码库进行的特性测试列表 —— 你的程序将很少用到ES6的全部 —— 是很容易失控而且易错的。

“https://featuretests.io”的“特性测试服务”为这种挫折提供了解决方案。

你可以将这个服务的库加载到你的页面中，而它会加载最新的测试定义并运行所有的特性测试。在可能的情况下，它将使用Web Worker的后台处理中这样做，以降低性能上的开销。它还会使用LocalStorage持久化来缓存测试的结果 —— 以一种可以被所有你访问的使用这个服务的站点所共享的方式，这将及大地降低测试需要在每个浏览器实例上运行的频度。

你可以在每一个用户的浏览器上进行运行时特性测试，而且你可以使用这些测试结果动态地向用户传递最适合他们环境的代码（不多也不少）。

另外，这个服务还提供工具和API来扫描你的文件以判定你需要什么特性，这样你就能够完全自动化你的分割投递构建过程。

对ES6的所有以及未来的部分进行特性测试，以确保对于任何给定的环境都只有最佳的代码会被加载和运行 —— FeatureTests.io使这成为可能。

尾部调用优化（TCO）

- 尾部调用优化（TCO）
  - 重写尾部调用
  - 非TCO优化
  - 元？
    - 自我调整的代码

尾部调用优化（TCO）

通常来说，当从一个函数内部发起对另一个函数的调用时，就会分配一个 栈帧 来分离地管理这另一个函数调用的变量/状态。这种分配不仅花费一些处理时间，还会消耗一些额外的内存。

一个调用栈链从一个函数到另一个再到另一个，通常至多拥有10-15跳。在这些场景下，内存使用不太可能是某种实际问题。

然而，当你考虑递归编程（一个函数频繁地调用自己） —— 或者使用两个或更多的函数相互调用而构成相互递归 —— 调用栈就可能轻易地到达上百，上千，或更多层的深度。如果内存的使用无限制地增长下去，你可能看到了它将导致的问题。

JavaScript引擎不得不设置一个随意的限度来防止这样的编程技术耗尽浏览器或设备的内存。这就是为什么我们会在到达这个限度时得到令人沮丧的“RangeError: Maximum call stack size exceeded”。

警告： 调用栈深度的限制是不由语言规范控制的。它是依赖于具体实现的，而且将会根据浏览器和设备不同而不同。你绝不应该带着可精确观察到的限度的强烈臆想进行编码，因为它们还很可能在每个版本中变化。

一种称为 尾部调用 的特定函数调用模式，可以以一种避免额外的栈帧分配的方法进行优化。如果额外的分配可以被避免，那么就没有理由随意地限制调用栈的深度，这样引擎就可以让它们没有边界地运行下去。

一个尾部调用是一个带有函数调用的 `return` 语句，除了返回它的值，函数调用之后没有任何事情需要发生。

这种优化只能在 `strict` 模式下进行。又一个你总是应该用 `strict` 编写所有代码的理由！

这个函数调用 不是 在尾部：

```
1. "use strict";
2.
3. function foo(x) {
4.     return x * 2;
```

```
}

function bar(x) {
    // 不是一个尾部调用
    return 1 + foo( x );
}

bar( 10 );                // 21
```

在 `foo(x)` 调用完成后必须进行 `1 + ..` ，所以那个 `bar(..)` 调用的状态需要被保留。

但是下面的代码段中展示的 `foo(..)` 和 `bar(..)` 都是位于尾部，因为它们都是在自身代码路径上（除了 `return` 以外）发生的最后一件事：

```
"use strict";

function foo(x) {
    return x * 2;
}

function bar(x) {
    x = x + 1;
    if (x > 10) {
        return foo( x );
    }
    else {
        return bar( x + 1 );
    }
}

bar( 5 );               // 24
bar( 15 );               // 32
```

在这个程序中， `bar(..)` 明显是递归，但 `foo(..)` 只是一个普通的函数调用。这两个函数调用都位于 恰当的尾部位置。 `x + 1` 在 `bar(..)` 调用之前被求值，而且不论这个调用何时完成，所有将要放生的只有 `return` 。

这些形式的恰当尾部调用（Proper Tail Calls — PTC）是可以被优化的 — 称为尾部调用优化（TCO）— 于是额外的栈帧分配是不必要的。与为下一个函数调用创建新的栈帧不同，引擎会重用既存的栈帧。这能够工作是因为一个函数不需要保留任何当前状态 — 在PTC之后的状态下不会发生任何事情。

TCO意味着调用栈可以有多深实际上是没有限度的。这种技巧稍稍改进了一般程序中的普通函数调用，但更重要的是它打开了一扇大门：可以使用递归表达程序，即使它的调用栈深度有成千上万层。

我们不再局限于单纯地在理论上考虑用递归解决问题了，而是可以在真实的JavaScript程序中使用

它！

作为ES6，所有的PTC都应该是可以以这种方式优化的，不论递归与否。

重写尾部调用

然而，障碍是只有PTC是可以被优化的；非PTC理所当然地依然可以工作，但是将造成往常那样的栈帧分配。如果你希望优化机制启动，就必须小心地使用PTC构造你的函数。

如果你有一个没有用PTC编写的函数，你可能会发现你需要手动地重新安排你的代码，使它成为合法的TCO。

考虑如下代码：

```
"use strict";

function foo(x) {
    if (x <= 1) return 1;
    return (x / 2) + foo( x - 1 );
}

foo( 123456 );            // RangeError
```

对 `foo(x-1)` 的调用不是一个PTC，因为在 `return` 之前它的结果必须被加上 `(x / 2)` 。

但是，要使这段代码在一个ES6引擎中是合法的TCO，我们可以像下面这样重写它：

```
"use strict";

var foo = (function(){
    function _foo(acc,x) {
        if (x <= 1) return acc;
        return _foo( (x / 2) + acc, x - 1 );
    }

    return function(x) {
        return _foo( 1, x );
    };
})();

foo( 123456 );            // 3810376848.5
```

如果你在一个实现了TCO的ES6引擎中运行前面这个代码段，你将会如展示的那样得到答案 `3810376848.5` 。然而，它仍然会在非TCO引擎中因为 `RangeError` 而失败。

非TCO优化

有另一种技术可以重写代码，让调用栈不随每次调用增长。

一个这样的技术称为 蹦床，它相当于让每一部分结果表示为一个函数，这个函数要么返回另一个部分结果函数，要么返回最终结果。然后你就可以简单地循环直到你不再收到一个函数，这时你就得到了结果。考虑如下代码：

```
1. "use strict";
2.
3. function trampoline( res ) {
4.     while (typeof res == "function") {
5.         res = res();
6.     }
7.     return res;
8. }
9.
10. var foo = (function(){
11.     function _foo(acc,x) {
12.         if (x <= 1) return acc;
13.         return function partial(){
14.             return _foo( (x / 2) + acc, x - 1 );
15.         };
16.     }
17.
18.     return function(x) {
19.         return trampoline( _foo( 1, x ) );
20.     };
21. })();
22.
23. foo( 123456 );            // 3810376848.5
```

这种返工需要一些最低限度的改变来将递归抽出到 `trampoline(..)` 中的循环中：

1. 首先，我们将`return _foo ..`这一行包装进函数表达式`return partial() {..`。
2. 然后我们将`_foo(1,x)`包装进`trampoline(..)`调用。

这种技术之所以不受调用栈限制的影响，是因为每个内部的 `partial(..)` 函数都只是返回到 `trampoline(..)` 的 `while` 循环中，这个循环运行它然后再一次循环迭代。换言之， `partial(..)` 并不递归地调用它自己，它只是返回另一个函数。栈的深度维持不变，所以它需要运行多久就可以运行多久。

蹦床表达的是，内部的 `partial()` 函数使用在变量 `x` 和 `acc` 上的闭包来保持迭代与迭代之间的状态。它的优势是循环的逻辑可以被抽出到一个可重用的 `trampoline(..)` 工具函数中，许多库都提供这个工具的各种版本。你可以使用不同的蹦床算法在你的程序中重用 `trampoline(..)` 多次。

当然，如果你真的想要深度优化（于是可复用性不予考虑），你可以摒弃闭包状态，并将对 `acc` 的状态追踪，与一个循环一起内联到一个函数的作用域内。这种技术通常称为 递归展开：

```
"use strict";

function foo(x) {
    var acc = 1;
    while (x > 1) {
        acc = (x / 2) + acc;
        x = x - 1;
    }
    return acc;
}

foo( 123456 );            // 3810376848.5
```

算法的这种表达形式很容易阅读，而且很可能是在我们探索过的各种形式中性能最好的（严格地说）一个。很明显它看起来是一个胜利者，而且你可能会想知道为什么你曾尝试其他的方式。

这些是为什么你可能不想总是手动地展开递归的原因：

- 与为了复用而将弹簧（循环）逻辑抽出去相比，我们内联了它。这在仅有一个这样的例子需要考虑时工作的很好，但只要你在程序中有五六个或更多这样的东西时，你将很可能想要一些可复用性来将让事情更简短、更易管理一些。
- 这里的例子为了展示不同的形式而被故意地搞得很简单。在现实中，递归算法有着更多的复杂性，比如相互递归（有多于一个的函数调用它自己）。

  你在这条路上走得越远，展开 优化就变得越复杂和越依靠手动。你很快就会失去所有可读性的认知价值。递归，甚至是PTC形式的递归的主要优点是，它保留了算法的可读性，并将性能优化的任务交给引擎。

如果你使用PTC编写你的算法，ES6引擎将会实施TCO来使你的代码运行在一个定长深度的栈中（通过重用栈帧）。你将在得到递归的可读性的同时，也得到性能上的大部分好处与无限的运行长度。

元？

TCO与元编程有什么关系？

正如我们在早先的“特性测试”一节中讲过的，你可以在运行时判定一个引擎支持什么特性。这也包括TCO，虽然判定的过程相当粗暴。考虑如下代码：

```
"use strict";

try {
```

```
    (function foo(x){
        if (x < 5E5) return foo( x + 1 );
    })( 1 );

    TCO_ENABLED = true;
}
catch (err) {
    TCO_ENABLED = false;
}
```

在一个非TCO引擎中，递归循环最终将会失败，抛出一个被 `try..catch` 捕获的异常。否则循环将由TCO轻易地完成。

讨厌，对吧？

但是围绕着TCO特性进行的元编程（或者，没有它）如何给我们的代码带来好处？简单的答案是你可以使用这样的特性测试来决定加载一个你的应用程序的使用递归的版本，还是一个被转换/转译为不需要递归的版本。

自我调整的代码

但这里有另外一种看待这个问题的方式：

```
"use strict";

function foo(x) {
    function _foo() {
        if (x > 1) {
            acc = acc + (x / 2);
            x = x - 1;
            return _foo();
        }
    }

    var acc = 1;

    while (x > 1) {
        try {
            _foo();
        }
        catch (err) { }
    }

    return acc;
}

foo( 123456 );          // 3810376848.5
```

这个算法试图尽可能多地使用递归来工作，但是通过作用域中的变量 `x` 和 `acc` 来跟踪这个进程。如果整个问题可以通过递归没有错误地解决，很好。如果引擎在某一点终止了递归，我们简单地使用 `try..catch` 捕捉它，然后从我们离开的地方再试一次。

我认为这是一种形式的元编程，因为你在运行时期间探测着引擎是否能（递归地）完成任务的能力，并绕过了任何可能制约你的（非TCO的）引擎的限制。

一眼（或者是两眼！）看上去，我打赌这段代码要比以前的版本难看许多。它运行起来还相当地慢一些（在一个非TCO环境中长时间运行的情况下）。

它主要的优势是，除了在非TCO引擎中也能完成任意栈大小的任务外，这种对递归栈限制的“解法”要比前面展示的蹦床和手动展开技术灵活得多。

实质上，这种情况下的 `_foo()` 实际上是任意递归任务，甚至是相互递归的某种替身。剩下的内容是应当对任何算法都可以工作的模板代码。

唯一的“技巧”是为了能够在达到递归限制的事件发生时继续运行，递归的状态必须保存在递归函数外部的作用域变量中。我们是通过将 `x` 和 `acc` 留在 `_foo()` 函数外面这样做的，而不是像早先那样将它们作为参数值传递给 `_foo()` 。

几乎所有的递归算法都可以采用这种方法工作。这意味着它是在你的程序中，进行最小的重写就能利用TCO递归的最广泛的可行方法。

这种方式仍然使用一个PTC，意味着这段代码将会 渐进增强：从在一个老版浏览器中使用许多次循环（递归批处理）来运行，到在一个ES6+环境中完全利用TCO递归。我觉得这相当酷！

复习

复习

元编程是当你将程序的逻辑转向关注它自身（或者它的运行时环境）时进行的编程，要么为了调查它自己的结构，要么为了修改它。元编程的主要价值是扩展语言的普通机制来提供额外的能力。

在ES6以前，JavaScript已经有了相当的元编程能力，但是ES6使用了几个新特性及大地提高了它的地位。

从对匿名函数的函数名推断，到告诉你一个构造器是如何被调用的元属性，你可以前所未有地在程序运行期间来调查它的结构。通用Symbols允许你覆盖固有的行为，比如将一个对象转换为一个基本类型值的强制转换。代理可以拦截并自定义各种在对象上的底层操作，而且 `Reflect` 提供了模拟它们的工具。

特性测试，即便是对尾部调用优化这样微妙的语法行为，将元编程的焦点从你的程序提升到JS引擎的能力本身。通过更多地了解环境可以做什么，你的程序可以在运行时将它们自己调整到最佳状态。

你应该进行元编程吗？我的建议是：先集中学习这门语言的核心机制是如何工作的。一旦你完全懂得了JS本身可以做什么，就是开始利用这些强大的元编程能力将这门语言向前推进的时候了！

第八章：ES6 以后

- 第八章：ES6以后

第八章：ES6以后

在本书写作的时候，ES6（*ECMAScript 2015*）的最终草案即将为了ECMA的批准而进行最终的官方投票。但即便是在ES6已经被最终定稿的时候，TC39协会已经在为了ES7/2016和将来的特性进行努力的工作。

正如我们在第一章中讨论过的，预计JS进化的节奏将会从好几年升级一次加速到每年进行一次官方的版本升级（因此采用编年命名法）。这将会彻底改变JS开发者学习与跟上这门语言脚步的方式。

但更重要的是，协会实际上将会一个特性一个特性地进行工作。只要一种特性的规范被定义完成，而且通过在几种浏览器中的实验性实现打通了关节，那么这种特性就会被认为足够稳定并可以开始使用了。我们都被强烈鼓励一旦特性准备好就立即采用它，而不是等待什么官方标准投票。如果你还没学过ES6，现在上船的日子已经过了！

在本书写作时，一个未来特性提案的列表和它们的状态可以在这里看到(https://github.com/tc39/ecma262#current-proposals)。

在所有我们支持的浏览器实现这些新特性之前，转译器和填补是我们如何桥接它们的方法。Babel，Traceur，和其他几种主流转译器已经支持了一些最可能稳定下来的ES6之后的特性。

认识到这一点，是时候看一看它们之中的一些了。让我们开始吧！

警告： 这些特性都处于开发的各种阶段。虽然它们很可能确定下来，而且将与本章的内容看起来相似，但还是要抱着更多质疑的态度看待本章的内容。这一章将会在本书未来的版本中随着这些（和其他的！）特性的确定而演化。

- async function
- Object.observe(..)
- 指数操作符
- 对象属性与 ...
- Array#includes(..)
- SIMD
- WebAssembly (WASM)
- 复习

async function

- `async function`
  - 警告

`async function`

我们在第四章的“Generators + Promises”中提到过，generator `yield` 一个promise给一个类似运行器的工具，它会在promise完成时推进generator —— 有一个提案是要为这种模式提供直接的语法支持。让我们简要看一下这个被提出的特性，它称为 `async function` 。

回想一下第四章中的这个generator的例子：

```
run( function *main() {
    var ret = yield step1();

    try {
        ret = yield step2( ret );
    }
    catch (err) {
        ret = yield step2Failed( err );
    }

    ret = yield Promise.all([
        step3a( ret ),
        step3b( ret ),
        step3c( ret )
    ]);

    yield step4( ret );
} )
.then(
    function fulfilled(){
        // `*main()` 成功地完成了
    },
    function rejected(reason){
        // 噢，什么东西搞错了
    }
);
```

被提案的 `async function` 语法可以无需 `run(..)` 工具就表达相同的流程控制逻辑，因为JS将会自动地知道如何寻找promise来等待和推进。考虑如下代码：

```
async function main() {
    var ret = await step1();
```

```
3.
4.      try {
5.          ret = await step2( ret );
6.      }
7.      catch (err) {
8.          ret = await step2Failed( err );
9.      }
10.
11.     ret = await Promise.all( [
12.         step3a( ret ),
13.         step3b( ret ),
14.         step3c( ret )
15.     ] );
16.
17.     await step4( ret );
18. }
19.
20. main()
21. .then(
22.     function fulfilled(){
23.         // `main()` 成功地完成了
24.     },
25.     function rejected(reason){
26.         // 噢，什么东西搞错了
27.     }
28. );
```

取代 `function *main() { ..` 声明的，是我们使用 `async function main() { ..` 形式声明。而取代 `yield` 一个promise的，是我们 `await` 这个promise。运行 `main()` 函数的调用实际上返回一个我们可以直接监听的promise。这与我们从一个 `run(main)` 调用中拿回一个promise是等价的。

你看到对称性了吗？ `async function` 实质上是 generators + promises + `run(..)` 模式的语法糖；它们在底层的操作是相同的！

如果你是一个C#开发者而且这种 `async` / `await` 看起来很熟悉，那是因为这种特性就是直接由C#的特性启发的。看到语言提供一致性是一件好事！

Babel、Traceur 以及其他转译器已经对当前的 `async function` 状态有了早期支持，所以你已经可以使用它们了。但是，在下一节的“警告”中，我们将看到为什么你也许还不应该上这艘船。

注意： 还有一个 `async function*` 的提案，它应当被称为“异步generator”。你可以在同一段代码中使用 `yield` 和 `await` 两者，甚至是在同一个语句中组合这两个操作： `x = await yield y` 。“异步generator”提案看起来更具变化 —— 也就是说，它返回一个没有还没有完全被计算好的值。一些人觉得它应当是一个 可监听对象（*observable*），有些像是一个迭代器和promise的组合。就目前来说，我们不会进一步探讨这个话题，但是会继续关注它的演变。

警告

关于 `async function` 的一个未解的争论点是，因为它仅返回一个promise，所以没有办法从外部 撤销 一个当前正在运行的 `async function` 实例。如果这个异步操作是资源密集型的，而且你想在自己确定不需要它的结果时能立即释放资源，这可能是一个问题。

举例来说：

```
async function request(url) {
    var resp = await (
        new Promise( function(resolve,reject){
            var xhr = new XMLHttpRequest();
            xhr.open( "GET", url );
            xhr.onreadystatechange = function(){
                if (xhr.readyState == 4) {
                    if (xhr.status == 200) {
                        resolve( xhr );
                    }
                    else {
                        reject( xhr.statusText );
                    }
                }
            };
            xhr.send();
        } )
    );

    return resp.responseText;
}

var pr = request( "http://some.url.1" );

pr.then(
    function fulfilled(responseText){
        // ajax 成功
    },
    function rejected(reason){
        // 噢，什么东西搞错了
    }
);
```

我构想的 `request(..)` 有点儿像最近被提案要包含进web平台的 `fetch(..)` 工具。我们关心的是，例如，如果你想要用 `pr` 值以某种方法指示撤销一个长时间运行的Ajax请求会怎么样？

Promise是不可撤销的（在本书写作时）。在我和其他许多人看来，它们就不应该是可以被撤销的（参见本系列的 异步与性能）。而且即使一个proimse确实拥有一个 `cancel()` 方法，那么一定意

味着调用 `pr.cancel()` 应当真的沿着promise链一路传播一个撤销信号到 `async function` 吗？

对于这个争论的几种可能的解决方案已经浮出水面：

- `async function`将根本不能被撤销（现状）
- 一个“撤销存根”可以在调用时传递给一个异步函数
- 将返回值改变为一个新增的可撤销promsie类型
- 将返回值改变为非promise的其他东西（比如，可监听对象，或带有promise和撤销能力的控制存根）

在本书写作时， `async function` 返回普通的promise，所以完全改变返回值不太可能。但是现在下定论还是为时过早了。让我们持续关注这个讨论吧。

Object.observe(..)

- `Object.observe(..)`
  - 自定义变化事件
  - 中止监听

`Object.observe(..)`

前端web开发的圣杯之一就是数据绑定 —— 监听一个数据对象的更新并同步这个数据的DOM表现形式。大多数JS框架都为这些类型的操作提供某种机制。

在ES6后期，我们似乎很有可能看到这门语言通过一个称为 `Object.observe(..)` 的工具，对此提供直接的支持。实质上，它的思想是你可以建立监听器来监听一个对象的变化，并在一个变化发生的任何时候调用一个回调。例如，你可相应地更新DOM。

你可以监听六种类型的变化：

- `add`
- `update`
- `delete`
- `reconfigure`
- `setPrototype`
- `preventExtensions`

默认情况下，你将会收到所有这些类型的变化的通知，但是你可以将它们过滤为你关心的那一些。

考虑如下代码：

```
var obj = { a: 1, b: 2 };

Object.observe(
    obj,
    function(changes){
        for (var change of changes) {
            console.log( change );
        }
    },
    [ "add", "update", "delete" ]
);

obj.c = 3;
// { name: "c", object: obj, type: "add" }

obj.a = 42;
// { name: "a", object: obj, type: "update", oldValue: 1 }
```

```
18.
19. delete obj.b;
20. // { name: "b", object: obj, type: "delete", oldValue: 2 }
```

除了主要的 `"add"` 、 `"update"` 、和 `"delete"` 变化类型：

- `"reconfigure"` 变化事件在对象的一个属性通过 `Object.defineProperty(..)` 而重新配置时触发，比如改变它的 `writable` 属性。更多信息参见本系列的 *this* 与对象原型。
- `"preventExtensions"` 变化事件在对象通过 `Object.preventExtensions(..)` 被设置为不可扩展时触发。

  因为 `Object.seal(..)` 和 `Object.freeze(..)` 两者都暗示着 `Object.preventExtensions(..)` ，所以它们也将触发相应的变化事件。另外， `"reconfigure"` 变化事件也会为对象上的每个属性被触发。
- `"setPrototype"` 变化事件在一个对象的 `[[Prototype]]` 被改变时触发，不论是使用 `__proto__` setter，还是使用 `Object.setPrototypeOf(..)` 设置它。

注意，这些变化事件在会在变化发生后立即触发。不要将它们与代理（见第七章）搞混，代理是可以在动作发生之前拦截它们的。对象监听让你在变化（或一组变化）发生之后进行应答。

自定义变化事件

除了六种内建的变化事件类型，你还可以监听并触发自定义变化事件。

考虑如下代码：

```
1. function observer(changes){
2.     for (var change of changes) {
3.         if (change.type == "recalc") {
4.             change.object.c =
5.                 change.object.oldValue +
6.                 change.object.a +
7.                 change.object.b;
8.         }
9.     }
10. }
11.
12. function changeObj(a,b) {
13.     var notifier = Object.getNotifier( obj );
14.
15.     obj.a = a * 2;
16.     obj.b = b * 3;
17.
18.     // queue up change events into a set
```

```
    notifier.notify( {
        type: "recalc",
        name: "c",
        oldValue: obj.c
    } );
}

var obj = { a: 1, b: 2, c: 3 };

Object.observe(
    obj,
    observer,
    ["recalc"]
);

changeObj( 3, 11 );

obj.a;            // 12
obj.b;            // 30
obj.c;            // 3
```

变化的集合（ `"recalc"` 自定义事件）为了投递给监听器而被排队，但还没被投递，这就是为什么 `obj.c` 依然是 `3` 。

默认情况下，这些变化将在当前事件轮询（参见本系列的 异步与性能）的末尾被投递。如果你想要立即投递它们，使用 `Object.deliverChangeRecords(observer)` 。一旦这些变化投递完成，你就可以观察到 `obj.c` 如预期地更新为：

```
obj.c;            // 42
```

在前面的例子中，我们使用变化完成事件的记录调用了 `notifier.notify(..)` 。将变化事件的记录进行排队的一种替代形式是使用 `performChange(..)` ，它把事件的类型与事件记录的属性（通过一个函数回调）分割开来。考虑如下代码：

```
notifier.performChange( "recalc", function(){
    return {
        name: "c",
        // `this` 是被监听的对象
        oldValue: this.c
    };
} );
```

在特定的环境下，这种关注点分离可能与你的使用模式匹配的更干净。

中止监听

正如普通的事件监听器一样，你可能希望停止监听一个对象的变化事件。为此，你可以使用 `Object.unobserve(..)` 。

举例来说：

```
1. var obj = { a: 1, b: 2 };
2.
3. Object.observe( obj, function observer(changes) {
4.     for (var change of changes) {
5.         if (change.type == "setPrototype") {
6.             Object.unobserve(
7.                 change.object, observer
8.             );
9.             break;
10.         }
11.     }
12. } );
```

在这个小例子中，我们监听变化事件直到我们看到 `"setPrototype"` 事件到来，那时我们就不再监听任何变化事件了。

指数操作符

指数操作符

为了使JavaScript以与 `Math.pow(..)` 相同的方式进行指数运算，有一个操作符被提出了。考虑如下代码：

```
var a = 2;

a ** 4;             // Math.pow( a, 4 ) == 16

a **= 3;         // a = Math.pow( a, 3 )
a;                 // 8
```

注意： `**` 实质上在Python、Ruby、Perl、和其他语言中都与此相同。

对象属性与 ...

对象属性与 `...`

正如我们在第二章的“太多，太少，正合适”一节中看到的，`...` 操作符在扩散或收集一个数组上的工作方式是显而易见的。但对象会怎么样？

这样的特性在ES6中被考虑过，但是被推迟到ES6之后（也就是“ES7”或者“ES2016”或者……）了。这是它在“ES6以后”的时代中可能的工作方式：

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

console.log( o3.a, o3.b, o3.c, o3.d );
// 1 2 3 4
```

`...` 操作符也可能被用于将一个对象的被解构属性收集到另一个对象：

```
var o1 = { b: 2, c: 3, d: 4 };
var { b, ...o2 } = o1;

console.log( b, o2.c, o2.d );        // 2 3 4
```

这里，`...o2` 将被解构的 `c` 和 `d` 属性重新收集到一个 `o2` 对象中（与 `o1` 不同，`o2` 没有 `b` 属性）。

重申一下，这些只是正在考虑之中的ES6之后的提案。但是如果它们能被确定下来就太酷了。

Array#includes(..)

`Array#includes(..)`

JS开发者需要执行的极其常见的一个任务就是在一个值的数组中搜索一个值。完成这项任务的方式曾经总是：

```
1. var vals = [ "foo", "bar", 42, "baz" ];
2.
3. if (vals.indexOf( 42 ) >= 0) {
4.     // 找到了！
5. }
```

进行 `>= 0` 检查是因为 `indexOf(..)` 在找到结果时返回一个 `0` 或更大的数字值，或者在没找到结果时返回 `-1` 。换句话说，我们在一个布尔值的上下文环境中使用了一个返回索引的函数。而由于 `-1` 是truthy而非falsy，所以我们不得不手动进行检查。

在本系列的 类型与文法 中，我探索了另一种我稍稍偏好的模式：

```
1. var vals = [ "foo", "bar", 42, "baz" ];
2.
3. if (~vals.indexOf( 42 )) {
4.     // 找到了！
5. }
```

这里的 `~` 操作符使 `indexOf(..)` 的返回值与一个值的范围相一致，这个范围可以恰当地强制转换为布尔型。也就是， `-1` 产生 `0` （falsy），而其余的东西产生非零值（truthy），而这正是我们判定是否找到值的依据。

虽然我觉得这是一种改进，但有另一些人强烈反对。然而，没有人会质疑 `indexOf(..)` 的检索逻辑是完美的。例如，在数组中查找 `NaN` 值会失败。

于是一个提案浮出了水面并得到了大量的支持 —— 增加一个真正的返回布尔值的数组检索方法，称为 `includes(..)` ：

```
1. var vals = [ "foo", "bar", 42, "baz" ];
2.
3. if (vals.includes( 42 )) {
4.     // 找到了！
5. }
```

注意： `Array#includes(..)` 使用了将会找到 `NaN` 值的匹配逻辑，但将不会区分 `-0` 与 `0` （参

见本系列的 类型与文法）。如果你在自己的程序中不关心 `-0` 值，那么它很可能正是你希望的。如果你 确实 关心 `-0` ，那么你就需要实现你自己的检索逻辑，很可能是使用 `Object.is(..)` 工具（见六章）。

SIMD

SIMD

我们在本系列的 异步与性能 中详细讲解了一个指令，多个数据（SIMD），但因为它是未来JS中下一个很可能被确定下来的特性，所以这里简要地提一下。

SIMD API 暴露了各种底层（CPU）指令，它们可以同时操作一个以上的数字值。例如，你可以指定两个拥有4个或8个数字的 向量，然后一次性分别相乘所有元素（数据并行机制！）。

考虑如下代码：

```
var v1 = SIMD.float32x4( 3.14159, 21.0, 32.3, 55.55 );
var v2 = SIMD.float32x4( 2.1, 3.2, 4.3, 5.4 );

SIMD.float32x4.mul( v1, v2 );
// [ 6.597339, 67.2, 138.89, 299.97 ]
```

SIMD将会引入 `mul(..)` （乘法）之外的几种其他操作，比如 `sub()` 、 `div()` 、 `abs()` 、 `neg()` 、 `sqrt()` 、以及其他许多。

并行数学操作对下一代的高性能JS应用程序至关重要。

WebAssembly (WASM)

- WebAssembly (WASM)

WebAssembly (WASM)

在本书的第一版将近完成的时候，Brendan Eich 突然宣布了一个有可能对JavaScript未来的道路产生重大冲击的公告：WebAssembly（WASM）。我们不能在这里详细地探讨WASM，因为在本书写作时这个话题为时过早了。但如果不简要地提上一句，这本书就不够完整。

JS语言在近期（和近未来的）设计的改变上所承受的最大压力之一，就是渴望它能够成为从其他语言（比如 C/C++，ClojureScript，等等）转译/交叉编译来的、合适的目标语言。显然，作为JavaScript运行的代码性能是一个主要问题。

正如在本系列的 异步与性能 中讨论过的，几年前一组在Mozilla的开发者给JavaScript引入了一个称为ASM.js的想法。AMS.js是一个合法JS的子集，它大幅地制约了使代码难于被JS引擎优化的特定行为。其结果就是兼容AMS.js的代码在一个支持ASM的引擎上可以显著地快速运行，几乎可以与优化过的原生C语言的等价物相媲美。许多观点认为，对于那些将要由JavaScript编写的渴求性能的应用程序来说，ASM.js很可能将是它们的基干。

换言之，在浏览器中条条大路通过JavaScript通向运行的代码。

直到WASM公告之前，是这样的。WASM提供了另一条路线，让其他语言不必非得首先通过JavaScript就能将浏览器的运行时环境作为运行的目标。实质上，如果WASM启用，JS引擎将会生长出额外的能力 —— 执行可以被视为有些与字节码相似的二进制代码（就像在JVM上运行的那些东西）。

WASM提出了一种高度压缩的代码AST（语法树）的二进制表示格式，它可以继而像JS引擎以及它的基础结构直接发出指令，无需被JS解析，甚至无需按照JS的规则动作。像C或C++这样的语言可以直接被编译为WASM格式而非ASM.js，并且由于跳过JS解析而得到额外的速度优势。

短期内，WASM与AMS.js、JS不相上下。但是最终，人们预期WASM将会生长出新的能力，那将超过JS能做的任何事情。例如，让JS演化出像线程这样的根本特性 —— 一个肯定会对JS生态系统造成重大冲击的改变 —— 作为一个WASM未来的扩展更有希望，也会缓解改变JS的压力。

事实上，这张新的路线图为许多语言服务于web运行时开启了新的道路。对于web平台来说，这真是一个激动人心的新路线！

它对JS意味着什么？JS将会变得无关紧要或者“死去”吗？绝对不是。ASM.js在接下来的几年中很可能看不到太多未来，但JS在数量上的绝对优势将它安全地锚定在web平台中。

WASM的拥护者们说，它的成功意味着JS的设计将会被保护起来，远离那些最终会迫使它超过自己合理性的临界点的压力。人们估计WASM将会成为应用程序中高性能部分的首选目标语言，这些部分曾用各

种各样不同的语言编写过。

有趣的是，JavaScript是未来不太可能以WASM为目标的语言之一。可能有一些未来的改变会切出JS的一部分，而使这一部分更适于以WASM作为目标，但是这件事情看起来优先级不高。

虽然JS很可能与WASM没什么关联，但JS代码和WASM代码将能够以最重要的方式进行交互，就像当下的模块互动一样自然。你可以想象，调用一个 `foo()` 之类的JS函数而使它实际上调用一个同名WASM函数，它具备远离你其余JS的制约而运行的能力。

至少是在可预见的未来，当下以JS编写的东西可能将继续总是由JS编写。转译为JS的东西将可能最终至少考虑以WASM为目标。对于那些需要极致性能，而且在抽象的层面上没有余地的东西，最有可能的选择是找一种合适的非JS语言编写，然后以WASM为目标语言。

这个转变很有可能将会很慢，会花上许多年成形。WASM在所有的主流浏览器上固定下来可能最快也要花几年。同时，WASM项目(https://github.com/WebAssembly)有一个早期填补，来为它的基本原则展示概念证明。

但随着时间的推移，也随着WASM学到新的非JS技巧，不难想象一些当前是JS的东西被重构为以WASM作为目标的语言。例如，框架中性能敏感的部分，游戏引擎，和其他被深度使用的工具都很可能从这样的转变中获益。在web应用程序中使用这些工具的开发者们并不会在使用或整合上注意到太多不同，但确实会自动地利用这些性能和能力。

可以确定的是，随着WASM变得越来越真实，它对JavaScript设计路线的影响就越来越多。这可能是开发者们应当关注的最重要的“ES6以后”的话题。

复习

复习

如果这个系列的其他书目实质上提出了这个挑战，“你（可能）不懂JS（不像自己想象的那么懂）”，那么这本书就是在说，“你不再懂JS了”。这本书讲解了在ES6中加入到语言里的一大堆新东西。它是一个新语言特性的精彩集合，也是将永远改进我们JS程序的范例。

但JS不是到ES6就完了！还早得很呢。已经有好几个“ES6之后”的特性处于开发的各个阶段。在这一章中，我们简要地看了一些最有可能很快会被固定在JS中的候选特性。

`async function` 是建立在 `generators + promises` 模式（见第四章）上的强大语法糖。 `Object.observe(..)` 为监听对象变化事件增加了直接原生的支持，它对实现数据绑定至关重要。 `**` 指数作符，针对对象属性的 `...` ，以及 `Array#includes(..)` 都是对现存机制的简单而有用的改进。最后，SIMD将高性能JS的演化带入一个新纪元。

听起来很俗套，但JS的未来是非常光明的！这个系列，以及这本书的挑战，现在是各位读者的职责了。你还在等什么？是时候开始学习和探索了！

附录A：鸣谢

- 你不懂JS：ES6 与未来
- 附录A：鸣谢

你不懂JS：ES6 与未来

附录A：鸣谢

为了这本书和整个系列的诞生，我有很多人要感谢。

首先，我必须感谢我的妻子 Christen Simpson，和我的两个孩子 Ethan 和 Emily，忍受着老爹总是在电脑上敲打。即使在没有写书时，我对 JavaScript 的痴迷也将我的眼睛粘在屏幕上太久了。我从家庭那里借来的时间是这些书可以如此深入和完整地向你，读者，解释 JavaScript 的原因。我欠我的家庭一切。

我要感谢我在 O'Reilly 的编辑，他们是 Simon St.Laurent 和 Brian MacDonald，还有其他的编辑和市场员工。和他们一起工作很棒，而且在这种"开源"写作，编辑，和生产的实验期间提供了特别的通融。

感谢许多通过提供编辑意见和订正来参与使这部丛书变得更好的朋友们，他们是 Shelley Powers，Tim Ferro，Evan Borden，Forrest L. Norvell，Jennifer Davis，Jesse Harlin，Kris Kowal，Rick Waldron，Jordan Harband，Benjamin Gruenbaum，Vyacheslav Egorov，David Nolen，和许多其他人。一个巨大感谢送给为本书作序的 Rick Waldron。

感谢社区中无数的朋友们，包括 TC39 协会的成员，他们和我们分享了那么多的知识，特别是以耐心和细节容忍我无休止的问题和探究。John-David Dalton，Juriy "kangax" Zaytsev，Mathias Bynens，Axel Rauschmayer，Nicholas Zakas，Angus Croll，Reginald Braithwaite，Dave Herman，Brendan Eich，Allen Wirfs-Brock，Bradley Meck，Domenic Denicola，David Walsh，Tim Disney，Peter van der Zee，Andrea Giammarchi，Kit Cambridge，Eric Elliott，和其他许多我甚至不能接触到的人。

你不懂*JS* 系列丛书诞生于 Kickstarter，所以我也要感谢我的所有（将近）500位慷慨的支持者，没有他们这部丛书不可能诞生：

> *Jan Szpila, nokiko, Murali Krishnamoorthy, Ryan Joy, Craig Patchett, pdqtrader, Dale Fukami, ray hatfield, R0drigo Perez [Mx], Dan Petitt, Jack Franklin, Andrew Berry, Brian Grinstead, Rob Sutherland, Sergi Meseguer, Phillip Gourley, Mark Watson, Jeff Carouth, Alfredo Sumaran, Martin Sachse, Marcio Barrios, Dan, AimelyneM, Matt Sullivan, Delnatte Pierre-Antoine, Jake Smith, Eugen Tudorancea, Iris, David Trinh, simonstl, Ray Daly, Uros Gruber, Justin Myers, Shai Zonis, Mom & Dad, Devin Clark, Dennis Palmer, Brian Panahi Johnson, Josh Marshall, Marshall, Dennis Kerr, Matt Steele, Erik Slagter, Sacah, Justin Rainbow, Christian Nilsson, Delapouite, D.Pereira, Nicolas*

> *Hoizey, George V. Reilly, Dan Reeves, Bruno Laturner, Chad Jennings, Shane King, Jeremiah Lee Cohick, od3n, Stan Yamane, Marko Vucinic, Jim B, Stephen Collins, Ægir Þorsteinsson, Eric Pederson, Owain, Nathan Smith, Jeanetteurphy, Alexandre ELISÉ, Chris Peterson, Rik Watson, Luke Matthews, Justin Lowery, Morten Nielsen, Vernon Kesner, Chetan Shenoy, Paul Tregoing, Marc Grabanski, Dion Almaer, Andrew Sullivan, Keith Elsass, Tom Burke, Brian Ashenfelter, David Stuart, Karl Swedberg, Graeme, Brandon Hays, John Christopher, Gior, manoj reddy, Chad Smith, Jared Harbour, Minoru TODA, Chris Wigley, Daniel Mee, Mike, Handyface, Alex Jahraus, Carl Furrow, Rob Foulkrod, Max Shishkin, Leigh Penny Jr., Robert Ferguson, Mike van Hoenselaar, Hasse Schougaard, rajan venkataguru, Jeff Adams, Trae Robbins, Rolf Langenhuijzen, Jorge Antunes, Alex Koloskov, Hugh Greenish, Tim Jones, Jose Ochoa, Michael Brennan-White, Naga Harish Muvva, Barkóczi Dávid, Kitt Hodsden, Paul McGraw, Sascha Goldhofer, Andrew Metcalf, Markus Krogh, Michael Mathews, Matt Jared, Juanfran, Georgie Kirschner, Kenny Lee, Ted Zhang, Amit Pahwa, Inbal Sinai, Dan Raine, Schabse Laks, Michael Tervoort, Alexandre Abreu, Alan Joseph Williams, NicolasD, Cindy Wong, Reg Braithwaite, LocalPCGuy, Jon Friskics, Chris Merriman, John Pena, Jacob Katz, Sue Lockwood, Magnus Johansson, Jeremy Crapsey, Grzegorz Pawłowski, nico nuzzaci, Christine Wilks, Hans Bergren, charles montgomery, Ariel בר-לבב Fogel, Ivan Kolev, Daniel Campos, Hugh Wood, Christian Bradford, Frédéric Harper, Ionuț Dan Popa, Jeff Trimble, Rupert Wood, Trey Carrico, Pancho Lopez, Joël kuijten, Tom A Marra, Jeff Jewiss, Jacob Rios, Paolo Di Stefano, Soledad Penades, Chris Gerber, Andrey Dolganov, Wil Moore III, Thomas Martineau, Kareem, Ben Thouret, Udi Nir, Morgan Laupies, jory carson-burson, Nathan L Smith, Eric Damon Walters, Derry Lozano-Hoyland, Geoffrey Wiseman, mkeehner, KatieK, Scott MacFarlane, Brian LaShomb, Adrien Mas, christopher ross, Ian Littman, Dan Atkinson, Elliot Jobe, Nick Dozier, Peter Wooley, John Hoover, dan, Martin A. Jackson, Héctor Fernando Hurtado, andy ennamorato, Paul Seltmann, Melissa Gore, Dave Pollard, Jack Smith, Philip Da Silva, Guy Israeli, @megalithic, Damian Crawford, Felix Gliesche, April Carter Grant, Heidi, jim tierney, Andrea Giammarchi, Nico Vignola, Don Jones, Chris Hartjes, Alex Howes, john gibbon, David J. Groom, BBox, Yu 'Dilys' Sun, Nate Steiner, Brandon Satrom, Brian Wyant, Wesley Hales, Ian Pouncey, Timothy Kevin Oxley, George Terezakis, sanjay raj, Jordan Harband, Marko McLion, Wolfgang Kaufmann, Pascal Peuckert, Dave Nugent, Markus Liebelt, Welling Guzman, Nick Cooley, Daniel Mesquita, Robert Syvarth, Chris Coyier, Rémy Bach, Adam Dougal, Alistair Duggin, David Loidolt, Ed Richer, Brian Chenault, GoldFire Studios, Carles Andrés, Carlos Cabo, Yuya Saito, roberto ricardo, Barnett Klane, Mike Moore, Kevin Marx, Justin Love, Joe Taylor, Paul Dijou, Michael Kohler, Rob Cassie, Mike Tierney, Cody Leroy Lindley, tofuji, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes, Rhys Brett-Bowen, Dmitry, Aziz Khoury, Dean, Scott Tolinski - Level Up, Clement Boirie, Djordje Lukic, Anton Kotenko, Rafael Corral, Philip Hurwitz, Jonathan Pidgeon, Jason Campbell, Joseph C., SwiftOne, Jan Hohner, Derick Bailey, getify, Daniel Cousineau, Chris Charlton, Eric Turner, David Turner, Joël Galeran, Dharma Vagabond, adam, Dirk van Bergen, dave ♥♬★ furf, Vedran Zakanj, Ryan McAllen, Natalie Patrice Tucker, Eric J. Bivona, Adam Spooner, Aaron Cavano, Kelly Packer, Eric J, Martin Drenovac, Emilis, Michael Pelikan, Scott F. Walter, Josh Freeman, Brandon Hudgeons, vijay chennupati, Bill Glennon, Robin R., Troy Forster, otaku_coder, Brad, Scott, Frederick Ostrander, Adam Brill, Seb Flippence, Michael Anderson, Jacob, Adam Randlett, Standard, Joshua Clanton, Sebastian Kouba, Chris Deck, SwordFire, Hannes Papenberg, Richard Woeber, hnzz, Rob Crowther, Jedidiah Broadbent, Sergey Chernyshev, Jay-Ar Jamon, Ben Combee, luciano bonachela, Mark Tomlinson, Kit Cambridge, Michael Melgares, Jacob Adams, Adrian Bruinhout, Bev Wieber, Scott Puleo, Thomas Herzog, April Leone, Daniel Mizieliński, Kees van Ginkel, Jon Abrams, Erwin Heiser, Avi Laviad, David newell, Jean-Francois Turcot, Niko Roberts, Erik Dana, Charles Neill, Aaron Holmes, Grzegorz Ziółkowski, Nathan Youngman, Timothy, Jacob Mather, Michael Allan, Mohit Seth, Ryan Ewing, Benjamin Van Treese, Marcelo Santos, Denis Wolf, Phil Keys, Chris Yung, Timo Tijhof, Martin Lekvall, Agendine, Greg Whitworth, Helen Humphrey, Dougal Campbell, Johannes Harth, Bruno Girin, Brian Hough, Darren Newton, Craig McPheat, Olivier Tille, Dennis Roethig, Mathias Bynens, Brendan Stromberger, sundeep, John Meyer, Ron Male, John F Croston III, gigante, Carl Bergenhem, B.J. May, Rebekah Tyler, Ted Foxberry, Jordan Reese, Terry Suitor, afeliz, Tom Kiefer, Darragh Duffy, Kevin Vanderbeken, Andy Pearson, Simon Mac Donald, Abid Din, Chris Joel, Tomas Theunissen, David Dick, Paul Grock, Brandon Wood, John Weis, dgrebb, Nick Jenkins, Chuck Lane, Johnny Megahan, marzsman, Tatu Tamminen, Geoffrey Knauth, Alexander Tarmolov, Jeremy Tymes, Chad Auld, Sean Parmelee, Rob Staenke, Dan Bender, Yannick derwa, Joshua Jones, Geert Plaisier, Tom LeZotte, Christen Simpson, Stefan Bruvik, Justin Falcone, Carlos Santana, Michael Weiss, Pablo Villoslada, Peter deHaan, Dimitris Iliopoulos, seyDoggy, Adam Jordens, Noah Kantrowitz, Amol M, Matthew*

> *Winnard, Dirk Ginader, Phinam Bui, David Rapson, Andrew Baxter, Florian Bougel, Michael George, Alban Escalier, Daniel Sellers, Sasha Rudan, John Green, Robert Kowalski, David I. Teixeira (@ditma, Charles Carpenter, Justin Yost, Sam S, Denis Ciccale, Kevin Sheurs, Yannick Croissant, Pau Fracés, Stephen McGowan, Shawn Searcy, Chris Ruppel, Kevin Lamping, Jessica Campbell, Christopher Schmitt, Sablons, Jonathan Reisdorf, Bunni Gek, Teddy Huff, Michael Mullany, Michael Fürstenberg, Carl Henderson, Rick Yoesting, Scott Nichols, Hernán Ciudad, Andrew Maier, Mike Stapp, Jesse Shawl, Sérgio Lopes, jsulak, Shawn Price, Joel Clermont, Chris Ridmann, Sean Timm, Jason Finch, Aiden Montgomery, Elijah Manor, Derek Gathright, Jesse Harlin, Dillon Curry, Courtney Myers, Diego Cadenas, Arne de Bree, João Paulo Dubas, James Taylor, Philipp Kraeutli, Mihai Păun, Sam Gharegozlou, joshjs, Matt Murchison, Eric Windham, Timo Behrmann, Andrew Hall, joshua price, Théophile Villard*

这部丛书是以开源的风格书写的，包括编辑和生产。我们感激 GitHub 使这样的事情在社区中成为可能！

再次感谢所有无数的朋友，尽管我不能叫上名字但是我依然亏欠感谢。但愿这部丛书被我们所有人“拥有”，并为增进对 JavaScript 语言的意识和理解做出贡献，成为当下和未来所有社区贡献者的助益。