

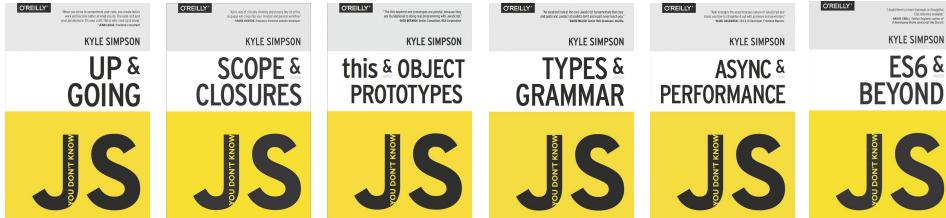
# Table of Contents

Introduction	1.1
前言	1.2
入门与进阶	1.3
序	1.3.1
第一章：进入编程	1.3.2
第二章：进入JavaScript	1.3.3
第三章：进入YDKJS	1.3.4
附录A：鸣谢	1.3.5
作用域与闭包	1.4
第一章：什么是作用域？	1.4.1
第二章：词法作用域	1.4.2
第三章：函数与块儿作用域	1.4.3
第四章：提升	1.4.4
第五章：作用域闭包	1.4.5
附录A：动态作用域	1.4.6
附录B：填补块儿作用域	1.4.7
附录C：词法this	1.4.8
附录D：鸣谢	1.4.9
this与对象原型	1.5
序	1.5.1
第一章: this还是That?	1.5.2
第二章: this豁然开朗！	1.5.3
第三章: 对象	1.5.4
第四章: 混合（淆）“类”的对象	1.5.5
第五章: 原型	1.5.6
第六章: 行为委托	1.5.7
附录A: ES6 class	1.5.8
附录B: 鸣谢	1.5.9
类型与文法	1.6
序	1.6.1

第一章：类型	1.6.2
第二章：值	1.6.3
第三章：原生类型	1.6.4
第四章：强制转换	1.6.5
第五章：文法	1.6.6
附录A：混合环境下的JavaScript	1.6.7
附录B：鸣谢	1.6.8
异步与性能	1.7
序	1.7.1
第一章：异步：现在与稍后	1.7.2
第二章：回调	1.7.3
第三章：Promise	1.7.4
第四章：Generator	1.7.5
第五章：程序性能	1.7.6
第六章：基准分析与调优	1.7.7
附录A：库：asynquence	1.7.8
附录B：高级异步模式	1.7.9
附录C：鸣谢	1.7.10
ES6与未来	1.8
序	1.8.1
第一章：ES？现在与未来	1.8.2
第二章：语法	1.8.3
第三章：组织	1.8.4
第四章：异步流程控制	1.8.5
第五章：集合	1.8.6
第六章：新增API	1.8.7
第七章：元编程	1.8.8
第八章：ES6之后	1.8.9
附录A：鸣谢	1.8.10

# 你不懂JS（系列丛书）

这是一套深入探讨JavaScript语言核心机制的系列丛书。本系列的第一版已经完成。



欢迎通过提交PR改进代码段，讲解等，来为此内容的质量做出贡献。虽然也欢迎拼写错误的修改，但是它们很可能将在一般的编辑过程中被找出来，因此对本代码库不一定很重要。

要了解更多关于这个系列丛书背后的动机和角度，请参阅[前言](#)。

## 书目

- 在线阅读（免费！）：“[入门与进阶](#)”，已出版：[现在购买](#)印刷版，但是ebook格式是免费的！
- 在线阅读（免费！）：“[作用域与闭包](#)”，已出版：[现在购买](#)
- 在线阅读（免费！）：“[this与对象原型](#)”，已出版：[现在购买](#)
- 在线阅读（免费！）：“[类型与文法](#)”，已出版：[现在购买](#)
- 在线阅读（免费！）：“[异步与性能](#)”，已出版：[现在购买](#)
- 在线阅读（免费！）：“[ES6与未来](#)”，已出版：[现在购买](#)

## 出版

这些书在这里作为原稿公布，但也通过O'Reilly出版社编辑，生产，以及出版。

如果你喜欢这里的内容，并且想要支持更多像它一样的内容，请通过你的图书资源，在这些书上市之后购买它们。:)

除了购买这些书以外，如果你想要对本作品做出经济上的贡献，我有一个[patreon](#)。我将永远感激你的慷慨。



## 对面教学

这些书的内容很大程度上衍生自我职业中（公开的和私营企业培训的形式）教授的一系列教学材料，称为“高级JS：‘你需要知道’的部分”。

如果你喜欢这些内容并且想要联系我进行关于这些内容，或其他关于JS/HTML5/node.js话题的培训，请通过这里罗列的渠道联系我：

<http://getify.me>

## 在线视频教学

我还有一些以请求方式发布的视频JS教学资料。我通过[Frontend Masters](#)教授课程，比如我的[高级JS培训班](#)（更多课程陆续更新中！）。

同样的课程还可以[通过 Pluralsight 找到](#)。

## 内容贡献

非常感谢你向本作品做出的任何贡献。

但是在提交PR以前请仔细阅读[内容贡献指引](#)。

## 许可 & 版权

The materials herein are all (c) 2013-2016 Kyle Simpson.



This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivs 4.0 Unported License](#).

# 你不懂JS

## 前言

我确信你注意到了，但是这个系列图书标题中的“JS”不是一个用来诅咒JavaScript的缩写，虽然有时我们可能都能看出它是在诅咒这门语言的怪异之处！

自从web的最早期开始，JavaScript就一直是在我们消费内容时驱动互动体验的基础技术。虽然闪烁的鼠标轨迹和烦人的弹出框可能是JavaScript的起源，但是在差不多20年以后，JavaScript的技术和能力已经增长了许多个数量级，在世界上最广泛应用的软件平台——web——的核心上，很少有人怀疑它的重要性。

但作为一种语言，它一直总是都是一大堆批评的目标，部分是因为它的遗留问题，但更大程度上是因为它的设计思想。正如 Brendan Eich 曾经说过的，就连名字都让人联想它是更成熟的老大哥“Java”的“笨小弟”。但是这个名字只不过是政治上和市场上的巧合。这两种语言在读多方面有着广泛的不同。“JavaScript”与“Java”的联系，就和“嘉年华（Carnival）”和“车（Car）”一样。

因为JavaScript从几种语言中借用了各种概念和语法惯例，包括高傲的C语言风格的过程式性质，以及微妙的，不那么明显的Scheme/Lisp语言风格的函数式性质，所以它对于广大的开发者用户来说极其容易上手。JavaScript的“Hello World”是如此简单，以至于这门语言在早期接触时就很吸引人而且很容易让人适应。

虽然JavaScript也许是最容易上手和使用的语言之一，但与其他的语言相比，它的古怪之处使得牢固掌握这门语言是一个不常见的现象。像C或C++这样的语言需要相当深度的知识才能写出完整的程序，而完整的JavaScript程序可以，而且通常都是，仅仅触及了这门语言的皮毛。

精巧的概念植根于这门语言的深处，而不是简单地像看起来那样浮于表面，比如将函数作为回调传递，它鼓励JavaScript开发者就那么直接使用这门语言而不必过于担心底层发生了什么。

它是一种具有广泛吸引力的，简单、易用的语言，同时也是一个复杂和微妙的语言机制集合，即使是对于经验丰富的JavaScript开发者来说，不仔细研究就无法真正懂得。

这就是JavaScript的矛盾之处，这门语言的致命弱点，我们当下面临的挑战。因为JavaScript可以不搞懂就使用，所以这门语言经常是从没被搞懂过。

## 使命

如果每次你在JavaScript中遭遇惊诧和挫折时，你的反应都是像某些人习惯的那样将它加入黑名单，那么你很快就会将JavaScript的丰富内涵掏成一个空壳。

虽然这部分子集曾经被称为著名的“好的部分”，但我恳求你，亲爱的读者，把它看作是“简单的部分”，“安全的部分”，甚至是“不完整的部分”。

这套 *你不懂JavaScript* 系列丛书提出了一个相反的挑战：学习并深刻理解JavaScript的全部，甚至是而且特别是“艰难的部分”。

这里，我们迎面挑战这种趋势：JS开发者总是得过且过地学习“将就够用”的东西，而从来不强迫他们自己去学习这门语言究竟是为何与如何工作的。另外，我们摒弃那些当路途艰难时常见的逃跑意见。

我不满足于，你也不应当满足于，一看到某些东西好用就停下来，而不真正知道为什么。我礼貌地挑战你沿着这条颠簸的“少有人走的路”走下去，并拥抱JavaScript和它能做的一起。有了这些知识，没有什么技术，框架，本周最流行的词儿，是你理解不了的。

这些书的每一本都承担这门语言的一个具体的核心部分——这些内容都是最常被误解或不了解的，并且非常深入、穷尽地探究它们。你应当在读过之后对你的理解有坚定的信心，不仅仅是理论上的，而且是对实际的“你需要懂得的”部分。

你目前知道的JavaScript可能是被那些没有完全懂得它的人传授下来的部分。那些JavaScript只是真正的这门语言的影子。你还不真正懂得JavaScript，但如果你深入这个系列，你会的。读下去，我的朋友。JavaScript在等着你。

## 总结

JavaScript很了不起。很容易学习它的一部分，但是完整地（甚至是充分地）学习它可就难太多了。当开发者们遭遇困惑时，他们经常责备这门语言而不是自己对它理解的匮乏。这些书的目的就是要修正这些情况，极大地巩固这门语言中你现在可以，而且应当，深刻懂得的东西。

注意：这本书中的许多例子都假定运行在现代化的（以及与未来接轨的）JavaScript引擎环境中，比如ES6。一些代码可能会在老版本的（前ES6）引擎中不能如描述的那样工作。

# 你不懂JS：入门与进阶

## 目录

- 序
- 前言
- 第一章：进入编程
  - 代码
  - 亲自尝试
  - 操作符
  - 值与类型
  - 代码注释
  - 变量
  - 块儿
  - 条件
  - 循环
  - 函数
  - 练习
- 第二章：进入JavaScript
  - 值与类型
  - 变量
  - 条件
  - Strict模式
  - 函数作为值
  - `this` 标识符
  - 原型
  - 旧的与新的
  - 非JavaScript
- 第三章：进入YDKJS
  - 作用域与闭包
  - `this`与对象原型
  - 类型与文法
  - 异步与性能
  - ES6与未来
- 附录A：鸣谢



# 你不懂JS：入门与进阶

## 序

你学的最后一个新东西是什么？

也许是一门外语，比如意大利语或德语。或者可能是一种图像编辑器，比如 Photoshop。或者是一种烹饪技术，木工活，日常锻炼。我想让你回忆一下你最终学会它时的感觉：醍醐灌顶的时刻。当事情从模糊不清变得豁然开朗，正如你掌握了如何使用台锯，或者理解了法语中雄性名词和雌性名词的区别。那种感觉怎么样？非常美妙，对吧？

现在我想让你再多向前回忆一些，找到你学会新技能之前的那一刻。它感觉如何？可能有点儿吓人，也可能有点儿沮丧，是吧？在某个时刻，我们都还不知道我们现在知道的事情，而这完全没问题；我们是从某处开始的。学习新的东西是一次激动人心的冒险，特别是当你想高效地学习它时。

我教授过许多面向初学者的编程课程。上我课的学生们经常试着通过阅读博客或者拷贝粘贴代码来自学HTML或JavaScript这样的东西，但是他们都没能真正掌握能够使他们编写出自己渴望的结果的技能。而且，因为他们没有真正把握关于编程的特定问题的内在和外在，他们不能编写强大的代码或调试自己的程序，因为他们没有真正地理解发生的事情。

我总是相信教授我的课程的正确方法，意味着我教授Web标准，语义标记，良好注释的代码，和其他的最佳实践。我使用一种彻底的方式讲解问题来阐明如何做与为何做，而非通过复制粘贴来倒腾代码。当你努力理解你的代码时，你就在创造更好的成果，并在编程上变得更加纯熟。代码不再仅仅是你的工作，而是你的作品。这就是为什么我喜爱入门与进阶。

Kyle通过深入讲解语法和术语给我们带来了一个对JavaScript的全面介绍。这本书不是浅尝辄止，而是让我们真正地理解我们将要编写的东西。

能够在你的网站中复制JQuery代码段是不够的，就像在Photoshop中仅仅学习如何打开，关闭和保存一个文档是不够的一样。确实，只要我学会了一些关于编程的基本我就可以制造并分享一些我的设计。但是没有合理地了解这些工具和它们背后的机制，我又如何定义一个网格，或者建造一个合理的类型系统，或者为Web优化图像呢？JavaScript也一样。不知道循环如何工作，或者如何定义变量，或者作用域是什么，我们将不能写出最好的代码。我们不想安于这种次优的状态——这毕竟是我们的作品。

你对JavaScript探索得越多，它就变得越清晰。闭包，对象，和方法这样的词现在可能看起来与你还有些距离，但是这本书将会帮你搞清楚这些术语。我希望你在开始阅读这本书时保持学会东西之前与之后的那两种感觉。它看起来可能有些令人望而却步，但是你已经拿起了这本书，你开启了一个了不起的旅程来磨练自己的知识。入门与进阶是我们理解编程之路的开端。享受醍醐灌顶的时刻吧！

Jenn Lukas

[jennlukas.com](http://jennlukas.com), @jennlukas

前端顾问

# 你不懂JS：入门与进阶

## 第一章：进入编程

欢迎来到 你不懂JS（YDKJS）系列。

入门与进阶 是一个对几种编程基本概念的介绍 —— 当然我们是特别倾向于JavaScript（经常略称为JS）的 —— 以及如何看待与理解本系列的其他书目。特别是如果你刚刚接触编程和/或JavaScript，这本书将简要地探索你需要什么来 入门与进阶。

这本书从很高的角度来解释编程的基本原则开始。它基本上假定你是在没有或很少的编程经验的情况下开始阅读 YDKJS 的，而且你期待这些书可以透过JavaScript的镜头帮助你开启一条理解编程的道路。

第一章应当作为一个快速的概览来阅读，它讲述为了 进入编程 你将想要多加学习和实践的东西。有许多其他精彩的编程介绍资源可以帮你在这个话题上走得更远，而且我鼓励你学习它们来作为这一章的补充。

一旦你对一般的编程基础感到适应了，第二章将指引你熟悉JavaScript风格的编程。第二章介绍了JavaScript是什么，但是同样的，它不是一个全面的指引 —— 那是其他 YDKJS 书目的任务！

如果你已经相当熟悉JavaScript，那么就首先看一下第三章作为 YDKJS 内容的简要一瞥，然后一头扎进去吧！

## 代码

让我们从头开始。

一个程序，经常被称为 源代码 或者只是 代码，是一组告诉计算机要执行什么任务的特殊指令。代码通常保存在文本文件中，虽然你也可以使用JavaScript在一个浏览器的开发者控制台中直接键入代码 —— 我们一会儿就会讲解。

合法的格式与指令的组合规则被称为一种 计算机语言，有时被称作它的 语法，这和英语教你如何拼写单词，和如何使用单词与标点创建合法的句子差不多是相同的。

## 语句

在一门计算机语言中，一组单词，数字，和执行一种具体任务的操作符构成了一个 语句。在 JavaScript 中，一个语句可能看起来像下面这样：

```
a = b * 2;
```

字符 `a` 和 `b` 被称为 **变量**（参见“**变量**”），它们就像简单盒子，你可以把任何东西存储在其中。在程序中，变量持有将被程序使用的值（比如数字 `42`）。可以认为它们就是值本身的标志占位符。

相比之下，`2` 本身只是一个值，称为一个 **字面值**，因为它没有被存入一个变量，是独立的。

字符 `=` 和 `*` 是 **操作符**（见“**操作符**”）——它们使用值和变量实施动作，比如赋值和数学乘法。

在JavaScript中大多数语句都以末尾的分号（`;`）结束。

语句 `a = b * 2;` 告诉计算机，大致上，去取得当前存储在变量 `b` 中的值，将这个值乘以 `2`，然后将结果存回到另一个我们称为 `a` 变量里面。

程序只是许多这样的语句的集合，它们一起描述为了执行你的程序的意图所要采取的所有步骤。

## 表达式

语句是由一个或多个 **表达式** 组成的。一个表达式是一个引用，指向变量或值，或者一组用操作符组合的变量和值。

例如：

```
a = b * 2;
```

这个语句中有四个表达式：

- `2` 是一个 **字面量表达式**
- `b` 是一个 **变量表达式**，它意味着取出它的当前值
- `b * 2` 是一个 **算数表达式**，它意味着执行乘法
- `a = b * 2` 是一个 **赋值表达式**，它意味着将表达式 `b * 2` 的结果赋值给变量 `a`（稍后有更多关于赋值的内容）

一个独立的普通表达式也被称为一个 **表达式语句**，比如下面的：

```
b * 2;
```

这种风格的表达式语句不是很常见也没什么用，因为一般来说它不会对程序的运行有任何影响——它将取得 `b` 的值并乘以 `2`，但是之后不会对结果做任何事情。

一种更常见的表达式语句是调用表达式语句（见“函数”），因为整个语句本身是一个函数调用表达式：

```
alert( a );
```

## 执行一个程序

这些程序语句的集合如何告诉计算机要做什么？这个程序需要被执行，也称为运行这个程序。

在开发者们阅读与编写时，像 `a = b * 2` 这样的语句很有帮助，但是它实际上不是计算机可以直接受理解的形式。所以一个计算机上的特殊工具（不是一个解释器就是一个编译器）被用于将你编写的代码翻译为计算机可以理解的命令。

对于某些计算机语言，这种命令的翻译经常是在每次程序运行时从上向下，一行接一行完成的，这通常成为代码的解释。

对于另一些语言，这种翻译是提前完成的，成为代码的编译，所以当程序稍后运行时，实际上运行的东西已经是编译好，随时可以运行的计算机指令了。

JavaScript通常被断言为是解释型的，因为你的JavaScript源代码在它每次运行时都被处理。但这并不是完全准确的。JavaScript引擎实际上在即时地编译程序然后立即运行编译好的代码。

注意：更多关于JavaScript编译的信息，参见本系列的作用域与闭包的前两章。

## 亲自尝试

这一章将用简单的代码段来介绍每一个编程概念，它们都是用JavaScript写的（当然！）。

有一件事情怎么强调都不过分：在你通读本章时——而且你可能需要花时间读好几遍——你应当通过自己编写代码来实践这些概念中的每一个。最简单的方法就是打开你手边的浏览器（Firefox，Chrome，IE，等等）的开发者工具控制台。

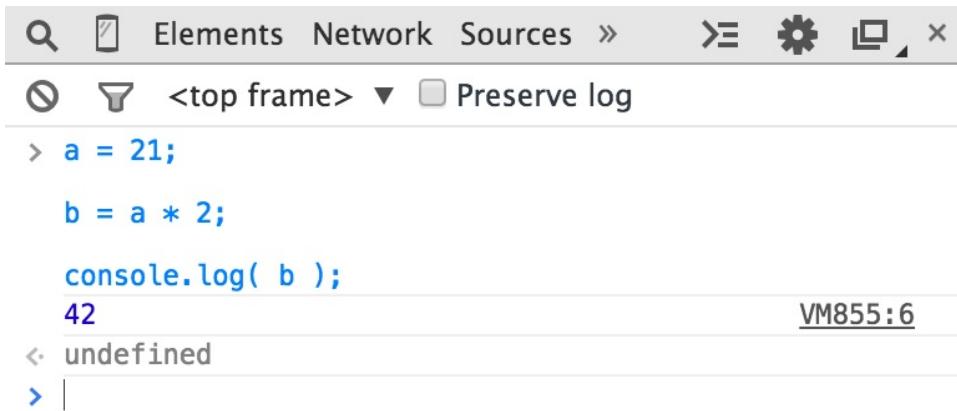
提示：一般来说，你可以使用快捷键或者菜单选项来启动开发者控制台。更多关于启动和使用你最喜欢的浏览器的控制台的细节，参见“精通开发者工具控制台”(<http://blog.teamtreehouse.com/mastering-developer-tools-console>)。要在控制台中一次键入多行，可以使用`+``来移动到下一行。一旦你敲击```，控制台将运行你刚刚键入的任何东西。

让我们熟悉一下在控制台中运行代码的过程。首先，我建议你在浏览器中打开一个新的标签页。我喜欢在地址栏中键入`about:blank`来这么做。然后，确认你的开发者控制台是打开的，就像我们刚刚提到的那样。

现在，键入如下代码看看它是怎么运行的：

```
a = 21;
b = a * 2;
console.log( b );
```

在Chrome的控制台中键入前面的代码应该会产生如下的东西：



The screenshot shows the Chrome DevTools interface with the 'Elements' tab selected. In the bottom-left corner of the main pane, there is a small terminal-like window displaying the following code and its execution results:

```
> a = 21;
b = a * 2;
console.log( b );
42
<- undefined
> |
```

To the right of the terminal, the file path "VM855:6" is displayed.

继续，试试吧。学习编程的最佳方式就是开始编码！

## 输出

在前一个代码段中，我们使用了 `console.log(..)`。让我们简单地看看这一行代码在做什么。

你也许已经猜到了，它正是我们如何在开发者控制台中打印文本（也就是向用户输出）的方法。这个语句有两个性质，我们应当解释一下。

首先，`log(b)` 部分被称为一个函数调用（见“函数”）。这里发生的事情是，我们将变量 `b` 交给这个函数，它向变量 `b` 要来它的值，并在控制台中打印。

第二，`console.` 部分是一个对象引用，这个对象就是找到 `log(..)` 函数的地方。我们会在第二章中详细讲解对象和它们的属性。

另一种创建你可以看到的输出的方式是运行 `alert(..)` 语句。例如：

```
alert( b );
```

如果你运行它，你会注意到它不会打印输出到控制台，而是显示一个内容为变量 `b` 的“OK”弹出框。但是，一般来说与使用 `alert(..)` 相比，使用 `console.log(..)` 会使学习编码和在控制台运行你的程序更简单一些，因为你可以一次输出许多值，而不必干扰浏览器的界面。

在这本书中，我们将使用 `console.log(..)` 来输出。

## 输入

虽然我们在讨论输出，你也许还想知道输入（例如，从用户那里获得信息）。

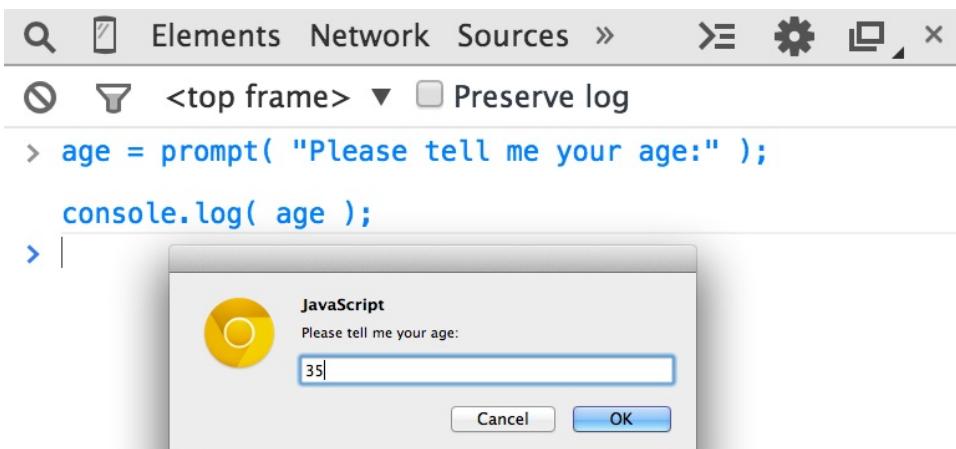
对于HTML网页来说，输入发生的最常见的方式是向用户显示一个他们可以键入的form元素，然后使用JS将这些值读入你程序的变量中。

但是为了单纯的学习和展示的目的——也就是你在这本书中将通篇看到的——有一个获取输入的更简单的方法。使用 `prompt(..)` 函数：

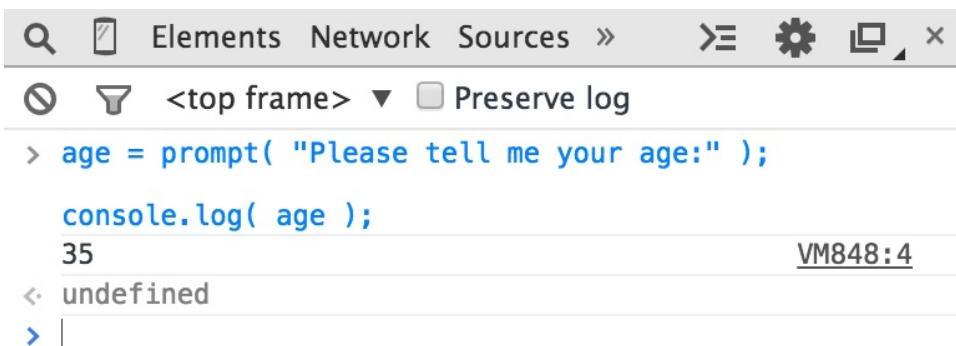
```
age = prompt( "Please tell me your age:" );
console.log( age );
```

正如你可能已经猜到的，你传递给 `prompt(..)` 的消息——在这个例子中，“Please tell me your age:”——被打印在弹出框中。

它应当和下面的东西很相似：



一旦你点击“OK”提交输入的文本，你将会看到你输入的值被存储在变量 `age` 中，然后我们使用 `console.log(..)` 把它输出：



为了让我们在学习基本编程概念时使事情保持简单，本书中的例子不要求输入。但是现在你已经看到了如何使用 `prompt(..)`，如果你想挑战一下自己，你可以试着在探索这些例子时使用输入。

# 操作符

操作符是我们如何在变量和值上实施操作的方式。我们已经见到了两种JavaScript操作符，`=` 和 `*`。

`*` 操作符实施数学乘法。够简单的，对吧？

`=` 操作符用于赋值——我们首先计算 `=` 右手边的值（源值）然后将它放进我们在左手边指定的变量中（目标变量）。

警告：对于指定赋值，这看起来像是一种奇怪的倒置。与 `a = 42` 不同，一些人喜欢把顺序反转过来，于是源值在左而目标变量在右，就像 `42 -> a`（这不是合法的JavaScript！）。不幸的是，`a = 42` 顺序的形式，和与其相似的变种，在现代编程语言中是十分流行的。如果它让你觉得不自然，那么就花些时间在脑中演练这个顺序并习惯它。

考虑如下代码：

```
a = 2;
b = a + 1;
```

这里，我们将值 `2` 赋值给变量 `a`。然后，我们取得变量 `a` 的值（还是 `2`），把它加 `1` 得到值 `3`，然后将这个值存储到变量 `b` 中。

虽然在技术上说 `var` 不是一个操作符，但是你将在每一个程序中都需要这个关键字，因为它是你声明（也就是创建）变量（见“变量”）的主要方式。

你应当总是在使用变量前用名称声明它。但是对于每个作用域（见“作用域”）你只需要声明变量一次；它可以根据需要使用任意多次。例如：

```
var a = 20;

a = a + 1;
a = a * 2;

console.log( a ); // 42
```

这里是一些在JavaScript中最常见的操作符：

- 赋值：比如 `a = 2` 中的 `=`。
- 数学：`+`（加法），`-`（减法），`*`（乘法），和 `/`（除法），比如 `a * 3`。
- 复合赋值：`+=`，`-=`，`*=`，和 `/=` 都是复合操作符，它们组合了数学操作和赋值，比如 `a += 2`（与 `a = a + 2` 相同）。
- 递增/递减：`++`（递增），`--`（递减），比如 `a++`（和 `a = a + 1` 很相似）。
- 对象属性访问：比如 `console.log()` 的 `.`。

对象是一种值，它可以在被称为属性的，被具体命名的位置上持有其他的值。`obj.a` 意味着一个称为 `obj` 的对象值有一个名为 `a` 的属性。属性可以用 `obj["a"]` 这种替代的方式访问。参见第二章。

- 等价性：`==`（宽松等价），`===`（严格等价），`!=`（宽松不等价），`!==`（严格不等价），比如 `a == b`。

参见“值与类型”和第二章。

- 比较：`<`（小于），`>`（大于），`<=`（小于或宽松等价），`>=`（大于或宽松等价），比如 `a <= b`。

参见“值与类型”和第二章。

- 逻辑：`&&`（与），`||`（或），比如 `a || b` 它选择 `a` 或 `b` 中的一个。

这些操作符用于表达复合的条件（见“条件”），比如如果 `a` 或者 `b` 成立。

注意：更多细节，以及在此没有提到的其他操作符，可以参见Mozilla开发者网络（MDN）的“表达式与操作符”([https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions\\_and\\_Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators))。

## 值与类型

如果你问一个手机店的店员一种特定手机的价格，而他们说“九十九块九毛九”（即，\$99.99），他们给了你一个实际的美元数字来表示你需要花多少钱才能买到它。如果你想两部这种手机，你可以很容易地心算这个值的两倍来得到你需要花费的\$199.98。

如果同一个店员拿起另一部相似的手机说它是“免费的”（也许在用手比划引号），那么他们就不是在给你一个数字，而是你的花费（\$0.00）的另一种表达形式——“免费”这个词。

当你稍后问到这个手机是否带充电器时，回答可能仅仅是“是”或者“不”。

以同样的方式，当你在程序中表达一个值时，你根据你打算对这些值做什么来选择不同的表达形式。

在编程术语中值的这些不同的表达形式称为 **类型**。JavaScript中对这些所谓的 **基本类型** 值都有内建的类型：

- 但你需要做数学计算时，你需要一个 `number`。
- 当你需要在屏幕上打印一个值时，你需要一个 `string`（一个或多个字符，单词，句子）。
- 当你需要在你的程序中做决定时，你需要一个 `boolean`（`true` 或 `false`）。

在源代码中直接包含的值称为 `字面量`。`string` 字面量被双引号 `"..."` 或单引号 `('...')` 包围——唯一的区别是风格上的偏好。`number` 和 `boolean` 字面量用它们本身来表示（即，`42`，`true`，等等）。

考虑如下代码：

```
"I am a string";
'I am also a string';

42;

true;
false;
```

在 `string / number / boolean` 值的类型以外，编程语言通常会提供 数组，对象，函数 等更多的类型。我们会在本章和下一章中讲解更多关于值和类型的内容。

## 类型间转换

如果你有一个 `number` 但需要将它打印在屏幕上，那么你就需要将这个值转换为一个 `string`，在JavaScript中这种转换称为“强制转换”。类似地，如果某些人在一个电商网页的 `form` 中输入一系列数字，那么它是一个 `string`，但是如果你需要使用这个值去做数学运算，那么你就需要将它 强制转换 为一个 `number`。

为了在 类型 之间强制转换，JavaScript提供了几种不同的工具。例如：

```
var a = "42";
var b = Number( a );

console.log( a );    // "42"
console.log( b );    // 42
```

使用上面展示的 `Number(..)`（一个内建函数）是一种从任意其他类型到 `number` 类型的 明确 的 强制转换。这应当是相当直白的。

但是一个具有争议的话题是，当你试着比较两个还不是相同类型的值时发生的事情，它需要 隐含 的 强制转换。

当比较字符串 `"99.99"` 和数字 `99.99` 时，大多数人同意它们是等价的。但是他们不完全相同，不是吗？它们是相同的值的两种不同表现形式，两个不同的 类型。你可以说它们是“宽松地等价”的，不是吗？

为了在这些常见情况下帮助你，JavaScript有时会启动 隐含 的 强制转换来把值转换为匹配的 类型。

所以如果你使用 `==` 宽松等价操作符来进行 `"99.99" == 99.99` 比较，JavaScript 会将左手边的 `"99.99"` 转换为它的 `number` 等价物 `99.99`。所以比较就变成了 `99.99 == 99.99`，这当然是成立的。

虽然隐含强制转换是为了帮助你而设计，但是它也可能把你搞糊涂，如果你没有花时间去学习控制它行为的规则。大多数开发者从没有这么做，所以常见的感觉是隐含的强制转换是令人困惑的，并且会产生意外的 bug 危害程序，因此应当避免使用。有时它甚至被称为这种语言中的设计缺陷。

然而，隐含强制转换是一种可以被学习的机制，而且是一种应当被所有想要认真对待 JavaScript 编程的人学习的机制。一旦你学习了这些规则，它不仅是消除了困惑，而且它实际上是你自己的程序变得更好！这种努力是值得的。

注意：关于强制转换的更多信息，参见本书第二章和本系列类型与文法的第四章。

## 代码注释

手机店店员可能会写下一些笔记，记下新出的手机的特性或者他们公司推出的新套餐。这些笔记仅仅是给店员使用的——他们不是给顾客读的。不管怎样，通过记录下为什么和如何告诉顾客他应当说的东西，这些笔记帮助店员更好的工作。

关于编写代码你要学的最重要的课程之一，就是它不仅仅是写给计算机的。代码的每一个字节都和写给编译器一样，也是写给开发者的。

你的计算机只关心机器码，一系列源自编译的 0 和 1。你几乎可以写出无限多种可以产生相同 0 和 1 序列的代码。所以你对如何编写程序作出的决定很重要——不仅是对你，也对你的团队中的其他成员，甚至是未来的自己。

你不仅应当努力去编写可以正确工作的程序，而且应当努力编写检视起来有道理的程序。你可以通过给变量（见“变量”）和函数（见“函数”）起一个好名字在这条路上走很远。

但另外一个重要的部分是代码注释。它们纯粹是为了向人类解释一些事情而在你的程序中插入的一点儿文本。解释器/编译器将总是忽略这些注释。

关于什么是良好注释的代码有许多意见；我们不能真正地定义绝对统一的规则。但是一些意见和指导是十分有用的：

- 没有注释的代码是次优的。
- 过多的注释（比如，每行都有注释）可能是代码编写的很烂的标志。
- 注释应当解释为什么，而不是是什么。它们可以选择性地解释如何做，如果代码特别令人困惑的话。

在 JavaScript 中，有两种可能的注释类型：单行注释和多行注释

考虑如下代码：

```
// 这是一个单行注释

/* 而这是
   一个多行
   注释。
   */


```

如果你想在一个语句的正上方，或者甚至是在行的末尾加一个注释，`//` 单行注释是很合适的。这一行上`//`之后的所有东西都将被视为注释（因此被编译器忽略），一直到行的末尾。在单行注释内部可以出现的内容没有限制。

考虑：

```
var a = 42;           // 生命的意义是 42
```

如果你想在注释中用好几行来解释一些事情，`/* .. */` 多行注释就很合适。

这是多行注释的一个常见用法：

```
/* 使用下面的值是因为
   它回答了
   全宇宙中所有的问题。 */
var a = 42;
```

它还可以出现在一行中的任意位置，甚至是一行的中间，因为`*/` 终结了它。例如：

```
var a = /* 随机值 */ 42;

console.log( a );    // 42
```

在多行注释中唯一不能出现的就是`*/`，因为这将干扰注释的结尾。

你绝对会希望通过养成注释代码的习惯来开始学习编程。在本书剩余的部分中，你将看到我使用注释来解释事情，请也在你自己的实践中这么做。相信我，所有阅读你的代码的人都会感谢你！

## 变量

大多数有用的程序都需要在程序运行整个过程中，追踪由于你的程序所意图的任务被调用的底层不同的操作而发生的值的变化。

要这样做的最简单的方法是将一个值赋予一个符号容器，称为一个 **变量** —— 因为在这个容器中的值可以根据需要不时 变化 而得名。

在某些编程语言中，你可以声明一个变量（容器）来持有特定类型的值，比如 `number` 或 `string`。因为防止了意外的类型转换，静态类型，也被称为 类型强制，通常被认为是对程序正确性有好处的。

另一些语言在值上强调类型而非在变量上。弱类型，也被称为 动态类型，允许变量在任意时刻持有任意类型的值。因为它允许一个变量在程序逻辑流程中代表一个值，而不论这个值在任意给定的时刻是什么类型，所以它被认为是对程序灵活性有好处的。

JavaScript使用的是后者，动态类型，这意味着变量可以持有任意类型的值而没有任何类型强制约束。

正如我们刚才提到的，我们使用 `var` 语句来声明一个变量——注意在这种声明中没有其他的类型信息。考虑这段简单的代码：

```
var amount = 99.99;

amount = amount * 2;

console.log( amount );           // 199.98

// 将 `amount` 转换为一个字符串，
// 并在开头加一个 "$"
amount = "$" + String( amount );

console.log( amount );           // "$199.98"
```

变量 `amount` 开始时持有数字 `99.99`，然后持有 `amount * 2` 的 `number` 结果，也就是 `199.98`。

第一个 `console.log(..)` 命令不得不隐含地将这个 `number` 值强制转换为一个 `string` 才能够打印出来。

然后语句 `amount = "$" + String(amount)` 明确地将值 `199.98` 强制转换为一个 `string` 并且在开头加入一个 `"$"` 字符。这时，`amount` 现在就持有这个 `string` 值 `$199.98`，所以第二个 `console.log(..)` 语句无需强制转换就可以把它打印出来。

JavaScript开发者将会注意到为值 `99.99`，`199.98`，和 `"$199.98"` 都使用变量 `amount` 的灵活性。静态类型的拥护者们将偏好于使用一个分离的变量，比如 `amountStr` 来持有这个值最后的 `"$199.98"` 表达形式，因为它是一个不同的类型。

不管哪种方式，你将会注意到 `amount` 持有一个在程序运行过程中不断变化的值，这展示了变量的主要目地：管理程序状态。

换句话说，在你程序运行的过程中状态追踪着值的改变。

变量的另一种常见用法是将值的设定集中化。当你为一个在程序中通篇不打算改变的值声明了一个变量时，它更一般地被称为 常量。

你经常会在程序的顶部声明这些常量，这样提供了一种方便：如果你需要改变一个值时你可以到唯一的地方去寻找。根据惯例，用做常量的JavaScript变量通常是大写的，在多个单词之间使用下划线 \_ 连接。

这里是一个呆萌的例子：

```
var TAX_RATE = 0.08;      // 8% sales tax

var amount = 99.99;

amount = amount * 2;

amount = amount + (amount * TAX_RATE);

console.log( amount );           // 215.9784
console.log( amount.toFixed( 2 ) ); // "215.98"
```

注意：`console.log(..)` 是一个函数 `log(..)` 作为一个在值 `console` 上的对象属性被访问，与此类似，这里的 `toFixed(..)` 是一个可以在值 `number` 上被访问的函数。JavaScript `number` 不会被自动地格式化为美元——引擎不知道你的意图，而且也没有通货类型。`toFixed(..)` 让我们指明四舍五入到小数点后多少位，而且它如我们需要的那样产生一个 `string`。

变量 `TAX_RATE` 只是因为惯例才是一个常量——在这个程序中没有什么特殊的东西可以防止它被改变。但是如果这座城市将它的消费税增至9%，我们仍然可以很容易地通过在一个地方将 `TAX_RATE` 被赋予的值改为 `0.09` 来更新我们的程序，而不是在程序通篇中寻找许多值 `0.08` 出现的地方然后更新它们全部。

在写作本书时，最新版本的JavaScript（通常称为“ES6”）引入了一个声明常量的新方法，用 `const` 代替 `var`：

```
// 在ES6中：
const TAX_RATE = 0.08;

var amount = 99.99;

// ..
```

常量就像带有不变的值的变量一样有用，常量还防止在初始设置之后的某些地方意外地改变它的值。如果你试着在第一个声明之后给 `TAX_RATE` 赋予一个不同的值，你的程序将会拒绝这个改变（而且在Strict模式下，会产生一个错误——见第二章的“Strict模式”）。

顺带一提，这种防止编程错误的“保护”与静态类型的类型强制很类似，所以你可以看到为什么在其他语言中的静态类型很吸引人。

注意：更多关于如何在你程序的变量中使用不同的值，参见本系列的 [类型与文法](#)。

## 块儿

在你买你的新手机时，手机店店员必须走过一系列步骤才能完成结算。

相似地，在代码中我们经常需要将一系列语句一起分为一组，这就是我们常说的 块儿。在 JavaScript 中，一个块儿被定义为包围在一个大括号 `{ .. }` 中的一个或多个语句。考虑如下代码：

```
var amount = 99.99;

// 一个普通的块儿
{
    amount = amount * 2;
    console.log( amount );      // 199.98
}
```

这种独立的 `{ .. }` 块儿是合法的，但是在 JS 程序中并不常见。一般来说，块儿是添附在一些其他的控制语句后面的，比如一个 `if` 语句（见“条件”）或者一个循环（见“循环”）。例如：

```
var amount = 99.99;

// 数值够大吗？
if (amount > 10) {           // <-- 添附在`if`上的块儿
    amount = amount * 2;
    console.log( amount );    // 199.98
}
```

我们将在下一节讲解 `if` 语句，但是如你所见，`{ .. }` 块儿带着它的两个语句被添附在 `if (amount > 10)` 后面；块儿中的语句将会仅在条件成立时被处理。

注意：与其他大多数语句不同（比如 `console.log(amount);`），一个块儿语句与不需要分号（`;`）来终结它。

## 条件

“你想来一个额外的屏幕贴膜吗？只要 \$9.99。”热心的手机店店员请你做个决定。而你也许需要首先咨询一下钱包或银行帐号的状态才能回答这个问题。但很明显，这只是一个简单的“是与否”的问题。

在我们的程序中有好几种方式可以表达 条件（也就是决定）。

最常见的一个就是 `if` 语句。实质上，你在说，“如果 这个条件成立，做后面的……”。例如：

```

var bank_balance = 302.13;
var amount = 99.99;

if (amount < bank_balance) {
    console.log( "I want to buy this phone!" );
}

```

`if` 语句在括号 ( ) 之间需要一个表达式，它不是被视作 `true` 就是被视作 `false`。在这个程序中，我们提供了表达式 `amount < bank_balance`，它确实会根据变量 `bank_balance` 中的值被求值为 `true` 或 `false`。

如果条件不成立，你甚至可以提供一个另外的选择，称为 `else` 子句。考虑下面的代码：

```

const ACCESSORY_PRICE = 9.99;

var bank_balance = 302.13;
var amount = 99.99;

amount = amount * 2;

// 我们买得起配件吗？
if ( amount < bank_balance ) {
    console.log( "I'll take the accessory!" );
    amount = amount + ACCESSORY_PRICE;
}
// 否则：
else {
    console.log( "No, thanks." );
}

```

在这里，如果 `amount < bank_balance` 是 `true`，我们将打印出 `"I'll take the accessory!"` 并在我们的变量 `amount` 上加 `9.99`。否则，`else` 子句说我们将礼貌地回应 `"No, thanks."`，并保持 `amount` 不变。

正如我们在早先的“值与类型”中讨论的，一个还不是所期望类型的值经常会被强制转换为那种类型。`if` 语句期待一个 `boolean`，但如果你传给它某些还不是 `boolean` 的东西，强制转换就会发生。

JavaScript 定义了一组特定的被认为是“`falsy`”的值，因为在强制转换为 `boolean` 时，它们将变为 `false` —— 这些值包括 `0` 和 `""`。任何不再这个 `falsy` 列表中的值都自动是“`truthy`”—— 当强制转换为 `boolean` 时它们变为 `true`。`truthy` 值包括 `99.99` 和 `"free"` 这样的东西。更多信息参见第二章的“`Truthy`与`Falsy`”。

除了 `if` 条件还以其他形式存在。例如，`switch` 语句可以被用作一系列 `if..else` 语句的缩写（见第二章）。循环（见“循环”）使用一个条件来决定循环是否应当继续或停止。

注意：关于在条件的测试表达式中可能发生的隐含强制转换的更深层的信息，参见本系列的类型与文法的第四章。

## 循环

在繁忙的时候，有一张排队单，上面记载着需要和手机店店员谈话的顾客。虽然排队单上还有许多人，但是她只需要持续服务下一位顾客就好了。

重复一组动作直到特定的条件失败——换句话说，仅在条件成立时重复——就是程序循环的工作；循环可以有不同的形式，但是它们都符合这种基本行为。

一个循环包含测试条件和一个块儿（通常是`{ ... }`）。每次循环块儿执行，都称为一次迭代。

例如，`while` 循环和 `do..while` 循环形式就说明了这种概念——重复一块儿语句直到一个条件不再求值得 `true`：

```
while (numOfCustomers > 0) {
    console.log( "How may I help you?" );
    // 服务顾客......

    numOfCustomers = numOfCustomers - 1;
}

// 与

do {
    console.log( "How may I help you?" );
    // 服务顾客......

    numOfCustomers = numOfCustomers - 1;
} while (numOfCustomers > 0);
```

这些循环之间唯一的实际区别是，条件是在第一次迭代之前（`while`）还是之后（`do..while`）被测试。

在这两种形式中，如果条件测试得 `false`，那么下一次迭代就不会运行。这意味着如果条件初始时就是 `false`，那么 `while` 循环就永远不会运行，但是一个 `do..while` 循环将仅运行一次。

有时你会为了计数一组特定的数字来进行循环，比如从 `0` 到 `9`（十个数）。你可以通过设定一个值为 `0` 的循环迭代变量，比如 `i`，并在每次迭代时将它递增 `1`。

警告：由于种种历史原因，编程语言几乎总是用从零开始的方式来计数的，这意味着计数开始于 `0` 而不是 `1`。如果你不熟悉这种思维模式，一开始它可能十分令人困惑。为了更适应它，花些时间练习从 `0` 开始数数吧！

条件在每次迭代时都会被测试，好像在循环内部有一个隐含的 `if` 语句一样。

你可以使用JavaScript的 `break` 语句来停止一个循环。另外，我们可以看到如果没有 `break` 机制，就会极其容易地创造一个永远运行的循环。

让我们展示一下：

```
var i = 0;

// 一个 `while..true` 循环将会永远运行，对吧？
while (true) {
    // 停止循环？
    if ((i <= 9) === false) {
        break;
    }

    console.log( i );
    i = i + 1;
}

// 0 1 2 3 4 5 6 7 8 9
```

警告：这未必是你想在你的循环中使用的实际形式。它是仅为了说明的目的才出现在这里的。

虽然一个 `while`（或 `do..while`）可以手动完成任务，但是为了同样的目的，还有一种称为 `for` 循环的语法形式：

```
for (var i = 0; i <= 9; i = i + 1) {
    console.log( i );
}

// 0 1 2 3 4 5 6 7 8 9
```

如你所见，对于这两种循环形式来说，前10次迭代（`i` 的值从 `0` 到 `9`）的条件 `i <= 9` 都是 `true`，而且一旦 `i` 值为 `10` 就变为 `false`。

`for` 循环有三个子句：初始化子句（`var i=0`），条件测试子句（`i <= 9`），和更新子句（`i = i + 1`）。所以如果你想要使用循环迭代来计数，`for` 是一个更紧凑而且更易理解和编写的形式。

还有一些意在迭代特定的值的特殊循环形式，比如迭代一个对象的属性（见第二章），它隐含的测试条件是所有的属性是否都被处理过了。无论循环是何种形式，“循环直到条件失败”的概念是它们共有的。

## 函数

手机店的店员可能不会拿着一个计算器到处走，用它来搞清税费和最终的购物款。这是一个她需要定义一次然后一遍又一遍地重用的任务。很有可能的是，公司有一个带有内建这些“功能”的收银机（电脑，平板电脑，等等）。

相似地，几乎可以肯定你的程序想要将代码的任务分割成可以重用的片段，而不是频繁地多次重复自己。这么做的方法是定义一个 `function`。

一个函数一般来说是一段被命名的代码，它可以使用名称来被“调用”，而每次调用它内部的代码就会运行。考虑如下代码：

```
function printAmount() {
    console.log( amount.toFixed( 2 ) );
}

var amount = 99.99;

printAmount(); // "99.99"

amount = amount * 2;

printAmount(); // "199.98"
```

函数可以选择性地接收参数值（也就是参数）——你传入的值。而且它们还可以选择性地返回一个值。

```
function printAmount(amount) {
    console.log( amount.toFixed( 2 ) );
}

function formatAmount() {
    return "$" + amount.toFixed( 2 );
}

var amount = 99.99;

printAmount( amount * 2 );           // "199.98"

amount = formatAmount();
console.log( amount );            // "$99.99"
```

函数 `printAmount(..)` 接收一个参数，我们称之为 `amt`。函数 `formatAmount()` 返回一个值。当然，你也可以在同一个函数中组合这两种技术。

函数经常被用于你打算多次调用的代码，但它们对于仅将有关联的代码组织在一个命名的集合中也很有用，即便你只打算调用它们一次。

考虑如下代码：

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // 计算带有税费的新费用
    amt = amt + (amt * TAX_RATE);

    // 返回新费用
    return amt;
}

var amount = 99.99;

amount = calculateFinalPurchaseAmount( amount );

console.log( amount.toFixed( 2 ) );           // "107.99"
```

虽然 `calculateFinalPurchaseAmount(..)` 只被调用了一次，但是将它的行为组织进一个分离的带名称的函数，让使用它逻辑的代码（`amount = calculateFinal...` 语句）更干净。如果函数中拥有更多的语句，这种好处将会更加明显。

## 作用域

如果你向手机店的店员询问一款她们店里没有的手机，那么她就不能卖给你你想要的。她只能访问她们店库房里的手机。你不得不到另外一家店里去看看能不能找到你想要的手机。

编程对这种概念有一个术语：作用域（技术上讲称为 词法作用域）。在 JavaScript 中，每个函数都有自己的作用域。作用域基本上就是变量的集合，也是如何使用名称访问这些变量的规则。只有在这个函数内部的代码才能访问这个函数作用域内的变量。

在同一个作用域内变量名必须是唯一的——不能有两个不同的变量 `a` 并排出现。但是相同的变量名 `a` 可以出现在不同的作用域中。

```

function one() {
    // 这个 `a` 仅属于函数 `one()`
    var a = 1;
    console.log( a );
}

function two() {
    // 这个 `a` 仅属于函数 `two()`
    var a = 2;
    console.log( a );
}

one();      // 1
two();      // 2

```

另外，一个作用域可以嵌套在另一个作用域中，就像生日Party上的小丑在一个气球的里面吹另一个气球一样。如果一个作用域嵌套在另一个中，那么在内部作用域中的代码就可以访问这两个作用域中的变量。

考虑如下代码：

```

function outer() {
    var a = 1;

    function inner() {
        var b = 2;

        // 我们可以在这里同时访问 `a` 和 `b`
        console.log( a + b );    // 3
    }

    inner();

    // 我们在这里只能访问 `a`
    console.log( a );        // 1
}

outer();

```

词法作用域规则说，在一个作用域中的代码既可以访问这个作用域中的变量，又可以访问任何在它外面的作用域的变量。

所以，在函数 `inner()` 内部的代码可以同时访问变量 `a` 和 `b`，但是仅在 `outer()` 中的代码只能访问 `a` —— 它不能访问 `b` 因为这个变量仅存在于 `inner()` 内部。

回忆一下先前的这个代码段：

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // 计算带有税费的新费用
    amt = amt + (amt * TAX_RATE);

    // 返回新费用
    return amt;
}
```

因为词法作用域，常数 `TAX_RATE`（变量）可以从 `calculateFinalPurchaseAmount(...)` 函数中访问，即便它没有被传入这个函数。

注意：关于词法作用域的更多信息，参见本系列的作用域与闭包的前三章。

## 练习

在编程的学习中绝对没有什么可以替代练习。我写的再好也不可能使你成为一个程序员。

带着这样的意识，让我们试着练习一下我们在本章学到的一些概念。我将给出“需求”，而你首先试着实现它。然后参考下面的代码清单来看看我是怎么处理它的。

- 写一个程序来计算你购买手机的总价。你将不停地购买手机直到你的银行账户上的钱都用光（提示：循环！）。你还将为每个手机购买配件，只要你的花费低于你心理预算。
- 在你计算完购买总价之后，加入税费，然后用合适的格式打印出计算好的购买总价。
- 最后，将总价与你银行账户上的余额作比较，来看看那你是买的起。
- 你应当为“税率”，“手机价格”，“配件价格”和“花费预算”设置一些常数，也为你的“银行账户余额”设置一个变量。
- 你应当为税费的计算和价格的格式化——使用一个“\$”并四舍五入到小数点后两位——定义函数。
- 加分挑战：试着在这个程序中利用输入，也许是使用在前面的“输入”中讲过的 `prompt(..)`。比如，你可能会提示用户输入它们的银行账户余额。发挥创造力好好玩儿吧！

好的，去吧。试试看。在你自己实践过之前不要偷看我的代码清单！

注意：因为这是一本JavaScript书，很明显我将使用JavaScript解决这个联系。但是目前你可以使用其他的语言，如果你感觉更适应的话。

对于这个练习，这是我的JavaScript解决方案：

```
const SPENDING_THRESHOLD = 200;
const TAX_RATE = 0.08;
const PHONE_PRICE = 99.99;
const ACCESSORY_PRICE = 9.99;

var bank_balance = 303.91;
var amount = 0;

function calculateTax(amount) {
    return amount * TAX_RATE;
}

function formatAmount(amount) {
    return "$" + amount.toFixed( 2 );
}

// 只要你还有钱就不停地买手机
while (amount < bank_balance) {
    // 买个新手机
    amount = amount + PHONE_PRICE;

    // 还买得起配件吗？
    if (amount < SPENDING_THRESHOLD) {
        amount = amount + ACCESSORY_PRICE;
    }
}

// 也别忘了给政府交钱
amount = amount + calculateTax( amount );

console.log(
    "Your purchase: " + formatAmount( amount )
);
// Your purchase: $334.76

// 你买的起吗？
if (amount > bank_balance) {
    console.log(
        "You can't afford this purchase. :("
    );
}
// 你买不起 :(
```

注意：运行这个JavaScript程序的最简单的方法是将它键入到你手边的浏览器的开发者控制台中。

你做的怎么样？看了我的代码之后，现在再试一次也没什么不好。而且你可以改变某些常数来看看使用不同的值时这个程序运行的如何。

## 复习

学习编程不一定是个复杂而且巨大的过程。你只需要在脑中装进几个基本的概念。

它们就像构建块儿。要建一座高塔，你就要从堆砌构建块儿开始。编程也一样。这里是一些编程中必不可少的构建块儿：

- 你需要 操作符 来在值上实施动作。
- 你需要 值 和 类型 来试试不同种类的动作，比如在 `number` 上做数学，或者使用 `string` 输出。
- 你需要 变量 在你程序执行的过程中存储数据（也就是 状态）。
- 你需要 条件，比如 `if` 语句来做决定。
- 你需要 循环 来重复任务，直到一个条件不再成立。
- 你需要 函数 来将你的代码组织为有逻辑的和可复用的块儿。

代码注释是一种编写更好可读性代码的有效方法，它使你的代码更易理解，维护，而且如果稍后出现问题的话更易修改。

最后，不要忽视练习的力量。学习写代码的最好方法就是写代码。

现在，我很高兴看到你在学习编码的道路上走得很好！保持下去。不要忘了看看其他编程初学者的资源（书，博客，在线教学，等等）。这一章和这本书是一个很好的开始，但它们只是一个简要的介绍。

下一章将会复习许多本章中的概念，但是是从更加专门于JavaScript的视角，这将突出将在本系列的剩余部分将要深度剖析的大多数主要话题。

# 你不懂JS：入门与进阶

## 第二章：进入JavaScript

在前一章中，我介绍了编程的基本构建块儿，比如变量，循环，条件，和函数。当然，所有被展示的代码都是JavaScript。但是在这一章中，为了作为一个JS开发者入门和进阶，我们想要特别集中于那些你需要知道的关于JavaScript的事情。

我们将在本章中介绍好几个概念，它们将会在后续的 *YDKJS* 丛书中全面地探索。你可以将这一章看作是这个系列的其他书目中将要详细讲解的话题的一个概览。

特别是如果你刚刚接触JavaScript，那么你应当希望花相当一段时间来多次复习这里的概念和代码示例。任何好的基础都是一砖一瓦积累起来的，所以不要指望你会在第一遍通读后就立即理解了全部内容。

你深入学习JavaScript的旅途从这里开始。

注意：正如我在第一章中说过的，在你通读这一章的同时，你绝对应该亲自尝试这里所有的代码。要注意的是，这里的有些代码假定最新版本的JavaScript（通常称为“ES6”，ECMAScript的第六个版本——ECMAScript是JS语言规范的官方名称）中引入的功能是存在的。如果你碰巧在使用一个老版本的，前ES6时代的浏览器，这些代码可能不好用。应当使用一个更新版本的现代浏览器（比如Chrome，Firefox，或者IE）。

## 值与类型

正如我们在第一章中宣称的，JavaScript拥有带类型的值，没有带类型的变量。下面是可用的内建类型：

- `string`
- `number`
- `boolean`
- `null` 和 `undefined`
- `object`
- `symbol` (ES6新增类型)

JavaScript提供了一个 `typeof` 操作符，它可以检查一个值并告诉你它的类型是什么：

```

var a;
typeof a; // "undefined"

a = "hello world";
typeof a; // "string"

a = 42;
typeof a; // "number"

a = true;
typeof a; // "boolean"

a = null;
typeof a; // "object" -- 奇怪的bug

a = undefined;
typeof a; // "undefined"

a = { b: "c" };
typeof a; // "object"

```

来自 `typeof` 的返回值总是六个（ES6中是七个！——“symbol”类型）字符串值之一。也就是说，`typeof "abc"` 返回 `"string"`，不是 `string`。

注意在这个代码段中变量 `a` 是如何持有每种不同类型的值的，而且尽管表面上看起来很像，但是 `typeof a` 并不是在询问“`a` 的类型”，而是“当前 `a` 中的值的类型”。在JavaScript中只有值拥有类型；变量只是这些值的简单容器。

`typeof null` 是一个有趣的例子，因为当你期望它返回 `"null"` 时，它错误地返回了 `"object"`。

警告：这是JS中一直存在的一个bug，但是看起来它永远都不会被修复了。在网络上有太多的代码依存于这个bug，因此修复它将会导致更多的bug！

另外，注意 `a = undefined`。我们明确地将 `a` 设置为值 `undefined`，但是在行为上这与一个还没有被设定值的变量没有区别，比如在这个代码段顶部的 `var a;`。一个变量可以用好几种不同的方式得到这样的“`undefined`”值状态，包括没有返回值的函数和使用 `void` 操作符。

## 对象

`object` 类型指的是一种复合值，你可以在它上面设定属性（带名称的位置），每个属性持有各自的任意类型的值。它也许是JavaScript中最有用的数据类型之一。

```

var obj = {
  a: "hello world",
  b: 42,
  c: true
};

obj.a;          // "hello world"
obj.b;          // 42
obj.c;          // true

obj["a"];       // "hello world"
obj["b"];       // 42
obj["c"];       // true

```

可视化地考虑这个 `obj` 值可能会有所帮助：

`obj`

a: "hello world"	b: 42	c: true
---------------------	----------	------------

属性既可以使用 点号标记法（例如，`obj.a`）访问，也可以使用 方括号标记法（例如，`obj["a"]`）访问。点号标记法更短而且一般来说更易于阅读，因此在可能的情况下它都是首选。

如果你有一个名称中含有特殊字符的属性名称，方括号标记法就很有用，比如 `obj["hello world!"]` —— 当通过方括号标记法访问时，这样的属性经常被称为 键。[ ] 标记法要求一个变量（下一节讲解）或者一个 `string` 字面量（它需要包装进 `"..."` 或 `'...'`）。

当然，如果你想访问一个属性/键，但是它的名称被存储在另一个变量中时，方括号标记法也很有用。例如：

```

var obj = {
  a: "hello world",
  b: 42
};

var b = "a";

obj[b];          // "hello world"
obj["b"];        // 42

```

注意：更多关于JavaScript的 `object` 的信息，请参见本系列的 `this`与对象原型，特别是第三章。

在JavaScript程序中有另外两种你将会经常打交道的值类型：数组 和 函数。但与其说它们是内建类型，这些类型应当被认为更像是子类型 —— `object` 类型的特化版本。

## 数组

一个数组是一个 `object`，它不使用特殊的带名称的属性/键持有（任意类型的）值，而是使用数字索引的位置。例如：

```
var arr = [
  "hello world",
  42,
  true
];

arr[0];           // "hello world"
arr[1];           // 42
arr[2];           // true
arr.length;        // 3

typeof arr;        // "object"
```

注意：从零开始计数的语言，比如JS，在数组中使用 `0` 作为第一个元素的索引。

可视化地考虑 `arr` 很能会有所帮助：

`arr`

0: "hello world"	1: 42	2: true
------------------	-------	---------

因为数组是一种特殊的对象（正如 `typeof` 所暗示的），所以它们可以拥有属性，包括一个可以自动被更新的 `length` 属性。

理论上你可以使用你自己的命名属性将一个数组用作一个普通对象，或者你可以使用一个 `object` 但是给它类似于数组的数字属性（`0`，`1`，等等）。然而，这么做一般被认为是分别误用了这两种类型。

最好且最自然的方法是为数字定位的值使用数组，而为命名属性使用 `object`。

## 函数

另一个你将在JS程序中到处使用的 `object` 子类型是函数：

```

function foo() {
    return 42;
}

foo.bar = "hello world";

typeof foo;           // "function"
typeof foo();         // "number"
typeof foo.bar;       // "string"

```

同样地，函数也是 `object` 的子类型——`typeof` 返回 `"function"`，这暗示着 `"function"` 是一种主要类型——因此也可以拥有属性，但是你一般仅会在有限情况下才使用函数对象属性（比如 `foo.bar`）。

注意：更多关于JS的值和它们的类型的信息，参见本系列的 [类型与文法](#) 的前两章。

## 内建类型的方法

我们刚刚讨论的内建类型和子类型拥有十分强大和有用的行为，它们作为属性和方法暴露出来。

例如：

```

var a = "hello world";
var b = 3.14159;

a.length;           // 11
a.toUpperCase();    // "HELLO WORLD"
b.toFixed(4);       // "3.1416"

```

使调用 `a.toUpperCase()` 成为可能的原因，要比这个值上存在这个方法的说法复杂一些。

简而言之，有一个 `String` (`s` 大写) 对象包装器形式，通常被称为“原生类型”，与 `string` 基本类型配成一对儿；正是这个对象包装器的原型上定义了 `toUpperCase()` 方法。

当你通过引用一个属性或方法（例如，前一个代码段中的 `a.toUpperCase()`）将一个像 `"hello world"` 这样的基本类型值当做一个 `object` 来使用时，JS自动地将这个值“封箱”为它对应的对象包装器（这个操作是隐藏在幕后的）。

一个 `string` 值可以被包装为一个 `String` 对象，一个 `number` 可以被包装为一个 `Number` 对象，而一个 `boolean` 可以被包装为一个 `Boolean` 对象。在大多数情况下，你不担心或者直接使用这些值的对象包装器形式——在所有实际情况中首选基本类型值形式，而JavaScript会帮你搞定剩下的一切。

注意：关于JS原生类型和“封箱”的更多信息，参见本系列的 [类型与文法](#) 的第三章。要更好地理解对象原型，参见本系列的 `this` 与对象原型的第五章。

## 值的比较

在你的JS程序中你将需要进行两种主要的值的比较：等价 和 不等价。任何比较的结果都是严格的 boolean 值（`true` 或 `false`），无论被比较的值的类型是什么。

## 强制转换

在第一章中我们简单地谈了一下强制转换，我们在此回顾它。

在JavaScript中强制转换有两种形式：明确的 和 隐含的。明确的强制转换比较简单，因为你在代码中明显地看到一个类型转换到另一个类型将会发生，而隐含的强制转换更像是另外一些操作的不明显的副作用引发的类型转换。

你可能听到过像“强制转换是邪恶的”这样情绪化的观点，这是因为一个清楚的事实——强制转换在某些地方会产生一些令人吃惊的结果。也许没有什么能比当一个语言吓到开发者时更能唤起他们的沮丧心情了。

强制转换并不邪恶，它也不一定是令人吃惊的。事实上，你使用类型强制转换构建的绝大部分情况是十分合理和可理解的，而且它甚至可以用来增强你代码的可读性。但我们不会在这个话题上过度深入——本系列的 类型与文法 的第四章将会进行全面讲解。

这是一个 明确 强制转换的例子：

```
var a = "42";
var b = Number(a);
a;           // "42"
b;           // 42 -- 数字！
```

而这是一个 隐含 强制转换的例子：

```
var a = "42";
var b = a * 1;    // 这里 "42" 被隐含地强制转换为 42
a;           // "42"
b;           // 42 -- 数字！
```

## Truthy 与 Falsy

在第一章中，我们简要地提到了值的“truthy”和“falsy”性质：当一个非 boolean 值被强制转换为一个 boolean 时，它是变成 `true` 还是 `false`。

在JavaScript中“falsy”的明确列表如下：

- `""` (空字符串)
- `0`, `-0`, `Nan` (非法的 `number`)
- `null`, `undefined`
- `false`

任何不在这个“`falsy`”列表中的值都是“`truthy`”。这是其中的一些例子：

- `"hello"`
- `42`
- `true`
- `[ ]`, `[ 1, "2", 3 ]` (数组)
- `{ }`, `{ a: 42 }` (对象)
- `function foo() { .. }` (函数)

重要的是要记住，一个非 `boolean` 值仅在实际上被强制转换为一个 `boolean` 时才遵循这个“`truthy`”/“`falsy`”强制转换。把你搞糊涂并不困难——当一个场景看起来像是将一个值强制转换为 `boolean`，可其实它不是。

## 等价性

有四种等价性操作符：`==`，`===`，`!=`，和 `!==`。`!` 形式当然是与它们相对应操作符平行的“不等”版本；不等 (*non-equality*) 不应当与 不等价性 (*inequality*) 相混淆。

`==` 和 `===` 之间的不同通常被描述为，`==` 检查值的等价性而 `===` 检查值和类型两者的等价性。然而，这是不准确的。描述它们的合理方式是，`==` 在允许强制转换的条件下检查值的等价性，而 `===` 是在不允许强制转换的条件下检查值的等价性；因此 `===` 常被称为“严格等价”。

考虑这个隐含强制转换，它在 `==` 宽松等价性比较中允许，而 `===` 严格等价性比较中不允许：

```
var a = "42";
var b = 42;

a == b;           // true
a === b;          // false
```

在 `a == b` 的比较中，JS注意到类型不匹配，于是它经过一系列有顺序的步骤将一个值或者它们两者强制转换为一个不同的类型，直到类型匹配为止，然后就可以检查一个简单的值等价性。

如果你仔细想一想，通过强制转换 `a == b` 可以有两种方式给出 `true`。这个比较要么最终成为 `42 == 42`，要么成为 `"42" == "42"`。那么是哪一种呢？

答案：`"42"` 变成 `42`，于是比较成为 `42 == 42`。在一个这样简单的例子中，只要最终结果是一样的，处理的过程走哪一条路看起来并不重要。但在一些更复杂的情况下，这不仅对比较的最终结果很重要，而且对你如何得到这个结果也很重要。

`a === b` 产生 `false`，因为强制转换是不允许的，所以简单值的比较很明显将会失败。许多开发者感觉 `==` 更可靠，所以他们提倡一直使用这种形式而远离 `===`。我认为这种观点是非常短视的。我相信 `==` 是一种可以改进程序的强大工具，如果你花时间去学习它的工作方式。

我们不会详细地讲解强制转换在 `==` 比较中是如何工作的。它的大部分都是相当合理的，但是有一些重要的极端用例要小心。你可以阅读ES5语言规范的11.9.3部分（<http://www.ecma-international.org/ecma-262/5.1/>）来了解确切的规则，而且与围绕这种机制的所有负面炒作比起来，你会对这它是多么的直白而感到吃惊。

为了将这许多细节归纳为一个简单的包装，并帮助你在各种情况下判断是否使用 `==` 或 `===`，这是我的简单规则：

- 如果一个比较的两个值之一可能是 `true` 或 `false` 值，避免 `==` 而使用 `===`。
- 如果一个比较的两个值之一可能是这些具体的值（`0`，`""`，或 `[]` —— 空数组），避免 `==` 而使用 `===`。
- 在所有其他情况下，你使用 `==` 是安全的。它不仅安全，而且在许多情况下它可以简化你的代码并改善可读性。

这些规则归纳出来的东西要求你严谨地考虑你的代码：什么样的值可能通过这个被比较等价性的变量。如果你可以确定这些值，那么 `==` 就是安全的，使用它！如果你不能确定这些值，就使用 `===`。就这么简单。

`!=` 不等价形式对应于 `==`，而 `!==` 形式对应于 `===`。我们刚刚讨论的所有规则和注意点对这些非等价比较都是平行适用的。

如果你在比较两个非基本类型值，比如 `object`（包括 `function` 和 `array`），那么你应当特别小心 `==` 和 `===` 的比较规则。因为这些值实际上是通过引用持有的，`==` 和 `===` 比较都将简单地检查这个引用是否相同，而不是它们底层的值。

例如，`array` 默认情况下会通过使用逗号（`,`）连接所有值来被强制转换为 `string`。你可能认为两个内容相同的 `array` 将是 `==` 相等的，但它们不是：

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

a == c;      // true
b == c;      // true
a == b;      // false
```

注意：更多关于 `==` 等价性比较规则的信息，参见ES5语言规范（11.9.3部分），和本系列的类型与文法的第四章；更多关于值和引用的信息，参见它的第二章。

## 不等价性

`<`，`>`，`<=`，和`>=`操作符用于不等价性比较，在语言规范中被称为“关系比较”。一般来说它们将与`number`这样的可比较有序值一起使用。`3 < 4`是很容易理解的。

但是JavaScript`string`值也可进行不等价性比较，它使用典型的字母顺序规则（`"bar" < "foo"`）。

那么强制转换呢？与`==`比较相似的规则（虽然不是完全相同！）也适用于不等价操作符。要注意的是，没有像`===`严格等价操作符那样不允许强制转换的“严格不等价”操作符。

考虑如下代码：

```
var a = 41;
var b = "42";
var c = "43";

a < b;           // true
b < c;           // true
```

这里发生了什么？在ES5语言规范的11.8.5部分中，它说如果`<`比较的两个值都是`string`，就像`b < c`，那么这个比较将会以字典顺序（也就是像字典中字母的排列顺序）进行。但如果两个值之一不是`string`，就像`a < b`，那么两个值就将被强制转换成`number`，并进行一般的数字比较。

在可能不同类型的值之间进行比较时，你可能遇到的最大的坑——记住，没有“严格不等价”可用——是其中一个值不能转换为合法的数字，例如：

```
var a = 42;
var b = "foo";

a < b;           // false
a > b;           // false
a == b;          // false
```

等一下，这三个比较怎么可能都是`false`？因为在`<`和`>`的比较中，值`b`被强制转换为了“非法的数字值”，而且语言规范说`Nan`既不大于其他值，也不小于其他值。

`==`比较失败于不同的原因。如果`a == b`被解释为`42 == NaN`或者`"42" == "foo"`都会失败——正如我们前面讲过的，这里是前一种情况。

注意：关于不等价比较规则的更多信息，参见ES5语言规范的11.8.5部分，和本系列的类型与文法第四章。

## 变量

在JavaScript中，变量名（包括函数名）必须是合法的标识符（*identifiers*）。当你考虑非传统意义上的字符时，比如Unicode，标识符中合法字符的严格和完整的规则就有点儿复杂。如果你仅考虑典型的ASCII字母数字的字符，那么这个规则还是很简单的。

一个标识符必须以 `a - z`，`A - Z`，`$`，或 `_` 开头。它可以包含任意这些字符外加数字 `0 - 9`。

一般来说，变量标识符的规则也通用适用于属性名称。然而，有一些不能用作变量名，但是可以用作属性名的单词。这些单词被称为“保留字（*reserved words*）”，包括JS关键字（`for`，`in`，`if`，等等）和 `null`，`true` 和 `false`。

注意：更多关于保留字的信息，参见本系列的类型与文法的附录A。

## 函数作用域

你使用 `var` 关键字声明的变量将属于当前的函数作用域，如果声明位于任何函数外部的顶层，它就属于全局作用域。

## 提升

无论 `var` 出现在一个作用域内部的何处，这个声明都被认为是属于整个作用域，而且在作用域的所有位置都是可以访问的。

这种行为称为 提升，比喻一个 `var` 声明在概念上 被移动 到了包含它的作用域的顶端。技术上讲，这个过程通过代码的编译方式进行解释更准确，但是我们先暂且跳过那些细节。

考虑如下代码：

```

var a = 2;

foo();           // 可以工作，因为 `foo()` 声明被“提升”了

function foo() {
  a = 3;

  console.log(a);    // 3

  var a;            // 声明被“提升”到了 `foo()` 的顶端
}

console.log(a);  // 2

```

**警告：**在一个作用域中依靠变量提升来在 `var` 声明出现之前使用一个变量是不常见的，也不是个好主意；它可能相当使人困惑。而使用被提升的函数声明要常见得多，也更为人所接受，就像我们在 `foo()` 正式声明之前就调用它一样。

## 嵌套的作用域

当你声明了一个变量时，它就在这个作用域内的任何地方都是可用的，包括任何下层/内部作用域。例如：

```

function foo() {
  var a = 1;

  function bar() {
    var b = 2;

    function baz() {
      var c = 3;

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

    baz();
    console.log(a, b);        // 1 2
  }

  bar();
  console.log(a);            // 1
}

foo();

```

注意 `c` 在 `bar()` 的内部是不可用的，因为它是仅在内部的 `baz()` 作用域中被声明的，并且 `b` 因为同样的原因在 `foo()` 内是不可用的。

如果你试着在一个作用域内访问一个不可用的变量的值，你就会得到一个被抛出的 `ReferenceError`。如果你试着为一个还没有被声明的变量赋值，那么根据“`strict`模式”的状态，你会要么得到一个在顶层全局作用域中创建的变量（不好！），要么得到一个错误。让我们看一下：

```
function foo() {
    a = 1;      // `a` 没有被正式声明
}

foo();
a;           // 1 -- 噢，自动全局变量 :(
```

这是一种非常差劲儿的做法。别这么干！总是给你的变量进行正式声明。

除了在函数级别为变量创建声明，ES6允许你使用 `let` 关键字声明属于个别块儿（一个 `{ .. }`）的变量。除了一些微妙的细节，作用域规则将大致上与我们刚刚看到的函数相同：

```
function foo() {
    var a = 1;

    if (a >= 1) {
        let b = 2;

        while (b < 5) {
            let c = b * 2;
            b++;

            console.log( a + c );
        }
    }
}

foo();
// 5 7 9
```

因为使用了 `let` 而非 `var`，`b` 将仅属于 `if` 语句而不是整个 `foo()` 函数的作用域。相似地，`c` 仅属于 `while` 循环。对于以更加细粒度的方式管理你的变量作用域来说，块儿作用域是非常有用的，它将使你的代码随着时间的推移更加易于维护。

注意：关于作用域的更多信息，参见本系列的作用域与闭包。更多关于 `let` 块儿作用域的信息，参见本系列的 `ES6与未来`。

## 条件

除了我们在第一章中简要介绍过的 `if` 语句，JavaScript还提供了几种其他值得我们一看的条件机制。

有时你可能发现自己在像这样写一系列的 `if..else..if` 语句：

```
if (a == 2) {  
    // 做一些事情  
}  
else if (a == 10) {  
    // 做另一些事情  
}  
else if (a == 42) {  
    // 又是另外一些事情  
}  
else {  
    // 这里是备用方案  
}
```

这种结构好用，但有一点儿繁冗，因为你需要为每一种情况都指明 `a` 的测试。这里有另一种选项，`switch` 语句：

```
switch (a) {  
    case 2:  
        // 做一些事情  
        break;  
    case 10:  
        // 做另一些事情  
        break;  
    case 42:  
        // 又是另外一些事情  
        break;  
    default:  
        // 这里是备用方案  
}
```

如果你想仅让一个 `case` 中的语句运行，`break` 是很重要的。如果你在一个 `case` 中省略了 `break`，并且这个 `case` 成立或运行，那么程序的执行将会不管下一个 `case` 语句是否成立而继续执行它。这种所谓的“掉落”有时是有用/期望的：

```

switch (a) {
  case 2:
  case 10:
    // 一些很酷的事情
    break;
  case 42:
    // 另一些事情
    break;
  default:
    // 备用方案
}

```

这里，如果 `a` 是 `2` 或 `10`，它就会执行“一些很酷的事情”的代码语句。

在JavaScript中的另一种条件形式是“条件操作符”，经常被称为“三元操作符”。它像是一个单独的 `if..else` 语句的更简洁的形式，比如：

```

var a = 42;

var b = (a > 41) ? "hello" : "world";

// 与此相似：

// if (a > 41) {
//   b = "hello";
// }
// else {
//   b = "world";
// }

```

如果测试表达式（这里是 `a > 41`）求值为 `true`，那么就会得到第一个子句（`"hello"`），否则得到第二个子句（`"world"`），而且无论结果为何都会被赋值给 `b`。

条件操作符不一定非要用赋值，但是这绝对是最常见的用法。

注意：关于测试条件和 `switch` 与 `? :` 的其他模式的更多信息，参见本系列的 [类型与文法](#)。

## Strict模式

ES5在语言中加入了一个“strict模式”，它收紧了一些特定行为的规则。一般来说，这些限制被视为使代码符合一组更安全和更合理的指导方针。另外，坚持strict模式一般会使你的代码对引擎有更强的可优化性。strict模式对代码有很大的好处，你应当在你所有的程序中使用它。

根据你摆放strict模式注解的位置，你可以为一个单独的函数，或者是整个一个文件切换到strict模式：

```

function foo() {
    "use strict";

    // 这部分代码是strict模式的

    function bar() {
        // 这部分代码是strict模式的
    }
}

// 这部分代码不是strict模式的

```

将它与这个相比：

```

"use strict";

function foo() {
    // 这部分代码是strict模式的

    function bar() {
        // 这部分代码是strict模式的
    }
}

// 这部分代码是strict模式的

```

使用strict模式的一个关键不同（改善！）是，它不允许因为省略了 var 而进行隐含的全局变量声明：

```

function foo() {
    "use strict";      // 打开strict模式
    a = 1;            // 缺少`var`，ReferenceError
}

foo();

```

如果你在代码中打开strict模式，并且得到错误，或者代码开始变得有bug，这可能会诱使你避免使用strict模式。但是纵容这种直觉不是一个好主意。如果strict模式在你的程序中导致了问题，那么这标志着在你的代码中几乎可以肯定有应该修改的东西。

strict模式不仅将你的代码保持在更安全的道路上，也不仅将使你的代码可优化性更强，它还代表着这种语言未来的方向。对于你来说，现在就开始习惯于strict模式要比一直回避它容易得多——以后再进行这种转变只会更难！

注意：关于strict模式的更多信息，参见本系列的 [类型与文法](#) 的第五章。

## 函数作为值

至此，我们已经将函数作为JavaScript中主要的作用域机制讨论过了。你可以回想一下典型的 `function` 声明语法是这样的：

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

虽然从这种语法中看起来不明显，`foo` 基本上是一个位于外围作用域的变量，它给了被声明的 `function` 一个引用。也就是说，`function` 本身是一个值，就像 `42` 或 `[1,2,3]` 一样。

这可能听起来像是一个奇怪的概念，所以花点儿时间仔细考虑一下。你不仅可以向一个 `function` 传递一个值（参数值），而且一个函数本身可以是一个值，它能够赋值给变量，传递给其他函数，或者从其它函数中返回。

因此，一个函数值应当被认为是一个表达式，与任何其他的值或表达式很相似。

考虑如下代码：

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

var x = function bar(){
    // ..
};
```

第一个被赋值给变量 `foo` 的函数表达式称为 **匿名函数表达式**，因为它没有“名称”。

第二个函数表达式是命名的（`bar`），它还被赋值给变量 `x` 作为它的引用。命名函数表达式一般来说更理想，虽然匿名函数表达式仍然极其常见。

更多信息参见本系列的作用域与闭包。

## 立即被调用的函数表达式（IIFE）

在前一个代码段中，哪一个函数表达式都没有被执行——除非我们使用了 `foo()` 或 `x()`。

有另一种执行函数表达式的方法，它通常被称为一个 **立即被调用的函数表达式（IIFE）**：

```
(function IIFE(){
    console.log( "Hello!" );
})();
// "Hello!"
```

围绕在函数表达式 `(function IIFE(){ .. })` 外部的 `( .. )` 只是一个微妙的JS文法，我们需要它来防止函数表达式被看作一个普通的函数声明。

在表达式末尾的最后的 `() —— }())()`；这一行 —— 才是实际立即执行它前面的函数表达式的东西。

这看起来可能很奇怪，但它不像第一眼看上去那么陌生。考虑这里的 `foo` 和 `IIFE` 之间的相似性：

```
function foo() { .. }

// `foo` 是函数引用表达式，然后用`()` 执行它
foo();

// `IIFE` 是函数表达式，然后用`()` 执行它
(function IIFE(){ .. })();
```

如你所见，在执行它的 `()` 之前列出 `(function IIFE(){ .. })`，与在执行它的 `()` 之前定义 `foo` 实质上是相同的；在这两种情况下，函数引用都使用立即在它后面的 `()` 执行。

因为 `IIFE` 只是一个函数，而函数可以创建变量作用域，以这样的风格使用一个 `IIFE` 经常被用于定义变量，而这些变量将不会影响围绕在 `IIFE` 外面的代码：

```
var a = 42;

(function IIFE(){
  var a = 10;
  console.log(a);    // 10
})();

console.log(a);      // 42
```

`IIFE` 还可以有返回值：

```
var x = (function IIFE(){
  return 42;
})();

x;      // 42
```

值 `42` 从被执行的命名为 `IIFE` 的函数中 `return`，然后被赋值给 `x`。

## 闭包

闭包是JavaScript中最重要，却又经常最少为人知的概念之一。我不会在这里涵盖更深的细节，你可以参照本系列的作用域与闭包。但我想说几件关于它的事情，以便你了解它的一般概念。它将是你的JS技术结构中最重要的技术之一。

你可以认为闭包是这样一种方法：即使函数已经完成了运行，它依然可以“记住”并持续访问函数的作用域。

考虑如下代码：

```
function makeAdder(x) {
    // 参数 `x` 是一个内部变量

    // 内部函数 `add()` 使用 `x`，所以它对 `x` 拥有一个“闭包”
    function add(y) {
        return y + x;
    };

    return add;
}
```

每次调用外部的 `makeAdder(..)` 所返回的对内部 `add(..)` 函数的引用可以记住被传入 `makeAdder(..)` 的 `x` 值。现在，让我们使用 `makeAdder(..)`：

```
// `plusOne` 得到一个指向内部函数 `add(..)` 的引用，
// `add()` 函数拥有对外部 `makeAdder(..)` 的参数 `x`
// 的闭包
var plusOne = makeAdder( 1 );

// `plusTen` 得到一个指向内部函数 `add(..)` 的引用，
// `add()` 函数拥有对外部 `makeAdder(..)` 的参数 `x`
// 的闭包
var plusTen = makeAdder( 10 );

plusOne( 3 );           // 4 <-- 1 + 3
plusOne( 41 );          // 42 <-- 1 + 41

plusTen( 13 );          // 23 <-- 10 + 13
```

这段代码的工作方式是：

- 当我们调用 `makeAdder(1)` 时，我们得到一个指向它内部的 `add(..)` 的引用，它记住了 `x` 是 `1`。我们称这个函数引用为 `plusOne(..)`。
- 当我们调用 `makeAdder(10)` 时，我们得到了另一个指向它内部的 `add(..)` 引用，它记住了 `x` 是 `10`。我们称这个函数引用为 `plusTen(..)`。
- 当我们调用 `plusOne(3)` 时，它在 `3`（它内部的 `y`）上加 `1`（被 `x` 记住的），于是我们得到结果 `4`。
- 当我们调用 `plusTen(13)` 时，它在 `13`（它内部的 `y`）上加 `10`（被 `x` 记住的），于是

我们得到结果 23 。

如果这看起来很奇怪和令人困惑，不要担心——它确实是的！要完全理解它需要很多的练习。

但是相信我，一旦你理解了它，它就是编程中最强大最有用的技术之一。让你的大脑在闭包中煎熬一会是绝对值得的。在下一节中，我们将进一步实践闭包。

## 模块

在JavaScript中闭包最常见的用法就是模块模式。模块让你定义对外面世界不可见的私有实现细节（变量，函数），和对外面可访问的公有API。

考虑如下代码：

```
function User(){
    var username, password;

    function doLogin(user,pw) {
        username = user;
        password = pw;

        // 做登录的工作
    }

    var publicAPI = {
        login: doLogin
    };

    return publicAPI;
}

// 创建一个 `User` 模块的实例
var fred = User();

fred.login( "fred", "12Battery34!" );
```

函数 `User()` 作为一个外部作用域持有变量 `username` 和 `password`，以及内部 `doLogin()` 函数；它们都是 `User` 模块内部的私有细节，是不能从外部世界访问的。

**警告：** 我们在这里没有调用 `new User()`，这是有意为之的，虽然对大多数读者来说那可能更常见。`User()` 只是一个函数，不是一个要被初始化的对象，所以它只是被一般地调用了。使用 `new` 将是不合适的，而且实际上会浪费资源。

执行 `User()` 创建了 `User` 模块的一个实例——一个全新的作用域会被创建，而每个内部变量/函数的一个全新的拷贝也因此而被创建。我们将这个实例赋值给 `fred`。如果我们再次运行 `User()`，我们将会得到一个与 `fred` 完全分离的新的实例。

内部的 `doLogin()` 函数在 `username` 和 `password` 上拥有闭包，这意味着即便 `User()` 函数已经完成了运行，它依然持有对它们的访问权。

`publicAPI` 是一个带有一个属性/方法的对象，`login` 是一个指向内部 `doLogin()` 函数的引用。当我们从 `User()` 中返回 `publicAPI` 时，它就变成了我们称为 `fred` 的实例。

在这个时候，外部的 `User()` 函数已经完成了执行。一般说来，你会认为像 `username` 和 `password` 这样的内部变量将会消失。但是在这里它们不会，因为在 `login()` 函数里有一个闭包使它们继续存活。

这就是为什么我们可以调用 `fred.login(..)` —— 和调用内部的 `doLogin(..)` 一样 —— 而且它依然可以访问内部变量 `username` 和 `password`。

这样对闭包和模块模式的简单一瞥，你很有可能还是有点儿糊涂。没关系！要把它装进你的大脑确实需要花些功夫。

以此为起点，关于更多深入细节的探索可以去读本系列的作用域与闭包。

## this 标识符

在JavaScript中另一个经常被误解的概念是 `this` 标识符。同样，在本系列的 `this` 与对象原型中有好几章关于它的内容，所以在这里我们只简要的介绍一下概念。

虽然 `this` 可能经常看起来是与“面向对象模式”有关的，但在JS中 `this` 是一个不同的概念。

如果一个函数在它内部拥有一个 `this` 引用，那么这个 `this` 引用通常指向一个 `object`。但是指向哪一个 `object` 要看这个函数是如何被调用的。

重要的是要理解 `this` 不是指函数本身，这是最常见的误解。

这是一个快速的说明：

```

function foo() {
    console.log( this.bar );
}

var bar = "global";

var obj1 = {
    bar: "obj1",
    foo: foo
};

var obj2 = {
    bar: "obj2"
};

// -----

foo();           // "global"
obj1.foo();      // "obj1"
foo.call( obj2 ); // "obj2"
new foo();        // undefined

```

关于 `this` 如何被设置有四个规则，它们被展示在这个代码段的最后四行中：

- `foo()` 最终在非strict模式中将 `this` 设置为全局对象——在strict模式中，`this` 将会是 `undefined` 而且你会在访问 `bar` 属性时得到一个错误——所以 `this.bar` 的值是 `global`。
- `obj1.foo()` 将 `this` 设置为对象 `obj1`。
- `foo.call(obj2)` 将 `this` 设置为对象 `obj2`。
- `new foo()` 将 `this` 设置为一个新的空对象。

底线：要搞清楚 `this` 指向什么，你必须检视当前的函数是如何被调用的。它将是我们刚刚看到的四种中的一种，而这将会回答 `this` 是什么。

注意：关于 `this` 的更多信息，参见本系列的 `this与对象原型` 的第一和第二章。

## 原型

JavaScript中的原型机制十分复杂。我们在这里近仅仅扫它一眼。要了解关于它的所有细节，你需要花相当的时间来学习本系列的 `this与对象原型` 的第四到六章。

当你引用一个对象上的属性时，如果这个属性不存在，JavaScript将会自动地使用这个对象的内部原型引用来寻找另外一个对象，在它上面查询你想要的属性。你可以认为它几乎是在属性缺失时的备用对象。

从一个对象到它备用对象的内部原型引用链接发生在这个对象被创建的时候。说明它的最简单的方法是使用称为 `Object.create(..)` 的内建工具。

考虑如下代码：

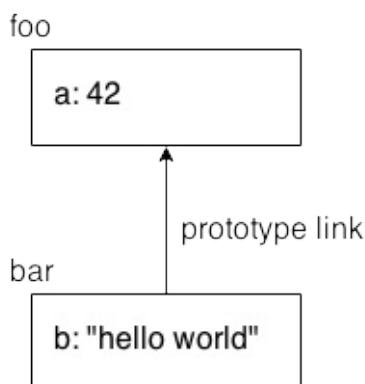
```
var foo = {
  a: 42
};

// 创建 `bar` 并将它链接到 `foo`
var bar = Object.create( foo );

bar.b = "hello world";

bar.b;           // "hello world"
bar.a;           // 42 <-- 委托到 `foo`
```

将对象 `foo` 和 `bar` 以及它们的关系可视化也许会有所帮助：



属性 `a` 实际上不存在于对象 `bar` 上，但是因为 `bar` 被原型链接到 `foo`，JavaScript 自动地退到对象 `foo` 上去寻找 `a`，而且在这里找到了它。

这种链接看起来是语言的一种奇怪的特性。这种特性最常被使用的方式——我会争辩说这是一种滥用——是用来模拟/模仿“类”机制的“继承”。

使用原型的更自然的方式是一种称为“行为委托”的模式，在这种模式中你有意地将你的被链接的对象设计为可以从一个委托到另一个的部分所需的行为中。

注意：更多关于原型和行为委托的信息，参见本系列的 *this与对象原型* 的第四到六章。

## 旧的与新的

以我们已经介绍过的JS特性，和将在这个系列的其他部分中讲解的相当一部分特性都是新近增加的，不一定在老版本的浏览器中可用。事实上，语言规范中的一些最新特性甚至在任何稳定的浏览中都没有被实现。

那么，你拿这些新东西怎么办？你只能等上几年或者十几年直到老版本浏览器归于尘土？

这确实是许多人认为的情况，但是它不是JS健康进步的方式。

有两种主要的技术可以将新的JavaScript特性“带到”老版本的浏览器中：填补和转译。

## 填补

“填补（Polyfilling）”是一个认为发明的词（由Remy Sharp创造）

（<https://remysharp.com/2010/10/08/what-is-a-polyfill>）。它是指拿来一个新特性的定义并制造一段行为等价的代码，但是这段代码可以运行在老版本的JS环境中。

例如，ES6定义了一个称为 `Number.isNaN(..)` 的工具，来为检查 `Nan` 值提供一种准确无误的方法，同时废弃原来的 `isNaN(..)` 工具。这个工具可以很容易填补，因此你可开始在你的代码中使用它，而不管最终用户是否在一个ES6浏览器中。

考虑如下代码：

```
if (!Number.isNaN) {
    Number.isNaN = function isNaN(x) {
        return x !== x;
    };
}
```

`if` 语句决定着在这个工具已经存在的ES6环境中不再进行填补。如果它还不存在，我们就定义 `Number.isNaN(..)`。

注意：我们在这里做的检查利用了 `NaN` 值的怪异之处，即它们是整个语言中唯一与自己不相等的值。所以 `NaN` 是唯一可能使 `x !== x` 为 `true` 的值。

并不是所有的新特性都可以完全填补。有时一种特性的大部分行为可以被填补，但是仍然存在一些小的偏差。在实现你自己的添补时你应当非常非常小心，来确保你尽可能严格地遵循语言规范。

或者更好地，使用一组你信任的，经受过检验的添补，比如那些由ES5-Shim（<https://github.com/es-shims/es5-shim>）和ES6-Shim（<https://github.com/es-shims/es6-shim>）提供的。

## 转译

没有任何办法可以添补语言中新增加的语法。在老版本的JS引擎中新的语法将因为不可识别/不合法而抛出一个错误。

所以更好的选择是使用一个工具将你的新版本代码转换为等价的老版本代码。这个处理通常被称为“转译（transpiling）”，表示转换 + 编译。

实质上，你的源代码是使用新的语法形式编写的，但是你向浏览器部署的是转译过的旧语法形式。你一般会将转译器插入到你的构建过程中，与你的代码linter和代码压缩器类似。

你可能想知道为什么要麻烦地使用新语法编写程序又将它转译为老版本代码——为什么不直接编写老版本代码呢？

关于转译你应当注意几个重要的原因：

- 在语言中新加人的语法是为了使你的代码更具可读性和维护性而设计的。老版本的等价物经常会绕多得多的圈子。你应当首选编写新的和干净的语法，不仅为你自己，也为了开发团队的其他的成员。
- 如果你仅为老版本浏览器转译，而给最新的浏览器提供新语法，那么你就可以利用浏览器对新语法进行的性能优化。这也让浏览器制造商有更多真实世界的代码来测试它们的实现和优化方法。
- 提早使用新语法可以允许它在真实世界中被测试得更加健壮，这给JavaScript协会（TC39）提供了更早的反馈。如果问题被发现的足够早，他们就可以在那些语言设计错误变得无法挽回之前改变/修改它。

这是一个转译的简单例子。ES6增加了一个称为“默认参数值”的新特性。它看起来像是这样：

```
function foo(a = 2) {
  console.log(a);
}

foo();           // 2
foo( 42 );      // 42
```

简单，对吧？也很有用！但是这种新语法在前ES6引擎中是不合法的。那么转译器将会对这段代码做什么才能使它在老版本环境中运行呢？

```
function foo() {
  var a = arguments[0] !== (void 0) ? arguments[0] : 2;
  console.log(a);
}
```

如你所见，它检查 `arguments[0]` 值是否是 `void 0`（也就是 `undefined`），而且如果是，就提供默认值 `2`；否则，它就赋值被传递的任何东西。

除了可以现在就在老版本浏览器中使用更好的语法以外，观察转译后的代码实际上更清晰地解释了意图中的行为。

仅从ES6版本的代码看来，你可能还不理解 `undefined` 是唯一不能作为参数默认值的明确传递的值，但是转译后的代码使这一点清楚的多。

关于转译要强调的最后一个细节是，现在它们应当被认为是JS开发的生态系统和过程中的标准部分。JS将继续以比以前快得多的速度进化，所以每几个月就会有新语法和新特性被加入进来。

如果你默认地使用一个转译器，那么你将总是可以在你发现新语法有用时，立即开始使用它，而不必为了让今天的浏览器被淘汰而等上好几年。

有好几个了不起的转译器供你选择。这是一些在本书写作时存在的好选择：

- Babel (<https://babeljs.io>) (formerly 6to5): 将 ES6+ 转译为 ES5
- Traceur (<https://github.com/google/traceur-compiler>): 将 ES6, ES7, 和以后特性转译为 ES5

## 非JavaScript

至此，我们讨论过的所有东西都限于JS语言本身。现实是大多数JS程序都是在浏览器这样的环境中运行并与之互动的。你在你的代码中编写的很大一部分东西，严格地说，不是直接由JavaScript控制的。这听起来可能有点奇怪。

你将会遇到的最常见的非JavaScript程序是DOM API。例如：

```
var el = document.getElementById( "foo" );
```

当你的代码运行在一个浏览器中时，变量 `document` 作为一个全局变量存在。它不是由JS引擎提供的，也不为JavaScript语言规范所控制。它采取了某种与普通JS `object` 极其相似的形式，但它不是真正的 `object`。它是一种特殊的 `object`，经常被称为“宿主对象”。

另外，`document` 上的 `getElementById(..)` 方法看起来像一个普通的JS函数，但它只是一个微微暴露出来的接口，指向由浏览器DOM提供的内建方法。在一些（新一代的）浏览器中，这一层可能也是由JS实现的，但是传统的DOM和它的行为是由像C/C++这样的语言实现的。

另一个例子是输入/输出（I/O）。

大家最喜爱的 `alert(..)` 在用户的浏览器窗口中弹出一个消息框。`alert(..)` 是由浏览器提供给你的JS程序的，而不是JS引擎本身。你进行的调用将消息发送给浏览器内部，它来处理消息框的绘制与显示。

`console.log()` 也一样；你的浏览器提供这样的机制并将它们挂在开发者工具中。

这本书，和整个这个系列，聚焦于JavaScript这种语言。这就是为什么你看不到任何涵盖这些非JavaScript机制的重要内容。不管怎样，你需要小心它们，因为它们将在你写的每一个JS程序中存在！

## 复习

学习JavaScript风格编程的第一步是对它的核心机制有一个基本的了解，比如值，类型，函数闭包，`this`，和原型。

当然，这些话题中的每一个都会衍生出比你在这里见到的多得多的内容，这也是为什么它们在这个系列剩下的部分中拥有自己的章节和书目。在你对本章中的概念和代码示例感到相当适应之后，这个系列的其他部分正等着你真正地深入挖掘和了解这门语言。

这本书的最后一章将会对这个系列的每一卷的内容，以及它们所涵盖的我们在这里还没有探索过的概念，进行简单地总结。

# 你不懂JS：入门与进阶

## 第三章：进入YDKJS

这个系列丛书到底是为了什么？简单地说，它的目的是认真地学习 *JavaScript* 的所有部分，不仅是这门语言的某些人称之为“好的部分”的子集，也不仅是让你在工作中搞定任务所需的小部分的知识。

其他语言中，认真的开发者总是希望努力学习他们主要使用的语言的大部分或全部，但是JS开发者由于通常不太学习这门语言而在人群中显得很扎眼。这不是一件好事，而且我们也不应当继续将之视为常态。

你不懂JS（YDKJS）系列的立场是与学习JS的通常方式形成鲜明的对比，而且与你将会读到的其他JS书籍不同。它挑战你超越自己的舒适区，对每一个你遇到的行为问一个更深入的“为什么”。你准备好接受挑战了吗？

我将用这最后一章的篇幅来简要地总结一下这个系列其他书目的内容，和如何在 YDKJS 的基础上最有效地建立学习JS的基础。

## 作用域与闭包

也许你需要快速接受的基础之一，就是在JavaScript中变量的作用域是如何工作的。关于作用域仅有传闻中的模糊观念是不够的。

作用域与闭包 从揭穿常见的误解开始：JS是“解释型语言”因此是不被编译的。不对。

JS引擎在你的代码执行的前一刻（有时是在执行期间！）编译它。所以我们首先深入了解编译器处理我们代码的方式，以此来理解它如何找到并处理变量和函数的声明。沿着这条道路，我们将见到JS变量作用域管理的特有隐喻，“提升”。

对“词法作用域”的极其重要的理解，是我们在这本书最后一章探索闭包时所需的基石。闭包也许是JS所有的概念中最重要的一个，但如果你没有首先牢牢把握住作用域的工作方式，那么闭包将很可能依然不在你的掌握之中。

闭包的一个重要应用是模块模式，正如我们在本书第二章中简要介绍过的那样。模块模式也许是JavaScript的所有代码组织模式中最流行的一种；深刻理解它应当是你的首要任务之一。

## this与对象原型

也许关于JavaScript传播得最广泛和持久的谬误之一是认为 `this` 关键字指代它所出现的函数。可怕的错误。

`this` 关键字是根据函数如何被执行而动态绑定的，而事实上有四种简单的规则可以用来理解和完全决定 `this` 绑定。

和 `this` 密切相关的是对象原型属性，它是一种属性的查询链，与查询词法作用域变量的方式相似。但是原型中包含的是另一个关于JS的巨大谬误：模拟（山寨）类和继承（所谓的“原型继承”）的想法。

不幸的是，渴望将类和继承的设计模式思想带入JavaScript只是你能做的最差劲儿的事情，因为虽然语法可能欺骗你，使你认为有类这样的东西存在，但实际上原型机制在行为上是根本相反的。

目前的问题是，是忽略这种错位并假装你实现的是“继承”更好，还是学习并接纳对象原型系统实际的工作方式更恰当。后者被称为“行为委托”更合适。

这不光是语法上的偏好问题。委托是一种完全不同的，更强大的设计模式，其中的原因之一就是它取代了使用类和继承进行设计的需要。但是对于以谈论JavaScript的一生为主题的几乎所有的其他博客，书籍，和论坛来说，这些断言绝对是打脸的。

我对委托和继承做出的宣言不是源于对语言和其语法的厌恶，而是来自于渴望看到这门语言的真实力量被正确地利用，渴望看到无尽的困惑与沮丧被一扫而光。

但是我举出的关于原型和委托的例子可要比我在这里乱说的东西复杂得多。如果你准备好重新思考你认为你所了解的关于JavaScript“类”和“继承”的一切，我给你一个机会来“服用红色的药丸”，并且看一看本系列的 `this` 与对象原型的第四到六章。

## 类型与文法

这个系列的第三本书主要集中于解决另一个极具争议的话题：类型强制转换。也许没有什么话题能比你谈论隐含的强制转换造成的困惑更能使JS开发者感到沮丧了。

到目前为止，惯例的智慧说隐含强制转换是这门语言的“坏的部分”，并且应当不计一切避免它。事实上，有些人已经到了将它称为语言设计的“缺陷”的地步了。确实存在这么一些工具，它们的全部工作就是扫描你的代码，并在你进行任何强制转换，甚至是做有些像强制转换的事情时报警。

但是强制转换真的如此令人困惑，如此的坏，如此的不可信，以至于只要你使用它，你的代码从一开始就灭亡了吗？

我说不。在第一到三章中建立了对类型和值真正的工作方式的理解后，第四章参与了这个辩论，并从强制转换的角落和缝隙全面地讲解它的工作方式。我们将看到强制转换的那一部分真的令人惊讶，而且如果花时间去学习，那一部分实际上完全是合理的。

但我不仅仅要说强制转换是合理的和可以学习的，我断言强制转换是一种你应当在代码中使用的极其有用而且完全被低估的工具。我要说在合理使用的情况下，强制转换不仅可以工作，而且会使你的代码更好。所有唱反调的和怀疑的人当然会嘲笑这样的立场，但我相信它是让你玩儿好JS游戏的主要按键之一。

你是想继续人云亦云，还是想将所有的臆测放在一边，用一个全新的视角观察强制转换？这个系列的类型与文法将会强制转换你的想法。

## 异步与性能

这个系列的前三本书聚焦于这门语言的核心技术，但是第四本书稍稍开出一个分支来探讨在这门语言技术之上的管理异步编程的模式。异步不仅对于性能和我们的应用程序很关键，而且它日渐成为改进可写性和可维护性的关键因素。

这本书从搞清楚许多令人困惑的术语和概念开始，比如“异步”，“并行”和“并发”。而且深入讲解了这些东西如何适用和不适用于JS。

然后我们继续检视作为开启异步的主要方法：回调。但我们很快就会看到，对于现代异步编程的需求来说，单靠回调自身是远远不够的。我们将找出仅使用回调编码的两种主要的不足之处：控制反转（IoC）信任丢失和缺乏线性的可推理性。

为了解决这两种主要的不足，ES6引入了两种新的机制（实际上也是模式）：`promise` 和 `generator`。

`Promise`是一个“未来值”的一种与时间无关的包装，它让你推理并组合这些未来值而不必关心它们是否已经准备好。另外，它们通过将回调沿着一个可信赖和可组装的`promise`机制传递，有效地解决了IoC信任问题。

`Generator`给JS函数引入了一种新的执行模式，`generator`可以在 `yield` 点被暂停而稍后异步地被继续。这种“暂停-继续”的能力让`generator`在幕后异步地被处理，使看起来同步，顺序执行的代码成为可能。如此，我们就解决了回调的非线性，非本地跳转的困惑，并因此使我们的异步代码看起来是更容易推理的同步代码。

但是，是`promise`与`generator`的组合给了我们JavaScript中最有效的异步代码模式。事实上，在即将到来的ES7与之后的版本中，大多数精巧的异步性肯定会建立在这个基础之上。为了认真地在一个异步的世界中高效地编程，你将需要对`promise`与`generator`的组合十分适应。

如果`promise`和`generator`是关于表达一些模式，这些模式让你的程序更加并发地运行，而因此在更短的时间内完成更多的处理，那么JS在性能优化上就拥有许多其他的方面值得探索。

第五章钻研的话题是使用Web Worker的程序并行性和使用SIMD的数据并行性，以及像`ASM.js`这样的底层优化技术。第六章从正确的基准分析技术的角度来观察性能优化，包括什么样的性能值得关心而什么应当忽略。

高效地编写JavaScript意味着编写的代码可以突破这种限制壁垒：在范围广泛的浏览器和其他环境中动态运行。这需要我们进行更多复杂的详细计划与努力，才能使一个程序从“可以工作”到“工作得很好”。

给你编写合理且高效的JavaScript代码所需的全部工具与技能，异步与性能就是为此而设计的。

## ES6与未来

至此，无论你感觉自己已经将JavaScript掌握的多么好，现实是JavaScript从来没有停止过进化，而且进化的频率正在飞快地增长。这个事实几乎就是本系列精神的含义，拥抱我们永远不会完全懂得的JS的所有部分，因为只要你掌握了它的全部，就会有你需要学习的新的东西到来。

这本书专注于这门语言在中短期的发展前景，不仅是像ES6这样已知的东西，还包括在未来可能的东西。

虽然这个系列的所有书目采纳的是在编写它们时JavaScript的状态，也就是ES6正在被接纳的半途中，但是这个系列更主要地集中于ES5。现在我们想要将注意力转移到ES6，ES7，和.....

因为在编写本书时ES6已经近于完成，ES6与未来首先将ES6中确定的东西分割为几个关键的范畴，包括新的语法，新的数据结构（集合），和新的处理能力以及API。我们将在各种细节的层面讲解这些新的ES6特性中的每一个，包括复习我们在本系列的其他书目中遇到过的细节。

这是一些值得一读的激动人心的ES6特性：解构，参数默认值，symbol，简洁方法，计算属性，箭头函数，块儿作用域，promise，generator，iterator，模块，代理，weakmap，以及很多，很多别的东西！呼，ES6真是不容小觑！

这本书的第一部分是一张路线图，为了对你将要在以后几年中编写和探索的新改进的JavaScript做好准备，它指明了你需要学习的所有东西。

这本书稍后的部分将注意力转向简要地介绍一些我们将在近未来可能看到的JavaScript的新东西。在这里最重要的是，要理解在后ES6时代，JS很可能将会一个特性一个特性地进化，而不是一个版本一个版本地进化，这意味着我们将在比你想象的早得多的时候，看到这些近未来的到来。

JavaScript的未来是光明的。这不正是我们开始学习它好时机吗！？

## 复习

YDKJS 系列投身于这样的命题：所有的JS开发者都可以，也应该学习这门伟大语言的每一部分。没有任何个人意见，没有任何框架的设想，没有任何项目的期限可以作为你从没有学习和深入理解JavaScript的借口。

我们聚焦这门语言中的每一个重要领域，为之专著一本很短但是内容非常稠密的书，来全面地探索它的——你也许认为自己知道但可能并不全面——所有部分。

“你不懂JS”不是一种批评或羞辱。它是我们所有人，包括我自己，都必须正视的一种现实。学习JavaScript不是一个最终目标，而是一个过程。我们还不懂JavaScript。但是我们会的！

# 你不懂JS：入门与进阶

## 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Jenn Lukas为本书作序。

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 Tregoining, 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 Hodsdon, 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 ב-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 enamorato, 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 Ziolkowski, 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 Viloslada, 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 Gharegozlu, 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。

# 你不懂JS：作用域与闭包

## 目录

- 序
- 前言
- 第一章：什么是作用域？
  - 编译器理论
  - 理解作用域
  - 嵌套的作用域
  - 错误
- 第二章：词法作用域
  - 词法分析时
  - 欺骗词法作用域
- 第三章：函数与块儿作用域
  - 函数中的作用域
  - 隐藏于普通作用域
  - 函数作为作用域
  - 块儿作为作用域
- 第四章：提升
  - 先有鸡还是先有蛋？
  - 编译器再次袭来
  - 函数优先
- 第五章：作用域闭包
  - 启示
  - 事实真相
  - 现在我能看到
  - 循环 + 闭包
  - 模块
- 附录A：动态作用域
- 附录B：填补块儿作用域
- 附录C：词法this
- 附录D：鸣谢

# 你不懂JS：作用域与闭包

## 第一章：什么是作用域？

几乎所有语言的最基础模型之一就是在变量中存储值，并且在稍后取出或修改这些值的能力。事实上，在变量中存储值和取出值的能力，给程序赋予了状态。

如果没有这样的概念，一个程序虽然可以执行一些任务，但是它们将会受到极大的限制而且不会非常有趣。

但是在我们的程序中纳入变量，引出了我们现在将要解决的最有趣的问题：这些变量存活在哪里？换句话说，它们被存储在哪儿？而且，最重要的是，我们的程序如何在需要它们的时候找到它们？

回答这些问题需要一组明确定义的规则，它定义如何在某些位置存储变量，以及如何在稍后找到这些变量。我们称这组规则为：作用域。

但是，这些作用域规则是在哪里，如何被设置的？

## 编译器理论

根据你与各种编程语言打交道的水平不同，这也许是不证自明的，或者这也许令人吃惊，尽管JavaScript一般被划分到“动态”或者“解释型”语言的范畴，但是其实它是一个编译型语言。它不是像许多传统意义上的编译型语言那样预先被编译好，编译的结果也不能在各种不同的分布式系统间移植。

但是无论如何，JavaScript引擎在实施许多与传统的语言编译器相同的步骤，虽然是以一种我们平常不能发觉的更精巧的方式。

在传统的编译型语言处理中，一块儿源代码，你的程序，在它被执行之前典型地将会经历三个步骤，大致被称为“编译”：

1. 分词/词法分析：将一连串字符打断成（对于语言来说）有意义的片段，称为token（记号）。举例来说，考虑这段程序：`var a = 2;`。这段程序很可能被打断成如下 token：`var`，`a`，`=`，`2`，和`；`。空格也许会被保留为一个token，这要看它是否是有意义的。

注意：分词和词法分析之间的区别是微妙和学术上的，其中心在于这些token是否以无状态或有状态的方式被识别。简而言之，如果分词器去调用有状态的解析规则来弄清`a`是否应当被考虑为一个不同的token，还是只是其他token的一部分，那么这就是词

法分析。

2. 解析：将一个token的流（数组）转换为一个嵌套元素的树，它总体上表示了程序的语法结构。这棵树称为“AST”（Abstract Syntax Tree —— 抽象语法树）。

`var a = 2;` 的树也许开始于称为 `VariableDeclaration` （变量声明）顶层节点，带有一个称为 `Identifier` （标识符）的子节点（它的值为 `a`），和另一个称为 `AssignmentExpression` （赋值表达式）的子节点，而这个子节点本身带有一个称为 `NumericLiteral` （数字字面量）的子节点（它的值为 `2`）。

3. 代码生成：这个处理将AST转换为可执行的代码。这一部分将根据语言，它的目标平台等因素有很大的不同。

所以，与其深陷细节，我们不如笼统地说，有一种方法将我们上面描述的 `var a = 2;` 的 AST转换为机器指令，来实际上创建一个称为 `a` 的变量（包括分配内存等等），然后在 `a` 中存入一个值。

注意：引擎如何管理系统资源的细节远比我们要挖掘的东西深刻，所以我们将理所当然地认为引擎有能力按其需要创建和存储变量。

和大多数其他语言的编译器一样，JavaScript引擎要比这区区三步复杂太多了。例如，在解析和代码生成的处理中，一定会存在优化执行效率的步骤，包括压缩冗余元素，等等。

所以，我在此描绘的只是大框架。但是我想你很快就会明白为什么我们涵盖的这些细节是重要的，虽然是在很高的层次上。

其一，JavaScript引擎没有（像其他语言的编译器那样）大把的时间去优化，因为JavaScript的编译和其他语言不同，不是提前发生在一个编译的步骤中。

对JavaScript来说，在许多情况下，编译发生在代码被执行前的仅仅几微妙之内（或更少！）。为了确保最快的性能，JS引擎将使用所有的招数（比如JIT，它可以懒编译甚至是热编译，等等），而这远超出了我们的关于“作用域”的讨论。

为了简单起见，我们可以说，任何JavaScript代码段在它执行之前（通常是刚好在它执行之前！）都必须被编译。所以，JS编译器将把程序 `var a = 2;` 拿过来，并首先编译它，然后准备运行它，通常是立即的。

## 理解作用域

我们将采用的学习作用域的方法，是将这个处理过程想象为一场对话。但是，谁在进行这场对话呢？

### 演员

让我们见一见处理程序 `var a = 2;` 时进行互动的演员吧，这样我们就能理解稍后将要听到的它们的对话：

1. 引擎：负责从始至终的编译和执行我们的JavaScript程序。
2. 编译器：引擎的朋友之一；处理所有的解析和代码生成的重活儿（见前一节）。
3. 作用域：引擎的另一个朋友；收集并维护一张所有被声明的标识符（变量）的列表，并对当前执行中的代码如何访问这些变量强制实施一组严格的规则。

为了全面理解 JavaScript 是如何工作的，你需要开始像引擎（和它的朋友们）那样思考，问它们问的问题，并像它们一样回答。

## 反复

当你看到程序 `var a = 2;` 时，你很可能认为它是一个语句。但这不是我们的新朋友引擎所看到的。事实上，引擎看到两个不同的语句，一个是编译器将在编译期间处理的，一个是引擎将在执行期间处理的。

那么，然我们来分析引擎和它的朋友们将如何处理程序 `var a = 2;`。

编译器将对这个程序做的第一件事情，是进行词法分析来将它分解为一系列 token，然后这些 token 被解析为一棵树。但是当编译器到了代码生成阶段时，它会以一种与我们可能想象的不同的方式来对待这段程序。

一个合理的假设是，编译器将会产生可以用这种假想代码概括的代码：“为一个变量分配内存，将它标记为 `a`，然后将值 `2` 贴在这个变量里”。不幸的是，这不是十分准确。

编译器将会这样处理：

1. 遇到 `var a`，编译器让作用域去查看对于这个特定的作用域集合，变量 `a` 是否已经存在了。如果是，编译器就忽略这个声明并继续前进。否则，编译器就让作用域去为这个作用域集合声明一个称为 `a` 的新变量。
2. 然后编译器为引擎生成稍后要执行的代码，来处理赋值 `a = 2`。引擎运行的代码首先让作用域去查看在当前的作用域集合中是否有一个称为 `a` 的变量可以访问。如果有，引擎就使用这个变量。如果没有，引擎就查看其他地方（参加下面的嵌套作用域一节）。

如果引擎最终找到一个变量，它就将值 `2` 赋予它。如果没有，引擎将会举起它的手并喊出一个错误！

总结来说：对于一个变量赋值，发生了两个不同的动作：第一，编译器声明一个变量（如果先前没有在当前作用域中声明过），第二，当执行时，引擎在作用域中查询这个变量并给它赋值，如果找到的话。

## 编译器术语

为了继续更深入地理解，我们需要一点儿更多的编译器术语。

当引擎执行编译器在第二步为它产生的代码时，它必须查询变量 `a` 来看它是否已经被声明过了，而且这个查询是咨询作用域的。但是引擎实施的查询的类型会影响查询的结果。

在我们这个例子中，引擎将会对变量 `a` 实施一个“LHS”查询。另一种查询的类型称为“RHS”。

我打赌你能猜出“L”和“R”是什么意思。这两个术语表示“Left-hand Side（左手边）”和“Right-hand Side（右手边）”

什么的.....边？赋值操作的。

换言之，当一个变量出现在赋值操作的左手边时，会进行LHS查询，当一个变量出现在赋值操作的右手边时，会进行RHS查询。

实际上，我们可以表述得更准确一点儿。对于我们的目的来说，一个RHS是难以察觉的，因为它简单地查询某个变量的值，而LHS查询是试着找到变量容器本身，以便它可以赋值。从这种意义上说，RHS的含义实质上不是真正的“一个赋值的右手边”，更准确地说，它只是意味着“不是左手边”。

在这一番油腔滑调之后，你也可以认为“RHS”意味着“取得他/她的源（值）”，暗示着RHS的意思是“去取.....的值”。

让我们挖掘得更深一些。

当我说：

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

这个指向 `a` 的引用是一个RHS引用，因为这里没有东西被赋值给 `a`。而是我们在查询 `a` 并取得它的值，这样这个值可以被传递进 `console.log(..)`。

作为对比：

```
a = 2;
```

这里指向 `a` 的引用是一个LHS引用，因为我们实际上不关心当前的值是什么，我们只是想找到这个变量，将它作为 `= 2` 赋值操作的目标。

注意：LHS和RHS意味着“赋值的左/右手边”未必像字面上那样意味着“= 赋值操作符的左/右边”。赋值有几种其他的发生形式，所以最好在概念上将它考虑为：“赋值的目标（LHS）”和“赋值的源（RHS）”。

考虑这段程序，它既有LHS引用又有RHS引用：

```
function foo(a) {  
    console.log( a ); // 2  
}  
  
foo( 2 );
```

调用 `foo(..)` 的最后一行作为一个函数调用要求一个指向 `foo` 的RHS引用，意味着，“去查询 `foo` 的值，并把它交给我”。另外，`(..)` 意味着 `foo` 的值应当被执行，所以它最好实际上是一个函数！

这里有一个微妙但重要的赋值。你发现了吗？

你可能错过了这个代码段隐含的 `a = 2`。它发生在当值 `2` 作为参数值传递给 `foo(..)` 函数时，这时值 `2` 被赋值给参数 `a`。为了（隐含地）给参数 `a` 赋值，进行了一个LHS查询。

这里还有一个 `a` 的值的RHS引用，它的结果值被传入 `console.log(..)`。`console.log(..)` 需要一个引用来执行。它为 `console` 对象进行一个RHS查询，然后发生一个属性解析来看它是否有一个称为 `log` 的方法。

最后，我们可以将这一过程概念化为，在将值 `2`（通过变量 `a` 的RHS查询得到的）传入 `log(..)` 时发生了一次LHS/RHS的交换。在 `log(..)` 的原生实现内部，我们可以假定它拥有参数，其中的第一个（也许被称为 `arg1`）在 `2` 被赋值给它之前，进行了一次LHS引用查询。

注意：你可能会试图将函数声明 `function foo(a) {...}` 概念化为一个普通的变量声明和赋值，比如 `var foo` 和 `foo = function(a){...}`。这样做会诱使你认为函数声明涉及了一次LHS查询。

然而，一个微妙但重要的不同是，在这种情况下 编译器 同时处理声明和代码生成期间的值的定义，如此当 引擎 执行代码时，没有必要将一个函数值“赋予” `foo`。因此，认为函数声明是一个我们在这里讨论的LHS查询赋值是不太合适的。

## 引擎/作用域对话

```
function foo(a) {  
    console.log( a ); // 2  
}  
  
foo( 2 );
```

让我们将上面的（处理这个代码段的）交互想象为一场对话。这场对话将会有点儿像这样进行：

引擎：嘿 作用域，我有一个 `foo` 的RHS引用。听说过它吗？

作用域：啊，是的，听说过。编译器 刚在一秒钟之前声明了它。它是一个函数。给你。

引擎：太棒了，谢谢！好的，我要执行 `foo` 了。

引擎：嘿，作用域，我得到了一个 `a` 的LHS引用，听说过它吗？

作用域：啊，是的，听说过。编译器 刚才将它声明为 `foo` 的一个正式参数了。给你。

引擎：一如既往的给力，作用域。再次感谢你。现在，该把 `2` 赋值给 `a` 了。

引擎：嘿，作用域，很抱歉又一次打扰你。我需要RHS查询 `console` 。听说过它吗？

作用域：没关系，引擎，这是我一天到晚的工作。是的，我得到 `console` 了。它是一个内建对象。给你。

引擎：完美。查找 `log(..)` 。好的，很好，它是一个函数。

引擎：嘿，作用域。你能帮我查一下 `a` 的RHS引用吗？我想我记得它，但只是想再次确认一下。

作用域：你是对的，引擎。同一个家伙，没变。给你。

引擎：酷。传递 `a` 的值，也就是 `2`，给 `log(..)` 。

...

## 小测验

检查你目前为止的理解。确保你扮演 引擎，并与 作用域“对话”：

```
function foo(a) {  
    var b = a;  
    return a + b;  
}  
  
var c = foo( 2 );
```

1. 找到所有的LHS查询（有3处！）。

2. 找到所有的RHS查询（有4处！）。

注意： 小测验答案参见本章的复习部分！

## 嵌套的作用域

我们说过 作用域 是通过标识符名称查询变量的一组规则。但是，通常会有多于一个的作用域需要考虑。

就像一个代码块儿或函数被嵌套在另一个代码块儿或函数中一样，作用域被嵌套在其他的作用域中。所以，如果在直接作用域中找不到一个变量的话，引擎就会咨询下一个外层作用域，如此继续直到找到这个变量或者到达最外层作用域（也就是全局作用域）。

考虑这段代码：

```
function foo(a) {  
    console.log( a + b );  
}  
  
var b = 2;  
  
foo( 2 ); // 4
```

`b` 的RHS引用不能在函数 `foo` 的内部被解析，但是可以在包围着它的作用域（这个例子中是全局作用域）中解析。

所以，重返引擎和作用域的对话，我们会听到：

引擎：“嘿， `foo` 的作用域，听说过 `b` 吗？我得到一个它的RHS引用。”

作用域：“没有，从没听说过。问问别人吧。”

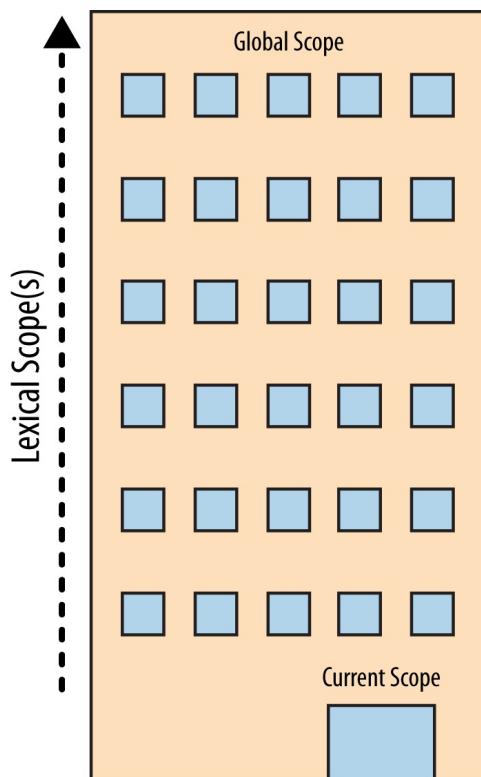
引擎：“嘿， `foo` 外面的作用域，哦，你是全局作用域，好吧，酷。听说过 `b` 吗？我得到一个它的RHS引用。”

作用域：“是的，当然有。给你。”

遍历嵌套作用域的简单规则：引擎从当前执行的作用域开始，在那里查找变量，如果没有找到，就在上一级继续查找，如此类推。如果到了最外层的全局作用域，那么查找就会停止，无论它是否找到了变量。

## 建筑的隐喻

为了将嵌套作用域解析的过程可视化，我想让你考虑一下这个高层建筑。



这个建筑物表示我们程序的嵌套作用域规则集合。无论你在哪里，建筑的第一层表示你当前执行的作用域。建筑的顶层表示全局作用域。

你通过在你当前的楼层中查找来解析LHS和RHS引用，如果你没有找到它，就做电梯到上一层楼，在那里寻找，然后再上一层，如此类推。一旦你到了顶层（全局作用域），你要么找到了你想要的东西，要么没有。但是不管怎样你都不得不停止了。

## 错误

为什么我们区别LHS和RHS那么重要？

因为在变量还没有被声明（在所有被查询的作用域中都没找到）的情况下，这两种类型查询的行为不同。

考虑如下代码：

```
function foo(a) {
  console.log( a + b );
  b = a;
}

foo( 2 );
```

当 `b` 的RHS查询第一次发生时，它是找不到的。它被说成是一个“未声明”的变量，因为它在作用域中找不到。

如果RHS查询在嵌套的作用域的任何地方都找不到一个值，这会导致引擎抛出一个 `ReferenceError`。必须要注意的是这个错误的类型是 `ReferenceError`。

相比之下，如果引擎在进行一个LHS查询并到达了顶层（全局作用域）都没有找到它，而且如果程序没有运行在“Strict模式”`note-strictmode`下，那么这个全局作用域将会在全局作用域中创建一个同名的新变量，并把它交给引擎。

“不，之前没有这样的东西，但是我可以帮忙给你创建一个。”

在ES5中被加入的“Strict模式”`note-strictmode`，有许多与一般/宽松/懒惰模式不同的行为。其中之一就是不允许自动/隐含的全局变量创建。在这种情况下，将不会有全局作用域的变量交回给LHS查询，并且类似于RHS的情况，引擎将抛出一个 `ReferenceError`。

现在，如果一个RHS查询的变量被找到了，但是你试着去做一些这个值不可能做到的事，比如将一个非函数的值作为函数运行，或者引用 `null` 或者 `undefined` 值的属性，那么引擎就会抛出一个不同种类的错误，称为 `TypeError`。

`ReferenceError` 是关于作用域解析失败的，而 `TypeError` 暗示着作用域解析成功了，但是试图对这个结果进行了一个非法/不可能的动作。

## 复习

作用域是一组规则，它决定了在哪里和如何查找一个变量（标识符）。这种查询也许是向这个变量赋值，这时变量是一个LHS（左手边）引用，或者是为取得它的值，这时变量是一个RHS（右手边）引用。

LHS引用得自赋值操作。作用域相关的赋值可以通过 `=` 操作符发生，也可以通过向函数参数传递（赋予）参数值发生。

JavaScript引擎在执行代码之前首先会编译它，这样做，它将 `var a = 2;` 这样的语句分割为两个分离的步骤：

1. 首先，`var a` 在当前作用域中声明。这是在最开始，代码执行之前实施的。
2. 稍后，`a = 2` 查找这个变量（LHS引用），并且如果找到就向它赋值。

LHS和RHS引用查询都从当前执行中的作用域开始，如果有需要（也就是，它们在这里没能找到它们要找的东西），它们会在嵌套的作用域中一路向上，一次一个作用域（层）地查找这个标识符，直到它们到达全局作用域（顶层）并停止，既可能找到也可能没找到。

未满足的RHS引用会导致 `ReferenceError` 被抛出。未满足的LHS引用会导致一个自动的，隐含地创建的同名全局变量（如果不是“Strict模式”`note-strictmode`），或者一个 `ReferenceError`（如果是“Strict模式”`note-strictmode`）。

## 小测验答案

```
function foo(a) {  
    var b = a;  
    return a + b;  
}  
  
var c = foo( 2 );
```

1. 找出所有的LHS查询（有3处！）。

`c = ..`, `a = 2` (隐含的参数赋值) 和 `b = ..`

2. 找出所有的RHS查询（有4处！）。

`foo(2.., = a; , a + .. 和 .. + b`

note-strictmode . MDN: [Strict Mode ↪](#)

# 你不懂JS：作用域与闭包

## 第二章：词法作用域

在第一章中，我们将“作用域”定义为一组规则，它主宰着引擎如何通过标识符名称在当前的作用域，或者在包含它的任意嵌套作用域中来查询一个变量，

作用域的工作方式有两种占统治地位的模型。其中的第一种是最最常见，在绝大多数的编程语言中被使用的。它称为词法作用域，我们将深入检视它。另一种仍然被一些语言（比如Bash脚本，Perl中的一些模式，等等）使用的模型，称为动态作用域。

动态作用域在附录A中讲解。我在这里提到它仅仅是为词法作用域提供一个对比，而词法作用域是JavaScript采用的作用域模型。

### 词法分析时

正如我们在第一章中讨论的，标准语言编译器的第一个传统步骤称为词法分析（也就是分词）。如果你回忆一下，词法分析处理是检查一串源代码字符，并给token赋予语义含义作为某种有状态解析的输出。

正是这个概念给理解词法作用域是什么提供了基础，也是这个名词的渊源。

要定义它有点儿兜圈子，词法作用域是在词法分析时被定义的作用域。换句话说，词法作用域是基于，你，在写程序时，变量和作用域的块儿在何处被编写决定的，因此它在词法分析器处理你的代码时（基本上）是固定不变的。

注意：我们将会稍稍看到有一些方法可以骗过词法作用域，从而在词法分析器处理过后改变它，但是这些方法都是使人皱眉头的。事实上公认的最佳实践是，将词法作用域看作是仅仅依靠词法的，因此在本质上完全是编写时决定的。

让我们考虑这段代码：

```

function foo(a) {
  var b = a * 2;

  function bar(c) {
    console.log( a, b, c );
  }

  bar(b * 3);
}

foo( 2 ); // 2 4 12

```

在这个代码实例中有三个固有的嵌套作用域。将这些作用域考虑为套在一起的气泡可能有助于思考。

```

function foo(a) {
  var b = a * 2;
  function bar(c) {
    console.log( a, b, c );
  }
  bar(b * 3);
}

foo( 2 ); // 2, 4, 12

```

气泡1包围着全局作用域，它里面只有一个标识符：`foo`。

气泡2包围着作用域`foo`，它含有三个标识符：`a`，`bar`和`b`。

气泡3包围着作用域`bar`，它里面只包含一个标识符：`c`。

作用域气泡是根据作用域的块儿被写在何处定义的，一个嵌套在另一个内部，等等。在下一章中，我们将讨论作用域的不同单位，但是就现在来说，让我们认为每一个函数创建了一个新的作用域气泡。

`bar`的气泡完全被包含在`foo`的气泡中，因为（而且只因为）这就是我们选择定义函数`bar`的位置。

注意这些嵌套的气泡是严格嵌套的。我们没有讨论气泡可以跨越边界的维恩图（Venn diagrams）。换句话说，没有那个函数的气泡可以同时（部分地）存在于另外两个外部的作用域气泡中，就像没有函数可以部分地存在于它的两个父函数中一样。

## 查询

这些作用域气泡的结构和相对位置完全解释了引擎在查找一个标识符时，它需要查看的所有地方。

在上面的代码段中，引擎执行语句 `console.log(..)` 并开始查找三个被引用的变量 `a`，`b` 和 `c`。它首先从最内部的作用域气泡开始，也就是 `bar(..)` 函数的作用域。在这里它找不到 `a`，所以它向上走一层，到外面下一个最近的作用域气泡，`foo(..)` 的作用域。它在这里找到了 `a`，于是它就使用这个 `a`。同样的事情也发生在 `b` 上。但是对于 `c`，它在 `bar(..)` 内部就找到了。

如果在 `bar(..)` 内部和 `foo(..)` 内部都有一个 `c`，那么 `console.log(..)` 语句将会找到并使用 `bar(..)` 中的那一个，绝不会到达 `foo(..)` 中的那一个。

一旦找到第一个匹配，作用域查询就停止了。相同的标识符名称可以在嵌套作用域的多个层中被指定，这称为“遮蔽（shadowing）”（内部的标识符“遮蔽”了外部的标识符）。无论如何遮蔽，作用域查询总是从当前被执行的最内侧的作用域开始，向外/向上不断查找，直到第一个匹配才停止。

注意：全局变量也自动地是全局对象（在浏览器中是 `window`，等等）的属性，所以不直接通过全局变量的词法名称，而通过将它作为全局对象的一个属性引用来间接地引用，是可能的。

```
window.a
```

这种技术给出了访问全局变量的方法，没有它全局变量将因为被遮蔽而不可访问。然而，被遮蔽的非全局变量是无法访问的。

不管函数是从哪里被调用的，也不论它是如何被调用的，它的词法作用域是由这个函数被声明的位置唯一定义的。

词法作用域查询仅仅在处理头等标识符时实施，比如 `a`，`b`，和 `c`。如果你在一段代码中拥有一个 `foo.bar.baz` 的引用，词法作用域查询将在查找 `foo` 标识符时实施，但一旦定位这个变量，对象属性访问规则将会分别接管 `bar` 和 `baz` 属性的解析。

## 欺骗词法作用域

如果词法作用域仅仅是由函数被声明的位置定义的，而且这个位置完全是一个编写时的决定，那么怎么可能有办法在运行时“修改”（也就是，作弊欺骗）词法作用域呢？

JavaScript有两种这样的机制。在广大的社区中它们都等同地被认为是让人皱眉头的，在你代码中使用它们是一种差劲儿的做法。但是关于它们的具有代表性的争论经常错过了最重要的一点：欺骗词法作用域会导致更低下的性能。

在我讲解性能的问题以前，先让我们看看这两种机制是如何工作的。

## eval

JavaScript中的 `eval(..)` 函数接收一个字符串作为参数值，并将这个字符串的内容看作是好像它已经被实际编写在程序的那个位置上。换句话说，你可以用编程的方式在你编写好的代码内部生成代码，而且你可以运行这个生成的代码，就好像它在编写时就已经在那里了一样。

如果以这种观点来评价 `eval(..)`，那么 `eval(..)` 是如何允许你修改词法作用域环境应当是很清楚的：欺骗并假装这个编写时（也就是，词法）代码一直就在那里。

在 `eval(..)` 被执行的后续代码行中，引擎 将不会“知道”或“关心”前面的代码是被动态翻译的，而且因此修改了词法作用域环境。引擎 将会像它一直做的那样，简单地进行词法作用域查询。

考虑如下代码：

```
function foo(str, a) {
    eval(str); //作弊!
    console.log(a, b);
}

var b = 2;

foo("var b = 3;", 1); // 1 3
```

在 `eval(..)` 调用的位置上，字符串 `"var b = 3"` 被看作是一直就存在在那里的代码。因为这个代码恰巧声明了一个新的变量 `b`，它就修改了现存的 `foo(..)` 的词法作用域。事实上，就像上面提到的那样，这个代码实际上在 `foo(..)` 内部创建了变量 `b`，它遮蔽了声明在外部（全局）作用域中的 `b`。

当 `console.log(..)` 调用发生时，它会在 `foo(..)` 的作用域中找到 `a` 和 `b`，而且绝不会找到外部的 `b`。这样，我们就打印出"1 3"而不是一般情况下的"1, 2"。

注意：在这个例子中，为了简单起见，我们传入的“代码”字符串是固定的文字。但是它可以通过根据你的程序逻辑将字符拼接在一起，很容易地以编程方式创建。`eval(..)` 通常被用于执行动态创建的代码，因为动态地对一段实质上源自字符串字面值的静态代码进行求值，并不会比直接编写这样的代码带来更多真正的好处。

默认情况下，如果 `eval(..)` 执行的代码字符串包含一个或多个声明（变量或函数）的话，这个动作就会修改这个 `eval(..)` 所在的词法作用域。技术上讲，`eval(..)` 可以通过种种技巧（超出了我们这里的讨论范围）被“间接”调用，而使它在全局作用域的上下文中执行，如此修改全局作用域。但不论那种情况，`eval(..)` 都可以在运行时修改一个编写时的词法作用域。

注意：当 `eval(..)` 被用于一个操作它自己的词法作用域的 `strict` 模式程序时，在 `eval(..)` 内部做出的声明不会实际上修改包围它的作用域。

```
function foo(str) {
  "use strict";
  eval( str );
  console.log( a ); // ReferenceError: a is not defined
}

foo( "var a = 2" );
```

在 JavaScript 中还有其他的工具拥有与 `eval(..)` 非常类似的效果。`setTimeout(..)` 和 `setInterval(..)` 可以为它们各自的第一个参数值接收一个字符串，其内容将会被 `eval` 为一个动态生成的函数的代码。这种老旧的，遗产行为早就被废弃了。别这么做！

`new Function(..)` 函数构造器类似地为它的最后一个参数值接收一个代码字符串，来把它转换为一个动态生成的函数（前面的参数值，如果有的话，将作为新函数的命名参数）。这种函数构造器语法要比 `eval(..)` 稍稍安全一些，但在你的代码中它仍然应当被避免。

在你的代码中动态生成代码的用例少的不可思议，因为在性能上的倒退使得这种能力几乎总是得不偿失。

## with

JavaScript 的另一个使人皱眉头（而且现在被废弃了！），而且可以欺骗词法作用域的特性是 `with` 关键字。有许多种合法的方式可以讲解 `with`，但是我在此选择从它如何与词法作用域互动并影响词法作用域的角度来讲解它。

讲解 `with` 的典型方式是作为一种缩写，来引用一个对象的多个属性，而不必每次都重复对对象引用本身。

例如：

```

var obj = {
  a: 1,
  b: 2,
  c: 3
};

// 重复“obj”显得更“繁冗”
obj.a = 2;
obj.b = 3;
obj.c = 4;

// “更简单”的缩写
with (obj) {
  a = 3;
  b = 4;
  c = 5;
}

```

然而，这里发生的事情要比只是一个对象属性访问的便捷缩写要多得多。考虑如下代码：

```

function foo(obj) {
  with (obj) {
    a = 2;
  }
}

var o1 = {
  a: 3
};

var o2 = {
  b: 3
};

foo( o1 );
console.log( o1.a ); // 2

foo( o2 );
console.log( o2.a ); // undefined
console.log( a ); // 2 -- 哦，全局作用域被泄漏了！

```

在这个代码示例中，创建了两个对象 `o1` 和 `o2`。一个有 `a` 属性，而另一个没有。`foo(..)` 函数接收一个对象引用 `obj` 作为参数值，并在这个引用上调用 `with (obj) {}`。在 `with` 块儿内部，我们制造了一个变量 `a` 的看似是普通词法引用的东西，实际上是一个LHS引用（见第一章），并将值 `2` 赋予它。

当我们传入 `o1` 时，赋值 `a = 2` 找到属性 `o1.a` 并赋予它值 `2`，正如在后续的 `console.log(o1.a)` 语句反应的那样。然而，当我们传入 `o2`，因为它没有 `a` 属性，没有这样的属性被创建，所以 `o2.a` 还是 `undefined`。

但是之后我们注意到一个特别的副作用，赋值 `a = 2` 创建了一个全局变量 `a`。这怎么可能？

`with` 语句接收一个对象，这个对象有0个或多个属性，并将这个对象视为好像它是一个完全隔离的词法作用域，因此这个对象的属性被视为在这个“作用域”中词法定义的标识符。

注意：尽管一个 `with` 块儿将一个对象视为一个词法作用域，但是在 `with` 块儿内部的一个普通 `var` 声明将不会归于这个 `with` 块儿的作用域，而是归于包含它的函数作用域。

如果 `eval(..)` 函数接收一个含有一个或多个声明的代码字符串，它就会修改现存的词法作用域，而 `with` 语句实际上是从你传递给它的对象中凭空制造了一个全新的词法作用域。

以这种方式理解的话，当我们传入 `o1` 时 `with` 语句声明的“作用域”就是 `o1`，而且这个“作用域”拥有一个对应于 `o1.a` 属性的“标识符”。但当我们使用 `o2` 作为“作用域”时，它里面没有这样的 `a` “标识符”，于是LHS标识符查询（见第一章）的普通规则发生了。

“作用域” `o2` 中没有，`foo(..)` 的作用域中也没有，甚至连全局作用域中都没有找到标识符 `a`，所以当 `a = 2` 被执行时，其结果就是自动全局变量被创建（因为我们没有在strict模式下）。

`with` 在运行时将一个对象和它的属性转换为一个带有“标识符”的“作用域”，这个奇怪想法有些烧脑。但是对于我们看到的结果来说，这是我能给出的最清晰的解释。

注意：除了使用它们是个坏主意意外，`eval(..)` 和 `with` 都受Strict模式的影响（制约）。`with` 干脆就不允许使用，而虽然 `eval(..)` 还保有其核心功能，但各种间接形式的或不安全的 `eval(..)` 是不允许的。

## 性能

通过在运行时修改，或创建新的词法作用域，`eval(..)` 和 `with` 都可以欺骗编写时定义的词法作用域。

你可能会问，那又有什么大不了的？如果它们提供了更精巧的功能和编码灵活性，那它们不是好的特性吗？不。

JavaScript 引擎在编译阶段进行许多性能优化工作。其中的一些优化原理都归结为实质上在进行词法分析时可以静态地分析代码，并提前决定所有的变量和函数声明都在什么位置，这样在执行期间就可以少花些力气来解析标识符。

但如果引擎在代码中找到一个 `eval(..)` 或 `with`，它实质上就不得不假定自己知道的所有标识符的位置可能是不合法的，因为它不可能在词法分析时就知道你将会向 `eval(..)` 传递什么样的代码来修改词法作用域，或者你可能会向 `with` 传递的对象有什么样的内容来创建一个新的将被查询的词法作用域。

换句话说，悲观地看，如果 `eval(..)` 或 `with` 出现，那么它将做的几乎所有的优化都会变得没有意义，所以它就会简单地根本不做任何优化。

你的代码几乎肯定会趋于运行的更慢，只因为你在代码的任何地方引入了一个了 `eval(..)` 或 `with`。无论引擎将在努力限制这些悲观臆测的副作用上表现得多么聪明，都没有任何办法可以绕过这个事实：没有优化，代码就运行的更慢。

## 复习

词法作用域意味着作用域是由编写时函数被声明的位置的决策定义的。编译器的词法分析阶段实质上可以知道所有的标识符是在哪里和如何声明的，并如此在执行期间预测它们将如何被查询。

在JavaScript中有两种机制可以“欺骗”词法作用域：`eval(..)` 和 `with`。前者可以通过对一个拥有一个或多个声明的“代码”字符串进行求值，来（在运行时）修改现存的词法作用域。后者实质上是通过将一个对象引用看作一个“作用域”，并将这个对象的属性看作作用域中的标识符，（同样，也是在运行时）创建一个全新的词法作用域。

这些机制的缺点是，它压制了引擎在作用域查询上进行编译期优化的能力，因为引擎不得不悲观地假定这样的优化是不合法的。这两种特性的结果就是代码将会运行的更慢。不要使用它们。

# 你不懂JS：作用域与闭包

## 第三章：函数与块儿作用域

正如我们在第二章中探索的，作用域由一系列“气泡”组成，这些“气泡”的每一个就像一个容器或篮子，标识符（变量，函数）就在它里面被声明。这些气泡整齐地互相嵌套在一起，而且这种嵌套是在编写时定义的。

但是到底是什么才能制造一个新气泡？只能是函数吗？JavaScript中的其他结构可以创建作用域的气泡吗？

### 函数中的作用域

对这些问题的最常见的回答是，JavaScript拥有基于函数的作用域。也就是，你声明的每一个函数都为自己创建了一个气泡，而且没有其他的结构可以创建它们自己的作用域气泡。但是就像我们一会儿就会看到的，这不完全正确。

但首先，让我们探索一下函数作用域和它的含义。

考虑这段代码：

```
function foo(a) {
  var b = 2;

  // 一些代码

  function bar() {
    // ...
  }

  // 更多代码

  var c = 3;
}
```

在这个代码段中，`foo(..)` 的作用域气泡包含标识符 `a`，`b`，`c` 和 `bar`。一个声明出现在作用域何处是无关紧要的，不管怎样，变量和函数属于包含它们的作用域气泡。在下一章中我们将会探索这到底是如何工作的。

`bar(..)` 拥有它自己的作用域气泡。全局作用域也一样，它仅含有一个标识符：`foo`。

因为 `a`，`b`，`c`，和 `bar` 都属于 `foo(..)` 的作用域气泡，所以它们在 `foo(..)` 外部是不可访问的。也就是，接下来的代码都会得到 `ReferenceError` 错误，因为这些标识符在全局作用域中都不可用：

```
bar(); // 失败

console.log( a, b, c ); // 3个都失败
```

然而，所有这些标识符（`a`，`b`，`c`，和 `bar`）在 `foo(..)` 内部都是可以访问的，而且在 `bar(..)` 内部实际上也都是可用的（假定在 `bar(..)` 内部没有遮蔽标识符的声明）。

函数作用域支持着这样的想法：所有变量都属于函数，而且贯穿整个函数始终都可以使用和重用（而且甚至可以在嵌套的作用域中访问）。这种设计方式可以十分有用，而且肯定可以完全利用JavaScript的“动态”性质——变量可以根据需要接受不同种类型的值。

另一方面，如果你不小心提防，跨越整个作用域存在的变量可能会导致一些以外的陷阱。

## 隐藏于普通作用域

考虑一个函数的传统方式是，你声明一个函数，并在它内部添加代码。但是相反的想法也同样强大和有用：拿你所编写的代码的任意一部分，在它周围包装一个函数声明，这实质上“隐藏”了这段代码。

其实际结果是在这段代码周围创建了一个作用域气泡，这意味着现在在这段代码中的任何声明都将绑在这个新的包装函数的作用域上，而不是前一个包含它们的作用域。换句话说，你可以通过将变量和函数围在一个函数的作用域中来“隐藏”它们。

为什么“隐藏”变量和函数是一种有用的技术？

有各种原因驱使着这种基于作用域的隐藏。它们主要是由一种称为“最低权限原则”的软件设计原则引起的 [note-leastprivilege](#)，有时也被称为“最低授权”或“最少曝光”。这个原则规定，在软件设计中，比如一个模块/对象的API，你应当只暴露所需要的最低限度的东西，而“隐藏”其他的一切。

这个原则可以扩展到用哪个作用域来包含变量和函数的选择。如果所有的变量和函数在全局作用域中，它们将理所当然地对任何嵌套的作用域是可访问的。但这回违背“最少……”原则，因为你（很可能）暴露了许多你本应当保持为私有的变量和函数，而这些代码的恰当用法是不鼓励访问这些变量/函数的。

例如：

```

function doSomething(a) {
    b = a + doSomethingElse( a * 2 );
    console.log( b * 3 );
}

function doSomethingElse(a) {
    return a - 1;
}

var b;

doSomething( 2 ); // 15

```

在这个代码段中，变量 `b` 和函数 `doSomethingElse(..)` 很可能是 `doSomething(..)` 如何工作的“私有”细节。允许外围的作用域“访问”`b` 和 `doSomethingElse(..)` 不仅没必要而且可能是“危险的”，因为它们可能会以种种意外的方式，有意或无意地被使用，而这也许违背了 `doSomething(..)` 假设的前提条件。

一个更“恰当”的设计是讲这些私有细节隐藏在 `doSomething(..)` 的作用域内部，比如：

```

function doSomething(a) {
    function doSomethingElse(a) {
        return a - 1;
    }

    var b;

    b = a + doSomethingElse( a * 2 );
    console.log( b * 3 );
}

doSomething( 2 ); // 15

```

现在，`b` 和 `doSomethingElse(..)` 对任何外界影响都是不可访问的，而是仅仅由 `doSomething(..)` 控制。它的功能和最终结果不受影响，但是这种设计将私有细节保持为私有的，这通常被认为是好的软件。

## 避免冲突

将变量和函数“隐藏”在一个作用域内部的另一个好处是，避免两个同名但用处不同的标识符之间发生无意的冲突。冲突经常导致值被意外地覆盖。

例如：

```

function foo() {
  function bar(a) {
    i = 3; // 在外围的for循环的作用域中改变`i`
    console.log( a + i );
  }

  for (var i=0; i<10; i++) {
    bar( i * 2 ); // 噢，无限循环！
  }
}

foo();

```

`bar(..)` 内部的赋值 `i = 3` 意外地覆盖了在 `foo(..)` 的`for`循环中声明的 `i`。在这个例子中，这将导致一个无限循环，因为 `i` 被设定为固定的值 `3`，而它将永远 `< 10`。

`bar(..)` 内部的赋值需要声明一个本地变量来使用，不论选用什么样的标识符名称。`var i = 3;` 将修复这个问题（并将为 `i` 创建一个前面提到的“遮蔽变量”声明）。一个另外的，不是代替的，选项是完全选择另外一个标识符名称，比如 `var j = 3;`。但是你的软件设计也许会自然而然地使用相同的标识符名称，所以在这种情况下利用作用域来“隐藏”你的内部声明是你最好/唯一的选择。

## 全局“名称空间”

变量冲突（可能）发生的一个特别强有力的例子是在全局作用域中。多个库被加载到你的程序中时可以十分容易地互相冲突，如果它们没有适当地隐藏它们的内部/私有函数和变量。

这样的库通常会在全局作用域中使用一个足够独特的名称来创建一个单独的变量声明，它经常是一个对象。然后这个对象被用作这个库的一个“名称空间”，所有要明确暴露出来的功能都被作为属性挂在这个对象（名称空间）上，而不是将它们自身作为顶层词法作用域的标识符。

例如：

```

var MyReallyCoolLibrary = {
  awesome: "stuff",
  doSomething: function() {
    // ...
  },
  doAnotherThing: function() {
    // ...
  }
};

```

## 模块管理

另一种回避冲突的选择是更加现代的“模块”方式，它使用任意一种依赖管理器。使用这些工具，没有库可以向全局作用域添加任何标识符，取而代之的是使用依赖管理器的各种机制，要求库的标识符被明确地导入到另一个指定的作用域中。

应该可以看到，这些工具并不拥有可以豁免于词法作用域规则的“魔法”功能。它们简单地使用这里讲解的作用域规则，来强制标识符不会被注入任何共享的作用域，而是保持在私有的，不易冲突的作用域中，这防止了任何意外的作用域冲突。

因此，如果你选择这样做的话，你可以防御性地编码，并在实际上不使用依赖管理器的情况下，取得与使用它们相同的结果。关于模块模式的更多信息参见第五章。

## 函数作为作用域

我们已经看到，我们可以拿来一段代码并在它周围包装一个函数，而这实质上对外部作用域“隐藏”了这个函数内部作用域包含的任何变量或函数声明。

例如：

```
var a = 2;

function foo() { // <-- 插入这个

    var a = 3;
    console.log(a); // 3

} // <-- 和这个
foo(); // <-- 还有这个

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

虽然这种技术“可以工作”，但它不一定非常理想。它引入了几个问题。首先是我们不得不声明一个命名函数 `foo()`，这意味着这个标识符名称 `foo` 本身就“污染”了外围作用域（在这个例子中是全局）。我们要不得不通过名称（`foo()`）明确地调用这个函数来使被包装的代码真正运行。

如果这个函数不需要名称（或者，这个名称不污染外围作用域），而且如果这个函数能自动地被执行就更理想了。

幸运的是，JavaScript给这两个问题提供了一个解决方法。

```

var a = 2;

(function foo(){ // <-- 插入这个

    var a = 3;
    console.log( a ); // 3

})(); // <-- 和这个

console.log( a ); // 2

```

让我们分析一下这里发生了什么。

首先注意，与仅仅是 `function...` 相对，这个包装函数语句以 `(function...)` 开头。虽然这看起来像是一个微小的细节，但实际上这是重大改变。与将这个函数视为一个标准的声明不同的是，这个函数被视为一个函数表达式。

注意：区分声明与表达式的最简单的方法是，这个语句中（不仅仅是一行，而是一个独立的语句）“`function`”一词的位置。如果“`function`”是这个语句中的第一个东西，那么它就是一个函数声明。否则，它就是一个函数表达式。

这里我们可以观察到一个函数声明和一个函数表达式之间的关键不同是，它的名称作为一个标识符被绑定在何处。

比较这两个代码段。在第一个代码段中，名称 `foo` 被绑定在外围作用域中，我们用 `foo()` 直接调用它。在第二个代码段中，名称 `foo` 没有被绑定在外围作用域中，而是被绑定在它自己的函数内部。

换句话说，`(function foo(){ .. })` 作为一个表达式意味着标识符 `foo` 仅能在 `..` 代表的作用域中被找到，而不是在外部作用域中。将名称 `foo` 隐藏在它自己内部意味着它不会多余地污染外围作用域。

## 匿名与命名

你可能对函数表达式作为回调参数再熟悉不过了，比如：

```

setTimeout( function(){
    console.log("I waited 1 second!");
}, 1000 );

```

这称为一个“匿名函数表达式”，因为 `function()...` 上没有名称标识符。函数表达式可以是匿名的，但是函数声明不能省略名称——那将是不合法的JS程序。

匿名函数表达式可以快速和很容易地键入，而且许多库和工具往往鼓励使用这种代码惯用风格。然而，它们有几个缺点需要考虑：

1. 在栈轨迹上匿名函数没有有用的名字可以表示，这使得调试更加困难。
2. 没有名称的情况下，如果这个函数需要为了递归等目的引用它自己，那么就需要很不幸地使用被废弃的 `arguments.callee` 引用。另一个需要自引用的例子是，当一个事件处理器函数在被触发后想要把自己解除绑定。
3. 匿名函数省略的名字经常对提供更易读/易懂的代码很有帮助。一个描述性的名字可以帮助代码自解释。

内联函数表达式很强大且很有用——匿名和命名的问题并不会贬损这一点。给你的函数表达式提供一个名称就可以十分有效地解决这些缺陷，而且没有实际的坏处。最佳的方法是总是命名你的函数表达式：

```
setTimeout( function timeoutHandler(){ // <-- 看，我有一个名字！
  console.log( "I waited 1 second!" );
}, 1000 );
```

## 立即调用函数表达式

```
var a = 2;

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

})();

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

得益于包装在一个 `()` 中，我们有了一个作为表达式的函数，我们可以通过在末尾加入另一个 `()` 来执行这个函数，就像 `(function foo(){ .. })()`。第一个外围的 `()` 使这个函数变成表达式，而第二个 `()` 执行这个函数。

这个模式是如此常见，以至于几年前开发者社区同意给它一个术语：**IIFE**，它表示“立即被调用的函数表达式”（Immediately Invoked Function Expression）。

当然，IIFE不一定需要一个名称——IIFE的最常见形式是使用一个匿名函数表达式。虽然少见一些，与匿名函数表达式相比，命名的IIFE拥有前述所有的好处，所以它是一个可以采用的好方式。

```

var a = 2;

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

})();

console.log( a ); // 2

```

传统的IIFE有一种稍稍变化的形式，一些人偏好：`(function(){ .. }())`。仔细观察不同之处。在第一种形式中，函数表达式被包在`( )`中，然后用于调用的`()`出现在它的外侧。在第二种形式中，用于调用的`()`被移动到用于包装的`( )`内侧。

这两种形式在功能上完全相同。这纯粹是一个你偏好的风格的选择。

IIFE的另一种十分常见的变种是，利用它们实际上只是函数调用的事实，来传入参数值。

例如：

```

var a = 2;

(function IIFE( global ){
    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

})( window );

console.log( a ); // 2

```

我们传入`window`对象引用，但是我们将参数命名为`global`，这样我们对于全局和非全局引用就有了一个清晰的文体上的划分。当然，你可以从外围作用域传入任何你想要的东西，而且你可以将参数命名为任何适合你的名称。这几乎仅仅是文体上的选择。

这种模式的另一种应用解决了一个小问题：默认的`undefined`标识符的值也许会被不正确地覆盖掉，而导致意外的结果。通过将参数命名为`undefined`，同时不为它传递任何参数值，我们就可以保证在一个代码块中`undefined`标识符确实是是一个未定义的值。

```
undefined = true; // 给其他的代码埋地雷！别这么干！

(function IIFE( undefined ){
    var a;
    if (a === undefined) {
        console.log( "Undefined is safe here!" );
    }
})();
```

IIFE还有另一种变种将事情的顺序倒了过来，要被执行的函数在调用和传递给它的参数之后给出。这种模式被用于UMD（Universal Module Definition——统一模块定义）项目。一些人发现它更干净和移动一些，虽然有点儿繁冗。

```
var a = 2;

(function IIFE( def ){
    def( window );
})(function def( global ){
    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2
});
```

`def` 函数表达式带这个代码段的后半部分被定义，然后作为一个参数（也叫 `def`）被传递给在代码段前半部分定义的 `IIFE` 函数。最后，参数 `def`（函数）被调用，并将 `window` 作为 `global` 参数传入。

## 块儿作为作用域

虽然函数是最常见的作用域单位，而且当然也是在市面上流通的绝大多数JS中最为广泛传播的设计方式，但是其他的作用域单位也是可能的，而且使用这些作用域单位可以导致更好，对于维护来说更干净的代码。

JavaScript之外的许多其他语言都支持块儿作用域，所以有这些语言背景的开发者习惯于这种思维模式，然而那些主要在JavaScript中工作的开发者可能会发现这个概念有些陌生。

但即使你从没用块儿作用域的方式写过一行代码，你可能依然对JavaScript中这种极其常见的惯用法很熟悉：

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

我们在`for`循环头的内部直接声明了变量 `i`，因为我们意图很可能是仅在这个`for`循环内部的上下文环境中使用 `i`，而实质上忽略了这个变量实际上将自己划入了外围作用域中（函数或全局）的事实。

这就是有关块儿作用域的一切。尽可能封闭地，尽可能局部地，在变量将被使用的位置声明它。另一个例子是：

```
var foo = true;

if (foo) {
    var bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}
```

我们仅在`if`语句的上下文环境中使用变量 `bar`，所以我们将它声明在`if`块儿的内部是有些道理的。然而，当使用 `var` 时，我们在何处声明变量是无关紧要的，因为它们将总是属于外围作用域。这个代码段实质上为了代码风格的原因“假冒”了块儿作用域，并依赖于我们要管好自己，不要在这个作用域的其他地方意外地使用 `bar`。

从将信息隐藏在函数中到将信息隐藏在我们代码的块儿中，块儿作用域是一种扩展了早先的“最低权限暴露原则”[note-leastprivilege](#) 的工具。

再次考虑这个`for`循环的例子：

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

为什么要用仅将（或者至少是，仅应当）在这个`for`循环中使用的变量 `i` 去污染一个函数的整个作用域呢？

但更重要的是，开发者们也许偏好于检查他们自己来防止在变量预期的目的之外意外地（重）使用它们，例如如果你试着在错误的地方使用变量会导致一个未知变量的错误。对于变量 `i` 的块儿作用域（如果它是可能的话）将使 `i` 仅在`for`循环内部可用，使得如果在函数的其他地方访问 `i` 将导致一个错误。这有助于保证变量不会被糊涂地重用或者难于维护。

但是，悲惨的现实是，表面上看来，JavaScript没有块儿作用域的能力。

更确切地说，直到你再深入一些才有。

## with

我们在第二章中学习了 `with`。虽然它是一个使人皱眉头的结构，但它确实是一个（一种形式的）块儿作用域的例子，它从对象中创建的作用域仅存在于这个 `with` 语句的生命周期中，而不再外围作用域中。

## try/catch

一个鲜为人知的事实，JavaScript在ES3中明确指出在 `try/catch` 的 `catch` 子句中声明的变量，是属于 `catch` 块儿的块儿作用域的。

例如：

```
try {
    undefined(); //用非法的操作强制产生一个异常！
}
catch (err) {
    console.log( err ); // 好用！
}

console.log( err ); // ReferenceError: `err` not found
```

如你所见，`err` 仅存在于 `catch` 子句中，并且在你试着从其他地方引用他时抛出一个错误。

注意： 虽然这种行为已经被明确规定，而且对于几乎所有的标准JS环境（也许除了老IE）来说都是成立的，但是如果你在同一个作用域中有两个或多个 `catch` 子句，而它们又各自用相同的标识符名称声明了它们表示错误的变量时，许多linter依然会报警。实际上这不是重定义，因为这些变量都安全地位于块儿作用域中，但是linter看起来依然，恼人地，抱怨这个事实。

为了避免这些不必要的警告，一些开发者将他们的 `catch` 变量命名为 `err1`，`err2`，等等。另一些开发者干脆关闭linter对重复变量名的检查。

`catch` 的块儿作用域性质看起来像是一个没用的，只有学院派意义的事实，但是参看附录B来了解更多它如何有用的信息。

## let

至此，我们看到JavaScript仅仅有一些奇怪的小众行为暴露了块儿作用域功能。如果这就是我们拥有的一切，而且许多许多年以来这确实就是我们拥有的一切，那么块作用域对JavaScript开发者来说就不是非常有用。

幸运的是，ES6改变了这种状态，并引入了一个新的关键字 `let`，作为另一种声明变量的方式伴随着 `var`。

`let` 关键字将变量声明附着在它所在的任何块儿（通常是一个`{ ... }`）的作用域中。换句话说，`let` 为它的变量声明隐含地劫持了任意块儿的作用域。

```
var foo = true;

if (foo) {
  let bar = foo * 2;
  bar = something( bar );
  console.log( bar );
}

console.log( bar ); // ReferenceError
```

使用 `let` 将一个变量附着在一个现存的块儿上有些隐晦。它可能会使人困惑——在你开发和设计代码时，如果你不仔细注意哪些块儿的作用域包含了变量，并且习惯于将块儿四处移动，将它们包进其他的块儿中，等等。

为块儿作用域创建明确的块儿可以解决这些问题中的一些，使变量附着在何处更加明显。通常来说，明确的代码要比隐晦或微妙的代码好。这种明确的块儿作用域风格很容易达成，而且它与块儿作用域在其他语言中的工作方式匹配得更自然：

```
var foo = true;

if (foo) {
  { // <-- 明确的块儿
    let bar = foo * 2;
    bar = something( bar );
    console.log( bar );
  }
}

console.log( bar ); // ReferenceError
```

我们可以在一个语句是合法文法的任何地方，通过简单地引入一个`{ ... }` 来为 `let` 创建一个任意的可以绑定的块儿。在这个例子中，我们在`if`语句内部制造了一个明确的块儿，在以后的重构中将整个块儿四处移动可能会更容易，而且不会影响外围的`if`语句的位置和语义。

注意：另一个明确表达块儿作用域的方法，参见附录B。

在第四章中，我们将讲解提升（hoisting），它讲述关于声明在它们所出现的整个作用域中都被认为是存在的。

然而，使用 `let` 做出的声明将不会在它们所出现的整个块儿的作用域中提升。如此，直到声明语句为止，声明将不会“存在”于块儿中。

```
{
  console.log( bar ); // ReferenceError!
  let bar = 2;
}
```

## 垃圾回收

块儿作用域的另一个有用之处是关于闭包和释放内存的垃圾回收。我们将简单地在这里展示一下，但是闭包机制将在第五章中详细讲解。

考虑这段代码：

```
function process(data) {
  // 做些有趣的事
}

var someReallyBigData = { .. };

process( someReallyBigData );

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
  console.log("button clicked");
}, /*capturingPhase=*/false );
```

点击事件的处理器回调函数 `click` 根本不需要 `someReallyBigData` 变量。这意味着从理论上讲，在 `process(..)` 运行之后，这个消耗巨大内存的数据结构可以被作为垃圾回收。然而，JS引擎很可能（虽然这要看具体实现）将会仍然将这个结构保持一段时间，因为 `click` 函数在整个作用域上拥有一个闭包。

块儿作用域可以解决这个问题，使引擎清楚地知道它不必再保持 `someReallyBigData` 了：

```

function process(data) {
    // 做些有趣的事
}

// 运行过后，任何定义在这个块中的东西都可以消失了
{
    let someReallyBigData = { ... };

    process( someReallyBigData );
}

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );

```

声明可以将变量绑定在本地的明确的块儿是一种强大的工具，你可以把它加入你的工具箱。

## let 循环

一个使 `let` 闪光的特殊例子是我们先前讨论的`for`循环。

```

for (let i=0; i<10; i++) {
    console.log( i );
}

console.log( i ); // ReferenceError

```

在`for`循环头部的 `let` 不仅将 `i` 绑定在`for`循环体中，而且实际上，它会对每一次循环的迭代重新绑定 `i`，确保它被赋予来自上一次循环迭代末尾的值。

这是描绘这种为每次迭代进行绑定的行为的另一种方式：

```

{
    let j;
    for (j=0; j<10; j++) {
        let i = j; // 每次迭代都重新绑定
        console.log( i );
    }
}

```

这种为每次迭代进行的绑定有趣的原因将在第五章中我们讨论闭包时变得明朗。

因为 `let` 声明附着于任意的块儿，而不是外围的函数作用域（或全局），所以在重构代码时可能会有一些坑需要额外小心：现存的代码拥有对函数作用域的 `var` 声明有隐藏的依赖，但你想要用 `let` 来取代 `var`。

考虑如下代码：

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;

    if (baz > bar) {
        console.log( baz );
    }

    // ...
}
```

这段代码可以相当容易地重构为：

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;

    // ...
}

if (baz > bar) {
    console.log( baz );
}
```

但是，当使用块儿作用域变量时要小心这样的变化：

```
var foo = true, baz = 10;

if (foo) {
    let bar = 3;

    if (baz > bar) { // <-- 移动时不要忘了`bar`
        console.log( baz );
    }
}
```

附录B介绍了一种块作用域的（更加明确的）替代形式，它可能会在这些场景下提供更易于维护/重构的更健壮的代码。

## const

除了 `let` 之外，ES6还引入了 `const`，它也创建一个块儿作用域变量，但是它的值是固定的（常量）。任何稍后改变它的企图都将导致错误。

```
var foo = true;

if (foo) {
    var a = 2;
    const b = 3; // 存在于包含它的`if`作用域中

    a = 3; // 没问题！
    b = 4; // 错误！

}

console.log(a); // 3
console.log(b); // ReferenceError!
```

## 复习

在JavaScript中函数是最常见的作用域单位。在另一个函数内部声明的变量和函数，实质上对任何外围“作用域”都是“隐藏的”，这是优秀软件的一个有意的设计原则。

但是函数绝不是唯一的作用域单位。块儿作用域指的是这样一种想法：变量和函数可以属于任意代码块儿（一般来说，就是任意的`{ .. }`），而不是仅属于外围的函数。

从ES3开始，`try/catch` 结构在 `catch` 子句上拥有块儿作用域。

在ES6中，引入了 `let` 关键字（`var` 关键字的表兄弟）允许在任意代码块中声明变量。`if(..){ let a = 2; }` 将会声明变量 `a`，而它实质上劫持了 `if` 的`{ .. }` 块儿的作用域，并将自己附着在这里。

虽然有些人对此深信不疑，但是块儿作用域不应当被认为是 `var` 函数作用域的一个彻头彻尾的替代品。两种机能共存的，而且开发者们可以并且应当同时使用函数作用域和块儿作用域技术——在它们各自可以产生更好，更易读/易维护代码的地方。

note-leastprivilege. [Principle of Least Privilege](#) ←

# 你不懂JS：作用域与闭包

## 第四章：提升

至此，你应当对作用域的想法，以及变量如何根据它们被声明的方式和位置附着在不同的作用域层级上感到相当适应了。函数作用域和块儿作用域的行为都是依赖于这个相同规则的：在一个作用域中声明的任何变量都附着在这个作用域上。

但是关于出现在一个作用域内各种位置的声明如何附着在作用域上，有一个微妙的细节，而这个细节正是我们要在这里检视的。

### 先有鸡还是先有蛋？

有一种倾向认为你在JavaScript程序中看到的所有代码，在程序执行的过程中都是从上到下一行一行地被解释执行的。虽然这大致上是对的，但是这种猜测中的一个部分可能会导致你错误地考虑你的程序。

考虑这段代码：

```
a = 2;  
  
var a;  
  
console.log( a );
```

你觉得在 `console.log(..)` 语句中会打印出什么？

许多开发者会期望 `undefined`，因为语句 `var a` 出现在 `a = 2` 之后，这很自然地看起来像是这个变量被重定义了，并因此被赋予了默认的 `undefined`。然而，输出将是 `2`。

考虑另一个代码段：

```
console.log( a );  
  
var a = 2;
```

你可能会被诱导而这样认为，因为上一个代码段展示了一种看起来不是从上到下的行为，也许在这个代码段中，也会打印 `2`。另一些人认为，因为变量 `a` 在它被声明之前就被使用了，所以这一定会导致一个 `ReferenceError` 被抛出。

不幸的是，两种猜测都不正确。输出是 `undefined`。

那么。这里发生了什么？看起来我们遇到了一个先有鸡还是先有蛋的问题。哪一个现有？声明（“蛋”），还是赋值（“鸡”）？

## 编译器再次袭来

要回答这个问题，我们需要回头引用第一章关于编译器的讨论。回忆一下，引擎实际上将会在它解释执行你的JavaScript代码之前编译它。编译过程的一部分就是找到所有的声明，并将它们关联在合适的作用域上。第二章向我们展示了这是词法作用域的核心。

所以，考虑这件事情的最佳方式是，在你的代码的任何部分被执行之前，所有的声明，变量和函数，都会首先被处理。

当你看到 `var a = 2;` 时，你可能认为这是一个语句。但是JavaScript实际上认为这是两个语句：`var a;` 和 `a = 2;`。第一个语句，声明，是在编译阶段被处理的。第二个语句，赋值，为了执行阶段而留在原处。

于是我们的第一个代码段应当被认为是这样被处理的：

```
var a;  
  
a = 2;  
  
console.log( a );
```

.....这里的一部分是编译，而第二部分是执行。

相似地，我们的第二个代码段实际上被处理为：

```
var a;  
  
console.log( a );  
  
a = 2;
```

所以，关于这种处理的一个有些隐喻的考虑方式是，变量和函数声明被从它们在代码流中出现的位置“移动”到代码的顶端。这就产生了“提升”这个名字。

换句话说，先有蛋（声明），后有鸡（赋值）。

注意：只有声明本身被提升了，而任何赋值或者其他执行逻辑都被留在原处。如果提升会重新安排我们代码的可执行逻辑，那就会是一场灾难了。

```
foo();  
  
function foo() {  
    console.log( a ); // undefined  
  
    var a = 2;  
}
```

函数 `foo` 的声明（在这个例子中它还包含一个隐含的，实际为函数的值）被提升了，因此第一行的调用是可以执行的。

还需要注意的是，提升是以作用域为单位的。所以虽然我们的前一个代码段被简化为仅含有全局作用域，但是我们现在检视的函数 `foo(..)` 本身展示了，`var a` 被提升至 `foo(..)` 的顶端（很明显，不是程序的顶端）。所以这个程序也许可以更准确地解释为：

```
function foo() {  
    var a;  
  
    console.log( a ); // undefined  
  
    a = 2;  
}  
  
foo();
```

函数声明会被提升，就像我们看到的。但是函数表达式不会。

```
foo(); // 不是 ReferenceError，而是 TypeError!  
  
var foo = function bar() {  
    // ...  
};
```

变量标识符 `foo` 被提升并被附着在这个程序的外围作用域（全局），所以 `foo()` 不会作为一个 `ReferenceError` 而失败。但 `foo` 还没有值（如果它不是函数表达式，而是一个函数声明，那么它就会有值）。所以，`foo()` 就是试图调用一个 `undefined` 值，这是一个 `TypeError` 非法操作。

同时回想一下，即使它是一个命名的函数表达式，这个名称标识符在周围作用域中也是不可用的：

```
foo(); // TypeError  
bar(); // ReferenceError  
  
var foo = function bar() {  
    // ...  
};
```

这个代码段可以（使用提升）更准确地解释为：

```
var foo;  
  
foo(); // TypeError  
bar(); // ReferenceError  
  
foo = function() {  
    var bar = ...self...  
    // ...  
}
```

## 函数优先

函数声明和变量声明都会被提升。但一个微妙的细节（可以在拥有多个“重复的”声明的代码中出现）是，函数会首先被提升，然后才是变量。

考虑这段代码：

```
foo(); // 1  
  
var foo;  
  
function foo() {  
    console.log( 1 );  
}  
  
foo = function() {  
    console.log( 2 );  
};
```

`1` 被打印了，而不是 `2`！这个代码段被引擎解释执行为：

```
function foo() {
    console.log( 1 );
}

foo(); // 1

foo = function() {
    console.log( 2 );
};
```

注意那个 `var foo` 是一个重复（因此被无视）的声明，即便它出现在 `function foo()...` 声明之前，因为函数声明是在普通变量之前被提升的。

虽然多个/重复的 `var` 声明实质上是被忽略的，但是后续的函数声明确实会覆盖前一个。

```
foo(); // 3

function foo() {
    console.log( 1 );
}

var foo = function() {
    console.log( 2 );
};

function foo() {
    console.log( 3 );
}
```

虽然这一切听起来不过是一些学院派的细节，但是它表明了一个事实：在同一个作用域内的重复定义是一个十分差劲儿的主意，而且经常会导致令人困惑的结果。

在普通的块儿内部出现的函数声明一般会被提升至外围的作用域，而不是像这段代码暗示的那样有条件地被定义：

```
foo(); // "b"

var a = true;
if (a) {
    function foo() { console.log( "a" ); }
}
else {
    function foo() { console.log( "b" ); }
}
```

然而，重要的是要注意这种行为是不可靠的，而且是未来版本的JavaScript将要改变的对象，所以避免在块儿中声明函数可能是最好的做法。

## 复习

我们可能被诱导而将 `var a = 2` 看作是一个语句，但是JavaScript引擎可不这么看。它将 `var a` 和 `a = 2` 看作两个分离的语句，第一个是编译期的任务，而第二个是执行时的任务。

这将导致在一个作用域内的所有声明，不论它们出现在何处，都会在代码本身被执行前首先被处理。你可以将它可视化为声明（变量与函数）被“移动”到它们各自的作用域顶部，这就是我们所说的“提升”。

声明本身会被提升，但不是赋值，即便是函数表达式的赋值，也不会被提升。

要小心重复声明，特别是将一般的变量声明和函数声明混在一起——如果你这么做的话，危险就在眼前！

# 你不懂**JS**：作用域与闭包

## 第五章：作用域闭包

希望我们是带着对作用域工作方式的健全，坚实的理解来到这里的。

我们将我们的注意力转向这个语言中一个重要到不可思议，但是一直难以捉摸的，几乎是神话般的部分：闭包。如果你至此一直跟随着我们关于词法作用域的讨论，那么你会感觉闭包将在很大程度上没那么令人激动，几乎是显而易见的。有一个魔法师坐在幕后，现在我们即将见到他。不，他的名字不是Crockford！

如果你还对词法作用域感到不安，那么现在就是在继续之前回过头去再复习一下第二章的好时机。

### 启示

对于那些对JavaScript有些经验，但是也许从没全面掌握闭包概念的人来说，理解闭包看起来就像是必须努力并作出牺牲才能到达的涅槃状态。

回想几年前我对JavaScript有了牢固的掌握，但是不知道闭包是什么。它暗示着这种语言有着另外的一面，它许诺了甚至比我已经拥有的还多的力量，它取笑并嘲弄我。我记得我通读早期框架的源代码试图搞懂它到底是如何工作的。我记得第一次“模块模式”的某些东西融入我的大脑。我记得那依然栩栩如生的啊哈！一刻。

那时我不明白的东西，那个花了我好几年时间才搞懂的东西，那个我即将传授给你的东西，是这个秘密：在**JavaScript**中闭包无所不在，你只是必须认出它并接纳它。闭包不是你必须学习新的语法和模式才能使用的特殊的可选的工具。不，闭包甚至不是你必须像卢克在原力中修炼那样，一定要学会使用并掌握的武器。

闭包是依赖于词法作用域编写代码而产生的结果。它们就这么发生了。要利用它们你甚至不需要有意地创建闭包。闭包在你的代码中一直在被创建和使用。你缺少的是恰当的思维环境，来识别，接纳，并以自己的意志利用闭包。

启蒙的时刻应该是：哦，闭包已经在我的代码中到处发生了，现在我终于看到它们了。理解闭包就像是尼欧第一次见到母体。

### 事实真相

好了，夸张和对电影的无耻引用够多了。

为了理解和识别闭包，这里有一个你需要知道的简单粗暴的定义：

闭包就是函数能够记住并访问它的词法作用域，即使当这个函数在它的词法作用域之外执行时。

让我们跳进代码来说明这个定义：

```
function foo() {
  var a = 2;

  function bar() {
    console.log( a ); // 2
  }

  bar();
}

foo();
```

根据我们对嵌套作用域的讨论，这段代码应当看起来很熟悉。由于词法作用域查询规则（在这个例子中，是一个RHS引用查询），函数 `bar()` 可以访问外围作用域的变量 `a`。

这是“闭包”吗？

好吧，技术上……也许是。但是根据我们上面的“你需要知道”的定义……不确切。我认为解释 `bar()` 引用 `a` 的最准确的方式是根据词法作用域查询规则，但是那些规则仅仅是闭包的一个很重要的！一部分。

从纯粹的学院派角度讲，上面的代码段被认为是函数 `bar()` 在函数 `foo()` 的作用域上有一个闭包（而且实际上，它甚至对其他的作用域也可以访问，比如这个例子中的全局作用域）。换一种略有不同的说法是，`bar()` 闭住了 `foo()` 的作用域。为什么？因为 `bar()` 嵌套地出现在 `foo()` 内部。简单直白。

但是，这样一来闭包的定义就是不能直接观察到的了，我们也不能看到闭包在这个代码段中被行使。我们清楚地看到词法作用域，但是闭包仍然像代码后面谜一般的模糊阴影。

让我们考虑这段将闭包完全照亮的代码：

```

function foo() {
  var a = 2;

  function bar() {
    console.log( a );
  }

  return bar;
}

var baz = foo();

baz(); // 2 -- 哇噢，看到闭包了，伙计。

```

函数 `bar()` 对于 `foo()` 内的作用域拥有词法作用域访问权。但是之后，我们拿起 `bar()`，这个函数本身，将它像值一样传递。在这个例子中，我们 `return``bar` 引用的函数对象本身。

在执行 `foo()` 之后，我们将它返回的值（我们里面的 `bar()` 函数）赋予一个称为 `baz` 的变量，然后我们实际地调用 `baz()`，这将理所当然地调用我们内部的函数 `bar()`，只不过是通过一个不同的标识符引用。

`bar()` 被执行了，必然的。但是在这个例子中，它是在它被声明的词法作用域 外部 被执行的。

`foo()` 被执行之后，一般说来我们会期望 `foo()` 的整个内部作用域都将消失，因为我们知道引擎启用了垃圾回收器 在内存不再被使用时来回收它们。因为很显然 `foo()` 的内容不再被使用了，所以看起来它们很自然地应该被认为是消失了。

但是闭包的“魔法”不会让这发生。内部的作用域实际上依然“在使用”，因此将不会消失。谁在使用它？函数 `bar()` 本身。

有赖于它被声明的位置，`bar()` 拥有一个词法作用域闭包覆盖着 `foo()` 的内部作用域，闭包为了能使 `bar()` 在以后任意的时刻可以引用这个作用域而保持它的存在。

`bar()` 依然拥有对那个作用域的引用，而这个引用称为闭包。

所以，在几微妙之后，当变量 `baz` 被调用时（调用我们最开始标记为 `bar` 的内部函数），它理所当地对编写时的词法作用域拥有访问权，所以它可以如我们所愿地访问变量 `a`。

这个函数在它被编写时的词法作用域之外被调用。闭包使这个函数可以继续访问它在编写时被定义的词法作用域。

当然，函数可以被作为值传递，而且实际上在其他位置被调用的所有各种方式，都是观察/行使闭包的例子。

```

function foo() {
  var a = 2;

  function baz() {
    console.log( a ); // 2
  }

  bar( baz );
}

function bar(fn) {
  fn(); // 看妈妈，我看到闭包了！
}

```

我们将内部函数 `baz` 传递给 `bar`，并调用这个内部函数（现在被标记为 `fn`），当我们这么做时，它覆盖在 `foo()` 内部作用域的闭包就可以通过 `a` 的访问观察到。

这样的函数传递也可以是间接的。

```

var fn;

function foo() {
  var a = 2;

  function baz() {
    console.log( a );
  }

  fn = baz; // 将`baz`赋值给一个全局变量
}

function bar() {
  fn(); // 看妈妈，我看到闭包了！
}

foo();
bar(); // 2

```

无论我们使用什么方法将内部函数传送到它的词法作用域之外，它都将维护一个指向它最开始被声明时的作用域的引用，而且无论我们什么时候执行它，这个闭包就会被行使。

## 现在我能看到

前面的代码段有些学术化，而且是人工构建来说明 闭包的使用的。但我保证过给你的东西不止是一个新的酷玩具。我保证过闭包是在你的现存代码中无处不在的东西。现在让我们看看真相。

```
function wait(message) {
    setTimeout( function timer(){
        console.log( message );
    }, 1000 );
}

wait( "Hello, closure!" );
```

我们拿来一个内部函数（名为 `timer`）将它传递给 `setTimeout(..)`。但是 `timer` 拥有覆盖 `wait(..)` 的作用域的闭包，实际上保持并使用着对变量 `message` 的引用。

在我们执行 `wait(..)` 一千毫秒之后，要不是内部函数 `timer` 依然拥有覆盖着 `wait()` 内部作用域的闭包，它早就会消失了。

在引擎的内脏深处，内建的工具 `setTimeout(..)` 拥有一些参数的引用，可能称为 `fn` 或者 `func` 或者其他诸如此类的东西。引擎去调用这个函数，它调用我们的内部 `timer` 函数，而词法作用域依然完好无损。

闭包。

或者，如果你信仰jQuery（或者就此而言，其他的任何JS框架）：

```
function setupBot(name, selector) {
    $( selector ).click( function activator(){
        console.log( "Activating: " + name );
    } );
}

setupBot( "Closure Bot 1", "#bot_1" );
setupBot( "Closure Bot 2", "#bot_2" );
```

我不确定你写的是什么代码，但我通常写一些代码来负责控制全球的闭包无人机军团，所以这完全是真实的！

把玩笑放在一边，实质上无论何时何地只要你将函数作为头等的值看待并将它们传来传去的话，你就可能看到这些函数行使闭包。计时器，事件处理器，Ajax请求，跨窗口消息，web worker，或者任何其他的异步（或同步！）任务，当你传入一个回调函数，你就在它周围悬挂了一些闭包！

注意：第三章介绍了IIFE模式。虽然人们常说IIFE（独自）是一个可以观察到闭包的例子，但是根据我们上面的定义，我有些不同意。

```
var a = 2;

(function IIFE(){
    console.log( a );
})();
```

这段代码“好用”，但严格来说它不是在观察闭包。为什么？因为这个函数（就是我们这里命名为“IIFE”的那个）没有在它的词法作用域之外执行。它仍然在它被声明的相同作用域中（那个同时持有 a 的外围/全局作用域）被调用。a 是通过普通的词法作用域查询找到的，不是通过真正的闭包。

虽说技术上闭包可能发生在声明时，但它不是严格地可以观察到的，因此，就像人们说的，它是一颗在森林中倒掉的树，但没人听得到它。

虽然IIFE本身不是一个闭包的例子，但是它绝对创建了作用域，而且它是用来创建可以被闭包的最常见的工具之一。所以IIFE确实与闭包有强烈的关联，即便它们本身不使用闭包。

亲爱的读者，现在把这本书放下。我有一个任务给你。去打开一些你最近的JavaScript代码。寻找那些被你作为值的函数，并识别你已经在那里使用了闭包，而你以前甚至可能不知道它。

我会等你。

现在……你看到了！

## 循环 + 闭包

用来展示闭包最常见最权威的例子是老实巴交的for循环。

```
for (var i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

注意：当你将函数放在循环内部时Linter经常会抱怨，因为不理解闭包的错误在开发者中太常见了。我们在这里讲解如何正确地利用闭包的全部力量。但是Linter通常不理解这样的微妙之处，所以它们不管怎样都将抱怨，认为你实际上不知道你在做什么。

这段代码的精神是，我们一般将期待它的行为是分别打印数字“1”，“2”，……“5”，一次一个，一秒一个。

实际上，如果你运行这段代码，你会得到“6”被打印5次，一秒一个。

啊？

首先，让我们解释一下“6”是从哪儿来的。循环的终结条件是 `i <=5`。第一次满足这个条件时 `i` 是6。所以，输出的结果反映的是 `i` 在循环终结后的最终值。

如果多看两眼的话这其实很明显。超时的回调函数都将在循环的完成之后立即运行。实际上，就计时器而言，即便在每次迭代中它是 `setTimeout(.., 0)`，所有这些回调函数也都仍然是严格地在循环之后运行的，因此每次都打印 6。

但是这里有个更深刻的问题。要是想让它实际上如我们在语义上暗示的那样动作，我们的代码缺少了什么？

缺少的东西是，我们试图暗示在迭代期间，循环的每次迭代都“捕捉”一份对 `i` 的拷贝。但是，虽然所有这5个函数在每次循环迭代中分离地定义，由于作用域的工作方式，它们都闭包在同一个共享的全局作用域上，而它事实上只有一个 `i`。

这么说来，所有函数共享一个指向相同的 `i` 的引用是理所当然的。循环结构的某些东西往往迷惑我们，使我们认为这里有其他更精巧的东西在工作。但是这里没有。这与根本没有循环，5个超时回调仅仅一个接一个地被声明没有区别。

好了，那么，回到我们火烧眉毛的问题。缺少了什么？我们需要更多铃声被闭包的作用域。明确地说，我们需要为循环的每次迭代都准备一个新的被闭包的作用域。

我们在第三章中学到，IIFE通过声明并立即执行一个函数来创建作用域。

让我们试试：

```
for (var i=1; i<=5; i++) {
  (function(){
    setTimeout( function timer(){
      console.log( i );
    }, i*1000 );
  })();
}
```

这好用吗？试试。我还会等你。

我来为你终结悬念。不好用。但是为什么？很明显我们现在有了更多的词法作用域。每个超时回调函数确实闭包在每次迭代时分别被每个IIFE创建的作用域中。

拥有一个被闭包的空的作用域是不够的。仔细观察。我们的IIFE只是一个空的什么也不做的作用域。它内部需要一些东西才能变得对我们有用。

它需要它自己的变量，在每次迭代时持有值 `i` 的一个拷贝。

```

for (var i=1; i<=5; i++) {
  (function(){
    var j = i;
    setTimeout( function timer(){
      console.log( j );
    }, j*1000 );
  })();
}

```

万岁！它好用了！

有些人偏好一种稍稍变形的形式：

```

for (var i=1; i<=5; i++) {
  (function(j){
    setTimeout( function timer(){
      console.log( j );
    }, j*1000 );
  })( i );
}

```

当然，因为这些IIFE只是函数，我们可以传入 `i`，如果我们乐意的话可以称它为 `j`，或者我们甚至可以再次称它为 `i`。不管哪种方式，这段代码都能工作。

在每次迭代内部使用的IIFE为每次迭代创建了新的作用域，这给了我们的超时回调函数一个机会在每次迭代时闭包一个新的作用域，这些作用域中的每一个都拥有一个持有正确的迭代值的变量给我们访问。

问题解决了！

## 重温块儿作用域

仔细观察我们前一个解决方案的分析。我们使用了一个IIFE来在每一次迭代中创建新的作用域。换句话说，我们实际上每次迭代都需要一个块儿作用域。我们在第三章展示了 `let` 声明，它劫持一个块儿并且就在这个块儿中声明一个变量。

这实质上将块儿变成了一个我们可以闭包的作用域。所以接下来的牛逼代码“就是好用”：

```

for (var i=1; i<=5; i++) {
  let j = i; // 呀，给闭包的块儿作用域！
  setTimeout( function timer(){
    console.log( j );
  }, j*1000 );
}

```

但是，这还不是全部！（用我最棒的Bob Barker嗓音）在用于for循环头部的 let 声明被定义了一种特殊行为。这种行为说，这个变量将不是只为循环声明一次，而是为每次迭代声明一次。并且，它将在每次后续的迭代中被上一次迭代末尾的值初始化。

```
for (let i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

这有多酷？块儿作用域和闭包携手工作，解决世界上所有的问题。我不知道你怎么样，但这使我成了一个快乐的JavaScript开发者。

## 模块

还有其他的代码模式利用了闭包的力量，但是它们都不像回调那样浮于表面。让我们来检视它们中最强大的一种：模块。

```
function foo() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }
}
```

就现在这段代码来说，没有发生明显的闭包。我们只是拥有一些私有数据变量 something 和 another，和几个内部函数 doSomething() 和 doAnother()，它们都拥有覆盖在 foo() 内部作用域上的词法作用域（因此是闭包！）。

但是现在考虑这段代码：

```

function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
}

var foo = CoolModule();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3

```

在JavaScript中我们称这种模式为 模块。实现模块模式的最常见方法经常被称为“揭示模块”，它是我们在里展示的方式的变种。

让我们检视关于这段代码的一些事情。

首先，`CoolModule()` 只是一个函数，但它 必须被调用 才能成为一个被创建的模块实例。没有外部函数的执行，内部作用域的创建和闭包都不会发生。

第二，`CoolModule()` 函数返回一个对象，通过对象字面量语法 `{ key: value, ... }` 标记。这个我们返回的对象拥有指向我们内部函数的引用，但是没有指向我们内部数据变量的引用。我们可以将它们保持为隐藏和私有的。可以很恰当地认为这个返回值对象实质上是一个 我们模块的公有API。

这个返回值对象最终被赋值给外部变量 `foo`，然后我们可以在这个API上访问那些属性，比如 `foo.doSomething()`。

注意：从我们的模块中返回一个实际的对象（字面量）不是必须的。我们可以仅仅直接返回一个内部函数。`jQuery`就是一个很好地例子。`jQuery` 和 `$` 标识符是`jQuery`“模块”的公有 API，但是它们本身只是一个函数（这个函数本身可以有属性，因为所有的函数都是对象）。

`doSomething()` 和 `doAnother()` 函数拥有模块“实例”内部作用域的闭包（通过实际调用 `CoolModule()` 得到的）。当我们通过返回值对象的属性引用，将这些函数传送到词法作用域外部时，我们就建立好了可以观察和行使闭包的条件。

更简单地说，行使模块模式有两个“必要条件”：

1. 必须有一个外部的外围函数，而且它必须至少被调用一次（每次创建一个新的模块实例）。
2. 外围的函数必须至少返回一个内部函数，这样这个内部函数才拥有私有作用域的闭包，并且可以访问和/或修改这个私有状态。

一个仅带有一个函数属性的对象不是真正的模块。从可观察的角度来说，一个从函数调用中返回的对象，仅带有数据属性而没有闭包的函数，也不是真正的模块。

上面的代码段展示了一个称为 `CoolModule()` 独立的模块创建器，它可以被调用任意多次，每次创建一个新的模块实例。这种模式的一个稍稍的变化是当你只想要一个实例的时候，某种“单例”：

```
var foo = (function CoolModule() {
  var something = "cool";
  var another = [1, 2, 3];

  function doSomething() {
    console.log( something );
  }

  function doAnother() {
    console.log( another.join( " ! " ) );
  }

  return {
    doSomething: doSomething,
    doAnother: doAnother
  };
})();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

这里，我们将模块放进一个IIFE（见第三章）中，而且我们立即调用它，并把它的返回值直接赋值给我们单独的模块实例标识符 `foo`。

模块只是函数，所以它们可以接收参数：

```
function CoolModule(id) {
    function identify() {
        console.log( id );
    }

    return {
        identify: identify
    };
}

var foo1 = CoolModule( "foo 1" );
var foo2 = CoolModule( "foo 2" );

foo1.identify(); // "foo 1"
foo2.identify(); // "foo 2"
```

另一种在模块模式上微小但是强大的变化是，为你作为公有API返回的对象命名：

```
var foo = (function CoolModule(id) {
    function change() {
        // 修改公有 API
        publicAPI.identify = identify2;
    }

    function identify1() {
        console.log( id );
    }

    function identify2() {
        console.log( id.toUpperCase() );
    }

    var publicAPI = {
        change: change,
        identify: identify1
    };

    return publicAPI;
})( "foo module" );

foo.identify(); // foo module
foo.change();
foo.identify(); // FOO MODULE
```

通过在模块实例内部持有一个指向公有API对象的内部引用，你可以从内部修改这个模块，包括添加和删除方法，属性，和改变它们的值。

## 现代的模块

各种模块依赖加载器/消息机制实质上都是将这种模块定义包装进一个友好的API。与其检视任意一个特定的库，不如让我（仅）为了说明的目的展示一个非常简单的概念证明：

```
var MyModules = (function Manager() {
    var modules = {};

    function define(name, deps, impl) {
        for (var i=0; i<deps.length; i++) {
            deps[i] = modules[deps[i]];
        }
        modules[name] = impl.apply(impl, deps );
    }

    function get(name) {
        return modules[name];
    }

    return {
        define: define,
        get: get
    };
})();
```

这段代码的关键部分是 `modules[name] = impl.apply(impl, deps)`。这为一个模块调用了它的定义的包装函数（传入所有依赖），并将返回值，也就是模块的API，存储到一个用名称追踪的内部模块列表中。

这里是我可能如何使用它来定义一个模块：

```

MyModules.define( "bar", [], function(){
    function hello(who) {
        return "Let me introduce: " + who;
    }

    return {
        hello: hello
    };
} );

MyModules.define( "foo", ["bar"], function(bar){
    var hungry = "hippo";

    function awesome() {
        console.log( bar.hello( hungry ).toUpperCase() );
    }

    return {
        awesome: awesome
    };
} );

var bar = MyModules.get( "bar" );
var foo = MyModules.get( "foo" );

console.log(
    bar.hello( "hippo" )
); // Let me introduce: hippo

foo.awesome(); // LET ME INTRODUCE: HIPPO

```

模块“foo”和“bar”都使用一个返回公有API的函数来定义。“foo”甚至接收一个“bar”的实例作为依赖参数，并且可以因此使用它。

花些时间检视这些代码段，来完全理解将闭包的力量付诸实践给我们带来的好处。关键之处在于，对于模块管理器来说真的没有什么特殊的“魔法”。它们只是满足了我在上面列出的模块模式的两个性质：调用一个函数定义包装器，并将它的返回值作为这个模块的API保存下来。

换句话说，模块就是模块，即便你在它们上面放了一个友好的包装工具。

## 未来的模块

ES6为模块的概念增加了头等的语法支持。当通过模块系统加载时，ES6将一个文件视为一个独立的模块。每个模块可以导入其他的模块或者特定的API成员，也可以导出它们自己的公有API成员。

注意：基于函数的模块不是一个可以被静态识别的模式（编译器可以知道的东西），所以它们的API语义直到运行时才会被考虑。也就是说，你实际上可以在运行期间修改模块的API（参见早先 publicAPI 的讨论）。

相比之下，ES6模块API是静态的（这些API不会在运行时改变）。因为编译器知道它，它可以（也确实在作！）在（文件加载和）编译期间检查一个指向被导入模块的成员的引用是否实际存在。如果API引用不存在，编译器就会在编译时抛出一个“早期”错误，而不是等待传统的动态运行时解决方案（和错误，如果有的话）。

ES6模块没有“内联”格式，它们必须被定义在一个分离的文件中（每个模块一个）。浏览器/引擎拥有一个默认的“模块加载器”（它是可以被覆盖的，但是这超出我们在此讨论的范围），它在模块被导入时同步地加载模块文件。

考虑这段代码：

### bar.js

```
function hello(who) {
    return "Let me introduce: " + who;
}

export hello;
```

### foo.js

```
// 仅仅从“bar”模块中导入`hello()`、
import hello from "bar";

var hungry = "hippo";

function awesome() {
    console.log(
        hello( hungry ).toUpperCase()
    );
}

export awesome;
```

```
// 导入`foo`和`bar`整个模块
module foo from "foo";
module bar from "bar";

console.log(
    bar.hello( "rhino" )
); // Let me introduce: rhino

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

注意：需要使用前两个代码片段中的内容分别创建两个分离的文件“**foo.js**”和“**bar.js**”。然后，你的程序将加载/导入这些模块来使用它们，就像第三个片段那样。

`import` 在当前的作用域中导入一个模块的API的一个或多个成员，每个都绑定到一个变量（这个例子中是 `hello`）。`module` 将整个模块的API导入到一个被绑定的变量（这个例子中是 `foo`，`bar`）。`export` 为当前模块的公有API导出一个标识符（变量，函数）。在一个模块的定义中，这些操作符可以根据需要使用任意多次。

在模块文件内部的内容被视为像是包围在一个作用域闭包中，就像早先看到的使用函数闭包的模块那样。

## 复习

闭包就像在JavaScript内部被隔离开的魔法世界，看起来少为人知，只有很少一些最勇敢的灵魂才能到达。但是它实际上只是一个标准的，而且几乎明显的事——我们如何在函数即是值，而且可以被随意传递的词法作用域环境中编写代码，

闭包就是当一个函数即使是在它的词法作用域之外被调用时，也可以记住并访问它的词法作用域。

如果我们不能小心地识别它们和它们的工作方式，闭包可能会绊住我们，例如在循环中。但它们也是一种极其强大的工具，以各种形式开启了像 模块 这样的模式。

模块要求两个关键性质：1) 一个被调用的外部包装函数，来创建外围作用域。2) 这个包装函数的返回值必须包含至少一个内部函数的引用，这个函数才拥有包装函数内部作用域的闭包。

现在我们看到了闭包在我们的代码中无处不在，而且我们有能力识别它们，并为了我们自己的利益利用它们！

# 你不懂JS：作用域与闭包

## 附录A：动态作用域

在第二章中，作为与JavaScript中（事实上，其他大多数语言也是）作用域的工作方式模型——“词法作用域”的对比，我们谈到了“动态作用域”。

我们将简单地检视动态作用域，来彻底说明这种比较。但更重要的是，对于JavaScript中的另一种机制（`this`）来说动态作用域实际上是它的一个近亲表兄，我们将在本系列的“`this`与对象原型”中详细讲解这种机制。

正如我们在第二章中看到的，词法作用域是一组关于引擎如何查询变量和它在何处能够找到变量的规则。词法作用域的关键性质是，它是在代码编写时被定义的（假定你不使用`eval()`或`with`作弊的话）。

动态作用域看起来在暗示，有充分的理由，存在这样一种模型，它的作用域是在运行时被确定的，而不是在编写时静态地确定的。让我们通过代码来说明这样的实际情况：

```
function foo() {
  console.log( a ); // 2
}

function bar() {
  var a = 3;
  foo();
}

var a = 2;

bar();
```

在`foo()`的词法作用域中指向`a`的RHS引用将被解析为全局变量`a`，它将导致输出结果为值`2`。

相比之下，动态作用域本身不关心函数和作用域是在哪里和如何被声明的，而是关心它们是从何处被调用的。换句话说，它的作用域链条是基于调用栈的，而不是代码中作用域的嵌套。

所以，如果JavaScript拥有动态作用域，当`foo()`被执行时，理论上下面的代码将得出`3`作为输出结果。

```

function foo() {
    console.log( a ); // 3 (不是 2!)
}

function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();

```

这怎么可能？因为当 `foo()` 不能为 `a` 解析出一个变量引用时，它不会沿着嵌套的（词法）作用域链向上走一层，而是沿着调用栈向上走，以找到 `foo()` 是从何处被调用的。因为 `foo()` 是从 `bar()` 中被调用的，它就会在 `bar()` 的作用域中检查变量，并且在这里找到持有值 3 的 `a`。

奇怪吗？此时此刻你可能会这样认为。

但这可能只是因为你仅在拥有词法作用域的代码中工作过。所以动态作用域看起来陌生。如果你仅使用动态作用域的语言编写过代码，它看起来就是很自然的，而词法作用域将是个怪东西。

要清楚，JavaScript 实际上没有动态作用域。它拥有词法作用域。简单明了。但是 `this` 机制有些像动态作用域。

关键的差异：词法作用域是编写时的，而动态作用域（和 `this`）是运行时的。词法作用域关心的是 函数在何处被声明，但是动态作用域关心的是 函数 从何处 被调用。

最后：`this` 关心的是 函数是如何被调用的，这揭示了 `this` 机制与动态作用域的想法有多么紧密的关联。要了解更多关于 `this` 的细节，请阅读“[this与对象原型](#)”。

# 你不懂JS：作用域与闭包

## 附录B：填补块儿作用域

在第三章中，我们探索了块儿作用域。我们看到最早在ES3中引入的 `with` 和 `catch` 子句都是存在于JavaScript中的块儿作用域的小例子。

但是ES6引入的 `let` 最终使我们的代码有了完整的，不受约束的块作用域能力。不论是在功能上还是在代码风格上，块作用域都使许多激动人心的事情成为可能。

但要是我们想在前ES6环境中使用块儿作用域呢？

考虑这段代码：

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

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

它在ES6环境下工作的非常好。但是我们能在前ES6中这么做吗？`catch` 就是答案。

```
try{throw 2}catch(a){
  console.log( a ); // 2
}

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

哇！这真是看起来丑陋和奇怪的代码。我们看到一个 `try/catch` 似乎强制抛出一个错误，但是这个它抛出的“错误”只是一个值 `2`。然后接收它的变量声明是在 `catch(a)` 子句中。三观：毁尽。

没错，`catch` 子句拥有块儿作用域，这意味着它可以被用于在前ES6环境中填补块儿作用域。

“但是……”，你说。“……没人愿意写这么丑的代码！”你是对的。也没人编写由CoffeeScript编译器输出的（某些）代码。这不是重点。

重点是工具可以将ES6代码转译为能够在前ES6环境中工作的代码。你可以使用块儿作用域编写代码，并从这样的功能中获益，然后让一个编译工具来掌管生成将在部署之后实际工作的代码。

这实际上是所有（嗯哼，大多数）ES6特性首选的迁移路径：在从前ES6到ES6的转变过程中，使用一个代码转译器将ES6代码转换为ES5兼容的代码。

## Traceur

Google维护着一个称为“Traceur”[note-traceur](#)的项目，它的任务正是为了广泛使用ES6特性而将它转译为前ES6（大多数是ES5，但不是全部！）代码。TC39协会依赖这个工具（和其他的工具）来测试他们所规定的特性的语义。

Traceur将从我们的代码段中产生出什么？你猜对了！

```
{
  try {
    throw undefined;
  } catch (a) {
    a = 2;
    console.log( a );
  }
}

console.log( a );
```

所以，使用这种工具，我们可以开始利用块儿作用域，无论我们是否面向ES6，因为try/catch从ES3那时就开始存在了（并且这样工作）。

## 隐含的与明确的块儿

在第三章中，在我们介绍块儿作用域时，我们认识了一些关于代码可维护性/可重构性的潜在陷阱。有什么其他的方法可以利用块儿作用域同时减少这些负面影响吗？

考虑一下let的这种形式，它被称为“let块儿”或“let语句”（和以前的“let声明”对比来说）。

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

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

与隐含地劫持一个既存的块儿不同，let语句为它的作用域绑定明确地创建了一个块儿。这个明确的块儿不仅更显眼，而且在代码重构方面健壮得多，从文法上讲，它通过强制所有的声明都位于块儿的顶部而产生了某种程度上更干净的代码。这使任何块儿都更易于观察，更易于知道什么属于这个作用域和什么不属于这个作用域。

作为一种模式，它是与许多人在函数作用域中采纳的方式相对照的——它们手动地将所有 `var` 声明移动/提升到函数的顶部。`let`语句有意地将它们放在块儿的顶部，而且如果你没有通篇到处使用 `let` 声明，那么你的块儿作用域声明就会在某种程度上更易于识别和维护。

但是，这里有一个问题。`let`语句的形式没有包含在ES6中。就连官方的Traceur编译器也不接受这种形式的代码。

我们有两个选择。我们可以使用ES6合法的语法格式化，再加上一点儿代码规则：

```
/*let*/ { let a = 2;
  console.log( a );
}

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

但是，工具就意味着要解决我们的问题。所以另一个选项是编写明确的`let`语句块儿，并让工具将他转换为合理的，可以工作的代码。

所以，我建造了一个称为“`let-er`<sup>note-let\_er</sup>”的工具来解决这个问题。`let-er`是一个编译期代码转译器，它唯一的任务就是找到`let`语句形式并转译它们。它将你的代码其他部分原封不动地留下，包括任何`let`声明。你可以安全地将 `let-er` 用于ES6转译器的第一步，然后如果有需要，你可以将你的代码通过Traceur这样的东西。

另外，`let-er`有一个配置标志 `--es6`，当它打开时（默认是关闭），会改变生成的代码的种类。与使用 `try/catch` 的ES3填补黑科技不同的是，`let-er` 将拿着我们的代码并产生完全兼容ES6的代码，没有黑科技：

```
{
  let a = 2;
  console.log( a );
}

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

所以，你可以立即开始使用 `let-er`，而且可以面向所有前ES6环境，当你仅关心ES6时，你可以加入配置标志并立即仅面向ES6。

而且最重要的是，你可以使用更好的和更明确的`let`语句形式，即便它（还）不是任何ES官方版本的一部分。

## 性能

让我在 `try/catch` 的性能问题上加入最后一个快速的说明，并/或解决这个问题：“为什么不使用一个IIFE来创建作用域？”

首先，`try/catch` 的性能是慢一些，但是没有任何合理的假设表明它必须是这样，或者它总是这样。因为TC39认可的官方ES6转译器使用`try/catch`，Traceur团队已经让Chrome去改进`try/catch`的性能了，而且它们有很明显的动力这样做。

第二，IIFE和`try/catch`不是一个公平的“苹果对苹果”的比较，因为一个包装着任意代码的函数改变了这段代码的含义，以及它的`this`，`return`，`break`，和`continue`的含义。IIFE不是一个合适一般替代品。它只能在特定的情况下手动使用。

真正的问题变成了：你是否想要使用块儿作用域。如果是，这些工具给你提供了这些选择。如果不，那就在你的代码中继续使用`var`！

note-traceur . Google Traceur ↵  
note-let\_er \: let-er

# 你不懂JS：作用域与闭包

## 附录C：词法this

这本书通篇没有讲解 `this` 机制的任何细节，有一个ES6的话题以一种重要的方式将 `this` 与词法作用域联系了起来，我们将快速检视它一下。

ES6为函数声明增加了一种特殊的语法形式，称为“箭头函数”。它看起来像这样：

```
var foo = a => {
  console.log( a );
};

foo( 2 ); // 2
```

这个所谓的“大箭头”经常被称为是乏味烦冗的（讽刺）`function` 关键字的缩写。

但是在箭头函数上发生的一些事情要重要得多，而且这与在你的声明中少敲几下键盘无关。

简单地说，这段代码有一个问题：

```
var obj = {
  id: "awesome",
  cool: function coolFn() {
    console.log( this.id );
  }
};

var id = "not awesome";

obj.cool(); // awesome

setTimeout( obj.cool, 100 ); // not awesome
```

这个问题就是在 `cool()` 函数上丢失了 `this` 绑定。有各种方法可以解决这个问题，但一个经常被重复的解决方案是 `var self = this;`。

它可能看起来像：

```

var obj = {
  count: 0,
  cool: function coolFn() {
    var self = this;

    if (self.count < 1) {
      setTimeout( function timer(){
        self.count++;
        console.log( "awesome?" );
      }, 100 );
    }
  }
};

obj.cool(); // awesome?

```

用不过于深入细节的方式讲，`var self = this` 的“解决方案”免除了理解和正确使用 `this` 绑定的整个问题，而是退回到我们也许感到更舒服的东西上面：词法作用域。`self` 变成了一个可以通过词法作用域和闭包解析的标识符，而且一直不关心 `this` 绑定发生了什么。

人们不喜欢写繁冗的东西，特别是当他们一次又一次重复它的时候。于是，ES6的一个动机是帮助缓和这些场景，将常见的惯用法问题固定下来，就像这一个。

ES6的解决方案，箭头函数，引入了一种称为“词法this”的行为。

```

var obj = {
  count: 0,
  cool: function coolFn() {
    if (this.count < 1) {
      setTimeout( () => { // 箭头函数能好用？
        this.count++;
        console.log( "awesome?" );
      }, 100 );
    }
  }
};

obj.cool(); // awesome?

```

简单的解释是，当箭头函数遇到它们的 `this` 绑定时，它们的行为与一般的函数根本不同。它们摒弃了 `this` 绑定的所有一般规则，而是将它们的立即外围词法作用域作为 `this` 的值，无论它是什么。

于是，在这个代码段中，箭头函数不会以不可预知的方式丢掉 `this` 绑定，它只是“继承”`cool()` 函数的 `this` 绑定。

虽然这使代码更短，但在我看来，箭头函数只不过是将一个开发者们常犯的错误固化成了语言的语法，这混淆了“`this`绑定”规则与“词法作用域”规则。

换一种说法：为什么要使用 `this` 风格的编码形式来招惹麻烦和繁冗？只要通过将它与词法作用域混合把它剔除掉就好。对于给定的一段代码只采纳一种方式或另一种看起来才是自然的，而不是在同一段代码中将它们混在一起。

注意：源自箭头函数的另一个非议是，它们是匿名的，不是命名的。参见第三章来了解为什么匿名函数不如命名函数理想的原因。

在我看来，这个“问题”的更恰当的解决方式是，正确地使用并接受 `this` 机制。

```
var obj = {
  count: 0,
  cool: function coolFn() {
    if (this.count < 1) {
      setTimeout( function timer(){
        this.count++; // `this` 因为 `bind(..)` 所以安全
        console.log( "more awesome" );
      }.bind( this ), 100 ); // 看，`bind()`!
    }
  }
};

obj.cool(); // more awesome
```

不管是偏好箭头函数的新的词法`this`行为，还是偏好经得起考验的 `bind()`，重要的是要注意箭头函数不仅仅是关于可以少打一些“function”。

它们拥有一种我们应当学习并理解的，有意的行为上的不同，而且如果我们这样选择，就可以利用它们。

现在我们完全理解了词法作用域（和闭包！），理解词法`this`应该是小菜一碟！

# 你不懂JS：作用域与闭包

## 附录D: 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Shane Hudson为本书作序。

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 Tregoining, 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 Hodsdon, 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 ב-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 enamorato, 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 Ziolkowski, 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 Viloslada, 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 Gharegozlu, 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。

# 你不懂JS: **this** 与对象原型

## 目录

- 序
- 前言
- 第一章: `this` 是什么?
  - 为什么要用 `this` ?
  - 困惑
  - 什么是 `this` ?
- 第二章: `this` 豁然开朗!
  - 调用点 (Call-site)
  - 仅仅是规则
  - 一切皆有顺序
  - 绑定的特例
  - 词法 `this`
- 第三章: 对象
  - 语法
  - 类型
  - 内容
  - 迭代
- 第四章: 混合 (淆) “类”的对象
  - 类理论
  - 类机制
  - 类继承
  - 混合 (Mixin)
- 第五章: 原型
  - `[[Prototype]]`
  - “类”
  - “(原型) 继承”
  - 对象链接
- 第六章: 行为委托
  - 迈向面向委托的设计
  - Classes vs. Objects
  - 更简单的设计
  - 更好的语法
  - 自省

- 附录A: ES6 class
- 附录B: 鸣谢

# 你不懂JS: **this** 与对象原型

## 序

在我读这本书为写这篇序言做准备时，我被迫反思我是如何学习JavaScript，和在我用它进行编程和开发的最近15年它改变了多少。

当我15年前开始使用JavaScript时，在你的网页上使用CSS和JS这样的非HTML技术的实践称为DHTML或动态HTML。回到那时，JavaScript的用途有很大的不同，并且倾向于在你的网页上加入动画雪花，或者在状态栏上显示告知时间的动态时钟。可以说，在我的职业生涯早期，因为这些我经常能在因特网上找到的新奇小玩意儿，我真的没有太注意JavaScript。

直到2005年我第一次重新认识到JavaScript是一个我需要更加重视的真正的编程语言。在挖掘研究了Google Maps的第一个beta版后，我被它的潜力吸引住了。那时，Google Maps是第一个同种类的应用——它允许你用鼠标移动地图，放缩，请求服务器而不必刷新页面——都是通过JavaScript。它看起来就像魔法！

当什么东西看起来像魔法时，这通常都一个信号：你正处在用新方法做事的黎明。噢，我没有错——快进到今天，我敢说JavaScript是同时用于客户端和服务端编程的主要语言之一，而且我不会用其他方式这么说。

在我回顾过去的15年时，我的一个遗憾是在2005年以前我没有给JavaScript更多机会，或者更确切地说，我缺乏远见来看到JavaScript是一个真正的编程语言，就像C++，C#，Java和许多其他语言一样有用。

如果我在自己的职业生涯一开始就拥有这套 你不懂JS 系列丛书，我们的职业经历将和今天有很大的不同。我喜欢这个系列的一个地方是：当你通读这个系列时，它在建立你的理解的水平上讲解JS，而且用一种有趣且信息丰富的方式。

**this**与对象原型 是这个系列的一个绝妙的续作。它漂亮且自然地建立于前一本书之上，作用于与闭包，将知识扩展至JS语言中十分重要的部分，**this** 关键字和原型。这两个简单的东西是你将在未来的书中学到的东西的枢纽，因为他们是用JavaScript进行真正的编程的基础。如何创建对象，关联它们，和扩展它们来表达你的应用中的东西，是用JavaScript建立大型和复杂应用程序所必要的。没有它们，用JavaScript制造复杂应用程序（比如Google Maps）将是不可能的。

我敢说绝大多数web开发者可能从没建立过JavaScript对象，而只是将这个语言当做按钮和AJAX请求的事件绑定胶水。我曾经在我职业生涯的某一点上属于这个群体，但是当我学习了如何掌握原型和在JavaScript中创建对象后，一个充满可能性的世界向我打开了大门。如果你

属于仅仅会写事件绑定胶水代码的那一类，这本书是必读的；如果你只是需要进修，这本书是你一定会用到的资源。不管怎样，你不会失望的。相信我！

Nick Berardi

[nickberardi.com](http://nickberardi.com), @nberardi

# 你不懂JS: **this** 与对象原型

## 第一章: **this** 是什么?

JavaScript中最令人困惑的机制之一就是 `this` 关键字。它是一个在每个函数作用域中自动定义的特殊标识符关键字，但即便是一些老练的开发者也对它到底指向什么感到困扰。

任何足够先进的技术都跟魔法没有区别。-- Arthur C. Clarke

JavaScript的 `this` 机制实际上没有那么先进，但是开发者们总是在大脑中引用这句话来表达“复杂”和“混乱”，毫无疑问，如果没有清晰的理解，在你的困惑中 `this` 可能看起来就是彻头彻尾的魔法。

注意：“`this`”这个词是在一般的论述中极常用的代词。所以，特别是在口头论述中，很难确定我们是在将“`this`”作为一个代词使用，还是在将它作为一个实际的关键字识别符使用。为了表意清晰，我会总是使用 `this` 来代表特殊的关键字，而在其他情况下使用“`this`”或 `this` 或 `this.`

## 为什么要用 **this** ?

如果对于那些老练的JavaScript开发者来说 `this` 机制都是如此的令人费解，那么有人会问为什么这种机制会有用？它带来的麻烦不是比好处多吗？在讲解如何有用之前，我们应当先来看看为什么有用。

让我们试着展示一下 `this` 的动机和用途：

```
function identify() {
    return this.name.toUpperCase();
}

function speak() {
    var greeting = "Hello, I'm " + identify.call( this );
    console.log( greeting );
}

var me = {
    name: "Kyle"
};

var you = {
    name: "Reader"
};

identify.call( me ); // KYLE
identify.call( you ); // READER

speak.call( me ); // Hello, I'm KYLE
speak.call( you ); // Hello, I'm READER
```

如果这个代码段如何工作让你困惑，不要担心！我们很快就会讲解它。只是简要地将这些问题放在旁边，以便于我们可以更清晰的探究为什么。

这个代码片段允许 `identify()` 和 `speak()` 函数对多个环境对象（`me` 和 `you`）进行复用，而不是针对每个对象定义函数的分离版本。

与使用 `this` 相反地，你可以明确地将环境对象传递给 `identify()` 和 `speak()`。

```
function identify(context) {
    return context.name.toUpperCase();
}

function speak(context) {
    var greeting = "Hello, I'm " + identify( context );
    console.log( greeting );
}

identify( you ); // READER
speak( me ); // Hello, I'm KYLE
```

然而，`this` 机制提供了更优雅的方式来隐含地“传递”一个对象引用，导致更加干净的API设计和更容易的复用。

你的使用模式越复杂，你就会越清晰地看到：将执行环境作为一个明确参数传递，通常比传递 `this` 执行环境要乱。当我们探索对象和原型时，你将会看到一组可以自动引用恰当执行环境对象的函数是多么有用。

## 困惑

我们很快就要开始讲解 `this` 是如何实际工作的，但我们首先要摒弃一些误解——它实际上不是如何工作的。

在开发者们用太过于字面的方式考虑“`this`”这个名字时就会产生困惑。这通常会产生两种臆测，但都是不对的。

## 它自己

第一种常见的倾向是认为 `this` 指向函数自己。至少，这是一种语法上的合理推测。

为什么你想要在函数内部引用它自己？最通常的理由是递归（在函数内部调用它自己）这样的情形，或者是一个在第一次被调用时会解除自己绑定的事件处理器。

初次接触JS机制的开发者们通常认为，将函数作为一个对象（JavaScript中所有的函数都是对象！），可以让你在方法调用之间储存状态（属性中的值）。这当然是可能的，而且有一些有限的用处，但这本书的其余部分将会阐述许多其他的模式，提供比函数对象更好的地方来存储状态。

过一会儿我们将探索一个模式，来展示 `this` 是如何不让一个函数像我们可能假设的那样，得到它自身的引用的。

考虑下面的代码，我们试图追踪函数(`foo`)被调用了多少次：

```
function foo(num) {
    console.log("foo: " + num);

    // 追踪`foo`被调用了多少次
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo(i);
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// `foo`被调用了多少次？
console.log(foo.count); // 0 -- 这他妈怎么回事.....?
```

`foo.count` 依然 是 `0`，即使四个 `console.log` 语句明明告诉我们 `foo(..)` 实际上被调用了四次。这种失败来源于对于 `this` (在 `this.count++` 中)的含义进行了过于字面化的解释。

当代码执行 `foo.count = 0` 时，它确实在函数对象 `foo` 中加入了一个 `count` 属性。但是对于函数内部的 `this.count` 引用，`this` 其实根本就不指向那个函数对象，即便属性名称一样，但根对象也不同，因而产生了混淆。

注意：一个负责任的开发者 应当 在这里提出一个问题：“如果我递增的 `count` 属性不是我以为的那个，那是哪个 `count` 被我递增了？”。实际上，如果他再挖的深一些，他会发现自己不小心创建了一个全局变量 `count` (第二章解释了这是如何发生的)，而且它当前的值是 `Nan`。当然，一旦他发现这个不寻常的结果后，他会有一堆其他的问题：“它怎么是全局的？为什么它是 `Nan` 而不是某个正确的计数值？”。（见第二章）

与停在这里来深究为什么 `this` 引用看起来不是如我们期待的那样工作，并且回答那些尖锐且重要的问题相反，许多开发者简单地完全回避这个问题，转向一些其他的另类解决方法，比如创建另一个对象来持有 `count` 属性：

```
function foo(num) {
    console.log("foo: " + num);

    // 追踪foo被调用了多少次
    data.count++;
}

var data = {
    count: 0
};

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo(i);
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// foo被调用了多少次?
console.log(data.count); // 4
```

虽然这种方式确实“解决”了问题，但不幸的是它简单地忽略了真正的问题——缺乏对 `this` 的含义和其工作方式上的理解——反而退回到了一个他更加熟悉的机制的舒适区：词法作用域。

注意：词法作用域是一个完善且有用的机制；我不是在用任何方式贬低它的作用（参见本系列的“作用域与闭包”）。但在如何使用 `this` 这个问题上总是靠猜，而且通常都犯错，并不是一个退回到词法作用域，而且从不学习为什么 `this` 不跟你合作的好理由。

为了从函数对象内部引用它自己，一般来说通过 `this` 是不够的。你通常需要通过一个指向它的词法标识符（变量）得到函数对象的引用。

考虑这两个函数：

```
function foo() {
    foo.count = 4; // `foo` 引用它自己
}

setTimeout(function(){
    // 匿名函数（没有名字）不能引用它自己
}, 10);
```

第一个函数，称为“命名函数”，`foo` 是一个引用，可以用于在它内部引用自己。

但是在第二个例子中，传递给 `setTimeout(..)` 的回调函数没有名称标识符（所以被称为“匿名函数”），所以没有恰当的办法引用函数对象自己。

注意：在函数中有一个老牌儿但是现在被废弃的，而且令人皱眉头的 `arguments.callee` 引用也指向当前正在执行的函数的函数对象。这个引用通常是匿名函数在自己内部访问函数对象的唯一方法。然而，最佳的办法是完全避免使用匿名函数，至少是对于那些需要自引用的函数，而使用命名函数（表达式）。`arguments.callee` 已经被废弃而且不应该再使用。

对于当前我们的例子来说，另一个好用的解决方案是在每一个地方都使用 `foo` 标识符作为函数对象的引用，而根本不用 `this`：

```
function foo(num) {
    console.log("foo: " + num);

    // 追踪`foo`被调用了多少次
    foo.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo(i);
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// `foo`被调用了多少次？
console.log(foo.count); // 4
```

然而，这种方法也类似地回避了对 `this` 的真正理解，而且完全依靠变量 `foo` 的词法作用域。

另一种解决问题的方法是强迫 `this` 指向 `foo` 函数对象：

```
function foo(num) {
  console.log("foo: " + num);

  // 追踪`foo`被调用了多少次
  // 注意：由于`foo`的被调用方式（见下方），`this`现在确实是`foo`、
  this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
  if (i > 5) {
    // 使用`call(..)`，我们可以保证`this`指向函数对象(`foo`)
    foo.call(foo, i);
  }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// `foo`被调用了多少次？
console.log(foo.count); // 4
```

与回避 `this` 相反，我们接受它。我们将会更完整地讲解这样的技术如何工作，所以如果你依然有点儿糊涂，不要担心！

## 它的作用域

第二常见的对 `this` 的含义的误解，是它不知怎的指向了函数的作用域。这是一个刁钻的问题，因为在某一种意义上它有正确的部分，而在另外一种意义上，它是严重的误导。

明确地说，`this` 不会以任何方式指向函数的词法作用域。作用域好像是一个将所有可用标识符作为属性的对象，这从内部来说是对的。但是JavaScript代码不能访问作用域“对象”。它是引擎的内部实现。

考虑下面代码，它（失败的）企图跨越这个边界，用 `this` 来隐含地引用函数的词法作用域：

```

function foo() {
    var a = 2;
    this.bar();
}

function bar() {
    console.log( this.a );
}

foo(); //undefined

```

这个代码段里不只是一个错误。虽然它看起来是在故意瞎搞，但你看到的这段代码，是从公共的帮助论坛社区中被交换的真实代码中提取出来的。真是难以想象对 `this` 的臆想是多么的误导人。

首先，试图通过 `this.bar()` 来引用 `bar()` 函数。它几乎可以说是碰巧能够工作，我们过一会儿再解释它是如何工作的。调用 `bar()` 最自然的方式是省略开头的 `this.`，而仅对标识符进行词法引用。

然而，写下这段代码的开发者试图用 `this` 在 `foo()` 和 `bar()` 的词法作用域间建立一座桥，使得 `bar()` 可以访问 `foo()` 内部作用域的变量 `a`。这样的桥是不可能的。你不能使用 `this` 引用在词法作用域中查找东西。这是不可能的。

每当你感觉自己正在试图使用 `this` 来进行词法作用域的查询时，提醒你自己：这里没有桥。

## 什么是 `this` ?

我们已经列举了各种不正确的臆想，现在让我们把注意力 `this` 机制是如何真正工作的。

我们早先说过，`this` 不是编写时绑定，而是运行时绑定。它依赖于函数调用的上下文条件。`this` 绑定和函数声明的位置无关，反而和函数被调用的方式有关。

当一个函数被调用时，会建立一个活动记录，也称为执行环境。这个记录包含函数是从何处 (`call-stack`) 被调用的，函数是如何被调用的，被传递了什么参数等信息。这个记录的属性之一，就是在函数执行期间将被使用的 `this` 引用。

下一章中，我们将会学习寻找函数的调用点 (**call-site**) 来判定它的执行如何绑定 `this`。

## 复习

对于那些没有花时间学习 `this` 绑定机制如何工作的JavaScript开发者来说，`this` 绑定一直是困惑的根源。猜测，试错，或者盲目地从Stack Overflow的回答中复制粘贴，都不是有效或正确利用 `this` 这么重要的机制的方法。

为了学习 `this`，你必须首先学习 `this` 不是什么，不论是哪种把你误导至何处的臆测或误解。`this` 既不是函数自身的引用，也不是函数词法作用域的引用。

`this` 实际上是在函数被调用时建立的一个绑定，它指向什么 是完全由函数被调用的调用点来决定的。

# 你不懂JS: **this** 与对象原型

## 第二章: **this** 豁然开朗 !

在第一章中，我们摒弃了种种对 `this` 的误解，并且学习了 `this` 是一个完全根据调用点（函数是如何被调用的）而为每次函数调用建立的绑定。

### 调用点 (Call-site)

为了理解 `this` 绑定，我们不得不理解调用点：函数在代码中被调用的位置（不是被声明的位置）。我们必须考察调用点来回答这个问题：这个 `this` 指向什么？

一般来说寻找调用点就是：“找到一个函数是在哪里被调用的”，但不总是那么简单，比如某些特定的编码模式会使真正的调用点变得不那么明确。

考虑调用栈 (**call-stack**) （使我们到达当前执行位置而被调用的所有方法的堆栈）是十分重要的。我们关心的调用点就位于当前执行中的函数之前的调用。

我们来展示一下调用栈和调用点：

```

function baz() {
    // 调用栈是: `baz`
    // 我们的调用点是global scope (全局作用域)

    console.log( "baz" );
    bar(); // <-- `bar`的调用点
}

function bar() {
    // 调用栈是: `baz` -> `bar`
    // 我们的调用点位于`baz`

    console.log( "bar" );
    foo(); // <-- `foo`的call-site
}

function foo() {
    // 调用栈是: `baz` -> `bar` -> `foo`
    // 我们的调用点位于`bar`

    console.log( "foo" );
}

baz(); // <-- `baz`的调用点

```

在分析代码来寻找（从调用栈中）真正的调用点时要小心，因为它是影响 `this` 绑定的唯一因素。

注意：你可以通过按顺序观察函数的调用链在你的大脑中建立调用栈的视图，就像我们在上面代码段中的注释那样。但是这很痛苦而且易错。另一种观察调用栈的方式是使用你的浏览器的调试工具。大多数现代的桌面浏览器都内建开发者工具，其中就包含JS调试器。在上面的代码段中，你可以在调试工具中为 `foo()` 函数的第一行设置一个断点，或者简单的在这第一行上插入一个 `debugger` 语句。当你运行这个网页时，调试工具将会停止在这个位置，并且向你展示一个到达这一行之前所有被调用过的函数的列表，这就是你的调用栈。所以，如果你想调查 `this` 绑定，可以使用开发者工具取得调用栈，之后从上向下找到第二个记录，那就是你真正的调用点。

## 仅仅是规则

现在我们将注意力转移到调用点如何决定在函数执行期间 `this` 指向哪里。

你必须考察调用点并判定4种规则中的哪一个适用。我们将首先独立的解释一下这4种规则中的每一种，之后我们来展示一下如果有多种规则可以适用调用点时，它们的优先顺序。

### 默认绑定 (Default Binding)

我们要考察的第一种规则来源于函数调用的最常见的情况：独立函数调用。可以认为这种 `this` 规则是在没有其他规则适用时的默认规则。

考虑这个代码段：

```
function foo() {  
    console.log( this.a );  
}  
  
var a = 2;  
  
foo(); // 2
```

第一点要注意的，如果你还没有察觉到，是在全局作用域中的声明变量，也就是 `var a = 2`，是全局对象的同名属性的同义词。它们不是互相拷贝对方，它们就是彼此。正如一个硬币的两面。

第二，我们看到当 `foo()` 被调用时，`this.a` 解析为我们的全局变量 `a`。为什么？因为在这种情况下，对此方法调用的 `this` 实施了默认绑定，所以使 `this` 指向了全局对象。

我们怎么知道这里适用 默认绑定？我们考察调用点来看看 `foo()` 是如何被调用的。在我们的代码段中，`foo()` 是被一个直白的，毫无修饰的函数引用调用的。没有其他的我们将要展示的规则适用于这里，所以 默认绑定 在这里适用。

如果 `strict mode` 在这里生效，那么对于 默认绑定 来说全局对象是不合法的，所以 `this` 将被设置为 `undefined`。

```
function foo() {  
    "use strict";  
  
    console.log( this.a );  
}  
  
var a = 2;  
  
foo(); // TypeError: `this` is `undefined`
```

一个微妙但是重要的细节是：即便所有的 `this` 绑定规则都是完全基于调用点，如果 `foo()` 的内容没有在 `strict mode` 下执行，对于 默认绑定 来说全局对象是唯一合法的；`foo()` 的调用点的 `strict mode` 状态与此无关。

```

function foo() {
    console.log( this.a );
}

var a = 2;

(function(){
    "use strict";

    foo(); // 2
})();

```

注意：在你的代码中故意混用 strict mode 和非 strict mode 通常是让人皱眉头的。你的程序整体可能应当不是 Strict 就是 非Strict。然而，有时你可能会引用与你的 Strict 模式不同的第三方包，所以对这些微妙的兼容性细节要多加小心。

## 隐含绑定（Implicit Binding）

另一种要考虑的规则是：调用点是否有一个环境对象（context object），也称为拥有者（owning）或容器（containing）对象，虽然这些名词可能有些误导人。

考虑这段代码：

```

function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

obj.foo(); // 2

```

首先，注意 `foo()` 被声明然后作为引用属性添加到 `obj` 上的方式。无论 `foo()` 是否一开始就在 `obj` 上被声明，还是后来作为引用添加（如上面代码所示），都是这个函数被 `obj` 所“拥有”或“包含”。

然而，调用点使用 `obj` 环境来引用 函数，所以你可以说 `obj` 对象在函数被调用的时间点上“拥有”或“包含”这个 函数引用。

不论你怎样称呼这个模式，在 `foo()` 被调用的位置上，它被冠以一个指向 `obj` 的对象引用。当一个方法引用存在一个环境对象时，隐含绑定 规则会说：是这个对象应当被用于这个函数调用的 `this` 绑定。

因为 `obj` 是 `foo()` 调用的 `this`，所以 `this.a` 就是 `obj.a` 的同义词。

只有对象属性引用链的最后一层是影响调用点的。比如：

```
function foo() {
    console.log( this.a );
}

var obj2 = {
    a: 42,
    foo: foo
};

var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo(); // 42
```

## 隐含地丢失 (Implicitly Lost)

this 绑定最常让人沮丧的事情之一，就是当一个 隐含绑定 丢失了它的绑定，这通常意味着它会退回到 默认绑定，根据 strict mode 的状态，结果不是全局对象就是 undefined 。

考虑这段代码：

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var bar = obj.foo; // 函数引用！

var a = "oops, global"; // `a`也是一个全局对象的属性

bar(); // "oops, global"
```

尽管 bar 似乎是 obj.foo 的引用，但实际上它只是另一个 foo 自己的引用而已。另外，起作用的调用点是 bar() ，一个直白，毫无修饰的调用，因此 默认绑定 适用于这里。

这种情况发生的更加微妙，更常见，更意外的方式，是当我们考虑传递一个回调函数时：

```

function foo() {
    console.log( this.a );
}

function doFoo(fn) {
    // `fn` 只不过`foo`的另一个引用

    fn(); // <-- 调用点!
}

var obj = {
    a: 2,
    foo: foo
};

var a = "oops, global"; // `a`也是一个全局对象的属性

doFoo( obj.foo ); // "oops, global"

```

参数传递仅仅是一种隐含的赋值，而且因为我们在传递一个函数，它是一个隐含的引用赋值，所以最终结果和我们前一个代码段一样。

那么如果接收你所传递回调的函数不是你的，而是语言内建的呢？没有区别，同样的结果。

```

function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var a = "oops, global"; // `a`也是一个全局对象的属性

setTimeout( obj.foo, 100 ); // "oops, global"

```

把这个粗糙的，理论上的 `setTimeout()` 假想实现当做JavaScript环境内建的实现的话：

```

function setTimeout(fn,delay) {
    // (通过某种方法) 等待`delay`毫秒
    fn(); // <-- 调用点!
}

```

正如我们刚刚看到的，我们的回调函数丢掉他们的 `this` 绑定是十分常见的事情。但是另一种 `this` 使我们吃惊的方式是，接收我们的回调的函数故意改变调用的 `this`。那些很受欢迎的事件处理JavaScript包就十分喜欢强制你的回调的 `this` 指向触发事件的DOM元素。虽然有

时这很有用，但其他时候这简直能气死人。不幸的是，这些工具很少给你选择。

不管哪一种意外改变 `this` 的方式，你都不能真正地控制你的回调函数引用将如何被执行，所以你（还）没有办法控制调用点给你一个故意的绑定。我们很快就会看到一个方法，通过固定 `this` 来解决这个问题。

## 明确绑定 (Explicit Binding)

用我们刚看到的 隐含绑定，我们不得不改变目标对象使它自身包含一个对函数的引用，而后使用这个函数引用属性来间接地（隐含地）将 `this` 绑定到这个对象上。

但是，如果你想强制一个函数调用使用某个特定对象作为 `this` 绑定，而不在这个对象上放置一个函数引用属性呢？

JavaScript语言中的“所有”函数都有一些工具（通过他们的 `[[Prototype]]` ——待会儿详述）可以用于这个任务。特别是，函数拥有 `call(..)` 和 `apply(..)` 方法。从技术上讲，JavaScript宿主环境有时会提供一些很特别的函数，它们没有这些功能，但这很少见。绝大多数被提供的函数，当然还有你将创建的所有的函数，都可以访问 `call(..)` 和 `apply(..)`。

这些工具如何工作？它们接收的第一个参数都是一个用于 `this` 的对象，之后使用这个指定的 `this` 来调用函数。因为你已经直接指明你想让 `this` 是什么，所以我们称这种方式为 明确绑定 (*explicit binding*)。

考虑这段代码：

```
function foo() {
  console.log( this.a );
}

var obj = {
  a: 2
};

foo.call( obj ); // 2
```

通过 `foo.call(..)` 使用 明确绑定 来调用 `foo`，允许我们强制函数的 `this` 指向 `obj`。

如果你传递一个简单原始类型值（`string`，`boolean`，或 `number` 类型）作为 `this` 绑定，那么这个原始类型值会被包装在它的对象类型中（分别是 `new String(..)`，`new Boolean(..)`，或 `new Number(..)`）。这通常称为“boxing（封箱）”。

注意：就 `this` 绑定的角度讲，`call(..)` 和 `apply(..)` 是完全一样的。它们确实在处理其他参数上的方式不同，但那不是我们当前关心的。

不幸的是，单独依靠 明确绑定 仍然不能为我们先前提到的问题提供解决方案，也就是函数“丢失”自己原本的 `this` 绑定，或者被第三方框架覆盖，等等问题。

## 硬绑定 (Hard Binding)

但是有一个 明确绑定 的变种确实可以实现这个技巧。考虑这段代码：

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};

var bar = function() {
    foo.call( obj );
};

bar(); // 2
setTimeout( bar, 100 ); // 2

// `bar`将`foo`的`this`硬绑定到`obj`
// 所以它不可以被覆盖
bar.call( window ); // 2
```

我们来看看这个变种是如何工作的。我们创建了一个函数 `bar()`，在它的内部手动调用 `foo.call(obj)`，由此强制 `this` 绑定到 `obj` 并调用 `foo`。无论你过后怎样调用函数 `bar`，它总是手动使用 `obj` 调用 `foo`。这种绑定即明确又坚定，所以我们称之为 硬绑定 (*hard binding*)

用 硬绑定 将一个函数包装起来的最典型的方法，是为所有传入的参数和传出的返回值创建一个通道：

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = function() {
    return foo.apply( obj, arguments );
};

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

另一种表达这种模式的方法是创建一个可复用的帮助函数：

```

function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

// 简单的`bind`帮助函数
function bind(fn, obj) {
    return function() {
        return fn.apply( obj, arguments );
    };
}

var obj = {
    a: 2
};

var bar = bind( foo, obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5

```

由于硬绑定是一个如此常用的模式，它已作为ES5的内建工具提供：`Function.prototype.bind`，像这样使用：

```

function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = foo.bind( obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5

```

`bind(..)` 返回一个硬编码的新函数，它使用你指定的 `this` 环境来调用原本的函数。

注意：在ES6中，`bind(..)` 生成的硬绑定函数有一个名为 `.name` 的属性，它源自于原始的目标函数（*target function*）。举例来说：`bar = foo.bind(..)` 应该会有一个 `bar.name` 属性，它的值为 `"bound foo"`，这个值应当会显示在调用栈轨迹的函数调用名称中。

## API调用的“环境”

确实，许多包中的函数，和许多在JavaScript语言以及宿主环境中的内建函数，都提供一个可选参数，通常称为“环境（context）”，这种设计作为一种替代方案来确保你的回调函数使用特定的 `this` 而不必非得使用 `bind(..)`。

举例来说：

```
function foo(el) {
  console.log( el, this.id );
}

var obj = {
  id: "awesome"
};

// 使用`obj`作为`this`来调用`foo(..)`
[1, 2, 3].forEach( foo, obj ); // 1 awesome 2 awesome 3 awesome
```

从内部来说，这种类型的函数几乎可以确定是通过 `call(..)` 或 `apply(..)` 使用了明确绑定来节省你的麻烦。

## new 绑定 ( new Binding )

第四种也是最后一种 `this` 绑定规则，需要我们重新思考关于JavaScript中对函数和对象的常见误解。

在传统的面向类语言中，“构造器”是附着在类上的一种特殊方法，当使用 `new` 操作符来初始化一个类时，这个类的构造器就会被调用。通常看起来像这样：

```
something = new MyClass(..);
```

JavaScript拥有 `new` 操作符，而且它使用的代码模式看起来基本和我们在面向类语言中看到的一样；大多数开发者猜测JavaScript机制是某种相似的东西。但是，实际上JavaScript的机制和 `new` 在JS中的用法所暗示的面向类的功能没有任何联系。

首先，让我们重新定义JavaScript的“构造器”是什么。在JS中，构造器仅仅是一个函数，它们偶然地被前置的 `new` 操作符调用。它们不依附于类，它们也不初始化一个类。它们甚至不是一种特殊的函数类型。它们本质上只是一般的函数，在被使用 `new` 来调用时改变了行为。

比如，`Number(..)` 函数作为一个构造器来说，引用ES5.1的语言规范：

### 15.7.2 The Number 构造器

当Number作为new表达式的一部分被调用时，它是一个构造器：它初始化这个新创建的对象。

所以，任何关联在对象上的函数，包括像 `Number(..)`（见第三章）这样的内建对象函数都可以在前面加上 `new` 来被调用，这使函数调用成为一个构造器调用（**constructor call**）。这是一个重要且微妙的区别：实际上不存在“构造器函数”这样的东西，而只有函数的构造器调用。

当在函数前面被加入 `new` 调用时，也就是构造器调用时，下面这些事情会自动完成：

1. 一个全新的对象会凭空创建（就是被构建）
2. 这个新构建的对象会被接入原形链（`[[Prototype]] -linked`）
3. 这个新构建的对象被设置为函数调用的 `this` 绑定
4. 除非函数返回一个它自己的其他对象，这个被 `new` 调用的函数将自动返回这个新构建的对象。

步骤1，3和4是我们当下要讨论的。我们现在跳过第2步，在第五章回来讨论。

考虑这段代码：

```
function foo(a) {
  this.a = a;
}

var bar = new foo( 2 );
console.log( bar.a ); // 2
```

通过在前面使用 `new` 来调用 `foo(..)`，我们构建了一个新的对象并把这个新对象作为 `foo(..)` 调用的 `this`。 `new` 是函数调用可以绑定 `this` 的最后一种方式，我们称之为 `new` 绑定（*new binding*）。

## 一切皆有顺序

如此，我们已经揭示了函数调用中的4种 `this` 绑定规则。你需要做的一切就是找到调用点然后考察哪一种规则适用于它。但是，如果调用点上有很多种规则都适用呢？这些规则必须有一个优先顺序，我们下面就来展示这些规则以什么样的优先顺序实施。

很显然，默认绑定在4种规则中拥有最低的优先权。所以我们先把它放在一边。

隐含绑定和明确绑定哪一个更优先呢？我们来测试一下：

```
function foo() {
    console.log( this.a );
}

var obj1 = {
    a: 2,
    foo: foo
};

var obj2 = {
    a: 3,
    foo: foo
};

obj1.foo(); // 2
obj2.foo(); // 3

obj1.foo.call( obj2 ); // 3
obj2.foo.call( obj1 ); // 2
```

所以, 明确绑定的优先权要高于隐含绑定, 这意味着你应当在考察隐含绑定之前首先考察明确绑定是否适用。

现在, 我们只需要搞清楚 *new* 绑定的优先级位于何处。

```
function foo(something) {
    this.a = something;
}

var obj1 = {
    foo: foo
};

var obj2 = {};

obj1.foo( 2 );
console.log( obj1.a ); // 2

obj1.foo.call( obj2, 3 );
console.log( obj2.a ); // 3

var bar = new obj1.foo( 4 );
console.log( obj1.a ); // 2
console.log( bar.a ); // 4
```

好了, *new* 绑定的优先级要高于隐含绑定。那么你觉得 *new* 绑定的优先级较之于明确绑定是高还是低呢?

注意: `new` 和 `call / apply` 不能同时使用, 所以 `new foo.call(obj1)` 是不允许的, 也就是不能直接对比测试 `new` 绑定 和 明确绑定。但是我们依然可以使用 硬绑定 来测试这两个规则的优先级。

在我们进入代码中探索之前, 回想一下 硬绑定 物理上是如何工作的, 也就是 `Function.prototype.bind(..)` 创建了一个新的包装函数, 这个函数被硬编码为忽略它自己的 `this` 绑定 (不管它是什么), 转而手动使用我们提供的。

因此, 这似乎看起来很明显, 硬绑定 (明确绑定的一种) 的优先级要比 `new` 绑定 高, 而且不能被 `new` 覆盖。

我们检验一下:

```
function foo(something) {
  this.a = something;
}

var obj1 = {};

var bar = foo.bind( obj1 );
bar( 2 );
console.log( obj1.a ); // 2

var baz = new bar( 3 );
console.log( obj1.a ); // 2
console.log( baz.a ); // 3
```

哇! `bar` 是硬绑定到 `obj1` 的, 但是 `new bar(3)` 没有想我们期待的那样将 `obj1.a` 变为 `3`。反而, 硬绑定 (到 `obj1`) 的 `bar(..)` 调用可以被 `new` 所覆盖。因为 `new` 被实施, 我们得到一个名为 `baz` 的新创建的对象, 而且我们确实看到 `baz.a` 的值为 `3`。

如果你回头看看我们的“山寨”绑定帮助函数, 这很令人吃惊:

```
function bind(fn, obj) {
  return function() {
    fn.apply( obj, arguments );
  };
}
```

如果你推导这段帮助代码如何工作, 会发现对于 `new` 操作符调用来说没有办法去像我们观察到的那样, 将绑定到 `obj` 的硬绑定覆盖。

但是ES5的内建 `Function.prototype.bind(..)` 更加精妙, 实际上十分精妙。这里是MDN网页上为 `bind(..)` 提供的polyfill (低版本兼容填补工具) :

```

if (!Function.prototype.bind) {
    Function.prototype.bind = function(oThis) {
        if (typeof this !== "function") {
            // 可能的与 ECMAScript 5 内部的 IsCallable 函数最接近的东西,
            throw new TypeError( "Function.prototype.bind - what " +
                "is trying to be bound is not callable"
            );
        }

        var aArgs = Array.prototype.slice.call( arguments, 1 ),
            fToBind = this,
            fNOP = function(){},
            fBound = function(){
                return fToBind.apply(
                    (
                        this instanceof fNOP &&
                        oThis ? this : oThis
                    ),
                    aArgs.concat( Array.prototype.slice.call( arguments ) )
                );
            }
;

        fNOP.prototype = this.prototype;
        fBound.prototype = new fNOP();

        return fBound;
    };
}

```

注意：在ES5中，就将与 `new` 一起使用的硬绑定函数（参照下面来看为什么这有用）而言，上面的 `bind(..)` polyfill与内建的 `bind(..)` 是不同的。因为polyfill不能像内建工具那样，没有 `.prototype` 就能创建函数，这里使用了一些微妙而间接的方法来近似模拟相同的行为。如果你打算将硬绑定函数和 `new` 一起使用而且依赖于polyfill，应当多加小心。

允许 `new` 进行覆盖的部分是这里：

```

this instanceof fNOP &&
oThis ? this : oThis

// ... 和:

fNOP.prototype = this.prototype;
fBound.prototype = new fNOP();

```

我们不会实际深入解释这个花招儿是如何工作的（这很复杂而且超出了我们当前的讨论范围），但实质上这个工具判断硬绑定函数是否是用 `new` 被调用的（结果是用一个它新构建的对象作为 `this`），如果是，它就用那个新构建的 `this` 而非先前为 `this` 指定的硬绑定。

为什么 new 可以覆盖 硬绑定 这件事很有用？

这种行为的主要原因是，创建一个实质上忽略 this 的硬绑定而预先设置一部分或所有的参数的函数（这个函数可以与 new 一起使用来构建对象）。bind(..) 的一个能力是，任何在第一个 this 绑定参数之后被传入的参数，默认地作为当前函数的标准参数（技术上这称为“局部应用（partial application）”，是一种“柯里化（currying）”）。

比如：

```
function foo(p1,p2) {  
    this.val = p1 + p2;  
}  
  
// 在这里使用`null`是因为在这种场景下我们不关心`this`的硬绑定  
// 而且反正它将会被`new`调用覆盖掉！  
var bar = foo.bind( null, "p1" );  
  
var baz = new bar( "p2" );  
  
baz.val; // p1p2
```

## 判定 this

现在，我们可以按照优先顺序来总结一下从函数调用的调用点来判定 this 的规则了。按照这个顺序来问问题，然后在第一个规则适用的地方停下。

1. 函数是和 new 一起被调用的吗（new绑定）？如果是， this 就是新构建的对象。

```
var bar = new foo()
```

2. 函数是用 call 或 apply 被调用（明确绑定），甚至是隐藏在 bind 硬绑定之中吗？如果是， this 就是明确指定的对象。

```
var bar = foo.call( obj2 )
```

3. 函数是用环境对象（也称为拥有者或容器对象）被调用的吗（隐含绑定）？如果是， this 就是那个环境对象。

```
var bar = obj1.foo()
```

4. 否则，使用默认的 this （默认绑定）。如果在 strict mode 下，就是 undefined ，否则是 global 对象。  
`var bar = foo()`

以上，就是理解对于普通的函数调用来说的 this 绑定规则所需的全部。是的…几乎是全部。

## 绑定的特例

正如通常的那样，对于这些“规则”有一些例外。

在某些场景下 `this` 绑定会让人很吃惊，比如在你试图实施一种绑定，然而最终得到的是默认绑定规则的绑定行为（见前面的内容）。

## 被忽略的 `this`

如果你传递 `null` 或 `undefined` 作为 `call`，`apply` 或 `bind` 的 `this` 绑定参数，那么这些值会被忽略掉，取而代之的是默认绑定规则将适用于这个调用。

```
function foo() {
  console.log( this.a );
}

var a = 2;

foo.call( null ); // 2
```

为什么你会向 `this` 绑定故意传递像 `null` 这样的值？

使用 `apply(..)` 来将一个数组散开，从而作为函数调用的参数，是一个很常见的做法。相似地，`bind(..)` 可以 `curry` 参数（预设值），也是很有帮助的。

```
function foo(a,b) {
  console.log( "a:" + a + ", b:" + b );
}

// 将数组散开作为参数
foo.apply( null, [2, 3] ); // a:2, b:3

// 用`bind(..)`进行柯里化
var bar = foo.bind( null, 2 );
bar( 3 ); // a:2, b:3
```

这两种工具都要求第一个参数是 `this` 绑定。如果想让使用的函数不关心 `this`，你就需要一个占位值，而且正如这个代码段中展示的，`null` 看起来是一个合理的选择。

注意：虽然我们在这本书中没有涵盖，但是ES6中有一个扩散操作符：`...`。它让你无需使用 `apply(..)` 而在语法上将一个数组“散开”作为参数，比如 `foo(...[1,2])` 表示 `foo(1,2)` —— 如果 `this` 绑定没有必要，可以在语法上回避它。不幸的是，柯里化在ES6中没有语法上的替代品，所以 `bind(..)` 调用的 `this` 参数依然需要注意。

可是，在你不关心 `this` 绑定而一直使用 `null` 的时候，有些潜在的“危险”。如果你这样处理一些函数调用（比如，不归你管控的第三方包），而且那些函数确实使用了 `this` 引用，那么默认绑定规则意味着它可能会不经意间引用（或者改变，更糟糕！）`global` 对象（在浏览器

中是 `window` ) 。

很显然，这样的陷阱会导致多种 非常难诊断和追踪的Bug。

## 更安全的 `this`

也许某些“更安全”的实践是：为了 `this` 而传递一个特别建立好的对象，这个对象保证不会对你的程序产生副作用。从网络学（或军事）上借用一个词，我们可以建立一个“DMZ”（非军事区）对象——只不过是一个完全为空，没有委托（见第五，六章）的对象。

如果我们为了忽略自己认为不用关心的 `this` 绑定，而总是传递一个DMZ对象，我们就可以确定任何对 `this` 的隐藏或意外的使用将会被限制在这个空对象中，也就是说这个对象将 `global` 对象和副作用隔离开来。

因为这个对象是完全为空的，我个人喜欢给他一个变量名为 `ø`（空集合的数学符号的小写）。在许多键盘上（比如Mac的美式键盘），这个符号可以很容易地用 `⌥ + o` (`option + o`) 打出来。有些系统还允许你为某个特殊符号设置快捷键。如果你不喜欢 `ø` 符号，或者你的键盘没那么好打，你当然可以叫它任意你希望的名字。

无论你叫它什么，创建完全为空的对象的最简单方法就是 `Object.create(null)`（见第五章）。`Object.create(null)` 和 `{}` 很相似，但是没有 `Object.prototype` 的委托，所以它比 `{}` “空得更彻底”。

```
function foo(a,b) {
  console.log( "a:" + a + ", b:" + b );
}

// 我们的DMZ空对象
var ø = Object.create( null );

// 将数组散开作为参数
foo.apply( ø, [2, 3] ); // a:2, b:3

// 用`bind(..)`进行currying
var bar = foo.bind( ø, 2 );
bar( 3 ); // a:2, b:3
```

不仅在功能上更“安全”，`ø` 还会在代码风格上产生些好处，它在语义上可能会比 `null` 更清晰的表达“我想让 `this` 为空”。当然，你可以随自己喜欢来称呼你的DMZ对象。

## 间接

另外一个要注意的是，你可以（故意或非故意地！）创建对函数的“间接引用（indirect reference）”，在那样的情况下，当那个函数引用被调用时，默认绑定规则也会适用。

一个最常见的 间接引用 产生方式是通过赋值：

```

function foo() {
  console.log( this.a );
}

var a = 2;
var o = { a: 3, foo: foo };
var p = { a: 4 };

o.foo(); // 3
(p.foo = o.foo)(); // 2

```

赋值表达式 `p.foo = o.foo` 的结果值是一个刚好指向底层函数对象的引用。如此，起作用的调用点就是 `foo()`，而非你期待的 `p.foo()` 或 `o.foo()`。根据上面的结果，默认绑定规则适用。

提醒：无论你如何得到适用默认绑定的函数调用，被调用函数的内容的 strict mode 状态——而非函数的调用点——决定了 `this` 引用的值：不是 global 对象（在非 strict mode 下），就是 `undefined`（在 strict mode 下）。

## 软化绑定 (Softening Binding)

我们之前看到硬绑定是一种通过强制函数绑定到特定的 `this` 上，来防止函数调用在不经意间退回到默认绑定的策略（除非你用 `new` 去覆盖它！）。问题是，硬绑定极大地降低了函数的灵活性，阻止我们手动使用隐含绑定或后续的明确绑定尝试来覆盖 `this`。

如果有这样的办法就好了：为默认绑定提供不同的默认值（不是 global 或 `undefined`），同时保持函数可以通过隐含绑定或明确绑定技术来手动绑定 `this`。

我们可以构建一个所谓的软绑定工具来模拟我们期望的行为。

```

if (!Function.prototype.softBind) {
    Function.prototype.softBind = function(obj) {
        var fn = this,
            curried = [].slice.call( arguments, 1 ),
            bound = function bound() {
                return fn.apply(
                    (!this ||
                        (typeof window !== "undefined" &&
                            this === window) ||
                        (typeof global !== "undefined" &&
                            this === global)
                    ) ? obj : this,
                    curried.concat.apply( curried, arguments )
                );
            };
        bound.prototype = Object.create( fn.prototype );
        return bound;
    };
}

```

这里提供的 `softBind(..)` 工具的工作方式和ES5内建的 `bind(..)` 工具很相似，除了我们的软绑定行为。他用一种逻辑将指定的函数包装起来，这个逻辑在函数调用时检查 `this`，如果它是 `global` 或 `undefined`，就使用预先指定的默认值（`obj`），否则保持 `this` 不变。它也提供了可选的柯里化行为（见先前的 `bind(..)` 讨论）。

我们来看看它的用法：

```

function foo() {
    console.log("name: " + this.name);
}

var obj = { name: "obj" },
    obj2 = { name: "obj2" },
    obj3 = { name: "obj3" };

var fooOBJ = foo.softBind( obj );

fooOBJ(); // name: obj

obj2.foo = foo.softBind(obj);
obj2.foo(); // name: obj2  <---- 看!!!

fooOBJ.call( obj3 ); // name: obj3  <---- 看!

setTimeout( obj2.foo, 10 ); // name: obj  <---- 退回到软绑定

```

软绑定版本的 `foo()` 函数可以如展示的那样被手动 `this` 绑定到 `obj2` 或 `obj3`，如果默认绑定适用时会退到 `obj`。

## 词法 this

我们刚刚涵盖了一般函数遵守的4种规则。但是ES6引入了一种不适用于这些规则特殊的函数：箭头函数（arrow-function）。

箭头函数不是通过 `function` 声明的，而是通过所谓的“大箭头”操作符：`=>`。与使用4种标准的 `this` 规则不同的是，箭头函数从封闭它的（`function`或`global`）作用域采用 `this` 绑定。

我们来展示一下箭头函数的词法作用域：

```
function foo() {
  // 返回一个arrow function
  return (a) => {
    // 这里的`this`是词法上从`foo()``采用
    console.log( this.a );
  };
}

var obj1 = {
  a: 2
};

var obj2 = {
  a: 3
};

var bar = foo.call( obj1 );
bar.call( obj2 ); // 2, 不是3!
```

在 `foo()` 中创建的箭头函数在词法上捕获 `foo()` 调用时的 `this`，不管它是什么。因为 `foo()` 被 `this` 绑定到 `obj1`，`bar`（被返回的箭头函数的一个引用）也将会被 `this` 绑定到 `obj1`。一个箭头函数的词法绑定是不能被覆盖的（就连 `new` 也不行！）。

最常见的用法是用于回调，比如事件处理器或计时器：

```
function foo() {
  setTimeout(() => {
    // 这里的`this`是词法上从`foo()``采用
    console.log( this.a );
  }, 100);
}

var obj = {
  a: 2
};

foo.call( obj ); // 2
```

虽然箭头函数提供除了使用 `bind(..)` 外，另外一种在函数上来确保 `this` 的方式，这看起来很吸引人，但重要的是要注意它们本质是用被广泛理解的词法作用域来禁止了传统的 `this` 机制。在ES6之前，我们为此已经有了相当常用的模式，这些模式几乎和ES6的箭头函数的精神没有区别：

```
function foo() {
  var self = this; // 词法上捕获`this`
  setTimeout( function(){
    console.log( self.a );
  }, 100 );
}

var obj = {
  a: 2
};

foo.call( obj ); // 2
```

虽然对不想用 `bind(..)` 的人来说 `self = this` 和箭头函数都是看起来不错的“解决方案”，但它们实质上逃避了 `this` 而非理解和接受它。

如果你发现你在写 `this` 风格的代码，但是大多数或全部时候，你都用词法上的 `self = this` 或箭头函数“技巧”抵御 `this` 机制，那么也许你应该：

1. 仅使用词法作用域并忘掉虚伪的 `this` 风格代码。
2. 完全接受 `this` 风格机制，包括在必要的时候使用 `bind(..)`，并尝试避开 `self = this` 和箭头函数的“词法this”技巧。

一个程序可以有效地同时利用两种风格的代码（词法和 `this`），但是在同一个函数内部，特别是对同种类型的查找，混合这两种机制通常是自找很难维护的代码，而且可能是聪明过了头。

## 复习

为执行中的函数判定 `this` 绑定需要找到这个函数的直接调用点。找到之后，4种规则将会以这个优先顺序施用于调用点：

1. 被 `new` 调用？使用新构建的对象。
2. 被 `call` 或 `apply`（或 `bind`）调用？使用指定的对象。
3. 被持有调用的环境对象调用？使用那个环境对象。
4. 默认：`strict mode` 下是 `undefined`，否则就是全局对象。

小心偶然或不经意的默认绑定规则调用。如果你想“安全”地忽略 `this` 绑定，一个像 `o = Object.create(null)` 这样的“DMZ”对象是一个很好的占位值，来保护 `global` 对象不受意外的副作用影响。

与这4种绑定规则不同，ES6的箭头方法使用词法作用域来决定 `this` 绑定，这意味着它们采用封闭他们的函数调用作为 `this` 绑定（无论它是什么）。它们实质上是ES6之前的 `self = this` 代码的语法替代品。

# 你不懂JS: **this** 与对象原型

## 第三章：对象

在第一和第二章中，我们讲解了 `this` 绑定如何根据函数调用的调用点指向不同的对象。但究竟什么是对象，为什么我们需要指向它们？这一章我们就来详细探索一下对象。

### 语法

对象来自于两种形式：声明（字面）形式，和构造形式。

一个对象的字面语法看起来像这样：

```
var myObj = {
  key: value
  // ...
};
```

构造形式看起来像这样：

```
var myObj = new Object();
myObj.key = value;
```

构造形式和字面形式的结果是完全同种类的对象。唯一真正的区别在于你可以向字面声明一次性添加一个或多个键/值对，而对于构造形式，你必须一个一个地添加属性。

注意：像刚才展示的那样使用“构造形式”来创建对象是极其少见的。你很有可能总是想使用字面语法形式。对大多数内建的对象也一样（后述）。

### 类型

对象是大多数JS工程依赖的基本构建块儿。它们是JS的6中主要类型（在语言规范中称为“语言类型”）中的一种。

- `string`
- `number`
- `boolean`
- `null`

- `undefined`
- `object`

注意简单基本类型（`string`，`number`，`boolean`，`null`，和 `undefined`）自身不是 `object`。`null` 有时会被当成一个对象类型，但是这种误解源自于一个语言中的 Bug，它使得 `typeof null` 错误地（令人困惑地）返回字符串 `"object"`。实际上，`null` 是它自己的基本类型

一个常见的错误论断是“**JavaScript**中的一切都是对象”。这明显是不对的。

对比来看，存在几种特殊的对象子类型，我们可以称之为 复杂基本类型。

`function` 是对象的一种子类型（技术上讲，叫做“可调用对象”）。函数在 JS 中被称为“头等 (first class)”类型，就因为它们基本上就是普通的对象（附带有可调用的行为语义），而且它们可以像其他普通的对象那样被处理。

数组也是一种形式的对象，带有特别的行为。数组在内容的组织上要稍稍比一般的对象更加结构化。

## 内建对象

有几种其他的对象子类型，通常称为内建对象。对于其中的一些来说，它们的名称看起来暗示着它们和它们对应的基本类型有着直接的联系，但事实上，它们的关系更复杂，我们一会儿就开始探索。

- `String`
- `Number`
- `Boolean`
- `Object`
- `Function`
- `Array`
- `Date`
- `RegExp`
- `Error`

如果你依照和其他语言的相似性来看的话，比如 Java 语言的 `String` 类，这些内建类型有着实际类型的外观，甚至是类 (class) 的外观，

但是在 JS 中，它们实际上仅仅是内建的函数。这些内建函数的每一个都可以被用作构造器（也就是，一个函数可以和 `new` 操作符一起调用——参照第二章），其结果是一个新构建的相应子类型的对象。比如：

```

var strPrimitive = "I am a string";
typeof strPrimitive;                                // "string"
strPrimitive instanceof String;                     // false

var strObject = new String("I am a string");
typeof strObject;                                  // "object"
strObject instanceof String;                      // true

// 考察 object 子类型
Object.prototype.toString.call(strObject);        // [object String]

```

我们会在本章稍后详细地看到 `Object.prototype.toString...` 到底是如何工作的，但简单地说，我们可以通过借用基本的默认 `toString()` 方法来考察子类型的内部，而且你可以看到它揭示了 `strObject` 实际上是由 `String` 构造器创建的对象。

基本类型值 "I am a string" 不是一个对象，它是一个不可变的基本字面值。为了对它进行操作，比如检查它的长度，访问它的各个独立字符内容等等，都需要一个 `String` 对象。

幸运的是，在必要的时候语言会自动地将 "string" 基本类型转换为 `String` 对象类型，这意味着你几乎从不需要明确地创建对象。主流的JS社区都强烈推荐尽可能地使用字面形式的值，而非使用构造的对象形式。

考虑下面的代码：

```

var strPrimitive = "I am a string";

console.log(strPrimitive.length);                  // 13

console.log(strPrimitive.charAt(3));              // "m"

```

在这两个例子中，我们在字符串的基本类型上调用属性和方法，引擎会自动地将它转换为 `String` 对象，所以这些属性/方法的访问可以工作。

当使用如 `42.359.toFixed(2)` 这样的方法时，同样的转换也发生在数字基本字面量 `42` 和包装对象 `new Number(42)` 之间。同样的还有 `Boolean` 对象和 "boolean" 基本类型。

`null` 和 `undefined` 没有对象包装的形式，仅有它们的基本类型值。相比之下，`Date` 的值仅可以由它们的构造对象形式创建，因为它们没有对应的字面形式。

无论使用字面还是构造形式，`Object`，`Array`，`Function`，和 `RegExp`（正则表达式）都是对象。在某些情况下，构造形式确实会比对应的字面形式提供更多的创建选项。因为对象可以被任意一种方式创建，更简单的字面形式几乎是所有人的首选。仅仅在你需要使用额外的选项时使用构建形式。

`Error` 对象很少在代码中明示地被创建，它们通常在抛出异常时自动地被创建。它们可以由 `new Error(..)` 构造形式创建，但通常是不必要的。

## 内容

正如刚才提到的，对象的内容由存储在特定命名的位置上的（任意类型的）值组成，我们称这些值为属性。

有一个重要的事情需要注意：当我们说“内容”时，似乎暗示这些值实际上存储在对象内部，但那只不过表面现象。引擎会根据自己的实现来存储这些值，而且通常都不是把它们存储在容器对象内部。在容器内存储的是这些属性的名称，它们像指针（技术上讲，叫引用（reference））一样指向值存储的地方。

考虑下面的代码：

```
var myObject = {
  a: 2
};

myObject.a;      // 2

myObject["a"];   // 2
```

为了访问在 `myObject` 在位置 `a` 的值，我们需要使用 `.` 或 `[ ]` 操作符。`.a` 语法通常称为“属性（property）”访问，而 `["a"]` 语法通常称为“键（key）”访问。在现实中，它们俩都访问相同的位置，而且会拿出相同的值，`2`，所以这些术语可以互换使用。从现在起，我们将使用最常见的术语——“属性访问”。

两种语法的主要区别在于，`.` 操作符后面需要一个 标识符（Identifier）兼容的属性名，而 `[".."]` 语法基本可以接收任何兼容UTF-8/unicode的字符串作为属性名。举个例子，为了引用一个名为“Super-Fun!”的属性，你不得不使用 `["Super-Fun!"]` 语法访问，因为 `Super-Fun!` 不是一个合法的 `Identifier` 属性名。

而且，由于 `[".."]` 语法使用字符串的 值 来指定位置，这意味着程序可以动态地组建字符串的值。比如：

```

var wantA = true;
var myObject = {
  a: 2
};

var idx;

if (wantA) {
  idx = "a";
}

// 稍后

console.log( myObject[idx] ); // 2

```

在对象中，属性名总是字符串。如果你使用字符串以外（基本）类型的值，它会首先被转换为字符串。这甚至包括在数组中常用于索引的数字，所以要小心不要将对象和数组使用的数字搞混了。

```

var myObject = { };

myObject[true] = "foo";
myObject[3] = "bar";
myObject[myObject] = "baz";

myObject["true"];           // "foo"
myObject["3"];             // "bar"
myObject["[object Object]"]; // "baz"

```

## 计算型属性名

如果你需要将一个计算表达式作为一个键名称，那么我们刚刚描述的 `myObject[...]` 属性访问语法是十分有用的，比如 `myObject[prefix + name]`。但是当使用字面对象语法声明对象时则没有什么帮助。

ES6加入了计算型属性名，在一个字面对象声明的键名称位置，你可以指定一个表达式，用`[ ]`括起来：

```
var prefix = "foo";

var myObject = {
  [prefix + "bar"]: "hello",
  [prefix + "baz"]: "world"
};

myObject["foobar"]; // hello
myObject["foobaz"]; // world
```

计算型属性名的最常见用法，可能是用于ES6的 `Symbol`，我们将不会在本书中涵盖关于它的细节。简单地说，它们是新的基本数据类型，拥有一个不透明不可知的值（技术上讲是一个 `string` 值）。你将被强烈地不鼓励使用一个 `Symbol` 的实际值（这个值理论上会因JS引擎的不同而不同），所以 `Symbol` 的名称，比如 `Symbol.Something`（这是个瞎编的名称！），才是你会使用的：

```
var myObject = {
  [Symbol.Something]: "hello world"
};
```

## 属性（Property） vs. 方法（Method）

有些开发者喜欢在讨论对一个对象的属性访问时做一个区别，如果这个被访问的值恰好是一个函数的话。因为这诱使人们认为函数属于这个对象，而且在其他语言中，属于对象（也就是“类”）的函数被称作“方法”，所以相对于“属性访问”，我们常能听到“方法访问”。

有趣的是，语言规范也做出了同样的区别。

从技术上讲，函数绝不会“属于”对象，所以，说一个对象的引用上刚好被访问的函数自动是一个“方法”，看起来有些像是延伸了语义。

有些函数确实拥有 `this` 引用，而且有时这些 `this` 引用指向调用点的对象引用。但这个用法真的没有使这个函数比其他函数更像“方法”，因为 `this` 是在运行时在调用点动态绑定的，这使得它与这个对象的关系至多是间接的。

每次你访问一个对象的属性都是一个属性访问，无论你得到什么类型的值。如果你恰好从属性访问中得到一个函数，它也没有魔法般地在那时成为一个“方法”。一个从属性访问得来的函数没有任何特殊性（隐式 `this` 绑定之外的可能性在刚才已经解释过了）。

举个例子：

```

function foo() {
  console.log( "foo" );
}

var someFoo = foo;      // 对`foo`的变量引用

var myObject = {
  someFoo: foo
};

foo;                  // function foo(){..}

someFoo;              // function foo(){..}

myObject.someFoo;    // function foo(){..}

```

`someFoo` 和 `myObject.someFoo` 只不过是同一个函数的两个分离的引用，它们中的任何一个都不意味着这个函数很特别或被其他对象所“拥有”。如果上面的 `foo()` 定义里面拥有一个 `this` 引用，那么 `myObject.someFoo` 的隐式绑定将会是这个两个引用间唯一可以观察到的不同。它们中的任何一个都没有称为“方法”的道理。

也许有人会争辩，函数变成了方法，不是在定义期间，而是在调用的执行期间，根据它是如何在调用点被调用的（是否带有一个环境对象引用——细节见第二章）。甚至这种解读也有些牵强。

可能最安全的结论是，在JavaScript中，“函数”和“方法”是可以互换使用的。

注意：ES6加入了 `super` 引用，它通常是和 `class`（见附录A）一起使用的。`super` 的行为方式（静态绑定，而非动态绑定），给了这种说法更多的权重：一个 `super` 绑定到某处的函数比起“函数”更像一个“方法”。但是同样地，这仅仅是微妙的语义上的（和机制上的）细微区别。

就算你声明一个函数表达式作为字面对象的一部分，那个函数都不会魔法般地属于这个对象——仍然仅仅是同一个函数对象的多个引用罢了。

```

var myObject = {
  foo: function foo() {
    console.log( "foo" );
  }
};

var someFoo = myObject.foo;

someFoo;          // function foo(){..}

myObject.foo;     // function foo(){..}

```

注意：在第六章中，我们会为字面对象的 `foo: function foo(){ .. }` 声明语法介绍一种ES6的简化语法。

## 数组

数组也使用 `[ ]` 访问形式，但正如上面提到的，在存储值的方式和位置上它们的组织更加结构化（虽然仍然在存储值的类型上没有限制）。数组采用 数字索引，这意味着值被存储的位置，通常称为 下标，是一个非负整数，比如 `0` 和 `42`。

```
var myArray = [ "foo", 42, "bar" ];

myArray.length;           // 3

myArray[0];              // "foo"

myArray[2];              // "bar"
```

数组也是对象，所以即便每个索引都是正整数，你还可以在数组上添加属性：

```
var myArray = [ "foo", 42, "bar" ];

myArray.baz = "baz";

myArray.length;           // 3

myArray.baz;              // "baz"
```

注意，添加命名属性（不论是使用 `.` 还是 `[ ]` 操作符语法）不会改变数组的 `length` 所报告的值。

你可以把一个数组当做普通的键/值对象使用，并且从不添加任何数字下标，但这不是好主意，因为数组对它本来的用途有特定的行为和优化，正如普通对象那样。使用对象来存储键/值对，而用数组在数字下标上存储值。

小心：如果你试图在一个数组上添加属性，但是属性名 看起来 像一个数字，那么最终它会成为一个数字索引（也就是改变了数组的内容）：

```
var myArray = [ "foo", 42, "bar" ];

myArray["3"] = "baz";

myArray.length;           // 4

myArray[3];              // "baz"
```

## 复制对象

当开发者们初次拿起Javascript语言时，最常需要的特性就是如何复制一个对象。看起来应该有一个内建的 `copy()` 方法，对吧？但是事情实际上比这复杂一些，因为在默认情况下，复制的算法应当是什么，并不明确。

比如，考虑这个对象：

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

var anotherObject = {
  c: true
};

var anotherArray = [];

var myObject = {
  a: 2,
  b: anotherObject,    // 引用，不是拷贝！
  c: anotherArray,     // 又一个引用！
  d: anotherFunction
};

anotherArray.push( anotherObject, myObject );
```

一个 `myObject` 的拷贝究竟应该怎么表现？

首先，我们应该回答它是一个浅 (*shallow*) 还是一个深 (*deep*) 拷贝？一个浅拷贝 (*shallow copy*) 会得到一个新对象，它的 `a` 是值 `2` 的拷贝，但 `b`，`c` 和 `d` 属性仅仅是引用，它们指向被拷贝对象中引用的相同位置。一个深拷贝 (*deep copy*) 将不仅复制 `myObject`，还会复制 `anotherObject` 和 `anotherArray`。但之后我们让 `anotherArray` 拥有 `anotherObject` 和 `myObject` 的引用，所以那些也应当被复制而不是仅保留引用。现在由于循环引用，我们得到了一个无限循环复制的问题。

我们应当检测循环引用并打破循环遍历吗（不管位于深处的，没有完全复制的元素）？我们应当报错退出吗？或者介于两者之间？

另外，“复制”一个函数意味着什么，也不是很清楚。有一些技巧，比如提取一个函数源代码的 `toString()` 序列化表达（这个源代码会因实现不同而不同，而且根据被考察的函数的类型，其结果甚至在所有引擎上都不可靠）。

那么我们如何解决所有这些刁钻的问题？不同的JS框架都各自挑选自己的解释并且做出自己的选择。但是哪一种（如果有的话）才是JS应当作为标准采用的呢？长久以来，没有明确答案。

一个解决方案是，JSON安全的对象（也就是，可以被序列化为一个JSON字符串，之后还可以被重新变换为拥有相同的结构和值的对象）可以简单地这样复制：

```
var newObj = JSON.parse( JSON.stringify( someObj ) );
```

当然，这要求你保证你的对象是JSON安全的。对于某些情况，这没什么大不了的。而对另一些情况，这还不够。

同时，浅拷贝相当易懂，而且没有那么多问题，所以ES6为此任务已经定义了 `Object.assign(..)`。 `Object.assign(..)` 接收目标对象作为第一个参数，然后是一个或多个源对象作为后续参数。它会在源对象上迭代所有的可枚举 (*enumerable*)，*owned keys* (直接拥有的键)，并把它们拷贝到目标对象上 (仅通过 `=` 赋值)。它还会很方便地返回目标对象，正如下面你可以看到的：

```
var newObj = Object.assign( {}, myObject );

newObj.a;                      // 2
newObj.b === anotherObject;     // true
newObj.c === anotherArray;      // true
newObj.d === anotherFunction;   // true
```

注意：在下一部分中，我们将讨论“属性描述符 (property descriptors)”并展示 `Object.defineProperty(..)` 的使用。然而在 `Object.assign(..)` 中发生的复制是单纯的 `=` 式赋值，所以任何在源对象属性的特殊性质 (比如 `writable`) 在目标对象上都不会保留。

## 属性描述符 (Property Descriptors)

在ES5之前，JavaScript语言没有给出直接的方法，让你的代码可以考察或描述属性的性质间的区别，比如属性是否为只读。

在ES5中，所有的属性都用 属性描述符 (Property Descriptors) 来描述。

考虑这段代码：

```
var myObject = {
  a: 2
};

Object.getOwnPropertyDescriptor( myObject, "a" );
// {
//   value: 2,
//   writable: true,
//   enumerable: true,
//   configurable: true
// }
```

正如你所见，我们普通的对象属性 `a` 的属性描述符（称为“数据描述符”，因为它仅持有一个数据值）的内容要比 `value` 为 `2` 多得多。它还包含另外3个性质：`writable`，`enumerable`，和 `configurable`。

当我们创建一个普通属性时，可以看到属性描述符的各种性质的默认值，我们可以用 `Object.defineProperty(..)` 来添加新属性，或使用期望的性质来修改既存的属性（如果它是 `configurable` 的！）。

举例来说：

```
var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: true,
    enumerable: true
} );
myObject.a; // 2
```

使用 `defineProperty(..)`，我们手动明确地在 `myObject` 上添加了一个直白的，普通的 `a` 属性。然而，你通常不会使用这种手动方法，除非你想要把描述符的某个性质修改为不同的值。

## 可写性 (Writable)

`writable` 控制着你改变属性值的能力。

考虑这段代码：

```
var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // 不可写 !
    configurable: true,
    enumerable: true
} );
myObject.a = 3;
myObject.a; // 2
```

如你所见，我们对 `value` 的修改悄无声息地失败了。如果我们在 `strict mode` 下进行尝试，会得到一个错误：

```
"use strict";

var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // not writable!
    configurable: true,
    enumerable: true
} );
myObject.a = 3; // TypeError
```

这个 `TypeError` 告诉我们，我们不能改变一个不可写属性。

注意：我们一会儿就会讨论`getters/setters`，但是简单地说，你可以观察到 `writable:false` 意味着值不可改变，和你定义一个空的`setter`是有些等价的。实际上，你的空`setter`在被调用时需要扔出一个 `TypeError`，来和 `writable:false` 保持一致。

## 可配置性（Configurable）

只要属性当前是可配置的，我们就可以使用同样的 `defineProperty(..)` 工具，修改它的描述符定义。

```
var myObject = {
    a: 2
};

myObject.a = 3;
myObject.a; // 3

Object.defineProperty( myObject, "a", {
    value: 4,
    writable: true,
    configurable: false, // 不可配置！
    enumerable: true
} );

myObject.a; // 4
myObject.a = 5;
myObject.a; // 5

Object.defineProperty( myObject, "a", {
    value: 6,
    writable: true,
    configurable: true,
    enumerable: true
} ); // TypeError
```

最后的 `defineProperty(..)` 调用导致了一个 `TypeError`，这与 `strict mode` 无关，如果你试图改变一个不可配置属性的描述符定义，就会发生 `TypeError`。要小心：如你所看到的，将 `configurable` 设置为 `false` 是一个单向操作，不可撤销！

注意：这里有一个需要注意的微小例外：即便属性已经是 `configurable:false`，`writable` 总是可以没有错误地从 `true` 改变为 `false`，但如果已经是 `false` 的话不能变回 `true`。

`configurable:false` 阻止的另外一个事情是使用 `delete` 操作符移除既存属性的能力。

```
var myObject = {
  a: 2
};

myObject.a; // 2
delete myObject.a;
myObject.a; // undefined

Object.defineProperty( myObject, "a", {
  value: 2,
  writable: true,
  configurable: false,
  enumerable: true
} );

myObject.a; // 2
delete myObject.a;
myObject.a; // 2
```

如你所见，最后的 `delete` 调用失败了（无声地），因为我们将 `a` 属性设置成了不可配置。

`delete` 仅用于直接从目标对象移除该对象的属性（可以被移除的属性）。如果一个对象的属性是某个其他对象/函数的最后一个现存的引用，而你 `delete` 了它，那么这就移除了这个引用，于是现在那个没有被任何地方引用的对象/函数就可以被作为垃圾回收。但是，将 `delete` 当做一个像其他语言（如C/C++）中那样的释放内存工具是不正确的。`delete` 仅仅是一个对象属性移除操作——没有更多别的含义。

## 可枚举性（`Enumerable`）

我们将要在这里提到的最后一个描述符性质是 `enumerable`（还有另外两个，我们将在一会儿讨论 `getter/setters` 时谈到）。

它的名称可能已经使它的功能很明显了，这个性质控制着一个属性是否能在特定的对象属性枚举操作中出现，比如 `for..in` 循环。设置为 `false` 将会阻止它出现在这样的枚举中，即使它依然完全是可以访问的。设置为 `true` 会使它出现。

所有普通的用户定义属性都默认是可 `enumerable` 的，正如你通常希望的那样。但如果你有一个特殊的属性，你想让它对枚举隐藏，就将它设置为 `enumerable:false`。

我们一会儿就更加详细地演示可枚举性，所以在大脑中给这个话题上打一个书签。

## 不可变性 (Immutability)

有时我们希望将属性或对象（有意或无意地）设置为不可改变的。ES5用几种不同的微妙方式，加入了对此功能的支持。

一个重要的注意点是：所有这些方法都创建的是浅不可变性。也就是说，它们仅影响对象和它的直属属性的性质。如果对象拥有对其他对象（数组，对象，函数等）的引用，那个对象的内容不会受影响，任然保持可变。

```
myImmutableObject.foo; // [1,2,3]
myImmutableObject.foo.push( 4 );
myImmutableObject.foo; // [1,2,3,4]
```

在这段代码中，我们假想 `myImmutableObject` 已经被创建，而且被保护为不可变。但是，为了保护 `myImmutableObject.foo` 的内容（也是一个对象——数组），你将需要使用下面的一个或多个方法将 `foo` 设置为不可变。

注意：在JS程序中创建完全不可动摇的对象是不那么常见的。有些特殊情况当然需要，但作为一个普通的设计模式，如果你发现自己想要封印 (`seal`) 或冻结 (`freeze`) 你所有的对象，那么你可能想要退一步来重新考虑你的程序设计，让它对对象值的潜在变化更加健壮。

## 对象常量 (Object Constant)

通过将 `writable:false` 与 `configurable:false` 组合，你可以实质上创建了一个作为对象属性的常量（不能被改变，重定义或删除），比如：

```
var myObject = {};
Object.defineProperty( myObject, "FAVORITE_NUMBER", {
  value: 42,
  writable: false,
  configurable: false
} );
```

## 防止扩展 (Prevent Extensions)

如果你想防止一个对象被添加新的属性，但另一方面保留其他既存的对象属性，调用 `Object.preventExtensions(...)`：

```

var myObject = {
  a: 2
};

Object.preventExtensions( myObject );

myObject.b = 3;
myObject.b; // undefined

```

在 `非-strict mode` 模式下，`b` 的创建会无声地失败。在 `strict mode` 下，它会抛出 `TypeError`。

## 封印 (Seal)

`Object.seal(..)` 创建一个“封印”的对象，这意味着它实质上在当前的对象上调用 `Object.preventExtensions(..)`，同时也将它所有的既存属性标记为 `configurable:false`。

所以，你既不能添加更多的属性，也不能重新配置或删除既存属性（虽然你依然可以修改它们的值）。

## 冻结 (Freeze)

`Object.freeze(..)` 创建一个冻结的对象，这意味着它实质上在当前的对象上调用 `Object.seal(..)`，同时也将它所有的“数据访问”属性设置为 `writable:false`，所以他们的值不可改变。

这种方法是你可以从对象自身获得的最高级别的不可变性，因为它阻止任何对对象或对象的直属属性的改变（虽然，如上面提到的，任何被引用的对象的内容不受影响）。

你可以“深度冻结”一个对象：在这个对象上调用 `Object.freeze(..)`，然后递归地迭代所有它引用的对象（目前还没有受过影响的），然后在它们上也调用 `Object.freeze(..)`。但是要小心，这可能会影响其他（共享的）你并不打算影响的对象。

## [[Get]]

关于属性访问如何工作有一个重要的细节。

考虑下面的代码：

```

var myObject = {
  a: 2
};

myObject.a; // 2

```

`myObject.a` 是一个属性访问，但是它并不是看起来那样，仅仅在 `myObject` 中寻找一个名为 `a` 的属性。

根据语言规范，上面的代码实际上在 `myObject` 上执行了一个 `[[Get]]` 操作（有些像 `[[Get]]()` 函数调用）。对一个对象进行默认的内建 `[[Get]]` 操作，会首先检查对象，寻找一个拥有被请求的名称的属性，如果找到，就返回相应的值。

然而，如果按照被请求的名称没能找到属性，`[[Get]]` 的算法定义了另一个重要的行为。我们会在第五章来解释接下来会发生什么（遍历 `[[Prototype]]` 链，如果有的话）。

但 `[[Get]]` 操作的一个重要结果是，如果它通过任何方法都不能找到被请求的属性的值，那么它会返回 `undefined`。

```
var myObject = {
  a: 2
};

myObject.b; // undefined
```

这个行为和你通过标识符名称来引用变量不同。如果你引用了一个在可用的词法作用域内无法解析的变量，其结果不是像对象属性那样返回 `undefined`，而是抛出 `ReferenceError`。

```
var myObject = {
  a: undefined
};

myObject.a; // undefined

myObject.b; // undefined
```

从值的角度来说，这两个引用没有区别——它们的结果都是 `undefined`。然而，在 `[[Get]]` 操作的底层，虽然不明显，但是比起处理引用 `myObject.a`，处理 `myObject.b` 的操作要多做一些潜在的工作。

如果仅仅考察结果的值，你无法分辨一个属性是存在并持有一个 `undefined` 值，还是因为属性根本不存在所以 `[[Get]]` 无法返回某个特定值而返回默认的 `undefined`。但是，你很快就能看到你其实可以分辨这两种场景。

## [[Put]]

既然为了从一个属性中取得值而存在一个内部定义的 `[[Get]]` 操作，那么很明显应该也存在一个默认的 `[[Put]]` 操作。

这很容易让人认为，给一个对象的属性赋值，将会在这个对象上调用 `[[Put]]` 来设置或创建这个属性。但是实际情况却有一些微妙的不同。

调用 `[[Put]]` 时，它根据几个因素表现不同的行为，包括（影响最大的）属性是否已经在对象中存在了。

如果属性存在，`[[Put]]` 算法将会大致检查：

1. 这个属性是访问器描述符吗（见下一节"Getters 与 Setters"）？如果是，而且是**setter**，就调用**setter**。
2. 这个属性是 `writable` 为 `false` 数据描述符吗？如果是，在非 `strict mode` 下无声地失败，或者在 `strict mode` 下抛出 `TypeError`。
3. 否则，像平常一样设置既存属性的值。

如果属性在当前的对象中不存在，`[[Put]]` 操作会变得更微妙和复杂。我们将在第五章讨论 `[[Prototype]]` 时再次回到这个场景，更清楚地解释它。

## Getters 与 Setters

对象默认的 `[[Put]]` 和 `[[Get]]` 操作分别完全控制着如何设置既存或新属性的值，和如何取得既存属性。

注意：使用较先进的语言特性，覆盖整个对象（不仅是每个属性）的默认 `[[Put]]` 和 `[[Get]]` 操作是可能的。这超出了我们要在这本书中讨论的范围，但我们在后面的“你不懂JS”系列中涵盖此内容。

ES5引入了一个方法来覆盖这些默认操作的一部分，但不是在对象级别而是针对每个属性，就是通过**getters**和**setters**。**Getter**是实际上调用一个隐藏函数来取得值的属性。**Setter**是实际上调用一个隐藏函数来设置值的属性。

当你将一个属性定义为拥有**getter**或**setter**或两者兼备，那么它的定义就成为了“访问器描述符”（与“数据描述符”相对）。对于访问器描述符，它的 `value` 和 `writable` 性质没有意义而被忽略，取而代之的是JS将会考虑属性的 `set` 和 `get` 性质（还有 `configurable` 和 `enumerable`）。

考虑下面的代码：

```

var myObject = {
    // 为`a`定义一个getter
    get a() {
        return 2;
    }
};

Object.defineProperty(
    myObject,      // 目标对象
    "b",          // 属性名
    {             // 描述符
        // 为`b`定义getter
        get: function(){ return this.a * 2 },
        // 确保`b`作为对象属性出现
        enumerable: true
    }
);

myObject.a; // 2

myObject.b; // 4

```

不管是通过在字面对象语法中使用 `get a() { .. }`，还是通过使用 `defineProperty(..)` 明确定义，我们都在对象上创建了一个没有实际持有值的属性，访问它们将会自动地对getter函数进行隐藏的函数调用，其返回的任何值就是属性访问的结果。

```

var myObject = {
    // 为`a`定义getter
    get a() {
        return 2;
    }
};

myObject.a = 3;

myObject.a; // 2

```

因为我们仅为 `a` 定义了一个getter，如果之后我们试着设置 `a` 的值，赋值操作并不会抛出错误而是无声地将赋值废弃。就算这里有一个合法的setter，我们的自定义getter将返回值硬编码为仅返回 `2`，所以赋值操作是没有意义的。

为了使这个场景更合理，正如你可能期望的那样，每个属性还应当被定义一个覆盖默认 `[[Put]]` 操作（也就是赋值）的setter。几乎可确定，你将总是想要同时声明getter和setter（仅有它们中的一个经常会导致以外的行为）：

```

var myObject = {
    // 为 `a` 定义getter
    get a() {
        return this._a_;
    },
    // 为 `a` 定义setter
    set a(val) {
        this._a_ = val * 2;
    }
};

myObject.a = 2;

myObject.a; // 4

```

注意：在这个例子中，我们实际上将赋值操作（`[[Put]]` 操作）指定的值 `2` 存储到了另一个变量 `_a_` 中。`_a_` 这个名称只是用在这个例子中的单纯的惯例，并不意味着它的行为有什么特别之处——它和其他普通属性没有区别。

## 存在性 (Existence)

我们早先看到，像 `myObject.a` 这样的属性访问可能会得到一个 `undefined` 值，无论是它明确存储着 `undefined` 还是属性 `a` 根本就不存在。那么，如果这两种情况的值相同，我们还怎么区别它们呢？

我们可以查询一个对象是否拥有特定的属性，而不必取得那个属性的值：

```

var myObject = {
    a: 2
};

("a" in myObject);           // true
("b" in myObject);           // false

myObject.hasOwnProperty("a"); // true
myObject.hasOwnProperty("b"); // false

```

`in` 操作符会检查属性是否存在于对象中，或者是否存在于 `[[Prototype]]` 链对象遍历的更高层中（详见第五章）。相比之下，`hasOwnProperty(..)` 仅仅检查 `myObject` 是否拥有属性，但不会查询 `[[Prototype]]` 链。我们会在第五章详细讲解 `[[Prototype]]` 时，回来讨论这两个操作重要的不同。

通过委托到 `Object.prototype`，所有的普通对象都可以访问 `hasOwnProperty(..)`（详见第五章）。但是创建一个不链接到 `Object.prototype` 的对象也是可能的（通过 `Object.create(null)` ——详见第五章）。这种情况下，像 `myObject.hasOwnProperty(..)` 这

样的方法调用将会失败。

在这种场景下，一个进行这种检查的更健壮的方式

是 `Object.prototype.hasOwnProperty.call(myObject, "a")`，它借用基本的 `hasOwnProperty(..)` 方法而且使用明确的 `this` 绑定（详见第二章）来对我们的 `myObject` 实施这个方法。

注意：`in` 操作符看起来像是要检查一个值在容器中的存在性，但是它实际上检查的是属性名的存在性。在使用数组时注意这个区别十分重要，因为我们会有很多的冲动来进行 `4 in [2, 4, 6]` 这样的检查，但是这总是不像我们想象的那样工作。

## 枚举 (Enumeration)

先前，在学习 `enumerable` 属性描述符性质时，我们简单地解释了“可枚举性 (enumerability)”的含义。现在，让我们来更加详细地重新审视它。

```
var myObject = { };

Object.defineProperty(
  myObject,
  "a",
  // 使 `a` 可枚举，如一般情况
  { enumerable: true, value: 2 }
);

Object.defineProperty(
  myObject,
  "b",
  // 使 `b` 不可枚举
  { enumerable: false, value: 3 }
);

myObject.b; // 3
("b" in myObject); // true
myObject.hasOwnProperty("b"); // true

// .....

for (var k in myObject) {
  console.log(k, myObject[k]);
}

// "a" 2
```

你会注意到，`myObject.b` 实际上存在，而且拥有可以访问的值，但是它不出现在 `for..in` 循环中（然而令人诧异的是，它的 `in` 操作符的存在性检查通过了）。这是因为“enumerable”基本上意味着“如果对象的属性被迭代时会被包含在内”。

注意：将 `for..in` 循环实施在数组上可能会给出意外的结果，因为枚举一个数组将不仅包含所有的数字下标，还包含所有的可枚举属性。所以一个好主意是：将 `for..in` 循环仅用于对象，而为存储在数组中的值使用传统的 `for` 循环并用数字索引迭代。

意外一个可以区分可枚举和不可枚举属性的方法是：

```
var myObject = { };

Object.defineProperty(
  myObject,
  "a",
  // 使 `a` 可枚举，如一般情况
  { enumerable: true, value: 2 }
);

Object.defineProperty(
  myObject,
  "b",
  // 使 `b` 不可枚举
  { enumerable: false, value: 3 }
);

myObject.propertyIsEnumerable( "a" ); // true
myObject.propertyIsEnumerable( "b" ); // false

Object.keys( myObject ); // ["a"]
Object.getOwnPropertyNames( myObject ); // ["a", "b"]
```

`propertyIsEnumerable(..)` 测试一个给定的属性名是否直接存在于对象上，并且是 `enumerable:true`。

`Object.keys(..)` 返回一个所有可枚举属性的数组，而 `Object.getOwnPropertyNames(..)` 返回一个所有属性的数组，不论能不能枚举。

`in` 和 `hasOwnProperty(..)` 区别于它们是否查询 `[[Prototype]]` 链，而 `Object.keys(..)` 和 `Object.getOwnPropertyNames(..)` 都只考察直接给定的对象。

(当下) 没有与 `in` 操作符的查询方式 (在整个 `[[Prototype]]` 链上遍历所有的属性，如我们在第五章解释的) 等价的，内建的方法可以得到一个所有属性的列表。你可以近似地模拟一个这样的工具：递归地遍历一个对象的 `[[Prototype]]` 链，在每一层都从 `Object.keys(..)` 中取得一个列表——仅包含可枚举属性。

## 迭代 (Iteration)

`for..in` 循环迭代一个对象上 (包括它的 `[[Prototype]]` 链) 所有的可迭代属性。但如果你想要迭代值呢？

在数字索引的数组中，典型的迭代所有的值的办法是使用标准的 `for` 循环，比如：

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

for (var i = 0; i < myArray.length; i++) {
    console.log( myArray[i] );
}

// 1 2 3
```

但是这并没有迭代所有的值，而是迭代了所有的下标，然后由你使用索引来引用值，比如 `myArray[i]`。

ES5还为数组加入了几个迭代帮助方法，包括 `forEach(..)`，`every(..)`，和 `some(..)`。这些帮助方法的每一个都接收一个回调函数，这个函数将施用于数组中的每一个元素，仅在如何响应回调的返回值上有所不同。

`forEach(..)` 将会迭代数组中所有的值，并且忽略回调的返回值。`every(..)` 会一直迭代到最后，或者当回调返回一个 `false`（或“falsy”）值，而 `some(..)` 会一直迭代到最后，或者当回调返回一个 `true`（或“truthy”）值。

这些在 `every(..)` 和 `some(..)` 内部的特殊返回值有些像普通 `for` 循环中的 `break` 语句，它们可以在迭代执行到末尾之前将它结束掉。

如果你使用 `for..in` 循环在一个对象上进行迭代，你也只能间接地得到值，因为它实际上仅仅迭代对象的所有可枚举属性，让你自己手动地去访问属性来得到值。

注意：与以有序数字的方式（`for` 循环或其他迭代器）迭代数组的下标比较起来，迭代对象属性的顺序是不确定的，而且可能会因JS引擎的不同而不同。对于需要跨平台环境保持一致性的问题，不要依赖观察到的顺序，因为这个顺序是不可靠的。

但是如果你想直接迭代值，而不是数组下标（或对象属性）呢？ES6加入了一个有用的 `for..of` 循环语法，用来迭代数组（和对象，如果这个对象有定义的迭代器）：

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

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

// 1
// 2
// 3
```

`for..of` 循环要求被迭代的东西提供一个迭代器对象（从一个在语言规范中叫做 `@@iterator` 的默认内部函数那里得到），每次循环都调用一次这个迭代器对象的 `next()` 方法，循环迭代的内容就是这些连续的返回值。

数组拥有内建的 `@@iterator`，所以正如展示的那样，`for..of` 对于它们很容易使用。但是让我们使用内建的 `@@iterator` 来手动迭代一个数组，来看看它是怎么工作的：

```
var myArray = [ 1, 2, 3 ];
var it = myArray[Symbol.iterator]();

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

注意：我们使用一个ES6的 `Symbol : Symbol.iterator` 来取得一个对象的 `@@iterator` 内部属性。我们在本章中简单地提到过 `Symbol` 的语义（见“计算型属性名”），同样的原理适用这里。你总是希望通过 `Symbol` 名称引用，而不是它可能持有的特殊的值，来引用这样特殊的属性。同时，与这个名称的含义无关，`@@iterator` 本身不是迭代器对象，而是一个返回迭代器对象的方法——一个重要的细节！

正如上面的代码段揭示的，迭代器的 `next()` 调用的返回值是一个 `{ value: ... , done: ... }` 形式的对象，其中 `value` 是当前迭代的值，而 `done` 是一个 `boolean`，表示是否还有更多内容可以迭代。

注意值 `3` 和 `done:false` 一起返回，猛地一看会有些奇怪。你不得不第四次调用 `next()`（在前一个代码段的 `for..of` 循环会自动这样做）来得到 `done:true`，而使自己知道迭代已经完成。这个特别之处的原因超出了我们要在这里讨论的范围，但是它来自于ES6生成器函数的语义。

虽然数组可以在 `for..of` 循环中自动迭代，但普通的对象没有内建的 `@@iterator`。这种故意省略的原因要比我们将在这里解释的更复杂，但一般来说，为了未来的对象类型，最好不要加入那些可能最终被证明是麻烦的实现。

但是可以为你想要迭代的对象定义你自己的默认 `@@iterator`。比如：

```

var myObject = {
  a: 2,
  b: 3
};

Object.defineProperty( myObject, Symbol.iterator, {
  enumerable: false,
  writable: false,
  configurable: true,
  value: function() {
    var o = this;
    var idx = 0;
    var ks = Object.keys( o );
    return {
      next: function() {
        return {
          value: o[ks[idx++]],
          done: (idx > ks.length)
        };
      }
    };
  }
} );

// 手动迭代`myObject`
var it = myObject[Symbol.iterator]();
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { value:undefined, done:true }

// 用`for..of`迭代`myObject`
for (var v of myObject) {
  console.log( v );
}
// 2
// 3

```

注意：我们使用了 `Object.defineProperty(..)` 来自定义我们的 `@@iterator`（很大程度上是因为我们可以将它指定为不可枚举的），但是通过将 `Symbol` 作为一个计算型属性名（在本章前面的部分讨论过），我们也可以直接声明它，比如 `var myObject = { a:2, b:3, [Symbol.iterator]: function(){ /* .. */ } }`。

每次 `for..of` 循环在 `myObject` 的迭代器对象上调用 `next()` 时，迭代器内部的指针将会向前移动并返回对象属性列表的下一个值（关于对象属性/值迭代顺序，参照前面的注意事项）。

我们刚刚演示的迭代，是一个简单的一个值一个值的迭代，当然你可以为你的自定义数据结构定义任意复杂的迭代方法，只要你觉得合适。对于操作用自定义对象来说，自定义迭代器与ES6的 `for..of` 循环相组合，是一个新的强大的语法工具。

举个例子，一个 `Pixel` (像素) 对象列表（拥有 `x` 和 `y` 的坐标值）可以根据距离原点  $(0, 0)$  的直线距离决定它的迭代顺序，或者过滤掉那些“太远”的点，等等。只要你的迭代器从 `next()` 调用返回期望的 `{ value: ... }` 返回值，并在迭代结束后返回一个 `{ done: true }` 值，ES6 的 `for..of` 循环就可以迭代它。

其实，你甚至可以生成一个永远不会“结束”，并且总会返回一个新值（比如随机数，递增值，唯一的识别符等等）的“无穷”迭代器，虽然你可能不会将这样的迭代器用于一个没有边界 `for..of` 循环，因为它永远不会结束，而且会阻塞你的程序。

```
var randoms = {
  [Symbol.iterator]: function() {
    return {
      next: function() {
        return { value: Math.random() };
      }
    };
  }
};

var randoms_pool = [];
for (var n of randoms) {
  randoms_pool.push( n );

  // 不要超过边界！
  if (randoms_pool.length === 100) break;
}
```

这个迭代器会“永远”生成随机数，所以我们小心地仅从中取出100个值，以使我们的程序不被阻塞。

## 复习

JS中的对象拥有字面形式（比如 `var a = { .. }`），和构造形式（比如 `var a = new Array(..)`）。字面形式几乎总是首选，但在某些情况下，构造形式提供更多的构建选项。

许多人错误地声称“Javascript中的一切都是对象”，这是不对的。对象是6种（或7中，看你从哪个方面说）基本类型之一。对象有子类型，包括 `function`，还可以被行为特化，比如 `[object Array]` 作为内部的标签表示子类型数组。

对象是键/值对的集合。通过 `.propName` 或 `["propName"]` 语法，值可以作为属性访问。不管属性什么时候被访问，引擎实际上会调用内部默认的 `[[Get]]` 操作（在设置值时调用 `[[Put]]` 操作），它不仅直接在对象上查找属性，在没有找到时还会遍历 `[[Prototype]]` 链（见第五章）。

属性有一些可以通过属性描述符控制的特定性质，比如 `writable` 和 `configurable`。另外，对象拥有它的不可变性（它们的属性也有），可以通过使用 `Object.preventExtensions(..)`，`Object.seal(..)`，和 `Object.freeze(..)` 来控制几种不同等级的不可变性。

属性不必非要包含值——它们也可以是带有`getter/setter`的“访问器属性”。它们也可以是可枚举或不可枚举的，这控制它们是否会在 `for..in` 这样的循环迭代中出现。

你也可以使用ES6的 `for..of` 语法，在数据结构（数组，对象等）中迭代值，它寻找一个内建或自定义的 `@@iterator` 对象，这个对象由一个 `next()` 方法组成，通过这个 `next()` 方法每次迭代一个数据。

# 你不懂JS: **this** 与对象原型

## 第四章：混合（淆）“类”的对象

接着我们上一章对对象的探索，我们很自然的将注意力转移到“面向对象（OO）编程”，与“类（class）”。我们先将“面向类”作为设计模式来看看，之后我们再考察“类”的机制：“实例化（instantiation）”，“继承（inheritance）”与“相对多态(relative polymorphism)”。

我们将会看到，这些概念并不是非常自然地映射到JS的对象机制上，以及许多JavaScript开发者为了克服这些挑战所做的努力（mixins等）。

注意：这一章花了相当一部分时间（前一半！）在着重解释“面向对象编程”理论上。在后半部分讨论“Mixins（混合）”时，我们最终会将这些理论与真实且实际的JavaScript代码联系起来。但是这里首先要跨过许多概念和假想代码，所以可别迷路了——坚持下去！

### 类理论

“类/继承”描述了一种特定的代码组织和结构形式——一种在我们的软件中对真实世界的建模方法。

OO或者面向类的编程强调数据和操作它的行为之间有固有的联系（当然，依数据的类型和性质不同而不同！），所以合理的设计是将数据和行为打包在一起（也称为封装）。这有时在正式的计算机科学中称为“数据结构”。

比如，表示一个单词或短语的一系列字符通常称为“`string`（字符串）”。这些字符就是数据。但你几乎从来不关心数据，你总是想对数据做事情，所以可以向数据实施的行为（计算它的长度，在末尾添加数据，检索，等等）都被设计成为 `String` 类的方法。

任何给定的字符串都是这个类的一个实例，这个类是一个整齐的集合包装：字符数据和我们可以对它进行的操作功能。

类还隐含着对一个特定数据结构的一种分类方法。我们这么做的方法是，将一个给定的结构考虑为一个更加泛化的基础定义的具体种类。

让我们通过一个最常被引用的例子来探索这种分类处理。一辆车可以被描述为一“类”更泛化的东西——载具——的具体实现。

我们在软件中通过定义 `Vehicle` 类和 `car` 类来模型化这种关系。

`Vehicle` 的定义可能会包含像动力 (引擎等) , 载人能力等等, 这些都是行为。我们在 `Vehicle` 中定义的都是所有 (或大多数) 不同类型的载具 (飞机, 火车, 机动车) 都共同拥有的东西。

在我们的软件中为每一种不同类型的载具一次又一次地重定义“载人能力”这个基本性质可能没有道理。反而, 我们在 `Vehicle` 中把这个能力定义一次, 之后当我们定义 `car` 时, 我们简单地指出它从基本的 `Vehicle` 定义中“继承” (或“扩展”)。`car` 的定义就是特化了一般的 `Vehicle` 定义。

虽然 `Vehicle` 和 `car` 用方法的形式集约地定义了行为, 但一个实例中的数据就像一个唯一的车牌号一样属于一辆具体的车。

这样, 类, 继承, 和实例化就诞生了。

另一个关于类的关键概念是“多态 (polymorphism)”, 它描述这样的想法: 一个来自于父类的泛化行为可以被子类覆盖, 从而使它更加具体。实际上, 相对多态让我们在覆盖行为中引用基础行为。

类理论强烈建议父类和子类对相同的行为共享同样的方法名, 以便于子类 (差异化地) 覆盖父类。我们即将看到, 在你的JavaScript代码中这么做会导致种种困难和脆弱的代码。

## “类”设计模式

你可能从没把类当做一种“设计模式”考虑过, 因为最常见的是关于流行的“面向对象设计模式”的讨论, 比如“迭代器 (Iterator)”, “观察者 (Observer)”, “工厂 (Factory)”, “单例 (Singleton)”等等。当以这种方式表现时, 几乎可以假定OO的类是我们实现所有 (高级) 设计模式的底层机制, 好像对所有代码来说OO是一个给定的基础。

取决于你在编程方面接受过的正规教育的水平, 你可能听说过“过程式编程 (procedural programming)”: 一种不用任何高级抽象, 仅仅由过程 (也就是函数) 调用其他函数来构成的描述代码的方式。你可能被告知过, 类是一个将过程式风格的“面条代码”转换为结构良好, 组织良好代码的恰当的方法。

当然, 如果你有“函数式编程 (functional programming)”的经验, 你可能知道类只是几种常见设计模式中的一种。但是对于其他人来说, 这可能是第一次你问自己, 类是否真的是代码的根本基础, 或者它们是在代码顶层上的选择性抽象。

有些语言 (比如Java) 不给你选择, 所以这根本没什么选择性——一切都是类。其他语言如C/C++或PHP同时给你过程式和面向类的语法, 在使用哪种风格合适或混合风格上, 留给开发者更多选择。

## JavaScript的“类”

在这个问题上JavaScript属于哪一边？JS拥有一些像类的语法元素（比如 `new` 和 `instanceof`）有一阵子了，而且在最近的ES6中，还有一些追加的，比如 `class` 关键字（见附录A）。

但这意味着JavaScript实际上拥有类吗？直白且简单：没有。

由于类是一种设计模式，你可以，用相当的努力（我们将在本章剩下的部分看到），近似实现很多经典类的功能。JS在通过提供看起来像类的语法，来努力满足用类进行设计的极其广泛的渴望。

虽然我们好像有了看起来像类的语法，但是好像JavaScript机制在抵抗你使用类设计模式，因为在底层，这些你正在上面工作的机制运行的十分不同。语法糖和（极其广泛被使用的）JS“Class”库废了很大力气来把这些真实情况对你隐藏起来，但你迟早会面对现实：你在其他语言中遇到的类和你在JS中模拟的“类”不同。

总而言之，类是软件设计中的一种可选模式，你可以选择在JavaScript中使用或不使用它。因为许多开发者都对面向类的软件设计情有独钟，我们将在本章剩下的部分中探索一下，为了使用JS提供的东西维护类的幻觉要付出什么代价，和我们经历的痛苦。

## 类机制

在许多面向类语言中，“标准库”都提供一个叫“栈”（压栈，弹出等）的数据结构，用一个 `stack` 类表示。这个类拥有一组变量来存储数据，还拥有一组可公开访问的行为（“方法”），这些行为使你的代码有能力与（隐藏的）数据互动（添加或移除数据等等）。

但是在这样的语言中，你不是直接在 `stack` 上操作（除非制造一个静态的类成员引用，但这超出了我们要讨论的范围）。`Stack` 类仅仅是任何的“栈”都会做的事情的一个抽象解释，但它本身不是一个“栈”。为了得到一个可以对之进行操作的实在的数据结构，你必须实例化这个 `stack` 类。

## 建筑物

传统的“类（class）”和“实例（instance）”的比拟源自于建筑物的建造。

一个建筑师会规划出一栋建筑的所有性质：多宽，多高，在哪里有多少窗户，甚至墙壁和天花板用什么材料。在这个时候，她并不关心建筑物将被建造在哪里，她也不关心有多少这栋建筑的拷贝将被建造。

同时她也不关心这栋建筑的内容——家具，墙纸，吊扇等等——她仅关心建筑物含有何种结构。

她生产的建筑学上的蓝图仅仅是建筑物的“方案”。它们不实际构成我们可以进入其中并坐下的建筑物。为了这个任务我们需要一个建筑工。建筑工会拿走方案并精确地依照它们建造这栋建筑物。在真正的意义上，他是在将方案中意图的性质 **拷贝** 到物理建筑物中。

一旦完成，这栋建筑就是蓝图方案的一个物理实例，一个有望是实质完美的拷贝。然后建筑工就可以移动到隔壁将它再重做一遍，建造另一个拷贝。

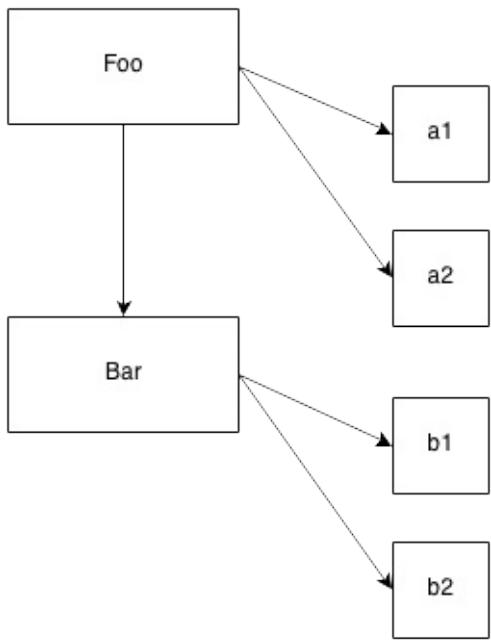
建筑物与蓝图间的关系是间接的。你可以检视蓝图来了解建筑物是如何构造的，但对于直接考察建筑物的每一部分，仅有蓝图是不够的。如果你想打开一扇门，你不得不走进建筑物自身——蓝图仅仅是为了用来表示门的位置而在纸上画的线条。

一个类就是一个蓝图。为了实际得到一个对象并与之互动，我们必须从类中建造（也就是实例化）某些东西。这种“构建”的最终结果是一个对象，典型地称为一个“实例”，我们可以按需要直接调用它的方法，访问它的公共数据属性。

这个对象是所有在类中被描述的特性的拷贝。

你不太指望走进一栋建筑之后发现，一份用于规划这栋建筑物的蓝图被裱起来挂在墙上，虽然蓝图可能在办公室的公共记录的文件中。相似地，你一般不会使用对象实例来直接访问和操作类，但是这至少对于判定对象实例来自于哪个类是可能的。

与考虑对象实例与它源自的类的任何间接关系相比，考虑类和对象实例的直接关系更有用。一个类通过拷贝操作被实例化为对象的形式。



如你所见，箭头由左向右，从上至下，这表示着概念上和物理上发生的拷贝操作。

## 构造器 (**Constructor**)

类的实例由类的一种特殊方法构建，这个方法的名称通常与类名相同，称为“**构造器** (**constructor**)”。这个方法的明确的工作，就是初始化实例所需的所有信息（状态）。

比如，考虑下面这个类的假想代码（语法是自创的）：

```
class CoolGuy {
    specialTrick = nothing

    CoolGuy( trick ) {
        specialTrick = trick
    }

    showOff() {
        output( "Here's my trick: ", specialTrick )
    }
}
```

为了制造一个 `CoolGuy` 实例，我们需要调用类的构造器：

```
Joe = new CoolGuy( "jumping rope" )

Joe.showOff() // Here's my trick: jumping rope
```

注意，`CoolGuy` 类有一个构造器 `CoolGuy()`，它实际上就是在我们说 `new CoolGuy(..)` 时调用的。我们从这个构造器拿回一个对象（类的一个实例），我们可以调用 `showOff()` 方法，来打印这个特定的 `CoolGuy` 的特殊才艺。

显然，跳绳使 `Joe` 看起来很酷。

类的构造器 属于 那个类，几乎总是和类同名。同时，构造器大多数情况下总是需要用 `new` 来调用，以便使语言的引擎知道你想要构建一个新的类的实例。

## 类继承

在面向类的语言中，你不仅可以定义一个可以初始化它自己的类，你还可以定义另外一个类继承 自第一个类。

这第二个类通常被称为“子类”，而第一个类被称为“父类”。这些名词明显地来自于亲子关系的比拟，虽然这种比拟有些扭曲，就像你马上要看到的。

当一个家长拥有一个和他有血缘关系的孩子时，家长的遗传性质会被拷贝到孩子身上。明显地，在大多数生物繁殖系统中，双亲都平等地贡献基因进行混合。但是为了这个比拟的目的，我们假设只有一个亲人。

一旦孩子出现，他或她就从亲人那里分离出来。这个孩子受其亲人的继承因素的严重影响，但是独一无二。如果这个孩子拥有红色的头发，这并不意味这他的亲人的头发曾经是红色，或者会自动变成红色。

以相似的方式，一旦一个子类被定义，它就分离且区别于父类。子类含有一份从父类那里得来的行为的初始拷贝，但它可以覆盖这些继承的行为，甚至是定义新行为。

重要的是，要记住我们在讨论父类和子类，而不是物理上的东西。这就是这个亲子比拟让人糊涂的地方，因为我们实际上应当说父类就是亲人的DNA，而子类就是孩子的DNA。我们不得不从两套DNA制造出（也就是初始化）人，用得到的物理上存在的人来与之进行谈话。

让我们把生物学上的亲子放在一边，通过一个稍稍不同的角度来看看继承：不同种类型的载具。这是用来理解继承的最经典（也是争议不断的）的比拟。

让我们重新审视本章前面的 `Vehicle` 和 `Car` 的讨论。考虑下面表达继承的类的假想代码：

```

class Vehicle {
    engines = 1

    ignition() {
        output( "Turning on my engine." )
    }

    drive() {
        ignition()
        output( "Steering and moving forward!" )
    }
}

class Car inherits Vehicle {
    wheels = 4

    drive() {
        inherited:drive()
        output( "Rolling on all ", wheels, " wheels!" )
    }
}

class SpeedBoat inherits Vehicle {
    engines = 2

    ignition() {
        output( "Turning on my ", engines, " engines." )
    }

    pilot() {
        inherited:drive()
        output( "Speeding through the water with ease!" )
    }
}

```

注意：为了简洁明了，这些类的构造器被省略了。

我们定义 `Vehicle` 类，假定它有一个引擎，有一个打开打火器的方法，和一个行驶的方法。但你永远也不会制造一个泛化的“载具”，所以在这里它只是一个概念的抽象。

然后我们定义了两种具体的载具：`car` 和 `SpeedBoat`。它们都继承 `Vehicle` 的泛化性质，但之后它们都对这些性质进行了合适的特化。一辆车有4个轮子，一艘快艇有两个引擎，意味着它需要在打火时需要特别注意要启动两个引擎。

## 多态（Polymorphism）

`car` 定义了自己的 `drive()` 方法，它覆盖了从 `Vehicle` 继承来的同名方法。但是，`Car` 的 `drive()` 方法调用了 `inherited:drive()`，这表示 `Car` 可以引用它继承的，覆盖之前的原版 `drive()`。`SpeedBoat` 的 `pilot()` 方法也引用了它继承的 `drive()` 拷贝。

这种技术称为“多态（polymorphism）”，或“虚拟多态（virtual polymorphism）”。对我们当前的情况更具体一些，我们称之为“相对多态（relative polymorphism）”。

多态这个话题比我们可以在这里谈到的内容要宽泛的多，但我们当前的“相对”意味着一个特殊层面：任何方法都可以引用位于继承层级上更高一层的其他方法（同名或不同名）。我们说“相对”，因为我们不绝对定义我们想访问继承的哪一层（也就是类），而实质上在说“向上一层”来相对地引用。

在许多语言中，在这个例子中使用 `inherited:` 的地方使用了 `super` 关键字，它的基于这样的想法：一个“超类（super class）”是当前类的父亲/祖先。

多态的另一个方面是，一个方法名可以在继承链的不同层面上有多种定义，而且在解析哪个方法在被调用时，这些定义可以适当地被自动选择。

在我们上面的例子中，我们看到这种行为发生了两次：`drive()` 在 `Vehicle` 和 `car` 中定义，而 `ignition()` 在 `Vehicle` 和 `SpeedBoat` 中定义。

注意：另一个传统面向类语言通过 `super` 给你的能力，是从子类的构造器中直接访问父类构造器。这很大程度上是对的，因为对真正的类来说，构造器属于这个类。然而在JS中，这是相反的——实际上认为“类”属于构造器（`Foo.prototype...` 类型引用）更恰当。因为在JS中，父子关系仅存在于它们各自的构造器的两个 `.prototype` 对象间，构造器本身不直接关联，而且没有简单的方法从一个中相对引用另一个（参见附录A，看看ES6中用 `super` “解决”此问题的 `class`）。

可以从 `ignition()` 中具体看出多态的一个有趣的含义。在 `pilot()` 内部，一个相对多态引用指向了（继承的）`Vehicle` 版本的 `drive()`。而这个 `drive()` 仅仅通过名称（不是相对引用）来引用 `ignition()` 方法。

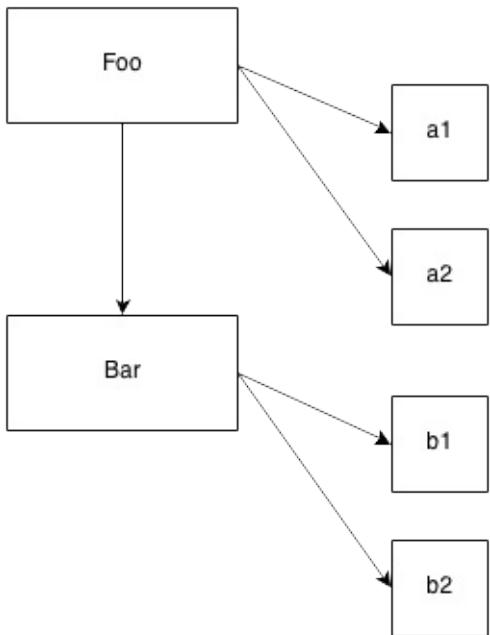
语言的引擎会使用哪一个版本的 `ignition()`？是 `Vehicle` 的还是 `SpeedBoat` 的？它会使用 `SpeedBoat` 版本的 `ignition()`。如果你能初始化 `Vehicle` 类自身，并且调用它的 `drive()`，那么语言引擎将会使用 `Vehicle` 的 `ignition()` 定义。

换句话说，`ignition()` 方法的定义，根据你引用的实例是哪个类（继承层级）而多态（改变）。

这看起来过于深入学术细节了。不过为了好好地与 JavaScript 的 `[[Prototype]]` 机制的类似行为进行对比，理解这些细节还是很重要的。

如果类是继承而来的，对这些类本身（不是由它们创建的对象）有一个方法可以相对地引用它们继承的对象，这个相对引用通常称为 `super`。

记得刚才这幅图：



注意对于实例化(`a1`，`a2`，`b1`，和`b2`)和继承(`Bar`)，箭头如何表示拷贝操作。

从概念上讲，看起来子类 `Bar` 可以使用相对多态引用（也就是 `super`）来访问它的父类 `Foo` 的行为。然而在现实中，子类不过是被给予了一份它从父类继承来的行为的拷贝而已。如果子类“覆盖”一个它继承的方法，原版的方法和覆盖版的方法实际上都是存在的，所以它们都是可以访问的。

不要让多态把你搞糊涂，使你认为子类是链接到父类上的。子类得到一份它需要从父类继承的东西的拷贝。类继承意味着拷贝。

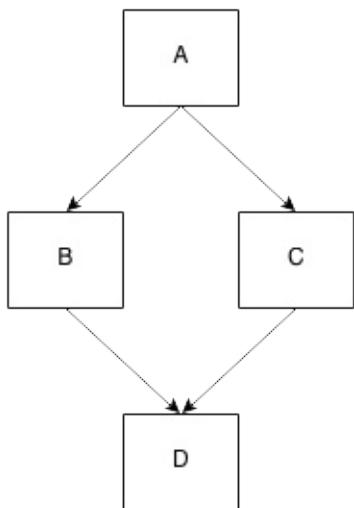
## 多重继承 (Multiple Inheritance)

能回想起我们早先提到的亲子和DNA吗？我们说过这个比拟有些奇怪，因为生物学上大多数后代来自于双亲。如果类可以继承自其他两个类，那么这个亲子比拟会更合适一些。

有些面向类的语言允许你指定一个以上的“父类”来进行“继承”。多重继承意味着每个父类的定义都被拷贝到子类中。

表面上看来，这是对面向类的一个强大的加成能力，给我们能力去将更多功能组合在一起。然而，这无疑会产生一些复杂的问题。如果两个父类都提供了名为 `drive()` 的方法，在子类中的 `drive()` 引用将会解析为哪个版本？你会总是不得不手动指明哪个父类的 `drive()` 是你想要的，从而失去一些多态继承的优雅之处吗？

还有另外一个所谓的“钻石问题”：子类“D”继承自两个父类（“B”和“C”），它们两个又继承自共通的父类“A”。如果“A”提供了方法 `drive()`，而“B”和“C”都覆盖（多态地）了这个方法，那么当“D”引用 `drive()` 时，它应当使用那个版本呢（`B:drive()` 还是 `C:drive()`）？



事情会比我们这样窥豹一斑能看到的复杂得多。我们在这里把它们记下来，以便于我们可以将它和JavaScript机制的工作方式比较。

JavaScript更简单：它不为“多重继承”提供原生机制。许多人认为这是好事，因为省去的复杂性要比“减少”的功能多得多。但是这并不能阻挡开发者们用各种方法来模拟它，我们接下来就看看。

## 混合（Mixin）

当你“继承”或是“实例化”时，JavaScript的对象机制不会自动地执行拷贝行为。很简单，在JavaScript中没有“类”可以拿来实例化，只有对象。而且对象也不会被拷贝到另一个对象中，而是被链接在一起（详见第五章）。

因为在其他语言中观察到的类的行为意味着拷贝，让我们来看看JS开发者如何在JavaScript中模拟这种缺失的类的拷贝行为：`mixins`（混合）。我们会看到两种“mixin”：明确的（**explicit**）和隐含的（**implicit**）。

### 明确的 Mixin（Explicit Mixins）

让我们再次回顾前面的 `Vehicle` 和 `Car` 的例子。因为 JavaScript 不会自动地将行为从 `Vehicle` 拷贝到 `Car`，我们可以建造一个工具来手动拷贝。这样的工具经常被许多包/框架称为 `extend(..)`，但为了说明的目的，我们在这里叫它 `mixin(..)`。

```
// 大幅简化的`mixin(..)`示例：
function mixin( sourceObj, targetObj ) {
    for ( var key in sourceObj ) {
        // 仅拷贝非既存内容
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }

    return targetObj;
}

var Vehicle = {
    engines: 1,

    ignition: function() {
        console.log( "Turning on my engine." );
    },

    drive: function() {
        this.ignition();
        console.log( "Steering and moving forward!" );
    }
};

var Car = mixin( Vehicle, {
    wheels: 4,

    drive: function() {
        Vehicle.drive.call( this );
        console.log( "Rolling on all " + this.wheels + " wheels!" );
    }
} );
```

注意：重要的细节：我们谈论的不再是类，因为在 JavaScript 中没有类。`Vehicle` 和 `Car` 分别只是我们实施拷贝的源和目标对象。

`Car` 现在拥有了份从 `Vehicle` 得到的属性和函数的拷贝。技术上讲，函数实际上没有被复制，而是指向函数的引用被复制了。所以，`Car` 现有一个称为 `ignition` 的属性，它是一个 `ignition()` 函数引用的拷贝；而且它还有一个称为 `engines` 的属性，持有从 `Vehicle` 拷贝来的值 `1`。

`Car` 已经有了 `drive` 属性（函数），所以这个属性引用没有被覆盖（参见上面 `mixin(..)` 的 `if` 语句）。

## 重温“多态 (Polymorphism) ”

我们来考察一下这个语句：`Vehicle.drive.call( this )`。我将之称为“显式假想多态 (*explicit pseudo-polymorphism*)”。回想我们前一段假想代码的这一行是我们称之为“相对多态 (*relative polymorphism*)”的 `inherited:drive()`。

JavaScript没有能力实现相对多态 (ES6之前, 见附录A)。所以, 因为 `car` 和 `Vehicle` 都有一个名为 `drive()` 的函数, 为了在它们之间区别调用, 我们必须使用绝对 (不是相对) 引用。我们明确地用名称指出 `Vehicle` 对象, 然后在它上面调用 `drive()` 函数。

但如果我们说 `Vehicle.drive()`, 那么这个函数调用的 `this` 绑定将会是 `Vehicle` 对象, 而不是 `car` 对象 (见第二章), 那不是我们想要的。所以, 我们使用 `.call( this )` (见第二章) 来保证 `drive()` 在 `Car` 对象的环境中被执行。

注意: 如果 `Car.drive()` 的函数名称标识符没有与 `Vehicle.drive()` 的重叠 (也就是“遮蔽”; 见第五章), 我们就不会有机会演示“方法多态 (method polymorphism)”。因为那样的话, 一个指向 `Vehicle.drive()` 的引用会被 `mixin(..)` 调用拷贝, 而我们可以使用 `this.drive()` 直接访问它。被选用的标识符重叠 遮蔽就是为什么我们不得不使用更复杂的 显式假想多态 (*explicit pseudo-polymorphism*) 的原因。

在拥有相对多态的面向类的语言中, `Car` 和 `Vehicle` 间的连接被建立一次, 就在类定义的顶端, 这里是维护这种关系的唯一场所。

但是由于JavaScript的特殊性, 显式假想多态 (因为遮蔽!) 在每一个你需要这种 (假想) 多态引用的函数中 建立了一种脆弱的手动/显式链接。这可能会显著地增加维护成本。而且, 虽然显式假想多态可以模拟“多重继承”的行为, 但这只会增加复杂性和代码脆弱性。

这种方法的结果通常是更加复杂, 更难读懂, 而且 更难维护的代码。应当尽可能地避免使用显式假想多态, 因为在大部分层面上它的代价要高于利益。

## 混合拷贝 (Mixing Copies)

回忆上面的 `mixin(..)` 工具:

```
// 大幅简化的`mixin()`示例:
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        // 仅拷贝不存在的属性
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }
    return targetObj;
}
```

现在，我们考察一下 `mixin(..)` 如何工作。它迭代 `sourceObj` (在我们的例子中是 `Vehicle`) 的所有属性，如果在 `targetObj` (在我们的例子中是 `car`) 中没有名称与之匹配的属性，它就进行拷贝。因为我们是在初始对象存在的情况下进行拷贝，所以我们要小心不要将目标属性覆盖掉。

如果在指明 `car` 的具体内容之前，我们先进行拷贝，那么我们就可以省略对 `targetObj` 检查，但是这样做有些笨拙且低效，所以通常不优先选用：

```
// 另一种mixin，对覆盖不太“安全”
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        targetObj[key] = sourceObj[key];
    }

    return targetObj;
}

var Vehicle = {
    // ...
};

// 首先，创建一个空对象
// 将Vehicle的内容拷贝进去
var Car = mixin( Vehicle, {} );

// 现在拷贝Car的具体内容
mixin( {
    wheels: 4,

    drive: function() {
        // ...
    }
}, Car );
```

不论哪种方法，我们都显式地将 `Vehicle` 中的非重叠内容拷贝到 `car` 中。“`mixin`”这个名称来自于解释这个任务的另一种方法：`car` 混入 `Vehicle` 的内容，就像你吧巧克力碎片混入你最喜欢的曲奇饼面团。

这个拷贝操作的结果，是 `car` 将会独立于 `Vehicle` 运行。如果你在 `car` 上添加属性，它不会影响到 `Vehicle`，反之亦然。

注意：这里有几个小细节被忽略了。仍然有一些微妙的方法使两个对象在拷贝完成后还能互相“影响”对方，比如他们共享一个共通对象（比如数组）的引用。

由于两个对象还共享它们的共通函数的引用，这意味着即使手动将函数从一个对象拷贝（也就是混入）到另一个对象中，也不能实际上模拟发生在面向类的语言中的从类到实例的真正的复制。

JavaScript函数不能真正意义上地被复制（以标准，可靠的方式），所以你最终得到的是同一个共享的函数对象（函数是对象；见第三章）的被复制的引用。举例来说，如果你在一个共享的函数对象（比如 `ignition()`）上添加属性来修改它，`Vehicle` 和 `car` 都会通过这个共享的引用而受影响。

在JavaScript中明确的**mixin**是一种不错的机制。但是它们显得言过其实。和将一个属性定义两次相比，将属性从一个对象拷贝到另一个对象并不会产生多少实际的好处。这对我们刚才提到的给函数对象引用增加的微妙变化来说，显得尤为正确。

如果你明确地将两个或更多对象混入你的目标对象，你可以某种程度上模拟“多重继承”的行为，但是在将方法或属性从多于一个源对象那里拷贝过来时，没有直接的办法可以解决名称的冲突。有些开发者/包使用“`late binding`（延迟绑定）”和其他诡异的替代方法来解决问题，但从根本上讲，这些“技巧”通常得不偿失（而且低效！）。

要小心的是，仅在明确的**mixin**能够实际提高代码可读性时使用它，而如果你发现它使代码变得更很难追溯，或在对象间建立了不必要或笨重的依赖性时，要避免使用这种模式。

如果正确使用**mixin**使你的问题变得比以前困难，那么你可能应当停止使用**mixin**。实际上，如果你不得不使用复杂的包/工具来处理这些细节，这可能标志着你正走在更困难，也许没必要的道路上。在第六章中，我们将试着提取一种更简单的方法来实现我们期望的结果，同时免去这些周折。

## 寄生继承（Parasitic Inheritance）

明确的**mixin**模式的一个变种，在某种意义上是明确的而在某种意义上是隐含的，称为“寄生继承（`Parasitic Inheritance`）”，它主要是由Douglas Crockford推广的。

这是它如何工作：

```
// “传统的JS类” `Vehicle`  
function Vehicle() {  
    this.engines = 1;  
}  
Vehicle.prototype.ignition = function() {  
    console.log( "Turning on my engine." );  
};  
Vehicle.prototype.drive = function() {  
    this.ignition();  
    console.log( "Steering and moving forward!" );  
};  
  
// “寄生类” `Car`  
function Car() {  
    // 首先，`car`是一个`Vehicle`  
    var car = new Vehicle();  
  
    // 现在，我们修改`car`使它特化  
    car.wheels = 4;  
  
    // 保存一个`Vehicle::drive()`的引用  
    var vehDrive = car.drive;  
  
    // 覆盖 `Vehicle::drive()`  
    car.drive = function() {  
        vehDrive.call( this );  
        console.log( "Rolling on all " + this.wheels + " wheels!" );  
    };  
  
    return car;  
}  
  
var myCar = new Car();  
  
myCar.drive();  
// Turning on my engine.  
// Steering and moving forward!  
// Rolling on all 4 wheels!
```

如你所见，我们一开始从“父类”（对象）`Vehicle`制造了一个定义的拷贝，之后将我们的“子类”（对象）定义混入其中（按照需要保留父类的引用），最后将组合好的对象`car`作为子类实例传递出去。

注意：当我们调用`new Car()`时，一个新对象被创建并被`Car`的`this`所引用（见第二章）。但是由于我们没有使用这个对象，而是返回我们自己的`car`对象，所以这个初始化创建的对象就被丢弃了。所以，`Car()`可以不用`new`关键字调用，就可以实现和上面代码相同的功能，而且还可以节省对象的创建和回收。

## 隐含的 Mixin (Implicit Mixins)

隐含的 mixin 和前面解释的 显式假想多态 是紧密相关的。所以它们需要注意相同的事情。

考虑这段代码：

```

var Something = {
  cool: function() {
    this.greeting = "Hello World";
    this.count = this.count ? this.count + 1 : 1;
  }
};

Something.cool();
Something.greeting; // "Hello World"
Something.count; // 1

var Another = {
  cool: function() {
    // 隐式地将 `Something` 混入 `Another`
    Something.cool.call( this );
  }
};

Another.cool();
Another.greeting; // "Hello World"
Another.count; // 1 (不会和 `Something` 共享状态)

```

`Something.cool.call( this )` 既可以在“构造器”调用中使用（最常见的情况），也可以在方法调用中使用（如这里所示），我们实质上“借用”了 `Something.cool()` 函数并在 `Another` 环境下，而非 `Something` 环境下调用它（通过 `this` 绑定，见第二章）。结果是，`Something.cool()` 中进行的赋值被实施到了 `Another` 对象而非 `Something` 对象。

那么，这就是说我们将 `Something` 的行为“混入”了 `Another`。

虽然这种技术看起来有效利用了 `this` 再绑定的功能，也就是生硬地调用 `Something.cool.call( this )`，但是这种调用不能被作为相对（也更灵活的）引用，所以你应当提高警惕。一般来说，尽量避免使用这种结构来保持代码干净而且容易维护。

## 复习

类是一种设计模式。许多语言提供语法来启用自然而然的面向类的软件设计。JS也有相似的语法，但是它的行为和你在其他语言中熟悉的工作原理有很大的不同。

类意味着拷贝。

当一个传统的类被实例化时，就发生了类的行为向实例中拷贝。当类被继承时，也发生父类的行为向子类的拷贝。

多态（在继承链的不同层级上拥有同名的不同函数）也许看起来意味着一个从子类回到父类的相对引用链接，但是它仍然只是拷贝行的结果。

JavaScript 不会自动地（像类那样）在对象间创建拷贝。

mixin模式常用于在某种程度上模拟类的拷贝行为，但是这通常导致像显式假想多态那样(`OtherObj.methodName.call(this, ...)`)难看而且脆弱的语法，这样的语法又常导致更难懂和更难维护的代码。

明确的mixin和类 拷贝 又不完全相同，因为对象（和函数！）仅仅是共享的引用被复制，不是对象/函数自身被复制。不注意这样的微小之处通常是各种陷阱的根源。

一般来讲，在JS中模拟类通常会比解决当前真正的问题埋下更多的坑。

# 你不懂JS: **this** 与对象原型

## 第五章: 原型 (Prototype)

在第三，四章中，我们几次提到了 `[[Prototype]]` 链，但我们没有讨论它到底是什么。现在我们就详细讲解一下原型 (`prototype`)。

注意：所有模拟类拷贝行为的企图，也就是我们在前面第四章描述的内容，称为各种种类的“mixin”，和我们要在本章中讲解的 `[[Prototype]]` 链机制完全不同。

### `[[Prototype]]`

JavaScript中的对象有一个内部属性，在语言规范中称为 `[[Prototype]]`，它只是一个其他对象的引用。几乎所有的对象在被创建时，它的这个属性都被赋予了一个非 `null` 值。

注意：我们马上就会看到，一个对象拥有一个空的 `[[Prototype]]` 链接是可能的，虽然这有些不寻常。

考虑下面的代码：

```
var myObject = {
  a: 2
};

myObject.a; // 2
```

`[[Prototype]]` 引用有什么用？在第三章中，我们讲解了 `[[Get]]` 操作，它会在你引用一个对象上的属性时被调用，比如 `myObject.a`。对于默认的 `[[Get]]` 操作来说，第一步就是检查对象本身是否拥有一个 `a` 属性，如果有，就使用它。

注意：ES6的代理 (Proxy) 超出了我们要在本书内讨论的范围（将在本系列的后续书中涵盖！），但是如果加入 `Proxy`，我们在这里讨论的关于普通 `[[Get]]` 和 `[[Put]]` 的行为都是不被采用的。

但是如果 `myObject` 上不存在 `a` 属性时，我们就将注意力转向对象的 `[[Prototype]]` 链。

如果默认的 `[[Get]]` 操作不能直接在对象上找到被请求的属性，那么会沿着对象的 `[[Prototype]]` 链继续处理。

```

var anotherObject = {
  a: 2
};

// 创建一个链接到`anotherObject`的对象
var myObject = Object.create( anotherObject );

myObject.a; // 2

```

注意：我们马上就会解释 `Object.create(..)` 是做什么，如何做的。眼下先假设，它创建了一个对象，这个对象带有一个链到指定的对象的 `[[Prototype]]` 链接，这个链接就是我们要讲解的。

那么，我们现在让 `myObject``[[Prototype]]` 链到了 `anotherObject`。虽然很明显 `myObject.a` 实际上不存在，但是无论如何属性访问成功了（在 `anotherObject` 中找到了），而且确实找到了值 `2`。

但是，如果在 `anotherObject` 上也没有找到 `a`，而且如果它的 `[[Prototype]]` 链不为空，就沿着它继续查找。

这个处理持续进行，直到找到名称匹配的属性，或者 `[[Prototype]]` 链终结。如果在链条的末尾都没有找到匹配的属性，那么 `[[Get]]` 操作的返回结果为 `undefined`。

和这种 `[[Prototype]]` 链查询处理相似，如果你使用 `for..in` 循环迭代一个对象，所有在它的链条上可以到达的（并且是 `enumerable` ——见第三章）属性都会被枚举。如果你使用 `in` 操作符来测试一个属性在一个对象上的存在性，`in` 将会检查对象的整个链条（不管可枚举性）。

```

var anotherObject = {
  a: 2
};

// 创建一个链接到`anotherObject`的对象
var myObject = Object.create( anotherObject );

for (var k in myObject) {
  console.log("found: " + k);
}

// 找到: a

("a" in myObject); // true

```

所以，当你以各种方式进行属性查询时，`[[Prototype]]` 链就会一个链接一个链接地被查询。一旦找到属性或者链条终结，这种查询会就会停止。

## Object.prototype

但是 `[[Prototype]]` 链到底在哪儿“终结”？

每个普通的 `[[Prototype]]` 链的最顶端，是内建的 `Object.prototype`。这个对象包含各种在整个JS中被使用的共通工具，因为JavaScript中所有普通（内建，而非被宿主环境扩展的）的对象都“衍生自”（也就是，使它们的 `[[Prototype]]` 顶端为） `Object.prototype` 对象。

你会在这里发现一些你可能很熟悉的工具，比如 `.toString()` 和 `.valueOf()`。在第三章中，我们介绍了另一个：`.hasOwnProperty(..)`。还有另外一个你可能不太熟悉，但我们将在这第一章里讨论的 `Object.prototype` 上的函数是 `.isPrototypeOf(..)`。

## 设置与遮蔽属性

回到第三章，我们提到过在对象上设置属性要比仅仅在对象上添加新属性或改变既存属性的值更加微妙。现在我们将更完整地重温这个话题。

```
myObject.foo = "bar";
```

如果 `myObject` 对象已直接经拥有了普通的名为 `foo` 的数据访问器属性，那么这个赋值就和改变既存属性的值一样简单。

如果 `foo` 还没有直接存在于 `myObject`，`[[Prototype]]` 就会被遍历，就像 `[[Get]]` 操作那样。如果在链条的任何地方都没有找到 `foo`，那么就会像我们期望的那样，属性 `foo` 就以指定的值被直接添加到 `myObject` 上。

然而，如果 `foo` 已经存在于链条更高层的某处，`myObject.foo = "bar"` 赋值就可能会发生微妙的（也许令人诧异的）行为。我们一会儿就详细讲解。

如果属性名 `foo` 同时存在于 `myObject` 本身和从 `myObject` 开始的 `[[Prototype]]` 链的更高层，这样的情况称为遮蔽。直接存在于 `myObject` 上的 `foo` 属性会遮蔽任何出现在链条高层的 `foo` 属性，因为 `myObject.foo` 查询总是在寻找链条最底层的 `foo` 属性。

正如我们被暗示的那样，在 `myObject` 上的 `foo` 遮蔽没有看起来那么简单。我们现在来考察 `myObject.foo = "bar"` 赋值的三种场景，当 `foo` 不直接存在于 `myObject`，但存在于 `myObject` 的 `[[Prototype]]` 链的更高层：

1. 如果一个普通的名为 `foo` 的数据访问属性在 `[[Prototype]]` 链的高层某处被找到，而且没有被标记为只读（`writable:false`），那么一个名为 `foo` 的新属性就直接添加到 `myObject` 上，形成一个遮蔽属性。
2. 如果一个 `foo` 在 `[[Prototype]]` 链的高层某处被找到，但是它被标记为只读（`writable:false`），那么设置既存属性和在 `myObject` 上创建遮蔽属性都是不允许的。如果代码运行在 `strict mode` 下，一个错误会被抛出。否则，这个设置属性值的操作会被无声地忽略。不论怎样，没有发生遮蔽。
3. 如果一个 `foo` 在 `[[Prototype]]` 链的高层某处被找到，而且它是一个setter（见第三

章) , 那么这个 `setter` 总是被调用。没有 `foo` 会被添加到 (也就是遮蔽在) `myObject` 上, 这个 `foo` `setter` 也不会被重定义。

大多数开发者认为, 如果一个属性已经存在于 `[[Prototype]]` 链的高层, 那么对它的赋值 (`[[Put]]`) 将总是造成遮蔽。但如你所见, 这仅在刚才描述的三中场景中的一种 (第一种) 中是对的。

如果你想在第二和第三种情况中遮蔽 `foo` , 那你就不能使用 `= 赋值`, 而必须使用 `Object.defineProperty(..)` (见第三章) 将 `foo` 添加到 `myObject` 。

注意: 第二种情况可能是三种情况中最让人诧异的了。只读属性的存在会阻止同名属性在 `[[Prototype]]` 链的低层被创建 (遮蔽)。这个限制的主要原因是增强了类继承属性的幻觉。如果你想象位于链条高层的 `foo` 被继承 (拷贝) 至 `myObject` , 那么在 `myObject` 上强制 `foo` 属性不可写就有道理。但如果你将幻觉和现实分开, 而且认识到实际上没有这样的继承拷贝发生 (见第四, 五章) , 那么仅因为某些其他的对象上拥有不可写的 `foo` , 而导致 `myObject` 不能拥有 `foo` 属性就有些不自然。而且更奇怪的是, 这个限制仅限于 `= 赋值`, 当使用 `Object.defineProperty(..)` 时不被强制。

如果你需要在方法间进行委托, 方法的遮蔽会导致难看的 显式假想多态 (见第四章) 。一般来说, 遮蔽与它带来的好处相比太过复杂和微妙了, 所以你应当尽量避免它。第六章介绍另一种设计模式, 它提倡干净而且不鼓励遮蔽。

遮蔽甚至会以微妙的方式隐含地发生, 所以要想避免它必须小心。考虑这段代码:

```
var anotherObject = {
  a: 2
};

var myObject = Object.create( anotherObject );

anotherObject.a; // 2
myObject.a; // 2

anotherObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "a" ); // false

myObject.a++; // 噢, 隐式遮蔽!

anotherObject.a; // 2
myObject.a; // 3

myObject.hasOwnProperty( "a" ); // true
```

虽然看起来 `myObject.a++` 应当 (通过委托) 查询并原地递增 `anotherObject.a` 属性, 但是 `++` 操作符相当于 `myObject.a = myObject.a + 1` 。结果就是在 `[[Prototype]]` 上进行 `a` 的 `[[Get]]` 查询, 从 `anotherObject.a` 得到当前的值 `2` , 将这个值递增`1`, 然后将

值 3 用 `[[Put]]` 赋值到 `myObject` 上的新遮蔽属性 `a` 上。噢！

修改你的委托属性时要非常小心。如果你想递增 `anotherObject.a`，那么唯一正确的方法是 `anotherObject.a++`。

## “类”

现在你可能会想知道：“为什么一个对象需要链到另一个对象？”真正的好处是什么？这是一个很恰当的问题，但在我们能够完全理解和体味它是什么和如何有用之前，我们必须首先理解 `[[Prototype]]` 不是什么。

正如我们在第四章讲解的，在JavaScript中，对于对象来说没有抽象模式/蓝图，即没有面向类的语言中那样的称为类的东西。JavaScript 只有 对象。

实际上，在所有语言中，JavaScript 几乎是独一无二的，也许是唯一的可以被称为“面向对象”的语言，因为可以根本没有类而直接创建对象的语言很少，而JavaScript就是其中之一。

在JavaScript中，类不能（因为根本不存在）描述对象可以做什么。对象直接定义它自己的行为。这里 仅有 对象。

## “类”函数

在JavaScript中有一种奇异的行为被无耻地滥用了许多年来 山寨 成某些 看起来 像“类”的东西。我们来仔细看看这种方式。

“某种程度的类”这种奇特的行为取决于函数的一个奇怪的性质：所有的函数默认都会得到一个公有的，不可枚举的属性，称为 `prototype`，它可以指向任意的对象。

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

Foo.prototype; // { }
```

这个对象经常被称为“`Foo`的原型”，因为我们通过一个不幸地被命名为 `Foo.prototype` 的属性引用来访问它。然而，我们马上会看到，这个术语命中注定地将我们搞糊涂。为了取代它，我将它称为“以前被认为是`Foo`的原型的对象”。只是开个玩笑。“一个被随意标记为‘`Foo`点儿原型’的对象”，怎么样？

不管我们怎么称呼它，这个对象到底是什么？

解释它的最直接的方法是，每个由调用 `new Foo()`（见第二章）而创建的对象将最终（有些随意地）被 `[[Prototype]]` 链接到这个“`Foo`点儿原型”对象。

让我们描绘一下：

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

var a = new Foo();

Object.getPrototypeOf(a) === Foo.prototype; // true
```

当通过调用 `new Foo()` 创建 `a` 时，会发生的事情之一（见第二章了解所有四个步骤）是，`a` 得到一个内部 `[[Prototype]]` 链接，此链接链到 `Foo.prototype` 所指向的对象。

停一会儿来思考一下这句话的含义。

在面向类的语言中，可以制造一个类的多个拷贝（即“实例”），就像从模具中冲压出某些东西一样。我们在第四章中看到，这是因为初始化（或者继承）类的处理意味着，“将行为计划从这个类拷贝到物理对象中”，对于每个新实例这都会发生。

但是在JavaScript中，没有这样的拷贝处理发生。你不会创建类的多个实例。你可以创建多个对象，它们的 `[[Prototype]]` 连接至一个共通对象。但默认地，没有拷贝发生，如此这些对象彼此间最终不会完全分离和切断关系，而是链接在一起。

`new Foo()` 得到一个新对象（我们叫他 `a`），这个新对象 `a` 内部地被 `[[Prototype]]` 链接至 `Foo.prototype` 对象。

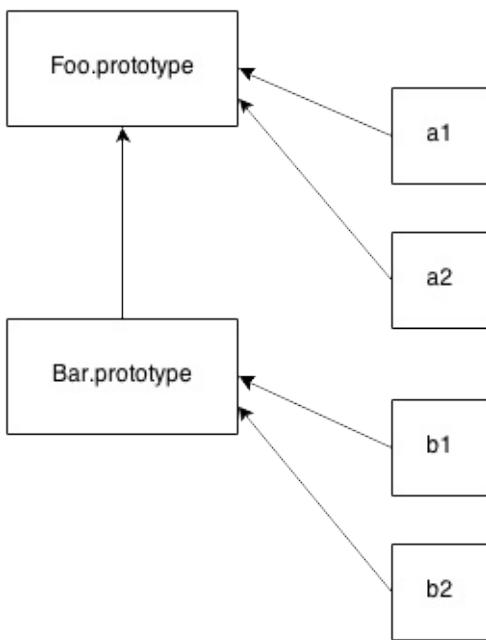
结果我们得到两个对象，彼此链接。如是而已。我们没有初始化一个对象。当然我们也没有做任何从一个“类”到一个实体对象拷贝。我们只是让两个对象互相链接在一起。

事实上，这个使大多数JS开发者无法理解的秘密，是因为 `new Foo()` 函数调用实际上几乎和建立链接的处理没有任何直接关系。它是某种偶然的副作用。`new Foo()` 是一个间接的，迂回的方法来得到我们想要的：一个被链接到另一个对象的对象。

我们能用更直接的方法得到我们想要的吗？可以！这位英雄就是 `Object.create(..)`。我们过会儿就谈到它。

## 名称的意义何在？

在JavaScript中，我们不从一个对象（“类”）向另一个对象（“实例”）拷贝。我们在对象之间制造链接。对于 `[[Prototype]]` 机制，视觉上，箭头的移动方向是从右至左，由下至上。



这种机制常被称为“原型继承（prototypal inheritance）”（我们很快就用代码说明），它经常被说成是动态语言版的“类继承”。这种说法试图建立在面向类世界中对“继承”含义的共识上。但是弄拧（意思是：抹平）了被理解语义，来适应动态脚本。

先入为主，“继承”这个词有很强烈的含义（见第四章）。仅仅在它前面加入“原型”来区别于JavaScript中实际上几乎相反的行为，使真相在泥泞般的困惑中沉睡了近二十年。

我想说，将“原型”贴在“继承”之前很大程度上搞反了它的实际意义，就像一只手拿着一个桔子，另一手拿着一个苹果，而坚持说苹果是一个“红色的桔子”。无论我在它前面放什么令人困惑的标签，那都不会改变一个水果是苹果而另一个是桔子的事实。

更好的方法是直白地将苹果称为苹果——使用最准确和最直接的术语。这样能更容易地理解它们的相似之处和许多不同之处，因为我们都对“苹果”的意义有一个简单的，共享的理解。

由于用语的模糊和歧义，我相信，对于解释JavaScript机制真正如何工作来说，“原型继承”这个标签（以及试图错误地应用所有面向类的术语，比如“类”，“构造器”，“实例”，“多态”等）本身带来的危害比好处多。

“继承”意味着拷贝操作，而JavaScript不拷贝对象属性（原生上，默认地）。相反，JS在两个对象间建立链接，一个对象实质上可以将对属性/函数的访问委托到另一个对象上。对于描述JavaScript对象链接机制来说，“委托”是一个准确得多的术语。

另一个有时被扔到JavaScript旁边的术语是“差分继承”。它的想法是，我们可以用一个对象与一个更泛化的对象的不同来描述一个它的行为。比如，你要解释汽车是一种载具，与其重新描述组成一个一般载具的所有特点，不如只说它有4个轮子。

如果你试着想象，在JS中任何给定的对象都是通过委托可用的所有行为的总和，而且在你思维中你扁平化所有的行为到一个有形的东西中，那么你就可以（八九不离十地）看到“差分继承”是如何自圆其说的。

但正如“原型继承”，“差分继承”假意使你的思维模型比在语言中物理发生的事情更重要。它忽视了这样一个事实：对象 `B` 实际上不是一个差异结构，而是由一些定义好的特定性质，与一些没有任何定义的“漏洞”组成的。正是通过这些“漏洞”（缺少定义），委托可以接管并且动态地用委托行为“填补”它们。

对象不是像“差分继承”的思维模型所暗示的那样，原生默认地，通过拷贝扁平化到一个单独的差异对象中。如此，对于描述JavaScript的 `[[Prototype]]` 机制如何工作来说，“差分继承”就不是自然合理。

你可以选择偏向“差分继承”这个术语和思维模型，这是个人口味的问题，但是不能否认这个事实：它仅仅符合你思维中的主观过程，不是引擎的物理行为。

## "构造器" (Constructors)

让我们回到早先的代码：

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

var a = new Foo();
```

到底是什么导致我们认为 `Foo` 是一个“类”？

其一，我们看到了 `new` 关键字的使用，就像面向类语言中人们构建类的对象那样。另外，它看起来我们事实上执行了一个类的构造器方法，因为 `Foo()` 实际上是个被调用的方法，就像当你初始化一个真实的类时这个类的构造器被调用的那样。

为了使“构造器”的语义更使人糊涂，被随意贴上标签的 `Foo.prototype` 对象还有另外一招。考虑这段代码：

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

Foo.prototype.constructor === Foo; // true

var a = new Foo();
a.constructor === Foo; // true
```

`Foo.prototype` 对象默认地（就在代码段中第一行中声明的地方！）得到一个公有的，称为 `.constructor` 的不可枚举（见第三章）属性，而且这个属性回头指向这个对象关联的函数（这里是 `Foo`）。另外，我们看到被“构造器”调用 `new Foo()` 创建的对象 `a` 看起来也拥有一个称为 `.constructor` 的属性，也相似地指向“创建它的函数”。

注意：这实际上不是真的。`a` 上没有 `.constructor` 属性，而 `a.constructor` 确实解析成了 `Foo` 函数，“constructor”并不像它看起来的那样实际意味着“被XX创建”。我们很快就会解释这个奇怪的地方。

哦，是的，另外……根据JavaScript世界中的惯例，“类”都以大写字母开头的单词命名，所以使用 `Foo` 而不是 `foo` 强烈地意味着我们打算让它成为一个“类”。这对你来说太明显了，对吧！？

注意：这个惯例是如此强大，以至于如果你在一个小写字母名称的方法上使用 `new` 调用，或并没有在一个大写字母开头的函数上使用 `new`，许多JS语法检查器将会报告错误。这是因为我们如此努力地想要在JavaScript中将（假的）“面向类”搞对，所以我们建立了这些语法规则来确保我们使用了大写字母，即便对JS引擎来讲，大写字母根本没有任何意义。

## 构造器还是调用？

上面的代码的段中，我们试图认为 `Foo` 是一个“构造器”，是因为我们用 `new` 调用它，而且我们观察到它“构建”了一个对象。

在现实中，`Foo` 不会比你的程序中的其他任何函数“更像构造器”。函数自身不是构造器。但是，当你在普通函数调用前面放一个 `new` 关键字时，这就将函数调用变成了“构造器调用”。事实上，`new` 在某种意义上劫持了普通函数并将它以另一种方式调用：构建一个对象，外加这个函数要做的其他任何事。

举个例子：

```
function NothingSpecial() {
  console.log( "Don't mind me!" );
}

var a = new NothingSpecial();
// "Don't mind me!"

a; // {}
```

`NothingSpecial` 仅仅是一个普通的函数，但当用 `new` 调用时，几乎是一种副作用，它会构建一个对象，并被我们赋值到 `a`。这个调用是一个构造器调用，但是 `NothingSpecial` 本身并不是一个构造器。

换句话说，在JavaScript中，更合适的说法是，“构造器”是在前面用 `new` 关键字调用的任何函数。

函数不是构造器，但是当且仅当 `new` 被使用时，函数调用是一个“构造器调用”。

## 机制

仅仅是这些原因使得JavaScript中关于“类”的讨论变得命运多舛吗？

不全是。JS开发者们努力地尽可能的模拟面向类：

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

Foo.prototype.myName = function() {
    return this.name;
};

var a = new Foo( "a" );
var b = new Foo( "b" );

a.myName(); // "a"
b.myName(); // "b"
```

这段代码展示了另外两种“面向类”的花招：

1. `this.name = name`：在每个对象（分别在 `a` 和 `b` 上；参照第二章关于 `this` 绑定的内容）上添加了 `.name` 属性，和类的实例包装数据值很相似。
2. `Foo.prototype.myName = ...`：这也许是更有趣的技术，它在 `Foo.prototype` 对象上添加了一个属性（函数）。现在，也许让人惊奇，`a.myName()` 可以工作。但是是如何工作的？

在上面的代码段中，有很强的倾向认为当 `a` 和 `b` 被创建时，`Foo.prototype` 上的属性/函数被拷贝到了 `a` 与 `b` 俩个对象上。但是，这没有发生。

在本章开头，我们解释了 `[[Prototype]]` 链，和它作为默认的 `[[Get]]` 算法的一部分，如何在不能直接在对象上找到属性引用时提供后备的查询步骤。

于是，得益于他们被创建的方式，`a` 和 `b` 都最终拥有一个内部的 `[[Prototype]]` 链接链到 `Foo.prototype`。当无法分别在 `a` 和 `b` 中找到 `myName` 时，就会在 `Foo.prototype` 上找到（通过委托，见第六章）。

## 复活“构造器”

回想我们刚才对 `.constructor` 属性的讨论，怎么看起来 `a.constructor === Foo` 为 `true` 意味着 `a` 上实际拥有一个 `.constructor` 属性，指向 `Foo`？不对。

这只不过是一种不幸的混淆。实际上，`.constructor` 引用也委托到了 `Foo.prototype`，它恰好有一个指向 `Foo` 的默认属性。

这看起来方便得可怕，一个被 `Foo` 构建的对象可以访问指向 `Foo` 的 `.constructor` 属性。但这只不过是安全感上的错觉。它是一个欢乐的巧合，几乎是误打误撞，通过默认的 `[[Prototype]]` 委托 `a.constructor` 恰好指向 `Foo`。实际上 `.constructor` 意味着“被XX构建”这种注定失败的臆测会以几种方式来咬到你。

第一，在 `Foo.prototype` 上的 `.constructor` 属性仅当 `Foo` 函数被声明时才出现在对象上。如果你创建一个新对象，并用它替换函数默认的 `.prototype` 对象引用，这个新对象上将不会魔法般地得到 `.constructor`。

考虑这段代码：

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

Foo.prototype = { /* .. */ }; // 创建一个新的prototype对象

var a1 = new Foo();
a1.constructor === Foo; // false!
a1.constructor === Object; // true!
```

`Object(..)` 没有“构建” `a1`，是吧？看起来确实是 `Foo()` “构建了”它。许多开发者认为 `Foo()` 在执行构建，但当你认为“构造器”意味着“被XX构建”时，一切就都崩塌了，因为如果那样的话，`a1.construcor` 应当是 `Foo`，但它不是！

发生了什么？`a1` 没有 `.constructor` 属性，所以它沿者 `[[Prototype]]` 链向上委托到了 `Foo.prototype`。但是这个对象也没有 `.constructor`（默认的 `Foo.prototype` 对象就会有！），所以它继续委托，这次轮到了 `Object.prototype`，委托链的最顶端。那个对象上确实拥有 `.constructor`，它指向内建的 `Object(..)` 函数。

误解，消除。

当然，你可以把 `.constructor` 加回到 `Foo.prototype` 对象上，但是要做一些手动工作，特别是如果你想要它与原生的行为吻合，并不可枚举时（见第三章）。

举例来说：

```

function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // 创建一个新的prototype对象

// 需要正确地“修复”丢失的`constructor`。
// 新对象上的属性以`Foo.prototype`的形式提供。
// `defineProperty(..)`的内容见第三章。
Object.defineProperty( Foo.prototype, "constructor" , {
  enumerable: false,
  writable: true,
  configurable: true,
  value: Foo // 使`constructor`指向`Foo`
} );

```

要修复 `.constructor` 要花不少功夫。而且，我们做的一切是为了延续“构造器”意味着“被XX构建”的误解。这是一种昂贵的假象。

事实上，一个对象上的 `.constructor` 默认地随意指向一个函数，而这个函数反过来拥有一个指向被这个对象称为 `.prototype` 的对象。“构造器”和“原型”这两个词仅有松散的默认含义，可能是真的也可能不是真的。最佳方案是提醒你自己，“构造器不是意味着被XX构建”。

`.constructor` 不是一个魔法般不可变的属性。它是不可枚举的（见上面的代码段），但是它的值是可写的（可以改变），而且，你可以在 `[[Prototype]]` 链上的任何对象上添加或覆盖（有意或无意地）名为 `constructor` 的属性，用你感觉合适的任何值。

根据 `[[Get]]` 算法如何遍历 `[[Prototype]]` 链，在任何地方找到的一个 `.constructor` 属性引用解析的结果可能与你期望的十分不同。

看到它的实际意义有多随便了吗？

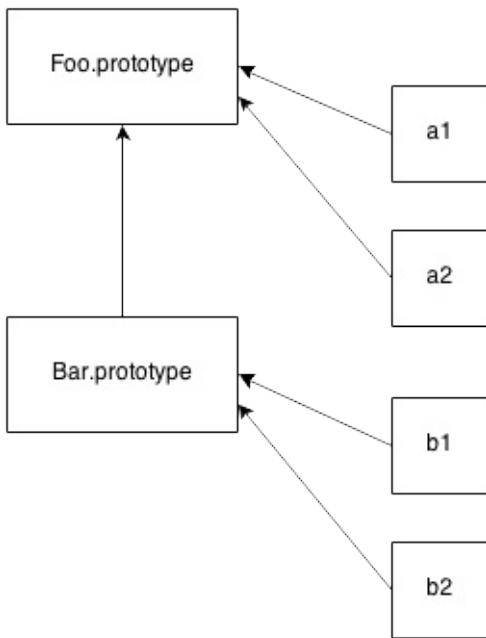
结果？某些像 `a1.constructor` 这样随意的对象属性引用实际上不能被认为是默认的函数引用。还有，我们马上就会看到，通过一个简单的省略，`a1.constructor` 可以最终指向某些令人诧异，没道理的地方。

`a1.constructor` 是极其不可靠的，在你的代码中不应依赖的不安全引用。一般来说，这样的引用应当尽量避免。

## “（原型）继承”

我们已经看到了一些近似的“类”机制骇进JavaScript程序。但是如果我们没有一种近似的“继承”，JavaScript的“类”将会更空洞。

实际上，我们已经看到了一个常被称为“原型继承”的机制如何工作：`a` 可以“继承自”`Foo.prototype`，并因此可以访问 `myName()` 函数。但是我们传统的想法认为“继承”是两个“类”间的关系，而非“类”与“实例”的关系。



回想之前这幅图，它不仅展示了从对象（也就是“实例”）`a1` 到对象 `Foo.prototype` 的委托，而且从 `Bar.prototype` 到 `Foo.prototype`，这酷似类继承的亲自概念。酷似，除了方向，箭头表示的是委托链接，而不是拷贝操作。

这里是一段典型的创建这样的链接的“原型风格”代码：

```

function Foo(name) {
    this.name = name;
}

Foo.prototype.myName = function() {
    return this.name;
};

function Bar(name, label) {
    Foo.call( this, name );
    this.label = label;
}

// 这里，我们创建一个新的`Bar.prototype`链接链到`Foo.prototype`、
Bar.prototype = Object.create( Foo.prototype );

// 注意！现在`Bar.prototype.constructor`不存在了，
// 如果你有依赖这个属性的习惯的话，可以被手动“修复”。

Bar.prototype.myLabel = function() {
    return this.label;
};

var a = new Bar( "a", "obj a" );

a.myName(); // "a"
a.myLabel(); // "obj a"
  
```

注意：要想知道为什么上面代码中的 `this` 指向 `a`，参见第二章。

重要的部分是 `Bar.prototype = Object.create( Foo.prototype )`。`Object.create(..)` 凭空创建了一个“新”对象，并将这个新对象内部的 `[[Prototype]]` 链接到你指定的对象上（在这里是 `Foo.prototype`）。

换句话说，这一行的意思是：“做一个新的链接到‘`Foo`点儿`prototype`’的‘`Bar`点儿`prototype`’对象”。

当 `function Bar() { .. }` 被声明时，就像其他函数一样，拥有一个链到默认对象的 `.prototype` 链接。但是那个对象没有链到我们希望的 `Foo.prototype`。所以，我们创建了一个新对象，链到我们希望的地方，并将原来的错误链接的对象扔掉。

注意：这里一个常见的误解/困惑是，下面两种方法也能工作，但是他们不会如你期望的那样工作：

```
// 不会如你期望的那样工作!
Bar.prototype = Foo.prototype;

// 会如你期望的那样工作
// 但会带有你可能不想要的副作用 :(
Bar.prototype = new Foo();
```

`Bar.prototype = Foo.prototype` 不会创建新对象让 `Bar.prototype` 链接。它只是让 `Bar.prototype` 成为 `Foo.prototype` 的另一个引用，将 `Bar` 直接链到 `Foo` 链着的同一个对象：`Foo.prototype`。这意味着当你开始赋值时，比如 `Bar.prototype.myLabel = ...`，你修改的不是一个分离的对象而是那个被分享的 `Foo.prototype` 对象本身，它将影响到所有链接到 `Foo.prototype` 的对象。这几乎可以确定不是你想要的。如果这正是你想要的，那么你根本就不需要 `Bar`，你应当仅使用 `Foo` 来使你的代码更简单。

`Bar.prototype = new Foo()` 确实创建了一个新的对象，这个新对象也的确链接到了我们希望的 `Foo.prototype`。但是，它是用 `Foo(..)` “构造器调用”来这样做的。如果这个函数有任何副作用（比如`logging`，改变状态，注册其他对象，向 `this` 添加数据属性，等等），这些副作用就会在链接时发生（而且很可能是对错误的对象！），而不是像可能希望的那样，仅最终在 `Bar()` 的“后裔”被创建时发生。

于是，我们剩下的选择就是使用 `Object.create(..)` 来制造一个新对象，这个对象被正确地链接，而且没有调用 `Foo(..)` 时所产生的副作用。一个轻微的缺点是，我们不得不创建新对象，并把旧的扔掉，而不是修改提供给我们的默认既存对象。

如果有一种标准且可靠地方法来修改既存对象的链接就好了。ES6之前，有一个非标准的，而且不是完全对所有浏览器通用的方法：通过可以设置的 `__proto__` 属性。ES6中增加了 `Object.setPrototypeOf(..)` 辅助工具，它提供了标准且可预见的方法。

让我们一对一对地比较ES6之前和ES6标准的技术如何处理将 `Bar.prototype` 链接至 `Foo.prototype` :

```
// ES6以前
// 扔掉默认既存的`Bar.prototype`
Bar.prototype = Object.create( Foo.prototype );

// ES6+
// 修改既存的`Bar.prototype`
Object.setPrototypeOf( Bar.prototype, Foo.prototype );
```

如果忽略 `Object.create(..)` 方式在性能上的轻微劣势（扔掉一个对象，然后被回收），其实它相对短一些而且可能比ES6+的方式更易读。但两种方式可能都只是语法表面现象。

## 考察“类”关系

如果你有一个对象 `a` 并且希望找到它委托至哪个对象呢（如果有的话）？考察一个实例（一个JS对象）的继承血统（在JS中是委托链接），在传统的面向类环境中称为自省 (*introspection*)（或反射 (*reflection*)）。

考虑下面的代码：

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

Foo.prototype.blah = ...;

var a = new Foo();
```

那么我们如何自省 `a` 来找到它的“祖先”（委托链）呢？一种方式是接受“类”的困惑：

```
a instanceof Foo; // true
```

`instanceof` 操作符的左边操作数接收一个普通对象，右边操作数接收一个函数。`instanceof` 回答的问题是：在 `a` 的整个 `[[Prototype]]` 链中，有没有出现被那个被 `Foo.prototype` 所随便指向的对象？

不幸的是，这意味着如果你拥有可以用于测试的函数（`Foo`，和它带有的 `.prototype` 引用），你只能查询某些对象（`a`）的“祖先”。如果你有两个任意的对象，比如 `a` 和 `b`，而且你想调查是否这些对象通过 `[[Prototype]]` 链相互关联，单靠 `instanceof` 帮不上什么忙。

注意：如果你使用内建的 `.bind(..)` 工具来制造一个硬绑定的函数（见第二章），这个被创建的函数将不会拥有 `.prototype` 属性。将 `instanceof` 与这样的函数一起使用时，将会透明地替换为创建这个硬绑定函数的目标函数的 `.prototype`。

将硬绑定函数用于“构造器调用”十分罕见，但如果你这么做，它会表现得好像是目标函数被调用了，这意味着将 `instanceof` 与硬绑定函数一起使用也会参照原版函数。

下面这段代码展示了试图通过“类”的语义和 `instanceof` 来推导两个对象间的关系是多么荒谬：

```
// 用来检查`o1`是否关联到（委托至）`o2`的帮助函数
function isRelatedTo(o1, o2) {
    function F(){}
    F.prototype = o2;
    return o1 instanceof F;
}

var a = {};
var b = Object.create( a );

isRelatedTo( b, a ); // true
```

在 `isRelatedTo(..)` 内部，我们借用一个一次性的函数 `F`，重新对它的 `.prototype` 赋值，使他随意地指向某个对象 `o2`，之后问是否 `o1` 是 `F` 的“一个实例”。很明显，`o1` 实际上不是继承或遗传自 `F`，甚至不是由 `F` 构建的，所以显而易见这种实践是愚蠢且让人困惑的。这个问题归根结底是将类的语义强加于 JavaScript 的尴尬，在这个例子中是由 `instanceof` 的间接语义揭露的。

第二种，也是更干净的方式，`[[Prototype]]` 反射：

```
Foo.prototype.isPrototypeOf( a ); // true
```

注意在这种情况下，我们并不真正关心（甚至不需要）`Foo`，我们仅需要一个对象（在我们的例子中就是随意标志为 `Foo.prototype`）来与另一个对象测试。`isPrototypeOf(..)` 回答的问题是：在 `a` 的整个 `[[Prototype]]` 链中，`Foo.prototype` 出现过吗？

同样的问题，和完全同样的答案。但是在第二种方式中，我们实际上不需要间接地引用一个 `.prototype` 属性将被自动查询的函数（`Foo`）。

我们只需要两个对象来考察它们之间的关系。比如：

```
// 简单地：`b`在`c`的`[[Prototype]]`链中出现过吗？
b.isPrototypeOf( c );
```

注意，这种方法根本不要求有一个函数（“类”）。它仅仅使用对象的直接引用 `b` 和 `c`，来查询他们的关系。换句话说，我们上面的 `isRelatedTo(..)` 工具是内建在语言中的，它的名字叫 `isPrototypeOf(..)`。

我们也可以直接取得一个对象的 `[[Prototype]]`。在ES5中，这么做的标准方法是：

```
Object.getPrototypeOf( a );
```

而且你将注意到对象引用是我们期望的：

```
Object.getPrototypeOf( a ) === Foo.prototype; // true
```

大多数浏览器（不是全部！）还一种长期支持的，非标准方法可以访问内部的 `[[Prototype]]`：

```
a.__proto__ === Foo.prototype; // true
```

这个奇怪的 `__proto__`（直到ES6才标准化！）属性“魔法般地”取得一个对象内部的 `[[Prototype]]` 作为引用，如果你想要直接考察（甚至遍历：`__proto__.__proto__...`）`[[Prototype]]` 链，这个引用十分有用。

和我们早先看到的 `.constructor` 一样，`__proto__` 实际上不存在于你考察的对象上（在我们的例子中是 `a`）。事实上，它存在于（不可枚举地；见第二章）内建的 `Object.prototype` 上，和其他的共通工具在一起(`.toString()`, `.isPrototypeOf(..)`, 等等)。

而且，`__proto__` 看起来像一个属性，但实际上将它看做是一个getter/setter（见第三章）更合适。

大致地，我们可以这样描述 `__proto__` 实现（见第三章，对象属性的定义）：

```
Object.defineProperty( Object.prototype, "__proto__", {
  get: function() {
    return Object.getPrototypeOf( this );
  },
  set: function(o) {
    // setPrototypeOf(..) as of ES6
    Object.setPrototypeOf( this, o );
    return o;
  }
});
```

所以，当我们访问 `a.__proto__` (取得它的值) 时，就好像调用 `a.__proto__()` (调用getter函数)。虽然getter函数存在于 `Object.prototype` 上 (参照第二章，`this` 绑定规则)，但这个函数调用将 `a` 用作它的 `this`，所以它相当于在说 `Object.getPrototypeOf(a)`。

`.__proto__` 还是一个可设置的属性，就像早先展示过的ES6 `Object.setPrototypeOf(..)`。然而，一般来说你 不应该改变一个既存对象的 `[[Prototype]]`。

在某些允许对 `Array` 定义“子类”的框架中，深度地使用了一些非常复杂，高级的技术，但是在一般的编程实践中经常是让人皱眉头的，因为这通常导致非常难理解/维护的代码。

注意：在ES6中，关键字 `class` 将允许某些近似方法，对像 `Array` 这样的内建类型“定义子类”。参见附录A中关于ES6中加入的 `class` 的讨论。

仅有一小部分例外（就像前面提到过的），会设置一个默认函数 `.prototype` 对象的 `[[Prototype]]`，使它引用其他的对象 (`Object.prototype` 之外的对象)。它们会避免将这个默认对象完全替换为一个新的链接对象。否则，为了在以后更容易地阅读你的代码 最好将对象的 `[[Prototype]]` 链接作为只读性质对待。

注意：针对双下划线，特别是在像 `__proto__` 这样的属性中开头的部分，JavaScript社区非官方地创造了一个术语：“dunder”。所以，那些JavaScript的“酷小子”们通常将 `__proto__` 读作“dunder proto”。

## 对象链接

正如我们看到的，`[[Prototype]]` 机制是一个内部链接，它存在于一个对象上，这个对象引用一些其他的对象。

这种链接（主要）在对第一个对象进行属性/方法引用，但这样的属性/方法不存在时实施。在这种情况下，`[[Prototype]]` 链接告诉引擎在那个被链接的对象上查找这个属性/方法。接下来，如果这个对象不能满足查询，它的 `[[Prototype]]` 又会被查找，如此继续。这个在对象间的一系列链接构成了所谓的“原形链”。

## 创建链接

我们已经彻底揭露了为什么JavaScript的 `[[Prototype]]` 机制和 类 不一样，而且我们也看到了如何在正确的对象间创建链接。

`[[Prototype]]` 机制的意义是什么？为什么总是见到JS开发者们费那么大力气（模拟类）在他们的代码中搞乱这些链接？

记得我们在本章很靠前的地方说过 `Object.create(..)` 是英雄吗？现在，我们准备好好看看为什么了。

```

var foo = {
    something: function() {
        console.log( "Tell me something good..." );
    }
};

var bar = Object.create( foo );

bar.something(); // Tell me something good...

```

`Object.create(..)` 创建了一个链接到我们指定的对象 (`foo`) 上的新对象 (`bar`)，这给了我们 `[[Prototype]]` 机制的所有力量（委托），而且没有 `new` 函数作为类和构造器调用产生的任何没必要的复杂性，搞乱 `.prototype` 和 `.constructor` 引用，或任何其他的多余的东西。

注意：`Object.create(null)` 创建一个拥有空（也就是 `null`）`[[Prototype]]` 链接的对象，如此这个对象不能委托到任何地方。因为这样的对象没有原形链， `instanceof`  操作符（前面解释过）没有东西可检查，所以它总返回 `false`。由于他们典型的用途是在属性中存储数据，这种特殊的空 `[[Prototype]]` 对象经常被称为“dictionaries（字典）”，这主要是因为它们没有可能受到在 `[[Prototype]]` 链上任何委托属性/函数的影响，所以它们是纯粹的扁平数据存储。

我们不需要类来在两个对象间创建有意义的关系。我们需要真正关心的唯一问题是对象为了委托而链接在一起，而 `Object.create(..)` 给我们这种链接并且没有一切关于类的烂设计。

## 填补 `Object.create()`

`Object.create(..)` 在ES5中被加入。你可能需要支持ES5之前的环境（比如老版本的IE），所以让我们来看一个 `Object.create(..)` 的简单部分填补工具，它甚至能在更老的JS环境中给我们所需的能力：

```

if (!Object.create) {
    Object.create = function(o) {
        function F(){};
        F.prototype = o;
        return new F();
    };
}

```

这个填补工具通过一个一次性的 `F` 函数并覆盖它的 `.prototype` 属性来指向我们想连接到的对象。之后我们用 `new F()` 构造器调用来制造一个将会链到我们指定对象上的新对象。

`Object.create(..)` 的这种用法是目前最常见的用法，因为他的这一部分是可以填补的。ES5标准的内建 `Object.create(..)` 还提供了一个附加的功能，它是不能被ES5之前的版本填补的。如此，这个功能的使用远没有那么常见。为了完整性，让我么看看这个附加功能：

```
var anotherObject = {
  a: 2
};

var myObject = Object.create( anotherObject, {
  b: {
    enumerable: false,
    writable: true,
    configurable: false,
    value: 3
  },
  c: {
    enumerable: true,
    writable: false,
    configurable: false,
    value: 4
  }
} );

myObject.hasOwnProperty( "a" ); // false
myObject.hasOwnProperty( "b" ); // true
myObject.hasOwnProperty( "c" ); // true

myObject.a; // 2
myObject.b; // 3
myObject.c; // 4
```

`Object.create(..)` 的第二个参数指定了要添加在新对象上的属性名，通过声明每个新属性的属性描述符（见第三章）。因为在ES5之前的环境中填补属性描述符是不可能的，所以 `object.create(..)` 的这个附加功能无法填补。

因为 `Object.create(..)` 的绝大多数用途都是使用填补安全的功能子集，所以大多数开发者在ES5之前的环境中使用这种部分填补也没有问题。

有些开发者采取严格得多的观点，也就是除非能够被完全填补，否则没有函数应该被填补。因为 `Object.create(..)` 可以部分填补的工具之一，这种较狭窄的观点会说，如果你需要在ES5之前的环境中使用 `Object.create(..)` 的任何功能，你应当使用自定义的工具，而不是填充，而且应当彻底远离使用 `Object.create` 这个名字。你可以定义自己的工具，比如：

```

function createAndLinkObject(o) {
    function F(){}
    F.prototype = o;
    return new F();
}

var anotherObject = {
    a: 2
};

var myObject = createAndLinkObject( anotherObject );

myObject.a; // 2

```

我不会分享这种严格的观点。我完全拥护如上面展示的 `Object.create(..)` 的常见部分填补，甚至在ES5之前的环境下在你的代码中使用它。我将选择权留给你。

## 链接作为候补？

也许这么想很吸引人：这些对象间的链接主要是为了给“缺失”的属性和方法提供某种候补。虽然这是一个可观察到的结果，但是我不认为这是考虑 `[[Prototype]]` 的正确方法。

考虑下面的代码：

```

var anotherObject = {
    cool: function() {
        console.log( "cool!" );
    }
};

var myObject = Object.create( anotherObject );

myObject.cool(); // "cool!"

```

得益于 `[[Prototype]]`，这段代码可以工作，但如果你这样写是为了万一 `myObject` 不能处理某些开发者可能会调用的属性/方法，而让 `anotherObject` 作为一个候补，你的软件大概会变得有点儿“魔法”并且更难于理解和维护。

这不是说候补在任何情况下都不是一个合适的设计模式，但它不是一个在JS中很常见的用法，所以如果你发现自己在这么做，那么你可能想要退一步并重新考虑它是否真的是合适且合理的设计。

注意：在ES6中，引入了一个称为 `Proxy`（代理）的高级功能，它可以提供某种“方法未找到”类型的行为。`Proxy` 超出了本书的范围，但会在以后的“你不懂JS”系列图书中详细讲解。

这里不要错过一个重要的细节。

例如，你打算为一个开发者设计软件，如果即使在 `myObject` 上没有 `cool()` 方法时调用 `myObject.cool()` 也能工作，会在你的API设计上引入一些“魔法”气息，这可能会使未来维护你的软件的开发者很吃惊。

然而你可以在你的API设计上少用些“魔法”，而仍然利用 `[[Prototype]]` 链接的力量。

```
var anotherObject = {
  cool: function() {
    console.log( "cool!" );
  }
};

var myObject = Object.create( anotherObject );

myObject.doCool = function() {
  this.cool(); // internal delegation!
};

myObject.doCool(); // "cool!"
```

这里，我们调用 `myObject.doCool()`，它是一个实际存在于 `myObject` 上的方法，这使我们的 API设计更清晰（没那么“魔法”）。在它内部，我们的实现依照 委托设计模式（见第六章），利用 `[[Prototype]]` 委托到 `anotherObject.cool()`。

换句话说，如果委托是一个内部实现细节，而非在你的API结构设计中简单地暴露出来，它倾向于减少意外/困惑。我们会在下一章中详细解释 委托。

## 复习

当试图在一个对象上进行属性访问，而对象没有该属性时，对象内部的 `[[Prototype]]` 链接定义了 `[[Get]]` 操作（见第三章）下一步应当到哪里寻找它。这种对象到对象的串行链接定义了对象的“原形链”（和嵌套的作用域链有些相似），在解析属性时发挥作用。

所有普通的对象用内建的 `Object.prototype` 作为原形链的顶端（就像作用域查询的顶端是全局作用域），如果属性没能在链条的前面任何地方找到，属性解析就会在这里停止。`toString()`，`valueOf()`，和其他几种共同工具都存在于这个 `Object.prototype` 对象上，这解释了语言中所有的对象是如何能够访问他们的。

使两个对象相互链接在一起的最常见的方法是将 `new` 关键字与函数调用一起使用，在它的四个步骤中（见第二章），就会建立一个新对象链接到另一个对象。

那个用 `new` 调用的函数有一个被随便地命名为 `.prototype` 的属性，这个属性所引用的对象恰好就是这个新对象链接到的“另一个对象”。带有 `new` 的函数调用通常被称为“构造器”，尽管实际上它们并没有像传统的面向类语言那样初始化一个类。

虽然这些JavaScript机制看起来和传统面向类语言的“初始化类”和“类继承”类似，而在JavaScript中的关键区别是，没有拷贝发生。取而代之的是对象最终通过 `[[Prototype]]` 链链接在一起。

由于各种原因，不光是前面提到的术语，“继承”（和“原型继承”）与所有其他的OO用语，在考虑JavaScript实际如何工作时都没有道理。

相反，“委托”是一个更确切的术语，因为这些关系不是 拷贝 而是委托 链接。

# 你不懂JS: **this** 与对象原型

## 第六章: 行为委托

在第五章中，我们详细地讨论了 `[[Prototype]]` 机制，和为什么对于描述“类”或“继承”来说它是那么使人糊涂和不合适。我们一路跋涉，不仅涉及了相当繁冗的语法（使代码凌乱的 `.prototype`），还有各种陷阱（比如使人吃惊的 `.constructor` 解析和难看的假想多态语法）。我们探索了许多人试图用抹平这些粗糙的区域而使用的各种“mixin”方法。

这时一个常见的反应是，想知道为什么这些看起来如此简单的事情这么复杂。现在我们已经拉开帷幕看到了它是多么麻烦，这并不奇怪：大多数JS开发者从不探究得这么深，而将这一团糟交给一个“类”包去帮他们处理。

我希望到现在你不会甘心于敷衍了事并把这样的细节丢给一个“黑盒”库。现在我们来深入讲解我们如何与应当如何以一种比类造成的困惑简单得多而且更直接的方式来考虑JS中对象的 `[[Prototype]]` 机制。

简单地复习一下第五章的结论，`[[Prototype]]` 机制是一种存在于一个对象上的内部链接，它指向一个其他对象。

当一个属性/方法引用在第一个对象上发生，而这样的属性/方法又不存在时，这个链接就会被使用。在这种情况下，`[[Prototype]]` 链接告诉引擎去那个被链接的对象上寻找该属性/方法。接下来，如果那个对象也不能满足查询，就沿着它的 `[[Prototype]]` 查询，如此继续。这种对象间一系列的链接构成了所谓的“原形链”。

换句话说，对于我们能在JavaScript中利用的功能的实际机制来说，其重要的实质全部在于被连接到其他对象的对象。

这个观点是理解本章其余部分的动机和方法的重要基础！

## 迈向面相委托的设计

为了将我们的思想恰当地集中在如何用最直截了当的方法使用 `[[Prototype]]`，我们必须认识到它代表一种根本上与类不同的设计模式（见第四章）。

注意\* 某些面相类的设计依然很有效，所以不要扔掉你知道的每一件事（扔掉大多数就行了！）。比如，封装就十分强大，而且与委托兼容的（虽然不那么常见）。

我们需要试着将我们的思维从类/继承的设计模式转变为行为代理设计模式。如果你已经用在教育/职业生涯中思考类的方式做了大多数或所有的编程工作，这可能感觉不舒服或不自然。你可能需要尝试这种思维过程好几次，才能适应这种非常不同的思考方式。

我将首先带你进行一些理论练习，之后我们会一对一地看一些更实际的例子来为自己的代码提供实践环境。

## 类理论

比方说我们有几个相似的任务（“XYZ”，“ABC”，等）需要在我们的软件中建模。

使用类，你设计这个场景的方式是：定义一个泛化的父类（基类）比如 `Task`，为所有的“同类”任务定义共享的行为。然后，你定义子类 `XYZ` 和 `ABC`，它们都继承自 `Task`，每个都分别添加了特化的行为来处理各自的任务。

重要的是，类设计模式将鼓励你发挥继承的最大功效，当你在 `XYZ` 任务中覆盖 `Task` 的某些泛化方法的定义时，你将会想利用方法覆盖（和多态），也许会利用 `super` 来调用这个方法泛化版本，为它添加更多的行为。你很可能会找到几个可以“抽象”到父类中，或在子类中特化（覆盖）的地方。

这是一些关于这个场景的假想代码：

```
class Task {
    id;

    // `Task()``构造器
    Task(ID) { id = ID; }
    outputTask() { output(id); }
}

class XYZ inherits Task {
    label;

    // `XYZ()``构造器
    XYZ(ID,Label) { super(ID); label = Label; }
    outputTask() { super(); output(label); }
}

class ABC inherits Task {
    // ...
}
```

现在，你可以初始化一个或多个 `XYZ` 子类的拷贝，并且使用这些实例来执行“XYZ”任务。这些实例已经同时拷贝了泛化的 `Task` 定义的行为和具体的 `XYZ` 定义的行为。类似地，`ABC` 类的实例将拷贝 `Task` 的行为和具体的 `ABC` 的行为。在构建完成之后，你一般会仅与这些实例互动（而不是类），因为每个实例都拷贝了完成计划任务的所有行为。

## 委托理论

但是现在然我们试着用 行为委托 代替 类 来思考同样的问题。

你将首先定义一个称为 `Task` 的 对象（不是一个类，也不是一个大多数JS开发者想让你相信的 `function`），而且它将拥有具体的行为，这些行为包含各种任务可以使用的（读作：委托至！）工具方法。然后，对于每个任务（“XYZ”，“ABC”），你定义一个 对象 来持有这个特定任务的数据/行为。你 链接 你的特定任务对象到 `Task` 工具对象，允许它们在必要的时候可以 委托 到它。

基本上，你认为执行任务“XYZ”就是从两个兄弟/对等的对象（`XYZ` 和 `Task`）中请求行为来完成它。与其通过类的拷贝将它们组合在一起，我们可以将他们保持在分离的对象中，而且可以在需要的情况下允许 `XYZ` 对象来 委托 到 `Task`。

这里是一些简单的代码，示意你如何实现它：

```
var Task = {
    setID: function(ID) { this.id = ID; },
    outputID: function() { console.log( this.id ); }
};

// 使`XYZ`委托到`Task`
var XYZ = Object.create( Task );

XYZ.prepareTask = function(ID, Label) {
    this.setID( ID );
    this.label = Label;
};

XYZ.outputTaskDetails = function() {
    this.outputID();
    console.log( this.label );
};

// ABC = Object.create( Task );
// ABC ... = ...
```

在这段代码中，`Task` 和 `XYZ` 不是类（也不是函数），它们 仅仅是对象。`XYZ` 通过 `Object.create()` 创建，来 `[[Prototype]]` 委托到 `Task` 对象（见第五章）。

作为与面相类（也就是，OO——面相对象）的对比，我称这种风格的代码为“**OLOO**”（objects-linked-to-other-objects（链接到其他对象的对象））。所有我们真正关心的是，对象 `XYZ` 委托到对象 `Task`（对象 `ABC` 也一样）。

在JavaScript中，`[[Prototype]]` 机制将 对象 链接到其他 对象。无论你多么想说服自己这不是真的，JavaScript没有像“类”那样的抽象机制。这就像逆水行舟：你可以做到，但你选择了逆流而上，所以很明显地，你会更困难地达到目的地。

**OLOO**风格的代码 中有一些需要注意的不同：

1. 前一个类的例子中的 `id` 和 `label` 数据成员都是 `XYZ` 上的直接数据属性（它们都不在 `Task` 上）。一般来说，当 `[[Prototype]]` 委托引入时，你想使状态保持在委托者上（`XYZ`，`ABC`），不是在委托上（`Task`）。
2. 在类的设计模式中，我们故意在父类（`Task`）和子类（`XYZ`）上采用相同的命名 `outputTask`，以至于我们可以利用覆盖（多态）。在委托的行为中，我们反其道而行之：我们尽一切可能避免在 `[[Prototype]]` 链的不同层级上给出相同的命名（称为“遮蔽”——见第五章），因为这些命名冲突会导致尴尬/脆弱的语法来消除引用的歧义（见第四章），而我们想避免它。这种设计模式不那么要求那些倾向于被覆盖的泛化的方法名，而是要求针对于每个对象的具体行为类型给出更具描述性的方法名。这实际上会产生更易于理解/维护的代码，因为方法名（不仅在定义的位置，而是扩散到其他代码中）变得更加明白（代码即文档）。
3. `this.setID(ID);` 位于对象 `XYZ` 的一个方法内部，它首先在 `XYZ` 上查找 `setID(..)`，但因为它不能在 `XYZ` 上找到叫这个名称的方法，`[[Prototype]]` 委托意味着它可以沿着链接到 `Task` 来寻找 `setID()`，这样当然就找到了。另外，由于调用点的隐含 `this` 绑定规则（见第二章），当 `setID()` 运行时，即便方法是在 `Task` 上找到的，这个函数调用的 `this` 绑定依然还是我们期望和想要的 `XYZ`。我们在代码稍后的 `this.outputID()` 中也看到了同样的事情。换句话说，我们可以使用存在于 `Task` 上的泛化工具与 `XYZ` 互动，因为 `XYZ` 可以委托至 `Task`。

行为委托 意味着：在某个对象（`XYZ`）的属性或方法没能在这个对象（`XYZ`）上找到时，让这个对象（`XYZ`）为属性或方法引用提供一个委托（`Task`）。

这是一个 极其强大的设计模式，与父类和子类，继承，多态等有很大的不同。与其在你的思维中纵向地，从上面父类到下面子类地组织对象，你应带并列地，对等地考虑对象，而且对象间拥有方向性的委托链接。

注意： 委托更适于作为内部实现的细节，而不是直接暴露在API接口的设计中。在上面的例子中，我们的API设计没必要有意地让开发者调用 `XYZ.setID()`（当然我们可以！）。我们以某种隐藏的方式将委托作为我们API的内部细节，即 `XYZ.prepareTask(..)` 委托到 `Task.setID(..)`。详细的内容，参照第五章的“链接作为候补？”中的讨论。

## 相互委托（不允许）

你不能在两个或多个对象间相互地委托（双向地）对方来创建一个循环。如果你使 `B` 链接到 `A`，然后试着让 `A` 链接到 `B`，那么你将得到一个错误。

这样的事情不被允许有些可惜（不是非常令人惊讶，但稍稍有些恼人）。如果你制造一个在任意一方都不存在的属性/方法引用，你就会在 `[[Prototype]]` 上得到一个无限递归的循环。但如果所有的引用都严格存在，那么 `B` 就可以委托至 `A`，或相反，而且它可以工作。这意味着你可以为了多种任务用这两个对象互相委托至对方。有一些情况这可能会有用。

但它不被允许是因为引擎的实现者发现，在设置时检查（并拒绝！）无限循环引用一次，要比每次你在一个对象上查询属性时都做相同检查的性能要高。

## 调试

我们将简单地讨论一个可能困扰开发者的微妙的细节。一般来说，JS语言规范不会控制浏览器开发者工具如何向开发者表示指定的值/结构，所以每种浏览器/引擎都自由地按需要解释这个事情。因此，浏览器/工具不总是意见统一。特别地，我们现在要考察的行为就是当前仅在Chrome的开发者工具中观察到的。

考虑这段传统的“类构造器”风格的JS代码，正如它将在Chrome开发者工具控制台中出现的：

```
function Foo() {}

var a1 = new Foo();

a1; // Foo {}
```

让我们看一下这个代码段的最后一行：对表达式 `a1` 进行求值的输出，打印 `Foo {}`。如果你在FireFox中试用同样的代码，你很可能会看到 `Object {}`。为什么会有不同？这些输出意味着什么？

Chrome实质上在说“`{}`是一个由名为‘Foo’的函数创建的空对象”。Firefox在说“`{}`是一个由Object普通构建的空对象”。这种微妙的区别是因为Chrome在像一个内部属性一样，动态跟踪执行创建的实际方法的名称，而其他浏览器不会跟踪这样的附加信息。

试图用JavaScript机制来解释它很吸引人：

```
function Foo() {}

var a1 = new Foo();

a1.constructor; // Foo(){}
a1.constructor.name; // "Foo"
```

那么，Chrome就是通过简单地查看对象的 `.constructor.name` 来输出“Foo”的？令人费解的是，答案既是“是”也是“不”。

考虑下面的代码：

```

function Foo() {}

var a1 = new Foo();

Foo.prototype.constructor = function Gotcha() {};

a1.constructor; // Gotcha(){}
a1.constructor.name; // "Gotcha"

a1; // Foo {}

```

即便我们将 `a1.constructor.name` 合法地改变为其他的东西（“Gotcha”），Chrome控制台依旧使用名称“Foo”。

那么，说明前面问题（它使用 `.constructor.name` 吗？）的答案是不，他一定在内部追踪其他的什么东西。

但是，且慢！让我们看看这种行为如何与OLOO风格的代码一起工作：

```

var Foo = {};

var a1 = Object.create( Foo );

a1; // Object {}

Object.defineProperty( Foo, "constructor", {
    enumerable: false,
    value: function Gotcha(){}
});

a1; // Gotcha {}

```

啊哈！**Gotcha**，Chrome的控制台 确实 寻找并且使用了 `.constructor.name`。实际上，就在写这本书的时候，正是这个行为被认定为是Chrome的一个Bug，而且就在你读到这里的时候，它可能已经被修复了。所以你可能已经看到了被修改过的 `a1; // Object{}`。

这个bug暂且不论，Chrome执行的（刚刚在代码段中展示的）“构造器名称”内部追踪（目前仅用于调试输出的目的），是一个仅在Chrome内部存在的扩张行为，它已经超出了JS语言规范要求的范围。

如果你不使用“构造器”来制造你的对象，就像我们在本章的OLOO风格代码中不鼓励的那样，那么你将会得到一个Chrome不会为其追踪内部“构造器名称”的对象，所以这样的对象将正确地仅仅被输出“Object {}”，意味着“从Object()构建生成的对象”。

不要认为 这代表一个OLOO风格代码的缺点。当你用OLOO编码而且用行为代理作为你的设计模式时，谁“创建了”（也就是，哪个函数被和 `new` 一起调用了？）一些对象是一个无关的细节。Chrome特殊的内部“构造器名称”追踪仅仅在你完全接受“类风格”编码时才有用，而在你

接受OLOO委托时是没有意义的。

## 思维模型比较

现在你至少在理论上可以看到“类”和“委托”设计模式的不同了，让我们看看这些设计模式在我们用来推导我们代码的思维模型上的含义。

我们将查看一些更加理论上的（“Foo”，“Bar”）代码，然后比较两种方法（OO vs. OLOO）的代码实现。第一段代码使用经典的（“原型的”）OO风格：

```
function Foo(who) {
    this.me = who;
}
Foo.prototype.identify = function() {
    return "I am " + this.me;
};

function Bar(who) {
    Foo.call( this, who );
}
Bar.prototype = Object.create( Foo.prototype );

Bar.prototype.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = new Bar( "b1" );
var b2 = new Bar( "b2" );

b1.speak();
b2.speak();
```

父类 `Foo`，被子类 `Bar` 继承，之后 `Bar` 被初始化两次：`b1` 和 `b2`。我们得到的是 `b1` 委托至 `Bar.prototype`，`Bar.prototype` 委托至 `Foo.prototype`。这对你来说应当看起来十分熟悉。没有太具开拓性的东西发生。

现在，让我们使用 OLOO 风格的代码 实现完全相同的功能：

```

var Foo = {
    init: function(who) {
        this.me = who;
    },
    identify: function() {
        return "I am " + this.me;
    }
};

var Bar = Object.create( Foo );

Bar.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = Object.create( Bar );
b1.init( "b1" );
var b2 = Object.create( Bar );
b2.init( "b2" );

b1.speak();
b2.speak();

```

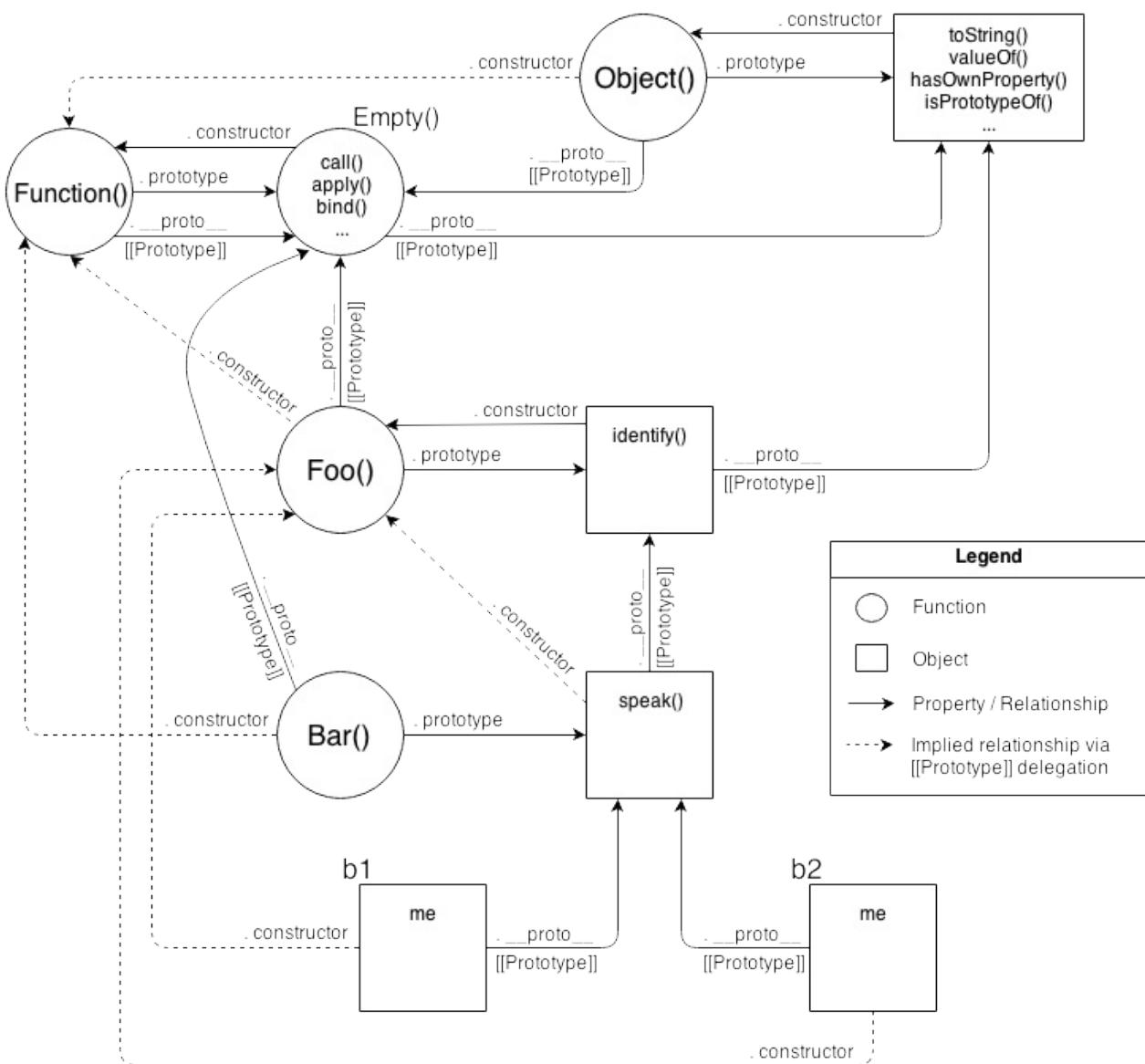
我们利用了完全相同的从 `Bar` 到 `Foo` 的 `[[Prototype]]` 委托，正如我们在前一个代码段中 `b1`，`Bar.prototype`，和 `Foo.prototype` 之间那样。我们仍然有3个对象链接在一起。

但重要的是，我们极大地简化了发生的所有其他事项，因为我们现在仅仅建立了相互链接的对象，而不需要所有其他讨厌且困惑的看起来像类（但动起来不像）的东西，还有构造器，原型和 `new` 调用。

问问你自己：如果我能用OLOO风格代码得到我用“类”风格代码得到的一样的东西，但OLOO更简单而且需要考虑的事情更少，**OLOO**不是更好吗？

让我们讲解一下这两个代码段间涉及的思维模型。

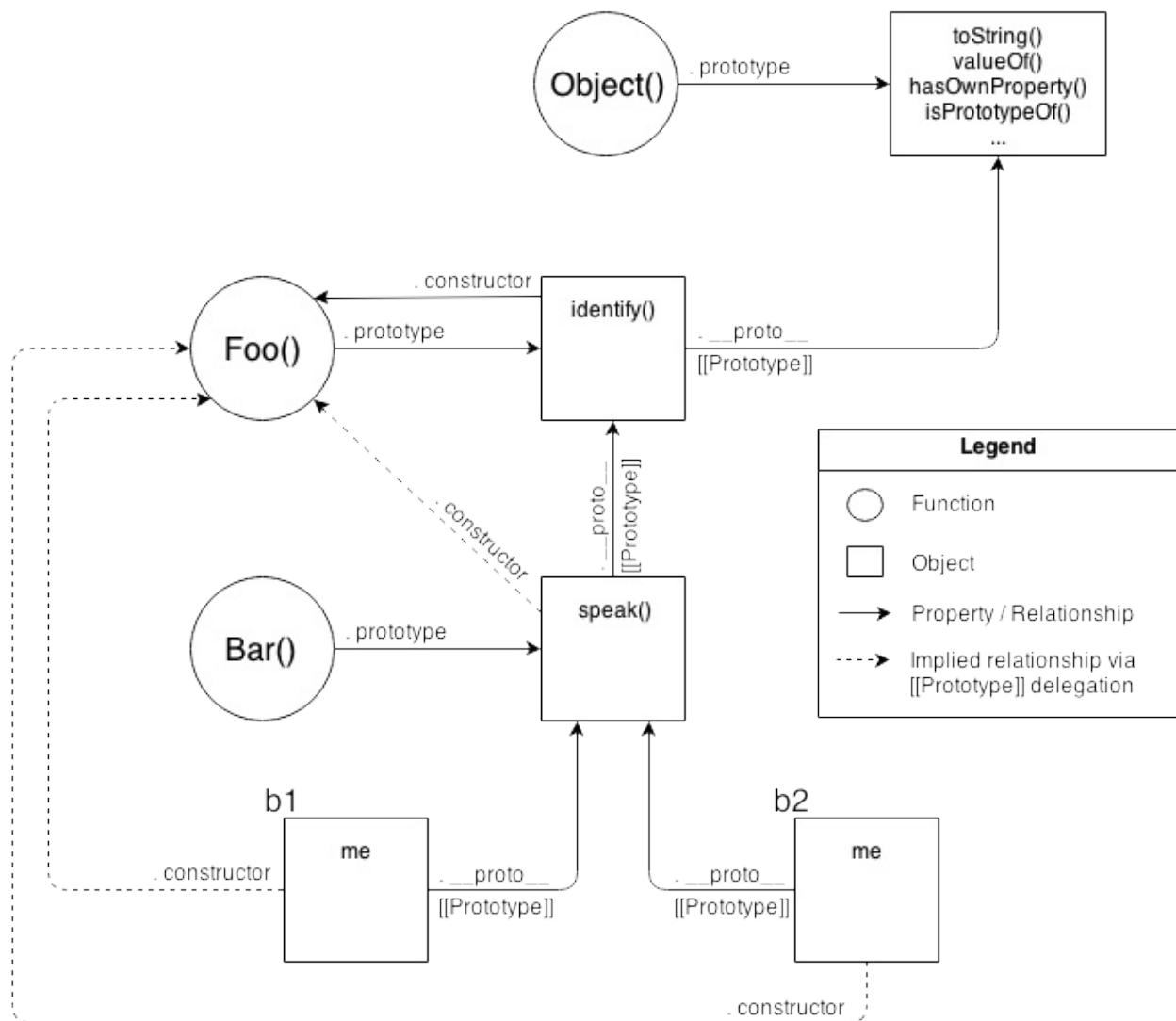
首先，类风给的代码段意味着这样的实体与它们的关系的思维模型：



实际上，这有点儿不公平/误导，因为它展示了许多额外的，你在技术上一直不需要知道（虽然你需要理解它）的细节。一个关键是，它是一系列十分复杂的关系。但另一个关键是：如果你花时间来沿着这些关系的箭头走，在JS的机制中有数量惊人的内部统一性。

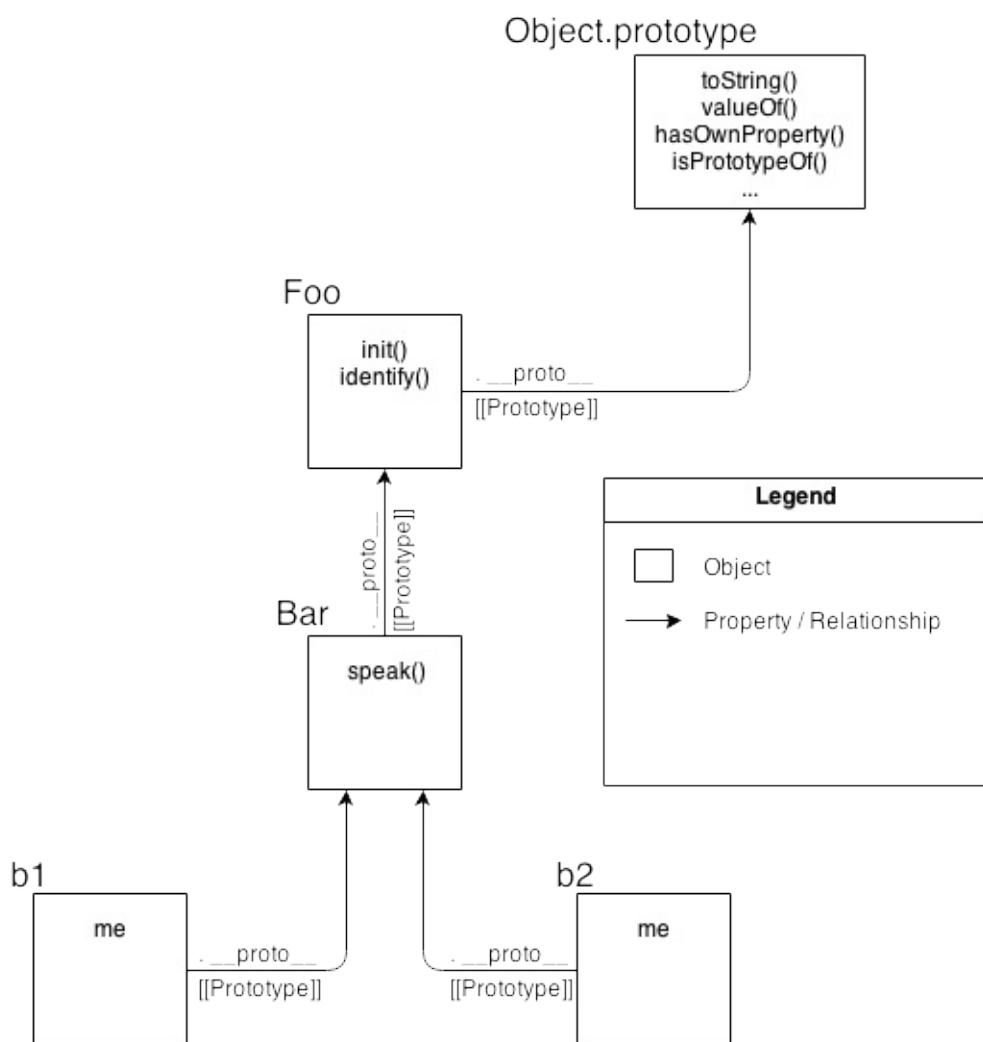
例如，JS函数可以访问 `call(..)`，`apply(..)` 和 `bind(..)`（见第二章）的能力是因为函数本身是对象，而函数对象还拥有一个 `[[Prototype]]` 链接，链到 `Function.prototype` 对象，它定义了那些任何函数对象都可以委托到的默认方法。JS可以做这些事情，你也能！

好了，现在让我们看一个这张图的稍稍简化的版本，用它来进行比较稍微“公平”一点——它仅展示了相关的实体与关系。



任然非常复杂，对吧？虚线描绘了当你在 `Foo.prototype` 和 `Bar.prototype` 间建立“继承”时的隐含关系，而且还没有修复丢失的 `.constructor` 属性引用（见第五章“复活构造器”）。即便将虚线去掉，每次你与对象链接打交道时，这个思维模型依然要变很多可怕的戏法。

现在，然我们看看OLOO风格代码的思维模型：



正如你所比较它们得到的，十分明显，OLOO风格的代码 需要关心的东西少太多了，因为 OLOO风格代码接受了 事实：我们唯一需要真正关心的事情是 链接到其他对象的对象。

所有其他“类”的烂设计用一种令人费解而且复杂的方式得到相同的结果。去掉那些东西，事情就变得简单得多（还不会失去任何功能）。

## Classes vs. Objects

我们已经看到了各种理论的探索和“类”与“行为委托”的思维模型的比较。现在让我们来看看更具体的代码场景，来展示你如何实际应用这些想法。

我们将首先讲解一种在前端网页开发中的典型场景：建造UI部件（按钮，下拉列表等等）。

### Widget“类”

因为你可能还是如此地习惯于OO设计模式，你很可能会立即这样考虑这个问题：一个父类（也许称为 `wedget`）拥有所有共通的基本部件行为，然后衍生的子类拥有具体的部件类型（比如 `Button`）。

注意：为了DOM和CSS的操作，我们将在这里使用JQuery，这仅仅是因为对于我们现在的讨论，它不是一个我们真正关心的细节。这些代码中不关心你用哪个JS框架（JQuery，Dojo，YUI等等）来解决如此无趣的问题。

让我们来看看，在没有任何“类”帮助库或语法的情况下，我们如何用经典风格的纯JS来实现“类”设计：

```

// 父类
function Widget(width,height) {
    this.width = width || 50;
    this.height = height || 50;
    this.$elem = null;
}

Widget.prototype.render = function($where){
    if (this.$elem) {
        this.$elem.css( {
            width: this.width + "px",
            height: this.height + "px"
        } ).appendTo( $where );
    }
};

// 子类
function Button(width,height,label) {
    // "super"构造器调用
    Widget.call( this, width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
}

// 使`Button`“继承”`Widget`
Button.prototype = Object.create( Widget.prototype );

// 覆盖“继承来的”`render(..)`
Button.prototype.render = function($where) {
    // "super"调用
    Widget.prototype.render.call( this, $where );
    this.$elem.click( this.onClick.bind( this ) );
};

Button.prototype.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
};

$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
    var btn2 = new Button( 150, 40, "World" );

    btn1.render( $body );
    btn2.render( $body );
} );

```

OO设计模式告诉我们要在父类中声明一个基础 `render(..)`，之后在我们的子类中覆盖它，但不是完全替代它，而是用按钮特定的行为增强这个基础功能。

注意 显示假想多态 的丑态，`Widget.call` 和 `Widget.prototype.render.call` 引用是为了伪装从子“类”方法得到“父类”基础方法支持的“super”调用。呃。

## ES6 class 语法糖

我们会在附录A中讲解ES6的 `class` 语法糖，但是让我们演示一下我们如何用 `class` 来实现相同的代码。

```
class Widget {
    constructor(width,height) {
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            }).appendTo( $where );
        }
    }
}

class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
        this.label = label || "Default";
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super.render( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    }
}

$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
    var btn2 = new Button( 150, 40, "World" );

    btn1.render( $body );
    btn2.render( $body );
});
```

毋庸置疑，通过使用ES6的 `class`，许多前面经典方法中语法的丑态被改善了。`super(..)` 的存在看起来非常适宜（但当你深入挖掘它时，不全是好事！）。

除了语法上的改进，这些都不是真正的类，因为他们仍然工作在 `[[Prototype]]` 机制之上。它们依然会受到思维模型不匹配的拖累，就像我们在第四，五章中，和直到现在探索的那样。附录A将会详细讲解ES6 `class` 语法和他的含义。我们将会看到为什么解决语法上的小问题不会实质上解决我们在JS中的类的困惑，虽然它做出了勇敢的努力假装解决了问题！

无论你是使用经典的原型语法还是新的ES6语法糖，你依然选择了使用“类”来对问题（UI部件）进行建模。正如我们前面几章试着展示的，在JavaScript中做这个选择会带给你额外的头疼和思维上的弯路。

## 委托部件对象

这是我们更简单的 `Widget / Button` 例子，使用了 **OLOO** 风格委托：

```

var Widget = {
    init: function(width,height){
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    },
    insert: function($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
};

var Button = Object.create( Widget );

Button.setup = function(width,height,label){
    // delegated call
    this.init( width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
};

Button.build = function($where) {
    // delegated call
    this.insert( $where );
    this.$elem.click( this.onClick.bind( this ) );
};

Button.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
};

$( document ).ready( function(){
    var $body = $( document.body );

    var btn1 = Object.create( Button );
    btn1.setup( 125, 30, "Hello" );

    var btn2 = Object.create( Button );
    btn2.setup( 150, 40, "World" );

    btn1.build( $body );
    btn2.build( $body );
} );

```

使用这种OLOO风格的方法，我们不认为 `Widget` 是一个父类而 `Button` 是一个子类，`Widget` 只是一个对象和某种具体类型的部件也许想要代理到的工具的集合，而且 `Button` 也是一个独立的对象（当然，带有委托至 `Widget` 的链接！）。

从设计模式的角度来看，我们没有像类的方法建议的那样，在两个对象中共享相同的 `render(..)` 方法名称，而是选择了更能描述每个特定任务的不同的名称。同样的原因，初始化方法被分别称为 `init(..)` 和 `setup(..)`。

不仅委托设计模式建议使用不同而且更具描述性的名称，而且在OLOO中这样做会避免难看的显式假想多态调用，正如你可以通过简单，相对的 `this.init(..)` 和 `this.insert(..)` 委托调用看到的。

语法上，我们也没有任何构造器，`.prototype` 或者 `new` 出现，它们事实上是不必要的设计。

现在，如果你再细心考察一下，你可能会注意到之前仅有一个调用（`var btn1 = new Button(..)`），而现在有了两个（`var btn1 = Object.create(Button)` 和 `btn1.setup(..)`）。这猛地看起来像是一个缺点（代码变多了）。

然而，即便是这样的事情，和经典原型风格比起来也是 **OLOO** 风格代码的优点。为什么？

用类的构造器，你“强制”（不完全是这样，但是被强烈建议）构建和初始化在同一个步骤中进行。然而，有许多种情况，能够将这两步分开做（就像你在OLOO中做的）更灵活。

举个例子，我们假定你在程序的最开始，在一个池中创建所有的实例，但你等到在它们被从池中找出并使用之前再用指定的设置初始化它们。我们的例子中，这两个调用紧挨在一起，当然它们也可以按需要发生在非常不同的时间和代码中非常不同的部分。

**OLOO** 对关注点分离原则有更好的支持，也就是创建和初始化没有必要合并在同一个操作中。

## 更简单的设计

OLOO除了提供表面上更简单（而且更灵活！）的代码之外，行为委托作为一个模式实际上会带来更简单的代码架构。让我们讲解最后一个例子来说明OLOO是如何简化你的整体设计的。

这个场景中我们将讲解两个控制器对象，一个用来处理网页的登录form（表单），另一个实际处理服务器的认证（通信）。

我们需要帮助工具来进行与服务器的Ajax通信。我们将使用JQuery（虽然其他的框架都可以），因为它不仅为我们处理Ajax，而且还返回一个类似Promise的应答，这样我们就可以在代码中使用 `.then(..)` 来监听这个应答。

注意：我们不会再这里讲到Promise，但我们在以后的 **你不懂JS** 系列中讲到。

根据典型的设计模式，我们在一个叫做 `controller` 的类中将任务分解为基本功能，之后我们会衍生出两个子类，`LoginController` 和 `AuthController`，它们都继承自 `controller` 而且特化某些基本行为。

```
// 父类
function Controller() {
    this.errors = [];
}
Controller.prototype.showDialog = function(title,msg) {
    // 在对话框中给用户显示标题和消息
};
Controller.prototype.success = function(msg) {
    this.showDialog( "Success", msg );
};
Controller.prototype.failure = function(err) {
    this.errors.push( err );
    this.showDialog( "Error", err );
};
```

```
// 子类
function LoginController() {
    Controller.call( this );
}
// 将子类链接到父类
LoginController.prototype = Object.create( Controller.prototype );
LoginController.prototype.getUser = function() {
    return document.getElementById( "login_username" ).value;
};
LoginController.prototype.getPassword = function() {
    return document.getElementById( "login_password" ).value;
};
LoginController.prototype.validateEntry = function(user,pw) {
    user = user || this.getUser();
    pw = pw || this.getPassword();

    if (!(user && pw)) {
        return this.failure( "Please enter a username & password!" );
    }
    else if (pw.length < 5) {
        return this.failure( "Password must be 5+ characters!" );
    }

    // 到这里了？输入合法！
    return true;
};
// 覆盖来扩展基本的`failure()`
LoginController.prototype.failure = function(err) {
    // "super"调用
    Controller.prototype.failure.call( this, "Login invalid: " + err );
};
```

```

// 子类
function AuthController(login) {
    Controller.call( this );
    // 除了继承外，我们还需要合成
    this.login = login;
}
// 将子类链接到父类
AuthController.prototype = Object.create( Controller.prototype );
AuthController.prototype.server = function(url,data) {
    return $.ajax( {
        url: url,
        data: data
    } );
};
AuthController.prototype.checkAuth = function() {
    var user = this.login.getUser();
    var pw = this.login.getPassword();

    if (this.login.validateEntry( user, pw )) {
        this.server( "/check-auth",{
            user: user,
            pw: pw
        })
        .then( this.success.bind( this ) )
        .fail( this.failure.bind( this ) );
    }
};
// 覆盖以扩展基本的`success()`
AuthController.prototype.success = function() {
    // "super"调用
    Controller.prototype.success.call( this, "Authenticated!" );
};
// 覆盖以扩展基本的`failure()`
AuthController.prototype.failure = function(err) {
    // "super"调用
    Controller.prototype.failure.call( this, "Auth Failed: " + err );
};

```

```

var auth = new AuthController(
    // 除了继承，我们还需要合成
    new LoginController()
);
auth.checkAuth();

```

我们有所有控制器分享的基本行为，它们是 `success(..)`，`failure(..)` 和 `showDialog(..)`。我们的子类 `LoginController` 和 `AuthController` 覆盖了 `failure(..)` 和 `success(..)` 来增强基本类的行为。还要注意的是，`AuthController` 需要一个 `LoginController` 实例来与登录form互动，所以它变成了一个数据属性成员。

另外一件要提的事情是，我们选择一些合成散布在继承的顶端。`AuthController` 需要知道 `LoginController`，所以我们初始化它（`new LoginController()`），使它一个成为 `this.login` 的类属性成员来引用它，这样 `AuthController` 才可以调用 `LoginController` 上的行为。

注意：这里可能会存在一丝冲动，就是使 `AuthController` 继承 `LoginController`，或者反过来，这样的话我们就会通过继承链得到虚拟合成。但是这是一个非常清晰的例子，表明对这个问题来讲，将类继承作为模型有什么问题，因为 `AuthController` 和 `LoginController` 都不特化对方的行为，所以它们之间的继承没有太大的意义，除非类是你唯一的设计模式。与此相反的是，我们在一些简单的合成中分层，然后它们就可以合作了，同时它们都享有继承自父类 `Controller` 的好处。

如果你熟悉面向类（OO）的设计，这都听该看起来十分熟悉和自然。

## 去类化

但是，我们真的需要用一个父类，两个子类，和一些合成来对这个问题建立模型吗？有办法利用OLOO风格的行为委托得到简单得多的设计吗？有的！

```
var LoginController = {
    errors: [],
    getUser: function() {
        return document.getElementById( "login_username" ).value;
    },
    getPassword: function() {
        return document.getElementById( "login_password" ).value;
    },
    validateEntry: function(user,pw) {
        user = user || this.getUser();
        pw = pw || this.getPassword();

        if (!(user && pw)) {
            return this.failure( "Please enter a username & password!" );
        }
        else if (pw.length < 5) {
            return this.failure( "Password must be 5+ characters!" );
        }

        // 到这里了？输入合法！
        return true;
    },
    showDialog: function(title,msg) {
        // 在对话框中向用于展示成功消息
    },
    failure: function(err) {
        this.errors.push( err );
        this.showDialog( "Error", "Login invalid: " + err );
    }
};
```

```
// 链接`AuthController`委托到`LoginController`、
var AuthController = Object.create( LoginController );

AuthController.errors = [];
AuthController.checkAuth = function() {
    var user = this.getUser();
    var pw = this.getPassword();

    if (this.validateEntry( user, pw )) {
        this.server( "/check-auth",{
            user: user,
            pw: pw
        })
        .then( this.accepted.bind( this ) )
        .fail( this.rejected.bind( this ) );
    }
};

AuthController.server = function(url,data) {
    return $.ajax( {
        url: url,
        data: data
    });
};

AuthController.accepted = function() {
    this.showDialog( "Success", "Authenticated!" )
};

AuthController.rejected = function(err) {
    this.failure( "Auth Failed: " + err );
};
```

因为 `AuthController` 只是一个对象 (`LoginController` 也是)，我们不需要初始化 (比如 `new AuthController()`) 就能执行我们的任务。所有我们要做的是：

```
AuthController.checkAuth();
```

当然，通过OLOO，如果你确实需要在委托链上创建一个或多个附加的对象时也很容易，而且仍然不需要任何像类实例化那样的东西：

```
var controller1 = Object.create( AuthController );
var controller2 = Object.create( AuthController );
```

使用行为委托，`AuthController` 和 `LoginController` 仅仅是对象，互相是水平对等的，而且没有被安排或关联成面向类中的父与子。我们有些随意地选择让 `AuthController` 委托至 `LoginController` —— 相反方向的委托也同样是有效的。

第二个代码段的主要要点是，我们只拥有两个实体 (`LoginController` and `AuthController`)，而不是之前的三个。

我们不需要一个基本的 `Controller` 类来在两个子类间“分享”行为，因为委托是一种可以给我们所需功能的，足够强大的机制。同时，就像之前注意的，我们也不需要实例化我们的对象来使它们工作，因为这里没有类，只有对象自身。另外，这里不需要合成作为委托来给两个对象差异化地合作的能力。

最后，由于没有让名称 `success(..)` 和 `failure(..)` 在两个对象上相同，我们避开了面向类的设计的多态陷阱：它将会需要难看的显式假想多态。相反，我们在 `AuthController` 上称它们为 `accepted()` 和 `rejected(..)` —— 对于他们的具体任务来说，稍稍更具描述性的名称。

底线：我们最终得到了相同的结果，但是用了（显著的）更简单的设计。这就是OLOO风格代码和行为委托设计模式的力量。

## 更好的语法

一个使ES6 `class` 看似如此诱人的更好的东西是（见附录A来了解为什么要避免它！），声明类方法的速记语法：

```
class Foo {
  methodName() { /* ... */ }
}
```

我们从声明中扔掉了单词 `function`，这使所有的JS开发者欢呼！

你可能已经注意到，而且为此感到沮丧：上面推荐的OLOO语法出现了许多 `function`，这看起来像对OLOO简化目标的诋毁。但它不必是！

在ES6中，我们可以在任何字面对象中使用简约方法声明，所以一个OLOO风格的对象可以用这种方式声明（与 `class` 语法中相同的语法糖）：

```
var LoginController = {
  errors: [],
  getUser() { // 看，没有`function`！
    // ...
  },
  getPassword() {
    // ...
  }
  // ...
};
```

唯一的区别是字面对象的元素间依然需要，逗号分隔符，而 `class` 语法不必如此。这是在整件事情上很小的让步。

还有，在ES6中，一个你使用的更笨重的语法（比如 `AuthController` 的定义中）：你一个一个地给属性赋值而不使用字面对象，可以改写为使用字面对象（于是你可以使用简约方法），而且你可以使用 `Object.setPrototypeOf(..)` 来修改对象的 `[[Prototype]]`，像这样：

```
// 使用更好的字面对象语法 w/ 简约方法！
var AuthController = {
  errors: [],
  checkAuth() {
    // ...
  },
  server(url,data) {
    // ...
  }
}
// ...

// 现在，链接`AuthController`委托至`LoginController`。
Object.setPrototypeOf( AuthController, LoginController );
```

ES6中的OLOO风格，与简明方法一起，变得比它以前友好得多（即使在以前，它也比经典的原型风格代码简单好看得多）。你不必非得选用类（复杂性）来得到干净漂亮的对象语法！

## 没有词法

简约方法确实有一个缺点，一个重要的细节。考虑这段代码：

```
var Foo = {
  bar() { /*..*/ },
  baz: function baz() { /*..*/ }
};
```

这是去掉语法糖后，这段代码将如何工作：

```
var Foo = {
  bar: function() { /*..*/ },
  baz: function baz() { /*..*/ }
};
```

看到区别了？`bar()` 的速记法变成了一个附着在 `bar` 属性上的匿名函数表达式 (`function()..`)，因为函数对象本身没有名称标识符。和拥有词法名称标识符 `baz`，附着在 `.baz` 属性上的手动指定的命名函数表达式 (`function baz()..`) 做个比较。

那又怎么样？在“你不懂JS”系列的“作用域与闭包”这本书中，我们详细讲解了匿名函数表达式的三个主要缺点。我们简单地重复一下它们，以便于我们和简明方法相比较。

一个匿名函数缺少 `name` 标识符：

1. 使调试时的栈追踪变得困难
2. 使自引用（递归，事件绑定等）变得困难
3. 使代码（稍稍）变得难于理解

第一和第三条不适用于简明方法。

虽然去掉语法糖使用 匿名函数表达式 一般会使栈追踪中没有 `name`。简明方法在语言规范中被要求去设置相应的函数对象内部的 `name` 属性，所以栈追踪应当可以使用它（这是依赖于具体实现的，所以不能保证）。

不幸的是，第二条 仍然是简明方法的一个缺陷。它们不会有词法标识符用来自引用。考虑：

```
var Foo = {
  bar: function(x) {
    if (x < 10) {
      return Foo.bar( x * 2 );
    }
    return x;
  },
  baz: function baz(x) {
    if (x < 10) {
      return baz( x * 2 );
    }
    return x;
  }
};
```

在这个例子中上面的手动 `Foo.bar(x*2)` 引用就足够了，但是在许多情况下，一个函数没必要能够这样做，比如使用 `this` 绑定，函数在委托中被分享到不同的对象，等等。你将会想要使用一个真正的自引用，而函数对象的 `name` 标识符是实现的最佳方式。

只要小心简明方法的这个注意点，而且如果当你陷入缺少自引用的问题时，仅仅为这个声明放弃简明方法语法，取代以手动的命名函数表达式 声明形式：`baz: function baz(){...}`。

## 自省

如果你花了很多时间在面向类的编程方式（不管是JS还是其他的语言），你可能会对 类型自省 很熟悉：自省一个实例来找出它是什么种类的对象。在类的实例上进行 类型自省 的主要目的是根据 对象是如何创建的 来推断它的结构/能力。

考虑这段代码，它使用 `instanceof`（见第五章）来自省一个对象 `a1` 来推断它的能力：

```

function Foo() {
    // ...
}

Foo.prototype.something = function(){
    // ...
}

var a1 = new Foo();

// 稍后

if (a1 instanceof Foo) {
    a1.something();
}

```

因为 `Foo.prototype` (不是 `Foo`!) 在 `a1` 的 `[[Prototype]]` 链上 (见第五章)，`instanceof` 操作符 (使人困惑地) 假装告诉我们 `a1` 是一个 `Foo` “类”的实例。有了这个知识，我们假定 `a1` 有 `Foo` “类”中描述的能力。

当然，这里没有 `Foo` 类，只有一个普通的函数 `Foo`，它恰好拥有一个引用指向一个随意的对象 (`Foo.prototype`)，而 `a1` 恰好委托链接至这个对象。通过它的语法，`instanceof` 假装检查了 `a1` 和 `Foo` 之间的关系，但它实际上告诉我们的却是 `a1` 和 `Foo.prototype` (这个随意被引用的对象) 是否有关联。

`instanceof` 在语义上的混乱 (和间接) 意味着，要使用以 `instanceof` 为基础的自省来查询对象 `a1` 是否与讨论中的对象有关联，你不得不拥有一个持有对这个对象引用的函数——你不能直接查询这两个对象是否有关联。

回想本章前面的抽象 `Foo` / `Bar` / `b1` 例子，我们在这里缩写一下：

```

function Foo() { /* .. */ }
Foo.prototype...

function Bar() { /* .. */ }
Bar.prototype = Object.create( Foo.prototype );

var b1 = new Bar( "b1" );

```

为了在这个例子中的实体上进行类型自省，使用 `instanceof` 和 `.prototype` 语义，这里有各种你可能需要实施的检查：

```
// 的`Foo`和`Bar`互相联系
Bar.prototype instanceof Foo; // true
Object.getPrototypeOf( Bar.prototype ) === Foo.prototype; // true
Foo.prototype.isPrototypeOf( Bar.prototype ); // true

// `b1`与`Foo`和`Bar`的联系
b1 instanceof Foo; // true
b1 instanceof Bar; // true
Object.getPrototypeOf( b1 ) === Bar.prototype; // true
Foo.prototype.isPrototypeOf( b1 ); // true
Bar.prototype.isPrototypeOf( b1 ); // true
```

可以说，其中有些烂透了。举个例子，直觉上（用类）你可能想说这样的东西 `Bar instanceof Foo` （因为很容易混淆“实例”的意义认为它包含“继承”），但在JS中这不是一个合理的比较。你不得不说 `Bar.prototype instanceof Foo`。

另一个常见，但也许健壮性更差的类型自省模式叫“duck typing（鸭子类型）”，比起 `instanceof` 来许多开发者都倾向于它。这个术语源自一则谚语，“如果它看起来像鸭子，叫起来像鸭子，那么它一定是一只鸭子”。

例如：

```
if (a1.something) {
    a1.something();
}
```

与其检查 `a1` 和一个持有可委托的 `something()` 函数的对象的关系，我们假设 `a1.something` 测试通过意味着 `a1` 有能力调用 `.something()`（不管是直接在 `a1` 上直接找到方法，还是委托至其他对象）。就其本身而言，这种假设没什么风险。

但是“鸭子类型”常常被扩展用于除了被测试关于对象能力以外的其他假设，这当然会在测试中引入更多风险（比如脆弱的设计）。

“鸭子类型”的一个值得注意的例子来自于ES6的Promises（就是我们前面解释过，将不再本书内涵盖的内容）。

由于种种原因，需要判定任意一个对象引用是否是一个Promise，但测试是通过检查对象是否恰好有 `then()` 函数出现在它上面来完成的。换句话说，如果任何对象恰好有一个 `then()` 方法，ES6的Promises将会无条件地假设这个对象是“thenable”的，而且因此会期望它按照所有的Promises标准行为那样一致地动作。

如果你有任何非Promise对象，而却不管因为什么它恰好拥有 `then()` 方法，你会被强烈建议使它远离ES6的Promise机制，来避免破坏这种假设。

这个例子清楚地展现了“鸭子类型”的风险。你应当仅在可控的条件下，保守地使用这种方式。

再次将我们的注意力转向本章中出现的OLOO风格的代码，类型自省变得清晰多了。让我们回想（并缩写）本章的 `Foo` / `Bar` / `b1` 的OLOO示例：

```
var Foo = { /* .. */ };

var Bar = Object.create( Foo );
Bar...

var b1 = Object.create( Bar );
```

使用这种OLOO方式，我们所拥有的一切都是通过 `[[Prototype]]` 委托关联起来的普通对象，这是我们可能会用到的大幅简化后的类型自省：

```
// `Foo`和`Bar`互相的联系
Foo.isPrototypeOf( Bar ); // true
Object.getPrototypeOf( Bar ) === Foo; // true

// `b1`与`Foo`和`Bar`的联系
Foo.isPrototypeOf( b1 ); // true
Bar.isPrototypeOf( b1 ); // true
Object.getPrototypeOf( b1 ) === Bar; // true
```

我们不再使用 `instanceof`，因为它令人迷惑地假装与类有关系。现在，我们只需要（非正式地）问这个问题，“你是我的一个原型吗？”。不再需要用 `Foo.prototype` 或者痛苦冗长的 `Foo.prototype.isPrototypeOf(..)` 来间接地查询了。

我想可以说这些检查比起前面一组自省检查，极大地减少了复杂性/混乱。又一次，我们看到了在JavaScript中OLOO要比类风格的编码简单（但有着相同的力量）。

## 复习

在你的软件体系结构中，类和继承是你可以选用或不选用的设计模式。多数开发者理所当然地认为类是组织代码的唯一（正确的）方法，但我们在那里看到了另一种不太常被提到的，但实际上十分强大的设计模式：行为委托。

行为委托意味着对象彼此是对等的，在它们自己当中相互委托，而不是父类与子类的关系。JavaScript的 `[[Prototype]]` 机制的设计本质，就是行为委托机制。这意味着我们可以选择挣扎着在JS上实现类机制，也可以欣然接受 `[[Prototype]]` 作为委托机制的本性。

当你仅用对象设计代码时，它不仅能简化你使用的语法，而且它还能实际上引领更简单的代码结构设计。

**OLOO**（链接到其他对象的对像）是一种没有类的抽象，而直接创建和关联对象的代码风格。OLOO十分自然地实现了基于 `[[Prototype]]` 的行为委托。



# 你不懂JS: **this** 与对象原型

## 附录A: ES6 class

如果说本书后半部分（第四到六章）有什么关键信息，那就是类是一种代码的可选设计模式（不是必要的），而且用像JavaScript这样的 `[[Prototype]]` 语言来实现它总是很尴尬。

虽然这种尴尬很大一部分关于语法，但不仅限于此。第四和第五章审视了相当多的难看语法，从使代码杂乱的 `.prototype` 引用的繁冗，到显式假想多态：当你在链条的不同层级上给方法相同的命名以试图实现从低层方法到高层方法的多态引用。`.constructor` 被错误地解释为“被XX构建”，这成为了一个不可靠的定义，也成为了另一个难看的语法。

但关于类的设计的问题要深刻多了。第四章指出在传统的面向类语言中，类实际上发生了从父类向子类，由子类向实例的拷贝动作，而在 `[[Prototype]]` 中，动作不是一个拷贝，而是相反——一个委托链接。

OLOO风格和行为委托接受了 `[[Prototype]]`，而不是将它隐藏起来，当比较它们的简单性时，类在JS中的问题就凸显出来。

### class

我们不必再次争论这些问题。我在这里简单地重提这些问题仅仅是为了使它们在你的头脑里保持新鲜，以使我们将注意力转向ES6的 `class` 机制。我们将在这里展示它如何工作，并且看看 `class` 是否实质上解决了任何这些“类”的问题。

让我们重温第六章的 `Widget / Button` 例子：

```

class Widget {
    constructor(width,height) {
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
}

class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
        this.label = label || "Default";
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super.render( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    }
}

```

除了语法上看起来更好，ES6还解决了什么？

- 不再有（某种意义上的，继续往下看！）指向 `.prototype` 的引用用来弄乱代码。
- `Button` 被声明为直接“继承自”（也就是 `extends`）`Widget`，而不是需要用 `Object.create(..)` 来替换 `.prototype` 链接的对象，或者用 `__proto__` 和 `Object.setPrototypeOf(..)` 来设置它。
- `super(..)` 现在给了我们非常有用相对多态的能力，所以在链条上某一个层级上的任何方法，可以引用链条上相对上一层的同名方法。第四章中有一个关于构造器的奇怪现象：构造器不属于它们的类，而且因此与类没有联系。`super(..)` 含有一个对此问题的解决方法——`super()` 会在构造器内部想如你期望的那样工作。
- `class` 字面语法对指定属性没有什么启发（仅对方法有）。这看起来限制了某些东西，但是绝大多数情况下期望一个属性（状态）存在于链条末端的“实例”以外的地方，这通常是一个错误和令人诧异（因为这个状态被隐含地在所有“实例”中“分享”）的。所以，也可以说 `class` 语法防止你出现错误。
- `extends` 甚至允许你用非常自然的方式扩展内建的对象（子）类型，比如 `Array` 或者 `RegExp`。在没有 `class .. extends` 的情况下这样做一直以来是一个极端复杂而令人沮

丧的任务，只有最熟练的框架作者曾经正确地解决过这个问题。现在，它是小菜一碟！

凭心而论，对大多数明显的（语法上的）问题，和经典的原型风格代码使人诧异的地方，这些确实是实质上的解决方案。

## class 的坑

然而，它不全是优点。在JS中将“类”作为一种设计模式，仍然有一些深刻和非常令人烦恼的问题。

首先，`class` 语法则可能会说服你JS在ES6中存在一个新的“类”机制。但不是这样。`class` 很大程度上仅仅是一个既存的`[[Prototype]]`（委托）机制的语法糖！

这意味着`class` 实际上不是像传统面向类语言那样，在声明时静态地拷贝定义。如果你在“父类”上更改/替换了一个方法（有意或无意地），子“类”和/或实例将会受到“影响”，因为它们在声明时没有得到一份拷贝，它们依然都使用那个基于`[[Prototype]]`的实时委托模型。

```
class C {
  constructor() {
    this.num = Math.random();
  }
  rand() {
    console.log( "Random: " + this.num );
  }
}

var c1 = new C();
c1.rand(); // "Random: 0.4324299..."

C.prototype.rand = function() {
  console.log( "Random: " + Math.round( this.num * 1000 ) );
};

var c2 = new C();
c2.rand(); // "Random: 867"

c1.rand(); // "Random: 432" -- oops!!!
```

这种行为只有在 你已经知道了 关于委托的性质，而不是期待从“真的类”中 拷贝 时，才看起来合理。那么你要问自己的问题是，为什么你为了根本上就和类不同的东西选择`class` 语法？

ES6的`class` 语法不是使观察和理解传统的类和委托对象间的不同 变得更困难了吗？

`class` 语法没有 提供声明类的属性成员的方法（仅对方法有）。所以如果你需要跟踪对象间分享的状态，那么你最终会回到难看的`.prototype` 语法，像这样：

```

class C {
  constructor() {
    // 确保修改的是共享状态
    // 不是设置实例上的遮蔽属性
    C.prototype.count++;

    // 这里，`this.count`通过委托如我们期望的那样工作
    console.log( "Hello: " + this.count );
  }
}

// 直接在原型对象上添加一个共享属性
C.prototype.count = 0;

var c1 = new C();
// Hello: 1

var c2 = new C();
// Hello: 2

c1.count === 2; // true
c1.count === c2.count; // true

```

这里最大的问题是，由于它将 `.prototype` 作为实现细节暴露（泄露！）出来，而背叛了 `class` 语法的初衷。

而且，我们还依然面临着那个令人诧异的陷阱：`this.count++` 将会隐含地在 `c1` 和 `c2` 两个对象上创建一个分离的遮蔽属性 `.count`，而不是更新共享的状态。`class` 没有在这个问题上给我们什么安慰，除了（大概是）通过缺少语法支持来暗示你根本就不应该这么做。

另外，无意地遮蔽依然是个灾难：

```

class C {
  constructor(id) {
    // 噢，一个坑，我们用实例上的属性值遮蔽了`id()`方法
    this.id = id;
  }
  id() {
    console.log( "Id: " + this.id );
  }
}

var c1 = new C( "c1" );
c1.id(); // TypeError -- `c1.id` 现在是字符串"c1"

```

还有一些关于 `super` 如何工作的微妙问题。你可能会假设 `super` 将会以一种类似与 `this` 得到绑定的方式（见第二章）来被绑定，也就是 `super` 总是会绑定到当前方法在 `[[Prototype]]` 链中的位置的更高一层。

然而，因为性能问题（`this` 绑定已经很耗费性能了），`super` 不是动态绑定的。它在声明时，被有些“静态地”绑定。不是什么大事儿，对吧？

恩……可能是，可能不是。如果你像大多数JS开发者那样，开始把函数赋值给不同的（来自于 `class` 定义的）对象，以各种不同的方式，你可能不会意识到在所有这些情况下，底层的 `super` 机制会不得不每次都重新绑定。

而且根据你每次赋值采取的语法方式不同，很有可能在某些情况下 `super` 不能被正确地绑定（至少不会像你期望的那样），所以你可能（在写作这里时，TC39正在讨论这个问题）会不得不使用 `toMethod(..)` 来手动绑定 `super`（有点儿像你不得不用 `bind(..)` 绑定 `this` —— 见第二章）。

你曾经可以给不同的对象赋予方法，来通过隐含绑定规则（见第二章），自动地利用 `this` 的动态性。但对于使用 `super` 的方法，同样的事情很可能不会发生。

考虑这里 `super` 应当怎样动作（对 `D` 和 `E`）：

```
class P {
  foo() { console.log( "P.foo" ); }
}

class C extends P {
  foo() {
    super();
  }
}

var c1 = new C();
c1.foo(); // "P.foo"

var D = {
  foo: function() { console.log( "D.foo" ); }
};

var E = {
  foo: C.prototype.foo
};

// E链接到D来进行委托
Object.setPrototypeOf( E, D );

E.foo(); // "P.foo"
```

如果你（十分合理地！）认为 `super` 将会在调用时自动绑定，你可能会期望 `super()` 将会自动地认识到 `E` 委托至 `D`，所以使用 `super()` 的 `E.foo()` 应当调用 `D.foo()`。

不是这样。由于实用主义的性能原因，`super` 不像 `this` 那样延迟绑定（也就是动态绑定）。相反它从调用时 `[[HomeObject]].[[Prototype]]` 派生出来，而 `[[HomeObject]]` 是在声明时静态绑定的。

在这个特定的例子中，`super()` 依然解析为 `P.foo()`，因为方法的 `[[HomeObject]]` 仍然是 `C` 而且 `C.[[Prototype]]` 是 `P`。

可能会有方法手动地解决这样的陷阱。在这个场景中使用 `toMethod(..)` 来绑定/重绑定方法的 `[[HomeObject]]`（设置这个对象的 `[[Prototype]]` 一起！）似乎会管用：

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

// E链接到D来进行委托
var E = Object.create( D );

// 手动绑定`foo`的`[[HomeObject]]`到
// `E`，因为`E.[[Prototype]]`是`D`，所以
// `super()`是`D.foo()`
E.foo = C.prototype.foo.toMethod( E, "foo" );

E.foo(); // "D.foo"
```

注意：`toMethod()` 克隆这个方法，然后将它的第一个参数作为 `homeObject`（这就是为什么我们传入 `E`），第二个参数（可选）用来设置新方法的 `name`（保持“`foo`”不变）。

除了这种场景以外，是否还有其他的极端情况会使开发者们落入陷阱还有待观察。无论如何，你将不得不费心保持清醒：在哪里引擎自动为你确定 `super`，和在哪里你不得不手动处理它。噢！

## 静态优于动态？

但是关于ES6的最大问题是，所有这些种种陷阱意味着 `class` 有点儿将你带入一种语法，它看起来暗示着（像传统的类那样）一旦你声明一个 `class`，它是一个东西的静态定义（将来会实例化）。使你完全忘记了这个事实：`c` 是一个对象，一个你可以直接互动的具体的东西。

在传统面向类的语言中，你从不会在晚些时候调整类的定义，所以类设计模式不提供这样的能力。但是JS的一个最强大的部分就是它是动态的，而且任何对象的定义都是（除非你将它设定为不可变）不固定的可变的东西。

`class` 看起来在暗示你不应该做这样的事情，通过强制你使用 `.prototype` 语法才能做到，或强制你考虑 `super` 的陷阱，等等。而且它对这种动态机制可能带来的一切陷阱几乎不提供任何支持。

换句话说，`class` 好像在告诉你：“动态太坏了，所以这可能不是一个好主意。这里有看似静态语法，把你的东西静态编码。”

关于JavaScript的评论是多么悲伤啊：动态太难了，让我们假装成（但实际上不是！）静态吧。

这些就是为什么ES6的`class` 伪装成一个语法头痛症的解决方案，但是它实际上把水搅得更浑，而且不容易对JS形成清晰简明的认识。

注意：如果你使用`.bind(..)` 工具制作一个硬绑定函数（见第二章），那么这个函数是不能像普通函数那样用ES6的`extend` 扩展的。

## 复习

`class` 在假装修复JS中的类/继承设计模式的问题上做的很好。但他实际上做的却正相反：它隐藏了许多问题，而且引入了其他微妙而且危险的东西。

`class` 为折磨了JavaScript语言将近20年的“类”的困扰做出了新的贡献。在某些方面，它问的问题比它解决的多，而且在`[[Prototype]]` 机制的优雅和简单之上，它整体上感觉像是一个非常不自然的匹配。

底线：如果ES6`class` 使稳健地利用`[[Prototype]]` 变得困难，而且隐藏了JS对象机制最重要的性质——对象间的实时委托链接——我们不应该认为`class` 产生的麻烦比它解决的更多，并且将它贬低为一种反模式吗？

我真的不能帮你回答这个问题。但我希望这本书已经在你从未经历过的深度上完全地探索了这个问题，而且已经给出了你自己回答这个问题所需的信息。

# 你不懂JS: *this & Object Prototypes*

## 附录B: 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，和许多其他人。一个巨大感谢送给Nick Berardi为本书作序。

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 Tregoin, 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 Hodsdén, 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 enamorato, 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, tofugi, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes, Rhys Brett-Bowen, Dmitry, Aziz Khouri, 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 Viloslada, 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 Gharegozlu, 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。

# 你不懂JS：类型与文法

## 目录

- 序
- 前言
- 第一章：类型
  - 类型的重要意义
  - 内建类型
  - 值作为类型
- 第二章：值
  - Arrays
  - Strings
  - Numbers
  - 特殊值
  - 值与引用
- 第三章：原生类型
  - 内部 `[[Class]]`
  - 封箱包装器
  - 开箱
  - 原生类型作为构造器
- 第四章：强制转换
  - 转换值
  - 抽象值操作
  - 明确的强制转换
  - 隐含的强制转换
  - 宽松等价与严格等价
  - 抽象关系比较
- 第五章：文法
  - 语句与表达式
  - 操作符优先级
  - 自动分号
  - 错误
  - 函数参数值
  - `try..finally`
  - `switch`
- 附录A：与环境混合的JavaScript

- 附录B: 鸣谢

# 你不懂JS：类型与文法

## 序

人们曾说，“JavaScript是唯一一种开发者在学会之前就使用的语言。”

我每次听到这句话都会笑出来，因为对我来说这是真的，而且我怀疑对于许多其他开发者也是。JavaScript，甚至可能还有CSS和HTML，在因特网出现的早期都不是大学中教授的核心计算机语言，所以个人开发很大程度上都是基于开发者的搜索和“看源代码”的能力来将这些基本的web语言拼凑在一起。

我还记得我的第一个高中网站项目。它的任务是创建任意类型的网上商店，而我作为一个James Bond的粉丝，决定创建一个黄金眼商店。它有一切东西：黄金眼的迷笛主题音乐在背景中播放，一个用JavaScript制作的瞄准十字在屏幕上跟踪着鼠标，还有在每次点击时播放一次枪响的音效。Q本应该会为这个网站中的杰作感到骄傲的。

我说这个故事是因为在那时我确实做了许多开发者今天在做的事情：我在我的项目中拷贝粘贴了大块儿的JavaScript代码，而根本不知道究竟发生了什么。像jQuery那样的工具包的广泛使用，以它们微不足道的方式，延续了这种不学习核心JavaScript的模式。

我不是在贬低JavaScript工具包的使用；毕竟，我还是MooToo，Is JavaScript团队的一员！但是JavaScript工具包如此强大的原因是因为它们的开发者了解基础，和它们的“坑”，并出色地施用了它们。和这些工具包的有用之处一样，了解这门语言的基础依然是极其重要的，而且有了Kyle Simpson的 你不懂JS 系列这样的书，没有借口不学习它们。

类型与文法，这个系列的第三部，是学习核心JavaScript基础的杰出教材，这些基础是拷贝粘贴和JavaScript工具包没有和绝不会教你的。强制转换和它的陷阱，原生类型与构造器，和JavaScript基本的全部方面都使用专门的代码示例进行了彻底地讲解。和这个系列的其他书籍一样，Kyle总是一针见血：没有作秀和文字游戏——这正是我喜爱类型的技术书籍。

享受类型与文法而且不要让它离你的桌子太远！

David Walsh

<http://davidwalsh.name>, [@davidwalshblog](https://twitter.com/davidwalshblog)

高级Web开发者，Mozilla

# 你不懂JS：类型与文法

## 第一章：类型

大多数开发者会说，动态语言（就像JS）没有类型。让我们看看ES5.1语言规范（<http://www.ecma-international.org/ecma-262/5.1/>）在这个问题上是怎么说的：

在本语言规范中的算法所操作的每一个值都有一种关联的类型。可能的值的类型就是那些在本条款中定义的类型。类型还进一步被分为ECMAScript语言类型和语言规范类型

一个ECMAScript语言类型对应于ECMAScript程序员使用ECMAScript语言直接操作的值。ECMAScript语言类型有Undefined，Null，Boolean，String，Number，和Object。

现在，如果你是一个强类型（静态类型的）语言的爱好者，你可能会反对“类型”一词的用法。在那些语言中，“类型”的含义要比它在JS这里的含义丰富得多。

有些人说JS不应该声称拥有“类型”，它们应被称为“标签”或者“子类型”。

去他的！我们将使用这个粗糙的定义（看起来和语言规范的定义相同，只是改变了措辞）：一个类型是一组固有的，内建的性质，对于引擎和开发者来说，它独一无二地标识了一个特定的值的行为，并将它与其他值区分开。

换句话说，如果引擎和开发者看待值 42（数字）与看待值 "42"（字符串）的方式不同，那么这两个值就拥有不同的类型 -- 分别是 number 和 string。当你使用 42 时，你就在试图做一些数字的事情，比如计算。但当你使用 "42" 时，你就在试图做一些字符串的事情，比如输出到页面上，等等。这两个值有着不同的类型。

这绝不是一个完美的定义。但是对于这里的讨论足够好了。而且它与JS描述它的方式并不矛盾。

## 类型的重要意义

抛开学术上关于定义的分歧，为什么JavaScript有或者没有类型那么重要？

对每一种类型和它的固有行为有一个正确的理解，对于理解如何正确和准确地转换两个不同类型的值来说是绝对必要的（参见第四章，强制转换）。几乎每一个被编写过的JS程序都需要以某种形式处理类型的强制转换，所以，你能负责任，有信心地这么做是很重要的。

如果你有一个 number 值 42，但你想像一个 string 那样对待它，比如从位置 1 中将 "2" 作为一个字符抽取出来，那么显然你需要首先将值从 number（强制）转换成一个 string。

这看起来十分简单。

但是这样的强制转换可能以许多不同的方式发生。其中有些方式是明确的，很容易推理的，和可靠的。但是如果你不小心，强制转换就可能以非常奇怪的，令人吃惊的方式发生。

强制转换的困惑可能是JavaScript开发者所经历的最深刻的挫败感之一。它曾经总是因为如此危险而为人所诟病，被认为是一个语言设计上的缺陷而应当被回避。

带着对JavaScript类型的全面理解，我们将要阐明为什么强制转换的坏名声是言过其实的，而且是有些冤枉的 -- 以此来反转你的视角，来看清强制转换的力量和用处。但首先，我们不得不更好地把握值与类型。

## 内建类型

JavaScript定义了7种内建类型：

- `null`
- `undefined`
- `boolean`
- `number`
- `string`
- `object`
- `symbol` -- 在ES6中被加入的！

注意：除了 `object` 所有这些类型都被称为“基本类型（primitives）”。

`typeof` 操作符可以检测给定值的类型，而且总是返回7种字符串值中的一种 -- 令人吃惊的是，对于我们刚刚列出的7中内建类型，它没有一个恰好的一对一匹配。

```
typeof undefined      === "undefined"; // true
typeof true          === "boolean";   // true
typeof 42            === "number";    // true
typeof "42"          === "string";   // true
typeof { life: 42 }  === "object";   // true

// 在ES6中被加入的！
typeof Symbol()     === "symbol";   // true
```

如上所示，这6种列出来的类型拥有相应类型的值，并返回一个与类型名称相同的字符串值。`Symbol` 是ES6的新数据类型，我们将在第三章中讨论它。

正如你可能已经注意到的，我在上面的列表中剔除了 `null`。它是特殊的 -- 特殊在它与 `typeof` 操作符组合时是有bug的。

```
typeof null === "object"; // true
```

要是它返回 "null" 就好了（而且是正确的！），但是这个原有的bug已经存在了近20年，而且好像永远也不会被修复了，因为有太多已经存在的web的内容依存着这个bug的行为，“修复”这个bug将会制造更多的“bug”并毁掉许多web软件。

如果你想要使用 `null` 类型来测试 `null` 值，你需要一个复合条件：

```
var a = null;
(!a && typeof a === "object"); // true
```

`null` 是唯一一个“**falsy**”（也叫类**false**；见第四章），但是在 `typeof` 检查中返回 "object" 的基本类型。

那么 `typeof` 可以返回的第7种字符串值是什么？

```
typeof function a(){ /* .. */ } === "function"; // true
```

很容易认为在JS中 `function` 是一种顶层的内建类型，特别是看到 `typeof` 操作符的这种行为时。然而，如果你阅读语言规范，你会看到它实际上是对象（`object`）的“子类型”。特别地，一个函数（`function`）被称为“可调用对象”——一个拥有 `[[Call]]` 内部属性，允许被调用的对象。

函数实际上是对象的事实十分有用。最重要的是，它们可以拥有属性。例如：

```
function a(b, c) {
  /* .. */
}
```

这个函数对象拥有一个 `length` 属性，它被设置为函数被声明时的正式参数的数量。

```
a.length; // 2
```

因为你使用了两个正式命名的参数（`b` 和 `c`）声明了函数，所以“函数的长度”是 `2`。

那么数组呢？它们是JS原生的，所以它们是一个特殊的类型咯？

```
typeof [1, 2, 3] === "object"; // true
```

不，它们仅仅是对象。考虑它们的最恰当的方法是，也将它们认为是对象的“子类型”（见第三章），带有被数字索引的附加性质（与仅仅使用字符串键的普通对象相反），并维护着一个自动更新的 `.length` 属性。

## 值作为类型

在JavaScript中，变量没有类型 -- 值才有类型。变量可以在任何时候，持有任何值。

另一种考虑JS类型的方式是，JS没有“类型强制”，也就是引擎不坚持认为一个变量总是持有与它开始存在时相同的初始类型的值。在一个赋值语句中，一个变量可以持有一个 `string`，而在下一个赋值语句中持有一个 `number`，如此类推。

值 `42` 有固有的类型 `number`，而且它的类型是不能被改变的。另一个值，比如 `string` 类型的 `"42"`，可以通过一个称为 强制转换 的处理从 `number` 类型的值 `42` 中创建出来（见第四章）。

如果你对一个变量使用 `typeof`，它不会像表面上看起来那样询问“这个变量的类型是什么？”，因为JS变量是没有类型的。取而代之的是，它会询问“在这个变量里的值的类型是什么？”

```
var a = 42;
typeof a; // "number"

a = true;
typeof a; // "boolean"
```

`typeof` 操作符总是返回字符串。所以：

```
typeof typeof 42; // "string"
```

第一个 `typeof 42` 返回 `"number"`，而 `typeof "number"` 是 `"string"`。

## undefined vs "undeclared"

当前还不拥有值的变量，实际上拥有 `undefined` 值。对这样的变量调用 `typeof` 将会返回 `"undefined"`：

```
var a;

typeof a; // "undefined"

var b = 42;
var c;

// 稍后
b = c;

typeof b; // "undefined"
typeof c; // "undefined"
```

大多数开发者考虑“`undefined`”这个词的方式会诱使他们认为它是“`undeclared`（未声明）”的同义词。然而在JS中，这两个概念十分不同。

一个“`undefined`”变量是在可访问的作用域中已经被声明过的，但是在这个时刻它里面没有任何值。相比之下，一个“`undeclared`”变量是在可访问的作用域中还没有被正式声明的。

考虑这段代码：

```
var a;

a; // undefined
b; // ReferenceError: b is not defined
```

一个恼人的困惑是浏览器给这种情形分配的错误消息。正如你所看到的，这个消息是“`b is not defined`”，这当然很容易而且很合理地使人将它与“`b is undefined.`”搞混。需要重申的是，“`undefined`”和“`is not defined`”是非常不同的东西。要是浏览器能告诉我们类似于“`b is not found`”或者“`b is no declared`”之类的东西就好了，那会减少这种困惑！

还有一种 `typeof` 与未声明变量关联的特殊行为，进一步增强了这种困惑。考虑这段代码：

```
var a;

typeof a; // "undefined"

typeof b; // "undefined"
```

`typeof` 操作符甚至为“`undeclared`”（或“`not defined`”）变量返回 “`undefined`” 。要注意的是，当我们执行 `typeof b` 时，即使 `b` 是一个未声明变量，也不会有错误被抛出。这是 `typeof` 的一种特殊的安全防卫行为。

和上面类似地，要是 `typeof` 与未声明变量一起使用时返回“`undeclared`”就好了，而不是将其结果值与不同的“`undefined`”情况混为一谈。

## typeof Undeclared

不管怎样，当在浏览器中处理JavaScript时这种安全防卫是一种有用的特性，因为浏览器中多个脚本文件会将变量加载到共享的全局名称空间。

**注意：**许多开发者相信，在全局名称空间中绝不应该有任何变量，而且所有东西应当被包含在模块和私有/隔离的名称空间中。这在理论上很伟大但在实践中几乎是不可能的；但它仍然是一个值得的努力方向！幸运的是，ES6为模块加入了头等支持，这终于使这一理论变得可行的多。

作为一个简单的例子，想象在你的程序中有一个“调试模式”，它是通过一个称为 `DEBUG` 的全局变量（标志）来控制的。在实施类似于在控制台上输出一条日志消息这样的调试任务之前，你想要检查这个变量是否被声明了。一个顶层的全局 `var DEBUG = true` 声明只包含在一个“`debug.js`”文件中，这个文件仅在你开发/测试时才被加载到浏览器中，而在生产环境中则不会。

然而，在你其他的程序代码中，你不得不小心你是如何检查这个全局的 `DEBUG` 变量的，这样你才不会抛出一个 `ReferenceError`。这种情况下 `typeof` 上的安全防卫就是我们的朋友。

```
// 噢，这将抛出一个错误！
if (DEBUG) {
    console.log( "Debugging is starting" );
}

// 这是一个安全的存在性检查
if (typeof DEBUG !== "undefined") {
    console.log( "Debugging is starting" );
}
```

即便你不是在对付用户定义的变量（比如 `DEBUG`），这种检查也是很有用的。如果你为一个内建的API做特性检查，你也会发现不带有抛出错误的检查很有帮助：

```
if (typeof atob === "undefined") {
    atob = function() { /*..*/ };
}
```

**注意：**如果你在为一个还不存在的特性定义一个“填补”，你可能想要避免使用 `var` 来声明 `atob`。如果你在 `if` 语句内部声明 `var atob`，即使这个 `if` 条件没有通过（因为全局的 `atob` 已经存在），这个声明也会被提升（参见本系列的作用域与闭包）到作用域的顶端。在某些浏览器中，对一些特殊类型的内建全局变量（常被称为“宿主对象”），这种重复声明也许会抛出错误。忽略 `var` 可以防止这种提升声明。

另一种不带有 `typeof` 的安全防卫特性，而对全局变量进行这些检查的方法是，将所有的全局变量作为全局对象的属性来观察，在浏览器中这个全局对象基本上是 `window` 对象。所以，上面的检查可以（十分安全地）这样做：

```
if (window.DEBUG) {
    // ...
}

if (!window.atob) {
    // ...
}
```

和引用未声明变量不同的是，在你试着访问一个不存在的对象属性时（即便是在全局的 `window` 对象上），不会有 `ReferenceError` 被抛出。

另一方面，一些开发者偏好避免手动使用 `window` 引用全局变量，特别是当你的代码需要运行在多种JS环境中时（例如不仅是在浏览器中，还在服务器端的`node.js`中），全局变量可能不总是称为 `window`。

技术上讲，这种 `typeof` 上的安全防卫即使在你不使用全局变量时也很有用，虽然这些情况不那么常见，而且一些开发者也许发现这种设计方式不那么理想。想象一个你想要其他人复制-粘贴到他们程序中或模块中的工具函数，在它里面你想要检查包含它的程序是否已经定义了一个特定的变量（以便于你可以使用它）：

```
function doSomethingCool() {
    var helper =
        (typeof FeatureXYZ !== "undefined") ?
        FeatureXYZ :
        function() { /*.. 默认的特性 ..*/ };

    var val = helper();
    // ...
}
```

`doSomethingCool()` 对称为 `FeatureXYZ` 变量进行检查，如果找到，就使用它，如果没找到，使用它自己的。现在，如果某个人在他的模块/程序中引入了这个工具，它会安全地检查我们是否已经定义了 `FeatureXYZ`：

```
// 一个IIFE（参见本系列的“作用域与闭包”中的“立即被调用的函数表达式”）
(function(){
    function FeatureXYZ() { /*... my XYZ feature ...*/ }

    // 引入 `doSomethingCool(..)`
    function doSomethingCool() {
        var helper =
            (typeof FeatureXYZ !== "undefined") ?
            FeatureXYZ :
            function() { /*... 默认的特性 ...*/ };

        var val = helper();
        // ...
    }

    doSomethingCool();
})();
```

这里，`FeatureXYZ` 根本不是一个全局变量，但我们仍然使用 `typeof` 的安全防卫来使检查变得安全。而且重要的是，我们在这里没有可以用于检查的对象（就像我们使用 `window.___` 对全局变量做的那样），所以 `typeof` 十分有帮助。

另一些开发者偏好一种称为“依赖注入”的设计模式，与 `doSomethingCool()` 隐含地检查 `FeatureXYZ` 是否在它外部/周围被定义过不同的是，它需要依赖明确地传递进来，就像这样：

```
function doSomethingCool(FeatureXYZ) {
    var helper = FeatureXYZ ||
        function() { /*... 默认的特性 ...*/ };

    var val = helper();
    // ...
}
```

在设计这样的功能时有许多选择。这些模式里没有“正确”或“错误”——每种方式都有各种权衡。但总的来说，`typeof` 的未声明安全防卫给了我们更多选项，这还是很不错的。

## 复习

JavaScript有7种内建类

型：`null`，`undefined`，`boolean`，`number`，`string`，`object`，`symbol`。它们可以被 `typeof` 操作符识别。

变量没有类型，但是值有类型。这些类型定义了值的固有行为。

许多开发者会认为“`undefined`”和“`undeclared`”大体上是同一个东西，但是在JavaScript中，它们是十分不同的。`undefined` 是一个可以由被声明的变量持有的值。“未声明”意味着一个变量从来没有被声明过。

JavaScript很不幸地将这两个词在某种程度上混为了一谈，不仅体现在它的错误消息上（“`ReferenceError: a is not defined`”），也体现在 `typeof` 的返回值上：对于两者它都返回 `"undefined"`。

然而，当对一个未声明的变量使用 `typeof` 时，`typeof` 上的安全防卫机制（防止一个错误）可以在特定的情况下非常有用。

# 你不懂JS：类型与文法

## 第二章：值

`array`，`string`，和`number`是任何程序的最基础构建块，但是JavaScript在这些类型上有一些要么使你惊喜要么使你惊讶的独特性质。

让我们来看几种JS内建的值类型，并探讨一下我们如何才能更加全面地理解并正确地利用它们的行为。

### Array

和其他强制类型的语言相比，JavaScript的`array`只是值的容器，而这些值可以是任何类型：`string`或者`number`或者`object`，甚至是另一个`array`（这也是你得到多维数组的方法）。

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

a.length;           // 3
a[0] === 1;         // true
a[2][0] === 3;     // true
```

你不需要预先指定`array`的大小，你可以仅声明它们并加入你觉得合适的值：

```
var a = [ ];

a.length;           // 0

a[0] = 1;
a[1] = "2";
a[2] = [ 3 ];

a.length;           // 3
```

警告：在一个`array`值上使用`delete`将会从这个`array`上移除一个值槽，但就算你移除了最后一个元素，它也不会更新`length`属性，所以多加小心！我们会在第五章讨论`delete`操作符的更多细节。

要小心创建“稀散”的`array`（留下或创建空的/丢失的值槽）：

```
var a = [ ];
a[0] = 1;
// 这里没有设置值槽`a[1]`、
a[2] = [ 3 ];

a[1];      // undefined
a.length;   // 3
```

虽然这可以工作，但你留下的“空值槽”可能会导致一些令人困惑的行为。虽然这样的值槽看起来拥有 `undefined` 值，但是它不会像被明确设置（`a[1] = undefined`）的值槽那样动作。更多信息可以参见第三章的“Array”。

`array` 是被数字索引的（正如你认为的那样），但微妙的是它们也是对象，可以在它们上面添加 `string` 键/属性（但是这些属性不会计算在 `array` 的 `length` 中）：

```
var a = [ ];
a[0] = 1;
a["foobar"] = 2;

a.length;      // 1
a["foobar"];   // 2
a.foobar;      // 2
```

然而，一个需要小心的坑是，如果一个可以被强制转换为10进制 `number` 的 `string` 值被用作键的话，它会认为你想使用 `number` 索引而不是一个 `string` 键！

```
var a = [ ];
a["13"] = 42;

a.length; // 14
```

一般来说，向 `array` 添加 `string` 键/属性不是一个好主意。最好使用 `object` 来持有键/属性形式的值，而将 `array` 专用于严格地数字索引的值。

## 类Array

偶尔你需要将一个类 `array` 值（一个数字索引的值的集合）转换为一个真正的 `array`，通常你可以对这些值的集合调用数组的工具函数（比如 `indexOf(..)`，`concat(..)`，`forEach(..)` 等等）。

举个例子，各种DOM查询操作会返回一个DOM元素的列表，对于我们转换的目的来说，这些列表不是真正的`array`但是也足够类似`array`。另一个常见的例子是，函数为了像列表一样访问它的参数值，而暴露了`arugumens`对象（类`array`，在ES6中被废弃了）。

一个进行这种转换的很常见的方法是对这个值借用`slice(..)`工具：

```
function foo() {
  var arr = Array.prototype.slice.call( arguments );
  arr.push( "bam" );
  console.log( arr );
}

foo( "bar", "baz" ); // ["bar", "baz", "bam"]
```

如果`slice()`没有用其他额外的参数调用，就像上面的代码段那样，它的参数的默认值会使它具有复制这个`array`（或者，在这个例子中，是一个类`array`）的效果。

在ES6中，还有一种称为`Array.from(..)`的内建工具可以执行相同任务：

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

注意：`Array.from(..)`拥有几种其他强大的能力，我们将在本系列的ES6与未来中涵盖它的细节。

## String

一个很常见的想法是，`string`实质上只是字符的`array`。虽然内部的实现可能是也可能不是`array`，但重要的是要理解JavaScript的`string`与字符的`array`确实不一样。它们的相似性几乎只是表面上的。

举个例子，让我们考虑这两个值：

```
var a = "foo";
var b = ["f", "o", "o"];
```

`String`确实与`array`有很肤浅的相似性 -- 也就是上面说的，类`array`-- 举例来说，它们都有一个`length`属性，一个`indexof(..)`方法（在ES5中仅有`array`版本），和一个`concat(..)`方法：

```

a.length;                      // 3
b.length;                      // 3

a.indexOf( "o" );              // 1
b.indexOf( "o" );              // 1

var c = a.concat( "bar" );      // "foobar"
var d = b.concat( [ "b", "a", "r" ] ); // [ "f", "o", "o", "b", "a", "r" ]

a === c;                       // false
b === d;                       // false

a;                             // "foo"
b;                             // [ "f", "o", "o" ]

```

那么，它们基本上都仅仅是“字符的数组”，对吧？不确切：

```

a[1] = "O";
b[1] = "O";

a; // "foo"
b; // [ "f", "O", "o" ]

```

JavaScript 的 `string` 是不可变的，而 `array` 是相当可变的。另外，在 JavaScript 中用位置访问字符的 `a[1]` 形式不总是广泛合法的。老版本的 IE 就不允许这种语法（但是它们现在允许了）。相反，正确的方式是 `a.charAt(1)`。

`string` 不可变性的进一步的后果是，`string` 上没有一个方法是可以原地修改它的内容的，而是创建并返回一个新的 `string`。与之相对的是，许多改变 `array` 内容的方法实际上是原地修改的。

```

c = a.toUpperCase();
a === c;    // false
a;          // "foo"
c;          // "FOO"

b.push( "!" );
b;          // [ "f", "o", "o", "!" ]

```

另外，许多 `array` 方法在处理 `string` 时非常有用，虽然这些方法不属于 `string`，但我们可以对我们的 `string` “借用”非变化的 `array` 方法：

```

a.join;           // undefined
a.map;           // undefined

var c = Array.prototype.join.call( a, "-" );
var d = Array.prototype.map.call( a, function(v){
    return v.toUpperCase() + ".";
} ).join( "" );

c;               // "f-o-o"
d;               // "F.O.O."

```

让我们来看另一个例子：翻转一个 `string`（顺带一提，这是一个JavaScript面试中常见的细小问题！）。`array` 拥有一个原地的 `reverse()` 修改器方法，但是 `string` 没有：

```

a.reverse;       // undefined

b.reverse();     // ["!", "o", "O", "f"]
b;              // ["!", "o", "O", "f"]

```

不幸的是，这种“借用”`array` 修改器不起作用，因为 `string` 是不可变的，因此它不能被原地修改：

```

Array.prototype.reverse.call( a );
// 仍然返回一个“foo”的String对象包装器（见第三章）：(

```

另一种迂回的做法（也是黑科技）是，将 `string` 转换为一个 `array`，实施我们想做的操作，然后将它转回 `string`。

```

var c = a
// 将`a`切分成一个字符的数组
.split( "" )
// 翻转字符的数组
.reverse()
// 将字符的数组连接回一个字符串
.join( "" );

c; // "oof"

```

如果你觉得这很难看，没错。不管怎样，对于简单的 `string` 它好用，所以如果你需要某些快速但是“脏”的东西，像这样的方式经常能满足你。

警告：小心！这种方法对含有复杂（unicode）字符（星号，多字节字符等）的 `string` 不起作用。你需要支持unicode的更精巧的工具库来准确地处理这种操作。在这个问题上可以咨询Mathias Bynens的作品：*Esrever* (<https://github.com/mathiasbynens/esrever>)。

另外一种考虑这个问题的方式是：如果你更经常地将你的“string”基本上作为字符的数组来执行一些任务的话，也许就将它们作为 array 而不是作为 string 存储更好。你可能会因此省去很多每次都将 string 转换为 array 的麻烦。无论何时你确实需要 string 的表现形式的话，你总是可以调用字符的 array 的 `join("")` 方法。

## Number

JavaScript只有一种数字类型：`number`。这种类型包含“整数”值和小数值。我说“整数”时加了引号，因为JS的一个长久以来为人诟病的原因是，和其他语言不同，JS没有真正的整数。这可能在未来某个时候会改变，但是目前，我们只有 `number` 可用。

所以，在JS中，一个“整数”只是一个没有小数部分的小数值。也就是说，`42.0` 和 `42` 一样是“整数”。

像大多数现代计算机语言，以及几乎所有的脚本语言一样，JavaScript的 `number` 的实现基于“IEEE 754”标准，通常被称为“浮点”。JavaScript明确地使用了这个标准的“双精度”（也就是“64位二进制”）格式。

在网络上有许多了不起的文章都在介绍二进制浮点数如何在内存中存储的细节，以及选择这些做法的意义。因为对于理解如何在JS中正确使用 `number` 来说，理解内存中的位模式不是必须的，所以我们将这个话题作为练习留给那些想要进一步挖掘IEEE 754的细节的读者。

### 数字的语法

在JavaScript中字面数字一般用10进制小数表达。例如：

```
var a = 42;
var b = 42.3;
```

小数的整数部分如果是 `0`，是可选的：

```
var a = 0.42;
var b = .42;
```

相似地，一个小数在 `.` 之后的小数部分如果是 `0`，是可选的：

```
var a = 42.0;
var b = 42.;
```

警告：`42.` 是极不常见的，如果你正在努力避免别人阅读你的代码时感到困惑，它可能不是一个好主意。但不管怎样，它是合法的。

默认情况下，大多数 `number` 将会以10进制小数的形式输出，并去掉末尾小数部分的 `0`。所以：

```
var a = 42.300;
var b = 42.0;

a; // 42.3
b; // 42
```

非常大或非常小的 `number` 将默认以指数形式输出，与 `toExponential()` 方法的输出一样，比如：

```
var a = 5E10;
a; // 50000000000
a.toExponential(); // "5e+10"

var b = a * a;
b; // 2.5e+21

var c = 1 / a;
c; // 2e-11
```

因为 `number` 值可以用 `Number` 对象包装器封装（见第三章），`number` 值可以访问内建在 `Number.prototype` 上的方法（见第三章）。举个例子，`toFixed(..)` 方法允许你指定一个值在被表示时，带有多少位小数：

```
var a = 42.59;

a.toFixed( 0 ); // "43"
a.toFixed( 1 ); // "42.6"
a.toFixed( 2 ); // "42.59"
a.toFixed( 3 ); // "42.590"
a.toFixed( 4 ); // "42.5900"
```

要注意的是，它的输出实际上是一个 `number` 的 `string` 表现形式，而且如果你指定的位数多于值持有的小数位数时，会在右侧补 `0`。

`toPrecision(..)` 很相似，但它指定的是有多少有效数字用来表示这个值：

```
var a = 42.59;

a.toPrecision( 1 ); // "4e+1"
a.toPrecision( 2 ); // "43"
a.toPrecision( 3 ); // "42.6"
a.toPrecision( 4 ); // "42.59"
a.toPrecision( 5 ); // "42.590"
a.toPrecision( 6 ); // "42.5900"
```

你不必非得使用持有这个值的变量来访问这些方法；你可以直接在 `number` 的字面上访问这些方法。但你不得不小心 `.` 操作符。因为 `.` 是一个合法数字字符，如果可能的话，它会首先被翻译为 `number` 字面的一部分，而不是被翻译为属性访问操作符。

```
// 不合法的语法：
42.toFixed( 3 ); // SyntaxError

// 这些都是合法的：
(42).toFixed( 3 ); // "42.000"
0.42.toFixed( 3 ); // "0.420"
42..toFixed( 3 ); // "42.000"
```

`42.toFixed(3)` 是不合法的语法，因为 `.` 作为 `42.` 字面（这是合法的 -- 参见上面的讨论！）的一部分被吞掉了，因此没有 `.` 属性操作符来表示 `.toFixed` 访问。

`42..toFixed(3)` 可以工作，因为第一个 `.` 是 `number` 的一部分，而第二个 `.` 是属性操作符。但它可能看起来很古怪，而且确实在实际的JavaScript代码中很少会看到这样的东西。实际上，在任何基本类型上直接访问方法是十分不常见的。但是不常见并不意味着坏或者错。

注意：有一些库扩展了内建的 `Number.prototype`（见第三章），使用 `number` 或在 `number` 上提供了额外的操作，所以在这些情况下，像使用 `10..makeItRain()` 来设定一个10秒钟的下钱雨的动画，或者其他诸如此类的傻事是完全合法的。

在技术上讲，这也是合法的（注意那个空格）：

```
42 .toFixed(3); // "42.000"
```

但是，尤其是对 `number` 字面量来说，这是特别使人糊涂的代码风格，而且除了使其他开发者（和未来的你）糊涂以外没有任何用处。避免它。

`number` 还可以使用科学计数法的形式指定，这在表示很大的 `number` 时很常见，比如：

```
var onethousand = 1E3; // 代表 1 * 10^3
var onemilliononehundredthousand = 1.1E6; // 代表 1.1 * 10^6
```

`number` 字面量还可以使用其他进制表达，比如二进制，八进制，和十六进制。

这些格式是可以在当前版本的JavaScript中使用的：

```
0xf3; // 十六进制的: 243
0Xf3; // 同上
```

```
0363; // 八进制的: 243
```

注意：从ES6 + `strict` 模式开始，不再允许 `0363` 这样的的八进制形式（新的形式参见后面的讨论）。`0363` 在非 `strict` 模式下依然是允许的，但是不管怎样你应当停止使用它，来拥抱未来（而且因为你现在应当在使用 `strict` 模式了！）。

至于ES6，下面的新形式也是合法的：

```
00363; // 八进制的: 243
00363; // 同上
```

```
0b11110011; // 二进制的: 243
0B11110011; // 同上
```

请为你的开发者同胞们做件好事：绝不要使用 `00363` 形式。把 `0` 放在大写的 `o` 旁边就是在制造困惑。保持使用小写的谓词 `0x`，`0b`，和 `0o`。

## 小数值

使用二进制浮点数的最出名（臭名昭著）的副作用是（记住，这是对所有使用IEEE 754的语言都成立的——不是许多人认为/假装仅在JavaScript中存在的问题）：

```
0.1 + 0.2 === 0.3; // false
```

从数学的意义上，我们知道这个语句应当为 `true`。为什么它是 `false`？

简单地说，`0.1` 和 `0.2` 的二进制表示形式是不精确的，所以它们相加时，结果不是精确地 `0.3`。而是非常接近的值：`0.30000000000000004`，但是如果你的比较失败了，“接近”是无关紧要的。

注意：JavaScript应当切换到可以精确表达所有值的一个不同的 `number` 实现吗？有些人认为应该。多年以来有许多选项出现过。但是没有一个被采纳，而且也许永远也不会。它看起来就像挥挥手然后说“已经改好那个bug了！”那么简单，但根本不是那么回事儿。如果真有这么简单，它绝对就在很久以前被改掉了。

现在的问题是，如果一些 `number` 不能被信任为精确的，这不是意味着我们根本不能使用 `number` 吗？当然不是。

在一些应用程序中你需要多加小心，特别是在对付小数的时候。还有许多（也许是大多数？）应用程序只处理整数，而且，最大只处理到几百万到几万亿。这些应用程序使用JS中的数字操作是，而且将总是，非常安全的。

要是我们确实需要比较两个 `number`，就像 `0.1 + 0.2` 与 `0.3`，而且知道这个简单的相等测试将会失败呢？

可以接受的最常见的做法是使用一个很小的“错误舍入”值作为比较的容差。这个很小的值经常被称为“机械极小值（`machine epsilon`）”，对于JavaScript来说这种 `number` 通常为 `2^-52` (`2.220446049250313e-16`)。

在ES6中，使用这个容差值预定义了 `Number.EPSILON`，所以你将会想要使用它，你也可以在前ES6中安全地填补这个定义：

```
if (!Number.EPSILON) {
    Number.EPSILON = Math.pow(2, -52);
}
```

我们可以使用这个 `Number.EPSILON` 来比较两个 `number` 的“等价性”（带有错误舍入的容差）：

```
function numbersCloseEnoughToEqual(n1, n2) {
    return Math.abs( n1 - n2 ) < Number.EPSILON;
}

var a = 0.1 + 0.2;
var b = 0.3;

numbersCloseEnoughToEqual( a, b );           // true
numbersCloseEnoughToEqual( 0.0000001, 0.0000002 ); // false
```

可以被表示的最大的浮点值大概是 `1.798e+308`（它真的非常，非常，非常大！），它为你预定义为 `Number.MAX_VALUE`。在极小的一端，`Number.MIN_VALUE` 大概是 `5e-324`，它不是负数但是非常接近于0！

## 安全整数范围

由于 `number` 的表示方式，对完全是 `number` 的“整数”而言有一个“安全”的值的范围，而且它要比 `Number.MAX_VALUE` 小得多。

可以“安全地”被表示的最大整数（也就是说，可以保证被表示的值是实际可以无误地表示的）是 `2^53 - 1`，也就是 `9007199254740991`，如果你插入一些数字分隔符，可以看到它刚好超过9万亿。所以对于 `number` 能表示的上限来说它确实是够TM大的。

在ES6中这个值实际上是由自动预定义的，它是 `Number.MAX_SAFE_INTEGER`。意料之中的是，还有一个最小值，`-9007199254740991`，它在ES6中定义为 `Number.MIN_SAFE_INTEGER`。

JS程序面临处理这样大的数字的主要情况是，处理数据库中的64位ID等等。64位数字不能使用 `number` 类型准确表达，所以在JavaScript中必须使用 `string` 表现形式存储（和传递）。

谢天谢地，在这样的大ID `number` 值上的数字操作（除了比较，它使用 `string` 也没问题）并不很常见。但是如果你确实需要在这些非常大的值上实施数学操作，目前来讲你需要使用一个大数字工具。在未来版本的JavaScript中，大数字也许会得到官方支持。

## 测试整数

测试一个值是否是整数，你可以使用ES6定义的 `Number.isInteger(..)`：

```
Number.isInteger( 42 );           // true
Number.isInteger( 42.000 );       // true
Number.isInteger( 42.3 );         // false
```

可以为前ES6填补 `Number.isInteger(..)`：

```
if (!Number.isInteger) {
    Number.isInteger = function(num) {
        return typeof num == "number" && num % 1 == 0;
    };
}
```

要测试一个值是否是安全整数，使用ES6定义的 `Number.isSafeInteger(..)`：

```
Number.isSafeInteger( Number.MAX_SAFE_INTEGER );      // true
Number.isSafeInteger( Math.pow( 2, 53 ) );            // false
Number.isSafeInteger( Math.pow( 2, 53 ) - 1 );         // true
```

可以为前ES6浏览器填补 `Number.isSafeInteger(..)`：

```
if (!Number.isSafeInteger) {
    Number.isSafeInteger = function(num) {
        return Number.isInteger( num ) &&
            Math.abs( num ) <= Number.MAX_SAFE_INTEGER;
    };
}
```

## 32位（有符号）整数

虽然整数可以安全地最大达到约9万亿（53比特），但有一些数字操作（比如位操作符）是仅仅为32位 `number` 定义的，所以对于被这样使用的 `number` 来说，“安全范围”一定会小得多。

这个范围是从 `Math.pow(-2, 31)` ( -2147483648 , 大约-21亿)  
到 `Math.pow(2, 31)-1` ( 2147483647 , 大约+21亿) 。

要强制 `a` 中的 `number` 值是32位有符号整数，使用 `a | 0` 。这可以工作是因为 | 位操作符仅仅对32位值起作用（意味着它可以只关注32位，而其他的位将被丢掉）。而且，和0进行“或”的位操作实质上是什么也不做。

注意：特定的特殊值（我们将在下一节讨论），比如 `Nan` 和 `Infinity` 不是“32位安全”的，当这些值被传入位操作符时将会通过一个抽象操作 `ToInt32` （见第四章）并为了位操作而简单地变成 `+0` 值。

## 特殊值

在各种类型中散布着一些特殊值，需要警惕的JS开发者小心，并正确使用。

### 不是值的值

对于 `undefined` 类型来说，有且仅有一个值：`undefined` 。对于 `null` 类型来说，有且仅有一个值：`null` 。所以对它们而言，这些文字既是它们的类型也是它们的值。

`undefined` 和 `null` 作为“空”值或者“没有”值，经常被认为是可以互换的。另一些开发者偏好于使用微妙的区别将它们区分开。举例来讲：

- `null` 是一个空值
- `undefined` 是一个丢失的值

或者：

- `undefined` 还没有值
- `null` 曾经有过值但现在没有

不管你选择如何“定义”和使用这两个值，`null` 是一个特殊的关键字，不是一个标识符，因此你不能将它作为一个变量对待来给它赋值（为什么你要给它赋值呢？！）。然而，`undefined` （不幸地）是一个标识符。噢。

## Undefined

在非 `strict` 模式下，给在全局上提供的 `undefined` 标识符赋一个值实际上是可能的（虽然这是一个非常不好的做法！）：

```
function foo() {
    undefined = 2; // 非常差劲儿的主意!
}

foo();
```

```
function foo() {
    "use strict";
    undefined = 2; // TypeError!
}

foo();
```

但是，在非 `strict` 模式和 `strict` 模式下，你可以创建一个名叫 `undefined` 局部变量。但这又是一个很差劲儿的主意！

```
function foo() {
    "use strict";
    var undefined = 2;
    console.log( undefined ); // 2
}

foo();
```

朋友永远不让朋友覆盖 `undefined`。

## void 操作符

虽然 `undefined` 是一个持有内建的值 `undefined` 的内建标识符（除非被修改——见上面的讨论！），另一个得到这个值的方法是 `void` 操作符。

表达式 `void` 会“躲开”任何值，所以这个表达式的结果总是值 `undefined`。它不会修改任何已经存在的值；只是确保不会有值从操作符表达式中返回来。

```
var a = 42;

console.log( void a, a ); // undefined 42
```

从惯例上讲（大约是从C语言编程中发展而来），要通过使用 `void` 来独立表现值 `undefined`，你可以使用 `void 0`（虽然，很明显，`void true` 或者任何其他的 `void` 表达式都做同样的事情）。在 `void 0`，`void 1` 和 `undefined` 之间没有实际上的区别。

但是在几种其他的环境下 `void` 操作符可以十分有用：如果你需要确保一个表达式没有结果值（即便它有副作用）。

举个例子：

```
function doSomething() {
    // 注意：`APP.ready`是由我们的应用程序提供的
    if (!APP.ready) {
        // 稍后再试一次
        return void setTimeout( doSomething, 100 );
    }

    var result;

    // 做其他一些事情
    return result;
}

// 我们能立即执行吗？
if (doSomething()) {
    // 马上处理其他任务
}
```

这里，`setTimeout(..)` 函数返回一个数字值（时间间隔定时器的唯一标识符，用于取消它自己），但是我们想 `void` 它，这样我们函数的返回值不会在 `if` 语句上给出一个成立的误报。

许多开发者宁愿将这些动作分开，这样的效用相同但不使用 `void` 操作符：

```
if (!APP.ready) {
    // 稍后再试一次
    setTimeout( doSomething, 100 );
    return;
}
```

一般来说，如果有那么一个地方，有一个值存在（来自某个表达式）而你发现这个值如果是 `undefined` 才有用，就使用 `void` 操作符。这可能在你的程序中不是非常常见，但如果在一些稀有的情况下你需要它，它就十分有用。

## 特殊的数字

`number` 类型包含几种特殊值。我们将会仔细考察每一种。

### 不是数字的数字

如果你不使用同为 `number`（或者可以被翻译为 10 进制或 16 进制的普通 `number` 的值）的两个操作数进行任何算数操作，那么操作的结果将失败而产生一个不合法的 `number`，在这种情况下你将得到 `NaN` 值。

`NaN` 在字面上代表“不是一个 `number` (Not a Number)”，但是正如我们即将看到的，这种文字描述十分失败而且容易误导人。将 `NaN` 考虑为“不合法数字”，“失败的数字”，甚至是“坏掉的数字”都要比“不是一个数字”准确得多。

举例来说：

```
var a = 2 / "foo";           // NaN

typeof a === "number";      // true
```

换句话说：“不是一个数字”的类型是‘数字’！为这使人糊涂的名字和语义欢呼吧。

`NaN` 是一种“哨兵值”（一个被赋予了特殊意义的普通的值），它代表 `number` 集合内的一种特殊的错误情况。这种错误情况实质上是：“我试着进行数学操作但是失败了，而这就是失败的 `number` 结果。”

那么，如果你有一个值存在某个变量中，而且你想要测试它是否是这个特殊的失败数字 `NaN`，你也许认为你可以直接将它与 `NaN` 本身比较，就像你能对其它的值做的那样，比如 `null` 和 `undefined`。不是这样。

```
var a = 2 / "foo";

a == NaN;    // false
a === NaN;   // false
```

`NaN` 是一个非常特殊的值，它从来不会等于另一个 `NaN` 值（也就是，它从来不等于它自己）。实际上，它是唯一一个不具有反射性的值（没有恒等性 `x === x`）。所以，`NaN !== NaN`。有点奇怪，对吧？

那么，如果不能与 `NaN` 进行比较（因为这种比较将总是失败），我们该如何测试它呢？

```
var a = 2 / "foo";

isNaN( a ); // true
```

够简单的吧？我们使用称为 `isNaN(..)` 的内建全局工具，它告诉我们这个值是否是 `NaN`。问题解决了！

别高兴得太早。

`isNaN(..)` 工具有一个重大缺陷。它似乎过于按照字面的意思（“不是一个数字”）去理解 `NaN` 的含义了——它的工作基本上是：“测试这个传进来的值是否不是一个 `number` 或者是一个 `number`”。但这不是十分准确。

```

var a = 2 / "foo";
var b = "foo";

a; // NaN
b; // "foo"

window.isNaN( a ); // true
window.isNaN( b ); // true -- 噢!

```

很明显，"foo" 根本不是一个 `number`，但它也绝不是一个 `Nan` 值！这个bug从最开始的时候就存在于JS中了（存在超过19年的坑）。

在ES6中，终于提供了一个替代它的工具：`Number.isNaN(..)`。有一个简单的填补，可以让你即使是在前ES6的浏览器中安全地检查 `Nan` 值：

```

if (!Number.isNaN) {
    Number.isNaN = function(n) {
        return (
            typeof n === "number" &&
            window.isNaN( n )
        );
    };
}

var a = 2 / "foo";
var b = "foo";

Number.isNaN( a ); // true
Number.isNaN( b ); // false -- 哟!

```

实际上，通过利用 `Nan` 与它自己不相等这个特殊的事，我们可以更简单地实现 `Number.isNaN(..)` 的填补。在整个语言中 `Nan` 是唯一一个这样的值；其他的值都总是等于它自己。

所以：

```

if (!Number.isNaN) {
    Number.isNaN = function(n) {
        return n !== n;
    };
}

```

怪吧？但是好用！

不管有意还是无意，在许多真实世界的JS程序中 `Nan` 可能是一个现实的问题。使用 `Number.isNaN(...)`（或者它的填补）这样的可靠测试来正确地识别它们是一个非常好的主意。

如果你正在程序中仅使用 `isNaN(...)`，悲惨的现实是你的程序有`bug`，即便是你还没有被它咬到！

## 无穷

来自于像C这样的传统编译型语言的开发者，可能习惯于看到编译器错误或者是运行时异常，比如对这样一个操作给出的“除数为0”：

```
var a = 1 / 0;
```

然而在JS中，这个操作是明确定义的，而且它的结果是值 `Infinity`（也就是 `Number.POSITIVE_INFINITY`）。意料之中的是：

```
var a = 1 / 0;    // Infinity
var b = -1 / 0;   // -Infinity
```

如你所见，`-Infinity`（也就是 `Number.NEGATIVE_INFINITY`）是从任一个被除数为负（不是两个都是负数！）的除0操作得来的。

JS使用有限的数字表现形式（IEEE 754 浮点，我们早先讨论过），所以和单纯的数学相比，它看起来甚至在做加法和减法这样的操作时都有可能溢出，这样的情况下你将会得到 `Infinity` 或 `-Infinity`。

例如：

```
var a = Number.MAX_VALUE;    // 1.7976931348623157e+308
a + a;                      // Infinity
a + Math.pow( 2, 970 );     // Infinity
a + Math.pow( 2, 969 );     // 1.7976931348623157e+308
```

根据语言规范，如果一个像加法这样的操作得到一个太大而不能表示的值，IEEE 754“就近舍入”模式将会指明结果应该是什么。所以粗略的意义上，`Number.MAX_VALUE + Math.pow( 2, 969 )` 比起 `Infinity` 更接近于 `Number.MAX_VALUE`，所以它“向下舍入”，而 `Number.MAX_VALUE + Math.pow( 2, 970 )` 距离 `Infinity` 更近，所以它“向上舍入”。

如果你对此考虑的太多，它会使你头疼的。所以别想了。我是认真的，停！

一旦你溢出了任意一个无限值，那么，就没有回头路了。换句最有诗意的话说，你可以从有限迈向无限，但不能从无限回归有限。

“无限除以无限等于什么”，这简直是一个哲学问题。我们幼稚的大脑可能会说“1”或“无限”。事实表明它们都不对。在数学上和在JavaScript中，`Infinity / Infinity` 不是一个有定义的操作。在JS中，它的结果为 `Nan`。

一个有限的正 `number` 除以 `Infinity` 呢？简单！`0`。那一个有限的负 `number` 处理 `Infinity` 呢？接着往下读！

## 零

虽然这可能使有数学头脑的读者困惑，JavaScript拥有普通的零 `0`（也称为正零 `+0`）和一个负零 `-0`。在我们讲解为什么 `-0` 存在之前，我们应该考察JS如何处理它，因为它可能十分令人困惑。

除了使用字面量 `-0` 指定，负的零还可以从特定的数学操作中得出。比如：

```
var a = 0 / -3; // -0
var b = 0 * -3; // -0
```

加法和减法无法得出负零。

在开发者控制台中考察一个负的零，经常显示为 `-0`，然而直到最近这才是一个常见情况，所以一些你可能遇到的老版本浏览器也许依然将它报告为 `0`。

但是根据语言规范，如果你试着将一个负零转换为字符串，它将总会被报告为 `"0"`。

```
var a = 0 / -3;

// 至少（有些浏览器）控制台是对的
a; // -0

// 但是语言规范坚持要向你撒谎！
a.toString(); // "0"
a + ""; // "0"
String(a); // "0"

// 奇怪的是，就连JSON也加入了骗局之中
JSON.stringify(a); // "0"
```

有趣的是，反向操作（从 `string` 到 `number`）不会撒谎：

```
+"-0"; // -0
Number("-0"); // -0
JSON.parse("-0"); // -0
```

警告：当你观察的时候，`JSON.stringify( -0 )` 产生 `"0"` 显得特别奇怪，因为它与反向操作不符：`JSON.parse( "-0" )` 将像你期望地那样报告 `-0`。

除了一个负零的字符串化会欺骗性地隐藏它实际的值外，比较操作符也被设定为（有意地）要说谎。

```
var a = 0;
var b = 0 / -3;

a == b;          // true
-0 == 0;         // true

a === b;         // true
-0 === 0;        // true

0 > -0;          // false
a > b;           // false
```

很明显，如果你想在你的代码中区分 `-0` 和 `0`，你就不能仅依靠开发者控制台的输出，你必须更聪明一些：

```
function isNegZero(n) {
  n = Number( n );
  return (n === 0) && (1 / n === -Infinity);
}

isNegZero( -0 );          // true
isNegZero( 0 / -3 );      // true
isNegZero( 0 );           // false
```

那么，除了学院派的细节以外，我们为什么需要一个负零呢？

在一些应用程序中，开发者使用值的大小来表示一部分信息（比如动画中每一帧的速度），而这个 `number` 的符号来表示另一部分信息（比如移动的方向）。

在这些应用程序中，举例来说，如果一个变量的值变成了 `0`，而它丢失了符号，那么你就丢失了它是从哪个方向移动到 `0` 的信息。保留零的符号避免了潜在的意外信息丢失。

## 特殊等价

正如我们上面看到的，当使用等价性比较时，值 `Nan` 和值 `-0` 拥有特殊的行为。`Nan` 永远不会和自己相等，所以你不得不使用 ES6 的 `Number.isNaN(..)`（或者它的填补）。相似地，`-0` 撒谎并假装它和普通的正零相等（即使使用 `==` 严格等价——见第四章），所以你不得不使用我们上面建议的某些 `isNegZero(..)` 黑科技工具。

在ES6中，有一个新工具可以用于测试两个值的绝对等价性，而没有任何这些例外。它称为 `Object.is(..)`：

```
var a = 2 / "foo";
var b = -3 * 0;

Object.is( a, NaN );      // true
Object.is( b, -0 );       // true

Object.is( b, 0 );         // false
```

对于前ES6环境，这是一个相当简单的 `Object.is(..)` 填补：

```
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;
    }
    // 其他情况
    return v1 === v2;
  };
}
```

`Object.is(..)` 可能不应当用于那些 `==` 或 `===` 已知安全的情况（见第四章“强制转换”），因为这些操作符可能高效得多，并且更惯用/常见。`Object.is(..)` 很大程度上是为这些特殊的等价情况准备的。

## 值与引用

在其他许多语言中，根据你使用的语法，值可以通过值拷贝，也可以通过引用拷贝来赋予/传递。

比如，在C++中如果你想要把一个 `number` 变量传递进一个函数，并使这个变量的值被更新，你可以用 `int& myNum` 这样的东西来声明函数参数，当你传入一个变量 `x` 时，`myNum` 将是一个指向 `x` 的引用；引用就像一个特殊形式的指针，你得到的是一个指向另一个变量的指针（像一个别名 (`alias`)）。如果你没有声明一个引用参数，被传入的值将总是被拷贝的，就算它是一个复杂的对象。

在JavaScript中，没有指针，并且引用的工作方式有一点儿不同。你不能拥有一个从一个JS变量到另一个JS变量的引用。这是完全不可能的。

JS中的引用指向一个（共享的）值，所以如果你有10个不同的引用，它们都总是同一个共享值的不同引用；它们没有一个是另一个的引用/指针。

另外，在JavaScript中，没有语法上的提示可以控制值和引用的赋值/传递。取而代之的是，值的类型用来唯一控制值是通过值拷贝，还是引用拷贝来赋予。

让我们来展示一下：

```
var a = 2;
var b = a; // `b`总是`a`中的值的拷贝
b++;
a; // 2
b; // 3

var c = [1,2,3];
var d = c; // `d`是共享值`[1,2,3]`的引用
d.push( 4 );
c; // [1,2,3,4]
d; // [1,2,3,4]
```

简单值（也叫基本标量）总是通过值拷贝来赋予/传

递：`null`，`undefined`，`string`，`number`，`boolean`，以及ES6的`symbol`。

复合值——`object`（包括`array`，和所有的对象包装器——见第三章）和`function`——总是在赋值或传递时创建一个引用的拷贝。

在上面的代码段中，因为`2`是一个基本标量，`a`持有一个这个值的初始拷贝，而`b`被赋予了这个值的另一个拷贝。当改变`b`时，你根本没有在改变`a`中的值。

但`c`和`d`两个都是同一个共享的值`[1,2,3]`的分离的引用。重要的是，`c`和`d`对值`[1,2,3]`的“拥有”程度上是一样的——它们只是同一个值的对等引用。所以，不管使用哪一个引用去修改（`.push(4)`）实际上共享的`array`值本身，影响的仅仅是这一个共享值，而且这两个引用将会指向新修改的值`[1,2,3,4]`。

因为引用指向的是值本身而不是变量，你不能使用一个引用来改变另一个引用所指向的值：

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

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

当我们做赋值操作 `b = [4,5,6]` 时，我们做的事情绝对不会对 `a` 所指向的位置（`[1,2,3]`）造成任何影响。如果那可能的话，`b` 就会是 `a` 的指针而不是这个 `array` 的引用——但是这样的能力在JS中是不存在的！

这样的困惑最常见于函数参数：

```
function foo(x) {
  x.push( 4 );
  x; // [1,2,3,4]

  // 稍后
  x = [4,5,6];
  x.push( 7 );
  x; // [4,5,6,7]
}

var a = [1,2,3];

foo( a );

a; // [1,2,3,4] 不是 [4,5,6,7]
```

当我们传入参数 `a` 时，它将一份 `a` 引用的拷贝赋值给 `x`。`x` 和 `a` 是指向相同的 `[1,2,3]` 的不同引用。现在，在函数内部，我们可以使用这个引用来改变值本身（`push(4)`）。但是当我们进行赋值操作 `x = [4,5,6]` 时，不可能影响原来的引用 `a` 所指向的东西——它仍然指向（已经被修改了的）值 `[1,2,3,4]`。

没有办法可以使用 `x` 引用来改变 `a` 指向哪里。我们只能修改 `a` 和 `x` 共通指向的那个共享值的内容。

要想改变 `a` 来使它拥有内容为 `[4,5,6,7]` 的值，你不能创建一个新的 `array` 并赋值——你必须修改现存的 `array` 值：

```
function foo(x) {
  x.push( 4 );
  x; // [1,2,3,4]

  // 稍后
  x.length = 0; // 原地清空既存的数组
  x.push( 4, 5, 6, 7 );
  x; // [4,5,6,7]
}

var a = [1,2,3];

foo( a );

a; // [4,5,6,7] 不是 [1,2,3,4]
```

正如你看到的，`x.length = 0` 和 `x.push(4,5,6,7)` 没有创建一个新的 `array`，但是修改了现存的共享 `array`。所以理所当然地，`a` 引用了新的内容 `[4,5,6,7]`。

记住：你不能直接控制/覆盖值拷贝和引用拷贝的行为——这些语义是完全由当前值的类型来控制的。

为了实质上地通过值拷贝传递一个复合值（比如一个 `array`），你需要手动制造一个它的拷贝，使被传递的引用不指向原来的值。比如：

```
foo( a.slice() );
```

不带参数的 `slice(..)` 方法默认地为这个 `array` 制造一个全新的（浅）拷贝。所以，我们传入的引用仅指向拷贝的 `array`，这样 `foo(..)` 不会影响 `a` 的内容。

反之——传递一个基本标量值，使它的值的变化可见，就像引用那样——你不得不将这个值包装在另一个可以通过引用拷贝来传递的复合值中（`object`，`array`，等等）：

```
function foo(wrapper) {
  wrapper.a = 42;
}

var obj = {
  a: 2
};

foo( obj );

obj.a; // 42
```

这里，`obj` 作为基本标量属性 `a` 的包装。当传递给 `foo(..)` 时，一个 `obj` 引用的拷贝被传入并设置给 `wrapper` 参数。我们现在可以使用 `wrapper` 引用来访问这个共享的对象，并更新它的值。在函数完成时，`obj.a` 将被更新为值 `42`。

你可能会遇到这样的情况，如果你想要传入一个像 `2` 这样的基本标量值的引用，你可以将这个值包装在它的 `Number` 对象包装器中（见第三章）。

这个 `Number` 对象的引用的拷贝将会被传递给函数是事实，但不幸的是，和你可能期望的不同，拥有一个共享独享的引用不会给你修改这个共享的基本值的能力：

```

function foo(x) {
  x = x + 1;
  x; // 3
}

var a = 2;
var b = new Number(a); // 或等价的 `Object(a)`

foo(b);
console.log(b); // 2, 不是 3

```

这里的问题是，底层的基本标量值是不可变的（`String` 和 `Boolean` 也一样）。如果一个 `Number` 对象持有一个基本标量值 `2`，那么这个 `Number` 对象就永远不能再持有另一个值；你只能用一个不同的值创建一个全新的 `Number` 对象。

当 `x` 用于表达式 `x + 1` 时，底层的基本标量值 `2` 被自动地从 `Number` 对象中开箱（抽出），所以 `x = x + 1` 这一行很微妙地将 `x` 从一个共享的 `Number` 对象的引用，改变为仅持有加法操作 `2 + 1` 的结果 `3` 的基本标量值。因此，外面的 `b` 仍然引用原来的未被改变/不可变的，持有 `2` 的 `Number` 对象。

你可以在 `Number` 对象上添加属性（只是不要改变它内部的基本值），所以你可间接地通过这些额外的属性交换信息。

不过，这可不太常见；对大多数开发者来说这可能不是一个好的做法。

与其这样使用 `Number` 包装器对象，使用早先的代码段中那样的手动对象包装器（`obj`）要好得多。这不是说像 `Number` 这样包装好的对象包装器没有用处——而是说在大多数情况下，你可能应该优先使用基本标量值的形式。

引用十分强大，但是有时候它们碍你的事儿，而有时你会在它们不存在时需要它们。你唯一可以用来控制引用与值拷贝的东西是值本身的类型，所以你必须通过你选用的值的类型来间接地影响赋值/传递行为。

## 复习

在JavaScript中，`array` 仅仅是数字索引的集合，可以容纳任何类型的值。`string` 是某种“类 `array`”，但它们有着不同的行为，如果你想要将它们作为 `array` 对待的话，必须要小心。JavaScript中的数字既包括“整数”也包括浮点数。

几种特殊值被定义在基本类型内部。

`null` 类型只有一个值 `null`，`undefined` 类型同样地只有 `undefined` 值。对于任何没有值存在的变量或属性，`undefined` 基本上是默认值。`void` 操作符允许你从任意另一个值中创建 `undefined` 值。

`number` 包含几种特殊值，比如 `NaN`（意为“不是一个数字”，但称为“非法数字”更合适）；`+Infinity` 和 `-Infinity`；还有 `-0`。

简单基本标量（`string`，`number` 等）通过值拷贝进行赋值/传递，而复合值（`object` 等）通过引用拷贝进行赋值/传递。引用与其他语言中的引用/指针不同——它们从不指向其他的变量/引用，而仅指向底层的值。

# 你不懂JS：类型与文法

## 第三章：原生类型

在第一和第二章中，我们几次提到了各种内建类型，通常称为“原生类型”，比如 `String` 和 `Number`。现在让我们来仔细检视它们。

这是最常用的原生类型的一览：

- `String()`
- `Number()`
- `Boolean()`
- `Array()`
- `Object()`
- `Function()`
- `RegExp()`
- `Date()`
- `Error()`
- `Symbol()` —— 在ES6中被加入的！

如你所见，这些原生类型实际上是内建函数。

如果你拥有像Java语言那样的背景，JavaScript的 `String()` 看起来像是你曾经用来创建字符串值的 `String(..)` 构造器。所以，你很快就会观察到你可以做这样的事情：

```
var s = new String( "Hello World!" );
console.log( s.toString() ); // "Hello World!"
```

这些原生类型的每一种确实可以被用作一个原生类型的构造器。但是被构建的东西可能与你想象的不同：

```
var a = new String( "abc" );
typeof a; // "object" ... 不是 "String"
a instanceof String; // true
Object.prototype.toString.call( a ); // "[object String]"
```

创建值的构造器形式（`new String("abc")`）的结果是一个基本类型值（`"abc"`）的包装器对象。

重要的是，`typeof` 显示这些对象不是它们自己的特殊类型，而是 `object` 类型的子类型。

这个包装器对象可以被进一步观察，像这样：

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

这个语句的输出会根据你使用的浏览器变化，因为对于开发者的查看，开发者控制台可以自由选择它认为合适的方式来序列化对象。

注意：在写作本书时，最新版的Chrome打印出这样的东西：`String {0: "a", 1: "b", 2: "c", length: 3, [[PrimitiveValue]]: "abc"}`。但是老版本的Chrome曾经只打印出这些：`String {0: "a", 1: "b", 2: "c"}`。当前最新版的Firefox打印 `String ["a", "b", "c"]`，但它曾经以斜体字打印 `"abc"`，点击它可以打开对象查看器。当然，这些结果是总频繁变更的，而且你的体验也许不同。

重点是，`new String("abc")` 为 `"abc"` 创建了一个字符串包装器对象，而不仅是基本类型值 `"abc"` 本身。

## 内部 `[[Class]]`

`typeof` 的结果为 `"object"` 的值（比如数组）被额外地打上了一个内部的标签属性 `[[Class]]`（请把它考虑为一个内部的分类方法，而非与传统的面向对象编码的类有关）。这个属性不能直接地被访问，但通常可以间接地通过在这个值上借用默认的 `Object.prototype.toString()` 方法调用来展示。举例来说：

```
Object.prototype.toString.call( [1,2,3] );           // "[object Array]"
Object.prototype.toString.call( /regex-literal/i );    // "[object RegExp]"
```

所以，对于这个例子中的数组来说，内部的 `[[Class]]` 值是 `"Array"`，而对于正则表达式，它是 `"RegExp"`。在大多数情况下，这个内部的 `[[Class]]` 值对应于关联这个值的内建的原生类型构造器（见下面的讨论），但事实却不总是这样。

基本类型呢？首先，`null` 和 `undefined`：

```
Object.prototype.toString.call( null );           // "[object Null]"
Object.prototype.toString.call( undefined );        // "[object Undefined]"
```

你会注意到，不存在 `Null()` 和 `Undefined()` 原生类型构造器，但不管怎样 `"Null"` 和 `"Undefined"` 是被暴露出来的内部 `[[Class]]` 值。

但是对于像 `string`，`number`，和 `boolean` 这样的简单基本类型，实际上会启动另一种行为，通常称为“封箱（boxing）”（见下一节“封箱包装器”）：

```
Object.prototype.toString.call( "abc" );      // "[object String]"
Object.prototype.toString.call( 42 );          // "[object Number]"
Object.prototype.toString.call( true );         // "[object Boolean]"
```

在这个代码段中，每一个简单基本类型都自动地被它们分别对应的对象包装器封箱，这就是为什么 `"String"`，`"Number"`，和 `"Boolean"` 分别被显示为内部 `[[Class]]` 值。

注意：从ES5发展到ES6的过程中，这里展示的 `toString()` 和 `[[Class]]` 的行为发生了一点儿改变，但我们在本系列的 *ES6与未来一书* 中讲解它们的细节。

## 封箱包装器

这些对象包装器服务于一个非常重要的目的。基本类型值没有属性或方法，所以为了访问 `.length` 或 `.toString()` 你需要这个值的对象包装器。值得庆幸的是，JS将会自动地封箱（也就是包装）基本类型值来满足这样的访问。

```
var a = "abc";
a.length; // 3
a.toUpperCase(); // "ABC"
```

那么，如果你想以通常的方式访问这些字符串值上的属性/方法，比如一个 `for` 循环的 `i < a.length` 条件，这么做看起来很有道理：一开始就得到一个这个值的对象形式，于是JS引擎就不需要隐含地为你创建一个。

但事实证明这是一个坏主意。浏览器们长久以来就对 `.length` 这样的常见情况进行性能优化，这意味着如果你试着直接使用对象形式（它们没有被优化过）进行“提前优化”，那么实际上你的程序将会变慢。

一般来说，基本上没有理由直接使用对象形式。让封箱在需要的地方隐含地发生会更好。换句话说，永远也不要写 `new String("abc")`，`new Number(42)` 这样的事情——应当总是偏向于使用基本类型字面量 `"abc"` 和 `42`。

## 对象包装器的坑

如果你确实选择要直接使用对象包装器，那么有几个坑你应该注意。

举个例子，考虑 `Boolean` 包装的值：

```
var a = new Boolean( false );
if (!a) {
    console.log( "Oops" ); // 永远不会运行
}
```

这里的问题是，虽然你为值 `false` 创建了一个对象包装器，但是对象本身是“*truthy*”（见第四章），所以使用对象的效果是与使用底层的值 `false` 本身相反的，这与通常的期望十分不同。

如果你想手动封箱一个基本类型值，你可以使用 `Object(..)` 函数（没有 `new` 关键字）：

```
var a = "abc";
var b = new String( a );
var c = Object( a );

typeof a; // "string"
typeof b; // "object"
typeof c; // "object"

b instanceof String; // true
c instanceof String; // true

Object.prototype.toString.call( b ); // "[object String]"
Object.prototype.toString.call( c ); // "[object String]"
```

再说一遍，通常不鼓励直接使用封箱的包装器对象（比如上面的 `b` 和 `c`），但你可能会遇到一些它们有用的罕见情况。

## 开箱

如果你有一个包装器对象，而你想要取出底层的基本类型值，你可以使用 `valueOf()` 方法：

```
var a = new String( "abc" );
var b = new Number( 42 );
var c = new Boolean( true );

a.valueOf(); // "abc"
b.valueOf(); // 42
c.valueOf(); // true
```

当以一种查询基本类型值的方式使用对象包装器时，开箱也会隐含地发生。这个处理的过程（强制转换）将会在第四章中更详细地讲解，但简单地说：

```
var a = new String( "abc" );
var b = a + ""; // `b` 拥有开箱后的基本类型值"abc"

typeof a; // "object"
typeof b; // "string"
```

## 原生类型作为构造器

对于 `array`，`object`，`function`，和正则表达式值来说，使用字面形式来创建它们的值几乎总是更好的选择，而且字面形式与构造器形式所创建的值是同一种对象（也就是，没有非包装的值）。

正如我们刚刚在上面看到的其他原生类型，除非你真的知道你需要这些构造器形式，一般来说应当避免使用它们，这主要是因为它们会带来一些你可能不会想要对付的异常和陷阱。

### Array(..)

```
var a = new Array( 1, 2, 3 );
a; // [1, 2, 3]

var b = [1, 2, 3];
b; // [1, 2, 3]
```

注意：`Array(..)` 构造器不要求在它前面使用 `new` 关键字。如果你省略它，它也会像你已经使用了一样动作。所以 `Array(1,2,3)` 和 `new Array(1,2,3)` 的结果是一样的。

`Array` 构造器有一种特殊形式，如果它仅仅被传入一个 `number` 参数，与将这个值作为数组的内容不同，它会被认为是用来“预定数组大小”（嗯，某种意义上）用的长度。

这是个可怕的主意。首先，你会意外地用错这种形式，因为它很容易忘记。

但更重要的是，其实没有预定数组大小这样的东西。你所创建的是一个空数组，并将这个数组的 `length` 属性设置为那个指定的数字值。

一个数组在它的值槽上没有明确的值，但是有一个 `length` 属性意味着这些值槽是存在的，在 JS 中这是一个诡异的数据结构，它带有一些非常奇怪且令人困惑的行为。可以创建这样的值的能力，完全源自于老旧的，已经废弃的，仅具有历史意义的功能（比如 `arguments` 这样的“类数组对象”）。

注意：带有至少一个“空值槽”的数组经常被称为“稀散数组”。

这是另外一个例子，展示浏览器的开发者控制台在如何表示这样的对象上有所不同，它产生了更多的困惑。

举例来说：

```
var a = new Array( 3 );
a.length; // 3
a;
```

在Chrome中 `a` 的序列化表达是（在本书写作时）：`[ undefined × 3 ]`。这真的很不幸。它暗示着在这个数组的值槽中有三个 `undefined` 值，而事实上这样的值槽是不存在的（所谓的“空值槽（empty slots）”——也是一个烂名字！）。

要观察这种不同，试试这段代码：

```
var a = new Array( 3 );
var b = [ undefined, undefined, undefined ];
var c = [];
c.length = 3;

a;
b;
c;
```

注意：正如你在这个例子中看到的 `c`，数组中的空值槽可以在数组的创建之后发生。将数组的 `length` 改变为超过它实际定义的槽值的数目，你就隐含地引入了空值槽。事实上，你甚至可以在上面的代码段中调用 `delete b[1]`，而这么做将会在 `b` 的中间引入一个空值槽。

对于 `b`（在当前的Chrome中），你会发现它的序列化表现为 `[ undefined, undefined, undefined ]`，与之相对的是 `a` 和 `c` 的 `[ undefined × 3 ]`。糊涂了吧？是的，大家都糊涂了。

更糟糕的是，在写作本书时，Firefox对 `a` 和 `c` 报告 `[ , , , ]`。你发现为什么这使人犯糊涂了吗？仔细看。三个逗号表示有四个值槽，不是我们期望的三个值槽。

什么！？Firefox在它们的序列化表达的末尾放了一个额外的`,`，因为在ES5中，列表（数组值，属性列表等等）末尾的逗号是允许的（被砍掉并忽略）。所以如果你在你的程序或控制台中敲入 `[ , , , ]` 值，你实际上得到的是一个底层为 `[ , , ]` 的值（也就是，一个带有三个空值槽的数组）。这种选择，虽然在阅读开发者控制台时使人困惑，但是因为它使拷贝粘贴的时候准确，所以被留了下来。

如果你现在在摇头或翻白眼儿，你并不孤单！（耸肩）

不幸的是，事情越来越糟。比在控制台的输出产生的困惑更糟的是，上面代码段中的 `a` 和 `b` 实际上在有些情况下相同，但在另一些情况下不同：

```
a.join( "-" ); // "---"
b.join( "-" ); // "---"

a.map(function(v,i){ return i; }); // [ undefined × 3 ]
b.map(function(v,i){ return i; }); // [ 0, 1, 2 ]
```

呃。

`a.map(..)` 调用会失败 是因为值槽根本就不实际存在，所以 `map(..)` 没有东西可以迭代。`join(..)` 的工作方式不同，基本上我们可以认为它是像这样被实现的：

```
function fakeJoin(arr,connector) {
  var str = "";
  for (var i = 0; i < arr.length; i++) {
    if (i > 0) {
      str += connector;
    }
    if (arr[i] !== undefined) {
      str += arr[i];
    }
  }
  return str;
}

var a = new Array( 3 );
fakeJoin( a, "-" ); // "---"
```

如你所见，`join(..)` 好用仅仅是因为它认为值槽存在，并循环至 `length` 值。不管 `map(..)` 内部是在做什么，它（显然）没有做出这样的假设，所以源自于奇怪的“空值槽”数组的结果出人意料，而且好像是失败了。

那么，如果你想要确实创建一个实际的 `undefined` 值的数组（不只是“空值槽”），你如何才能做到呢（除了手动以外）？

```
var a = Array.apply( null, { length: 3 } );
a; // [ undefined, undefined, undefined ]
```

糊涂了吧？是的。这里是它大概的工作方式。

`apply(..)` 是一个对所有函数可用的工具方法，它以一种特殊方式调用这个使用它的函数。

第一个参数是一个 `this` 对象绑定（在本系列的 `this` 与对象原型 中有详细讲解），在这里我们不关心它，所以我们将其设置为 `null`。第二个参数应该是一个数组（或像数组的东西——也就是“类数组对象”）。这个“数组”的内容作为这个函数的参数“扩散”开来。

所以，`Array.apply(..)` 在调用 `Array(..)` 函数，并将一个值（`{ length: 3 }` 对象值）作为它的参数值扩散开。

在 `apply(..)` 内部，我们可以预见这里有另一个 `for` 循环（有些像上面的 `join(..)`），它从 `0` 开始上升但不包含至 `length`（这个例子中是 `3`）。

对于每一个索引，它从对象中取得相应的键。所以如果这个数组对象参数在 `apply(..)` 内部被命名为 `arr`，那么这种属性访问实质上是 `arr[0]`，`arr[1]`，和 `arr[2]`。当然，没有一个属性是在 `{ length: 3 }` 对象值上存在的，所以这三个属性访问都将返回值 `undefined`。

换句话说，调用 `Array(..)` 的结局基本上是这样：`Array(undefined, undefined, undefined)`，这就是我们如何得到一个填满 `undefined` 值的数组的，而非仅仅是一些（疯狂的）空值槽。

虽然对于创建一个填满 `undefined` 值的数组来说，`Array.apply( null, { length: 3 } )` 是一个奇怪而且繁冗的方法，但是它要比使用砸自己的脚似的 `Array(3)` 空值槽要可靠和好得太多了。

底线：你在任何情况下，永远不，也不应该有意地创建并使用诡异的空值槽数组。就别这么干。它们是怪胎。

## **Object(..)**，**Function(..)**，和**RegExp(..)**

`Object(..)` / `Function(..)` / `RegExp(..)` 构造器一般来说也是可选的（因此除非是特别的目的，应当避免使用）：

```
var c = new Object();
c.foo = "bar";
c; // { foo: "bar" }

var d = { foo: "bar" };
d; // { foo: "bar" }

var e = new Function( "a", "return a * 2;" );
var f = function(a) { return a * 2; };
function g(a) { return a * 2; }

var h = new RegExp( "^a*b+", "g" );
var i = /^a*b+/g;
```

几乎没有理由使用 `new Object()` 构造器形式，尤其因为它强迫你一个一个地添加属性，而不是像对象的字面形式那样一次添加许多。

`Function` 构造器仅在最最罕见的情况下有用，也就是你需要动态地定义一个函数的参数和/或它的函数体。不要将 `Function(..)` 仅仅作为另一种形式的 `eval(..)`。你几乎永远不会需要用这种方式动态定义一个函数。

用字面量形式 (`/^a*b+/g`) 定义正则表达式是被大力采用的，不仅因为语法简单，而且还有性能的原因——JS引擎会在代码执行前预编译并缓存它们。和我们迄今看到的其他构造器形式不同，`RegExp(..)` 有一些合理的用途：用来动态定义一个正则表达式的范例。

```
var name = "Kyle";
var namePattern = new RegExp( "\b(?: " + name + ")+\b", "ig" );

var matches = someText.match( namePattern );
```

这样的场景在JS程序中一次又一次地合法出现，所以你有需要使用 `new RegExp("pattern", "flags")` 形式。

## Date(..) 和 Error(..)

`Date(..)` 和 `Error(..)` 原生类型构造器要比其他种类的原生类型有用得多，因为它们没有字面量形式。

要创建一个日期对象值，你必须使用 `new Date()`。`Date(..)` 构造器接收可选参数值来指定要使用的日期/时间，但是如果省略的话，就会使用当前的日期/时间。

目前你构建一个日期对象的最常见的理由是要得到当前的时间戳（一个有符号整数，从1970年1月1日开始算起的毫秒数）。你可以在一个日期对象实例上调用 `getTime()` 得到它。

但是在ES5中，一个更简单的方法是调用定义为 `Date.now()` 的静态帮助函数。而且在前ES5中填补它很容易：

```
if (!Date.now) {
  Date.now = function(){
    return (new Date()).getTime();
  };
}
```

注意：如果你不带 `new` 调用 `Date()`，你将会得到一个那个时刻的日期/时间的字符串表达。在语言规范中没有规定这个表达的确切形式，虽然各个浏览器趋向于赞同使用这样的东西：`"Fri Jul 18 2014 00:31:02 GMT-0500 (CDT)"`。

`Error(..)` 构造器（很像上面的 `Array()`）在有 `new` 与没有 `new` 时的行为是相同的。

你想要创建`error`对象的主要原因是，它会将当前的执行栈上下文捕捉进对象中（在大多数JS引擎中，在创建后使用只读的 `.stack` 属性表示）。这个栈上下文包含函数调用栈和`error`对象被创建时的行号，这使调试这个错误更简单。

典型地，你将与 `throw` 操作符一起使用这样的`error`对象：

```

function foo(x) {
  if (!x) {
    throw new Error( "x wasn't provided" );
  }
  // ...
}

```

`Error`对象实例一般拥有至少一个 `message` 属性，有时还有其他属性（你应当将它们作为只读的），比如 `type`。然而，与其检视上面提到的 `stack` 属性，最好是在`error`对象上调用 `toString()`（明确地调用，或者是通过强制转换隐含地调用——见第四章）来得到一个格式友好的错误消息。

提示：技术上讲，除了一般的 `Error(..)` 原生类型以外，还有几种特定错误的原生类型：`Evaluator(..)`，`RangeError(..)`，`ReferenceError(..)`，`SyntaxError(..)`，`TypeError(..)`，和 `URIError(..)`。但是手动使用这些特定错误原生类型十分少见。如果你的程序确实遭受了一个真实的异常，它们是会自动地被使用的（比如引用一个未声明的变量而得到一个 `ReferenceError` 错误）。

## Symbol(..)

在ES6中，新增了一个基本值类型，称为“`Symbol`（标志）”。`Symbol`是一种特殊的“独一无二”（不是严格保证的！）的值，可以作为对象上的属性使用而几乎不必担心任何冲突。它们主要是为特殊的ES6结构的内建行为设计的，但你也可以定义你自己的`symbol`。

`Symbol`可以用做属性名，但是你不能从你的程序中看到或访问一个`symbol`的实际值，从开发者控制台也不行。例如，如果你在开发者控制台中对一个`Symbol`求值，将会显示 `Symbol(Symbol.create)` 之类的东西。

在ES6中有几种预定义的`Symbol`，做为 `Symbol` 函数对象的静态属性访问，比如 `Symbol.create`，`Symbol.iterator` 等等。要使用它们，可以这样做：

```
obj[Symbol.iterator] = function(){ /*...*/ };
```

要定义你自己的`Symbol`，使用 `Symbol(..)` 原生类型。`Symbol(..)` 原生类型“构造器”很独特，因为它不允许你将 `new` 与它一起使用，这么做会抛出一个错误。

```

var mysym = Symbol( "my own symbol" );
mysym;           // Symbol(my own symbol)
mysym.toString(); // "Symbol(my own symbol)"
typeof mysym;    // "symbol"

var a = { };
a[mysym] = "foobar";

Object.getOwnPropertySymbols( a );
// [ Symbol(my own symbol) ]

```

虽然 `Symbol` 实际上不是私有的（在对象上使用 `Object.getOwnPropertySymbols(..)` 反射，揭示了 `Symbol` 其实是相当公开的），但是它们的主要用途可能是私有属性，或者类似的特殊属性。对于大多数开发者，他们也许会在属性名上加入 `_` 下划线前缀，这在经常在惯例上表示：“这是一个私有的/特殊的/内部的属性，别碰！”

注意：`Symbol` 不是 `object`，它们是简单的基本标量。

## 原生类型原型

每一个内建的原生构造器都拥有它自己的 `.prototype` 对象  
—— `Array.prototype`，`String.prototype` 等等。

对于它们特定的对象子类型，这些对象含有独特的行为。

例如，所有的字符串对象，和 `string` 基本值的扩展（通过封箱），都可以访问  
在 `String.prototype` 对象上做为方法定义的默认行为。

注意：做为文档惯例，`String.prototype.XYZ` 会被缩写为 `String#XYZ`，对于其它所有 `.prototype` 的属性都是如此。

- `String#indexOf(..)`：在一个字符串中找出一个子串的位置
- `String#charAt(..)`：访问一个字符串中某个位置的字符
- `String#substr(..)`，`String#substring(..)`，和 `String#slice(..)`：将字符串的一部分抽取为一个新字符串
- `String#toUpperCase()` 和 `String#toLowerCase()`：创建一个转换为大写或小写的新字符串
- `String#trim()`：创建一个截去开头或结尾空格的新字符串。

这些方法中没有一个是在原地修改字符串的。修改（比如大小写变换或去空格）会根据当前的值来创建一个新的值。

有赖于原型委托（见本系列的 `this` 与对象原型），任何字符串值都可以访问这些方法：

```
var a = " abc ";
a.indexOf("c"); // 3
a.toUpperCase(); // " ABC "
a.trim(); // "abc"
```

其他构造器的原型包含适用于它们类型的行为，比如 `Number#toFixed(..)`（将一个数字转换为一个固定小数位的字符串）和 `Array#concat(..)`（混合数组）。所有这些函数都可以访问 `apply(..)`，`call(..)`，和 `bind(..)`，因为 `Function.prototype` 定义了它们。

但是，一些原生类型的原型不仅仅是单纯的对象：

```
typeof Function.prototype;           // "function"
Function.prototype();              // 它是一个空函数！

RegExp.prototype.toString();        // "/(?:)/" — 空的正则表达式
"abc".match( RegExp.prototype );   // [""]
```

一个特别差劲儿的主意是，你甚至可以修改这些原生类型的原型（不仅仅是你可能熟悉的添加属性）：

```
Array.isArray( Array.prototype );    // true
Array.prototype.push( 1, 2, 3 );     // 3
Array.prototype;                   // [1,2,3]

// 别这么留着它，要不就等着怪事发生吧！
// 将`Array.prototype`重置为空
Array.prototype.length = 0;
```

如你所见，`Function.prototype` 是一个函数，`RegExp.prototype` 是一个正则表达式，而 `Array.prototype` 是一个数组。有趣吧？酷吧？

## 原型作为默认值

`Function.prototype` 是一个空函数，`RegExp.prototype` 是一个“空”正则表达式（也就是不匹配任何东西），而 `Array.prototype` 是一个空数组，这使它们成了可以赋值给变量的，很好的“默认”值——如果这些类型的变量还没有值。

例如：

```

function isThisCool(vals,fn,rx) {
  vals = vals || Array.prototype;
  fn = fn || Function.prototype;
  rx = rx || RegExp.prototype;

  return rx.test(
    vals.map( fn ).join( "" )
  );
}

isThisCool();           // true

isThisCool(
  ["a","b","c"],
  function(v){ return v.toUpperCase(); },
  /D/
);                      // false

```

注意：在ES6中，我们不再需要使用 `vals = vals || ..` 这样的默认值语法技巧了（见第四章），因为在函数声明中可以通过原生语法为参数设定默认值（见第五章）。

这种方式的一个微小的副作用是，`.prototype` 已经被创建了，而且是内建的，因此它仅被创建一次。相比之下，使用 `[]`，`function(){}()`，和 `/(?:)/` 这些值本身作为默认值，将会（很可能，要看引擎如何实现）在每次调用 `isThisCool(..)` 时重新创建这些值（而且稍可能要回收它们）。这可能会消耗内存/CPU。

另外，要非常小心不要对后续要被修改的值使用 `Array.prototype` 做为默认值。在这个例子中，`vals` 是只读的，但如果要在原地对 `vals` 进行修改，那你实际上修改的是 `Array.prototype` 本身，这将把你引到刚才提到的坑里！

注意：虽然我们指出了这些原生类型的原型和一些用处，但是依赖它们的时候要小心，更要小心以任何形式修改它们。更多的讨论见附录A“原生原型”。

## 复习

JavaScript为基本类型提供了对象包装器，被称为原生类型（`String`，`Number`，`Boolean`，等等）。这些对象包装器使这些值可以访问每种对象子类型的恰当行为（`String#trim()` 和 `Array#concat(..)`）。

如果你有一个像 `"abc"` 这样的简答基本类型标量，而且你想要访问它的 `length` 属性或某些 `String.prototype` 方法，JS会自动地“封箱”这个值（用它所对应种类的对象包装器把它包起来），以满足这样的属性/方法访问。



# 你不懂JS：类型与文法

## 第四章：强制转换

现在我们更全面地了解了JavaScript的类型和值，我们将注意力转向一个极具争议的话题：强制转换。

正如我们在第一章中提到的，关于强制转换到底是一个有用的特性，还是一个语言设计上的缺陷（或位于两者之间！），早就开始就争论不休了。如果你读过关于JS的其他书籍，你就会知道流行在世面上那种淹没一切的声音：强制转换是魔法，是邪恶的，令人困惑的，而且就是彻头彻尾的坏主意。

本着这个系列丛书的总体精神，我认为你应当直面你不理解的东西并设法更全面地搞懂它。而不是因为大家都这样做，或是你曾经被一些怪东西咬到就逃避强制转换。

我们的目标是全面地探索强制转换的优点和缺点（是的，它们有优点！），这样你就能在程序中对它是否合适做出明智的决定。

### 转换值

将一个值从一个类型明确地转换到另一个类型通常称为“类型转换（type casting）”，当这个操作隐含地完成时称为“强制转换（coercion）”（根据一个值如何被使用的规则来强制它变换类型）。

注意：这可能不明显，但是JavaScript强制转换总是得到基本标量值的一种，比如 `string`，`number`，或 `boolean`。没有强制转换可以得到像 `object` 和 `function` 这样的复杂值。第三章讲解了“封箱”，它将一个基本类型标量值包装在它们相应的 `object` 中，但在准确的意义上这不是真正的强制转换。

另一种区别这些术语的常见方法是：“类型转换（type casting/conversion）”发生在静态类型语言的编译时，而“类型强制转换（type coercion）”是动态类型语言的运行时转换。

然而，在JavaScript中，大多数人将所有这些类型的转换都称为 强制转换（coercion），所以我偏好的区别方式是使用“隐含强制转换（implicit coercion）”与“明确强制转换（explicit coercion）”。

其中的区别应当是很明显的：在观察代码时如果一个类型转换明显是有意为之的，那么它就是“明确强制转换”，而如果这个类型转换是做为其他操作的不那么明显的副作用发生的，那么它就是“隐含强制转换”。

例如，考虑这两种强制转换的方式：

```
var a = 42;

var b = a + "";           // 隐含强制转换

var c = String(a);       // 明确强制转换
```

对于 `b` 来说，强制转换是隐含地发生的，因为如果与 `+` 操作符组合的操作数之一是一个 `string` 值（`""`），这将使 `+` 操作成为一个 `string` 连接（将两个字符串加在一起），而 `string` 连接的一个（隐藏的）副作用将 `a` 中的值 `42` 强制转换为它的 `string` 等价物：`"42"`。

相比之下，`String(..)` 函数使一切相当明显，它明确地取得 `a` 中的值，并把它强制转换为一个 `string` 表现形式。

两种方式都能达到相同的效果：从 `42` 变成 `"42"`。但它们如何达到这种效果，才是关于 JavaScript 强制转换的热烈争论的核心。

注意：技术上讲，这里有一些在语法形式区别之上的，行为上的微妙区别。我们将在本章稍后，“隐含：Strings <--> Numbers”一节中仔细讲解。

“明确地”，“隐含地”，或“明显地”和“隐藏的副作用”这些术语，是相对的。

如果你确切地知道 `a + ""` 是在做什么，并且你有意地这么做来强制转换一个 `string`，你可能感觉这个操作已经足够“明确”了。相反，如果你从没见过 `String(..)` 函数被用于 `string` 强制转换，那么对你来说它的行为可能看起来太过隐蔽而让你感到“隐含”。

但我们在基于一个大众的，充分了解，但不是专家或 JS 规范爱好者的开发者的观点来讨论“明确”与“隐含”的。无论你的程度如何，或是没有在这个范畴内准确地找到自己，你都需要根据我们在这里的观察方式，相应地调整你的角度。

记住：我们自己写代码而也只有我们自己会读它，通常是很少见的。即便你是一个精通 JS 里里外外的专家，也要考虑一个经验没那么丰富的队友在读你的代码时感受如何。对于他们和对于你来说，“明确”或“隐含”的意义相同吗？

## 抽象值操作

在我们可以探究 明确 与 隐含 强制转换之前，我们需要学习一些基本规则，是它们控制着值如何变成一个 `string`，`number`，或 `boolean` 的。ES5 语言规范的第 9 部分用值的变形规则定义了几种“抽象操作”（“仅供内部使用的操作”的高大上说法）。我们将特别关注于：`ToString`，`ToNumber`，和 `ToBoolean`，并稍稍关注一下 `ToPrimitive`。

## ToString

当任何一个非 `string` 值被强制转换为一个 `string` 表现形式时，这个转换的过程是由语言规范的9.8部分的 `ToString` 抽象操作处理的。

内建的基本类型值拥有自然的字符串化形式：`null` 变为 `"null"`，`undefined` 变为 `"undefined"`，`true` 变为 `"true"`。`number` 一般会以你期望的自然方式表达，但正如我们在第二章中讨论的，非常小或非常大的 `number` 将会以指数形式表达：

```
// `1.07`乘以`1000`，7次
var a = 1.07 * 1000 * 1000 * 1000 * 1000 * 1000 * 1000 * 1000;

// 7次乘以3位 => 21位
a.toString(); // "1.07e21"
```

对于普通的对象，除非你指定你自己的，默认的 `toString()`（可以在 `Object.prototype.toString()` 找到）将返回内部 `[[Class]]`（见第三章），例如 `"[object Object]"`。

但正如早先所展示的，如果一个对象上拥有它自己的 `toString()` 方法，而你又以一种类似 `string` 的方式使用这个对象，那么它的 `toString()` 将会被自动调用，而且这个调用的 `string` 结果将被使用。

注意：技术上讲，一个对象被强制转换为一个 `string` 要通过 `ToPrimitive` 抽象操作（ES5语言规范，9.1部分），但是那其中的微妙细节将会在本章稍后的 `ToNumber` 部分中讲解，所以我们在这里先跳过它。

数组拥有一个覆盖版本的默认 `toString()`，将数组字符串化为它所有的值（每个都字符串化）的（字符串）连接，并用 `","` 分割每个值。

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

a.toString(); // "1,2,3"
```

重申一次，`toString()` 可以明确地被调用，也可以通过在一个需要 `string` 的上下文环境中使用一个非 `string` 来自动地被调用。

## JSON字符串化

另一种看起来与 `ToString` 密切相关的操作是，使用 `JSON.stringify(..)` 工具将一个值序列化为一个JSON兼容的 `string` 值。

重要的是要注意，这种字符串化与强制转换并不完全是同一种东西。但是因为它与上面讲的 `ToString` 规则有关联，我们将在这里稍微转移一下话题，来讲解JSON字符串化行为。

对于最简单的值，`JSON`字符串化行为基本上和 `toString()` 转换是相同的，除了序列化的结果总是一个 `string`：

```
JSON.stringify( 42 );      // "42"
JSON.stringify( "42" );    // ""42"" (一个包含双引号的字符串)
JSON.stringify( null );    // "null"
JSON.stringify( true );    // "true"
```

任何 `JSON` 安全的值都可以被 `JSON.stringify(..)` 字符串化。但是什么是 `JSON` 安全的？任何可以用 `JSON` 表现形式合法表达的值。

考虑 `JSON` 不安全的值可能更容易一些。一些例子是：`undefined`，`function`，(`ES6+`) `symbol`，和带有循环引用的 `object`（一个对象结构中的属性互相引用而造成了一个永不终结的循环）。对于标准的 `JSON` 结构来说这些都是非法的值，主要是因为它们不能移植到消费 `JSON` 值的其他语言中。

`JSON.stringify(..)` 工具在遇到 `undefined`，`function`，和 `symbol` 时将会自动地忽略它们。如果在一个 `array` 中遇到这样的值，它会被替换为 `null`（这样数组的位置信息就不会改变）。如果在一个 `object` 的属性中遇到这样的值，这个属性会被简单地剔除掉。

考虑下面的代码：

```
JSON.stringify( undefined );           // undefined
JSON.stringify( function(){ } );       // undefined

JSON.stringify( [1,undefined,function(){},4] ); // "[1,null,null,4]"
JSON.stringify( { a:2, b:function(){ } } ); // {"a":2}
```

但如果你试着 `JSON.stringify(..)` 一个带有循环引用的 `object`，就会抛出一个错误。

`JSON` 字符串化有一个特殊行为，如果一个 `object` 值定义了一个 `toJSON()` 方法，这个方法将会上被首先调用，以取得用于序列化的值。

如果你打算 `JSON` 字符串化一个可能含有非法 `JSON` 值的对象，或者如果这个对象中正好有不适合于序列化的值，那么你就应当为它定义一个 `toJSON()` 方法，返回这个 `object` 的一个 `JSON` 安全版本。

例如：

```
var o = { };

var a = {
  b: 42,
  c: o,
  d: function(){}
};

// 在`a`内部制造一个循环引用
o.e = a;

// 这回因循环引用而抛出一个错误
// JSON.stringify( a );

// 自定义一个JSON值序列化
a.toJSON = function() {
  // 序列化仅包含属性`b`
  return { b: this.b };
};

JSON.stringify( a ); // "{"b":42}"
```

一个很常见的误解是，`toJSON()` 应当返回一个JSON字符串化的表现形式。这可能是不正确的，除非你事实上想要字符串化 `string` 本身（通常不会！）。`toJSON()` 应当返回合适的实际普通值（无论什么类型），而 `JSON.stringify(..)` 自己会处理字符串化。

换句话说，`toJSON()` 应当被翻译为：“变为一个适用于字符串化的JSON安全的值”，不是像许多开发者错误认为的那样，“变为一个JSON字符串”。

考虑下面的代码：

```

var a = {
  val: [1, 2, 3],  

  // 可能正确！  

  toJSON: function(){
    return this.val.slice( 1 );
  }
};  

var b = {
  val: [1, 2, 3],  

  // 可能不正确！  

  toJSON: function(){
    return "[" +
      this.val.slice( 1 ).join() +
    "]";
  }
};  

JSON.stringify( a ); // "[2,3]"  

JSON.stringify( b ); // ""[2,3]""
```

在第二个调用中，我们字符串化了返回的 `string` 而不是 `array` 本身，这可能不是我们想要做的。

既然我们说到了 `JSON.stringify(..)`，那么就让我们来讨论一些不那么广为人知，但是仍然很有用的功能吧。

`JSON.stringify(..)` 的第二个参数值是可选的，它称为 替换器 (*replacer*)。这个参数值既可以是一个 `array` 也可以是一个 `function`。与 `toJSON()` 为序列化准备一个值的方式类似，它提供一种过滤机制，指出一个 `object` 的哪一个属性应该或不应该被包含在序列化形式中，来自定义这个 `object` 的递归序列化行为。

如果 替换器 是一个 `array`，那么它应当是一个 `string` 的 `array`，它的每一个元素指定了允许被包含在这个 `object` 的序列化形式中的属性名称。如果一个属性不存在于这个列表中，那么它就会被跳过。

如果 替换器 是一个 `function`，那么它会为 `object` 本身而被调用一次，并且为这个 `object` 中的每个属性都被调用一次，而且每次都被传入两个参数值，`key` 和 `value`。要在序列化中跳过一个 `key`，可以返回 `undefined`。否则，就返回被提供的 `value`。

```

var a = {
  b: 42,
  c: "42",
  d: [1, 2, 3]
};

JSON.stringify( a, ["b", "c"] ); // '{"b":42, "c":"42"}'

JSON.stringify( a, function(k, v){
  if (k !== "c") return v;
} );
// {"b":42, "d":[1,2,3]}

```

注意：在 `function` 替换器的情况下，第一次调用时 `key` 参数 `k` 是 `undefined`（而对象 `a` 本身会被传入）。`if` 语句会过滤掉名称为 `c` 的属性。字符串化是递归的，所以数组 `[1,2,3]` 会将它的每一个值（`1`，`2`，和 `3`）都作为 `v` 传递给替换器，并将索引值（`0`，`1`，和 `2`）作为 `k`。

`JSON.stringify(..)` 还可以接收第三个可选参数值，称为 填充符（`space`），在对人类友好的输出中它被用做缩进。填充符可以是一个正整数，用来指示每一级缩进中应当使用多少个空格字符。或者，填充符可以是一个 `string`，这时每一级缩进将会使用它的前十个字符。

```

var a = {
  b: 42,
  c: "42",
  d: [1, 2, 3]
};

JSON.stringify( a, null, 3 );
// "{"
//   "b": 42,
//   "c": "42",
//   "d": [
//     1,
//     2,
//     3
//   ]
// }"

JSON.stringify( a, null, "-----" );
// "{"
// -----"b": 42,
// -----"c": "42",
// -----"d": [
//       1,
//       2,
//       3
//     ]
// }"

```

记住，`JSON.stringify(..)` 并不直接是一种强制转换的形式。但是，我们在这里讨论它，是由于两个与 `ToString` 强制转换有关联的行为：

1. `string`，`number`，`boolean`，和 `null` 值在 `JSON` 字符串化时，与它们通过 `ToString` 抽象操作的规则强制转换为 `string` 值的方式基本上是相同的。
2. 如果传递一个 `object` 值给 `JSON.stringify(..)`，而这个 `object` 上拥有一个 `toJSON()` 方法，那么在字符串化之前，`toJSON()` 就会被自动调用将这个值（某种意义上）“强制转换”为 `JSON` 安全的。

## ToNumber

如果任何非 `number` 值，以一种要求它是 `number` 的方式被使用，比如数学操作，就会发生 ES5 语言规范在 9.3 部分定义的 `ToNumber` 抽象操作。

例如，`true` 变为 `1` 而 `false` 变为 `0`。`undefined` 变为 `Nan`，而（奇怪的是）`null` 变为 `0`。

对于一个 `string` 值来说，`ToNumber` 工作起来很大程度上与数字字面量的规则/语法很相似（见第三章）。如果它失败了，结果将是 `Nan`（而不是 `number` 字面量中会出现的语法错误）。一个不同之处的例子是，在这个操作中 `0` 前缀的八进制数不会被作为八进制数来处理（而仅作为普通的十进制小数），虽然这样的八进制数作为 `number` 字面量是合法的。

注意：`number` 字面量文法与用于 `string` 值的 `ToNumber` 间的区别极其微妙，在这里就不进一步讲解了。更多的信息可以参考ES语言规范的9.3.1部分。

对象（以及数组）将会首先被转换为它们的基本类型值的等价物，而后这个结果值（如果它还不是一个 `number` 基本类型）会根据刚才提到的 `ToNumber` 规则被强制转换为一个 `number`。

为了转换为基本类型值的等价物，`ToPrimitive` 抽象操作（ES5语言规范，9.1部分）将会查询这个值（使用内部的 `DefaultValue` 操作——ES5语言规范，8.12.8部分），看它有没有 `valueOf()` 方法。如果 `valueOf()` 可用并且它返回一个基本类型值，那么这个值就将用于强制转换。如果不是这样，但 `toString()` 可用，那么就由它来提供用于强制转换的值。

如果这两种操作都没提供一个基本类型值，就会抛出一个 `TypeError`。

在ES5中，你可以创建这样一个不可强制转换的对象——没有 `valueOf()` 和 `toString()`——如果它的 `[[Prototype]]` 的值为 `null`，这通常是通过 `Object.create(null)` 来创建的。关于 `[[Prototype]]` 的详细信息参见本系列的 `this`与对象原型。

注意：我们会在本章稍后讲解如何强制转换至 `number`，但对于下面的代码段，想象 `Number(..)` 函数就是那样做的。

考虑如下代码：

```

var a = {
    valueOf: function(){
        return "42";
    }
};

var b = {
    toString: function(){
        return "42";
    }
};

var c = [4,2];
c.toString = function(){
    return this.join( "" );      // "42"
};

Number( a );                  // 42
Number( b );                  // 42
Number( c );                  // 42
Number( "" );                 // 0
Number( [] );                 // 0
Number( [ "abc" ] );          // NaN

```

## ToBoolean

下面，让我们聊一聊在JS中 `boolean` 如何动作。市面上关于这个话题有许多的困惑和误解，所以集中注意力！

首先而且最重要的是，JS实际上拥有 `true` 和 `false` 关键字，而且它们的行为正如你所期望的 `boolean` 值一样。一个常见的误解是，值 `1` 和 `0` 与 `true / false` 是相同的。虽然这可能在其他语言中是成立的，但在JS中 `number` 就是 `number`，而 `boolean` 就是 `boolean`。你可以将 `1` 强制转换为 `true`（或反之），或将 `0` 强制转换为 `false`（或反之）。但它们不是相同的。

## Falsy值

但这还不是故事的结尾。我们需要讨论一下，除了这两个 `boolean` 值以外，当你把其他值强制转换为它们的 `boolean` 等价物时如何动作。

所有的JavaScript值都可以被划分进两个类别：

1. 如果被强制转换为 `boolean`，将成为 `false` 的值
2. 其它的一切值（很明显将变为 `true`）

我不是在出洋相。JS语言规范给那些在强制转换为 `boolean` 值时将会变为 `false` 的值定义了一个明确的，小范围的列表。

我们如何才能知道这个列表中的值是什么？在ES5语言规范中，9.2部分定义了一个 `ToBoolean` 抽象操作，它讲述了对所有可能的值而言，当你试着强制转换它们为 `boolean` 时究竟会发生什么。

从这个表格中，我们得到了下面所谓的“`falsy`”值列表：

- `undefined`
- `null`
- `false`
- `+0` , `-0` , and `Nan`
- `""`

就是这些。如果一个值在这个列表中，它就是一个“`falsy`”值，而且当你在它上面进行 `boolean` 强制转换时它会转换为 `false`。

通过逻辑上的推论，如果一个值不在这个列表中，那么它一定在另一个列表中，也就是我们称为“`truthy`”值的列表。但是JS没有真正定义一个“`truthy`”列表。它给出了一些例子，比如它说所有的对象都是`truthy`，但是语言规范大致上暗示着：任何没有明确地存在于`falsy`列表中的东西，都是`truthy`。

## Falsy 对象

等一下，这一节的标题听起来简直是矛盾的。我刚刚才说过 语言规范将所有对象称为 `truthy`，对吧？应该没有“`falsy` 对象”这样的东西。

这会是什么意思呢？

它可能诱使你认为它意味着一个包装了`falsy`值（比如 `""`，`0` 或 `false`）的对象包装器（见第三章）。但别掉到这个陷阱中。

注意：这个可能是一个语言规范的微妙笑话。

考虑下面的代码：

```
var a = new Boolean( false );
var b = new Number( 0 );
var c = new String( "" );
```

我们知道这三个值都是包装了明显是`falsy`值的对象（见第三章）。但这些对象是作为 `true` 还是作为 `false` 动作呢？这很容易回答：

```
var d = Boolean( a && b && c );
d; // true
```

所以，三个都作为 `true` 动作，这是唯一能使 `d` 得到 `true` 的方法。

提示：注意包在 `a && b && c` 表达式外面的 `Boolean( ... )` —— 你可能想知道为什么它在这儿。我们会在本章稍后回到这个话题，所以先做个心理准备。为了先睹为快，你可以自己试试如果没有 `Boolean( ... )` 调用而只有 `d = a && b && c` 时 `d` 是什么。

那么，如果“**falsy**对象”不是包装着**falsy**值的对象，它们是什么鬼东西？

刁钻的地方在于，它们可以出现在你的JS程序中，但它们实际上不是JavaScript本身的一部分。

什么！？

有些特定的情况，在普通的JS语义之上，浏览器已经创建了它们自己的某种 外来值的行为，也就是这种“**falsy**对象”的想法。

一个“**falsy**对象”看起来和动起来都像一个普通对象（属性，等等）的值，但是当你强制转换它为一个 `boolean` 时，它会变为一个 `false` 值。

为什么！？

最著名的例子是 `document.all`：一个由DOM（不是JS引擎本身）给你的JS程序提供的类数组（对象），它向你的JS程序暴露你页面上的元素。它曾经像一个普通对象那样动作——是一个**truthy**。但不再是了。

`document.all` 本身从来就不是“标准的”，而且从很早以前就被废弃/抛弃了。

“那他们就不能删掉它吗？”对不起，想得不错。但愿它们能。但是世面上有太多的遗产JS代码库依赖于它。

那么，为什么使它像**falsy**一样动作？因为从 `document.all` 到 `boolean` 的强制转换（比如在 `if` 语句中）几乎总是用来检测老的，非标准的IE。

IE从很早以前就开始顺应规范了，而且在许多情况下它在推动web向前发展的作用和其他浏览器一样多，甚至更多。但是所有那些老旧的 `if (document.all) { /* it's IE */ }` 代码依然留在世面上，而且大多数可能永远都不会消失。所有这些遗产代码依然假设它们运行在那些给IE用户带来差劲儿的浏览体验的，几十年前的老IE上，

所以，我们不能完全移除 `document.all`，但是IE不再想让 `if (document.all) { .. }` 代码继续工作了，这样现代IE的用户就能得到新的，符合标准的代码逻辑。

“我们应当怎么做？”“我知道了！让我们黑进JS的类型系统并假装 `document.all` 是**falsy**！”

呃。这很烂。这是一个大多数JS开发者们都不理解的疯狂的坑。但是其它的替代方案（对上面两败俱伤的问题什么都不做）还要烂得多那么一点点。

所以……这就是我们得到的：由浏览器给JavaScript添加的疯狂，非标准的“**falsy**对象”。耶！

## Truthy值

回到truthy列表。到底什么是truthy值？记住：如果一个值不在falsy列表中，它就是truthy。

考虑下面代码：

```
var a = "false";
var b = "0";
var c = "";

var d = Boolean( a && b && c );

d;
```

你期望这里的d是什么值？它要么是true要么是false。

它是true。为什么？因为尽管这些string值的内容看起来是falsy值，但是string值本身都是truthy，而这是因为在falsy列表中""是唯一的string值。

那么这些呢？

```
var a = [];           // 空数组 -- truthy 还是 falsy?
var b = {};          // 空对象 -- truthy 还是 falsy?
var c = function(){}; // 空函数 -- truthy 还是 falsy?

var d = Boolean( a && b && c );

d;
```

是的，你猜到了，这里的d依然是true。为什么？和前面的原因一样。尽管它们看起来像，但是[]，{}，和function(){}不在falsy列表中，因此它们是truthy值。

换句话说，truthy列表是无限长的。不可能制成一个这样的列表。你只能制造一个falsy列表并查询它。

花五分钟，把falsy列表写在便利贴上，然后粘在你的电脑显示器上，或者如果你愿意就记住它。不管哪种方法，你都可以在自己需要的时候通过简单地查询一个值是否在falsy列表中，来构建一个虚拟的truthy列表。

truthy和falsy的重要性在于，理解如果一个值在被（明确地或隐含地）强制转换为boolean值的话，它将如何动作。现在你的大脑中有了这两个列表，我们可以深入强制转换的例子本身了。

## 明确的强制转换

明确的 强制转换指的是明显且明确的类型转换。对于大多数开发者来说，有很多类型转换的用法可以清楚地归类于这种 明确的 强制转换。

我们在这里的目标是，在我们的代码中指明一些模式，在这些模式中我们可以清楚明白地将一个值从一种类型转换至另一种类型，以确保不给未来将读到这段代码的开发者留下任何坑。我们越明确，后来的人就越容易读懂我们的代码，也不必费太多的力气去理解我们的意图。

关于 明确的 强制转换可能很难找到什么主要的不同意见，因为它与被广泛接受的静态类型语言中的类型转换的工作方式非常接近。因此，我们理所当然地认为（暂且） 明确的 强制转换可以被认同为不是邪恶的，或没有争议的。虽然我们稍后会回到这个话题。

## 明确地：**Strings <--> Numbers**

我们将从最简单，也许是最常见强制转换操作开始：将值在 `string` 和 `number` 表现形式之间进行强制转换。

为了在 `string` 和 `number` 之间进行强制转换，我们使用内建的 `String(..)` 和 `Number(..)` 函数（我们在第三章中所指的“原生构造器”），但 非常重要的是，我们不在它们前面使用 `new` 关键字。这样，我们就不是在创建对象包装器。

取而代之的是，我们实际上在两种类型之间进行 明确地 强制转换：

```
var a = 42;
var b = String(a);

var c = "3.14";
var d = Number(c);

b; // "42"
d; // 3.14
```

`String(..)` 使用早先讨论的 `ToString` 操作的规则，将任意其它的值强制转换为一个基本类型的 `string` 值。`Number(..)` 使用早先讨论过的 `ToNumber` 操作的规则，将任意其他的值强制转换为一个基本类型的 `number` 值。

我称此为 明确的 强制转换是因为，一般对于大多数开发者来说这是十分明显的：这些操作的最终结果是适当的类型转换。

实际上，这种用法看起来与其他的静态类型语言中的用法非常相像。

举个例子，在C/C++中，你既可以说 `(int)x` 也可以说 `int(x)`，而且它们都将 `x` 中的值转换为一个整数。两种形式都是合法的，但是许多人偏向于后者，它看起来有点儿像一个函数调用。在JavaScript中，当你说 `Number(x)` 时，它看起来极其相似。在JS中它实际上是一个函数调用这个事实重要吗？并非如此。

除了 `String(..)` 和 `Number(..)`，还有其他的方法可以把这些值在 `string` 和 `number` 之间进行“明确地”转换：

```
var a = 42;
var b = a.toString();

var c = "3.14";
var d = +c;

b; // "42"
d; // 3.14
```

调用 `a.toString()` 在表面上是明确的（“`toString`”意味着“变成一个字符串”是很明白的），但是这里有一些藏起来的隐含性。`toString()` 不能在像 `42` 这样的基本类型值上调用。所以JS会自动地将 `42` “封箱”在一个对象包装器中（见第三章），这样 `toString()` 就可以针对这个对象调用。换句话讲，你可能会叫它“明确的隐含”。

这里的 `+c` 是 `+` 操作符的一元操作符（操作符只有一个操作数）形式。取代进行数学加法（或字符串连接——见下面的讨论）的是，一元的 `+` 明确地将它的操作数（`c`）强制转换为一个 `number` 值。

`+c` 是明确的强制转换吗？这要看你的经验和角度。如果你知道（现在你知道了！）一元 `+` 明确地意味着 `number` 强制转换，那么它就是相当明确和明显的。但是，如果你以前从没见过它，那么它看起来就极其困惑，晦涩，带有隐含的副作用，等等。

注意：在开源的JS社区中一般被接受的观点是，一元 `+` 是一个明确的强制转换形式。

即使你真的喜欢 `+c` 这种形式，它绝对会在有的地方看起来非常令人困惑。考虑下面的代码：

```
var c = "3.14";
var d = 5 + +c;

d; // 8.14
```

一元 `-` 操作符也像 `+` 一样进行强制转换，但它还会翻转数字的符号。但是你不能放两个减号 `--` 来使符号翻转回来，因为那将被解释为递减操作符。取代它的是，你需要这么做：`--` `"3.14"`，在两个减号之间加入空格，这将会使强制转换的结果为 `3.14`。

你可能会想到所有种类的可怕组合——一个二元操作符挨着另一个操作符的一元形式。这里有另一个疯狂的例子：

```
1 + - + + + - + 1; // 2
```

当一个一元 `+`（或 `-`）紧邻其他操作符时，你应当强烈地考虑避免使用它。虽然上面的代码可以工作，但几乎全世界都认为它是一个坏主意。即使是 `d = +c`（或者 `d =+ c !`）都太容易与 `d += c` 像混淆了，而后者完全是不同的东西！

注意：一元 `+` 的另一个极端使人困惑的地方是，被用于紧挨着另一个将要作为 `++` 递增操作符和 `--` 递减操作符的操作数。例如：`a +++b`，`a + ++b`，和 `a + + +b`。更多关于 `++` 的信息，参见第五章的“表达式副作用”。

记住，我们正努力变得明确并减少困惑，不是把事情弄得更糟！

## 从 `Date` 到 `number`

另一个一元 `+` 操作符的常见用法是将一个 `Date` 对象强制转换为一个 `number`，其结果是这个日期/时间值的unix时间戳（从世界协调时间的1970年1月1日0点开始计算，经过的毫秒数）表现形式：

```
var d = new Date("Mon, 18 Aug 2014 08:53:06 CDT");
+d; // 1408369986000
```

这种习惯性用法经常用于取得当前的现在时刻的时间戳，比如：

```
var timestamp = +new Date();
```

注意：一些开发者知道一个JavaScript中的特别的语法“技巧”，就是在构造器调用（一个带有 `new` 的函数调用）中如果没有参数值要传递的话，`()` 是可选的。所以你可能遇到 `var timestamp = +new Date;` 形式。然而，不是所有的开发者都同意忽略 `()` 可以增强可读性，因为它是一种不寻常的语法特例，只能适用于 `new fn()` 调用形式，而不能用于普通的 `fn()` 调用形式。

但强制转换不是从 `Date` 对象中取得时间戳的唯一方法。一个不使用强制转换的方式可能更好，因为它更加明确：

```
var timestamp = new Date().getTime();
// var timestamp = (new Date()).getTime();
// var timestamp = (new Date).getTime();
```

但是一个更更好的不使用强制转换的选择是使用ES5加入的 `Date.now()` 静态函数：

```
var timestamp = Date.now();
```

而且如果你想要为老版本的浏览器填补 `Date.now()` 的话，也十分简单：

```

if (!Date.now) {
  Date.now = function() {
    return +new Date();
  };
}

```

我推荐跳过与日期有关的强制转换形式。使用 `Date.now()` 来取得当前现在的时间戳，而使用 `new Date( ... ).getTime()` 来取得一个需要你指定的非现在日期/时间的时间戳。

## 奇异的 ~

一个经常被忽视并通常让人糊涂的JS强制操作符是波浪线 ~ 操作符（也叫“按位取反”，“比特非”）。许多理解它在做什么的人也总是想要避开它。但是为了坚持我们在本书和本系列中的精神，让我们深入并找出 ~ 是否有一些对我们有用的东西。

在第二章的“32位（有符号）整数”一节，我们讲解了在JS中位操作符是如何仅为32位操作定义的，这意味着我们强制它们的操作数遵循32位值的表现形式。这个规则如何发生是由 `ToInt32` 抽象操作（ES5语言规范，9.5部分）控制的。

`ToInt32` 首先进行 `ToNumber` 强制转换，这就是说如果值是 "123"，它在 `ToInt32` 规则实施之前会首先变成 123。

虽然它本身没有技术上进行强制转换（因为类型没有改变），但对一些特定的特殊 `number` 值使用位操作符（比如 | 或 ~）会产生一种强制转换效果，这种效果的结果是一个不同的 `number` 值。

举例来说，让我们首先考虑惯用的空操作 `0 | x`（在第二种章有展示）中使用的 | “比特或”操作符，它实质上仅仅进行 `ToInt32` 转换：

```

0 | -0;           // 0
0 | NaN;          // 0
0 | Infinity;    // 0
0 | -Infinity;   // 0

```

这些特殊的数字是不可用32位表现的（因为它们源自64位的IEEE 754标准——见第二章），所以 `ToInt32` 将这些值的结果指定为 0。

有争议的是，`0 | __` 是否是一种 `ToInt32` 强制转换操作的明确的形式，还是更倾向于隐含。从语言规范的角度来说，毫无疑问是明确的，但是如果你没有在这样的层次上理解位操作，它就可能看起来有点像隐含的魔法。不管怎样，为了与本章中其他的断言保持一致，我们称它为 明确的。

那么，让我们把注意力转回 ~。~ 操作符首先将值“强制转换”为一个32位 `number` 值，然后实施按位取反（翻转每一个比特位）。

注意：这与 `!` 不仅强制转换它的值为 `boolean` 而且还翻转它的每一位很相似（见后面关于“一元 `!`”的讨论）。

但是……什么！？为什么我们要关心被翻转的比特位？这是一些相当特殊的，微妙的东西。JS开发者需要推理个别比特位是十分少见的。

另一种考虑 `-` 定义的方法是，`-` 源自学校中的计算机科学/离散数学：`-` 进行二进制取补操作。太好了，谢谢，我完全明白了！

我们再试一次：`-x` 大致与 `-(x+1)` 相同。这很奇怪，但是稍微容易推理一些。所以：

```
~42; // -(42+1) ==> -43
```

你可能还在想 `-` 这个鬼东西到底和什么有关，或者对于强制转换的讨论它究竟有什么要緊。让我们快速进入要点。

考虑一下 `-(x+1)`。通过进行这个操作，能够产生结果 `0`（或者从技术上说 `-0 !`）的唯一的值是什么？`-1`。换句话说，`-` 用于一个范围的 `number` 值时，将会为输入值 `-1` 产生一个 `falsy`（很容易强制转换为 `false`）的 `0`，而为任意其他的输入产生`truthy`的 `number`。

为什么这要緊？

`-1` 通常称为一个“哨兵值”，它基本上意味着一个在同类型值（`number`）的更大的集合中被赋予了任意的语义。在C语言中许多函数使用哨兵值 `-1`，它们返回 `>= 0` 的值表示“成功”，返回 `-1` 表示“失败”。

JavaScript在定义 `string` 操作 `indexOf(..)` 时采纳了这种先例，它搜索一个子字符串，如果找到就返回它从0开始计算的索引位置，没有找到的话就返回 `-1`。

这样的情况很常见：不仅仅将 `indexOf(..)` 作为取得位置的操作，而且作为检查一个子字符串存在/不存在于另一个 `string` 中的 `boolean` 值。这就是开发者们通常如何进行这样的检查：

```
var a = "Hello World";

if (a.indexOf( "lo" ) >= 0) {    // true
    // 找到了!
}

if (a.indexOf( "lo" ) != -1) {    // true
    // 找到了
}

if (a.indexOf( "ol" ) < 0) {    // true
    // 没找到!
}

if (a.indexOf( "ol" ) == -1) {    // true
    // 没找到!
}
```

我感觉看着 `>= 0` 或 `== -1` 有些恶心。它基本上是一种“抽象泄漏”，这里它将底层的实现行为——使用哨兵值 `-1` 表示“失败”——泄漏到我的代码中。我倒是乐意隐藏这样的细节。

现在，我们终于看到为什么 `~` 可以帮到我们了！将 `~` 和 `indexOf()` 一起使用可以将值“强制转换”（实际上只是变形）为可以适当地强制转换为 `boolean` 的值：

```
var a = "Hello World";

~a.indexOf( "lo" );           // -4    <-- truthy!

if (~a.indexOf( "lo" )) {     // true
    // 找到了!
}

~a.indexOf( "ol" );           // 0     <-- falsy!
!~a.indexOf( "ol" );          // true

if (!~a.indexOf( "ol" )) {    // true
    // 没找到!
}
```

`~` 拿到 `indexOf(..)` 的返回值并将它变形：对于“失败”的 `-1` 我们得到 `falsy` 的 `0`，而其他的值都是 `truthy`。

注意：`~` 的假想算法 `-(x+1)` 暗示着 `~-1` 是 `-0`，但是实际上它产生 `0`，因为底层的操作其实是按位的，不是数学操作。

技术上讲，`if (~a.indexOf(..))` 仍然依靠隐含的强制转换将它的结果 `0` 变为 `false` 或非零变为 `true`。但总的来说，对我而言 `~` 更像一种明确的强制转换机制，只要你知道在这种惯用法中它的意图是什么。

我感觉这样的代码要比前面凌乱的 `>= 0 / == -1` 更干净。

### 截断比特位

在你遇到的代码中，还有一个地方可能出现 `~`：一些开发者使用双波浪线 `~~` 来截断一个 `number` 的小数部分（也就是，将它“强制转换”为一个“整数”）。这通常（虽然是错误的）被说成与调用 `Math.floor(..)` 的结果相同。

`~~` 的工作方式是，第一个 `~` 实施 `ToInt32` “强制转换”并进行按位取反，然后第二个 `~` 进行另一次按位取反，将每一个比特位都翻转回原来的状态。于是最终的结果就是 `ToInt32` “强制转换”（也叫截断）。

注意：`~~` 的按位双翻转，与双否定 `!!` 的行为非常相似，它将在稍后的“明确地：`* --> Boolean`”一节中讲解。

然而，`~~` 需要一些注意/澄清。首先，它仅在32位值上可以可靠地工作。但更重要的是，它在负数上工作的方式与 `Math.floor(..)` 不同！

```
Math.floor( -49.6 );      // -50
~~-49.6;                  // -49
```

把 `Math.floor(..)` 的不同放在一边，`~~x` 可以将值截断为一个（32位）整数。但是 `x | 0` 也可以，而且看起来还（稍微）省事儿一些。

那么，为什么你可能会选择 `~~x` 而不是 `x | 0`？操作符优先权（见第五章）：

```
~~1E20 / 10;           // 166199296
1E20 | 0 / 10;         // 1661992960
(1E20 | 0) / 10;       // 166199296
```

正如这里给出的其他建议一样，仅在读/写这样的代码的每一个人都知道这些操作符如何工作的情况下，才将 `~` 和 `~~` 作为“强制转换”和将值变形的明确机制。

## 明确地：解析数字字符串

将一个 `string` 强制转换为一个 `number` 的类似结果，可以通过从 `string` 的字符内容中解析（parsing）出一个 `number` 得到。然而在这种解析和我们上面讲解的类型转换之间存在着区别。

考虑下面的代码：

```
var a = "42";
var b = "42px";

Number( a );    // 42
parseInt( a );  // 42

Number( b );    // NaN
parseInt( b );  // 42
```

从一个字符串中解析出一个数字是容忍非数字字符的——从左到右，如果遇到非数字字符就停止解析——而强制转换是不容忍并且会失败而得出值 `Nan`。

解析不应当被视为强制转换的替代品。这两种任务虽然相似，但是有着不同的目的。当你不知道/不关心右手边可能有什么其他的非数字字符时，你可以将一个 `string` 作为 `number` 解析。当只有数字才是可接受的值，而且像 `"42px"` 这样的东西作为数字应当被排除时，就强制转换一个 `string`（变为一个 `number`）。

提示：`parseInt(..)` 有一个孪生兄弟，`parseFloat(..)`，它（听起来）从一个字符串中拉出一个浮点数。

不要忘了 `parseInt(..)` 工作在 `string` 值上。向 `parseInt(..)` 传递一个 `number` 绝对没有任何意义。传递其他任何类型也都没有意义，比如 `true`，`function(){..}` 或 `[1,2,3]`。

如果你传入一个非 `string`，你所传入的值首先将自动地被强制转换为一个 `string`（见早先的“`ToString`”），这很明显是一种隐藏的隐含强制转换。在你的程序中依赖这样的行为真的是一个坏主意，所以永远也不要将 `parseInt(..)` 与非 `string` 值一起使用。

在ES5之前，`parseInt(..)` 还存在另外一个坑，这曾是许多JS程序的bug的根源。如果你不传递第二个参数来指定使用哪种进制（也叫基数）来翻译数字的 `string` 内容，`parseInt(..)` 将会根据开头的字符进行猜测。

如果开头的两个字符是 `"0x"` 或 `"0X"`，那么猜测（根据惯例）将是你要将这个 `string` 翻译为一个16进制 `number`。否则，如果第一个字符是 `"0"`，那么猜测（也是根据惯例）将是你要将这个 `string` 翻译成8进制 `number`。

16进制的 `string`（以 `0x` 或 `0X` 开头）没那么容易搞混。但是事实证明8进制数字的猜测过于常见了。比如：

```
var hour = parseInt( selectedHour.value );
var minute = parseInt( selectedMinute.value );

console.log( "The time you selected was: " + hour + ":" + minute);
```

看起来无害，对吧？试着在小时上选择 `08` 在分钟上选择 `09`。你会得到 `0:0`。为什么？因为 `8` 和 `9` 都不是合法的8进制数。

ES5之前的修改很简单，但是很容易忘：总是在第二个参数值上传递 `10`。这完全是安全的：

```
var hour = parseInt( selectedHour.value, 10 );
var minute = parseInt( selectedMinute.value, 10 );
```

在ES5中，`parseInt(..)` 不再猜测八进制数了。除非你指定，否则它会假定为10进制（或者为 `"0x"` 前缀猜测16进制数）。这好多了。只是要小心，如果你的代码不得不运行在前ES5环境中，你仍然需要为基数传递 `10`。

## 解析非字符串

几年以前有一个挖苦JS的玩笑，使一个关于 `parseInt(..)` 行为的一个臭名昭著的例子备受关注，它取笑JS的这个行为：

```
parseInt( 1/0, 19 ); // 18
```

这里面设想（但完全不合法）的断言是，“如果我传入一个无限大，并从中解析出一个整数的话，我应该得到一个无限大，不是18”。没错，JS一定是疯了才得出这个结果，对吧？

虽然这是个明显故意造成的，不真实的例子，但是让我们放纵这种疯狂一小会儿，来检视一下JS是否真的那么疯狂。

首先，这其中最明显的原罪是将一个非 string 传入了 `parseInt(..)`。这样做是自找麻烦。但就算你这么做了，JS也会礼貌地将你传入的东西强制转换为它可以解析的 string。

有些人可能会争论说这是一种不合理的行为，`parseInt(..)` 应当拒绝在一个非 string 值上操作。它应该抛出一个错误吗？坦白地说，像Java那样。但是一想到JS应当开始在满世界抛出错误，以至于几乎每一行代码都需要用 `try..catch` 围起来，我就不寒而栗。

它应当返回 `Nan` 吗？也许。但是……要是这样呢：

```
parseInt( new String( "42" ) );
```

这也应当失败吗？它是一个非 string 值啊。如果你想让 `String` 对象包装器被开箱成 `"42"`，那么 `42` 先变成 `"42"`，以使 `42` 可以被解析回来就那么不寻常吗？

我会争论说，这种可能发生的半明确半隐含的强制转换经常可以成为非常有用的东西。比如：

```
var a = {
  num: 21,
  toString: function() { return String( this.num * 2 ); }
};

parseInt( a ); // 42
```

事实上 `parseInt(..)` 将它的值强制转换为 `string` 来实施解析是十分合理的。如果你传垃圾进去，那么你就会得到垃圾，不要责备垃圾桶——它只是忠实地尽自己的责任。

那么，如果你传入像 `Infinity`（很明显是 `1 / 0` 的结果）这样的值，对于它的强制转换来说哪种 `string` 表现形式最有道理呢？我脑中只有两种合理的选择：`"Infinity"` 和 `"∞"`。JS选择了 `"Infinity"`。我很高兴它这么选。

我认为在JS中所有的值都有某种默认的 `string` 表现形式是一件好事，这样它们就不是我们不能调试和推理的神秘黑箱了。

现在，关于19进制呢？很明显，这完全是伪命题和造作。没有真实的JS程序使用19进制。那太荒谬了。但是，让我们再一次放任这种荒谬。在19进制中，合法的数字字符是 0 - 9 和 a - i （大小写无关）。

那么，回到我们的 `parseInt( 1/0, 19 )` 例子。它实质上是 `parseInt( "Infinity", 19 )`。它如何解析？第一个字符是 "I"，在愚蠢的19进制中是值 18。第二个字符 "n" 不再合法的数字字符集内，所以这样的解析就礼貌地停止了，就像它在 "42px" 中遇到 "p" 那样。

结果呢？18。正如它应该的那样。对JS来说，并非一个错误或者 `Infinity` 本身，而是将我们带到这里的一系列的行为才是非常重要的，不应当那么简单地被丢弃。

其他关于 `parseInt(..)` 行为的，令人吃惊但又十分合理的例子还包括：

```
parseInt( 0.000008 );           // 0   ("0" from "0.000008")
parseInt( 0.0000008 );          // 8   ("8" from "8e-7")
parseInt( false, 16 );          // 250 ("fa" from "false")
parseInt( parseInt, 16 );        // 15  ("f" from "function..")

parseInt( "0x10" );            // 16
parseInt( "103", 2 );          // 2
```

其实 `parseInt(..)` 在它的行为上是相当可预见和一致的。如果你正确地使用它，你就能得到合理的结果。如果你不正确地使用它，那么你得到的疯狂结果并不是JavaScript的错。

## 明确地：`* --> Boolean`

现在，我们来检视从任意的非 `boolean` 值到一个 `boolean` 值的强制转换。

正如上面的 `String(..)` 和 `Number(..)`，`Boolean(..)`（当然，不带 `new !`）是强制进行 `ToBoolean` 转换的明确方法：

```

var a = "0";
var b = [];
var c = {};

var d = "";
var e = 0;
var f = null;
var g;

Boolean( a ); // true
Boolean( b ); // true
Boolean( c ); // true

Boolean( d ); // false
Boolean( e ); // false

Boolean( f ); // false
Boolean( g ); // false

```

虽然 `Boolean(...)` 是非常明确的，但是它并不常见也不为人所惯用。

正如一元 `+` 操作符将一个值强制转换为一个 `number`（参见上面的讨论），一元的 `!` 否定操作符可以将一个值明确地强制转换为一个 `boolean`。问题是它还将值从`truthy`翻转为`falsy`，或反之。所以，大多数JS开发者使用 `!!` 双否定操作符进行 `boolean` 强制转换，因为第二个 `!` 将会把它翻转回原本的`true`或`false`：

```

var a = "0";
var b = [];
var c = {};

var d = "";
var e = 0;
var f = null;
var g;

!!a;      // true
!!b;      // true
!!c;      // true

!!d;      // false
!!e;      // false
!!f;      // false
!!g;      // false

```

没有 `Boolean(...)` 或 `!!` 的话，任何这些 `ToBoolean` 强制转换都将隐含地发生，比如在一个 `if (...) ...` 语句这样使用 `boolean` 的上下文中。但这里的目标是，明确地强制一个值成为 `boolean` 来使 `ToBoolean` 强制转换的意图显得明明白白。

另一个 `ToBoolean` 强制转换的用例是，如果你想在数据结构的JSON序列化中强制转换一个 `true / false`：

```
var a = [
  1,
  function(){ /*..*/ },
  2,
  function(){ /*..*/ }
];

JSON.stringify( a ); // "[1,null,2,null]"

JSON.stringify( a, function(key,val){
  if (typeof val == "function") {
    // 强制函数进行 `ToBoolean` 转换
    return !!val;
  }
  else {
    return val;
  }
} );
// "[1,true,2,true]"
```

如果你是从Java来到JavaScript的话，你可能会认得这个惯用法：

```
var a = 42;

var b = a ? true : false;
```

`? :` 三元操作符将会测试 `a` 的真假，然后根据这个测试的结果相应地将 `true` 或 `false` 赋值给 `b`。

表面上，这个惯用法看起来是一种明确的 `ToBoolean` 类型强制转换形式，因为很明显它操作的结果要么是 `true` 要么是 `false`。

然而，这里有一个隐藏的隐含强制转换，就是表达式 `a` 不得不首先被强制转换为 `boolean` 来进行真假测试。我称这种惯用法为“明确地隐含”。另外，我建议你在JavaScript中完全避免这种惯用法。它不会提供真正的好处，而且会让事情变得更糟。

对于明确的强制转换 `Boolean(a)` 和 `!!a` 是好得多的选项。

## 隐含的强制转换

隐含的强制转换是指这样的类型转换：它们是隐藏的，由于其他动作隐含地发生的不明显的副作用。换句话说，任何（对你）不明显的类型转换都是隐含的强制转换。

虽然明确的强制转换的目的很明白，但是这可能太过明显——隐含的强制转换拥有相反的目的：使代码更难理解。

从表面上来看，我相信这就是许多关于强制转换的愤怒的源头。绝大多数关于“JavaScript强制转换”的抱怨实际上都指向了（不管他们是否理解它）隐含的强制转换。

注意：Douglas Crockford，“*JavaScript: The Good Parts*”的作者，在许多会议和他的作品中声称应当避免JavaScript强制转换。但看起来他的意思是隐含的强制转换是不好的（以他的意见）。然而，如果你读他自己的代码的话，你会发现相当多的强制转换的例子，明确和隐含都有！事实上，他的担忧主要在于`==`操作，但正如你将在本章中看到的，那只是强制转换机制的一部分。

那么，隐含强制转换是邪恶的吗？它很危险吗？它是JavaScript设计上的缺陷吗？我们应该尽一切力量避免它吗？

我打赌大多数读者都倾向于踊跃地欢呼，“是的！”

别那么着急。听我把话说完。

让我们在隐含的强制转换是什么，和可以是什么这个问题上采取一个不同的角度，而不是仅仅说它是“好的明确强制转换的反面”。这太过狭隘，而且忽视了一个重要的微妙细节。

让我们将隐含的强制转换的目的定义为：减少搞乱我们代码的繁冗，模板代码，和/或不必要的实现细节，不使它们的噪音掩盖更重要的意图。

## 用于简化的隐含

在我们进入JavaScript以前，我建议使用某个理论上是强类型的语言的假想代码来说明一下：

```
SomeType x = SomeType( AnotherType( y ) )
```

在这个例子中，我在`y`中有一些任意类型的值，想把它转换为`SomeType`类型。问题是，这种语言不能从当前`y`的类型直接走到`SomeType`。它需要一个中间步骤，它首先转换为`AnotherType`，然后从`AnotherType`转换到`SomeType`。

现在，要是这种语言（或者你可用这种语言创建自己的定义）允许你这么说呢：

```
SomeType x = SomeType( y )
```

难道一般来说你不会同意我们简化了这里的类型转换，降低了中间转换步骤的无谓的“噪音”吗？我的意思是，在这段代码的这一点上，能看到并处理`y`先变为`AnotherType`然后再变为`SomeType`的事实，真的是很重要的一件事吗？

有些人可能会争辩，至少在某些环境下，是的。但我想我可以做出相同的争辩说，在许多其他的环境下，不管是通过语言本身的还是我们自己的抽象，这样的简化通过抽象或隐藏这些细节 确实增强了代码的可读性。

毫无疑问，在幕后的某些地方，那个中间的步骤依然是发生的。但如果这样的细节在视野中隐藏起来，我们就可以将使 `y` 变为类型 `SomeType` 作为一个泛化操作来推理，并隐藏混乱的细节。

虽然不是一个完美的类比，我要在本章剩余部分争论的是，JS的 隐含的 强制转换可以被认为是给你的代码提供了一个类似的辅助。

但是，很重要的是，这不是一个无边界的，绝对的论断。绝对有许多 邪恶的东西 潜伏在 隐含 强制转换周围，它们对你的代码造成的损害要比任何潜在的可读性改善厉害的多。很清楚，我们不得不学习如何避免这样的结构，使我们不会用各种bug来毒害我们的代码。

许多开发者相信，如果一个机制可以做某些有用的事儿 **A**，但也可以被滥用或误用来做某些可怕的事儿 **Z**，那么我们就应当将这种机制整个儿扔掉，仅仅是为了安全。

我对你的鼓励是：不要安心于此。不要“把孩子跟洗澡水一起泼出去”。不要因为你只见到过它的“坏的一面”就假设 隐含 强制转换都是坏的。我认为这里有“好的一面”，而我想要帮助和启发你们更多的人找到并接纳它们！

## 隐含地：**Strings <--> Numbers**

在本章的早先，我们探索了 `string` 和 `number` 值之间的 明确 强制转换。现在，让我们使用 隐含 强制转换的方式探索相同任务。但在我们开始之前，我们不得不检视一些将会 隐含地 发生强制转换的操作的微妙之处。

为了服务于 `number` 的相加和 `string` 的连接两个目的，`+` 操作符被重载了。那么JS如何知道你想用的是哪一种操作呢？考虑下面的代码：

```
var a = "42";
var b = "0";

var c = 42;
var d = 0;

a + b; // "420"
c + d; // 42
```

是什么不同导致了 `"420"` 和 `42` ？一个常见的误解是，这个不同之处在于操作数之一或两者是否是一个 `string`，这意味着 `+` 将假设 `string` 连接。虽然这有一部分是对的，但实际情况要更复杂。

考虑如下代码：

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

a + b; // "1,23,4"
```

两个操作数都不是 `string`，但很明显它们都被强制转换为 `string` 然后启动了 `string` 连接。那么到底发生了什么？

(警告：语言规范式的深度细节就要来了，如果这会吓到你就跳过下面两段！)

根据ES5语言规范的11.6.1部分，`+` 的算法是（当一个操作数是 `object` 值时），如果两个操作数之一已经是一个 `string`，或者下列步骤产生一个 `string` 表达形式，`+` 将会进行连接。所以，当 `+` 的两个操作数之一收到一个 `object`（包括 `array`）时，它首先在这个值上调用 `ToPrimitive` 抽象操作（9.1部分），而它会带着 `number` 的上下文环境提示来调用 `[[DefaultValue]]` 算法（8.12.8部分）。

如果你仔细观察，你会发现这个操作现在和 `ToNumber` 抽象操作处理 `object` 的过程是一样的（参见早先的“`ToNumber`”一节）。在 `array` 上的 `valueOf()` 操作将会在产生一个简单基本类型时失败，于是它退回到一个 `toString()` 表现形式。两个 `array` 因此分别变成了 `"1,2"` 和 `"3,4"`。现在，`+` 就如你通常期望的那样连接这两个 `string`：`"1,23,4"`。

让我们把这些乱七八糟的细节放在一边，回到一个早前的，简化的解释：如果 `+` 的两个操作数之一是一个 `string`（或在上面的步骤中成为一个 `string`），那么操作就会是 `string` 连接。否则，它总是数字加法。

注意：关于强制转换，一个经常被引用的坑是 `[] + {}` 和 `{ } + []`，这两个表达式的结果分别是 `"[object Object]"` 和 `0`。虽然对此有更多的东西，但是我们将在第五章的“Block”中讲解这其中的细节。

这对隐含强制转换意味着什么？

你可以简单地通过将 `number` 和空 `string`""`` “相加”来把一个 `number` 强制转换为一个 `string`：

```
var a = 42;
var b = a + ``;

b; // "42"
```

提示：使用 `+` 操作符的数字加法是可交换的，这意味着 `2 + 3` 与 `3 + 2` 是相同的。使用 `+` 的字符串连接很明显通常不是可交换的，但是对于 `""` 的特定情况，它实质上是可交换的，因为 `a + ""` 和 `"" + a` 会产生相同的结果。

使用一个 `+ ""` 操作将 `number`（隐含地）强制转换为 `string` 是极其常见/惯用的。事实上，有趣的是，一些在口头上批评隐含强制转换得最严厉的人仍然在他们自己的代码中使用这种方式，而不是使用它的明确的替代形式。

在隐含强制转换的有用形式中，我认为这是一个很棒的例子，尽管这种机制那么频繁地被人诟病！

将 `a + ""` 这种隐含的强制转换与我们早先的 `String(a)` 明确的强制转换的例子相比较，有一个另外的需要小心的奇怪之处。由于 `ToPrimitive` 抽象操作的工作方式，`a + ""` 在值 `a` 上调用 `valueOf()`，它的返回值再最终通过内部的 `ToString` 抽象操作转换为一个 `string`。但是 `String(a)` 只直接调用 `toString()`。

两种方式的最终结果都是一个 `string`，但如果你使用一个 `object` 而不是一个普通的基本类型 `number` 的值，你可能不一定得到相同的 `string` 值！

考虑这段代码：

```
var a = {
  valueOf: function() { return 42; },
  toString: function() { return 4; }
};

a + "";           // "42"

String( a );    // "4"
```

一般来说这样的坑不会咬到你，除非你真的试着创建令人困惑的数据结构和操作，但如果你为某些 `object` 同时定义了你自己的 `valueOf()` 和 `toString()` 方法，你就应当小心，因为你强制转换这些值的方式将影响到结果。

那么另外一个方向呢？我们如何将一个 `string` 隐含强制转换为一个 `number`？

```
var a = "3.14";
var b = a - 0;

b; // 3.14
```

`-` 操作符是仅为数字减法定义的，所以 `a - 0` 强制 `a` 的值被转换为一个 `number`。虽然少见得多，`a * 1` 或 `a / 1` 也会得到相同的结果，因为这些操作符也是仅为数字操作定义的。

那么对 `-` 操作符使用 `object` 值会怎样呢？和上面的 `+` 的故事相似：

```
var a = [3];
var b = [1];

a - b; // 2
```

两个 `array` 值都不得不变为 `number`，但它们首先会被强制转换为 `string`（使用意料之中的 `toString()` 序列化），然后再强制转换为 `number`，以便 `-` 减法操作可以实施。

那么，`string` 和 `number` 值之间的隐含强制转换还是你总是在恐怖故事当中听到的丑陋怪物吗？我个人不这么认为。

比较 `b = String(a)`（明确的）和 `b = a + ""`（隐含的）。我认为在你的代码中会出现两种方式都有用的情况。当然 `b = a + ""` 在JS程序中更常见一些，不管一般意义上隐含强制转换的好处或害处的感觉如何，它都提供了自己的用途。

## 隐含地：**Booleans --> Numbers**

我认为隐含强制转换可以真正闪光的一个情况是，将特定类型的复杂 `boolean` 逻辑简化为简单的数字加法。当然，这不是一个通用的技术，而是一个特定情况的特定解决方法。

考虑如下代码：

```
function onlyOne(a, b, c) {
  return !!((a && !b && !c) ||
            (!a && b && !c) || (!a && !b && c));
}

var a = true;
var b = false;

onlyOne( a, b, b );    // true
onlyOne( b, a, b );    // true

onlyOne( a, b, a );    // false
```

这个 `onlyOne(...)` 工具应当仅在正好有一个参数是 `true /truthy` 时返回 `true`。它在 `truthy` 的检查上使用隐含的强制转换，而在其他的地方使用明确的强制转换，包括最后的返回值。

但如果我们要这个工具能够以相同的方式处理四个，五个，或者二十个标志值呢？很难想象处理所有那些比较的排列组合的代码实现。

但这里是 `boolean` 值到 `number`（很明显，`0` 或 `1`）的强制转换可以提供巨大帮助的地方：

```

function onlyOne() {
  var sum = 0;
  for (var i=0; i < arguments.length; i++) {
    // 跳过falsy值。与将它们视为0相同，但是避开NaN
    if (arguments[i]) {
      sum += arguments[i];
    }
  }
  return sum == 1;
}

var a = true;
var b = false;

onlyOne( b, a );           // true
onlyOne( b, a, b, b, b ); // true

onlyOne( b, b );           // false
onlyOne( b, a, b, b, b, a ); // false

```

注意：当然，除了在 `onlyOne(..)` 中的 `for` 循环，你可以更简洁地使用ES5的 `reduce(..)` 工具，但我不想因此而模糊概念。

我们在这里做的事情有赖于 `true /truthy` 的强制转换结果为 `1`，并将它们作为数字加起来。`sum += arguments[i]` 通过隐含的强制转换使这发生。如果在 `arguments` 列表中有且仅有一个值为 `true`，那么这个数字的和将是 `1`，否则和就不是 `1` 而不能使期望的条件成立。

我们当然本可以使用明确的强制转换：

```

function onlyOne() {
  var sum = 0;
  for (var i=0; i < arguments.length; i++) {
    sum += Number( !!arguments[i] );
  }
  return sum === 1;
}

```

我们首先使用 `!!arguments[i]` 来将这个值强制转换为 `true` 或 `false`。这样你就可以像 `onlyOne( "42", 0 )` 这样传入非 `boolean` 值了，而且它依然可以如意料的那样工作（要不然，你将会得到 `string` 连接，而且逻辑也不正确）。

一旦我们确认它是一个 `boolean`，我们就使用 `Number(..)` 进行另一个明确的强制转换来确保值是 `0` 或 `1`。

这个工具的明确强制转换形式“更好”吗？它确实像代码注释中解释的那样避开了 `NaN` 的陷阱。但是，这最终要看你的需要。我个人认为前一个版本，依赖于隐含的强制转换更优雅（如果你不传入 `undefined` 或 `NaN`），而明确的版本是一种不必要的繁冗。

但与我们在这里讨论的几乎所有东西一样，这是一个主观判断。

注意：不管是隐含的还是明确的方式，你可以通过将最后的比较从 `1` 改为 `2` 或 `5`，来分别很容易地制造 `onlyTwo(..)` 或 `onlyFive(..)`。这要比添加一大堆 `&&` 和 `||` 表达式要简单太多了。所以，一般来说，在这种情况下强制转换非常有用。

## 隐含地：`* --> Boolean`

现在，让我们将注意力转向目标为 `boolean` 值的隐含强制转换上，这是目前最常见，并且还是目前潜在的最麻烦的一种。

记住，隐含的强制转换是当你以强制一个值被转换的方式使用这个值时才启动的。对于数字和 `string` 操作，很容易就能看出这种强制转换是如何发生的。

但是，哪个种类的表达式操作（隐含地）要求/强制一个 `boolean` 转换呢？

1. 在一个 `if (...)` 语句中的测试表达式。
2. 在一个 `for (... ; ... ; ...)` 头部的测试表达式（第二个子句）。
3. 在 `while (...)` 和 `do..while(..)` 循环中的测试表达式。
4. 在 `? :` 三元表达式中的测试表达式（第一个子句）。
5. `||`（“逻辑或”）和 `&&`（“逻辑与”）操作符左手边的操作数（它用作测试表达式——见下面的讨论！）。

在这些上下文环境中使用的，任何还不是 `boolean` 的值，将通过本章早先讲解的 `ToBoolean` 抽象操作的规则，被隐含地强制转换为一个 `boolean`。

我们来看一些例子：

```

var a = 42;
var b = "abc";
var c;
var d = null;

if (a) {
    console.log( "yep" );           // yep
}

while (c) {
    console.log( "nope, never runs" );
}

c = d ? a : b;                   // "abc"

if ((a && d) || c) {
    console.log( "yep" );           // yep
}

```

在所有这些上下文环境中，非 `boolean` 值被隐含地强制转换为它们的 `boolean` 等价物，来决定测试的结果。

## || 和 && 操作符

很可能你已经在你用过的大多数或所有其他语言中见到过 `||`（“逻辑或”）和 `&&`（“逻辑与”）操作符了。所以假设它们在JavaScript中的工作方式和其他类似的语言基本上相同是很自然的。

这里有一个鲜为人知的，但很重要的，微妙细节。

其实，我会争辩这些操作符甚至不应当被称为“逻辑`_`操作符”，因为这样的名称没有完整地描述它们在做什么。如果让我给它们一个更准确的（也更蹩脚的）名称，我会叫它们“选择器操作符”或更完整的，“操作数选择器操作符”。

为什么？因为在JavaScript中它们实际上不会得出一个 `逻辑值`（也就是 `boolean`），这与它们在其他语言中的表现不同。

那么它们到底得出什么？它们得出两个操作数中的一个（而且仅有一个）。换句话说，它们在两个操作数的值中选择一个。

引用ES5语言规范的11.11部分：

一个`&&`或`||`操作符产生的值不见得是Boolean类型。这个产生的值将总是两个操作数表达式其中之一的值。

让我们展示一下：

```
var a = 42;
var b = "abc";
var c = null;

a || b;          // 42
a && b;          // "abc"

c || b;          // "abc"
c && b;          // null
```

等一下，什么！？想一想。在像C和PHP这样的语言中，这些表达式结果为 `true` 或 `false`，而在JS中（就此而言还有Python和Ruby！），结果来自于值本身。

`||` 和 `&&` 操作符都在第一个操作数（`a` 或 `c`）上进行 `boolean` 测试。如果这个操作数还不是 `boolean`（就像在这里一样），就会发生一次普通的 `ToBoolean` 强制转换，这样测试就可以进行了。

对于 `||` 操作符，如果测试结果为 `true`，`||` 表达式就将第一个操作数的值（`a` 或 `c`）作为结果。如果测试结果为 `false`，`||` 表达式就将第二个操作数的值（`b`）作为结果。

相反地，对于 `&&` 操作符，如果测试结果为 `true`，`&&` 表达式将第二个操作数的值（`b`）作为结果。如果测试结果为 `false`，那么 `&&` 表达式就将第一个操作数的值（`a` 或 `c`）作为结果。

`||` 或 `&&` 表达式的结果总是两个操作数之一的底层值，不是（可能是被强制转换来的）测试的结果。在 `c && b` 中，`c` 是 `null`，因此是 `falsy`。但是 `&&` 表达式本身的结果为 `null`（`c` 中的值），不是用于测试的强制转换来的 `false`。

现在你明白这些操作符如何像“操作数选择器”一样工作了吗？

另一种考虑这些操作数的方式是：

```
a || b;
// 大体上等价于：
a ? a : b;

a && b;
// 大体上等价于：
a ? b : a;
```

注意：我说 `a || b` “大体上等价”于 `a ? a : b`，是因为虽然结果相同，但是这里有一个微妙的不同。在 `a ? a : b` 中，如果 `a` 是一个更复杂的表达式（例如像调用 `function` 那样可能带有副作用），那么这个表达式 `a` 将有可能被求值两次（如果第一次求值的结果为 `truthy`）。相比之下，对于 `a || b`，表达式 `a` 仅被求值一次，而且这个值将被同时用于强制转换测试和结果值（如果合适的话）。同样的区别也适用于 `a && b` 和 `a ? b : a` 表达式。

很有可能你在没有完全理解之前你就已经使用了这个行为的一个极其常见，而且很有帮助的用法：

```
function foo(a,b) {
  a = a || "hello";
  b = b || "world";

  console.log( a + " " + b );
}

foo();           // "hello world"
foo( "yeah", "yeah!" ); // "yeah yeah!"
```

这种 `a = a || "hello"` 惯用法（有时被说成 C#“null合并操作符”的 JavaScript 版本）对 `a` 进行测试，如果它没有值（或仅仅是一个不期望的 `falsy` 值），就提供一个后备的默认值（`"hello"`）。

但是要小心！

```
foo( "That's it!", "" ); // "That's it! world" <-- Oops!
```

看到问题了吗？作为第二个参数的 `""` 是一个 **falsy** 值（参见本章早先的 `ToBoolean`），所以 `b = b || "world"` 测试失败，而默认值 `"world"` 被替换上来，即便本来的意图可能是想让明确传入的 `""` 作为赋给 `b` 的值。

这种 `||` 惯用法极其常见，而且十分有用，但是你不得不只在所有的 **falsy** 值应当被跳过时使用它。不然，你就需要在你的测试中更加具体，而且可能应该使用一个 `? :` 三元操作符。

这种默认值赋值惯用法是如此常见（和有用！），以至于那些公开激烈诽谤 JavaScript 强制转换的人都经常在它们的代码中使用！

那么 `&&` 呢？

有另一种在手动编写中不那么常见，而在 JS 压缩器中频繁使用的惯用法。`&&` 操作符会“选择”第二个操作数，当且仅当第一个操作数测试为 **truthy**，这种用法有时被称为“守护操作符”（参见第五章的“短接”）——第一个表达式的测试“守护”着第二个表达式：

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

var a = 42;

a && foo(); // 42
```

`foo()` 仅在 `a` 测试为 **truthy** 时会被调用。如果这个测试失败，这个 `a && foo()` 表达式语句将会无声地停止——这被称为“短接”——而且永远不会调用 `foo()`。

重申一次，几乎很少有人手动编写这样的东西。通常，他们会写 `if (a) { foo(); }`。但是 JS 压缩器选择 `a && foo()` 是因为它短的多。所以，现在，如果你不得不解读这样的代码，你就知道它是在做什么以及为什么了。

好了，那么 `||` 和 `&&` 在它们的功能上有些不错的技巧，只要你乐意让隐含的强制转换掺和进来。

注意：`a = b || "something"` 和 `a && b()` 两种惯用法都依赖于短接行为，我们将在第五章中讲述它的细节。

现在，这些操作符实际上不会得出 `true` 和 `false` 的事实可能使你的头脑有点儿混乱。你可能想知道，如果你的 `if` 语句和 `for` 循环包含 `a && (b || c)` 这样的复合的逻辑表达式，它们到底都是怎么工作的。

别担心！天没塌下来。你的代码（可能）没有问题。你只是可能从来没有理解在这个符合表达式被求值之后，有一个向 `boolean` 隐含的强制转换发生了。

考虑这段代码：

```
var a = 42;
var b = null;
var c = "foo";

if (a && (b || c)) {
    console.log( "yep" );
}
```

这段代码将会像你总是认为的那样工作，除了一个额外的微妙细节。`a && (b || c)` 的结果实际上 是 `"foo"`，不是 `true`。所以，这之后 `if` 语句强制值 `"foo"` 转换为一个 `boolean`，这理所当然地将是 `true`。

看到了？没有理由惊慌。你的代码可能依然是安全的。但是现在关于它在做什么和如何做，你知道了更多。

而且现在你理解了这样的代码使用 隐含的 强制转换。如果你依然属于“避开（隐含）强制转换阵营”，那么你就需要退回去并使所有这些测试明确：

```
if (!!a && (!!b || !!c)) {
    console.log( "yep" );
}
```

祝你好运！...对不起，只是逗个乐儿。

## Symbol 强制转换

在此为止，在 明确的 和 隐含的 强制转换之间几乎没有可以观察到的结果上的不同——只有代码的可读性至关重要。

但是ES6的Symbol在强制转换系统中引入了一个我们需要简单讨论的坑。由于一个明显超出了我们将在本书中讨论的范围的原因，从一个 `symbol` 到一个 `string` 的 明确 强制转换是允许的，但是相同的 隐含 强制转换是不被允许的，而且会抛出一个错误。

考虑如下代码：

```
var s1 = Symbol( "cool" );
String( s1 );                      // "Symbol(cool)"

var s2 = Symbol( "not cool" );
s2 + "";                           // TypeError
```

`symbol` 值根本不能强制转换为 `number`（不论哪种方式都抛出错误），但奇怪的是它们既可以明确地也可以隐含地强制转换为 `boolean`（总是 `true`）。

一致性总是容易学习的，而对付例外从来就不有趣，但是我们只需要在ES6 `symbol` 值和我们如何强制转换它们的问题上多加小心。

好消息：你需要强制转换一个 `symbol` 值的情况可能极其少见。它们典型的被使用的方式（见第三章）可能不会用到强制转换。

## 宽松等价与严格等价

宽松等价是 `==` 操作符，而严格等价是 `===` 操作符。两个操作符都被用于比较两个值的“等价性”，但是“宽松”和“严格”暗示着它们行为之间的一个非常重要的不同，特别是在它们如何决定“等价性”上。

关于这两个操作符的一个非常常见的误解是：“`==` 检查值的等价性，而 `===` 检查值和类型的等价性。”虽然这听起来很好很合理，但是不准确。无数知名的JavaScript书籍和文章都是这么说的，但不幸的是它们都错了。

正确的描述是：“`==` 允许在等价性比较中进行强制转换，而 `===` 不允许强制转换”。

### 等价性的性能

停下来思考一下第一种（不正确的）解释和这第二种（正确的）解释的不同。

在第一种解释中，看起来 `===` 明显的要比 `==` 做更多工作，因为它还必须检查类型。在第二种解释中，`==` 是要做更多工作的，因为它不得不在类型不同时走过强制转换的步骤。

不要像许多人那样落入陷阱中，认为这会与性能有任何关系，虽然在这个问题上 `==` 好像要比 `===` 慢一些。强制转换确实要花费一点点处理时间，但也就是仅仅几微妙（是的，1微妙就是一秒的百万分之一！）。

如果你比较同类型的两个值，`==` 和 `===` 使用的是相同的算法，所以除了在引擎实现上的一些微小的区别，它们做的应当是相同的工作。

如果你比较两个不同类型的值，性能也不是重要因素。你应当问自己的是：当比较这两个值时，我想要进行强制转换吗？

如果你想要进行强制转换，使用 `==` 宽松等价，但如果你不想进行强制转换，就使用 `===` 严格等价。

注意：这里暗示 `==` 和 `===` 都会检查它们的操作数的类型。不同之处在于它们在类型不同时如何反应。

## 抽象等价性

在ES5语言规范的11.9.3部分中，`==`操作符的行为被定义为“抽象等价性比较算法”。那里列出了一个详尽但简单的算法，它明确地指出了类型的每一种可能的组合，与对于每一种组合强制转化应当如何发生（如果有必要的话）。

警告：当（隐含的）强制转换被中伤为太过复杂和缺陷过多而不能成为有用的，好的部分时，遭到谴责的正是这些“抽象等价”规则。一般上，它们被认为对于开发者来说过于复杂和不直观而不能实际学习和应用，而且在JS程序中，和改善代码的可读性比起来，它倾向于导致更多的bug。我相信这是一种有缺陷的预断——读者都是整天都在写（而且读，理解）算法（也就是代码）的能干的开发者。所以，接下来的是用简单的词语来直白地解读“抽象等价性”。但我恳请你也去读一下ES5规范的11.9.3部分。我想你将会对它是多么合理而感到震惊。

基本上，它的第一个条款（11.9.3.1）是在说，如果两个被比较的值是同一类型，它们就像你期望的那样通过等价性简单自然地比较。比如，`42` 只和 `42` 相等，而 `"abc"` 只和 `"abc"` 相等。

在一般期望的结果中，有一些例外需要小心：

- `Nan` 永远不等于它自己（见第二章）
- `+0` 和 `-0` 是相等的（见第二章）

条款11.9.3.1的最后一个规定是关于 `object`（包括 `function` 和 `array`）的 `==` 宽松相等性比较。这样的两个值仅在它们引用完全相同的值时相等。这里没有强制转换发生。

注意：`===` 严格等价比较与11.9.3.1的定义一模一样，包括关于两个 `object` 的值的规定。很少有人知道，在两个 `object` 被比较的情况下，`==` 和 `===` 的行为相同！

11.9.3算法中的剩余部分指出，如果你使用 `==` 宽松等价来比较两个不同类型的值，它们两者或其中之一将需要被隐含地强制转换。由于这个强制转换，两个值最终归于同一类型，可以使用简单的值的等价性来直接比较它们相等与否。

注意：`!=` 宽松不等价操作是如你预料的那样定义的，它差不多就是 `==` 比较操作完整实施，之后对结果取反。这对于 `!==` 严格不等价操作也是一样的。

### 比较：`string` 与 `number`

为了展示 `==` 强制转换，首先让我们建立本章中早先的 `string` 和 `number` 的例子：

```
var a = 42;
var b = "42";

a === b;    // false
a == b;     // true
```

我们所预料的，`a === b` 失败了，因为不允许强制转换，而且值 `42` 和 `"42"` 确实是不同的。

然而，第二个比较 `a == b` 使用了宽松等价，这意味着如果类型偶然不同，这个比较算法将会对两个或其中一个值实施隐含的强制转换。

那么这里发生的究竟是那种强制转换呢？是 `a` 的值变成了一个 `string`，还是 `b` 的值 `"42"` 变成了一个 `number`？

在ES5语言规范中，条款11.9.3.4-5说：

1. 如果`Type(x)`是`Number`而`Type(y)`是`String`，返回比较`x == ToNumber(y)`的结果。
2. 如果`Type(x)`是`String`而`Type(y)`是`Number`，返回比较`ToNumber(x) == y`的结果。

**警告：**语言规范中使用 `Number` 和 `String` 作为类型的正式名称，虽然这本书中偏好使用 `number` 和 `string` 指代基本类型。别让语言规范中首字母大写的 `Number` 与 `Number()` 原生函数把你给搞糊涂了。对于我们的目的来说，类型名称的首字母大写是无关紧要的——它们基本上是同一个意思。

显然，语言规范说为了比较，将值 `"42"` 强制转换为一个 `number`。这个强制转换如何进行已经在前面讲过了，明确地说就是通过 `ToNumber` 抽象操作。在这种情况下十分明显，两个值 `42` 是相等的。

## 比较：任何东西与 `boolean`

当你试着将一个值直接与 `true` 或 `false` 相比较时，你会遇到 `==` 宽松等价的隐含强制转换中最大的一个坑。

考虑如下代码：

```
var a = "42";
var b = true;

a == b;    // false
```

等一下，这里发生了什么！？我们知道 `"42"` 是一个`truthy`值（见本章早先的部分）。那么它和 `true` 怎么不是 `==` 宽松等价的？

其中的原因既简单又刁钻得使人迷惑。它是如此的容易让人误解，许多JS开发者从来不会花费足够多的精力来完全掌握它。

让我们再次引用语言规范，条款11.9.3.6-7

1. 如果`Type(x)`是`Boolean`，返回比较`ToNumber(x) == y`的结果。
2. 如果`Type(y)`是`Boolean`，返回比较`x == ToNumber(y)`的结果。

我们来把它分解。首先：

```
var x = true;
var y = "42";

x == y; // false
```

Type(x) 确实是 Boolean，所以它会实施 ToNumber(x)，将 true 强制转换为 1。现在，1 == "42" 会被求值。这里面的类型依然不同，所以（实质上是递归地）我们再次向早先讲解过的算法求解，它将 "42" 强制转换为 42，而 1 == 42 明显是 false。

反过来，我们任然得到相同的结果：

```
var x = "42";
var y = false;

x == y; // false
```

这次 Type(y) 是 Boolean，所以 ToNumber(y) 给出 0。"42" == 0 递归地变为 42 == 0，这当然是 false。

换句话说，值 "42" 既不 == true 也不 == false。猛地一看，这看起来像句疯话。一个值怎么可能既不是truthy也不是falsy呢？

但这就是问题所在！你在问一个完全错误的问题。但这确实不是你的错，你的大脑在耍你。

"42" 的确是truthy，但是 "42" == true 根本就不是在进行一个boolean测试/强制转换，不管你的大脑怎么说，"42" 没有被强制转换为一个 boolean (true)，而是 true 被强制转换为一个 1，而后 "42" 被强制转换为 42。

不管我们喜不喜欢，ToBoolean 甚至都没参与到这里，所以 "42" 的真假是与 == 操作无关的！

而有关的是要理解 == 比较算法对所有不同类型组合如何动作。当 == 的任意一边是一个 boolean 值时，boolean 总是首先被强制转换为一个 number。

如果这对你来讲很奇怪，那么你不是一个人。我个人建议永远，永远，不要在任何情况下，使用 == true 或 == false。永远。

但时要记住，我在此说的仅与 == 有关。== true 和 == false 不允许强制转换，所以它们没有 ToNumber 强制转换，因而是安全的。

考虑如下代码：

```

var a = "42";

// 不好（会失败的！）：
if (a == true) {
    // ...
}

// 也不该（会失败的！）：
if (a === true) {
    // ...
}

// 足够好（隐含地工作）：
if (a) {
    // ...
}

// 更好（明确地工作）：
if (!!a) {
    // ...
}

// 也很好（明确地工作）：
if (Boolean(a)) {
    // ...
}

```

如果你在你的代码中一直避免使用 `== true` 或 `== false`（也就是与 `boolean` 的宽松等价），你将永远不必担心这种真/假的思维陷阱。

## 比较：`null` 与 `undefined`

另一个隐含强制转换的例子可以在 `null` 和 `undefined` 值之间的 `==` 宽松等价中看到。又再一次引述ES5语言规范，条款11.9.3.2-3：

1. 如果x是`null`而y是`undefined`，返回`true`。
2. 如果x是`undefined`而y是`null`，返回`true`。

当使用 `==` 宽松等价比较 `null` 和 `undefined`，它们是互相等价（也就是互相强制转换）的，而且在整个语言中不会等价于其他值了。

这意味着 `null` 和 `undefined` 对于比较的目的来说，如果你使用 `==` 宽松等价操作符来允许它们互相隐含地强制转换的话，它们可以被认为是不可区分的。

```

var a = null;
var b;

a == b;          // true
a == null;       // true
b == null;       // true

a == false;      // false
b == false;      // false
a == "";         // false
b == "";         // false
a == 0;          // false
b == 0;          // false

```

`null` 和 `undefined` 之间的强制转换是安全且可预见的，而且在这样的检查中没有其他的值会给出测试成立的误报。我推荐使用这种强制转换来允许 `null` 和 `undefined` 是不可区分的，如此将它们作为相同的值对待。

比如：

```

var a = doSomething();

if (a == null) {
    // ...
}

```

`a == null` 检查仅在 `doSomething()` 返回 `null` 或者 `undefined` 时才会通过，而在任何其他值的情况下将会失败，即便是 `0`，`false`，和 `""` 这样的 `falsy` 值。

这个检查的明确形式——不允许任何强制转换——（我认为）没有必要地难看太多了（而且性能可能有点儿不好！）：

```

var a = doSomething();

if (a === undefined || a === null) {
    // ...
}

```

在我看来，`a == null` 的形式是另一个用隐含强制转换增进了代码可读性的例子，而且是以一种可靠安全的方式。

## 比较：`object` 与非 `object`

如果一个 `object / function / array` 被与一个简单基本标量（`string`，`number`，或 `boolean`）进行比较，ES5 语言规范在条款 11.9.3.8-9 中这样说道：

1. 如果 `Type(x)` 是一个 `String` 或者 `Number` 而 `Type(y)` 是一个 `Object`， 返回比较 `x == ToPrimitive(y)` 的结果。
2. 如果 `Type(x)` 是一个 `Object` 而 `Type(y)` 是 `String` 或者 `Number`， 返回比较 `ToPrimitive(x) == y` 的结果。

注意：你可能注意到了，这些条款仅提到了 `String` 和 `Number`，而没有 `Boolean`。这是因为，正如我们早先引述的，条款 11.9.3.6-7 首先将任何出现的 `Boolean` 操作数强制转换为一个 `Number`。

考虑如下代码：

```
var a = 42;
var b = [ 42 ];

a == b;      // true
```

值 `[ 42 ]` 的 `ToPrimitive` 抽象操作（见先前的“抽象值操作”部分）被调用，结果为值 `"42"`。这里它就变为 `42 == "42"`，我们已经讲解过这将变为 `42 == 42`，所以 `a` 和 `b` 被认为是强制转换地等价。

提示：我们在本章早先讨论过的 `ToPrimitive` 抽象操作的所以奇怪之处 (`toString()`, `valueOf()`)，都在这里如你期望的那样适用。如果你有一个复杂的数据结构，而且你想在它上面定义一个 `valueOf()` 方法来为等价比较提供一个简单值的话，这将十分有用。

在第三章中，我们讲解了“拆箱”，就是一个基本类型值的 `object` 包装器（例如 `new String("abc")` 这样的形式）被展开，其底层的基本类型值（`"abc"`）被返回。这种行为与 `==` 算法中的 `ToPrimitive` 强制转换有关：

```
var a = "abc";
var b = Object( a );      // 与 `new String( a )` 相同

a === b;                  // false
a == b;                   // true
```

`a == b` 为 `true` 是因为 `b` 通过 `ToPrimitive` 强制转换为它的底层简单基本标量值 `"abc"`，它与 `a` 中的值是相同的。

然而由于 `==` 算法中的其他覆盖规则，有些值是例外。考虑如下代码：

```

var a = null;
var b = Object( a );      // 与`Object()`相同
a == b;                  // false

var c = undefined;
var d = Object( c );      // 与`Object()`相同
c == d;                  // false

var e = NaN;
var f = Object( e );      // 与`new Number( e )`相同
e == f;                  // false

```

值 `null` 和 `undefined` 不能被装箱——它们没有等价的对象包装器——所以 `Object(null)` 就像 `Object()` 一样，它们都仅仅产生一个普通对象。

`Nan` 可以被封箱到它等价的 `Number` 对象包装器中，当 `=` 导致拆箱时，比较 `Nan == Nan` 会失败，因为 `Nan` 永远不会自己相等（见第二章）。

## 边界情况

现在我们已经彻底检视了 `=` 宽松等价的隐含强制转换是如何工作的（从合理与惊讶两个方式），让我们召唤角落中最差劲儿的，最疯狂的情况，这样我们就能看到我们需要避免什么来防止被强制转换的bug咬到。

首先，让我们检视修改内建的原生`prototype`是如何产生疯狂的结果的：

一个拥有其他值的数字将会.....

```

Number.prototype.valueOf = function() {
    return 3;
};

new Number( 2 ) == 3;      // true

```

警告：`2 == 3` 不会掉到这个陷阱中，这是由于 `2` 和 `3` 都不会调用内建的 `Number.prototype.valueOf()` 方法，因为它们已经是基本 `number` 值，可以直接比较。然而，`new Number(2)` 必须通过 `ToPrimitive` 强制转换，因此调用 `valueOf()`。

邪恶吧？当然。任何人都不应当做这样的事情。你可以这么做，这个事实有时被当成批评强制转换和 `=` 的根据。但这种沮丧是被误导的。JavaScript 不会因为你能做这样的事情而不好，是做这样的事的开发者不好。不要陷入“我的编程语言应当保护我不受我自己伤害”的谬论。

接下来，让我们考虑另一个刁钻的例子，它将前一个例子的邪恶带到另一个水平：

```
if (a == 2 && a == 3) {
    // ...
}
```

你可能认为这是不可能的，因为 `a` 绝不会同时等于 `2` 和 `3`。但是“同时”是不准确的，因为第一个表达式 `a == 2` 严格地发生在 `a == 3` 之前。

那么，要是我们让 `a.valueOf()` 在每次被调用时拥有一种副作用，使它第一次被调用时返回 `2` 而第二次被调用时返回 `3` 呢？很简单：

```
var i = 2;

Number.prototype.valueOf = function() {
    return i++;
};

var a = new Number( 42 );

if (a == 2 && a == 3) {
    console.log( "Yep, this happened." );
}
```

重申一次，这些都是邪恶的技巧。不要这么做。也不要用它们来抱怨强制转换。潜在地滥用一种机制并不是谴责这种机制的充分证据。避开这些疯狂的技巧，并坚持强制转换的合法与合理的用法就好了。

## False-y 比较

关于 `==` 比较中隐含强制转换的最常见的抱怨，来自于 `falsy` 值互相比较时它们如何令人吃惊地动作。

为了展示，让我们看一个关于 `falsy` 值比较的极端例子的列表，来瞧瞧哪一个是合理的，哪一个是麻烦的：

```

"0" == null;           // false
"0" == undefined;    // false
"0" == false;          // true -- 噢！
"0" == NaN;            // false
"0" == 0;              // true
"0" == "";             // false

false == null;          // false
false == undefined;    // false
false == NaN;            // false
false == 0;              // true -- 噢！
false == "";             // true -- 噢！
false == [];             // true -- 噢！
false == {};             // false

"" == null;             // false
"" == undefined;        // false
"" == NaN;               // false
"" == 0;                 // true -- 噢！
"" == [];                // true -- 噢！
"" == {};                // false

0 == null;               // false
0 == undefined;          // false
0 == NaN;                 // false
0 == [];                  // true -- 噢！
0 == {};                  // false

```

在这24个比较的类表中，17个是十分合理和可预见的。比如，我们知道 "" 和 "NaN" 是根本不可能相等的值，并且它们确实不会强制转换以成为宽松等价的，而 "0" 和 0 是合理等价的，而且确实强制转换为宽松等价。

然而，这些比较中的7个被标上了“噢！”。作为误判的成立，它们更像是会将你陷进去的坑。 "" 和 0 绝对是有区别的不同的值，而且你很少会将它们作为等价的，所以它们的互相强制转换是一种麻烦。注意这里没有任何误判的不成立。

## 疯狂的情况

但是我们不必停留于此。我们可以继续寻找更能引起麻烦的强制转换：

```
[] == ![];           // true
```

噢，这看起来像是更高层次的疯狂，对吧！？你的大脑可能会欺骗你说，你在将一个truthy和 falsy值比较，所以结果 true 是令人吃惊的，因为我们知道一个值不可能同时为truthy和 falsy！

但这不是实际发生的事情。让我们把它分解一下。我们了解！一元操作符吧？它明确地使用 `ToBoolean` 规则将操作数强制转换为一个 `boolean`（而且它还会翻转真假性）。所以在 `[] == ![]` 执行之前，它实际上已经被翻译为了 `[] == false`。我们已将在上面的列表中见过了这种形式（`false == []`），所以它的令人吃惊的结果对我们来说并不新鲜。

其它的极端情况呢？

```
2 == [2];          // true
"" == [null];     // true
```

在关于 `ToNumber` 的讨论中我们说过，右手边的 `[2]` 和 `[null]` 值将会通过一个 `ToPrimitive` 强制转换，以使我们可以方便地与左手边的简单基本类型值进行比较。因为 `array` 值的 `valueOf()` 只是返回 `array` 本身，强制转换会退到 `array` 的字符串化上。

对于第一个比较的右手边的值来说，`[2]` 将变为 `"2"`，然后它会 `ToNumber` 强制转换为 `2`。`[null]` 就直接变成 `""`。

那么，`2 == 2` 和 `"" == ""` 是完全可以理解的。

如果你的直觉依然不喜欢这个结果，那么你的沮丧实际上与你可能认为的强制转换无关。这其实是在抱怨 `array` 值在强制转换为 `string` 值时的默认 `ToPrimitive` 行为。很可能，你只是希望 `[2].toString()` 不返回 `"2"`，或者 `[null].toString()` 不返回 `""`。

但是这些 `string` 强制转换到底应该得出什么结果？对于 `[2]` 的 `string` 强制转换，除了 `"2"` 我确实想不出来其他合适的结果，也许是 `"[2]"`——但这可能会在其他的上下文中很奇怪！

你可以正确地制造另一个例子：因为 `String(null)` 变成了 `"null"`，那么 `String([null])` 也应当变成 `"null"`。这是个合理的断言。所以，它才是真正的犯人。

隐含强制转换在这里并不邪恶。即使一个从 `[null]` 到 `string` 结果为 `""` 的明确强制转换也不。真正奇怪的是，`array` 值字符串化为它们内容的等价物是否有道理，和它是如何发生的。所以，应当将你沮丧的原因指向 `String([...])` 的规则，因为这里才是疯狂起源的地方。也许根本就不应该有 `array` 的字符串化强制转换？但这会在语言的其他部分造成许多的缺点。

另一个常被引用的著名的坑是：

```
0 == "\n";           // true
```

正如我们早先讨论的空 `""`，`\n`（或 `" "`，或其他任何空格的组合）是通过 `ToNumber` 强制转换的，而且结果为 `0`。你还希望空格被转换为其他的什么 `number` 值呢？明确的 `Number()` 给出 `0` 会困扰你吗？

空字符串和空格字符串可以转换为的，另一个真正唯一合理的 number 值是 NaN。但这真的会更好吗？" " == NaN 的比较当然会失败，但是不清楚我们是否真的修正了任何底层的问题。

真实世界中的JS程序由于 0 == "\n" 而失败的几率非常之低，而且这样的极端用例很容易避免。

在任何语言中，类型转换总是有极端用例——强制转换也不例外。这里讨论的是特定的一组极端用例的马后炮，但不是针对强制转换整体而言的争论。

底线：你可能遇到的几乎所有普通值间的疯狂强制转换（除了像早先那样有意而为的 valueOf() 或 toString() 黑科技），都能归结为我们在上面指出的7中情况的短列表。

对比这24个疑似强制转换的坑，考虑另一个像这样的列表：

```
42 == "43";           // false
"foo" == 42;          // false
"true" == true;       // false

42 == "42";           // true
"foo" == [ "foo" ];    // true
```

在这些非falsy，非极端的用例中（而且我们简直可以向这个列表中添加无限多个比较），强制转换完全是安全，合理，和可解释的。

## 可行性检查

好的，当我们深入观察隐含的强制转换时，我确实找到了一些疯狂的东西。难怪大多数开发者声称强制转换是邪恶而且应该避开的，对吧？

但是让我们退一步并做一下可行性检查。

通过大量比较，我们得到了一张7个麻烦的，坑人的强制转换的列表，但我们还得到了另一张（至少17个，但实际上有无限多个）完全正常和可以解释的强制转换的列表。

如果你在寻找一本“把孩子和洗澡水一起泼出去”的教科书，这就是了：由于一个仅有7个坑的列表，而抛弃整个强制转换（安全且有效的行为的无限大列表）。

一个更谨慎的反应是问，“我如何使用强制转换的好部分，而避开这几个坏的部分呢？”

然我们再看一次这个坏列表：

```
"0" == false;           // true -- 噢！
false == 0;             // true -- 噢！
false == "";            // true -- 噢！
false == [];            // true -- 噢！
"" == 0;                // true -- 噢！
"" == [];              // true -- 噢！
0 == [];               // true -- 噢！
```

这个列表中7个项目的4个与 `== false` 比较有关，我们早先说过你应当 总是，总是 避免的。

现在这个列表缩小到了3个项目。

```
"" == 0;                // true -- 噢！
"" == [];              // true -- 噢！
0 == [];               // true -- 噢！
```

这些都是你在一般的JavaScript程序中使用的合理的强制转换吗？在什么条件下它们会发生？

我不认为你在程序里有很大的可能要在一个 `boolean` 测试中使用 `== []`，至少在你知道自己在做什么的情况下。你可能会使用 `== ""` 或 `== 0`，比如：

```
function doSomething(a) {
  if (a == "") {
    // ...
  }
}
```

如果你偶然调用了 `doSomething(0)` 或 `doSomething([])`，你就会吓一跳。另一个例子：

```
function doSomething(a, b) {
  if (a == b) {
    // ...
  }
}
```

再一次，如果你调用 `doSomething("", 0)` 或 `doSomething([], "")` 时，它们会失败。

所以，虽然这些强制转换会咬到你的情况可能存在，而且你会小心地处理它们，但是它们可能不会在你的代码库中超级常见。

## 安全地使用隐含强制转换

我能给你的最重要的建议是：检查你的程序，并推理什么样的值会出现在 `==` 比较两边。为了避免这样的比较中的问题，这里有一些可以遵循的启发性规则：

1. 如果比较的任意一边可能出现 `true` 或者 `false` 值，那么就永远，永远不要使用 `==`。
2. 如果比较的任意一边可能出现 `[]`，`""`，或 `0` 这些值，那么认真地考虑不使用 `==`。

在这些场景中，为了避免不希望的强制转换，几乎可以确定使用 `===` 要比使用 `==` 好。遵循这两个简单的规则，可以有效地避免几乎所有可能会伤害你的强制转换的坑。

在这些情况下，使用更加明确/繁冗的方式会减少很多使你头疼的东西。

`==` 与 `===` 的问题其实可以更加恰当地表述为：你是否应当在比较中允许强制转换？

在许多情况下这样的强制转换会很有用，允许你更简练地表述一些比较逻辑（例如，`null` 和 `undefined`）。

对于整体来说，相对有几个 隐含 强制转换会真的很危险的情况。但是在这些地方，为了安全起见，绝对要使用 `===`。

提示：另一个强制转换保证不会咬到你的地方是 `typeof` 操作符。`typeof` 总是将返回给你 7 中字符串之一（见第一章），它们中没有一个是空 `""` 字符串。这样，检查某个值的类型时不会有任何情况与 隐含 强制转换相冲突。`typeof x == "function"` 就像 `typeof x === "function"` 一样 100% 安全可靠。从字面意义上讲，语言规范说这种情况下它们的算法是相同的。所以，不要只是因为你的代码工具告诉你这么做，或者（最差劲儿的）在某本书中有人告诉你 不要考虑它，而盲目地到处使用 `==`。你掌管着你的代码的质量。

隐含 强制转换是邪恶和危险的吗？在几个情况下，是的，但总体说来，不是。

做一个负责任和成熟的开发者。学习如何有效并安全地使用强制转换（明确的和 隐含 的两者）的力量。并教你周围的人也这么做。

这里是由 Alex Dorey (@dorey on GitHub) 制作的一个方便的表格，将各种比较进行了可视化：

Not equal

Loose equality  
Often gives "false" positives like "1" is true; [] is "0"

Strict equality  
Mostly evaluates as one would expect.

出处：<https://github.com/dorey/JavaScript-Equality-Table>

## 抽象关系比较

虽然这部分的隐含强制转换经常不为人所注意，但无论如何考虑比较 `a < b` 时发生了什么是很重要的（和我们如何深入检视 `a == b` 类似）。

在ES5语言规范的11.8.5部分的“抽象关系型比较”算法，实质上把自己分成了两个部分：如果比较涉及两个 `string` 值要做什么（后半部分），和除此之外的其他值要做什么（前半部分）。

注意：这个算法仅仅定义了 `a < b`。所以，`a > b` 作为 `b < a` 处理。

这个算法首先在两个值上调用 `ToPrimitive` 强制转换，如果两个调用的返回值之一不是 `string`，那么就使用 `ToNumber` 操作规则将这两个值强制转换为 `number` 值，并进行数字的比较。

举例来说：

```
var a = [ 42 ];
var b = [ "43" ];

a < b;      // true
b < a;      // false
```

注意：早先讨论的关于 `-0` 和 `Nan` 在 `==` 算法中的类似注意事项也适用于这里。

然而，如果 `<` 比较的两个值都是 `string` 的话，就会在字符串上进行简单的字典顺序（自然的字母顺序）比较：

```
var a = [ "42" ];
var b = [ "043" ];

a < b;      // false
```

`a` 和 `b` 不会被强制转换为 `number`，因为它们会在两个 `array` 的 `ToPrimitive` 强制转换后成为 `string`。所以，`"42"` 将会与 `"043"` 一个字符一个字符地进行比较，从第一个字符开始，分别是 `"4"` 和 `"0"`。因为 `"0"` 在字典顺序上小于 `"4"`，所以这个比较返回 `false`。

完全相同的行为和推理也适用于：

```
var a = [ 4, 2 ];
var b = [ 0, 4, 3 ];

a < b;      // false
```

这里，`a` 变成了 `"4,2"` 而 `b` 变成了 `"0,4,3"`，而字典顺序比较和前一个代码段一模一样。

那么这个怎么样：

```
var a = { b: 42 };
var b = { b: 43 };

a < b;      // ??
```

`a < b` 也是 `false`，因为 `a` 变成了 `[object Object]` 而 `b` 变成了 `[object Object]`，所以明显地 `a` 在字典顺序上不小于 `b`。

但奇怪的是：

```

var a = { b: 42 };
var b = { b: 43 };

a < b;      // false
a == b;     // false
a > b;      // false

a <= b;     // true
a >= b;     // true

```

为什么 `a == b` 不是 `true`？它们是相同的 `string` 值（`"[object Object]"`），所以看起来它们应当相等，对吧？不。回忆一下前面关于 `==` 如何与 `object` 引用进行工作的讨论。

那么为什么 `a <= b` 和 `a >= b` 的结果为 `true`，如果 `a < b` 和 `a == b` 和 `a > b` 都是 `false`？

因为语言规范说，对于 `a <= b`，它实际上首先对 `b < a` 求值，然后反转那个结果。因为 `b < a` 也是 `false`，所以 `a <= b` 的结果为 `true`。

到目前为止你解释 `<=` 在做什么的方式可能是：“小于或等于”。而这可能完全相反，JS更准确地将 `<=` 考虑为“不大于”（`!(a > b)`，JS将它作为 `(!b < a)`）。另外，`a >= b` 被解释为它首先被考虑为 `b <= a`，然后实施相同的推理。

不幸的是，没有像等价那样的“严格的关系型比较”。换句话说，没有办法防止 `a < b` 这样的关系型比较发生 隐含的 强制转换，除非在进行比较之前就明确地确保 `a` 和 `b` 是同种类型。

使用与我们早先 `==` 与 `===` 合理性检查的讨论相同的推理方法。如果强制转换有帮助并且合理安全，比如比较 `42 < "43"`，就使用它。另一方面，如果你需要在关系型比较上获得安全性，那么在使用 `<`（或 `>`）之前，就首先 明确地强制转换 这些值。

```

var a = [ 42 ];
var b = "043";

a < b;                      // false -- 字符串比较！
Number( a ) < Number( b );   // true -- 数字比较！

```

## 复习

在这一章中，我们将注意力转向了JavaScript类型转换如何发生，也叫 强制转换，按性质来说它要么是明确的 要么是 隐含的。

强制转换的名声很坏，但它实际上在许多情况下很有帮助。对于负责任的JS开发者来说，一个重要的任务就是花时间去学习强制转换的里里外外，来决定哪一部分将帮助他们改进代码，哪一部分他们真的应该回避。

明确的 强制转换时这样一种代码，它很明显地有意将一个值从一种类型转换到另一种类型。它的益处是通过减少困惑来增强了代码的可读性和可维护性。

隐含的 强制转换是作为一些其他操作的“隐藏的”副作用而存在的，将要发生的类型转换并不明显。虽然看起来 隐含的 强制转换是 明确的 反面，而且因此是不好的（确实，很多人这么认为！），但是实际上 隐含的 强制转换也是为了增强代码的可读性。

特别是对于 隐含的，强制转换必须被负责地，有意识地使用。懂得为什么你在写你正在写的代码，和它是如何工作的。同时也要努力编写其他人容易学习和理解的代码。

# 你不懂JS：类型与文法

## 第五章：文法

我们想要解决的最后一个主要话题是JavaScript的语法如何工作（也称为它的文法）。你可能认为你懂得如何编写JS，但是语言文法的各个部分中有太多微妙的地方导致了困惑和误解，所以我们想要深入这些部分并搞清楚一些事情。

注意：对于读者们来说，“文法（grammar）”一词不像“语法（syntax）”一词那么为人熟知。在许多意义上，它们是相似的词，描述语言如何工作的规则。它们有一些微妙的不同，但是大部分对于我们在这里的讨论无关紧要。JavaScript的文法是一种结构化的方式，来描述语法（操作符，关键字，等等）如何组合在一起形成结构良好，合法的程序。换句话说，抛开文法来讨论语法将会忽略许多重要的细节。所以我们在本章中注目的内容的最准确的描述是文法，尽管语言中的纯语法才是开发者们直接交互的。

### 语句与表达式

一个很常见的现象是，开发者们假定“语句（statement）”和“表达式（expression）”是大致等价的。但是这里我们需要区分它们俩，因为在我们的JS程序中它们有一些非常重要的区别。

为了描述这种区别，让我们借用一下你可能更熟悉的术语：英语。

一个“句子（sentence）”是一个表达想法的词汇的完整构造。它由一个或多个“短语（phrase）”组成，它们每一个都可以用标点符号或连词（“和”，“或”等等）连接。一个短语本身可以由更小的短语组成。一些短语是不完整的，而且本身没有太多含义，而另一些短语可以自成一句。这些规则总体地称为英语的文法。

JavaScript文法也类似。语句就是句子，表达式就是短语，而操作符就是连词/标点。

JS中的每一个表达式都可以被求值而成为一个单独的，具体的结果值。举例来说：

```
var a = 3 * 6;
var b = a;
b;
```

在这个代码段中，`3 * 6` 是一个表达式（求值得值 `18`）。而第二行的 `a` 也是一个表达式，第三行的 `b` 也一样。对表达式 `a` 和 `b` 求值都会得到在那一时刻存储在这些变量中的值，也就偶然是 `18`。

另外，这三行的每一行都是一个包含表达式的语句。`var a = 3 * 6` 和 `var b = a` 称为“声明语句（declaration statements）”因为它们每一个都声明了一个变量（并选择性地给它赋值）。赋值 `a = 3 * 6` 和 `b = a`（除去 `var`）被称为赋值表达式（assignment expressions）。

第三行仅仅含有一个表达式 `b`，但是它本身也是一个语句（虽然不是非常有趣的一个！）。这一般称为一个“表达式语句（expression statement）”。

## 语句完成值

一个鲜为人知的事实是，所有语句都有完成值（即使这个值只是 `undefined`）。

你要如何做才能看到一个语句的完成值呢？

最明显的答案是把语句敲进你的浏览器开发者控制台，因为当你运行它时，默认地控制台会报告最近一次执行的语句的完成值。

让我们考虑一下 `var b = a`。这个语句的完成值是什么？

`b = a` 赋值表达式给出的结果是被赋予的值（上面的 18），但是 `var` 语句本身给出的结果是 `undefined`。为什么？因为在语言规范中 `var` 语句就是这么定义的。如果你在你的控制台中敲入 `var a = 42`，你会看到 `undefined` 被报告而不是 `42`。

注意：技术上讲，事情要比这复杂一些。在ES5语言规范，12.2部分的“变量语句”中，`VariableDeclaration` 算法实际上返回了一个值（一个包含被声明变量的名称的 `string` —— 诡异吧！？），但是这个值基本上被 `VariableStatement` 算法吞掉了（除了在 `for..in` 循环中使用），而这强制产生一个空的（也就是 `undefined`）完成值。

事实上，如果你曾在你的控制台上（或者一个JavaScript环境的REPL——`read/evaluate/print/loop`工具）做过很多的代码实验的话，你可能看到过许多不同的语句都报告 `undefined`，而且你也许从来没理解它是什么和为什么。简单地说，控制台仅仅报告语句的完成值。

但是控制台打印出的完成值并不是我们可以在程序中使用的东西。那么我们该如何捕获完成值呢？

这是个更加复杂的任务。在我们解释如何之前，让我们先探索一下为什么你想这样做。

我们需要考虑其他类型的语句的完成值。例如，任何普通的 `{ .. }` 块儿都有一个完成值，即它所包含的最后一个语句/表达式的完成值。

考虑如下代码：

```
var b;

if (true) {
    b = 4 + 38;
}
```

如果你将这段代码敲入你的控制台/REPL，你可能会看到它报告 42，因为 42 是 if 块儿的完成值，它取自 if 的最后一个复制表达式语句 `b = 4 + 38`。

换句话说，一个块儿的完成值就像隐含地返回块儿中最后一个语句的值。

注意：这在概念上与CoffeeScript这样的语言很类似，它们隐含地从 `function` 中 `return` 值，这些值与函数中最后一个语句的值是相同的。

但这里有一个明显的问题。这样的代码是不工作的：

```
var a, b;

a = if (true) {
    b = 4 + 38;
};
```

我们不能以任何简单的语法/文法来捕获一个语句的完成值并将它赋值给另一个变量（至少是还不能！）。

那么，我们能做什么？

警告：仅用于演示的目的——不要实际地在你的真实代码中做如下内容！

我们可以使用臭名昭著的 `eval(...)`（有时读成“evil”）函数来捕获这个完成值。

```
var a, b;

a = eval( "if (true) { b = 4 + 38; }" );
a;      // 42
```

啊呀呀。这太难看了。但是这好用！而且它展示了语句的完成值是一个真实的东西，不仅仅仅是在控制台中，还可以在我们的程序中被捕获。

有一个称为“do表达式”的ES7提案。这是它可能工作的方式：

```

var a, b;

a = do {
    if (true) {
        b = 4 + 38;
    }
};

a; // 42

```

`do { ... }` 表达式执行一个块儿（其中有一个或多个语句），这个块儿中的最后一个语句的完成值将成为 `do` 表达式的完成值，它可以像展示的那样被赋值给 `a`。

这里的大意是能够将语句作为表达式对待——他们可以出现在其他语句内部——而不必将它们包装在一个内联的函数表达式中，并实施一个明确的 `return ...`。

到目前为止，语句的完成值不过是一些琐碎的事情。不顾随着JS的进化它们的重要性可能会进一步提高，而且很有希望的是 `do { ... }` 表达式将会降低使用 `eval(..)` 这样的东西的冲动。

警告：重复我刚才的训诫：避开 `eval(..)`。真的。更多解释参见本系列的作用域与闭包一书。

## 表达式副作用

大多数表达式没有副作用。例如：

```

var a = 2;
var b = a + 3;

```

表达式 `a + 3` 本身并没有副作用，例如改变 `a`。它有一个结果，就是 `5`，而且这个结果在语句 `b = a + 3` 中被赋值给 `b`。

一个最常见的（可能）带有副作用的表达式的例子是函数调用表达式：

```

function foo() {
    a = a + 1;
}

var a = 1;
foo(); // 结果：`undefined`，副作用：改变 `a`

```

还有其他的副作用表达式。例如：

```
var a = 42;
var b = a++;
```

表达式 `a++` 有两个分离的行为。首先，它返回 `a` 的当前值，也就是 `42`（然后它被赋值给 `b`）。但接下来，它改变 `a` 本身的值，将它增加1。

```
var a = 42;
var b = a++;

a;      // 43
b;      // 42
```

许多开发者错误的认为 `b` 和 `a` 一样拥有值 `43`。这种困惑源自没有完全考虑 `++` 操作符的副作用在什么时候发生。

`++` 递增操作符和 `--` 递减操作符都是一元操作符（见第四章），它们既可以用于后缀（“后面”）位置也可用于前缀（“前面”）位置。

```
var a = 42;

a++;    // 42
a;      // 43

++a;    // 44
a;      // 44
```

当 `++` 像 `++a` 这样用于前缀位置时，它的副作用（递增 `a`）发生在值从表达式中返回之前，而不是 `a++` 那样发生在之后。

注意：你认为 `++a++` 是一个合法的语法吗？如果你试一下，你将会得到一个 `ReferenceError` 错误，但为什么？因为有副作用的操作符要求一个变量引用来作为它们副作用的目标。对于 `++a++` 来说，`a++` 这部分会首先被求值（因为操作符优先级——参见下面的讨论），它会给出 `a` 在递增之前的值。但然后它试着对 `++42` 求值，这将（如果你试一下）会给出相同的 `ReferenceError` 错误，因为 `++` 不能直接在 `42` 这样的值上施加副作用。

有时它会被错误地认为，你可以通过将 `a++` 包进一个 `()` 中来封装它的后副作用，比如：

```
var a = 42;
var b = (a++);

a;      // 43
b;      // 42
```

不幸的是，`( )` 本身不会像我们希望的那样，定义一个新的被包装的表达式，而它会在 `a++` 表达式的后副作用之后求值。事实上，就算它能，`a++` 也会首先返回 `42`，而且除非你有另一个表达式在 `++` 的副作用之后对 `a` 再次求值，你也不会从这个表达式中得到 `43`，于是 `b` 不会被赋值为 `43`。

虽然，有另一种选择：`,` 语句序列逗号操作符。这个操作符允许你将多个独立的表达式语句连成一个单独的语句：

```
var a = 42, b;
b = ( a++, a );

a;      // 43
b;      // 43
```

注意：`a++, a` 周围的 `( ... )` 是必需的。其原因的操作符优先级，我们将在本章后面讨论。

表达式 `a++, a` 意味着第二个 `a` 语句表达式会在第一个 `a++` 语句表达式的后副作用之后进行求值，这表明它为 `b` 的赋值返回 `43`。

另一个副作用操作符的例子是 `delete`。正如我们在第二章中展示的，`delete` 用于从一个 `object` 或一个 `array` 值槽中移除一个属性。但它经常作为一个独立语句被调用：

```
var obj = {
  a: 42
};

obj.a;          // 42
delete obj.a;  // true
obj.a;          // undefined
```

如果被请求的操作是合法/可允许的，`delete` 操作符的结果值为 `true`，否则结果为 `false`。但是这个操作符的副作用是它移除了属性（或数组值槽）。

注意：我们说合法/可允许是什么意思？不存在的属性，或存在且可配置的属性（见本系列 `this` 与对象原型的第三章）将会从 `delete` 操作符中返回 `true`。否则，其结果将是 `false` 或者一个错误。

副作用操作符的最后一个例子，可能既是明显的也是不明显的，是 `=` 赋值操作符。

考虑如下代码：

```
var a;

a = 42;        // 42
a;            // 42
```

对于这个表达式来说，`a = 42` 中的 `=` 看起来似乎不是一个副作用操作符。但如果我们检视语句 `a = 42` 的结果值，会发现它就是刚刚被赋予的值（`42`），所以向 `a` 赋予的相同的值实质上是一种副作用。

提示：相同的原因也适用于 `+=`，`-=` 这样的复合赋值操作符的副作用。例如，`a = b += 2` 被处理为首先进行 `b += 2`（也就是 `b = b + 2`），然后这个赋值的结果被赋予 `a`。

这种赋值表达式（语句）得出被赋予的值的行为，主要在链式赋值上十分有用，就像这样：

```
var a, b, c;

a = b = c = 42;
```

这里，`c = 42` 被求值得出 `42`（带有将 `42` 赋值给 `c` 的副作用），然后 `b = 42` 被求值得出 `42`（带有将 `42` 赋值给 `b` 的副作用），而最后 `a = 42` 被求值（带有将 `42` 赋值给 `a` 的副作用）。

警告：一个开发者们常犯的错误是将链式赋值写成 `var a = b = 42` 这样。虽然这看起来是相同的东西，但它不是。如果这个语句发生在没有另外分离的 `var b`（在作用域的某处）来正式声明它的情况下，那么 `var a = b = 42` 将不会直接声明 `b`。根据 `strict` 模式的状态，它要么抛出一个错误，要么无意中创建一个全局变量（参见本系列的作用域与闭包）。

另一个要考虑的场景是：

```
function vowels(str) {
  var matches;

  if (str) {
    // 找出所有的元音字母
    matches = str.match( /[aeiou]/g );

    if (matches) {
      return matches;
    }
  }
}

vowels( "Hello World" ); // ["e", "o", "o"]
```

这可以工作，而且许多开发者喜欢这么做。但是使用一个我们可以利用赋值副作用的惯用法，可以通过将两个 `if` 语句组合为一个来进行简化：

```

function vowels(str) {
  var matches;

  // 找出所有的元音字母
  if (str && (matches = str.match( /[aeiou]/g ))) {
    return matches;
  }
}

vowels( "Hello World" ); // ["e", "o", "o"]

```

注意：`matches = str.match..` 周围的`( ... )`是必需的。其原因是操作符优先级，我们将在本章稍后的“操作符优先级”一节中讨论。

我偏好这种短一些的风格，因为我认为它明白地表示了两个条件其实是有关联的，而非分离的。但是与大多数JS中的风格选择一样，哪一种更好纯粹是个人意见。

## 上下文规则

在JavaScript文法规则中有好几个地方，同样的语法根据它们被使用的地方/方式不同意味着不同的东西。这样的东西可能，孤立的看，导致相当多的困惑。

我们不会在这里详尽地罗列所有这些情况，而只是指出常见的几个。

### { .. } 大括号

在你的代码中一对`{ .. }`大括号将主要出现在两种地方（随着JS的进化会有更多！）。让我们来看看它们每一种。

#### 对象字面量

首先，作为一个`object`字面量：

```

// 假定有一个函数`bar()`的定义

var a = {
  foo: bar()
};

```

我们怎么知道这是一个`object`字面量？因为`{ .. }`是一个被赋予给`a`的值。

注意：`a`这个引用被称为一个“l-值”（也称为左手边的值）因为它是赋值的目标。`{ .. }`是一个“r-值”（也称为右手边的值）因为它仅被作为一个值使用（在这里作为赋值的源）。

#### 标签

如果我们移除上面代码的 `var a =` 部分会发生什么？

```
// 假定有一个函数`bar()`的定义

{
  foo: bar()
}
```

许多开发者臆测 `{ ... }` 只是一个独立的没有被赋值给任何地方的 `object` 字面量。但事实上完全不同。

这里，`{ ... }` 只是一个普通的代码块儿。在JavaScript中拥有一个这样的独立 `{ ... }` 块儿并不是一个很惯用的形式（在其他语言中要常见得多！），但它是完美合法的JS文法。当与 `let` 块儿作用域声明组合使用时非常有用（见本系列的作用域与闭包）。

这里的 `{ ... }` 代码块儿在功能上差不多与附着在一些语句后面的代码块儿是相同的，比如 `for` / `while` 循环，`if` 条件，等等。

但如果它是一个一般代码块儿，那么那个看起来异乎寻常的 `foo: bar()` 语法是什么？它怎么会是合法的呢？

这是因为一个鲜为人知的（而且，坦白地说，不鼓励使用的）称为“打标签的语句”的JavaScript特性。`foo` 是语句 `bar()`（这个语句省略了末尾的`;`——见本章稍后的“自动分号”）的标签。但一个打了标签的语句有何意义？

如果JavaScript有一个 `goto` 语句，那么在理论上你就可以说 `goto foo` 并使程序的执行跳转到代码中的那个位置。`goto` 通常被认为是一种糟糕的编码惯用形式，因为它们使代码更难于理解（也称为“面条代码”），所以JavaScript没有一般的 `goto` 语句是一件非常好的事情。

然而，JS的确支持一种有限的，特殊形式的 `goto :标签` 跳转。`continue` 和 `break` 语句都可以选择性地接受一个指定的标签，在这种情况下程序流会有些像 `goto` 一样“跳转”。考虑一下代码：

```
// 用`foo`标记的循环
foo: for (var i=0; i<4; i++) {
    for (var j=0; j<4; j++) {
        // 每当循环相遇，就继续外层循环
        if (j == i) {
            // 跳到被`foo`标记的循环的下一次迭代
            continue foo;
        }

        // 跳过奇数的乘积
        if ((j * i) % 2 == 1) {
            // 内层循环的普通（没有被标记的）`continue`
            continue;
        }

        console.log( i, j );
    }
}

// 1 0
// 2 0
// 2 1
// 3 0
// 3 2
```

注意：`continue foo` 不意味着“走到标记为‘foo’的位置并继续”，而是，“继续标记为‘foo’的循环，并进行下一次迭代”。所以，它不是一个真正的随意的 `goto`。

如你所见，我们跳过了乘积为奇数的 `3 1` 迭代，而且被打了标签的循环跳转还跳过了 `1 1` 和 `2 2` 的迭代。

也许标签跳转的一个稍稍更有用的形式是，使用 `break __` 从一个内部循环里面跳出外部循环。没有带标签的 `break`，同样的逻辑有时写起来非常尴尬：

```
// 用`foo`标记的循环
foo: for (var i=0; i<4; i++) {
  for (var j=0; j<4; j++) {
    if ((i * j) >= 3) {
      console.log( "stopping!", i, j );
      // 跳出被`foo`标记的循环
      break foo;
    }

    console.log( i, j );
  }
}

// 0 0
// 0 1
// 0 2
// 0 3
// 1 0
// 1 1
// 1 2
// stopping! 1 3
```

注意：`break foo` 不意味着“走到‘foo’标记的位置并继续”，而是，“跳出标记为‘foo’的循环/代码块儿，并继续它后面的部分”。不是一个传统意义上的 `goto`，对吧？

对于上面的问题，使用不带标签的 `break` 将可能会牵连一个或多个函数，共享作用域中变量的访问，等等。它很可能要比带标签的 `break` 更令人糊涂，所以在这里使用带标签的 `break` 也许是更好的选择。

一个标签也可以用于一个非循环的块儿，但只有 `break` 可以引用这样的非循环标签。你可以使用带标签的 `break __` 跳出任何被标记的块儿，但你不能 `continue __` 一个非循环标签，也不能用一个不带标签的 `break` 跳出一个块儿。

```
function foo() {
  // 用`bar`标记的块儿
  bar: {
    console.log( "Hello" );
    break bar;
    console.log( "never runs" );
  }
  console.log( "World" );
}

foo();
// Hello
// World
```

带标签的循环/块儿极不常见，而且经常使人皱眉头。最好尽可能地避开它们；比如使用函数调用取代循环跳转。但是也许在一些有限的情况下它们会有用。如果你打算使用标签跳转，那么就确保使用大量注释在文档中记下你在做什么！

一个很常见的想法是，JSON是一个JS的恰当子集，所以一个JSON字符串（比如 `{"a":42}` —— 注意属性名周围的引号是JSON必需的！）被认为是一个合法的JavaScript程序。不是这样的！如果你试着把 `{"a":42}` 敲进你的JS控制台，你会得到一个错误。

这是因为语句标签周围不能有引号，所以 `"a"` 不是一个合法的标签，因此：不能出现在它后面。

所以，JSON确实是JS语法的子集，但是JSON本身不是合法的JS文法。

按照这个路线产生的一个极其常见的误解是，如果你将一个JS文件加载进一个 `<script src=..>` 标签，而它里面仅含有JSON内容的话（就像从API调用中得到那样），这些数据将作为合法的JavaScript被读取，但只是不能从程序中访问。JSON-P（将JSON数据包进一个函数调用的做法，比如 `foo({"a":42})`）经常被说成是解决了这种不可访问性，通过向你程序中的一个函数发送这些值。

不是这样的！实际上完全合法的JSON值 `{"a":42}` 本身将会抛出一个JS错误，因为它被翻译为一个带有非法标签的语句块儿。但是 `foo({"a":42})` 是一个合法的JS，因为在它里面，`{"a":42}` 是一个被传入 `foo(..)` 的 `object` 字面量值。所以，更合适的说法是，**JSON-P使JSON成为合法的JS文法！**

块儿

另一个常为人所诟病的JS坑（与强制转换有关——见第四章）是：

```
[] + {}; // "[object Object]"
{} + []; // 0
```

这看起来暗示着 `+` 操作符会根据第一个操作数是 `[]` 还是 `{}` 而给出不同的结果。但实际上这与它一点儿关系都没有！

在第一行中，`{}` 出现在 `+` 操作符的表达式中，因此被翻译为一个实际的值（一个空 `object`）。第四章解释过，`[]` 被强制转换为 `""` 因此 `{}` 也会被强制转换为一个 `string`：`"[object Object]"`。

但在第二行中，`{}` 被翻译为一个独立的 `{}` 空代码块儿（它什么也不做）。块儿不需要分号来终结它们，所以这里缺少分号不是一个问题。最终，`+ []` 是一个将 `[]` 明确强制转换为 `number` 的表达式，而它的值是 `0`。

对象解构

从ES6开始，你将看到`{ .. }`出现的另一个地方是“解构赋值”（更多信息参见本系列的 *ES6与未来*），确切地说是`object`解构。考虑下面的代码：

```
function getData() {
  // ...
  return {
    a: 42,
    b: "foo"
  };
}

var { a, b } = getData();

console.log( a, b ); // 42 "foo"
```

正如你可能看出来的，`var { a, b } = ..` 是ES6解构赋值的一种形式，它大体等价于：

```
var res = getData();
var a = res.a;
var b = res.b;
```

注意：`{ a, b }` 实际上是`{ a: a, b: b }` 的ES6解构缩写，两者都能工作，但是人们期望短一些的`{ a, b }` 能成为首选的形式。

使用一个`{ .. }` 进行对象解构也可用于被命名的函数参数，这时它是同种类的隐含对象属性赋值的语法糖：

```
function foo({ a, b, c }) {
  // 不再需要：
  // var a = obj.a, b = obj.b, c = obj.c
  console.log( a, b, c );
}

foo( {
  c: [1, 2, 3],
  a: 42,
  b: "foo"
} ); // 42 "foo" [1, 2, 3]
```

所以，我们使用`{ .. }` 的上下文环境整体上决定了它们的含义，这展示了语法和文法之间的区别。理解这些微妙之处以回避JS引擎进行意外的翻译是很重要的。

## else if 和可选块儿

一个常见的误解是JavaScript拥有一个`else if` 子句，因为你可以这么做：

```
if (a) {
    // ...
}
else if (b) {
    // ...
}
else {
    // ...
}
```

但是这里有一个JS文法隐藏的性质：它没有 `else if`。但是如果附着在 `if` 和 `else` 语句后面的代码块儿仅包含一个语句时，`if` 和 `else` 语句允许省略这些代码块儿周围的 `{ }`。毫无疑问，你以前已经见过这种现象很多次了：

```
if (a) doSomething( a );
```

许多JS编码风格指引坚持认为，你应当总是在一个单独的语句块儿周围使用 `{ }`，就像：

```
if (a) { doSomething( a ); }
```

然而，完全相同的文法规则也适用于 `else` 子句，所以你经常编写的 `else if` 形式实际上被解析为：

```
if (a) {
    // ...
}
else {
    if (b) {
        // ...
    }
    else {
        // ...
    }
}
```

`if (b) { .. } else { .. }` 是一个紧随着 `else` 的单独的语句，所以你在它周围放不放一个 `{ }` 都可以。换句话说，当你使用 `else if` 的时候，从技术上讲你就打破了那个常见的编码风格指导的规则，而且只是用一个单独的 `if` 语句定义了你的 `else`。

当然，`else if` 惯用法极其常见，而且减少了一级缩进，所以它很吸引人。无论你用哪种方式，就在你自己的编码风格指导/规则中明确地指出它，并且不要臆测 `else if` 是直接的文法规则。

## 操作符优先级

就像我们在第四章中讲解的，JavaScript版本的 `&&` 和 `||` 很有趣，因为它们选择并返回它们的操作数之一，而不是仅仅得出 `true` 或 `false` 的结果。如果只有两个操作数和一个操作符，这很容易推理。

```
var a = 42;
var b = "foo";

a && b;    // "foo"
a || b;    // 42
```

但是如果牵扯到两个操作符，和三个操作数呢？

```
var a = 42;
var b = "foo";
var c = [1,2,3];

a && b || c; // ???
a || b && c; // ???
```

要明白这些表达式产生什么结果，我们就需要理解当在一个表达式中有多于一个操作符时，什么样的规则统治着操作符被处理的方式。

这些规则称为“操作符优先级”。

我打赌大多数读者都觉得自己已经很好地理解了操作符优先级。但是和我们在本系列丛书中讲解的其他一切东西一样，我们将拨弄这种理解来看看它到底有多扎实，并希望能在这个过程中学到一些新东西。

回想上面的例子：

```
var a = 42, b;
b = ( a++, a );

a;      // 43
b;      // 43
```

要是我们移除了 `( )` 会怎样？

```
var a = 42, b;
b = a++, a;

a;      // 43
b;      // 42
```

等一下！为什么这改变了赋给 `b` 的值？

因为 `,` 操作符要比 `=` 操作符的优先级低。所以，`b = a++, a` 被翻译为 `(b = a++), a`。因为（如我们前面讲解的）`a++` 拥有后副作用，赋值给 `b` 的值就是在 `++` 改变 `a` 之前的值 42。

这只是为了理解操作符优先级所需的一个简单事实。如果你将要把 `,` 作为一个语句序列操作符使用，那么知道它实际上拥有最低的优先级是很重要的。任何其他的操作符都将要比 `,` 结合得更紧密。

现在，回想上面的这个例子：

```
if (str && (matches = str.match( /[aeiou]/g ))) {
    // ...
}
```

我们说过赋值语句周围的 `( )` 是必须的，但为什么？因为 `&&` 拥有的优先级比 `=` 更高，所以如果没有 `( )` 来强制结合，这个表达式将被作为 `(str && matches) = str.match..` 对待。但是这将是个错误，因为 `(str && matches)` 的结果将不是一个变量（在这里是 `undefined`），而是一个值，因此它不能成为 `=` 赋值的左边！

好了，那么你可能认为你已经搞定操作符优先级了。

让我们移动到更复杂的例子（在本章下面几节中我们将一直使用这个例子），来真正测试一下你的理解：

```
var a = 42;
var b = "foo";
var c = false;

var d = a && b || c ? c || b ? a : c && b : a;

d;           // ??
```

好的，邪恶，我承认。没有人会写这样的表达式串，对吧？也许不会，但是我们将使用它来检视将多个操作符链接在一起时的各种问题，而链接多个操作符是一个非常常见的任务。

上面的结果是 42。但是这根本没意思，除非我们自己能搞清楚这个答案，而不是将它插进 JS 程序来让 JavaScript 搞定它。

让我们深入挖掘一下。

第一个问题——你可能还从来没问过——是，第一个部分 `( a && b || c )` 是像 `(a && b) || c` 那样动作，还是像 `a && (b || c)` 那样动作？你能确定吗？你能说服你自己它们实际上是不同的吗？

```
(false && true) || true;      // true
false && (true || true);    // false
```

那么，这就是它们不同的证据。但是 `false && true || true` 到底是如何动作的？答案是：

```
false && true || true;      // true
(false && true) || true;    // true
```

那么我们有了答案。`&&` 操作符首先被求值，而 `||` 操作符第二被求值。

但这不是因为从左到右的处理顺序吗？让我们把操作符的顺序倒过来：

```
true || false && false;      // true
(true || false) && false;    // false -- 不
true || (false && false);    // true -- 这才是胜利者!
```

现在我们证明了 `&&` 首先被求值，然后才是 `||`，而且在这个例子中的顺序实际上是与一般希望的从左到右的顺序相反的。

那么什么导致了这种行为？操作符优先级。

每种语言都定义了自己的操作符优先级列表。虽然令人焦虑，但是JS开发者读过JS的列表却不太常见。

如果你熟知它，上面的例子一点儿都不会绊到你，因为你已经知道了 `&&` 要比 `||` 优先级高。但是我打赌有相当一部分读者不得不将它考虑一会。

注意：不幸的是，JS语言规范没有将它的操作符优先级罗列在一个方便，单独的位置。你不得不通读并理解所有的文法规则。所以我们将试着以一种更方便的格式排列出更常见和更有用的部分。要得到完整的操作符优先级列表，参见MDN网站的“操作符优先级”(\*

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator\\_Precedence](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence))。

## 短接

在第四章中，我们在一个边注中提到了操作符 `&&` 和 `||` 的“短接”性质。让我们更详细地重温它们。

对于 `&&` 和 `||` 两个操作符来说，如果左手边的操作数足够确定操作的结果，那么右手边的操作数将不会被求值。故而，有了“短接”（如果可能，它就会取捷径退出）这个名字。

例如，说 `a && b`，如果 `a` 是 `falsy` `b` 就不会被求值，因为 `&&` 操作数的结果已经确定了，所以再去麻烦地检查 `b` 是没有意义的。同样的，说 `a || b`，如果 `a` 是 `truthy`，那么操作的结果就已经确定了，所以没有理由再去检查 `b`。

这种短接非常有帮助，而且经常被使用：

```
function doSomething(opts) {
  if (opts && opts.cool) {
    // ...
  }
}
```

`opts && opts.cool` 测试的 `opts` 部分就像某种保护，因为如果 `opts` 没有被赋值（或不是一个 `object`），那么表达式 `opts.cool` 就将抛出一个错误。`opts` 测试失败加上短接意味着 `opts.cool` 根本不会被求值，因此没有错误！

相似地，你可以用 `||` 短接：

```
function doSomething(opts) {
  if (opts.cache || primeCache()) {
    // ...
  }
}
```

这里，我们首先检查 `opts.cache`，如果它存在，我们就不会调用 `primeCache()` 函数，如此避免了潜在的不必要的工作。

## 更紧密的绑定

让我们把注意力转回前面全是链接的操作符的复杂语句的例子，特别是`? :`三元操作符的部分。`? :`操作对的优先级与 `&&` 和 `||` 操作符比起来是高还是低？

```
a && b || c ? c || b ? a : c && b : a
```

它是更像这样：

```
a && b || (c ? c || (b ? a : c) && b : a)
```

还是这样？

```
(a && b || c) ? (c || b) ? a : (c && b) : a
```

答案是第二个。但为什么？

因为 `&&` 优先级比 `||` 高，而 `||` 优先级比 `? :` 高。

所以，表达式 `(a && b || c)` 在 `? :` 参与之前被首先求值。另一种常见的解释方式是，`&&` 和 `||` 要比 `? :` “结合的更紧密”。如果倒过来成立的话，那么 `c ? c..` 将结合的更紧密，那么它就会如 `a && b || (c ? c..)` 那样动作（就像第一种选择）。

## 结合性

所以，`&&` 和 `||` 操作符首先结合，然后是 `? :` 操作符。但是多个同等优先级的操作符呢？它们总是从左到右或是从右到左地处理吗？

一般来说，操作符不是左结合的就是右结合的，这要看分组是从左边发生还是从右边发生。

至关重要的是，结合性与从左到右或从右到左的处理不是同一个东西。

但为什么处理是从左到右或从右到左那么重要？因为表达式可以有副作用，例如函数调用：

```
var a = foo() && bar();
```

这里，`foo()` 首先被求值，然后根据表达式 `foo()` 的结果，`bar()` 可能会求值。如果 `bar()` 在 `foo()` 之前被调用绝对会得出不同的程序行为。

但是这个行为就是从左到右的处理（JavaScript中的默认行为！）——它与 `&&` 的结合性无关。在这个例子中，因为这里只有一个 `&&` 因此没有相关的分组，所以根本谈不上结合性。

但是像 `a && b && c` 这样的表达式，分组将会隐含地发生，意味着不是 `a && b` 就是 `b && c` 会先被求值。

技术上讲，`a && b && c` 将会作为 `(a && b) && c` 处理，因为 `&&` 是左结合的（顺带一提，`||` 也是）。然而，右结合的 `a && (b && c)` 也表现出相同的行为。对于相同的值，相同的表达式是按照相同的顺序求值的。

注意：如果假设 `&&` 是右结合的，它就会与你手动使用 `( )` 建立 `a && (b && c)` 这样的分组的处理方式一样。但是这仍然不意味着 `c` 将会在 `b` 之前被处理。右结合性的意思不是从右到左求值，它的意思是右结合的。不管哪种方式，无论分组/结合性怎样，严格的求值顺序将是 `a`，然后 `b`，然后 `c`（也就是从左到右）。

因此，除了使我们对它们定义的讨论更准确以外，`&&` 和 `||` 是左结合这件事没有那么重要。

但事情不总是这样。一些操作符根据左结合性与右结合性将会做出不同的行为。

考虑 `? :`（“三元”或“条件”）操作符：

```
a ? b : c ? d : e;
```

`? :` 是右结合的，那么哪种分组表现了它将被处理的方式？

- `a ? b : (c ? d : e)`
- `(a ? b : c) ? d : e`

答案是 `a ? b : (c ? d : e)`。不像上面的 `&&` 和 `||`，在这里右结合性很重要，因为对于一些（不是全部！）值的组合来说 `(a ? b : c) ? d : e` 的行为将会不同。

一个这样的例子是：

```
true ? false : true ? true : true;           // false
true ? false : (true ? true : true);         // false
(true ? false : true) ? true : true;          // true
```

在其他的值的组合中潜伏着更加微妙的不同，即便他们的最终结果是相同的。考虑：

```
true ? false : true ? true : false;          // false
true ? false : (true ? true : false);         // false
(true ? false : true) ? true : false;          // false
```

在这个场景中，相同的最终结果暗示着分组是没有实际意义的。然而：

```
var a = true, b = false, c = true, d = true, e = false;

a ? b : (c ? d : e); // false, 仅仅对 `a` 和 `b` 求值
(a ? b : c) ? d : e; // false, 对 `a`、`b` 和 `e` 求值
```

这样，我们就清楚地证明了 `? :` 是右结合的，而且在这个操作符与它自己链接的方式上，右结合性是发挥影响的。

另一个右结合（分组）的例子是 `=` 操作符。回想本章早先的链式赋值的例子：

```
var a, b, c;

a = b = c = 42;
```

我们早先断言过，`a = b = c = 42` 的处理方式是，首先对 `c = 42` 赋值求值，然后是 `b = ...`，最后是 `a = ...`。为什么？因为右结合性，它实际上这样看待这个语句：`a = (b = (c = 42))`。

记得本章前面，我们的复杂赋值表达式的实例吗？

```

var a = 42;
var b = "foo";
var c = false;

var d = a && b || c ? c || b ? a : c && b : a;

d;           // 42

```

随着我们使用优先级和结合性的知识把自己武装起来，我们应当可以像这样把这段代码分解为它的分组行为：

```
((a && b) || c) ? ((c || b) ? a : (c && b)) : a
```

或者，如果这样容易理解的话，可以用缩进表达：

```

(
  (a && b)
    ||
  c
)
?
(
  (c || b)
    ?
  a
    :
  (c && b)
)
:
a

```

让我们解析它：

1. `(a && b)` 是 `"foo"` .
2. `"foo" || c` 是 `"foo"` .
3. 对于第一个 `?` 测试，`"foo"` 是 `truthy`。
4. `(c || b)` 是 `"foo"` .
5. 对于第二个 `?` 测试，`"foo"` 是 `truthy`。
6. `a` 是 `42` .

就是这样，我们搞定了！答案是 `42`，正如我们早先看到的。其实它没那么难，不是吗？

## 消除歧义

现在你应该对操作符优先级（和结合性）有了更好的把握，并对理解多个链接的操作符如何动作感到更适应了。

但还存在一个重要的问题：我们应当一直编写完美地依赖于操作符优先级/结合性的代码吗？我们应该仅在有必要强制一种不同的处理顺序时使用（）手动分组吗？

或者，另一方面，我们应当这样认识吗：虽然这样的规则实际上是可以学懂的，但是太多的坑让我们不得不忽略自动优先级/结合性？如果是这样，我们应当总是使用（）手动分组并移除对这些自动行为的所有依赖吗？

这种争论是非常主观的，而且和第四章中关于隐含强制转换的争论是强烈对称的。大多数开发者对这两个争论的感觉是一样的：要么他们同时接受这两种行为并使用它们编码，要么他们同时摒弃两种行为并坚持手动/明确的写法。

当然，在这个问题上，我们不能给出比我在第四章中给出的更绝对的答案。但我向你展示了利弊，并且希望促进了你更深刻的理解，以使你可以做出合理而不是人云亦云的决定。

在我看来，这里有一个重要的中间立场。我们应当将操作符优先级/结合性与（）手动分组两者混合进我们的程序——我在第四章中对于隐含的强制转换的健康/安全用法做过同样的辩论，但当然不会没有界限地仅仅拥护它。

例如，对我来说 `if (a && b && c) ..` 是完全没问题的，而我不会为了明确表现结合性而写出 `if ((a && b) && c) ..`，因为我认为这过于繁冗了。

另一方面，如果我需要链接两个`? :`条件操作符，我会理所当然地使用（）手动分组来使我意图的逻辑表达的绝对清晰。

因此，我在这里的意见和在第四章中的相似：在操作符优先级/结合性可以使代码更短更干净的地方使用操作符优先级/结合性，在（）手动分组可以帮你创建更清晰的代码并减少困惑的地方使用（）手动分组

## 自动分号

当JavaScript认为在你的JS程序中特定的地方有一个`;`时，就算你没在那里放一个`;`，它就会进行ASI（Automatic Semicolon Insertion——自动分号插入）。

为什么它这么做？因为就算你只省略了一个必需的`;`，你的程序就会失败。不是非常宽容。ASI允许JS容忍那些通常被认为是不需要`;`的特定地方省略`;`。

必须注意的是，ASI将仅在换行存在时起作用。分号不会被插入一行的中间。

基本上，如果JS解析器在解析一行时发生了解析错误（缺少一个应有的`;`），而且它可以合理的插入一个`;`，它就会这么做。什么样的地方对插入是合理的？仅在一个语句和这一行的换行之间除了空格和/或注释没有别的东西时。

考虑如下代码：

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

JS应当将下一行的 `c` 作为 `var` 语句的一部分看待吗？如果在 `b` 和 `c` 之间的任意一个地方出现一个 `,`，它当然会的。但是因为没有，所以JS认为在 `b` 后面有一个隐含的 `;`（在换行处）。如此 `c;` 就剩下来作为一个独立的表达式语句。

类似地：

```
var a = 42, b = "foo";  
  
a  
b // "foo"
```

这仍然是一个没有错误的合法程序，因为表达式语句也接受ASI。

有一些特定的地方ASI很有帮助，例如：

```
var a = 42;  
  
do {  
    // ..  
} while (a) // <-- 这里需要;  
a;
```

文法要求 `do..while` 循环后面要有一个 `;`，但是 `while` 或 `for` 循环后面则没有。但是大多数开发者都不记得它！所以ASI帮助性地介入并插入一个。

如我们在本章早先说过的，语句块儿不需要 `;` 终结，所以ASI是不必要的：

```
var a = 42;  
  
while (a) {  
    // ..  
} // <-- 这里不需要;  
a;
```

另一个ASI介入的主要情况是，与 `break`，`continue`，`return`，和(ES6) `yield` 关键字：

```
function foo(a) {
  if (!a) return;
  a *= 2;
  // ...
}
```

这个 `return` 语句的作用不会超过换行到 `a *= 2` 表达式，因为ASI认为；终结了 `return` 语句。当然，`return` 语句可以很容易地跨越多行，只要 `return` 后面不是除了换行外什么都没有就行。

```
function foo(a) {
  return (
    a * 2 + 3 / 12
  );
}
```

同样的道理也适用于 `break`，`continue`，和 `yield`。

## 纠错

在JS社区中斗得最火热的宗教战争之一（除了制表与空格以外），就是是否应当严重/唯一地依赖ASI。

大多数的，但不是全部，分号是可选的，但是 `for (...)` 循环的头部的两个；是必须的。

在这场争论的正方，许多开发者相信ASI是一种有用的机制，允许他们通过省略除了必须（很少几个）以外的所有；写出更简洁（和更“美观”）的代码。他们经常断言因为ASI使许多；成为可选的，所以一个不带它们而正确编写的程序，与带着它们而正确编写的程序没有区别。

在这场争论的反方，许多开发者将断言有太多的地方可以成为意想不到的坑了，特别是对那些新来的，缺乏经验的开发者来说，无意间被魔法般插入的；改变了程序的含义。类似地，一些开发者将会争论如果他们省略了一个分号，这就是一个直白的错误，而且他们希望他们的工具（linter等等）在JS引擎背地里纠正它之前就抓住他。

让我分享一下我的观点。仔细阅读语言规范，会发现它暗示ASI是一个纠错过程。你可能会问，什么样的错误？明确地讲，是一个解析器错误。换句话说，为了使解析器失败的少一些，ASI让它更宽容。

但是宽容什么？在我看来，一个解析器错误发生的唯一方式是，它被给予了一个不正确/错误的程序去解析。所以虽然ASI在严格地纠正解析器错误，但是它得到这样的错误的唯一方式是，程序首先就写错了——在文法要求使用分号的地方忽略了它们。

所以，更直率地讲，当我听到有人声称他们想要省略“可选的分号”时，我的大脑就将它翻译为“我想尽量编写最能破坏解析器但依然可以工作的程序。”

我发现这种立场很荒唐，而且省几下键盘敲击和更“美观的代码”的观点是软弱无力的。

进一步讲，我不同意这和空格与制表符的争论是同一种东西——那纯粹是表面上的——我宁愿相信这是一个根本问题：是编写遵循文法要求的代码，还是编写依赖于文法异常但仅仅将之忽略不计的代码。

另一种看待这个问题的方式是，依赖ASI实质上将换行视为有意义的“空格”。像Python那样的其他语言中有真正的有意义的空格。但是就今天的JavaScript来说，认为它拥有有意义的换行真的合适吗？

我的意见是：在你知道分号是“必需的”地方使用分号，并且把你对ASI的臆测限制到最小。

不要光听我的一面之词。回到2012年，JavaScript的创造者Brendan Eich说过下面的话（<http://brendaneich.com/2012/04/the-infernal-semicolon/>）：

这个故事的精神是：ASI是一种（正式地说）语法错误纠正过程。如果你在好像有一种普遍的有意义的换行的规则的前提下开始编码，你将会陷入麻烦。.. 如果回到1995年五月的那十天，我希望我使换行在JS中更有意义。.. 如果ASI好像给了JS有意义的换行，那么要小心不要使用它。

## 错误

JavaScript不仅拥有不同的错误子类型（`TypeError`，`ReferenceError`，`SyntaxError`等等），而且和其他在运行时期间发生的错误相比，它的文法还定义了在编译时被强制执行的特定错误。

尤其是，早就有许多明确的情况应当被作为“早期错误”（编译期间）被捕获和报告。任何直接的语法错误都是一个早期错误（例如，`a = ,`），而且文法还定义了一些语法上合法但是无论怎样都不允许的东西。

因为你的代码还没有开始执行，这些错误不能使用`try..catch`捕获；它们只是会在你的程序进行解析/编译时导致失败。

提示：在语言规范中没有要求浏览器（和开发者工具）到底应当怎样报告错误。所以在下面的错误例子中，对于哪一种错误的子类型会被报告或它包含什么样的错误消息，你可能会在各种浏览器中看到不同的形式，

一个简单的例子是正则表达式字面量中的语法。这里的JS语法没有错误，而是不合法的正则表达式将会抛出一个早期错误：

```
var a = /+foo/;           // 错误！
```

一个赋值的目标必须是一个标识符（或者一个产生一个或多个标识符的ES6解构表达式），所以一个像 `42` 这样的值在这个位置上是不合法的，因此可以立即被报告：

```
var a;
42 = a;           // 错误!
```

ES5的 `strict` 模式定义了更多的早期错误。例如，在 `strict` 模式中，函数参数的名称不能重复：

```
function foo(a,b,a) { }           // 还好

function bar(a,b,a) { "use strict"; } // 错误!
```

另一种 `strict` 模式的早期错误是，一个对象字面量拥有一个以上的同名属性：

```
(function(){
  "use strict";

  var a = {
    b: 42,
    b: 43
  };           // 错误!
})();
```

注意：从语义上讲，这样的错误技术上不是语法错误，而是文法错误——上面的代码段是语法上合法的。但是因为没有 `GrammarError` 类型，一些浏览器使用 `SyntaxError` 代替。

## 过早使用变量

ES6定义了一个（坦白地说，让人困惑地命名的）新的概念，称为TDZ（“Temporal Dead Zone”——时间死区）

TDZ指的是代码中还不能使用变量引用的地方，因为它还没有到完成它所必须的初始化。

对此最明白的例子就是ES6的 `let` 块儿作用域：

```
{
  a = 2;           // ReferenceError!
  let a;
}
```

赋值 `a = 2` 在变量 `a`（它确实在 `{ .. }` 块儿作用域中）被声明 `let a` 初始化之前就访问它，所以 `a` 位于TDZ中并抛出一个错误。

有趣的是，虽然 `typeof` 有一个例外，它对于未声明的变量是安全的（见第一章），但是对于 TDZ 引用却没有这样的安全例外：

```
{
  typeof a;    // undefined
  typeof b;    // ReferenceError! (TDZ)
  let b;
}
```

## 函数参数值

另一个违反 TDZ 的例子可以在 ES6 的参数默认值（参见本系列的 *ES6 与未来*）中看到：

```
var b = 3;

function foo( a = 42, b = a + b + 5 ) {
  // ..
}
```

在赋值中的 `b` 引用将在参数 `b` 的 TDZ 中发生（不会被拉到外面的 `b` 引用），所以它会抛出一个错误。然而，赋值中的 `a` 是没有问题的，因为那时参数 `a` 的 TDZ 已经过去了。

当使用 ES6 的参数默认值时，如果你省略一个参数，或者你在它的位置上传递一个 `undefined` 值的话，就会应用这个默认值。

```
function foo( a = 42, b = a + 1 ) {
  console.log( a, b );
}

foo();                // 42 43
foo( undefined );    // 42 43
foo( 5 );            // 5 6
foo( void 0, 7 );    // 42 7
foo( null );         // null 1
```

注意：在表达式 `a + 1` 中 `null` 被强制转换为值 `0`。更多信息参考第四章。

从 ES6 参数默认值的角度看，忽略一个参数和传递一个 `undefined` 值之间没有区别。然而，有一个办法可以在一些情况下探测到这种区别：

```

function foo( a = 42, b = a + 1 ) {
    console.log(
        arguments.length, a, b,
        arguments[0], arguments[1]
    );
}

foo();           // 0 42 43 undefined undefined
foo( 10 );      // 1 10 11 10 undefined
foo( 10, undefined ); // 2 10 11 10 undefined
foo( 10, null ); // 2 10 null 10 null

```

即便参数默认值被应用到了参数 `a` 和 `b` 上，但是如果没有参数传入这些值槽，数组 `arguments` 也不会有任何元素。

反过来，如果你明确地传入一个 `undefined` 参数，在数组 `argument` 中就会为这个参数存在一个元素，但它将是 `undefined`，并且与同一值槽中的被命名参数将被提供的默认值不同。

虽然ES6参数默认值会在数组 `arguments` 的值槽和相应的命名参数变量之间造成差异，但是这种脱节也会以诡异的方式发生在ES5中：

```

function foo(a) {
    a = 42;
    console.log( arguments[0] );
}

foo( 2 );      // 42 (链接了)
foo();          // undefined (没链接)

```

如果你传递一个参数，`arguments` 的值槽和命名的参数总是链接到同一个值上。如果你省略这个参数，就没有这样的链接会发生。

但是在 `strict` 模式下，这种链接无论怎样都不存在了：

```

function foo(a) {
    "use strict";
    a = 42;
    console.log( arguments[0] );
}

foo( 2 );      // 2 (没链接)
foo();          // undefined (没链接)

```

依赖于这样的链接几乎可以肯定是一个坏主意，而且事实上这种连接本身是一种抽象泄漏，它暴露了引擎的底层实现细节，而不是一个合适的设计特性。

`arguments` 数组的使用已经废弃了（特别是被ES6 ... 剩余参数取代以后——参见本系列的 *ES6与未来*），但这不意味着它都是不好的。

在ES6以前，要得到向另一个函数传递的所有参数值的数组，`arguments` 是唯一的方法，它被证实十分有用。你也可以安全地混用被命名参数和 `arguments` 数组，只要你遵循一个简单的规则：绝不能同时引用一个被命名参数和它相应的 `arguments` 值槽。如果你能避开那种错误的实践，你就永远也不会暴露这种易泄漏的链接行为。

```
function foo(a) {
    console.log( a + arguments[1] ); // 安全!
}

foo( 10, 32 ); // 42
```

## try..finally

你可能很熟悉 `try..catch` 块儿是如何工作的。但是你有没有停下来考虑过可以与之成对出现的 `finally` 子句呢？事实上，你有没有意识到 `try` 只要求 `catch` 和 `finally` 两者之一，虽然如果有需要它们可以同时出现。

在 `finally` 子句中的代码 总是 运行的（无论发生什么），而且它总是在 `try` （和 `catch`，如果存在的话）完成后立即运行，在其他任何代码之前。从一种意义上说，你似乎可以认为 `finally` 子句中的代码是一个回调函数，无论块儿中的其他代码如何动作，它总是被调用。

那么如果在 `try` 子句内部有一个 `return` 语句将会怎样？很明显它将返回一个值，对吧？但是调用端代码是在 `finally` 之前还是之后才收到这个值呢？

```
function foo() {
    try {
        return 42;
    }
    finally {
        console.log( "Hello" );
    }

    console.log( "never runs" );
}

console.log( foo() );
// Hello
// 42
```

`return 42` 立即运行，它设置好 `foo()` 调用的完成值。这个动作完成了 `try` 子句而 `finally` 子句接下来立即运行。只有这之后 `foo()` 函数才算完成，所以被返回的完成值交给 `console.log(..)` 语句使用。

对于 `try` 内部的 `throw` 来说，行为是完全相同的：

```
function foo() {
  try {
    throw 42;
  }
  finally {
    console.log( "Hello" );
  }

  console.log( "never runs" );
}

console.log( foo() );
// Hello
// Uncaught Exception: 42
```

现在，如果一个异常从 `finally` 子句中被抛出（偶然地或有意地），它将会作为这个函数的主要完成值进行覆盖。如果 `try` 块儿中的前一个 `return` 已经设置好了这个函数的完成值，那么这个值就会被抛弃。

```
function foo() {
  try {
    return 42;
  }
  finally {
    throw "Oops!";
  }

  console.log( "never runs" );
}

console.log( foo() );
// Uncaught Exception: Oops!
```

其他的诸如 `continue` 和 `break` 这样的非线性控制语句表现出与 `return` 和 `throw` 相似的行为是没什么令人吃惊的：

```
for (var i=0; i<10; i++) {
  try {
    continue;
  }
  finally {
    console.log( i );
  }
}
// 0 1 2 3 4 5 6 7 8 9
```

`console.log(i)` 语句在 `continue` 语句引起的每次循环迭代的末尾运行。然而，它依然是运行在更新语句 `i++` 之前的，这就是为什么打印出的值是 `0..9` 而非 `1..10`。

注意：ES6在generator（参见本系列的 异步与性能）中增加了 `yield` 语句，generator从某些方面可以看作是中间的 `return` 语句。然而，和 `return` 不同的是，一个 `yield` 在generator被推进前不会完成，这意味着 `try { .. yield .. }` 还没有完成。所以附着在其上的 `finally` 子句将不会像它和 `return` 一起时那样，在 `yield` 之后立即运行。

一个在 `finally` 内部的 `return` 有着覆盖前一个 `try` 或 `catch` 子句中的 `return` 的特殊能力，但是仅在 `return` 被明确调用的情况下：

```

function foo() {
    try {
        return 42;
    }
    finally {
        // 这里没有 `return ..`，所以返回值不会被覆盖
    }
}

function bar() {
    try {
        return 42;
    }
    finally {
        // 覆盖前面的 `return 42`
        return;
    }
}

function baz() {
    try {
        return 42;
    }
    finally {
        // 覆盖前面的 `return 42`
        return "Hello";
    }
}

foo();    // 42
bar();    // undefined
baz();    // "Hello"

```

一般来说，在函数中省略 `return` 和 `return;` 或者 `return undefined;` 是相同的，但是在一个 `finally` 块儿内部，`return` 的省略不是用一个 `return undefined` 覆盖；它只是让前一个 `return` 继续生效。

事实上，如果将打了标签的 `break`（在本章早先讨论过）与 `finally` 相组合，我们真的可以制造一种疯狂：

```

function foo() {
  bar: {
    try {
      return 42;
    }
    finally {
      // 跳出标记为`bar`的块儿
      break bar;
    }
  }

  console.log( "Crazy" );

  return "Hello";
}

console.log( foo() );
// Crazy
// Hello

```

但是.....别这么做。说真的。使用一个 `finally` + 打了标签的 `break` 实质上取消了 `return`，这是你在尽最大的努力制造最令人困惑的代码。我打赌没有任何注释可以拯救这段代码。

## switch

让我们简单探索一下 `switch` 语句，某种 `if..else if..else..` 语句链的语法缩写。

```

switch (a) {
  case 2:
    // 做一些事
    break;
  case 42:
    // 做另一些事
    break;
  default:
    // 这里是后备操作
}

```

如你所见，它对 `a` 求值一次，然后将结果值与每个 `case` 表达式进行匹配（这里只是一些简单的值表达式）。如果找到一个匹配，就会开始执行那个匹配的 `case`，它将会持续执行直到遇到一个 `break` 或者遇到 `switch` 块儿的末尾。

这些可能不会令你吃惊，但是关于 `switch`，有几个你以前可能从没注意过的奇怪的地方。

首先，在表达式 `a` 和每一个 `case` 表达式之间的匹配与 `==` 算法（见第四章）是相同的。`switch` 经常在 `case` 语句中使用绝对值，就像上面展示的，因此严格匹配是恰当的。

然而，你也许希望允许宽松等价（也就是 `==`，见第四章），而这么做你需要“黑”一下 `switch` 语句：

```
var a = "42";

switch (true) {
  case a == 10:
    console.log( "10 or '10'" );
    break;
  case a == 42:
    console.log( "42 or '42'" );
    break;
  default:
    // 永远不会运行到这里
}
// 42 or '42'
```

这可以工作是因为 `case` 子句可以拥有任何表达式（不仅是简单值），这意味着它将用这个表达式的结果与测试表达式（`true`）进行严格匹配。因为这里 `a == 42` 的结果为 `true`，所以匹配成功。

尽管 `==`，`switch` 的匹配本身依然是严格的，在这里是 `true` 和 `true` 之间。如果 `case` 表达式得出 `truthy` 的结果而不是严格的 `true`，它就不会工作。例如如果在你的表达式中使用 `||` 或 `&&` 这样的“逻辑操作符”，这就可能咬到你：

```
var a = "hello world";
var b = 10;

switch (true) {
  case (a || b == 10):
    // 永远不会运行到这里
    break;
  default:
    console.log( "Oops" );
}
// Oops
```

因为 `(a || b == 10)` 的结果是 `"hello world"` 而不是 `true`，所以严格匹配失败了。这种情况下，修改的方法是强制表达式明确成为一个 `true` 或 `false`，比如 `case !(a || b == 10):`（见第四章）。

最后，`default` 子句是可选的，而且它不一定非要位于末尾（虽然那是一种强烈的惯例）。即使是在 `default` 子句中，是否遇到 `break` 的规则也是一样的：

```

var a = 10;

switch (a) {
  case 1:
  case 2:
    // 永远不会运行到这里
  default:
    console.log( "default" );
  case 3:
    console.log( "3" );
    break;
  case 4:
    console.log( "4" );
}
// default
// 3

```

注意：就像我们前面讨论的 `break`，`case` 子句内部的 `break` 也可以被打标签。

这段代码的处理方式是，它首先通过所有的 `case` 子句，没有找到匹配，然后它回到 `default` 子句开始执行。因为这里没有 `break`，它会继续走进已经被跳过的块儿 `case 3`，在遇到那个 `break` 后才会停止。

虽然这种有些迂回的逻辑在JavaScript中是明显可能的，但是它几乎不可能制造出合理或易懂的代码。要对你自己是否想要创建这种环状的逻辑流程保持怀疑，如果你真的想要这么做，确保你留下了大量的代码注释来解释你要做什么！

## 复习

JavaScript文法有相当多的微妙之处，我们作为开发者应当比平常多花一点儿时间来关注它。一点儿努力可以帮助你巩固对这个语言更深层次的知识。

语句和表达式在英语中有类似的概念——语句就像句子，而表达式就像短语。表达式可以是纯粹的/自包含的，或者他们可以有副作用。

JavaScript文法层面的语义用法规则（也就是上下文），是在纯粹的语法之上的。例如，用于你程序中不同地方的 `{ }` 可以意味着块儿，`object` 字面量，（ES6）解构语句，或者（ES6）被命名的函数参数。

JavaScript操作符都有严格定义的优先级（哪一个操作符首先结合）和结合性（多个操作符表达式如何隐含地分组）规则。一旦你学会了这些规则，你就可以自己决定优先级/结合性是否是为了它们自己有利而过于明确，或者它们是否会对编写更短，更干净的代码有所助益。

ASI（自动分号插入）是一种内建在JS引擎找中的解析器纠错机制，它允许JS引擎在特定的环境下，在需要 ; 但是被省略了的地方，并且插入可以纠正解析错误时，插入一个 ; 。有一场争论是关于这种行为是否暗示着大多数 ; 都是可选的（而且为了更干净的代码可以/应当省略），或者是否它意味着省略它们是在制造JS引擎帮你扫清的错误。

JavaScript有几种类型的错误，但很少有人知道它有两种类别的错误：“早期”（编译器抛出的不可捕获的）和“运行时”（可以 try..catch 的）。所有在程序运行之前就使它停止的语法错误都明显是早期错误，但也有一些别的错误。

函数参数值与它们正式声明的命名参数之间有一种有趣的联系。明确地说，如果你不小心， arguments 数组会有一些泄漏抽象行为的坑。尽可能避开 arguments ，但如果你必须使用它，那就设法避免同时使用 arguments 中带有位置的值槽，和相同参数的命名参数。

附着在 try （或 try..catch ）上的 finally 在执行处理顺序上提供了一些非常有趣的能力。这些能力中的一些可以很有帮助，但是它也可能制造许多困惑，特别是在与打了标签的块儿组合使用时。像往常一样，为了更好更干净的代码而使用 finally ，不是为了显得更聪明或更糊涂。

switch 为 if..else if.. 语句提供了一个不错的缩写形式，但是要小心许多常见的关于它的简化假设。如果你不小心，会有几个奇怪的地方绊倒你，但是 switch 手上也有一些隐藏的高招！

# 你不懂JS：类型与文法

## 附录A：与环境混合的JavaScript

当你的JS代码在真实世界中运行时，除了我们在本书中完整探索过的核心语言机制以外，它还有好几种不同的行为方式。如果JS纯粹地运行在一个引擎中，那么它就会按照语言规范非黑即白地动作，是完全可以预测的。但是JS很可能总是运行在一个宿主环境的上下文中，这将会给你的代码带来某种程度的不可预测性。

例如，当你的代码与源自于其他地方的代码并肩运行时，或者当你的代码在不同种类的JS引擎（不只是浏览器）中运行时，有些事情的行为就可能不同。

我们将简要地探索这些问题中的一些。

## Annex B（ECMAScript）

一个鲜为人知的事实是，这门语言的官方名称是ECMAScript（意指管理它的ECMA标准本体）。那么“JavaScript”是什么？JavaScript是这门语言常见的商业名称，当然，更恰当地说，JavaScript基本上是语言规范的浏览器实现。

官方的ECMAScript语言规范包含“Annex B”，它是为了浏览器中JS的兼容性，讨论那些与官方语言规范有偏差的特别部分。

考虑这些偏差部分的恰当方法是，它们仅在你的代码运行在浏览器中时才是确实会出现/合法的。如果你的代码总是运行在浏览器中，那你就不会看到明显的不同。如果不是（比如它可以运行在node.js、Rhino中，等等），或者你不确定，那么就要小心对待。

兼容性上的主要不同是：

- 八进制数字字面量是允许的，比如在非 `strict mode` 下的 `0123`（小数 `83`）。
- `window.escape(..)` 和 `window.unescape(..)` 允许你使用 `%` 分割的十六进制转义序列来转义或非转义字符串。例如：`window.escape("?foo=97%&bar=3%")` 产生 `"%3Ffoo%3D97%25%26bar%3D3%25"`
- `String.prototype.substr` 与 `String.prototype.substring` 十分相似，除了第二个参数是 `length`（要包含的字符数），而非结束（不含）的索引。

## Web ECMAScript

Web ECMAScript语言规范(<http://javascript.spec.whatwg.org/>)涵盖了官方ECMAScript语言规范与当前浏览器中JavaScript实现之间的不同。

换言之，这些项目是浏览器的“必须品”（为了相互兼容），但是（在本书编写时）没有列在官方语言规范的“Annex B”部分是：

- `<!--` 和 `-->` 是合法的单行注释分割符。
- `String.prototype` 拥有返回HTML格式化字符串的附加方法：`anchor(..)`、`big(..)`、`blink(..)`、`bold(..)`、`fixed(..)`、`fontcolor(..)`、`fontsize(..)`、`italics(..)`、`link(..)`、`small(..)`、`strike(..)`、和 `sub(..)`。注意：它们在实际应用中非常罕见，而且一般来说不鼓励使用，而是用其他内建DOM API或用户定义的工具取代。
- `RegExp` 扩展：`RegExp.$1 .. RegExp.$9`（匹配组）和 `RegExp.lastMatch / RegExp["$&"]`（最近的匹配）。
- `Function.prototype` 附加功能：`Function.prototype.arguments`（内部 `arguments` 对象的别名）和 `Function.caller`（内部 `arguments.caller` 的别名）。注意：`arguments` 和 `arguments.caller` 都被废弃了，所以你应当尽可能避免使用它们。这些别名更是这样——不要使用它们！

注意：其他的一些微小和罕见的偏差点没有包含在我们这里的列表中。有必要的话，更多详细信息可以参见外部的“Annex B”和“Web ECMAScript”文档。

一般来说，所有这些不同点都很少被使用，所以这些与语言规范有出入的地方不是什么重大问题。只是如果你依赖于其中任何一个的话，要小心。

## 宿主对象

JS中变量的行为有一些广为人知的例外——当它们是被自动定义，或由持有你代码的环境（浏览器等）创建并提供给JS时——也就是所谓的“宿主对象”（包括 `object` 和 `function` 两者）。

例如：

```
var a = document.createElement( "div" );
typeof a;                                // "object" -- 意料之中的
Object.prototype.toString.call( a );        // "[object HTMLElement]"
a.tagName;                                // "DIV"
```

`a` 不仅是一个 `object`，而且是一个特殊的宿主对象，因为它是一个DOM元素。它拥有一个不同的内部 `[[Class]]` 值（`"HTMLDivElement"`），而且带有预定义的（而且通常是不可更改的）属性。

另一个已经在第四章的“Falsy对象”一节中探讨过的同样的怪异之处是：存在这样一些对象，当被强制转换为 `boolean` 时，它们将（令人糊涂地）被转换为 `false` 而不是预期的 `true`。

另一些需要小心的宿主对象行为包括：

- 不能访问像 `toString()` 这样的 `object` 内建方法
- 不可覆盖
- 拥有特定的预定义只读属性
- 拥有一些 `this` 不可被重载为其他对象的方法
- 其他.....

为了使我们的JS代码与它外围的环境一起工作，宿主对象至关重要。但在你与宿主对象交互时是要特别注意，并且在推测它的行为时要小心，因为它们经常与普通的JS `object` 不符。

一个尽人皆知的你可能经常与之交互的宿主对象的例子，就是 `console` 对象和他的各种函数（`log(..)`、`error(..)` 等等）。`console` 对象是由宿主环境特别提供的，所以你的代码可以与之互动来进行各种开发相关的输出任务。

在浏览器中，`console` 与开发者工具控制台的显示相勾连，因此在node.js和其他服务器端JS环境中，`console` 一般连接着JavaScript环境系统进程的标准输出流（`stdout`）和标准错误流（`stderr`）。

## 全局DOM变量

你可能知道，在全局作用域中声明变量（用或者不用 `var`）不仅会创建一个全局变量，还会创建它的镜像：在 `global` 对象（浏览器中的 `window`）上的同名属性。

但少为人知的是，（由于浏览器的遗留行为）使用 `id` 属性创建DOM元素会创建同名的全局变量。例如：

```
<div id="foo"></div>
```

和：

```
if (typeof foo == "undefined") {
    foo = 42;           // 永远不会运行
}

console.log( foo );    // HTML元素
```

你可能臆测只有JS代码会创建这样的变量，并习惯于在这样假定的前提下进行全局变量检测（使用 `typeof` 或者 `.. in window` 检查），但是如你所见，你的宿主HTML页面的内容也会创建它们，如果你不小心它们就可以轻而易举地摆脱你的存在性检查。

这就是另一个你为什么应该尽全力避免使用全局变量的原因，如果你不得不这样做，那就使用不太可能冲突的变量名。但是你还是需要确认它不会与HTML的内容以及其他代码相冲突。

## 原生原型

最广为人知的，经典的JavaScript最佳实践智慧之一是：永远不要扩展原生原型。

当你将方法或属性添加到 `Array.prototype` 时，无论你想出什么样的（还）不存在于 `Array.prototype` 上名称，如果它是有用的、设计良好的、并且被恰当命名的新增功能，那么它就有很大的可能性被最终加入语言规范——这种情况下你的扩展就处于冲突之中。

这里有一个真实地发生在我身上的例子，很好地展示了这一点。

那时我正在为其他网站建造一个可嵌入的控件，而且我的控件依赖于JQuery（虽然任何框架都很可能遭受这样的坑）。它几乎可以在每一个网站上工作，但是我们碰到了一个它会完全崩溃的网站。

经过差不多一周的分析/调试之后，我发现这个出问题的网站有这样一段代码，埋藏在它的一个遗留文件的深处：

```
// Netscape 4 没有 Array.push
Array.prototype.push = function(item) {
    this[this.length] = item;
};
```

除了那疯狂的注释（谁还会关心Netscape 4！？），它看起来很合理，对吧？

问题是，在这段 Netscape 4 时代的代码被编写之后的某个时点，`Array.prototype.push` 被加入了语言规范，但是被加入的东西与这段代码是不兼容的。标准的 `push(..)` 允许一次加入多个项目，而这个黑进来的东西会忽略后续项目。

基本上所有的JS框架都有这样的代码——依赖于带有多个元素的 `push(..)`。在我的例子中，我在围绕着一个完全被毁坏的CSS选择器引擎进行编码。但是可以料想到还有其他十几处可疑的地方。

一开始编写这个 `push(..)` 黑科技的开发者称它为 `push`，这种直觉很正确，但是没有预见到添加多个元素。当然他们的初衷是好的，但是也埋下了一个地雷，当我差不多在10年之后路过时才不知不觉地踩上。

这里要吸取几个教训。

第一，不要扩展原生类型，除非你绝对确信你的代码将是运行在那个环境中的唯一代码。如果你不能100%确信，那么扩展原生类型就是危险的。你必须掂量掂量风险。

其次，不要无条件地定义扩展（因为你可能意外地覆盖原生类型）。就这个特定的例子，用代码说话就是：

```
if (!Array.prototype.push) {
    // Netscape 4 没有 Array.push
    Array.prototype.push = function(item) {
        this[this.length] = item;
    };
}
```

`if` 守护语句将会仅在JS环境中不存在 `push()` 时才定义那个 `push()` 黑科技。在我的情况下，这可能就够了。但即便是这种方式也不是没有风险：

1. 如果网站的代码（为了某些疯狂的理由！）有赖于忽略多个项目的 `push(..)` ，那么几年以后当标准的 `push(..)` 推出时，那些代码将会坏掉。
2. 如果有其他库被引入，并在这个 `if` 守护之前就黑进了 `push(..)` ，而且还是以一种不兼容的方式，那么它就在那一刻毁坏了这个网站。

这里的重点，坦白地讲，是一个没有得到JS开发者们足够重视的有趣问题：如果在你代码运行的环境中，你的代码不是唯一的存在，那么 你应该依赖于任何原生的内建行为吗？

严格的答案是 不，但这非常不切实际。你的代码通常不会为所有它依赖的内建行为重新定义它自己的、不可接触的私有版本。即便你能，那也是相当的浪费。

那么，你应当为内建行为进行特性测试，以及为了验证它能如你预期的那样工作而进行兼容性测试吗？但如果测试失败了——你的代码应当拒绝运行吗？

```
// 不信任 Array.prototype.push
(function(){
    if (Array.prototype.push) {
        var a = [];
        a.push(1,2);
        if (a[0] === 1 && a[1] === 2) {
            // 测试通过，可以安全使用！
            return;
        }
    }

    throw Error(
        "Array#push() is missing/broken!"
    );
})();
```

理论上，这貌似有些道理，但是为每一个内建方法设计测试还是非常不切实际。

那么，我们应当怎么做？我们应当信赖但验证（特性测试和兼容性测试）每一件事吗？我们应当假设既存的东西是符合规范的并让（由他人）造成的破坏任意传播吗？

没有太好的答案。可以观察到的唯一事实是，扩展原生原型是这些东西咬到你的唯一方式。

如果你不这么做，而且在你的应用程序中也没有其他人这么做，那么你就是安全的。否则，你就应当多多少少建立一些怀疑的、悲观的机制、并对可能的破坏做好准备。

在所有已知环境中，为你的代码准备一整套单元/回归测试是发现一些前述问题的方法，但是它不会对这些冲突为你做出任何实际的保护。

## Shims/Polyfills

人们常说，扩展一个原生类型唯一安全的地方是在一个（不兼容语言规范的）老版本环境中，因为它不太可能再改变了——带有新语言规范特性的新浏览器会取代老版本浏览器，而非改良它们。

如果你能预见未来，而且确信未来的标准将是怎样，比如 `Array.prototype.foobar`，那么现在就制造你自己的兼容版本来使用就是完全安全的，对吧？

```
if (!Array.prototype.foobar) {
    // 愚蠢，愚蠢
    Array.prototype.foobar = function() {
        this.push("foo", "bar");
    };
}
```

如果已经有了 `Array.prototype.foobar` 的规范，而且规定的行为与这个逻辑等价，那么你定义这样的代码段就十分安全，在这种情况下它通常称为一个“polyfill（填补）”（或者“shim（垫片）”）。

在你的代码库中引入这样的代码，对给那些没有更新到最新规范的老版本浏览器环境打“补丁”非常有用。为所有你支持的环境创建可预见的代码，使用填补是非常好的方法。

提示：ES5-Shim (<https://github.com/es-shims/es5-shim>) 是一个将项目代码桥接至ES5基准线的完整的shims/polyfills集合，相似地，ES6-Shim (<https://github.com/es-shims/es6-shim>) 提供了ES6新增的新API的shim。虽然API可以被填补，但新的语法通常是不能的。要桥接语法的部分，你将还需要使用一个ES6到ES5的转译器，比如Traceur (<https://github.com/google/traceur-compiler/wiki/GettingStarted>)。

如果有一个即将到来的标准，而且关于它叫什么名字和它将如何工作的讨论达成了一致，那么为了兼容面向未来的标准提前创建填补，被称为“prollyfill（probably-fill——预填补）”。

真正的坑是某些标准行为不能被（完全）填补/预填补。

在开发者社区中有这样一种争论：对于常见的情况一个部分地填补是否是可接受的，或者如果一个填补不能100%地与语言规范兼容是否应当避免它。

许多开发者至少会接受一些常见的部分填补（例如 `Object.create(..)`），因为没有被填补的部分是他们不管怎样都不会用到的。

一些开发者相信，包围着 `polyfill/shim` 的 `if` 守护语句应当引入某种形式的一致性测试，在既存的方法缺失或者测试失败时取代它。这额外的一层兼容性测试有时被用于将“`shim`”（兼容性测试）与“`polyfill`”（存在性测试）区别开。

这里的要点是，没有绝对正确的答案。即使是在老版本环境中“安全地”扩展原生类型，也不是100%安全的。在其他人代码存在的情况下依赖于（可能被扩展过的）原生类型也是一样。

在这两种情况下都应当小心地使用防御性的代码，并在文档中大量记录它的风险。

## <script>

大多数通过浏览器使用的网站/应用程序都将它们的代码包含在一个以上的文件中，在一个页面中含有几个或好几个分别加载这些文件的 `<script src=..></script>` 元素，甚至几个内联的 `<script> .. </script>` 元素也很常见。

但这些分离的文件/代码段是组成分离的程序，还是综合为一个JS程序？

（也许令人吃惊）现实是它们在极大程度上，但不是全部，像独立的JS程序那样动作。

它们所共享的一个东西是一个单独的 `global` 对象（在浏览器中是 `window`），这意味着多个文件可以将它们的代码追加到这个共享的名称空间中，而且它们都是可以交互的。

所以，如果一个 `script` 元素定义了一个全局函数 `foo()`，当第二个 `script` 运行时，它就可以访问并调用 `foo()`，就好像它自己已经定义过了这个函数一样。

但是全局变量作用域提升（参见本系列的作用域与闭包）不会跨越这些界线发生，所以下面的代码将不能工作（因为 `foo()` 的声明还没有被声明过），无论它们是否是内联的 `<script> .. </script>` 元素还是外部加载的 `<script src=..></script>` 文件：

```
<script>foo();</script>

<script>
  function foo() { .. }
</script>
```

但是这两个都将可以工作：

```
<script>
  foo();
  function foo() { .. }
</script>
```

或者：

```
<script>
  function foo() { .. }
</script>

<script>foo();</script>
```

另外，如果在一个 `script` 元素（内联或者外部的）中发生了一个错误，一个分离的独立的JS 程序将会失败并停止，但是任何后续的 `script` 都将会（依然在共享的 `global` 中）畅通无阻地运行。

你可以在你的代码中动态地创建 `script` 元素，并将它们插入到页面的DOM中，它们之中的代码基本上将会像从一个分离的文件中普通地加载那样运行：

```
var greeting = "Hello World";

var el = document.createElement( "script" );

el.text = "function foo(){ alert( greeting );\
} setTimeout( foo, 1000 );";

document.body.appendChild( el );
```

注意：当然，如果你试一下上面的代码段并将 `el.src` 设置为某些文件的URL，而非将 `el.text` 设置为代码内容，你就会动态地创建一个外部加载的 `<script src=..></script>` 元素。

内联代码块中的代码，与在外部文件中的相同的代码之间的一个不同之处是，在内联的代码块中，字符 `</script>` 的序列不能一起出现，因为（无论它在哪里出现）它将会被翻译为代码块的末尾。所以，小心这样的代码：

```
<script>
  var code = "<script>alert( 'Hello World' )</script>";
</script>
```

它看起来无害，但是在 `string` 字面量中出现的 `</script>` 将会不正常地终结 `script` 块，造成一个错误。绕过它最常见的一个方法是：

```
"</sc" + "ript>";
```

另外要小心的是，一个外部文件中的代码将会根据和文件一起被提供（或默认的）的字符集编码（UTF-8、ISO-8859-8等等）来翻译，但在内联在你HTML页面中的一个 `script` 元素中的相同代码将会根据这个页面的（或它默认的）字符集编码来翻译。

警告： charset 属性在内联script元素中不能工作。

关于内联 script 元素，另一个被废弃的做法是在内联代码的周围引入HTML风格或X(HT)ML风格的注释，就像：

```
<script>
<!--
alert( "Hello" );
//-->
</script>

<script>
<!--//--><![CDATA[//><!--
alert( "World" );
//--><!--]]>
</script>
```

这两种东西现在完全是不必要的了，所以如果你还在这么做，停下！

注意：实际上纯粹是因为这种老技术，JavaScript才把 `<!--` 和 `-->` (HTML风格的注释) 两者都被规定为合法的单行注释分隔符 (`var x = 2; <!-- valid comment 和 --> another valid line comment`)。永远不要使用它们。

## 保留字

ES5语言规范在第7.6.1部分中定义了一套“保留字”，它们不能被用作独立的变量名。技术上讲，有四个类别：“关键字”，“未来保留字”，`null`字面量，以及`true / false`布尔字面量。

像`function`和`switch`这样的关键字是显而易见的。像`enum`之类的未来保留字，虽然它们中的许多（`class`、`extends`等等）现在都已经实际被ES6使用了；但还有另外一些像`interface`之类的仅在strict模式下的保留字。

StackOverflow用户“art4theSould”创造性地将这些保留字编成了一首有趣的小诗  
(<http://stackoverflow.com/questions/26255/reserved-keywords-in-javascript/12114140#12114140>)：

Let this long package float, Goto private class if short. While protected with debugger  
case, Continue volatile interface. Instanceof super synchronized throw, Extends final  
export throws.

Try import double enum?

- False, boolean, abstract function, Implements typeof transient break! Void static,  
default do, Switch int native new. Else, delete null public var In return for const,  
true, char ...Finally catch byte.

注意：这首诗包含ES3中的保留字（`byte`、`long` 等等），它们在ES5中不再被保留了。

在ES5以前，这些保留字也不能被用于对象字面量中的属性名或键，但这种限制已经不复存在了。

所以，这是不允许的：

```
var import = "42";
```

但这是允许的：

```
var obj = { import: "42" };
console.log( obj.import );
```

你应当小心，有些老版本的浏览器（主要是老IE）没有完全地遵循这些规则，所以有些将保留字用作对象属性名的地方任然会造成问题。小心地测试所有你支持的浏览器环境。

## 实现的限制

JavaScript语言规范没有在诸如函数参数值的个数，或者字符串字面量的长度上做出随意的限制，但是由于不同引擎的实现细节，无论如何这些限制是存在的。

例如：

```
function addAll() {
    var sum = 0;
    for (var i=0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}

var nums = [];
for (var i=1; i < 100000; i++) {
    nums.push(i);
}

addAll( 2, 4, 6 );           // 12
addAll.apply( null, nums ); // 应该是：499950000
```

在某些JS引擎中，你将会得到正确答案 `499950000`，但在另一些引擎中（比如Safari 6.x），你会得到一个错误：“`RangeError: Maximum call stack size exceeded.`”

已知存在的其他限制的例子：

- 在字符串字面量（不是一个字符串变量）中允许出现的最大字符个数
- 在一个函数调用的参数值中可以发送的数据的大小（字节数，也称为栈的大小）
- 在一个函数声明中的参数数量
- 没有经过优化的调用栈最大深度（比如，使用递归时）：从一个函数到另一个函数的调用链能有多长
- JS程序可以持续运行并阻塞浏览器的秒数
- 变量名的最大长度
- ...

遭遇这些限制不是非常常见，但你应当知道这些限制存在并确实会发生，而且重要的是它们因引擎不同而不同。

## 复习

我们知道并且可以依赖于这样的事实：JS语言本身拥有一个标准，而且这个标准可预见地被所有现代浏览器/引擎实现了。这是非常好的一件事！

但是JavaScript几乎不会与世隔绝地运行。它会运行在混合了第三方库的环境中运行，而且有时甚至会在不同浏览器中不同的引擎/环境中运行。

对这些问题多加注意，会改进你代码的可靠性和健壮性。

# 你不懂JS：类型与文法

## 附录B: 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David Walsh为本书作序。

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 Tregoining, 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 Hodsdon, 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 ב-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 enamorato, 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 Ziolkowski, 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 Viloslada, 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 Gharegozlu, 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。

# 你不懂JS: 异步与性能

## 目录

- 序
- 前言
- 第一章: 异步:现在与稍后
  - 块儿 (Chunks) 中的程序
  - 事件轮询 (Event Loop)
  - 并行线程
  - 并发
  - Jobs
  - 语句排序
- 第二章: 回调
  - 延续
  - 顺序的大脑
  - 信任问题
  - 尝试拯救回调
- 第三章: Promise
  - 什么是 Promise ?
  - Thenable 鸭子类型 (Duck Typing)
  - Promise的信任
  - 链式流程
  - 错误处理
  - Promise 模式
  - Promise API概览
  - Promise 的限制
- 第四章: Generator
  - 打破运行至完成
  - 生成值
  - 异步地迭代 Generator
  - Generators + Promises
  - Generator 委托
  - Generator 并发
  - Thunks
  - 前ES6时代的 Generator
- 第五章: 程序性能

- Web Workers
- SIMD
- asm.js
- 第六章：基准分析与调优
  - 基准分析 (Benchmarking)
  - 上下文为王
  - jsPerf.com
  - 编写好的测试
  - 微观性能
  - 尾部调用优化 (TCO)
- 附录A：库：asynquence
- 附录B：高级异步模式
- 附录C：鸣谢

# 你不懂JS: 异步与性能

## 序

多年以前，我的雇主十分信任我来让我进行面试。如果我们要找某些拥有JavaScript技能的人，我的问卷的第一行是...实际上这不是真的，我首先会问问应聘者是否需要上个卫生间或者喝些饮料，因为平静是很重要的，但是一旦我确信可以和应聘者进行流畅的交流，我就要开始考察这位应聘者是否懂得JavaScript，还是只懂得jQuery。

并不是jQuery有什么错。它使你不必真的懂得JavaScript就可以做很多事，这是一个特性而不是一个bug。但是如果这份工作需要关于JavaScript性能和可维护性上的高级技能，你就需要一些懂得jQuery这样的库是如何组装在一起的人。你需要能够像他们一样操控JavaScript的核心。

如果我想对某人的核心JavaScript技能取得一些了解，我最感兴趣就是他们如何使用闭包（你已经读过这个系列的那本书了，对吧？），以及如何最大限度地利用异步性，而这就是这本书带给我们的。

对于初学者，你将被带领着学习回调，它是异步编程的面包和黄油。当然，面包和黄油并不能做一顿特别令人满意的午餐，但是下一课满是非常美味的promise！

如果你不懂得promise，现在是学习的时候了。现在在JavaScript和DOM中，Promise是提供异步返回值的官方方法。所有未来的异步DOM API都将使用它们，而且有许多已经这样做了，所以做好准备！在本次写作时，Promise已经在大多数主流浏览器中获得了支持，IE也很快会支持。一旦你完成了这一课，我希望你离开教室去学习下一刻，Generator。

Generator不声不响地溜进了Chrome和Firefox的稳定版本，因为，老实说，它们的复杂程度要比有趣程度大多了。或者说，直到我看到它们与promise组合起来之前我都是这么认为的。在此，它们成为了增强可读性和可维护性的重要工具。

至于甜点，好吧，我不会把惊喜放坏了，准备好凝视JavaScript的未来吧！许多特性在并发性和异步性上给了你越来越多的控制权。

好吧，我不会继续挡着你享受这本书了，让好戏开始吧！如果你已经在读这篇序之前度过了这本书的一些部分，给你10点异步加分！你值得拥有！

Jake Archibald

[jakearchibald.com](http://jakearchibald.com), @jaffathecake

Google Chrome 技术推广部



# 你不懂JS：异步与性能

## 第一章：异步：现在与稍后

在像JavaScript这样的语言中最重要但经常被误解的编程技术之一，就是如何表达和操作跨越一段时间的程序行为。

这不仅仅是关于从 `for` 循环开始到 `for` 循环结束之间发生的事情，当然它确实要花一些时间（几微妙到几毫秒）才能完成。它是关于你的程序 现在 运行的部分，和你的程序 稍后 运行的另一部分之间发生的事情——现在 和 稍后 之间有一个间隙，在这个间隙中你的程序没有活跃地执行。

几乎所有被编写过的（特别是用JS）大型程序都不得不使用这样或那样的方法来管理这个间隙，不管是等待用户输入，从数据库或文件系统请求数据，通过网络发送数据并等待应答，还是在规定的时间间隔重复某些任务（比如动画）。在所有这些各种方法中，你的程序都不得不跨越时间间隙管理状态。就像在伦敦众所周知的一句话（地铁门与月台间的缝隙）：“小心间隙。”

实际上，你程序中 现在 与 稍后 的部分之间的关系，就是异步编程的核心。

可以确定的是，异步编程在JS的最开始就出现了。但是大多数开发者从没认真地考虑过它到底是如何，为什么出现在他们的程序中的，也没有探索过其他处理异步的方式。足够好的方法总是老实巴交的回调函数。今天还有许多人坚持认为回调就绰绰有余了。

但是JS在使用范围和复杂性上不停地生长，作为运行在浏览器，服务器和每种可能的设备上的头等编程语言，为了适应它不断扩大的要求，我们在管理异步上感受到的痛苦日趋严重，人们迫切地需要一种更强大更合理的处理方法。

虽然眼前这一切看起来很抽象，但我保证，随着我们通读这本书你会更完整且坚实地解决它。在接下来的几章中我们将会探索各种异步JavaScript编程的新兴技术。

但在接触它们之前，我们将不得不更深刻地理解异步是什么，以及它在JS中如何运行。

## 块儿（Chunks）中的程序

你可能将你的JS程序写在一个 `.js` 文件中，但几乎可以确定你的程序是由几个代码块儿构成的，仅有其中的一个将会在 现在 执行，而其他的将会在 稍后 执行。最常见的代码块儿单位是 `function`。

大多数刚接触JS的开发者都可能会有的问题是，稍后 并不严格且立即地在 现在 之后发生。换句话说，根据定义，现在 不能完成的任务将会异步地完成，而且我们因此不会有你可能在直觉上期望或想要的阻塞行为。

考虑这段代码：

```
// ajax(..)是某个包中任意的Ajax函数
var data = ajax( "http://some.url.1" );

console.log( data );
// 噢！`data`一般不会有Ajax的结果
```

你可能意识到Ajax请求不会同步地完成，这意味着 `ajax(..)` 函数还没有任何返回的值可以赋值给变量 `data`。如果 `ajax(..)` 在应答返回之前能够阻塞，那么 `data = ..` 赋值将会正常工作。

但那不是我们使用Ajax的方式。我们现在 制造一个异步的Ajax请求，直到 稍后 我们才会得到结果。

从 现在“等到” 稍后 最简单的（但绝对不是唯一的，或最好的）方法，通常称为回调函数：

```
// ajax(..) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", function myCallbackFunction(data){

    console.log( data ); // Yay, 我得到了一些`data`!

});
```

警告：你可能听说过发起同步的Ajax请求是可能的。虽然在技术上是这样的，但你永远，永远不应该在任何情况下这样做，因为它将锁定浏览器的UI（按钮，菜单，滚动条，等等）而且阻止用户与任何东西互动。这是一个非常差劲的主意，你应当永远回避它。

在你提出抗议之前，不，你渴望避免混乱的回调不是使用阻塞的，同步的Ajax的正当理由。

举个例子，考虑下面的代码：

```

function now() {
    return 21;
}

function later() {
    answer = answer * 2;
    console.log("Meaning of life:", answer);
}

var answer = now();

setTimeout(later, 1000); // Meaning of life: 42

```

这个程序中有两个代码块儿：现在 将会运行的东西，和 稍后 将会运行的东西。这两个代码块分别是什么应当十分明显，但还是让我们以最明确的方式指出来：

现在：

```

function now() {
    return 21;
}

function later() { .. }

var answer = now();

setTimeout(later, 1000);

```

稍后：

```

answer = answer * 2;
console.log("Meaning of life:", answer);

```

你的程序一执行，现在 代码块儿就会立即运行。但 `setTimeout(..)` 还设置了一个 稍后 会发生的事件（一个超时事件），所以 `later()` 函数的内容将会在一段时间后（从现在开始1000毫秒）被执行。

每当你将一部分代码包进 `function` 并且规定它应当为了响应某些事件而执行（定时器，鼠标点击，Ajax应答等等），你就创建了一个 稍后 代码块儿，也因此在你的程序中引入了异步。

## 异步控制台

关于 `console.*` 方法如何工作，没有相应的语言规范或一组需求——它们不是JavaScript官方的一部分，而是由宿主环境添加到JS上的（见本丛书的 [类型与文法](#)）。

所以，不同的浏览器和JS环境各自为战，这有时会导致令人困惑的行为。

特别地，有些浏览器和某些条件下，`console.log(..)` 实际上不会立即输出它得到的东西。这个现象的主要原因可能是因为I/O处理很慢，而且是许多程序的阻塞部分（不仅是JS）。所以，对一个浏览器来说，可能的性能更好的处理方式是（从网页/UI的角度看），在后台异步地处理 `console I/O`，而你也许根本不知道它发生了。

虽然不是很常见，但是一种可能被观察到（不是从代码本身，而是从外部）的场景是：

```
var a = {
  index: 1
};

// 稍后
console.log( a ); // ??

// 再稍后
a.index++;
```

我们一般希望看到的是，就在 `console.log(..)` 语句被执行的那一刻，对象 `a` 被取得一个快照，打印出如 `{ index: 1 }` 的内容，如此在下一个语句 `a.index++` 执行时，它修改不同于 `a` 的输出，或者严格的在 `a` 的输出之后的某些东西。

大多数时候，上面的代码将会在你的开发者工具控制台中产生一个你期望的对象表现形式。但是同样的代码也可能运行在这样的情况下：浏览器告诉后台它需要推迟控制台I/O，这时，在对象在控制台中被表示的那个时间点，`a.index++` 已经执行了，所以它将显示 `{ index: 2 }`。

到底在什么条件下 `console I/O` 将被推迟是不确定的，甚至它能不能被观察到都是不确定的。只能当你在调试过程中遇到问题时——对象在 `console.log(..)` 语句之后被修改，但你却意外地看到了修改后的内容——意识到I/O的这种可能的异步性。

注意：如果你遇到了这种罕见的情况，最好的选择是使用JS调试器的断点，而不是依赖 `console` 的输出。第二好的选择是通过将目标对象序列化为一个 `string` 强制取得一个它的快照，比如用 `JSON.stringify(..)`。

## 事件轮询（Event Loop）

让我们来做一个（也许是令人震惊的）声明：尽管明确地允许异步JS代码（就像我们刚看到的超时），但是实际上，直到最近（ES6）为止，JavaScript本身从来没有任何内建的异步概念。

什么！？这听起来简直是疯了，对吧？事实上，它是真的。JS引擎本身除了在某个在被要求的时刻执行你程序的一个单独的代码块外，没有做过任何其他的事情。

“被‘谁’要求”？这才是重要的部分！

JS引擎没有运行在隔离的区域。它运行在一个宿主环境中，对大多数开发者来说这个宿主环境就是浏览器。在过去的几年中（但不特指这几年），JS超越了浏览器的界限进入到了其他环境中，比如服务器，通过Node.js这样的东西。其实，今天JavaScript已经被嵌入到所有种类的设备中，从机器人到电灯泡儿。

所有这些环境的一个共通的“线程”（一个“不那么微妙”的异步玩笑，不管怎样）是，他们都有一种机制：在每次调用JS引擎时，可以随着时间的推移执行你的程序的多个代码块儿，这称为“事件轮询（Event Loop）”。

换句话说，JS引擎对时间没有天生的感觉，反而是一个任意JS代码段的按需执行环境。是它周围的环境在不停地安排“事件”（JS代码的执行）。

那么，举例来说，当你的JS程序发起一个从服务器取得数据的Ajax请求时，你在一个函数（通常称为回调）中建立好“应答”代码，然后JS引擎就会告诉宿主环境，“嘿，我就要暂时停止执行了，但不管你什么时候完成了这个网络请求，而且你还得到一些数据的话，请回来调这个函数。”

然后浏览器就会为网络的应答设置一个监听器，当它有东西要交给你的时候，它会通过将回调函数插入事件轮询来安排它的执行。

那么什么是事件轮询？

让我们先通过一些假想代码来对它形成一个概念：

```
// `eventLoop`是一个像队列一样的数组（先进先出）
var eventLoop = [ ];
var event;

// “永远”执行
while (true) {
    // 执行一个"tick"
    if (eventLoop.length > 0) {
        // 在队列中取得下一个事件
        event = eventLoop.shift();

        // 现在执行下一个事件
        try {
            event();
        }
        catch (err) {
            reportError(err);
        }
    }
}
```

当然，这只是一个用来展示概念的大幅简化的假想代码。但是对于帮助我们建立更好的理解来说应该够了。

如你所见，有一个通过 `while` 循环来表现的持续不断的循环，这个循环的每一次迭代称为一个“tick”。在每一个“tick”中，如果队列中有一个事件在等待，它就会被取出执行。这些事件就是你的函数回调。

很重要并需要注意的是，`setTimeout(..)` 不会将你的回调放在事件轮询队列上。它设置一个定时器；当这个定时器超时的时候，环境才会把你的回调放进事件轮询，这样在某个未来的 tick 中它将会被取出执行。

如果在那时事件轮询队列中已经有了 20 个事件会怎么样？你的回调要等待。它会排到队列最后——没有一般的方法可以插队和跳到队列的最前方。这就解释了为什么 `setTimeout(..)` 计时器可能不会完美地按照预计时间触发。你得到一个保证（粗略地说）：你的回调不会再你指定的时间间隔之前被触发，但是可能会在这个时间间隔之后被触发，具体要看事件队列的状态。

换句话说，你的程序通常被打断成许多小的代码块儿，它们一个接一个地在事件轮询队列中执行。而且从技术上说，其他与你的程序没有直接关系的事件也可以穿插在队列中。

注意：我们提到了“直到最近”，暗示着 ES6 改变了事件轮询队列在何处被管理的性质。这主要是一个正式的技术规范，ES6 现在明确地指出了事件轮询应当如何工作，这意味着它技术上属于 JS 引擎应当关心的范畴内，而不仅仅是宿主环境。这么做的一个主要原因是为了引入 ES6 的 Promises（我们将在第三章讨论），因为人们需要有能力对事件轮询队列的排队操作进行直接，细粒度的控制（参见“协作”一节中关于 `setTimeout(..)` 的讨论）。

## 并行线程

将“异步”与“并行”两个词经常被混为一谈，但它们实际上是十分不同的。记住，异步是关于现在与稍后之间的间隙。但并行是关于可以同时发生的事情。

关于并行计算最常见的工具就是进程与线程。进程和线程独立地，可能同时地执行：在不同的处理器上，甚至在不同的计算机上，而多个线程可以共享一个进程的内存资源。

相比之下，一个事件轮询将它的任务打碎成一系列任务并串行地执行它们，不允许并行访问和更改共享的内存。并行与“串行”可能以在不同线程上的事件轮询协作的形式共存。

并行线程执行的穿插，与异步事件的穿插发生在完全不同的粒度等级上：

比如：

```
function later() {
  answer = answer * 2;
  console.log( "Meaning of life:", answer );
}
```

虽然 `later()` 的整个内容将被当做一个事件轮询队列的实体，但当考虑到将要执行这段代码的线程时，实际上也许会有许多不同的底层操作。比如，`answer = answer * 2` 首先需要读取当前 `answer` 的值，再把 `2` 放在某个地方，然后进行乘法计算，最后把结果存回到 `answer`。

在一个单线程环境中，线程队列中的内容都是底层操作真的无关紧要，因为没有什么可以打断线程。但如果你有一个并行系统，在同一个程序中有两个不同的线程，你很可能会得到无法预测的行为：

考虑这段代码：

```
var a = 20;

function foo() {
    a = a + 1;
}

function bar() {
    a = a * 2;
}

// ajax(..) 是一个给定的库中的随意Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

在JavaScript的单线程行为下，如果 `foo()` 在 `bar()` 之前执行，结果 `a` 是 `42`，但如果 `bar()` 在 `foo()` 之前执行，结果 `a` 将是 `41`。

如果JS事件共享相同的并列执行数据，问题将会变得微妙得多。考虑这两个假想代码段，它们分别描述了运行 `foo()` 和 `bar()` 中代码的线程将要执行的任务，并考虑如果它们在完全相同的时刻运行会发生什么：

线程1（`x` 和 `y` 是临时的内存位置）：

```
foo():
    a. 将`a`的值读取到`X`
    b. 将`1`存入`Y`
    c. 把`X`和`Y`相加，将结果存入`X`
    d. 将`X`的值存入`a`
```

线程2（`x` 和 `y` 是临时的内存位置）：

```
bar():
    a. 将`a`的值读取到`X`
    b. 将`2`存入`Y`
    c. 把`X`和`Y`相乘，将结果存入`X`
    d. 将`X`的值存入`a`
```

现在，让我们假定这两个线程在并行执行。你可能发现了问题，对吧？它们在临时的步骤中使用共享的内存位置 `x` 和 `y`。

如果步骤像这样发生，`a` 的最终结果什么？

```
1a (将`a`的值读取到`x`    ==> `20`)
2a (将`a`的值读取到`x`    ==> `20`)
1b (将`1`存入`y`    ==> `1`)
2b (将`2`存入`y`    ==> `2`)
1c (把`x`和`y`相加，将结果存入`x`    ==> `22`)
1d (将`x`的值存入`a`    ==> `22`)
2c (把`x`和`y`相乘，将结果存入`x`    ==> `44`)
2d (将`x`的值存入`a`    ==> `44`)
```

`a` 中的结果将是 `44`。那么这种顺序呢？

```
1a (将`a`的值读取到`x`    ==> `20`)
2a (将`a`的值读取到`x`    ==> `20`)
2b (将`2`存入`y`    ==> `2`)
1b (将`1`存入`y`    ==> `1`)
2c (把`x`和`y`相乘，将结果存入`x`    ==> `20`)
1c (把`x`和`y`相加，将结果存入`x`    ==> `21`)
1d (将`x`的值存入`a`    ==> `21`)
2d (将`x`的值存入`a`    ==> `21`)
```

`a` 中的结果将是 `21`。

所以，关于线程的编程十分刁钻，因为如果你不采取特殊的步骤来防止这样的干扰/穿插，你会得到令人非常诧异的，不确定的行为。这通常让人头疼。

JavaScript从不跨线程共享数据，这意味着不必关心这一层的不确定性。但这并不意味着JS总是确定性的。记得前面 `foo()` 和 `bar()` 的相对顺序产生两个不同的结果吗（`41` 或 `42`）？

注意：可能还不明显，但不是所有的不确定性都是坏的。有时候它无关紧要，有时候它是故意的。我们会在本章和后续几章中看到更多的例子。

## 运行至完成

因为JavaScript是单线程的，`foo()`（和`bar()`）中的代码是原子性的，这意味着一旦`foo()`开始运行，它的全部代码都会在`bar()`中的任何代码可以运行之前执行完成，反之亦然。这称为“运行至完成”行为。

事实上，运行至完成的语义会在`foo()`与`bar()`中有更多的代码时更明显，比如：

```

var a = 1;
var b = 2;

function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}

// ajax(..) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );

```

因为 `foo()` 不能被 `bar()` 打断，而且 `bar()` 不能被 `foo()` 打断，所以这个程序根据哪一个先执行只有两种可能的结果——如果线程存在，`foo()` 和 `bar()` 中的每一个语句都可能被穿插，可能的结果数量将会极大地增长！

代码块儿1是同步的（现在发生），但代码块儿2和3是异步的（稍后发生），这意味着它们的执行将会被时间的间隙分开。

代码块儿1：

```

var a = 1;
var b = 2;

```

代码块儿2（`foo()`）：

```

a++;
b = b * a;
a = b + 3;

```

代码块儿3（`bar()`）：

```

b--;
a = 8 + b;
b = a * 2;

```

代码块儿2和3哪一个都有可能先执行，所以这个程序有两个可能的结果，正如这里展示的：

结果1：

```

var a = 1;
var b = 2;

// foo()
a++;
b = b * a;
a = b + 3;

// bar()
b--;
a = 8 + b;
b = a * 2;

a; // 11
b; // 22

```

结果2：

```

var a = 1;
var b = 2;

// bar()
b--;
a = 8 + b;
b = a * 2;

// foo()
a++;
b = b * a;
a = b + 3;

a; // 183
b; // 180

```

同一段代码有两种结果仍然意味着不确定性！但是这是在函数（事件）顺序的水平上，而不是在使用线程时语句顺序的水平上（或者说，实际上是表达式操作的顺序上）。换句话说，他比线程更具有确定性。

当套用到JavaScript行为时，这种函数顺序的不确定性通常称为“竞合状态”，因为 `foo()` 和 `bar()` 在互相竞争看谁会先运行。明确地说，它是一个“竞合状态”因为你不能可靠地预测 `a` 与 `b` 将如何产生。

注意：如果在JS中不知怎的有一个函数没有运行至完成的行为，我们会有更多可能的结果，对吧？ES6中引入一个这样的东西（见第四章“生成器”），但现在不要担心，我们会回头讨论它。

## 并发

让我们想象一个网站，它显示一个随着用户向下滚动而逐步加载的状态更新列表（就像社交网络的新消息）。要使这样的特性正确工作，（至少）需要两个分离的“进程”同时执行（在同一个时间跨度内，但没必要是同一个时间点）。

注意：我们在这里使用带引号的“进程”，因为它们不是计算机科学意义上的真正的操作系统级别的进程。它们是虚拟进程，或者说任务，表示一组逻辑上关联，串行顺序的操作。我们将简单地使用“进程”而非“任务”，因为在术语层面它与我们讨论的概念的定义相匹配。

第一个“进程”将响应当用户向下滚动页面时触发的 `onscroll` 事件（发起取得新内容的Ajax请求）。第二个“进程”将接收返回的Ajax应答（将内容绘制在页面上）。

显然，如果用户向下滚动的足够快，你也许会看到在第一个应答返回并处理期间，有两个或更多的 `onscroll` 事件被触发，因此你将使 `onscroll` 事件和Ajax应答事件迅速触发，互相穿插在一起。

并发是当两个或多个“进程”在同一时间段内同时执行，无论构成它们的各个操作是否并行地（在同一时刻不同的处理器或内核）发生。你可以认为并发是“进程”级别的（或任务级别）的并行机制，而不是操作级别的并行机制（分割进程的线程）。

注意：并发还引入了这些“进程”间彼此互动的概念。我们稍后会讨论它。

在一个给定的时间跨度内（用户可以滚动的那几秒），让我们将每个独立的“进程”作为一系列事件/操作描绘出来：

“线程”1 (`onscroll` 事件):

```
onscroll, request 1
onscroll, request 2
onscroll, request 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
onscroll, request 7
```

“线程”2 (Ajax应答事件):

```
response 1
response 2
response 3
response 4
response 5
response 6
response 7
```

一个 `onscroll` 事件与一个 Ajax 应答事件很有可能在同一个时刻都准备好被处理了。比如我们在一个时间线上描绘一下这些事件的话：

```
onscroll, request 1
onscroll, request 2           response 1
onscroll, request 3           response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6           response 4
onscroll, request 7
response 6
response 5
response 7
```

但是，回到本章前面的事件轮询概念，JS 一次只能处理一个事件，所以不是 `onscroll`, `request 2` 首先发生就是 `response 1` 首先发生，但是他们不可能完全在同一时刻发生。就像学校食堂的孩子们一样，不管他们在门口挤成什么样，他们最后都不得不排成一个队来打饭！

让我们来描绘一下所有这些事件在事件轮询队列上穿插的情况：

事件轮询队列：

```
onscroll, request 1    <--- 进程1开始
onscroll, request 2
response 1             <--- 进程2开始
onscroll, request 3
response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
response 4
onscroll, request 7    <--- 进程1结束
response 6
response 5
response 7             <--- 进程2结束
```

“进程1”和“进程2”并发地运行（任务级别的并行），但是它们的个别事件在事件轮询队列上顺序地运行。

顺便说一句，注意到 `response 6` 和 `response 5` 没有按照预想的顺序应答吗？

单线程事件轮询是并发的一种表达（当然还有其他的表达，我们稍后讨论）。

## 非互动

在同一个程序中两个或更多的“进程”在穿插它们的步骤/事件时，如果它们的任务之间没有联系，那么他们就没必要互动。如果它们不互动，不确定性就是完全可以接受的。

举个例子：

```
var res = {};  
  
function foo(results) {  
    res.foo = results;  
}  
  
function bar(results) {  
    res.bar = results;  
}  
  
// ajax(..) 是某个包中任意的Ajax函数  
ajax( "http://some.url.1", foo );  
ajax( "http://some.url.2", bar );
```

`foo()` 和 `bar()` 是两个并发的“进程”，而且它们被触发的顺序是不确定的。但对我们的程序的结构来讲它们的触发顺序无关紧要，因为它们的行为相互独立所以不需要互动。

这不是一个“竞合状态”Bug，因为这段代码总能够正确工作，与顺序无关。

## 互动

更常见的是，通过作用域和/或DOM，并发的“进程”将有必要间接地互动。当这样的互动将要发生时，你需要协调这些互动行为来防止前面讲述的“竞合状态”。

这里是两个由于隐含的顺序而互动的并发“进程”的例子，它有时会出错：

```
var res = [];  
  
function response(data) {  
    res.push( data );  
}  
  
// ajax(..) 是某个包中任意的Ajax函数  
ajax( "http://some.url.1", response );  
ajax( "http://some.url.2", response );
```

并发的“进程”是那两个将要处理Ajax应答的 `response()` 调用。它们谁都有可能先发生。

假定我们期望的行为是 `res[0]` 拥有 “`http://some.url.1`” 调用的结果，而 `res[1]` 拥有 “`http://some.url.2`” 调用的结果。有时候结果确实是这样，而有时候则相反，要看哪一个调用首先完成。很有可能，这种不确定性是一个“竞合状态”Bug。

注意：在这些情况下要极其警惕你可能做出的主观臆测。比如这样的情况就没什么不寻常：一个开发者观察到 "http://some.url.2" 的应答“总是”比 "http://some.url.1" 要慢得多，也许有赖于它们所做的任务（比如，一个执行数据库任务而另一个只是取得静态文件），所以观察到的顺序看起来总是所期望的。就算两个请求都发到同一个服务器，而且它故意以确定的顺序应答，也不能真正保证应答回到浏览器的顺序。

所以，为了解决这样的竞合状态，你可以协调互动的顺序：

```
var res = [];

function response(data) {
    if (data.url == "http://some.url.1") {
        res[0] = data;
    }
    else if (data.url == "http://some.url.2") {
        res[1] = data;
    }
}

// ajax(..) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

无论哪个Ajax应答首先返回，我们都考察它的 `data.url`（当然，假设这样的数据会从服务器返回）来找到应答数据应当在 `res` 数组中占有的位置。`res[0]` 将总是持有 "`http://some.url.1`" 的结果，而 `res[1]` 将总是持有 "`http://some.url.2`" 的结果。通过简单的协调，我们消除了“竞合状态”的不确定性。

这个场景的同样道理可以适用于这样的情况：多个并发的函数调用通过共享的DOM互动，比如一个在更新 `<div>` 的内容而另一个在更新 `<div>` 的样式或属性（比如一旦DOM元素拥有内容就使它变得可见）。你可能不想在DOM元素拥有内容之前显示它，所以协调工作就必须保证正确顺序的互动。

没有协调的互动，有些并发的场景 总是出错（不仅仅是有时）。考虑下面的代码：

```

var a, b;

function foo(x) {
    a = x * 2;
    baz();
}

function bar(y) {
    b = y * 2;
    baz();
}

function baz() {
    console.log(a + b);
}

// ajax(..) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );

```

在这个例子中，不管 `foo()` 和 `bar()` 谁先触发，总是会使 `baz()` 运行的太早了（`a` 和 `b` 之一还是空的时候），但是第二个 `baz()` 调用将可以工作，因为 `a` 和 `b` 将都是可用的。

有许多不同的方法可以解决这个状态。这是简单的一种：

```

var a, b;

function foo(x) {
    a = x * 2;
    if (a && b) {
        baz();
    }
}

function bar(y) {
    b = y * 2;
    if (a && b) {
        baz();
    }
}

function baz() {
    console.log( a + b );
}

// ajax(..) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );

```

`baz()` 调用周围的 `if (a && b)` 条件通常称为“大门”，因为我们不能确定 `a` 和 `b` 到来的顺序，但在打开大门（调用 `baz()`）之前我们等待它们全部到达。

另一种你可能会遇到的并发互动状态有时称为“竞争”，单更准确地说应该叫“门闩”。它的行为特点是“先到者胜”。在这里不确定性是可以接受的，因为你明确指出“竞争”的终点线上只有一个胜利者。

考虑这段有问题的代码：

```
var a;

function foo(x) {
    a = x * 2;
    baz();
}

function bar(x) {
    a = x / 2;
    baz();
}

function baz() {
    console.log(a);
}

// ajax(..) 是某个包中任意的Ajax函数
ajax("http://some.url.1", foo);
ajax("http://some.url.2", bar);
```

不管哪一个函数最后触发（`foo()` 或 `bar()`），它不仅会覆盖前一个函数对 `a` 的赋值，还会重复调用 `baz()`（不太可能是期望的）。

所以，我们可以用一个简单的门闩来协调互动，仅让第一个过去：

```

var a;

function foo(x) {
  if (a == undefined) {
    a = x * 2;
    baz();
  }
}

function bar(x) {
  if (a == undefined) {
    a = x / 2;
    baz();
  }
}

function baz() {
  console.log(a);
}

// ajax(..) 是某个包中任意的Ajax函数
ajax("http://some.url.1", foo);
ajax("http://some.url.2", bar);

```

`if (a == undefined)` 条件仅会让 `foo()` 或 `bar()` 中的第一个通过，而第二个（以及后续所有的）调用将会被忽略。第二名什么也得不到！

注意：在所有这些场景中，为了简化说明的目的我们都用了全局变量，这里我们没有任何理由需要这么做。只要我们讨论中的函数可以访问变量（通过作用域），它们就可以正常工作。依赖于词法作用域变量（参见本丛书的作用域与闭包），和这些例子中实质上的全局变量，是这种并发协调形式的一个明显的缺点。在以后的几章中，我们会看到其他的在这方面干净得多的协调方法。

## 协作

另一种并发协调的表达称为“协作并发”，它并不那么看重在作用域中通过共享值互动（虽然这依然是允许的！）。它的目标是将一个长时间运行的“进程”打断为许多步骤或批处理，以至于其他的并发“进程”有机会将它们的操作穿插进事件轮询队列。

举个例子，考虑一个Ajax应答处理器，它需要遍历一个很长的结果列表来将值变形。我们将使用 `Array#map(..)` 来让代码短一些：

```
var res = [];

// `response(...)`从Ajax调用收到一个结果数组
function response(data) {
    // 连接到既存的`res`数组上
    res = res.concat(
        // 制造一个新的变形过的数组，所有的`data`值都翻倍
        data.map( function(val){
            return val * 2;
        })
    );
}

// ajax(...) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

如果 "http://some.url.1" 首先返回它的结果，整个结果列表将会一次性映射进 `res`。如果只有几千或更少的结果记录，一般来说不是什么大事。但假如有1千万个记录，那么就可能会花一段时间运行（在强大的笔记本电脑上花几秒钟，在移动设备上花的时间长得多，等等）。

当这样的“处理”运行时，页面上没有任何事情可以发生，包括不能有另一个 `response(..)` 调用，不能有UI更新，甚至不能有用户事件比如滚动，打字，按钮点击等。非常痛苦。

所以，为了制造协作性更强、更友好而且不独占事件轮询队列的并发系统，你可以在一个异步批处理中处理这些结果，在批处理的每一步都“让出”事件轮询来让其他等待的事件发生。

这是一个非常简单的方法：

```

var res = [];

// `response(...)`从Ajax调用收到一个结果数组
function response(data) {
    // 我们一次只处理1000件
    var chunk = data.splice( 0, 1000 );

    // 连接到既存的`res`数组上
    res = res.concat(
        // 制造一个新的变形过的数组，所有的`data`值都翻倍
        chunk.map( function(val){
            return val * 2;
        })
    );

    // 还有东西要处理吗？
    if (data.length > 0) {
        // 异步规划下一个批处理
        setTimeout( function(){
            response( data );
        }, 0 );
    }
}

// ajax(...) 是某个包中任意的Ajax函数
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );

```

我们以每次最大1000件作为一个块儿处理数据。这样，我们保证每个“进程”都是短时间运行的，即便这意味着会有许多后续的“进程”，在事件轮询队列上的穿插将会给我们一个响应性（性能）强得多的网站/应用程序。

当然，我们没有对任何这些“进程”的顺序进行互动协调，所以在 `res` 中的结果的顺序是不可预知的。如果要求顺序，你需要使用我们之前讨论的互动技术，或者在本书后续章节中介绍的其他技术。

我们使用 `setTimeout(..0)`（黑科技）来异步排程，基本上它的意思是“将这个函数贴在事件轮询队列的末尾”。

注意：从技术上讲，`setTimeout(..0)` 没有直接将一条记录插入事件轮询队列。计时器将会在下一个运行机会将事件插入。比如，两个连续的 `setTimeout(..0)` 调用不会严格保证以调用的顺序被处理，所以我们可能看到各种时间偏移的情况，使这样的事件的顺序是不可预知的。在Node.js中，一个相似的方式是 `process.nextTick(..)`。不管那将会有多少方便（而且通常性能更好），（还）没有一个直接的方法可以横跨所有环境来保证异步事件顺序。我们会在下一节详细讨论这个话题。

# Jobs

在ES6中，在事件轮询队列之上引入了一层新概念，称为“工作队列（Job queue）”。你最有可能接触它的地 方是在Promises（见第三章）的异步行为中。

不幸的是，它目前是一个没有公开API的机制，因此要演示它有些兜圈子。我们不得不仅在概念上描述它，这样当我们在第三章中讨论异步行为时，你将会理解那些动作行为是如何编程与处理的。

那么，我能找到的考虑它的最佳方式是：“工作队列”是一个挂靠在事件轮询队列的每个tick末尾的队列。在事件轮询的一个tick期间内，某些可能发生的隐含异步动作的行为将不会导致一个全新的事件加入事件轮询队列，而是在当前tick的工作队列的末尾加入一个新的记录（也就是一个Job）。

它好像是在说，“哦，另一件需要我稍后去做的事情，但是保证它在其他任何事情发生之间发生。”

或者，用一个比喻：事件轮询队列就像一个游乐园项目，一旦你乘坐完一次，你就不得不去队尾排队来乘坐下一次。而工作队列就像乘坐完后，立即插队乘坐下一次。

一个Job还可能会导致更多的Job被加入同一个队列的末尾。所以，一个在理论上可能的情况是，Job“轮询”（一个Job持续不断地加入其他Job等）会无限地转下去，从而拖住程序不能移动到下一个事件轮询tick。这与在你的代码中表达一个长时间运行或无限循环（比如 `while (true) ...`）在概念上几乎是一样的。

Job的精神有点儿像 `setTimeout(..0)` 黑科技，但以一种定义明确得多的方式实现，而且保证顺序：稍后，但尽快。

让我们想象一个用于Job排程的API，并叫它 `schedule(..)`。考虑如下代码：

```
console.log( "A" );

setTimeout( function(){
    console.log( "B" );
}, 0 );

// 理论上的 "Job API"
schedule( function(){
    console.log( "C" );

    schedule( function(){
        console.log( "D" );
    } );
} );
```

你肯能会期望它打印出 `A B C D`，但是它将会打出 `A C D B`，因为Job发生在当前的事件轮询 `tick` 的末尾，而定时器会在下一个事件轮询 `tick`（如果可用的话！）触发排程。

在第三章中，我们会看到 `Promises` 的异步行为是基于 `Job` 的，所以搞明白它与事件轮询行为的联系是很重要的。

## 语句排序

我们在代码中表达语句的顺序没有必要与 JS 引擎执行它们的顺序相同。这可能看起来像是个奇怪的论断，所以我们简单地探索一下。

但在我们开始之前，我们应当对一些事情十分清楚：从程序的角度看，语言的规则/文法（参见本丛书的 `类型与文法`）为语句的顺序决定了一个非常可预知、可靠的行为。所以我们将要讨论的是在你的 JS 程序中 应当永远观察不到的东西。

**警告：**如果你曾经 观察到 过我们将要描述的编译器语句重排，那明显是违反了语言规范，而且无疑是那个JS引擎的Bug——它应当被报告并且修复！但是更常见的是你 怀疑 JS 引擎里发生了什么疯狂的事，而事实上它只是你自己代码中的一个Bug（可能是一个“竞合状态”）——所以先检查那里，多检查几遍。在 JS 调试器使用断点并一行一行地步过你的代码，将是帮你 在你的代码 中找出这样的Bug的最强大的工具。

考虑下面的代码：

```
var a, b;

a = 10;
b = 30;

a = a + 1;
b = b + 1;

console.log( a + b ); // 42
```

这段代码没有任何异步表达（除了早先讨论的罕见的 `console` 异步I/O），所以最有可能的推 测是它会一行一行地、从上到下地处理。

但是，JS引擎 有可能，在编译完这段代码后（是的，JS是被编译的——见本丛书的 `作用域与闭包`）发现有机会通过（安全地）重新安排这些语句的顺序来使你的代码运行得更快。实质 上，只要你观察不到重排，一切都是合理的。

举个例子，引擎可能会发现如果实际上这样执行代码会更快：

```
var a, b;  
  
a = 10;  
a++;  
  
b = 30;  
b++;  
  
console.log( a + b ); // 42
```

或者是这样：

```
var a, b;  
  
a = 11;  
b = 31;  
  
console.log( a + b ); // 42
```

或者甚至是：

```
// 因为`a`和`b`都不再被使用，我们可以内联而且根本不需要它们！  
console.log( 42 ); // 42
```

在所有这些情况下，JS引擎在它的编译期间进行着安全的优化，而最终的 可观察到 的结果将是相同的。

但也有一个场景，这些特殊的优化是不安全的，因而也是不被允许的（当然，不是说它一点儿都没优化）：

```
var a, b;  
  
a = 10;  
b = 30;  
  
// 我们需要`a`和`b`递增之前的状态！  
console.log( a * b ); // 300  
  
a = a + 1;  
b = b + 1;  
  
console.log( a + b ); // 42
```

编译器重排会造成可观测的副作用（因此绝不会被允许）的其他例子，包括任何带有副作用的函数调用（特别是getter函数），或者ES6的Proxy对象（参见本丛书的 *ES6与未来*）。

考虑如下代码：

```

function foo() {
  console.log( b );
  return 1;
}

var a, b, c;

// ES5.1 getter 字面语法
c = {
  get bar() {
    console.log( a );
    return 1;
  }
};

a = 10;
b = 30;

a += foo();           // 30
b += c.bar;          // 11

console.log( a + b ); // 42

```

如果不是为了这个代码段中的 `console.log(..)` 语句（只是作为这个例子中观察副作用的方便形式），JS引擎将会更加自由，如果它想（谁知道它想不想！？），它会重排这段代码：

```

// ...

a = 10 + foo();
b = 30 + c.bar;

// ...

```

多亏JS语义，我们不会观测到看起来很危险的编译器语句重排，但是理解源代码被编写的方式（从上到下）与它在编译后运行的方式之间的联系是多么微弱，依然很重要的。

编译器语句重排几乎是并发与互动的微型比喻。作为一个一般概念，这样的意识可以帮你更好地理解异步JS代码流问题。

## 复习

一个JavaScript程序总是被打断为两个或更多的代码块儿，第一个代码块儿现在运行，下一个代码块儿稍后运行，来响应一个事件。虽然程序是一块儿一块儿地被执行的，但它们都共享相同的程序作用域和状态，所以对状态的每次修改都是在前一个状态之上的。

不论何时有事件要运行，事件轮询将运行至队列为空。事件轮询的每次迭代称为一个“tick”。用户交互，IO，和定时器会将事件在事件队列中排队。

在任意给定的时刻，一次只有一个队列中的事件可以被处理。当事件执行时，他可以直接或间接地导致一个或更多的后续事件。

并发是当两个或多个事件链条随着事件相互穿插，因此从高层的角度来看，它们在同时运行（即便在给定的某一时刻只有一个事件在被处理）。

在这些并发“进程”之间进行某种形式的互动协调通常是有必要的，比如保证顺序或防止“竞合状态”。这些“进程”还可以协作：通过将它们自己打断为小的代码块儿来允许其他“进程”穿插。

# 你不懂JS：异步与性能

## 第二章：回调

在第一章中，我们探讨了JavaScript中关于异步编程的术语和概念。我们的焦点是理解驱动所有“事件”（异步函数调用）的单线程（一次一个）事件轮询队列。我们还探讨了各种解释同时运行的事件链，或“进程”（任务，函数调用等）间的关系的并发模式。

我们在第一章的所有例子中，将函数作为独立的，不可分割的操作单位使用，在这些函数内部语句按照可预知的顺序运行（在编译器水平之上！），但是在函数顺序水平上，事件（也就是异步函数调用）可以以各种顺序发生。

在所有这些情况中，函数都是一个“回调”。因为无论什么时候事件轮询队列中的事件被处理时，这个函数都作为事件轮询“调用并返回”程序的目标。

正如你观察到的，在JS程序中，回调是到目前为止最常见的表达和管理异步的方式。确实，在JavaScript语言中回调是最基础的异步模式。

无数的JS程序，即便是最精巧最复杂的程序，都曾经除了回调外不依靠任何其他异步模式而编写（当然，和我们在第一章中探讨的并发互动模式一起）。回调函数是JavaScript的异步苦工，而且它工作得相当好。

除了……回调并不是没有缺点。许多开发者都对 *Promises* 提供的更好的异步模式感到兴奋不已。但是如果你不明白它在抽象什么，和为什么抽象，是不可能有效利用任何抽象机制的。

在本章中，我们将深入探讨这些话题，来说明为什么更精巧的异步模式（在本书的后续章节中探讨）是必要和被期望的。

## 延续

让我们回到在第一章中开始的异步回调的例子，但让我稍微修改它一下来画出重点：

```
// A
ajax( "...", function(...){
  // C
} );
// B
```

// A 和 // B 代表程序的前半部分（也就是现在），// C 标识了程序的后半部分（也就是稍后）。前半部分立即执行，然后会出现一个不知多久的“暂停”。在未来某个时刻，如果Ajax调用完成了，那么程序会回到它刚才离开的地方，并继续执行后半部分。

换句话说，回调函数包装或封装了程序的延续。

让我们把代码弄得更简单一些：

```
// A
setTimeout( function(){
    // C
}, 1000 );
// B
```

稍停片刻然后问你自己，你将如何描述（给一个不那么懂JS工作方式的人）这个程序的行为。来吧，大声说出来。这个很好的练习将使我的下一个观点更鲜明。

现在大多数读者可能在想或说着这样的话：“做A，然后设置一个等待1000毫秒的定时器，一旦它触发，就做C”。与你的版本有多接近？

你可能已经发觉了不对劲儿的地方，给了自己一个修正版：“做A，设置一个1000毫秒的定时器，然后做B，然后在超时事件触发后，做C”。这比第一个版本更准确。你能发现不同之处吗？

虽然第二个版本更准确，但是对于以一种将我们的大脑匹配代码，代码匹配JS引擎的方式讲解这段代码来说，这两个版本都是不足的。这里的鸿沟既是微小的也是巨大的，而且是理解回调作为异步表达和管理的关键。

只要我们以回调函数的方式引入一个延续（或者像许多程序员那样引入几十个！），我们就允许了一个分歧在我们的大脑如何工作和代码将运行的方式之间形成。当这两者背离时，我们的代码就不可避免地陷入这样的境地：更难理解，更难推理，更难调试，和更难维护。

## 顺序的大脑

我相信大多数读者都曾经听某个人说过（甚至你自己就曾这么说），“我能一心多用”。试图表现得一心多用的效果包含幽默（孩子们的拍头揉肚子游戏），平常的行为（边走边嚼口香糖），和彻头彻尾的危险（开车时发微信）。

但我们是一心多用的人吗？我们真的能执行两个意识，有意地一起行动并在完全同一时刻思考/推理它们两个吗？我们最高级的大脑功能有并行的多线程功能吗？

答案可能令你吃惊：可能不是这样。

我们的大脑其实就不是这样构成的。我们中大多数人（特别是A型人格！）都是自己不情愿承认的一个一心一用者。其实我们只能在任一给定的时刻考虑一件事情。

我不是说我们所有的下意识，潜意识，大脑的自动功能，比如心跳，呼吸，和眨眼。那些都是我们延续生命的重要任务，我们不会有意识地给它们分配大脑的能量。谢天谢地，当我们在3分钟内第15次刷朋友圈时，我们的大脑在后台（线程！）继续着这些重要任务。

相反我们讨论的是在某时刻我们的意识最前线的任务。对我来说，是现在正在写这本书。我还在这完全同一个时刻做其他高级的大脑活动吗？不，没有。我很快而且容易分心——在这最后的几段中有几十次了！

当我们模拟一心多用时，比如试着在打字的同时和朋友或家人通电话，实际上我们表现得更像一个快速环境切换器。换句话说，我们快速交替地在两个或更多任务间来回切换，在微小，快速的区块中同时处理每个任务。我们做的是如此之快，以至于从外界看开我们在平行地做这些事情。

难道这听起来不像异步事件并发吗（就像JS中发生的那样）？！如果不，回去再读一遍第一章！

事实上，将庞大复杂的神经内科世界简化为我希望可以在这里讨论的东西的一个方法是，我们的大脑工作起来有点儿像事件轮询队列。

如果你把我打得每一个字（或词）当做一个单独的异步事件，那么现在这一句话上就有十几处地方，可以让我的大脑被其他的事件打断，比如我的感觉，甚至只是我随机的想法。

我不会在每个可能的地方被打断并被拉到其他的“处理”上去（谢天谢地——要不这本书永远也写不完了！）。但是它发生得也足够频繁，以至于我感到我的大脑几乎持续不断地切换到各种不同的环境（也就是“进程”）。而且这和JS引擎可能会感觉到的十分相像。

## 执行与计划

好了，这么说来我们的大脑可以被认为是运行在一个单线程事件轮询队列中，就像JS引擎那样。这听起来是个不错的匹配。

但是我们需要比我们刚才分析的更加细致入微。在我们如何计划各种任务，和我们的大脑实际如何运行这些任务之间，有一个巨大，明显的不同。

再一次，回到这篇文章的写作的比拟上来。在我心里的粗略计划轮廓是继续写啊写，顺序地经过一系列在我思想中定好的点。我没有在这次写作期间计划任何的打扰或非线性的活动。但无论如何，我的大脑依然一直不停地切换。

即便在操作级别上我们的大脑是异步事件的，但我们还是用一种顺序的，同步的方式计划任务。“我得去商店，然后买些牛奶，然后去干洗店”。

你会注意到这种高级思维（规划）方式看起来不是那么“异步”。事实上，我们几乎很少会故意只用事件的形式思考。相反，我们小心，顺序地（A然后B然后C）计划，而且我们假设一个区间有某种临时的阻塞迫使B等待A，使C等待B。

当开发者编写代码时，他们规划一组将要发生的动作。如果他们是合格的开发者，他们会小心地规划。比如“我需要将 `z` 的值设为 `x` 的值，然后将 `x` 的值设为 `y` 的值”。

当我们编写同步代码时，一个语句接一个语句，它工作起来就像我们的跑腿todo清单：

```
// 交换`x`与`y`（通过临时变量`z`）
z = x;
x = y;
y = z;
```

这三个赋值语句是同步的，所以 `x=y` 会等待 `z=x` 完成，而 `y=z` 会相应地等待 `x=y` 完成。另一种说法是这三个语句临时地按照特定的顺序绑在一起执行，一个接一个。幸好我们不必在这里关心任何异步事件的细节。如果我们关心，代码很快就会变得非常复杂！

如果同步的大脑规划和同步的代码语句匹配的很好，那么我们的大脑能把异步代码规划得多好呢？

事实证明，我们在代码中表达异步的方式（用回调）和我们同步的大脑规划行为根本匹配的不是很好。

你能实际想象一下像这样规划你的跑腿todo清单的思维线索吗？

“我得去趟商店，但是我确信在路上我会接到一个电话，于是‘嗨，妈妈’，然后她开始讲话，我会在GPS上搜索商店的位置，但那会花几分钟加载，所以我把收音机音量调小以便听到妈妈讲话，然后我发现我忘了穿夹克而且外面很冷，但没关系，继续开车并和妈妈说话，然后安全带警报提醒我要系好，于是‘是的，妈，我系着安全带呢，我总是系着安全带！’。啊，GPS终于得到方向了，现在……”

虽然作为我们如何度过自己的一天，思考以什么顺序做什么事的规划听起来很荒唐，但这正是我们大脑在功能层面运行的方式。记住，这不是一心多用，而只是快速的环境切换。

我们这些开发者编写异步事件代码困难的原因，特别是当我们只有回调手段可用时，就是意识思考/规划的流动对我们大多数人是不自然的。

我们用一步接一步的方式思考，但是一旦我们从同步走向异步，在代码中可以用的工具（回调）不是以一步接一步的方式表达的。

而且这就是为什么正确编写和推理使用回调的异步JS代码是如此困难：因为它不是我们的大脑进行规划的工作方式。

注意：唯一比不知道为什么代码不好用更糟糕的是，从一开始就不知道为什么代码好用！这是一种经典的“纸牌屋”心理：“它好用，但不知为什，所以大家都别碰！”你可能听说过，“他人即地狱”（萨特），而程序员们模仿这种说法，“他人的代码即地狱”。我相信：“不明白我自己的代码才是地狱。”而回调正是肇事者之一。

## 嵌套/链接的回调

考虑下面的代码：

```
listen( "click", function handler(evt){
    setTimeout( function request(){
        ajax( "http://some.url.1", function response(text){
            if (text == "hello") {
                handler();
            }
            else if (text == "world") {
                request();
            }
        });
    }, 500);
});
```

你很可能一眼就能认出这样的代码。我们得到了三个嵌套在一起的函数链，每一个函数都代表异步序列（任务，“进程”）的一个步骤。

这样的代码常被称为“回调地狱（callback hell）”，有时也被称为“末日金字塔（pyramid of doom）”（由于嵌套的缩进使它看起来像一个放倒的三角形）。

但是“回调地狱”实际上与嵌套/缩进几乎无关。它是一个深刻得多的问题。我们将继续在本章剩下的部分看到它为什么和如何成为一个问题。

首先，我们等待“click”事件，然后我们等待定时器触发，然后我们等待Ajax应答回来，就在这时它可能会将所有这些再做一遍。

猛地一看，这段代码的异步性质可能看起来与顺序的大脑规划相匹配。

首先（现在），我们：

```
listen( "...", function handler(...){
    // ...
});
```

稍后，我们：

```
setTimeout( function request(...){
    // ...
}, 500);
```

再稍后，我们：

```
ajax( "...", function response(...){
    // ...
});
```

最后（最稍后），我们：

```
if ( ... ) {
    // ...
}
else ...
```

不过用这样的方式线性推导这段代码有几个问题。

首先，这个例子中我们的步骤在一条顺序的线上（1，2，3，和4.....）是一个巧合。在真实的异步JS程序中，经常会有许多噪音把事情搞乱，在我们从一个函数跳到下一个函数时不得不在大脑中把这些噪音快速地演练一遍。理解这样满载回调的异步流程不是不可能，但绝不自然或容易，即使是经历了很多练习后。

而且，有些更深层的，只是在这段代码中不明显的东西搞错了。让我们建立另一个场景（假想代码）来展示它：

```
doA( function(){
    doB();

    doC( function(){
        doD();
    } )

    doE();
} );

doF();
```

虽然根据经验你将正确地指出这些操作的真实顺序，但我打赌它第一眼看上去有些使人糊涂，而且需要一些协调的思维周期才能搞明白。这些操作将会以这种顺序发生：

- doA()
- doF()
- doB()
- doC()
- doE()
- doD()

你是在第一次浏览这段代码就看明白的吗？

好吧，你们肯定有些人在想我在函数的命名上不公平，故意引导你误入歧途。我发誓我只是按照从上到下出现的顺序命名的。不过让我再试一次：

```
doA( function(){
    doC();
    doD( function(){
        doF();
    } )
    doE();
} );
doB();
```

现在，我以他们实际执行的顺序用字母命名了。但我依然要打赌，即便是现在对这个场景有经验的情况下，大多数读者追踪 A -> B -> C -> D -> E -> F 的顺序并不是自然而然的。你的眼睛肯定在这段代码中上上下下跳了许多次，对吧？

就算它对你来说都是自然的，这里依然还有一个可能肆虐的灾难。你能发现它是什么吗？

如果 doA(..) 或 doD(..) 实际上不是如我们明显地假设的那样，不是异步的呢？嗯，现在顺序不同了。如果它们都是同步的（也许仅仅有时是这样，根据当时程序所处的条件而定），现在的顺序是 A -> C -> D -> F -> E -> B<sup>o</sup>。

你在背景中隐约听到的声音，正是成千上万双手掩面的JS开发者的叹息。

嵌套是问题吗？是它使追踪异步流程变得这么困难吗？当然，有一部分是。

但是让我不用嵌套重写一遍前面事件/超时/Ajax嵌套的例子：

```
listen( "click", handler );

function handler() {
    setTimeout( request, 500 );
}

function request(){
    ajax( "http://some.url.1", response );
}

function response(text){
    if (text == "hello") {
        handler();
    }
    else if (text == "world") {
        request();
    }
}
```

这样的代码组织形式几乎看不出来有前一种形式的嵌套/缩进困境，但它的每一处依然容易受到“回调地狱”的影响。为什么呢？

当我们线性地（顺序地）推理这段代码，我们不得不从一个函数跳到下一个函数，再跳到下一个函数，并在代码中弹来弹去以“看到”顺序流。并且要记住，这个简化的代码风格是某种最佳情况。我们都应该知道真实的JS程序代码经常更加神奇地错综复杂，使这样量级的顺序推理更加困难。

另一件需要注意的事是：为了将第2, 3, 4步链接在一起使他们相继发生，回调独自给我们的启示是将第2步硬编码在第1步中，将第3步硬编码在第2步中，将第4步硬编码在第3步中，如此继续。硬编码不一定是一件坏事，如果第2步应当总是在第3步之前真的是一个固定条件。

不过硬编码绝对会使代码变得更脆弱，因为它不考虑任何可能使在步骤前行的过程中出现偏差的异常情况。举个例子，如果第2步失败了，第3步永远不会到达，第2步也不会重试，或者移动到一个错误处理流程上，等等。

所有这些问题你都可以手动硬编码在每一步中，但那样的代码总是重复性的，而且不能在其他步骤或你程序的其他异步流程中复用。

即便我们的大脑可能以顺序的方式规划一系列任务（这个，然后这个，然后这个），但我们大脑运行的事件的性质，使恢复/重试/分流这样的流程控制几乎毫不费力。如果你出去购物，而且你发现你把购物单忘在家里了，这并不会因为你没有提前计划这种情况而结束这一天。你的大脑会很容易地绕过这个小问题：你回家，取购物单，然后回头去商店。

但是手动硬编码的回调（甚至带有硬编码的错误处理）的脆弱本性通常不那么优雅。一旦你最终指明了（也就是提前规划好了）所有各种可能性/路径，代码就会变得如此复杂以至于几乎不能维护或更新。

这才是“回调地狱”想表达的！嵌套/缩进基本上一个余兴表演，转移注意力的东西。

如果以上这些还不够，我们还没有触及两个或更多这些回调延续的链条同时发生会怎么样，或者当第三步分叉称为带有大门或门闩的“并行”回调，或者……我的天哪，我脑子疼，你呢？

你抓住这里的重点了吗？我们顺序的，阻塞的大脑规划行为和面向回调的异步代码不能很好地匹配。这就是需要清楚地阐明的关于回调的首要缺陷：它们在代码中表达异步的方式，是需要我们的大脑不得不斗争才能保持一致的。

## 信任问题

在顺序的大脑规划和JS代码中回调驱动的异步处理间的不匹配只是关于回调的问题的一部分。还有一些更深刻的问题值得担忧。

让我们再一次重温这个概念——回调函数是我们程序的延续（也就是程序的第二部分）：

```
// A
ajax( "...", function(...){
    // C
} );
// B
```

// A 和 // B 现在发生，在JS主程序的直接控制之下。但是 // c 被推迟到稍后发生，并且在另一部分的控制之下——这里是 ajax(..) 函数。在基本的感觉上，这样的控制交接一般不会让程序产生很多问题。

但是不要被这种控制切换不是什么大事的罕见情况欺骗了。事实上，它是回调驱动的设计的最可怕的（也是最微妙的）问题。这个问题围绕着一个想法展开：有时 ajax(..)（或者说你向之提交回调的部分）不是你写的函数，或者不是你可以直接控制的函数。很多时候它是一个由第三方提供的工具。

当你把你程序的一部分拿出来并把它执行的控制权移交给另一个第三方时，我们称这种情况为“控制倒转”。在你的代码和第三方工具之间有一个没有明言的“契约”——一组你期望被维护的东西。

## 五个回调的故事

为什么这件事情很重要可能不是那么明显。让我们来构建一个夸张的场景来生动地描绘一下信任危机。

想象你是一个开发者，正在建造一个贩卖昂贵电视的网站的结算系统。你已经将结算系统的各种页面顺利地制造完成。在最后一个页面，当用户点解“确定”购买电视时，你需要调用一个第三方函数（假如由一个跟踪分析公司提供），以便使这笔交易能够被追踪。

你注意到它们提供的是某种异步追踪工具，也许是最佳的性能，这意味着你需要传递一个回调函数。在你传入的这个程序的延续中，有你最后的代码——划客人的信用卡并显示一个感谢页面。

这段代码可能看起来像这样：

```
analytics.trackPurchase( purchaseData, function(){
    chargeCreditCard();
    displayThankyouPage();
} );
```

足够简单，对吧？你写好代码，测试它，一切正常，然后你把它部署到生产环境。大家都很开心！

6个月过去了，没有任何问题。你几乎已经忘了你曾写过的代码。一天早上，工作之前你先在咖啡店坐坐，悠闲地享用着你的拿铁，直到你接到老板慌张的电话要求你立即扔掉咖啡并冲进办公室。

当你到达时，你发现一位高端客户为了买同一台电视信用卡被划了5次，而且可以理解，他不高兴。客服已经道了歉并开始办理退款。但你的老板要求知道这是怎么发生的。“我们没有测试过这样的情况吗！？”

你甚至不记得你写过的代码了。但你还是往回挖掘试着找出是什么出错了。

在分析过一些日志之后，你得出的结论是，唯一的解释是分析工具不知怎么的，由于某些原因，将你的回调函数调用了5次而非一次。他们的文档中没有任何东西提到此事。

十分令人沮丧，你联系了客户支持，当然他们和你一样惊讶。他们同意将此事向上提交至开发者，并许诺给你回复。第二天，你收到一封很长的邮件解释他们发现了什么，然后你将它转发给了你的老板。

看起来，分析公司的开发者曾经制作了一些实验性的代码，在一定条件下，将会每秒重试一次收到的回调，在超时之前共计5秒。他们从没想要把这部分推到生产环境，但不知怎地他们这样做了，而且他们感到十分难堪而且抱歉。然后是许多他们如何定位错误的细节，和他们将要如何做以保证此事不再发生。等等，等等。

后来呢？

你找你的老板谈了此事，但是他对事情的状态不是感觉特别舒服。他坚持，而且你也勉强地同意，你不能再相信他们了（咬到你的东西），而你将需要指出如何保护放出的代码，使它们不再受这样的漏洞威胁。

修修补补之后，你实现了一些如下的特殊逻辑代码，团队中的每个人看起来都挺喜欢：

```
var tracked = false;

analytics.trackPurchase( purchaseData, function(){
    if (!tracked) {
        tracked = true;
        chargeCreditCard();
        displayThankyouPage();
    }
});
```

注意：对读过第一章的你来说这应当很熟悉，因为我们实质上创建了一个门闩来处理我们的回调被并发调用多次的情况。

但一个QA的工程师问，“如果他们没调你的回调怎么办？”噢。谁也没想过。

你开始布下天罗地网，考虑在他们调用你的回调时所有出错的可能性。这里是得到的分析工具可能不正常运行的方式的大致列表：

- 调用回调过早（在它开始追踪之前）
- 调用回调过晚（或不调）
- 调用回调太少或太多次（就像你遇到的问题！）
- 没能向你的回调传递必要的环境/参数
- 吞掉了可能发生的错误/异常
- ...

这感觉像是一个麻烦清单，因为它就是。你可能慢慢开始理解，你将要不得不为每一个传递到你不能信任的工具中的回调都创造一大堆的特殊逻辑。

现在你更全面地理解了“回调地狱”有多地狱。

## 不仅是其他人的代码

现在有些人可能会怀疑事情到底是不是如我所宣扬的这么大条。也许你根本就不和真正的第三方工具互动。也许你用的是进行了版本控制的API，或者自己保管的库，因此它的行为不会在你不知晓的情况下改变。

那么，好好思考这个问题：你能真正信任你理论上控制（在你的代码库中）的工具吗？

这样考虑：我们大多数人都同意，至少在某个区间内我们应当带着一些防御性的输入参数检查制造我们自己的内部函数，来减少/防止以外的问题。

过于相信输入：

```
function addNumbers(x,y) {
  // + 操作符使用强制转换重载为字符串连接
  // 所以根据传入参数的不同，这个操作不是严格的安全。
  return x + y;
}

addNumbers( 21, 21 );    // 42
addNumbers( 21, "21" );  // "2121"
```

防御不信任的输入：

```

function addNumbers(x,y) {
    // 保证数字输入
    if (typeof x != "number" || typeof y != "number") {
        throw Error( "Bad parameters" );
    }

    // 如果我们到达这里，+ 就可以安全地做数字加法
    return x + y;
}

addNumbers( 21, 21 );      // 42
addNumbers( 21, "21" );    // Error: "Bad parameters"

```

或者也许依然安全但更友好：

```

function addNumbers(x,y) {
    // 保证数字输入
    x = Number( x );
    y = Number( y );

    // + 将会安全地执行数字加法
    return x + y;
}

addNumbers( 21, 21 );      // 42
addNumbers( 21, "21" );    // 42

```

不管你怎么做，这类函数参数的检查/规范化是相当常见的，即便是我们理论上完全信任的代码。用一个粗俗的说法，编程好像是地缘政治学的“信任但验证”原则的等价物。

那么，这不是要推论出我们应当对异步函数回调的编写做相同的事，而且不仅是针对真正的外部代码，甚至要对一般认为是“在我们控制之下”的代码？我们当然应该。

但是回调没有给我们提供任何协助。我们不得不自己构建所有的装置，而且这通常最终成为许多我们要在每个异步回调中重复的模板/负担。

有关于回调的最麻烦的问题就是 控制反转 导致所有这些信任完全崩溃。

如果你有代码用到回调，特别是但不特指第三方工具，而且你还没有为所有这些 控制反转 的信任问题实施某些缓和逻辑，那么你的代码现在就有 bug，虽然它们还没咬到你。将来的bug 依然是bug。

确实是地狱。

## 尝试拯救回调

有几种回调的设计试图解决一些（不是全部！）我们刚才看到的信任问题。这是一种将回调模式从它自己的崩溃中拯救出来的勇敢，但注定失败的努力。

举个例子，为了更平静地处理错误，有些API设计提供了分离的回调（一个用作成功的通知，一个用作错误的通知）：

```
function success(data) {
    console.log( data );
}

function failure(err) {
    console.error( err );
}

ajax( "http://some.url.1", success, failure );
```

在这种设计的API中，`failure()` 错误处理器通常是可选的，而且如果不提供的话它会假定你想让错误被吞掉。呃。

注意：ES6的Promises的API使用的就是这种分离回调设计。我们将在下一章中详尽地讨论ES6的Promises。

另一种常见的回调设计模式称为“错误优先风格”（有时称为“Node风格”，因为它几乎在所有的Node.js的API中作为惯例使用），一个回调的第一个参数为一个错误对象保留（如果有的话）。如果成功，这个参数将会是空/falsey（而其他后续的参数将是成功的数据），但如果出现了错误的结果，这第一个参数就会被设置/truthy（而且通常没有其他东西会被传递了）：

```
function response(err,data) {
    // 有错？
    if (err) {
        console.error( err );
    }
    // 否则，认为成功
    else {
        console.log( data );
    }
}

ajax( "http://some.url.1", response );
```

这两种方法都有几件事情应当注意。

首先，它们没有像看起来那样真正解决主要的信任问题。在这两个回调中没有关于防止或过滤意外的重复调用的东西。而且，事情现在更糟糕了，因为你可能同时得到成功和失败信号，或者都得不到，你仍然不得不围绕着这两种情况写代码。

还有，不要忘了这样的事实：虽然它们是你可以引用的标准模式，但它们绝对更加繁冗，而且是不太可能复用的模板代码，所以你将会对在你应用程序的每一个回调中敲出它们感到厌倦。

回调从不被调用的信任问题怎么解决？如果这要紧（而且它可能应当要緊！），你可能需要设置一个超时来取消事件。你可以制作一个工具来帮你：

```
function timeoutify(fn,delay) {
  var intv = setTimeout( function(){
    intv = null;
    fn( new Error( "Timeout!" ) );
  }, delay )
;

  return function() {
    // 超时还没有发生？
    if (intv) {
      clearTimeout( intv );
      fn.apply( this, [ null ].concat( [].slice.call( arguments ) ) );
    }
  };
}
```

这是你如何使用它：

```
// 使用“错误优先”风格的回调设计
function foo(err,data) {
  if (err) {
    console.error( err );
  }
  else {
    console.log( data );
  }
}

ajax( "http://some.url.1", timeoutify( foo, 500 ) );
```

另一个信任问题是被调用的“过早”。在应用程序规范上讲，这可能涉及在某些重要的任务完成之前被调用。但更一般地，在那些即可以现在（同步地），也可以稍后（异步地）调用你提供的回调的工具中这个问题更明显。

这种围绕着同步或异步行为的不确定性，几乎总是导致非常难追踪的Bug。在某些圈子中，一个名叫Zalgo的可以导致人精神错乱的虚构怪物被用来描述这种同步/异步的噩梦。经常能听到人们喊“别放出Zalgo！”，而且它引出了一个非常响亮的建议：总是异步地调用回调，即使它是“立即”在事件轮询的下一个迭代中，这样所有的回调都是可预见的异步。

注意：更多关于Zalgo的信息，参见Oren Golan的“Don't Release Zalgo!（不要释放Zalgo！）”(<https://github.com/oren/oren.github.io/blob/master/posts/zalgo.md>)和Isaac Z. Schlueter的“Designing APIs for Asynchrony（异步API设计）”(<http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony>)。

考虑下面的代码：

```
function result(data) {
    console.log( a );
}

var a = 0;

ajax( "..pre-cached-url..", result );
a++;
```

这段代码是打印 0 (同步回调调用) 还是打印 1 (异步回调调用) ? 这.....要看情况。

你可以看到Zalgo的不可预见性能有多快地威胁你的JS程序。所以听起来傻呼呼的“别放出Zalgo”实际上是一个不可思议地常见且实在的建议——总是保持异步。

如果你不知道当前的API是否会总是异步地执行呢？你可以制造一个像 `asyncify(..)` 这样的工具：

```

function asyncify(fn) {
  var orig_fn = fn,
    intv = setTimeout( function(){
      intv = null;
      if (fn) fn();
    }, 0 )
  ;

  fn = null;

  return function() {
    // 触发太快，在`intv`计时器触发来
    // 表示异步回合已经过去之前？
    if (intv) {
      fn = orig_fn.bind.apply(
        orig_fn,
        // 将包装函数的`this`加入`bind(..)`调用的
        // 参数，同时currying其他所有的传入参数
        [this].concat( [].slice.call( arguments ) )
      );
    }
    // 已经是异步
    else {
      // 调用原版的函数
      orig_fn.apply( this, arguments );
    }
  };
}

```

你像这样使用 `asyncify(..)` :

```

function result(data) {
  console.log( a );
}

var a = 0;

ajax( "..pre-cached-url..", asyncify( result ) );
a++;

```

不管Ajax请求是由于存在于缓存中而解析为立即调用回调，还是它必须走过网线去取得数据而异步地稍后完成，这段代码总是输出 `1` 而不是 `0` —— `result(..)` 总是被异步地调用，这意味着 `a++` 有机会在 `result(..)` 之前运行。

噢耶，又一个信任问题被“解决了”！但它很低效，而且又有更多臃肿的模板代码让你的项目变得沉重。

这只是关于回调一遍又一遍地发生的故事。它们几乎可以做任何你想做的事，但你不得不努力工作来达到目的，而且大多数时候这种努力比你应当在推理这样的代码上所付出的多得多。

你可能发现自己希望有一些内建的API或语言机制来解决这些问题。终于ES6带着一个伟大的答案到来了，所以继续读下去！

## 复习

回调是JS中异步的基础单位。但是随着JS的成熟，它们对于异步编程的演化趋势来讲显得不够。

首先，我们的大脑用顺序的，阻塞的，单线程的语义方式规划事情，但是回调使用非线性，非顺序的方式表达异步流程，这使我们正确推理这样的代码变得非常困难。不好推理的代码是导致不好的Bug的不好的代码。

我们需要一个种方法，以更同步化，顺序化，阻塞的方式来表达异步，正如我们的大脑那样。

第二，而且是更重要的，回调遭受着 控制反转 的蹂躏，它们隐含地将控制权交给第三方（通常第三方工具不受你控制！）来调用你程序的 延续。这种控制权的转移使我们得到一张信任问题的令人不安的列表，比如回调是否会比我们期望的被调用更多次。

制造特殊的逻辑来解决这些信任问题是可能的，但是它比它应有的难度高多了，还会产生更笨重和更难维护的代码，而且在bug实际咬到你的时候代码会显得在这些危险上被保护的不够。

我们需要一个 所有这些信任问题 的一般化解决方案。一个可以被所有我们制造的回调复用，而且没有多余的模板代码负担的方案。

我们需要比回调更好的东西。目前为止它们做的不错，但JavaScript的 未来 要求更精巧和强大的异步模式。本书的后续章节将会深入这些新兴的发展变化。

# 你不懂JS: 异步与性能

## 第三章: Promises

在第二章中，我们定位了在使用回调表达程序异步性和管理并发的两个主要类别的不足：缺乏顺序性和缺乏可靠性。现在我们更亲近地理解了问题，是时候将我们的注意力转向解决它们的模式了。

我们首先想要解决的是 控制倒转 问题，信任是如此脆弱而且是如此的容易丢失。

回想一下，我们将我们的程序的延续包装进一个回调函数中，将这个回调交给另一个团体（甚至是潜在的外部代码），并双手合十祈祷它会做正确的事情并调用这个回调。

我们这么做是因为我们想说，“这是 稍后 将要发生的事，在当前的步骤完成之后。”

但是如果 我们能够反向倒转这种 控制倒转 呢？如果不是将我们程序的延续交给另一个团体，而是希望它返回给我们一个可以知道它何时完成的能力，然后我们的代码可以决定下一步做什么呢？

这种规范被称为 **Promise**。

Promise正在像风暴一样席卷JS世界，因为开发者和语言规范作者之流拼命地想要在他们的代码/设计中结束回调地狱的疯狂。事实上，大多数新被加入JS/DOM平台的异步API都是建立在Promise之上的。所以深入学习它们可能是个好主意，你不这么认为吗？

注意：“立即”这个词将在本章频繁使用，一般来说它指代一些Promise解析行为。然而，本质上在所有情况下，“立即”意味着就工作队列行为（参见第一章）而言，不是严格同步的现在的感觉。

## 什么是Promise？

当开发者们决定要学习一种新技术或模式的时候，他们的第一步总是“给我看代码！”。摸着石头过河对我们来讲是十分自然的。

但事实上仅仅考察API丢失了一些抽象过程。Promise是这样一种工具：它能非常明显地看出使用者是否理解了它是什么和关于什么，还是仅仅学习和使用API。

所以在我展示Promise的代码之前，我想在概念上完整地解释一下Promise到底是什么。我希望这能更好地指引你探索如何将Promise理论整合到你自己的异步流程中。

带着这样的想法，让我们来看两种类比，来解释Promise是什么。

## 未来的值

想象这样的场景：我走到快餐店的柜台前，点了一个起士汉堡。并交了1.47美元的现金。通过点餐和付款，我为得到一个值（起士汉堡）制造了一个请求。我发起了一个事务。

但是通常来说，起士汉堡不会立即到我手中。收银员交给一些东西代替我的起士汉堡：一个带有点餐排队号的收据。这个点餐号是一个“我欠你”的许诺（Promise），它保证我最终会得到我的起士汉堡。

于是我就拿着我的收据和点餐号。我知道它代表我的未来的起士汉堡，所以我无需再担心它——除了挨饿！

在我等待的时候，我可以做其他的事情，比如给我的朋友发微信说，“嘿，一块儿吃午餐吗？我要吃起士汉堡”。

我已经在用我的未来的起士汉堡进行推理了，即便它还没有到我手中。我的大脑可以这么做是因为它将点餐号作为起士汉堡的占位符号。这个占位符号实质上使这个值与时间无关。它是一个未来的值。

最终，我听到，“113号！”。于是我愉快地拿着收据走向柜台前。我把收据递给收银员，拿回我的起士汉堡。

换句话说，一旦我的未来的值准备好，我就用我的许诺值换回值本身。

但还有另外一种可能的输出。它们叫我的号，但当我去取起士汉堡时，收银员遗憾地告诉我，“对不起，看起来我们的起士汉堡卖光了。”把这种场景下顾客有多沮丧放在一边，我们可以看到未来的值的一个重要性质：它们既可以表示成功也可以表示失败。

每次我点起士汉堡时，我都知道我要么最终得到一个起士汉堡，要么得到起士汉堡卖光的坏消息，并且不得不考虑中午吃点儿别的东西。

注意：在代码中，事情没有这么简单，因为还隐含着一种点餐号永远也不会被叫到的情况，这时我们就被搁置在了一种无限等待的未解析状态。我们待会儿再回头处理这种情况。

## 现在和稍后的值

这一切也许听起来在思维上太过抽象而不能实施在你的代码中。那么，让我们更具体一些。

然而，在我们能介绍Promise是如何以这种方式工作之前，我们先看看我们已经明白的代码——回调！——是如何处理这些未来值的。

在你写代码来推导一个值时，比如在一个 `number` 上进行数学操作，不论你是否理解，对于这个值你已经假设了某些非常基础的事实——这个值已经是一个实在的现在值：

```
var x, y = 2;

console.log( x + y ); // NaN <-- 因为`x`还没有被赋值
```

`x + y` 操作假定 `x` 和 `y` 都已经被设定好了。用我们一会将要阐述的术语来讲，我们假定 `x` 和 `y` 的值已经被解析 (`resovle`) 了。

期盼 + 操作符本身能够魔法般地检测并等待 `x` 和 `y` 的值被解析（也就是准备好），然后仅在那之后才进行操作是没道理的。如果不同的语句现在完成而其他的稍后完成，这就会在程序中造成混乱，对吧？

如果两个语句中的一个（或两者同时）可能还没有完成，你如何才能推断它们的关系呢？如果语句2要依赖语句1的完成，那么这里仅有两种输出：不是语句1现在立即完成而且一切处理正常进行，就是语句1还没有完成，所以语句2将会失败。

如果这些东西听起来很像第一章的内容，很好！

回到我们的 `x + y` 的数学操作。想象有一种方法可以说，“将 `x` 和 `y` 相加，但如果它们中任意一个还没有被设置，就等到它们都被设置。尽快将它们相加。”

你的大脑也许刚刚跳进回调。好吧，那么...

```
function add(getX, getY, cb) {
  var x, y;
  getX( function(xVal){
    x = xVal;
    // 两者都准备好了？
    if (y != undefined) {
      cb( x + y ); // 发送加法的结果
    }
  });
  getY( function(yVal){
    y = yVal;
    // 两者都准备好了？
    if (x != undefined) {
      cb( x + y ); // 发送加法的结果
    }
  });
}

// `fetchX()` 和 `fetchY()` 是同步或异步的函数
add( fetchX, fetchY, function(sum){
  console.log( sum ); // 很简单吧？
} );
```

花点儿时间来感受一下这段代码的美妙（或者丑陋），我耐心地等你。

虽然丑陋是无法否认的，但是关于这种异步模式有一些非常重要的事情需要注意。

在这段代码中，我们将 `x` 和 `y` 作为未来的值对待，我们将 `add(..)` 操作表达为：（从外部看来）它并不关心 `x` 或 `y` 或它们两者现在是否可用。换句话所，它泛化了现在和稍后，如此我们可以信赖 `add(..)` 操作的一个可预测的结果。

通过使用一个临时一致的 `add(..)` —— 它跨越现在和稍后的行为是相同的——异步代码的推理变得容易的多了。

更直白地说：为了一致地处理现在和稍后，我们将它们都作为稍后：所有的操作都变成异步的。

当然，这种粗略的基于回调的方法留下了许多提升的空间。为了理解在不用关心未来的值在时间上什么时候变得可用的情况下推理它而带来的好处，这仅仅是迈出的一小步。

## Promise值

我们绝对会在本章的后面深入更多关于Promise的细节——所以如果这让你犯糊涂，不要担心——但让我们先简单地看一下我们如何通过 `Promise` 来表达 `x + y` 的例子：

```
function add(xPromise, yPromise) {
    // `Promise.all([ .. ])` 接收一个Promise的数组，
    // 并返回一个等待它们全部完成的新Promise
    return Promise.all( [xPromise, yPromise] )

    // 当这个Promise被解析后，我们拿起收到的`X`和`Y`的值，并把它们相加
    .then( function(values){
        // `values`是一个从先前被解析的Promise那里收到的消息数组
        return values[0] + values[1];
    } );
}

// `fetchX()` 和 `fetchY()` 分别为它们的值返回一个Promise，
// 这些值可能在 *现在* 或 *稍后* 准备好
add( fetchX(), fetchY() )

// 为了将两个数字相加，我们得到一个Promise。
// 现在我们链式地调用`then(..)`来等待返回的Promise被解析
.then( function(sum){
    console.log( sum ); // 这容易多了！
} );
```

在这个代码段中有两层Promise。

`fetchX()` 和 `fetchY()` 被直接调用，它们的返回值 (`promise!`) 被传入 `add(..)`。这些 `promise` 表示的值将在现在或稍后准备好，但是每个 `promise` 都将行为泛化为与时间无关。我们以一种时间无关的方式来推理 `x` 和 `y` 的值。它们是未来值。

第二层是由 `add(..)` 创建（通过 `Promise.all([ .. ])`）并返回的promise，我们通过调用 `then(..)` 来等待它。当 `add(..)` 操作完成后，我们的 `sum` 未来值就准备好并可以打印了。我们将等待 `x` 和 `y` 的未来值的逻辑隐藏在 `add(..)` 内部。

注意：在 `add(..)` 内部。`Promise.all([ .. ])` 调用创建了一个promise（它在等待 `promiseX` 和 `promiseY` 被解析）。链式调用 `.then(..)` 创建了另一个promise，它的 `return values[0] + values[1]` 这一行会被立即解析（使用加法的结果）。这样，我们链接在 `add(..)` 调用末尾的 `then(..)` 调用——在代码段最后——实际上是在第二个被返回的 promise 上进行操作，而非被 `Promise.all([ .. ])` 创建的第一个 promise。另外，虽然我们没有在这第二个 `then(..)` 的末尾链接任何操作，它也已经创建了另一个 promise，我们可以选择监听/使用它。这类Promise链的细节将会在本章后面进行讲解。

就像点一个起士汉堡，Promise的解析可能是一个拒绝（rejection）而非完成（fulfillment）。不同的是，被完成的Promise的值总是程序化的，而一个拒绝值——通常被称为“拒绝理由”——既可以被程序逻辑设置，也可以被运行时异常隐含地设置。

使用Promise，`then(..)` 调用实际上可以接受两个函数，第一个用作完成（正如刚才所示），而第二个用作拒绝：

```
add( fetchX(), fetchY() )
  .then(
    // 完成处理器
    function(sum) {
      console.log( sum );
    },
    // 拒绝处理器
    function(err) {
      console.error( err ); // 倒霉！
    }
  );

```

如果在取得 `x` 或 `y` 时出现了错误，或在加法操作时某些事情不知怎地失败了，`add(..)` 返回的promise就被拒绝了，传入 `then(..)` 的第二个错误处理回调函数会从promise那里收到拒绝的值。

因为Promise包装了时间相关的状态——等待当前值的完成或拒绝——从外部看来，Promise本身是时间无关的，如此Promise就可以用可预测的方式组合，而不用关心时间或底层的结果。

另外，一旦Promise被解析，它就永远保持那个状态——它在那个时刻变成了一个不可变的值——而且可以根据需要被监听任意多次。

注意：因为Promise一旦被解析就是外部不可变的，所以现在将这个值传递给任何其他团体都是安全的，而且我们知道它不会被意外或恶意地被修改。这在许多团体监听同一个Promise的解析时特别有用。一个团体去影响另一个团体对Promise解析的监听能力是不可能的。不可

变性听起来是一个学院派话题，但它实际上是Promise设计中最基础且最重要的方面之一，因此不能将它随意地跳过。

这是用于理解Promise的最强大且最重要的概念之一。通过大量的工作，你可以仅仅使用丑陋的回调组合来创建相同的效果，但这真的不是一个高效的策略，特别是你不得不一遍一遍地重复它。

Promise是一种用来包装与组合 未来值，并且可以很容易复用的机制。

## 完成事件

正如我们刚才看到的，一个独立的Promise作为一个 未来值 动作。但还有另外一种方式考虑Promise的解析：在一个异步任务的两个或以上步骤中，作为一种流程控制机制——俗称“这个然后那个”。

让我们想象调用 `foo(..)` 来执行某个任务。我们对它的细节一无所知，我们也不关心。它可能会立即完成任务，也可能会花一段时间完成。

我们仅仅想简单地知道 `foo(..)` 什么时候完成，以便于我们可以移动到下一个任务。换句话说，我们想要一种方法被告知 `foo(..)` 的完成，以便于我们可以继续。

在典型的JavaScript风格中，如果你需要监听一个通知，你很可能会想到事件 (`event`)。那么我们可以将我们的通知需求重新表述为，监听由 `foo(..)` 发出的 完成 (或 继续) 事件。

注意：将它称为一个“完成事件”还是一个“继续事件”取决于你的角度。你是更关心 `foo(..)` 发生的事情，还是更关心 `foo(..)` 完成之后发生的事情？两种角度都对而且都有用。事件通知告诉我们 `foo(..)` 已经完成，但是继续到下一个步骤也没问题。的确，你为了事件通知调用而传入的回调函数本身，在前面我们称它为一个 延续。因为 完成事件 更加聚焦于 `foo(..)`，也就是我们当前注意的东西，所以在这篇文章的其余部分我们稍稍偏向于使用 完成事件。

使用回调，“通知”就是被任务 (`foo(..)`) 调用的我们的回调函数。但是使用Promise，我们将关系扭转过来，我们希望能够监听一个来自于 `foo(..)` 的事件，当我们被通知时，做相应的处理。

首先，考虑一些假想代码：

```

foo(x) {
    // 开始做一些可能会花一段时间的事情
}

foo( 42 )

on (foo "completion") {
    // 现在我们可以做下一步了！
}

on (foo "error") {
    // 噢，在`foo(..)`中有某些事情搞错了
}

```

我们调用 `foo(..)` 然后我们设置两个事件监听器，一个给 `"completion"`，一个给 `"error"` —— `foo(..)` 调用的两种可能的最终结果。实质上，`foo(..)` 甚至不知道调用它的代码监听了这些事件，这构成了一个非常美妙的 **关注分离 (separation of concerns)**。

不幸的是，这样的代码将需要JS环境不具备的一些“魔法”（而且显得有些不切实际）。这里是一种用JS表达它的更自然的方式：

```

function foo(x) {
    // 开始做一些可能会花一段时间的事情

    // 制造一个`listener`事件通知能力并返回

    return listener;
}

var evt = foo( 42 );

evt.on( "completion", function(){
    // 现在我们可以做下一步了！
} );

evt.on( "failure", function(err){
    // 噢，在`foo(..)`中有某些事情搞错了
} );

```

`foo(..)` 明确地创建并返回了一个事件监听能力，调用方代码接收并在它上面注册了两个事件监听器。

很明显这反转了一般的面向回调代码，而且是有意为之。与将回调传入 `foo(..)` 相反，它返回一个我们称之为 `evt` 的事件能力，它接回调。

但如果你回想第二章，回调本身代表着一种 控制反转。所以反转回调模式实际上是 反转的反转，或者说是一个 控制非反转——将控制权归还给我们希望保持它的调用方代码，

一个重要的好处是，代码的多个分离部分都可以被赋予事件监听能力，而且它们都可在 `foo(..)` 完成时被独立地通知，来执行后续的步骤：

```
var evt = foo( 42 );

// 让`bar(..)`监听`foo(..)`的完成
bar( evt );

// 同时，让`baz(..)`监听`foo(..)`的完成
baz( evt );
```

控制非反转 导致了更好的 关注分离，也就是 `bar(..)` 和 `baz(..)` 不必卷入 `foo(..)` 是如何被调用的问题。相似地，`foo(..)` 也不必知道或关心 `bar(..)` 和 `baz(..)` 的存在或它们是否在等待 `foo(..)` 完成的通知。

实质上，这个 `evt` 对象是一个中立的第三方团体，在分离的关注点之间进行交涉。

## Promise“事件”

正如你可能已经猜到的，`evt` 事件监听能力是一个Promise的类比。

在一个基于Promise的方式中，前面的代码段将会使 `foo(..)` 创建并返回一个 `promise` 实例，而且这个`promise`将会被传入 `bar(..)` 和 `baz(..)`。

注意：我们监听的Promise解析“事件”并不是严格的事情（虽然它们为了某些目的表现得像事件），而且它们也不经常称为 "completion" 或 "error"。相反，我们用 `then(..)` 来注册一个 "then" 事件。或者也许更准确地讲，`then(..)` 注册了 "fulfillment (完成)" 和/或 "rejection (拒绝)" 事件，虽然我们在代码中不会看到这些名词被明确地使用。

考虑：

```
function foo(x) {
    // 开始做一些可能会花一段时间的事情

    // 构建并返回一个promise
    return new Promise( function(resolve,reject){
        // 最终需要调用`resolve(..)`或`reject(..)`
        // 它们是这个promise的解析回调
    } );
}

var p = foo( 42 );

bar( p );
baz( p );
```

注意：在 `new Promise( function(..){ .. } )` 中展示的模式通常被称为“**揭示构造器 (revealing constructor)**”。被传入的函数被立即执行（不会被异步推迟，像 `then(..)` 的回调那样），而且它被提供了两个参数，我们叫它们 `resolve` 和 `reject`。这些是Promise的解析函数。`resolve(..)` 一般表示完成，而 `reject(..)` 表示拒绝。

你可能猜到了 `bar(..)` 和 `baz(..)` 的内部看起来是什么样子：

```
function bar(fooPromise) {
  // 监听`foo(..)`的完成
  fooPromise.then(
    function(){
      // `foo(..)`现在完成了，那么做`bar(..)`的任务
    },
    function(){
      // 噢，在`foo(..)`中有某些事情搞错了
    }
  );
}

// `baz(..)`同上
```

Promise解析没有必要一定发送消息，就像我们将Promise作为未来值考察时那样。它可以仅仅作为一种流程控制信号，就像前面的代码中那样使用。

另一种表达方式是：

```
function bar() {
  // `foo(..)`绝对已经完成了，那么做`bar(..)`的任务
}

function oopsBar() {
  // 噢，在`foo(..)`中有某些事情搞错了，那么`bar(..)`不会运行
}

// `baz()`和`oopsBaz()`同上

var p = foo( 42 );

p.then( bar, oopsBar );

p.then( baz, oopsBaz );
```

注意：如果你以前见过基于Promise的代码，你可能会相信这段代码的最后两行应当写做 `p.then( .. ).then( .. )`，使用链接，而不是 `p.then(..); p.then(..)`。这将会是两种完全不同的行为，所以要小心！这种区别现在看起来可能不明显，但是它们实际上是我们目前还没有见过的异步模式：分割 (splitting) / 分叉 (forking)。不必担心！本章后面我们会回到这个话题。

与将 `p promise` 传入 `bar(..)` 和 `baz(..)` 相反，我们使用 `promise` 来控制 `bar(..)` 和 `baz(..)` 何时该运行，如果有这样的时刻。主要区别在于错误处理。

在第一个代码段的方式中，无论 `foo(..)` 是否成功 `bar(..)` 都会被调用，如果被通知 `foo(..)` 失败了的话它提供自己的后备逻辑。显然，`baz(..)` 也是这样做的。

在第二个代码段中，`bar(..)` 仅在 `foo(..)` 成功后才被调用，否则 `oopsBar(..)` 会被调用。`baz(..)` 也是。

两种方式本身都对。但会有一些情况使一种优于另一种。

在这两种方式中，从 `foo(..)` 返回的 `promise p` 都被用于控制下一步发生什么。

另外，两个代码段都以对同一个 `promise p` 调用两次 `then(..)` 结束，这展示了先前的观点，也就是 `Promise` (一旦被解析) 会永远保持相同的解析结果 (完成或拒绝)，而且可以按需要后续地被监听任意多次。

无论何时 `p` 被解析，下一步都将总是相同的，包括现在和稍后。

## Thenable 鸭子类型 (Duck Typing)

在 `Promise` 的世界中，一个重要的细节是如何确定一个值是否是纯粹的 `Promise`。或者更直接地说，一个值会不会像 `Promise` 那样动作？

我们知道 `Promise` 是由 `new Promise(..)` 语法构建的，你可能会想 `p instanceof Promise` 将是一个可以接受的检查。但不幸的是，有几个理由表明它不是完全够用。

主要原因是，你可以从其他浏览器窗口中收到 `Promise` 值 (`iframe` 等)，其他的浏览器窗口会拥有自己的不同于当前窗口/`frame` 的 `Promise`，这种检查将会在定位 `Promise` 实例时失效。

另外，一个库或框架可能会选择实现自己的 `Promise` 而不是用 ES6 原生的 `Promise` 实现。事实上，你很可能在根本没有 `Promise` 的老版本浏览器中通过一个库来使用 `Promise`。

当我们在本章稍后讨论 `Promise` 的解析过程时，为什么识别并同化一个非纯种但相似 `Promise` 的值仍然很重要会愈发明显。但目前只需要相信我，它是拼图中很重要的一块。

如此，人们决定识别一个 `Promise` (或像 `Promise` 一样动作的某些东西) 的方法是定义一种称为“`thenable`”的东西，也就是任何拥有 `then(..)` 方法的对象或函数。这种方法假定任何这样的值都是一个符合 `Promise` 的 `thenable`。

根据值的形状 (存在什么属性) 来推测它的“类型”的“类型检查”有一个一般的名称，称为“鸭子类型检查”——“如果它看起来像一只鸭子，并且叫起来相一致鸭子，那么它一定是一只鸭子”(参见本丛书的 `类型与文法`)。所以对 `thenable` 的鸭子类型检查可能大致是这样：

```

if (
  p !== null &&
  (
    typeof p === "object" ||
    typeof p === "function"
  ) &&
  typeof p.then === "function"
) {
  // 认为它是一个thenable!
}
else {
  // 不是一个thenable
}

```

晕！先把将这种逻辑在各种地方实现有点丑陋的事实放在一边不谈，这里还有更多更深层的麻烦。

如果你试着用一个偶然拥有 `then(..)` 函数的任意对象/函数来完成一个Promise，但你又没想把它当做一个Promise/thenable来对待，你的运气就用光了，因为它会被自动地识别为一个thenable并以特殊的规则来对待（见本章后面的部分）。

如果你不知道一个值上面拥有 `then(..)` 就更是这样。比如：

```

var o = { then: function(){ } };

// 使`v`用`[[Prototype]]`链接到`o`
var v = Object.create( o );

v.someStuff = "cool";
v.otherStuff = "not so cool";

v.hasOwnProperty( "then" );           // false

```

`v` 看起来根本不是一个Promise或thanable。它只是一个拥有一些属性的直白的对象。你可能只是想要把这个值像其他对象那样传递而已。

但你不知道的是，`v` 还 `[[Prototype]]` 连接着（见本丛书的 `this`与对象原型）另一个对象 `o`，在它上面偶然拥有一个 `then(..)`。所以thenable鸭子类型检查将会认为并假定 `v` 是一个thenable。噢。

它甚至不需要直接故意那么做：

```

Object.prototype.then = function(){};
Array.prototype.then = function(){};

var v1 = { hello: "world" };
var v2 = [ "Hello", "World" ];

```

v1 和 v2 都将被假定为是thenalbe的。你不能控制或预测是否有其他代码偶然或恶意地将 then(..) 加到 Object.prototype , Array.prototype , 或其他任何原生原型上。而且如果这个指定的函数并不将它的任何参数作为回调调用，那么任何用这样的值被解析的Promise都将无声地永远挂起！疯狂。

听起来难以置信或不太可能？也许。

要知道，在ES6之前就有几种广为人知的非Promise库在社区中存在了，而且它们已经偶然拥有了称为 then(..) 的方法。这些库中的一些选择了重命名它们自己的方法来回避冲突（这很烂！）。另一些则因为它们无法改变来回避冲突，简单地降级为“不兼容基于Promise的代码”的不幸状态。

用来劫持原先非保留的——而且听起来完全是通用的—— then 属性名称的标准决议是，没有值（或它的任何委托），无论是过去，现在，还是将来，可以拥有 then(..) 函数，不管是有意的还是偶然的，否则这个值将在Promise系统中被混淆为一个thenable，从而可能产生非常难以追踪的Bug。

警告：我不喜欢我们用thenable的鸭子类型来结束对Promise认知的方式。还有其他的选项，比如“branding”或者甚至是“anti-branding”；我们得到的似乎是一个最差劲儿的妥协。但它并不全是悲观与失望。thenable鸭子类型可以很有用，就像我们马上要看到的。只是要小心，如果thenable鸭子类型将不是Promise的东西误认为是Promise，它就可能成为灾难。

## Promise的信任

我们已经看过了两个强烈的类比，它们解释了Promise可以为我们的异步代码所做的事的不同方面。但如果我们停在这里，我们就可能会错过一个Promise模式建立的最重要的性质：信任。

随着 未来值 和 完成事件 的类别在我们探索的代码模式中的明确展开，有一个问题依然没有完全明确：Promise是为什么，以及如何被设计为来解决所有我们在第二章“信任问题”一节中提出的 控制倒转 的信任问题的。但是只要深挖一点儿，我们就可以发现一些重要的保证，来重建第二章中毁掉的对异步代码的信心！

让我们从复习仅使用回调的代码中的信任问题开始。当你传递一个回调给一个工具 foo(..) 的时候，它可能：

- 调用回调太早
- 调用回调太晚（或根本不调）
- 调用回调太少或太多次
- 没能传递必要的环境/参数
- 吞掉了任何可能发生的错误/异常

Promise的性质被有意地设计为给这些顾虑提供有用的答案。

## 调的太早

这种顾虑主要是代码是否会引入类Zalgo效应，也就是一个任务有时会同步完地成，而有时会异步地完成，这将导致竞合状态。

Promise被定义为不能受这种顾虑的影响，因为即便是立即完成的Promise（比如 `new Promise(function(resolve){ resolve(42); })`）也不可能被同步地监听。

也就是说，但你在Promise上调用 `then(..)` 的时候，即便这个Promise已经被解析了，你给 `then(..)` 提供的回调也将总是被异步地调用（更多关于这里的内客，参照第一章的"Jobs"）。

不必再插入你自己的 `setTimeout(..,0)` 黑科技了。Promise自动地防止了Zalgo效应。

## 调的太晚

和前一点相似，在 `resolve(..)` 或 `reject(..)` 被Promise创建机制调用时，一个Promise的 `then(..)` 上注册的监听回调将自动地被排程。这些被排程好的回调将在下一个异步时刻被可预测地触发（参照第一章的"Jobs"）。

同步监听是不可能的，所以不可能有一个同步的任务链的运行来“推迟”另一个回调的发生。也就是说，当一个Promise被解析时，所有在 `then(..)` 上注册的回调都将被立即，按顺序地，在下一个异步机会时被调用（再一次，参照第一章的"Jobs"），而且没有任何在这些回调中发生的事情可以影响/推迟其他回调的调用。

举例来说：

```
p.then( function(){
  p.then( function(){
    console.log( "C" );
  });
  console.log( "A" );
});
p.then( function(){
  console.log( "B" );
});
// A B C
```

这里，有赖于Promise如何定义操作，"C" 不可能干扰并优先于 "B"。

## Promise排程的怪现象

重要并需要注意的是，排程有许多微妙的地方：链接在两个分离的Promise上的回调之间的相对顺序，是不能可靠预测的。

如果两个promise `p1` 和 `p2` 都准备好被解析了，那么 `p1.then(..); p2.then(..)` 应当归结为首先调用 `p1` 的回调，然后调用 `p2` 的。但有一些微妙的情形可能会使这不成立，比如下面这样：

```

var p3 = new Promise( function(resolve,reject){
  resolve( "B" );
} );

var p1 = new Promise( function(resolve,reject){
  resolve( p3 );
} );

var p2 = new Promise( function(resolve,reject){
  resolve( "A" );
} );

p1.then( function(v){
  console.log( v );
} );

p2.then( function(v){
  console.log( v );
} );

// A B  <-- 不是你可能期望的 B A

```

我们稍后会更多地讲解这个问题，但如你所见，`p1` 不是被一个立即值所解析的，而是由另一个promise `p3` 所解析，而 `p3` 本身被一个值 "B" 所解析。这种指定的行为将 `p3` 展开到 `p1`，但是是异步地，所以在异步工作队列中 `p1` 的回调位于 `p2` 的回调之后（参照第一章的"Jobs"）。

为了回避这样的微妙的噩梦，你绝不应该依靠任何跨Promise的回调顺序/排程。事实上，一个好的实践方式是在代码中根本不要让多个回调的顺序成为问题。尽可能回避它。

## 根本不调回调

这是一个很常见的顾虑。Promise用几种方式解决它。

首先，没有任何东西（JS错误都不能）可以阻止一个Promise通知你它的解析（如果它被解析了的话）。如果你在一个Promise上同时注册了完成和拒绝回调，而且这个Promise被解析了，两个回调中的一个总会被调用。

当然，如果你的回调本身有JS错误，你可能不会看到你期望的结果，但是回调事实上已经被调用了。我们待会儿就会讲到如何在你的回调中收到关于一个错误的通知，因为就算是它们也不会被吞掉。

那如果Promise本身不管怎样永远没有被解析呢？即便是这种状态Promise也给出了答案，使用一个称为“竞赛（race）”的高级抽象。

```
// 一个使Promise超时的工具
function timeoutPromise(delay) {
  return new Promise( function(resolve,reject){
    setTimeout( function(){
      reject( "Timeout!" );
    }, delay );
  } );
}

// 为`foo()`设置一个超时
Promise.race( [
  foo(), // 尝试调用`foo()`
  timeoutPromise( 3000 ) // 给它3秒钟
] )
.then(
  function(){
    // `foo(..)`及时地完成了！
  },
  function(err){
    // `foo()`不是被拒绝了，就是它没有及时完成
    // 那么可以考察`err`来知道是哪种情况
  }
);

```

这种Promise的超时模式有更多的细节需要考虑，但我们待会儿再回头讨论。

重要的是，我们可以确保一个信号作为 `foo(..)` 的结果，来防止它无限地挂起我们的程序。

## 调太少或太多次

根据定义，对于被调用的回调来讲一次是一个合适的次数。“太少”的情况将会是0次，和我们刚刚考察的从不调用是相同的。

“太多”的情况则很容易解释。Promise被定义为只能被解析一次。如果因为某些原因，Promise的创建代码试着调用 `resolve(..)` 或 `reject(..)` 许多次，或者试着同时调用它们俩，Promise将仅接受第一次解析，而无声地忽略后续的尝试。

因为一个Promise仅能被解析一次，所以任何 `then(..)` 上注册的（每个）回调将仅仅被调用一次。

当然，如果你把同一个回调注册多次（比如 `p.then(f); p.then(f);`），那么它就会被调用注册的那么多次。响应函数仅被调用一次的保证并不能防止你砸自己的脚。

## 没能传入任何参数/环境

Promise可以拥有最多一个解析值（完成或拒绝）。

如果无论怎样你没有用一个值明确地解析它，它的值就是 `undefined`，就像JS中常见的那样。但不管是什值，它总是会被传入所有被注册的（并且适当地：完成或拒绝）回调中，不管是现在还是将来。

需要意识到的是：如果你使用多个参数调用 `resolve(..)` 或 `reject(..)`，所有第一个参数之外的后续参数都会被无声地忽略。虽然这看起来违反了我们刚才描述的保证，但并不确切，因为它构成了一种Promise机制的无效使用方式。其他的API无效使用方式（比如调用 `resolve(..)` 许多次）也都相似地被保护，所以Promise的行为在这里是一致的（除了有一点点让人沮丧）。

如果你想传递多个值，你必须将它们包装在另一个单独的值中，比如一个 `array` 或一个 `object`。

至于环境，JS中的函数总是保持他们被定义时所在作用域的闭包（见本系列的作用域与闭包），所以它们理所当然地可以继续访问你提供的环境状态。当然，这对仅使用回调的设计来讲也是对的，所以这不能算是Promise带来的增益——但尽管如此，它依然是我们可以依赖的保证。

## 吞掉所有错误/异常

在基本的感觉上，这是前一点的重述。如果你用一个理由（也就是错误消息）拒绝一个Promise，这个值就会被传入拒绝回调。

但是这里有一个更重要的事情。如果在Promise的创建过程中的任意一点，或者在监听它的解析的过程中，一个JS异常错误发生的话，比如 `TypeError` 或 `ReferenceError`，这个异常将会被捕获，并且强制当前的Promise变为拒绝。

举例来说：

```
var p = new Promise( function(resolve,reject){
    foo.bar();      // `foo`没有定义，所以这是一个错误！
    resolve( 42 );  // 永远不会跑到这里 :(
} );

p.then(
    function fulfilled(){
        // 永远不会跑到这里 :(
    },
    function rejected(err){
        // `err`将是一个来自`foo.bar()`那一行的`TypeError`异常对象
    }
);
```

在 `foo.bar()` 上发生的JS异常变成了一个你可以捕获并响应的Promise拒绝。

这是一个重要的细节，因为它有效地解决了另一种潜在的Zalgo时刻，也就是错误可能会产生一个同步的反应，而没有错误的部分还是异步的。Promise甚至将JS异常都转化为异步行为，因此极大地降低了发生竞合状态的可能性。

但是如果Promise完成了，但是在监听过程中（在一个 `then(..)` 上注册的回调上）出现了JS异常错误会怎样呢？即便是那些也不会丢失，但你可能会发现处理它们的方式有些令人诧异，除非你深挖一些：

```
var p = new Promise( function(resolve,reject){
    resolve( 42 );
} );

p.then(
    function fulfilled(msg){
        foo.bar();
        console.log( msg );      // 永远不会跑到这里 :(
    },
    function rejected(err){
        // 也永远不会跑到这里 :(
    }
);
```

等一下，这看起来 `foo.bar()` 发生的异常确实被吞掉了。不要害怕，它没有。但更深层次的东西出问题了，也就是我们没能成功地监听他。`p.then(..)` 调用本身返回另一个promise，是那个 promise 将会被 `TypeError` 异常拒绝。

为什么它不能调用我们在这里定义的错误处理器呢？表面上看起来是一个符合逻辑的行为。但它会违反Promise一旦被解析就不可变的基本原则。`p` 已经完成为值 `42`，所以它不能因为在监听 `p` 的解析时发生了错误，而在稍后变成一个拒绝。

除了违反原则，这样的行为还可能造成破坏，假如说有多个在promise `p` 上注册的 `then(..)` 回调，因为有些会被调用而有些不会，而且至于为什么是很明显的。

## 可信的Promise？

为了基于Promise模式建立信任，还有最后一个细节需要考察。

无疑你已经注意到了，Promise根本没有摆脱回调。它们只是改变了回调传递的位置。与将一个回调传入 `foo(..)` 相反，我们从 `foo(..)` 那里拿回某些东西（表面上是一个纯粹的Promise），然后我们将回调传入这个东西。

但为什么这要比仅使用回调的方式更可靠呢？我们如何确信我们拿回来的某些东西事实上是一个可信的Promise？这难道不是说我们相信它仅仅因为我们已经相信它了吗？

一个Promise经常被忽视，但是最重要的细节之一，就是它也为这个问题给出了解决方案。包含在原生的ES6 Promise实现中，它就是 `Promise.resolve(..)`。

如果你传递一个立即的，非Promise的，非thenable的值给 `Promise.resolve(..)`，你会得到一个用这个值完成的promise。换句话说，下面两个promise `p1` 和 `p2` 的行为基本上完全相同：

```
var p1 = new Promise( function(resolve,reject){  
    resolve( 42 );  
} );  
  
var p2 = Promise.resolve( 42 );
```

但如果你传递一个纯粹的Promise给 `Promise.resolve(..)`，你会得到这个完全相同的promise：

```
var p1 = Promise.resolve( 42 );  
  
var p2 = Promise.resolve( p1 );  
  
p1 === p2; // true
```

更重要的是，如果你传递一个非Promise的thenable值给 `Promise.resolve(..)`，它会试着将这个值展开，而且直到抽出一个最终具体的非Promise值之前，展开操作将会一直继续下去。

还记得我们先前讨论的thenable吗？

考虑这段代码：

```
var p = {  
  then: function(cb) {  
    cb( 42 );  
  }  
};  
  
// 这工作起来没问题，但要靠运气  
p  
.then(  
  function fulfilled(val){  
    console.log( val ); // 42  
  },  
  function rejected(err){  
    // 永远不会跑到这里  
  }  
);
```

这个 `p` 是一个thenable，但它不是一个纯粹的Promise。很走运，它是合理的，正如大多数情况那样。但是如果你得到的是看起来像这样的东西：

```

var p = {
  then: function(cb,errcb) {
    cb( 42 );
    errcb( "evil laugh" );
  }
};

p
.then(
  function fulfilled(val){
    console.log( val ); // 42
  },
  function rejected(err){
    // 噢，这里本不该运行
    console.log( err ); // evil laugh
  }
);

```

这个 `p` 是一个`thenable`，但它不是表现良好的`promise`。它是恶意的吗？或者它只是不知道`Promise`应当如何工作？老实说，这不重要。不管哪种情况，它都不那么可靠。

尽管如此，我们可以将这两个版本的 `p` 传入 `Promise.resolve(..)`，而且我们将会得到一个我们期望的泛化，安全的结果：

```

Promise.resolve( p )
.then(
  function fulfilled(val){
    console.log( val ); // 42
  },
  function rejected(err){
    // 永远不会跑到这里
  }
);

```

`Promise.resolve(..)` 会接受任何`thenable`，而且将它展开直至非`thenable`值。但你会从 `Promise.resolve(..)` 那里得到一个真正的，纯粹的`Promise`，一个你可以信任的东西。如果你传入的东西已经是一个纯粹的`Promise`了，那么你会单纯地将它拿回来，所以通过 `Promise.resolve(..)` 过滤来得到信任没有任何坏处。

那么我们假定，我们在调用一个 `foo(..)` 工具，而且不能确定我们能相信它的返回值是一个行为规范的`Promise`，但我们知道它至少是一个`thenable`。`Promise.resolve(..)` 将会给我们一个可靠的`Promise`包装器来进行链式调用：

```
// 不要只是这么做：
foo( 42 )
.then( function(v){
  console.log( v );
} );

// 相反，这样做：
Promise.resolve( foo( 42 ) )
.then( function(v){
  console.log( v );
} );
```

注意：将任意函数的返回值（thenable或不是thenable）包装在 `Promise.resolve(..)` 中的另一个好的副作用是，它可以很容易地将函数调用泛化为一个行为规范的异步任务。如果 `foo(42)` 有时返回一个立即值，而其他时候返回一个 `Promise`，`Promise.resolve(foo(42))`，将确保它总是返回 `Promise`。并且使代码成为回避 Zalgo 效应的更好的代码。

## 信任建立了

希望前面的讨论使你现在完全理解了 `Promise` 是可靠的，而且更为重要的是，为什么信任对于建造强壮，可维护的软件来说是如此关键。

没有信任，你能用 JS 编写异步代码吗？你当然能。我们 JS 开发者在除了回调以外没有任何东西的情况下，写了将近 20 年的异步代码了。

但是一旦你开始质疑你到底能够以多大的程度相信你的底层机制，它实际上多么可预见，多么可靠，你就会开始理解回调的信任基础多么的摇摇欲坠。

`Promise` 是一个用可靠语义来增强回调的模式，所以它的行为更合理更可靠。通过将回调的控制倒转 反置过来，我们将控制交给一个可靠的系统（`Promise`），它是为了将你的异步处理进行清晰的表达而特意设计的。

## 链式流程

我们已经被暗示过几次，但 `Promise` 不仅是一个单步的 这个然后那个 操作机制。当然，那是构建块儿，但事实证明我们可以将多个 `Promise` 串联在一起表达一系列的异步步骤。

使这一切能够工作的关键，是 `Promise` 的两个固有行为：

- 每次你在一个 `Promise` 上调用 `then(..)` 的时候，它都创建并返回一个新的 `Promise`，我们可以在它上面进行链接。
- 无论你从 `then(..)` 调用的完成回调中（第一个参数）返回什么值，它都做为被链接的 `Promise` 的完成。

我们首先来说一下这是什么意思，然后我们将会延伸出它是如何帮助我们创建异步顺序的控制流程的。考虑下面的代码：

```
var p = Promise.resolve( 21 );

var p2 = p.then( function(v){
    console.log( v );      // 21

    // 使用值`42`完成`p2`
    return v * 2;
} );

// 在`p2`后链接
p2.then( function(v){
    console.log( v );      // 42
} );
```

通过返回 `v * 2` (也就是 `42`)，我们完成了由第一个 `then(..)` 调用创建并返回的 `p2 promise`。当 `p2` 的 `then(..)` 调用运行时，它从 `return v * 2` 语句那里收到完成信号。当然，`p2.then(..)` 还会创建另一个 `promise`，我们将它存储在变量 `p3` 中。

但是不得不创建临时变量 `p2` (或 `p3` 等) 有点儿恼人。幸运的是，我们可以简单地将这些链接在一起：

```
var p = Promise.resolve( 21 );

p
.then( function(v){
    console.log( v );      // 21

    // 使用值`42`完成被链接的promise
    return v * 2;
} )
// 这里是被链接的promise
.then( function(v){
    console.log( v );      // 42
} );
```

那么现在第一个 `then(..)` 是异步序列的第一步，而第二个 `then(..)` 就是第二步。它可以根据你的需要延伸至任意长。只要持续不断地用每个自动创建的 `Promise` 在前一个 `then(..)` 末尾进行连接即可。

但是这里错过了某些东西。要是我们想让第2步等待第1步去做一些异步的事情呢？我们使用的是一个立即的 `return` 语句，它立即完成了链接中的 `promise`。

使Promise序列在每一步上都是真正异步的关键，需要回忆一下当你向`Promise.resolve(..)`传递一个Promise或thenable而非一个最终值时它如何执行。`Promise.resolve(..)`会直接返回收到的纯粹Promise，或者它会展开收到的thenable的值——并且它会递归地持续展开thenable。

如果你从完成（或拒绝）处理器中返回一个thenable或Promise，同样的展开操作也会发生。考虑这段代码：

```
var p = Promise.resolve( 21 );

p.then( function(v){
  console.log( v );      // 21

  // 创建一个promise并返回它
  return new Promise( function(resolve,reject){
    // 使用值`42`完成
    resolve( v * 2 );
  } );
} )
.then( function(v){
  console.log( v );      // 42
} );
```

即便我们把`42`包装在一个我们返回的promise中，它依然会被展开并作为下一个被链接的promise的解析，如此第二个`then(..)`仍然收到`42`。如果我们在这个包装promise中引入异步，一切还是会同样正常的工作：

```
var p = Promise.resolve( 21 );

p.then( function(v){
  console.log( v );      // 21

  // 创建一个promise并返回
  return new Promise( function(resolve,reject){
    // 引入异步！
    setTimeout( function(){
      // 使用值`42`完成
      resolve( v * 2 );
    }, 100 );
  } );
} )
.then( function(v){
  // 在上一步中的100毫秒延迟之后运行
  console.log( v );      // 42
} );
```

这真是不可思议的强大！现在我们可以构建一个序列，它可以有我们想要的任意多的步骤，而且每一步都可以按照需要来推迟下一步（或者不推迟）。

当然，在这些例子中一步一步向下传递的值是可选的。如果你没有返回一个明确的值，那么它假定一个隐含的 `undefined`，而且promise依然会以同样的方式链接在一起。如此，每个Promise的解析只不过是进行至下一步的信号。

为了演示更长的链接，让我们把推迟Promise的创建（没有解析信息）泛化为一个我们可以在多个步骤中复用的工具：

```
function delay(time) {
  return new Promise( function(resolve,reject){
    setTimeout( resolve, time );
  } );
}

delay( 100 ) // step 1
.then( function STEP2(){
  console.log( "step 2 (after 100ms)" );
  return delay( 200 );
} )
.then( function STEP3(){
  console.log( "step 3 (after another 200ms)" );
} )
.then( function STEP4(){
  console.log( "step 4 (next Job)" );
  return delay( 50 );
} )
.then( function STEP5(){
  console.log( "step 5 (after another 50ms)" );
} )
...

```

调用 `delay(200)` 创建了一个将在200毫秒内完成的promise，然后我们在第一个 `then(..)` 的完成回调中返回它，这将使第二个 `then(..)` 的promise等待这个200毫秒的promise。

注意：正如刚才描述的，技术上讲在这个交替中有两个promise：一个200毫秒延迟的promise，和一个被第二个 `then(..)` 链接的promise。但你可能会发现将这两个promise组合在一起更容易思考，因为Promise机制帮你把它们的状态自动地混合到了一起。从这个角度讲，你可以认为 `return delay(200)` 创建了一个promise来取代早前一个返回的被链接的promise。

老实说，没有任何消息进行传递的一系列延迟作为Promise流程控制的例子不是很有用。让我们来看一个更加实在的场景：

与计时器不同，让我们考虑发起Ajax请求：

```
// 假定一个`ajax( {url}, {callback} )`工具

// 带有Promise的ajax
function request(url) {
    return new Promise( function(resolve,reject){
        // `ajax(..)`的回调应当是我们的promise的`resolve(..)`函数
        ajax( url, resolve );
    } );
}
```

我们首先定义一个 `request(..)` 工具，它构建一个promise表示 `ajax(..)` 调用的完成：

```
request( "http://some.url.1/" )
.then( function(response1){
    return request( "http://some.url.2/?v=" + response1 );
} )
.then( function(response2){
    console.log( response2 );
} );
```

注意：开发者们通常遭遇的一种情况是，他们想用本身不支持Promise的工具（就像这里的 `ajax(..)`，它期待一个回调）进行Promise式的异步流程控制。虽然ES6原生的 `Promise` 机制不会自动帮我们解决这种模式，但是在实践中所有的Promise库会帮我们这么做。它们通常称这种处理为“提升（lifting）”或“promise化”或其他的什么名词。我们稍后再回头讨论这种技术。

使用返回Promise的 `request(..)`，通过用第一个URL调用它我们在链条中隐式地创建了第一步，然后我们用第一个 `then(..)` 在返回的promise末尾进行连接。

一旦 `response1` 返回，我们用它的值来构建第二个URL，并且发起第二个 `request(..)` 调用。这第二个 promise 是 `return` 的，所以我们的异步流程控制的第三步将会等待这个Ajax调用完成。最终，一旦 `response2` 返回，我们就打印它。

我们构建的Promise链不仅是一个表达多步骤异步序列的流程控制，它还扮演者将消息从一步传递到下一步的消息管道。

要是Promise链中的某一步出错了会怎样呢？一个错误/异常是基于每个Promise的，意味着在链条的任意一点捕获这些错误是可能的，而且这些捕获操作在那一点上将链条“重置”，使它回到正常的操作上来：

```

// 步骤 1:
request( "http://some.url.1/" )

// 步骤 2:
.then( function(response1){
    foo.bar(); // 没有定义，错误！

    // 永远不会跑到这里
    return request( "http://some.url.2/?v=" + response1 );
} )

// 步骤 3:
.then(
    function fulfilled(response2){
        // 永远不会跑到这里
    },
    // 拒绝处理器捕捉错误
    function rejected(err){
        console.log( err ); // 来自 `foo.bar()` 的 `TypeError` 错误
        return 42;
    }
)

// 步骤 4:
.then( function(msg){
    console.log( msg ); // 42
} );

```

当错误在第2步中发生时，第3步的拒绝处理器将它捕获。拒绝处理器的返回值（在这个代码段里是 `42`），如果有的话，将会完成下一步（第4步）的promise，如此整个链条又回到完成的状态。

**注意：**就像我们刚才讨论过的，当我们从一个完成处理器中返回一个promise时，它会被展开并有可能推迟下一步。这对从拒绝处理器中返回的promise也是成立的，这样如果我们在第3步返回一个promise而不是 `return 42`，那么这个promise就可能会推迟第4步。不管是在 `then(..)` 的完成还是拒绝处理器中，一个被抛出的异常都将导致下一个（链接着的）promise立即用这个异常拒绝。

如果你在一个promise上调用 `then(..)`，而且你只向它传递了一个完成处理器，一个假定的拒绝处理器会取而代之：

```

var p = new Promise( function(resolve,reject){
    reject( "Oops" );
} );

var p2 = p.then(
    function fulfilled(){
        // 永远不会跑到这里
    }
    // 如果忽略或者传入任何非函数的值，
    // 会有假定有一个这样的拒绝处理器
    // function(err) {
    //     throw err;
    // }
);

);

```

如你所见，这个假定的拒绝处理器仅仅简单地重新抛出错误，它最终强制 `p2`（链接着的 `promise`）用同样的错误进行拒绝。实质上，它允许错误持续地在Promise链上传播，直到遇到一个明确定义的拒绝处理器。

注意：稍后我们会讲到更多关于使用Promise进行错误处理的细节，因为会有更多微妙的细节需要关心。

如果没有一个恰当的合法的函数作为 `then(..)` 的完成处理器参数，也会有一个默认的处理器取而代之：

```

var p = Promise.resolve( 42 );

p.then(
    // 如果忽略或者传入任何非函数的值，
    // 会有假定有一个这样的完成处理器
    // function(v) {
    //     return v;
    // }
    null,
    function rejected(err){
        // 永远不会跑到这里
    }
);

```

如你所见，默认的完成处理器简单地将它收到的任何值传递给下一步（Promise）。

注意：`then(null,function(err){ .. })` 这种模式——仅处理拒绝（如果发生的话）但让成功通过——有一个缩写的API：`catch(function(err){ .. })`。我们会在下一节中更全面地涵盖 `catch(..)`。

然我们简要地复习一下使链式流程控制成为可能的Promise固有行为：

- 在一个Promise上的 `then(..)` 调用会自动生成一个新的Promise并返回。

- 在完成/拒绝处理器内部，如果你返回一个值或抛出一个异常，新返回的Promise（可以被链接的）将会相应地被解析。
- 如果完成或拒绝处理器返回一个Promise，它会被展开，所以无论它被解析为什么值，这个值都将变成从当前的 `then(..)` 返回的被链接的Promise的解析。

虽然链式流程控制很有用，但是将它认为是Promise的组合方式的副作用可能最准确，而不是它的主要意图。正如我们已经详细讨论过许多次的，Promise泛化了异步处理并且包装了与时间相关的值和状态，这才是让我们以这种有用的方式将它们链接在一起的原因。

当然，相对于我们在第二章中看到的一堆混乱的回调，这种链条的顺序表达是一个巨大的改进。但是仍然要跨过相当多的模板代码（`then(..)` and `function(){ .. }`）。在下一章中，我们将看到一种极大美化顺序流程控制的表达模式，生成器（generators）。

## 术语: **Resolve** (解析) , **Fulfill** (完成) , 和 **Reject** (拒绝)

在你更多深入地学习Promise之前，在“解析（`resolve`）”，“完成（`fulfill`）”，和“拒绝（`reject`）”这些名词之间还有一些我们需要辨明的小困惑。首先让我们考虑一下 `Promise(..)` 构造器：

```
var p = new Promise( function(X,Y){
  // X() 给 fulfillment (完成)
  // Y() 给 rejection (拒绝)
});
```

如你所见，有两个回调（标识为 `X` 和 `Y`）被提供了。第一个通常用于表示Promise完成了，而第二个总是表示Promise拒绝了。但“通常”是什么意思？它对这些参数的正确命名暗示着什么呢？

最终，这只是你的用户代码，和将被引擎翻译为没有任何含义的东西的标识符，所以在技术上它无紧要；`foo(..)` 和 `bar(..)` 在功能性上是相等的。但是你用的词不仅会影响你如何考虑这段代码，还会影响你所在团队的其他开发者如何考虑它。将精心策划的异步代码错误地考虑，几乎可以说要比面条一般的回调还要差劲儿。

所以，某种意义上你如何称呼它们很关键。

第二个参数很容易决定。几乎所有的文献都使用 `reject(..)` 做为它的名称，应为这正是它（唯一！）要做的，对于命名来说这是一个很好的选择。我也强烈推荐你一直使用 `reject(..)`。

但是关于第一个参数还是有些带有歧义，它在许多关于Promise的文献中常被标识为 `resolve(..)`。这个词明显地是与“resolution（解析）”有关，它在所有的文献中（包括本书）广泛用于描述给Promise设定一个最终的值/状态。我们已经使用“解析Promise（`resolve the Promise`）”许多次来意味Promise的完成（`fulfilling`）或拒绝（`rejecting`）。

但是如果这个参数看起来被用于特指Promise的完成，为什么我们不更准确地叫它 `fulfill(..)`，而是用 `resolve(..)` 呢？要回答这个问题，让我们看一下 `Promise` 的两个 API方法：

```
var fulfilledPr = Promise.resolve( 42 );
var rejectedPr = Promise.reject( "Oops" );
```

`Promise.resolve(..)` 创建了一个Promise，它被解析为它被给予的值。在这个例子中，`42` 是一个一般的，非Promise，非thenable的值，所以完成的promise `fulfilledPr` 是为值 `42` 创建的。`Promise.reject("Oops")` 为了原因 `"Oops"` 创建的拒绝的promise `rejectedPr`。

现在让我们来解释为什么如果“`resolve`”这个词（正如 `Promise.resolve(..)` 里的）被明确用于一个既可能完成也可能拒绝的环境时，它没有歧义，反而更加准确：

```
var rejectedTh = {
  then: function(resolved, rejected) {
    rejected( "Oops" );
  }
};

var rejectedPr = Promise.resolve( rejectedTh );
```

就像我们在本章前面讨论的，`Promise.resolve(..)` 将会直接返回收到的纯粹的Promise，或者将收到的thenable展开。如果展开这个thenable之后是一个拒绝状态，那么从 `Promise.resolve(..)` 返回的Promise事实上是相同的拒绝状态。

所以对于这个API方法来说，`Promise.resolve(..)` 是一个好的，准确的名称，因为它实际上既可以得到完成的结果，也可以得到拒绝的结果。

`Promise(..)` 构造器的第一个回调参数既可以展开一个thenable（与 `Promise.resolve(..)` 相同），也可以展开一个Promise：

```
var rejectedPr = new Promise( function(resolve,reject){
  // 用一个被拒绝的promise来解析这个promise
  resolve( Promise.reject( "Oops" ) );
} );

rejectedPr.then(
  function fulfilled(){
    // 永远不会跑到这里
  },
  function rejected(err){
    console.log( err );    // "Oops"
  }
);
```

现在应当清楚了，对于 `Promise(..)` 构造器的第一个参数来说 `resolve(..)` 是一个合适的名称。

**警告：**前面提到的 `reject(..)` 不会像 `resolve(..)` 那样进行展开。如果你向 `reject(..)` 传递一个Promise/thenable值，这个没有被碰过的值将作为拒绝的理由。一个后续的拒绝处理器将会受到你传递给 `reject(..)` 的实际的Promise/thenable，而不是它底层的立即值。

现在让我们将注意力转向提供给 `then(..)` 的回调。它们应当叫什么（在文献和代码中）？我的建议是 `fulfilled(..)` 和 `rejected(..)`：

```
function fulfilled(msg) {
  console.log( msg );
}

function rejected(err) {
  console.error( err );
}

p.then(
  fulfilled,
  rejected
);
```

对于 `then(..)` 的第一个参数的情况，它没有歧义地总是完成状态，所以没有必要使用带有双重意义的“`resolve`”术语。另一方面，ES6语言规范中使用 `onFulfilled(..)` 和 `onRejected(..)` 来标识这两个回调，所以它们是准确的术语。

## 错误处理

我们已经看过几个例子，Promise拒绝——既可以通过有意调用 `reject(..)`，也可以通过意外的JS异常——是如何在异步编程中允许清晰的错误处理的。让我们兜个圈子回去，将我们一带而过的一些细节弄清楚。

对大多数开发者来说，最自然的错误处理形式是同步的 `try..catch` 结构。不幸的是，它仅能用于同步状态，所以在异步代码模式中它帮不上什么忙：

```

function foo() {
  setTimeout( function(){
    baz.bar();
  }, 100 );
}

try {
  foo();
  // 稍后会从`baz.bar()`抛出全局错误
}
catch (err) {
  // 永远不会到这里
}

```

能有 `try..catch` 当然很好，但除非有某些附加的环境支持，它无法与异步操作一起工作。我们将会在第四章中讨论generator时回到这个话题。

在回调中，对于错误处理的模式已经有了一些新兴的模式，最有名的就是“错误优先回调”风格：

```

function foo(cb) {
  setTimeout( function(){
    try {
      var x = baz.bar();
      cb( null, x ); // 成功 !
    }
    catch (err) {
      cb( err );
    }
  }, 100 );
}

foo( function(err, val){
  if (err) {
    console.error( err ); // 倒霉 :(
  }
  else {
    console.log( val );
  }
} );

```

注意：这里的 `try..catch` 仅在 `baz.bar()` 调用立即地，同步地成功或失败时才能工作。如果 `baz.bar()` 本身是一个异步完成的函数，它内部的任何异步错误都不能被捕获。

我们传递给 `foo(..)` 的回调期望通过预留的 `err` 参数收到一个表示错误的信号。如果存在，就假定出错。如果不存在，就假定成功。

这类错误处理在技术上是异步兼容的，但它根本组织的不好。用无处不在的 `if` 语句检查将多层错误优先回调编织在一起，将不可避免地将你置于回调地狱的危险之中（见第二章）。

那么我们回到Promise的错误处理，使用传递给 `then(..)` 的拒绝处理器。Promise不使用流行的“错误优先回调”设计风格，反而使用“分割回调”的风格；一个回调给完成，一个回调给拒绝：

```
var p = Promise.reject( "Oops" );

p.then(
  function fulfilled(){
    // 永远不会到这里
  },
  function rejected(err){
    console.log( err ); // "Oops"
  }
);
```

虽然这种模式表面上看起来十分有道理，但是Promise错误处理的微妙之处经常使它有点儿相当难以全面把握。

考虑下面的代码：

```
var p = Promise.resolve( 42 );

p.then(
  function fulfilled(msg){
    // 数字没有字符串方法，
    // 所以这里抛出一个错误
    console.log( msg.toLowerCase() );
  },
  function rejected(err){
    // 永远不会到这里
  }
);
```

如果 `msg.toLowerCase()` 合法地抛出一个错误（它会的！），为什么我们的错误处理器没有得到通知？正如我们早先解释的，这是因为这个错误处理器是为 `p promise` 准备的，也就是已经被值 `42` 完成的那个promise。`p promise` 是不可变的，所以唯一可以得到错误通知的 promise 是由 `p.then(..)` 返回的那个，而在这里我们没有捕获它。

这应当解释了：为什么Promise的错误处理是易错的。错误太容易被吞掉了，而这很少是你有意这么做的。

**警告：**如果你以一种不合法的方式使用Promise API，而且有错误阻止正常的Promise构建，其结果将是一个立即被抛出的异常，而不是一个拒绝Promise。这是一些导致Promise构建失败的错误用法：`new Promise(null)`，`Promise.all()`，`Promise.race(42)` 等等。如果你没有

足够合法地使用Promise API来首先实际构建一个Promise，你就不能得到一个拒绝Promise！

## 绝望的深渊

几年前Jeff Atwood曾经写到：编程语言总是默认地以这样的方式建立，开发者们会掉入“绝望的深渊” (<http://blog.codinghorror.com/falling-into-the-pit-of-success/>) ——在这里意外会被惩罚——而你不得不更努力地使它正确。他恳求我们相反地创建“成功的深渊”，就是你会默认地掉入期望的（成功的）行为，而如此你不得不更努力地去失败。

毫无疑问，Promise的错误处理是一种“绝望的深渊”的设计。默认情况下，它假定你想让所有的错误都被Promise的状态吞掉，而且如果你忘记监听这个状态，错误就会默默地凋零/死去——通常是绝望的。

为了回避把一个被遗忘/抛弃的Promise的错误无声地丢失，一些开发者宣称Promise链的“最佳实践”是，总是将你的链条以 `catch(..)` 终结，就像这样：

```
var p = Promise.resolve( 42 );

p.then(
  function fulfilled(msg){
    // 数字没有字符串方法,
    // 所以这里抛出一个错误
    console.log( msg.toLowerCase() );
  }
)
.catch( handleErrors );
```

因为我们没有给 `then(..)` 传递拒绝处理器，默认的处理器会顶替上来，它仅仅简单地将错误传播到链条的下一个promise中。如此，在 `p` 中发生的错误，与在 `p` 之后的解析中（比如 `msg.toLowerCase()`）发生的错误都将会过滤到最后的 `handleErrors(..)` 中。

问题解决了，对吧？没那么容易！

要是 `handleErrors(..)` 本身也有错误呢？谁来捕获它？这里还有一个没人注意的 promise：`catch(..)` 返回的promise，我们没有对它进行捕获，也没注册拒绝处理器。

你不能仅仅将另一个 `catch(..)` 贴在链条末尾，因为它也可能失败。Promise链的最后一步，无论它是什么，总有可能，即便这种可能性逐渐减少，悬挂着一个困在未被监听的Promise中的，未被捕获的错误。

听起来像一个不可解的迷吧？

## 处理未被捕获的错误

这不是一个很容易就能完全解决的问题。但是有些接近于解决的方法，或者说更好的方法。

一些Promise库有一些附加的方法，可以注册某些类似于“全局的未处理拒绝”的处理器，全局上不会抛出错误，而是调用它。但是他们识别一个错误是“未被捕获的错误”的方案是，使用一个任意长的计时器，比如说3秒，从拒绝的那一刻开始计时。如果一个Promise被拒绝但没有错误处理在计时器被触发前注册，那么它就假定你不会注册监听器了，所以它是“未被捕获的”。

实践中，这个方法在许多库中工作的很好，因为大多数用法不会在Promise拒绝和监听这个拒绝之间有很明显的延迟。但是这个模式有点儿麻烦，因为3秒实在太随意了（即便它是实证过的），还因为确实有些情况你想让一个Promise在一段不确定的时间内持有它的拒绝状态，而且你不希望你的“未捕获错误”处理器因为这些具有正面含义的不成立（还没处理的“未捕获错误”）而被调用。

另一种常见的建议是，Promise应当增加一个 `done(..)` 方法，它实质上标志着Promise链的“终结”。`done(..)` 不会创建并返回一个Promise，所以传递给 `done(..)` 的回调很明显地不会链接上一个不存在的Promise链，并向它报告问题。

那么接下来会发什么？正如你通常在未处理错误状态下希望的那样，在 `done(..)` 的拒绝处理器内部的任何异常都作为全局的未捕获错误抛出（基本上扔到开发者控制台）：

```
var p = Promise.resolve( 42 );

p.then(
  function fulfilled(msg){
    // 数字没有字符串方法,
    // 所以这里抛出一个错误
    console.log( msg.toLowerCase() );
  }
)
.done( null, handleErrors );

// 如果`handleErrors(..)`自身发生异常，它会在这里被抛出到全局
```

这听起来要比永不终结的链条或随意的超时要吸引人。但最大的问题是，它不是ES6标准，所以不管听起来多么好，它成为一个可靠而普遍的解决方案还有很长的距离。

那我们就卡在这里了？不完全是。

浏览器有一个我们的代码没有的能力：它们可以追踪并确定一个对象什么时候被废弃并可以作为垃圾回收。所以，浏览器可以追踪Promise对象，当它们被当做垃圾回收时，如果在它们内部存在一个拒绝状态，浏览器就可以确信这是一个合法的“未捕获错误”，它可以信心十足地知道应当在开发者控制台上报告这一情况。

注意：在写作本书的时候，Chrome和Firefox都早已试图实现这种“未捕获拒绝”的能力，虽然至多也就是支持的不完整。

然而，如果一个Promise不被垃圾回收——通过许多不同的代码模式，这极其容易不经意地发生——浏览器的垃圾回收检测不会帮你知道或诊断你有一个拒绝的Promise静静地躺在附近。

还有其他选项吗？有。

## 成功的深渊

以下讲的仅仅是理论上，Promise 可能在某一天变成什么样的行为。我相信那会比我们现在拥有的优越许多。而且我想这种改变可能会发生在后ES6时代，因为我不认为它会破坏Web的兼容性。另外，如果你小心行事，它是可以被填补（polyfilled）/预填补（prollyfilled）的。让我们来看一下：

- Promise可以默认为是报告(向开发者控制台)一切拒绝的，就在下一个Job或事件轮询tick，如果就在这时Promise上没有注册任何错误处理器。
- 如果你希望拒绝的Promise在被监听前，将其拒绝状态保持一段不确定的时间。你可以调用 `defer()`，它会压制这个Promise自动报告错误。

如果一个Promise被拒绝，默认地它会吵吵闹闹地向开发者控制台报告这个情况（而不是默认不出声）。你既可以选择隐式地处理这个报告（通过在拒绝之前注册错误处理器），也可以选择明确地处理这个报告（使用 `defer()`）。无论哪种情况，你都控制着这种具有正面意义的不成立。

考虑下面的代码：

```
var p = Promise.reject( "Oops" ).defer();

// `foo(..)` 返回Promise
foo( 42 )
.then(
  function fulfilled(){
    return p;
  },
  function rejected(err){
    // 处理`foo(..)`的错误
  }
);
...
...
```

我们创建了 `p`，我们知道我们会为了使用/监听它的拒绝而等待一会儿，所以我们调用 `defer()`——如此就不会有全局的报告。`defer()` 单纯地返回同一个promise，为了链接的目的。

从 `foo(..)` 返回的promise 当即 就添附了一个错误处理器，所以这隐含地跳出了默认行为，而且不会有全局的关于错误的报告。

但是从 `then(..)` 调用返回的promise没有 `defer()` 或添附错误处理器，所以如果它被拒绝（从它内部的任意一个解析处理器中），那么它就会向开发者控制台报告一个未捕获错误。

这种设计称为成功的深渊。默认情况下，所有的错误不是被处理就是被报告——这几乎是所有开发者在几乎所有情况下所期望的。你要么不得不注册一个监听器，要么不得不有意什么都不做，并指示你要将错误处理推迟到稍后；你仅为这种特定情况选择承担额外的责任。

这种方式唯一真正的危险是，你 `defer()` 了一个Promise但是实际上没有监听/处理它的拒绝。

但你不得不有意地调用 `defer()` 来选择进入绝望深渊——默认是成功深渊——所以对于从你自己的错误中拯救你这件事来说，我们能做的不多。

我觉得对于Promise的错误处理还有希望（在后ES6时代）。我希望上层人物将会重新思考这种情况并考虑选用这种方式。同时，你可以自己实现这种方式（给读者们的挑战练习！），或使用一个聪明的Promise库来为你这么做。

注意：这种错误处理/报告的确切的模型已经在我的 `asynquence` Promise抽象库中实现，我们会在本书的附录A中讨论它。

## Promise模式

我们已经隐含地看到了使用Promise链的顺序模式（这个-然后-这个-然后-那个的流程控制），但是我们还可以在Promise的基础上抽象出许多其他种类的异步模式。这些模式用于简化异步流程控制的表达——它可以使我们的代码更易于推理并且更易于维护——即便是我们程序中最复杂的部分。

有两个这样的模式被直接编码在ES6原生的 `Promise` 实现中，所以我们免费的得到了它们，来作为我们其他模式的构建块儿。

### Promise.all([ .. ])

在一个异步序列（Promise链）中，在任何给定的时刻都只有一个异步任务在被协调——第2步严格地接着第1步，而第3步严格地接着第2步。但要是并发（也叫“并行地”）地去做两个或以上的步骤呢？

用经典的编程术语，一个“门（gate）”是一种等待两个或更多并行/并发任务都执行完再继续的机制。它们完成的顺序无关紧要，只是它们不得不都完成才能让门打开，继而让流程控制通过。

在Promise API中，我们称这种模式为 `all([ .. ])`。

比方说你想同时发起两个Ajax请求，在发起第三个Ajax请求发起之前，等待它们都完成，而不管它们的顺序。考虑这段代码：

```
// `request(..)`是一个兼容Promise的Ajax工具
// 就像我们在本章早前定义的

var p1 = request( "http://some.url.1/" );
var p2 = request( "http://some.url.2/" );

Promise.all( [p1,p2] )
.then( function(msgs){
  // `p1`和`p2`都已完成，这里将它们的消息传入
  return request(
    "http://some.url.3/?v=" + msgs.join( ", "
  );
} )
.then( function(msg){
  console.log( msg );
} );
```

`Promise.all([ .. ])` 期待一个单独的参数，一个 `array`，一般由 `Promise` 的实例组成。从 `Promise.all([ .. ])` 返回的 `promise` 将会收到完成的消息（在这段代码中是 `msgs`），它是一个由所有被传入的 `promise` 的完成消息按照被传入的顺序构成的 `array`（与完成的顺序无关）。

注意：技术上讲，被传入 `Promise.all([ .. ])` 的 `array` 的值可以包括 `Promise`，`thenable`，甚至是立即值。这个列表中的每一个值都实质上通过 `Promise.resolve(..)` 来确保它是一个可以被等待的纯粹的 `Promise`，所以一个立即值将被转化为这个值的一个 `Promise`。如果这个 `array` 是空的，主 `Promise` 将会立即完成。

从 `Promise.resolve(..)` 返回的主 `Promise` 将会在所有组成它的 `promise` 完成之后才会被完成。如果其中任意一个 `promise` 被拒绝，`Promise.all([ .. ])` 的主 `Promise` 将立即被拒绝，并放弃所有其他 `promise` 的结果。

要记得总是给每个 `promise` 添加拒绝/错误处理器，即使和特别是那个从 `Promise.all([ .. ])` 返回的 `promise`。

## Promise.race([ .. ])

虽然 `Promise.all([ .. ])` 并发地协调多个 `Promise` 并假定它们都需要被完成，但是有时候你只想应答“冲过终点的第一个 `Promise`”，而让其他的 `Promise` 被丢弃。

这种模式经典地被称为“闩”，但在 `Promise` 中它被称为一个“竞合（race）”。

警告：虽然“只有第一个冲过终点的算赢”是一个非常合适被比喻，但不幸的是“竞合（race）”是一个被占用的词，因为“竞合状态（race conditions）”通常被认为是程序中的 Bug（见第一章）。不要把 `Promise.race([ .. ])` 与“竞合状态（race conditions）”搞混了。

“竞合状态 (race conditions)”也期待一个单独的 `array` 参数，含有一个或多个Promise，`thenable`，或立即值。与立即值进行竞合并没有多大实际意义，因为很明显列表中的第一个会胜出——就像赛跑时有一个选手在终点线上起跑！

和 `Promise.all([ .. ])` 相似，`Promise.race([ .. ])` 将会在任意一个Promise解析为完成时完成，而且它会在任意一个Promise解析为拒绝时拒绝。

注意：一个“竞合 (race)”需要至少一个“选手”，所以如果你传入一个空的 `array`，`race([..])` 的主Promise将不会立即解析，反而是永远不会被解析。这是砸自己的脚！ES6应当将它规范为要么完成，要么拒绝，或者要么抛出某种同步错误。不幸的是，因为在ES6的 `Promise` 之前的Promise库的优先权高，他们不得不把这个坑留在这儿，所以要小心绝不要传入一个空 `array`。

让我们重温刚才的并发Ajax的例子，但是在 `p1` 和 `p2` 竞合的环境下：

```
// `request(..)`是一个兼容Promise的Ajax工具
// 就像我们在本章早前定义的

var p1 = request( "http://some.url.1/" );
var p2 = request( "http://some.url.2/" );

Promise.race( [p1,p2] )
.then( function(msg){
    // `p1`或`p2`会赢得竞合
    return request(
        "http://some.url.3/?v=" + msg
    );
} )
.then( function(msg){
    console.log( msg );
} );
```

因为只有一个Promise会胜出，所以完成的值是一个单独的消息，而不是一个像 `Promise.all([ .. ])` 中那样的 `array`。

## 超时竞合

我们早先看过这个例子，描述 `Promise.race([ .. ])` 如何能够用于表达“promise超时”模式：

```
// `foo()` 是一个兼容Promise

// `timeoutPromise(...)` 在早前定义过,
// 返回一个在指定延迟之后会被拒绝的Promise

// 为 `foo()` 设置一个超时
Promise.race([
  foo(), // 尝试 `foo()`
  timeoutPromise( 3000 ) // 给它3秒钟
])
.then(
  function(){
    // `foo(...)` 及时地完成了!
  },
  function(err){
    // `foo()` 要么是被拒绝了, 要么就是没有及时完成
    // 可以考察 `err` 来知道是哪一个原因
  }
);

```

这种超时模式在绝大多数情况下工作的很好。但这里有一些微妙的细节要考虑，而且坦率的说它们对于 `Promise.race([ ... ])` 和 `Promise.all([ ... ])` 都同样需要考虑。

## "Finally"

要问的关键问题是，“那些被丢弃/忽略的promise发生了什么？”我们不是从性能的角度在问这个问题——它们通常最终会变成垃圾回收的合法对象——而是从行为的角度（副作用等等）。Promise不能被取消——而且不应当被取消，因为那会摧毁本章稍后的“Promise不可取消”一节中要讨论的外部不可变性——所以它们只能被无声地忽略。

但如果前面例子中的 `foo()` 占用了某些资源，但超时首先触发而且导致这个promise被忽略了呢？这种模式中存在某种东西可以在超时后主动释放被占用的资源，或者取消任何它可能带来的副作用吗？要是你想做的全部只是记录下 `foo()` 超时的事实呢？

一些开发者提议，Promise需要一个 `finally(..)` 回调注册机制，它总是在Promise解析时被调用，而且允许你制定任何可能的清理操作。在当前的语言规范中它还不存在，但它可能会在ES7+中加入。我们不得不边走边看了。

它看起来可能是这样：

```
var p = Promise.resolve( 42 );

p.then( something )
.finally( cleanup )
.then( another )
.finally( cleanup );
```

注意：在各种Promise库中，`finally(..)` 依然会创建并返回一个新的Promise（为了使链条延续下去）。如果`cleanup(..)` 函数返回一个Promise，它将会链入链条，这意味着你可能还有我们刚才讨论的未处理拒绝的问题。

同时，我们可以制造一个静态的帮助工具来让我们观察（但不干涉）Promise的解析：

```
// 填补的安全检查
if (!Promise.observe) {
  Promise.observe = function(pr, cb) {
    // 从侧面观察`pr`的解析
    pr.then(
      function fulfilled(msg){
        // 异步安排回调（作为Job）
        Promise.resolve( msg ).then( cb );
      },
      function rejected(err){
        // 异步安排回调（作为Job）
        Promise.resolve( err ).then( cb );
      }
    );
    // 返回原本的promise
    return pr;
  };
}
```

这是我们在前面的超时例子中如何使用它：

```
Promise.race([
  Promise.observe(
    foo(), // 尝试`foo()`
    function cleanup(msg){
      // 在`foo()`之后进行清理，即便它没有及时完成
    }
  ),
  timeoutPromise( 3000 ) // 给它3秒钟
])
```

这个`Promise.observe(..)` 帮助工具只是描述你如何在不干扰Promise的情况下观测它的完成。其他的Promise库有他们自己的解决方案。不论你怎么做，你都将很可能有个地方想用来确认你的Promise没有意外地被无声地忽略掉。

## all([ .. ]) 与 race([ .. ]) 的变种

原生的ES6Promise带有内建的`Promise.all([ .. ])` 和`Promise.race([ .. ])`，这里还有几个关于这些语义的其他常用的变种模式：

- `none([ .. ])` 很像 `all([ .. ])`，但是完成和拒绝被转置了。所有的Promise都需要被拒绝——拒绝变成了完成值，反之亦然。
- `any([ .. ])` 很像 `all([ .. ])`，但它忽略任何拒绝，所以只有一个需要完成即可，而不是它们所有的。
- `first([ .. ])` 像是一个带有 `any([ .. ])` 的竞合，它忽略任何拒绝，而且一旦有一个Promise完成时，它就立即完成。
- `last([ .. ])` 很像 `first([ .. ])`，但是只有最后一个完成胜出。

某些Promise抽象工具库提供这些方法，但你也可以用Promise机制的 `race([ .. ])` 和 `all([ .. ])`，自己定义他们。

比如，这是我们如何定义 `first(..)`：

```
// 填补的安全检查
if (!Promise.first) {
  Promise.first = function(prs) {
    return new Promise( function(resolve,reject){
      // 迭代所有的promise
      prs.forEach( function(pr){
        // 泛化它的值
        Promise.resolve( pr )
        // 无论哪一个首先成功完成，都由它来解析主promise
        .then( resolve );
      } );
    } );
  };
}
```

注意：这个 `first(..)` 的实现不会在它所有的promise都被拒绝时拒绝；它会简单地挂起，很像 `Promise.race([])`。如果需要，你可以添加一些附加逻辑来追踪每个promise的拒绝，而且如果所有的都被拒绝，就在主promise上调用 `reject()`。我们将此作为练习留给读者。

## 并发迭代

有时候你想迭代一个Promise的列表，并对它们所有都实施一些任务，就像你可以对同步的 `array` 做的那样（比如，`forEach(..)`，`map(..)`，`some(..)`，和 `every(..)`）。如果对每个Promise实施的操作根本上是同步的，它们工作的很好，正如我们在前面的代码段中用过的 `forEach(..)`。

但如果任务从根本上是异步的，或者可以/应当并发地实施，你可以使用许多库提供的异步版本的这些工具方法。

比如，让我们考虑一个异步的 `map(..)` 工具，它接收一个 `array` 值（可以是Promise或任何东西），外加一个对数组中每一个值实施的函数（任务）。`map(..)` 本身返回一个promise，它的完成值是一个持有每个任务的异步完成值的 `array`（以与映射（mapping）相同的顺

序) :

```
if (!Promise.map) {
  Promise.map = function(vals,cb) {
    // 一个等待所有被映射的promise的新promise
    return Promise.all(
      // 注意：普通的数组`map(..)`，
      // 将值的数组变为promise的数组
      vals.map( function(val){
        // 将`val`替换为一个在`val`、
        // 异步映射完成后才解析的新promise
        return new Promise( function(resolve){
          cb( val, resolve );
        } );
      } )
    );
  };
}
```

注意：在这种 `map(..)` 的实现中，你无法表示异步拒绝，但如果一个在映射的回调内部发生一个同步的异常/错误，那么 `Promise.map(..)` 返回的主Promise就会拒绝。

让我们描绘一下对一组Promise（不是简单的值）使用 `map(..)`：

```
var p1 = Promise.resolve( 21 );
var p2 = Promise.resolve( 42 );
var p3 = Promise.reject( "Oops" );

// 将列表中的值翻倍，即便它们在Promise中
Promise.map( [p1,p2,p3], function(pr,done){
  // 确保列表中每一个值都是Promise
  Promise.resolve( pr )
  .then(
    // 将值作为`v`抽取出来
    function(v){
      // 将完成的`v`映射到新的值
      done( v * 2 );
    },
    // 或者，映射到promise的拒绝消息上
    done
  );
} )
.then( function(vals){
  console.log( vals ); // [42,84,"Oops"]
} );
```

## Promise API概览

让我们复习一下我们已经在本章中零散地展开的ES6 `Promise API`。

注意：下面的API尽管在ES6中是原生的，但也存在一些语言规范兼容的填补（不光是扩展Promise库），它们定义了 `Promise` 和与之相关的所有行为，所以即使是在前ES6时代的浏览器中你也以使用原生的Promise。这类填补的其中之一是“Native Promise Only” (<http://github.com/getify/native-promise-only>)，我写的！

## new Promise(..)构造器

**揭示构造器 (revealing constructor)** `Promise(..)` 必须与 `new` 一起使用，而且必须提供一个被同步/立即调用的回调函数。这个函数被传入两个回调函数，它们作为promise的解析能力。我们通常将它们标识为 `resolve(..)` 和 `reject(..)`：

```
var p = new Promise( function(resolve,reject){
    // `resolve(..)`给解析/完成的promise
    // `reject(..)`给拒绝的promise
} );
```

`reject(..)` 简单地拒绝promise，但是 `resolve(..)` 既可以完成promise，也可以拒绝promise，这要看它被传入什么值。如果 `resolve(..)` 被传入一个立即的，非Promise，非thenable的值，那么这个promise将用这个值完成。

但如果 `resolve(..)` 被传入一个Promise或者thenable的值，那么这个值将被递归地展开，而且无论它最终解析结果/状态是什么，都将被promise采用。

## Promise.resolve(..) 和 Promise.reject(..)

一个用于创建已被拒绝的Promise的简便方法是 `Promise.reject(..)`，所以这两个promise是等价的：

```
var p1 = new Promise( function(resolve,reject){
    reject( "Oops" );
} );

var p2 = Promise.reject( "Oops" );
```

与 `Promise.reject(..)` 相似，`Promise.resolve(..)` 通常用来创建一个已完成的Promise。然而，`Promise.resolve(..)` 还会展开thenable值（就像我们已经几次讨论过的）。在这种情况下，返回的Promise将会采用你传入的thenable的解析，它既可能是完成，也可能是拒绝：

```

var fulfilledTh = {
    then: function(cb) { cb( 42 ); }
};

var rejectedTh = {
    then: function(cb,errCb) {
        errCb( "Oops" );
    }
};

var p1 = Promise.resolve( fulfilledTh );
var p2 = Promise.resolve( rejectedTh );

// `p1`将是一个完成的promise
// `p2`将是一个拒绝的promise

```

而且要记住，如果你传入一个纯粹的Promise，`Promise.resolve(..)` 不会做任何事情；它仅仅会直接返回这个值。所以在你不知道其本性的值上调用 `Promise.resolve(..)` 不会有额外的开销，如果它偶然已经是一个纯粹的Promise。

## then(..) 和 catch(..)

每个Promise实例（不是 `Promise` API 名称空间）都有 `then(..)` 和 `catch(..)` 方法，它们允许你为Promise注册成功或拒绝处理器。一旦Promise被解析，它们中的一个就会被调用，但不是都会被调用，而且它们总是会被异步地调用（参见第一章的“Jobs”）。

`then(..)` 接收两个参数，第一个用于完成回调，第二个用户拒绝回调。如果它们其中之一被省略，或者被传入一个非函数的值，那么一个默认的回调就会分别顶替上来。默认的完成回调简单地将值向下传递，而默认的拒绝回调简单地重新抛出（传播）收到的拒绝理由。

`catch(..)` 仅仅接收一个拒绝回调作为参数，而且会自动的顶替一个默认的成功回调，就像我们讨论过的。换句话说，它等价于 `then(null,..)`：

```

p.then( fulfilled );
p.then( fulfilled, rejected );
p.catch( rejected ); // 或者`p.then( null, rejected )`

```

`then(..)` 和 `catch(..)` 也会创建并返回一个新的promise，它可以用来表达Promise链式流程控制。如果完成或拒绝回调有异常被抛出，这个返回的promise就会被拒绝。如果这两个回调之一返回一个立即，非Promise，非thenable值，那么这个值就会作为被返回的promise的完成。如果完成处理器指定地返回一个promise或thenable值这个值就会被展开而且变成被返回的promise的解析。

## Promise.all([ .. ]) 和 Promise.race([ .. ])

在ES6的 `Promise API` 的静态帮助方法 `Promise.all([ .. ])` 和 `Promise.race([ .. ])` 都创建一个 `Promise` 作为它们的返回值。这个 `promise` 的解析完全由你传入的 `promise` 数组控制。

对于 `Promise.all([ .. ])`，为了被返回的 `promise` 完成，所有你传入的 `promise` 都必须完成。如果其中任意一个被拒绝，返回的主 `promise` 也会立即被拒绝（丢弃其他所有 `promise` 的结果）。至于完成状态，你会收到一个含有所有被传入的 `promise` 的完成值的 `array`。至于拒绝状态，你仅会收到第一个 `promise` 拒绝的理由值。这种模式通常称为“门”：在门打开前所有人都必须到达。

对于 `Promise.race([ .. ])`，只有第一个解析（成功或拒绝）的 `promise` 会“胜出”，而且不论解析的结果是什么，都会成为被返回的 `promise` 的解析结果。这种模式通常成为“闩”：第一个打开门闩的人才能进来。考虑这段代码：

```
var p1 = Promise.resolve( 42 );
var p2 = Promise.resolve( "Hello World" );
var p3 = Promise.reject( "Oops" );

Promise.race( [p1,p2,p3] )
.then( function(msg){
    console.log( msg );           // 42
} );

Promise.all( [p1,p2,p3] )
.catch( function(err){
    console.error( err );        // "Oops"
} );

Promise.all( [p1,p2] )
.then( function(msgs){
    console.log( msgs );         // [42,"Hello World"]
} );
```

**警告：**要小心！如果一个空的 `array` 被传入 `Promise.all([ .. ])`，它会立即完成，但 `Promise.race([ .. ])` 却会永远挂起，永远不会解析。

ES6 的 `Promise API` 十分简单和直接。对服务于大多数基本的异步情况来说它足够好了，而且当你要把你的代码从回调地狱变为某些更好的东西时，它是一个开始的好地方。

但是依然还有许多应用程序所要求的精巧的异步处理，由于 `Promise` 本身所受的限制而不能解决。在下一节中，为了有效利用 `Promise` 库，我们将深入检视这些限制。

## Promise 的限制

本节中我们将要讨论的许多细节已经在这一章中被提及了，但我们将明确地复习这些限制。

## 顺序的错误处理

我们在本章前面的部分详细讲解了Promise风格的错误处理。Promise的设计方式——特别是他们如何链接——所产生的限制，创建了一个非常容易掉进去的陷阱，Promise链中的错误会被意外地无声地忽略掉。

但关于Promise的错误还有一些其他事情要考虑。因为Promise链只不过是将组成它的Promise连在一起，没有一个实体可以用来将整个链条表达为一个单独的东西，这意味着没有外部的方法能够监听可能发生的任何错误。

如果你构建一个不包含错误处理器的Promise链，这个链条的任意位置发生的任何错误都将沿着链条向下无限传播，直到被监听为止（通过在某一步上注册拒绝处理器）。所以，在这种特定情况下，拥有链条的最后一个promise的引用就够了（下面代码段中的 `p`），因为你可以在这里注册拒绝处理器，而且它会被所有传播的错误通知：

```
// `foo(..)`、`STEP2(..)` 和 `STEP3(..)`
// 都是promise兼容的工具

var p = foo( 42 )
  .then( STEP2 )
  .then( STEP3 );
```

虽然这看起来有点儿小糊涂，但是这里的 `p` 没有指向链条中的第一个promise（`foo(42)` 调用中来的那一个），而是指向了最后一个promise，来自于 `then(STEP3)` 调用的那个。

另外，这个promise链条上看不到一个步骤做了自己的错误处理。这意味着你可以在 `p` 上注册一个拒绝处理器，如果在链条的任意位置发生了错误，它就会被通知。

```
p.catch( handleErrors );
```

但如果这个链条中的某一步事实上做了自己的错误处理（也许是隐藏/抽象出去了，所以你看不到），那么你的 `handleErrors(..)` 就不会被通知。这可能是你想要的——它毕竟是一个“被处理过的拒绝”——但它也可能不是你想要的。完全缺乏被通知的能力（被“已处理过的”拒绝错误通知）是一个在某些用法中约束功能的一种限制。

它基本上和 `try..catch` 中存在的限制是相同的，它可以捕获一个异常并简单地吞掉。所以这不是一个 **Promise**特有的问题，但它确实是一个我们希望绕过的限制。

不幸的是，许多时候Promise链序列的中间步骤不会被留下引用，所以没有这些引用，你就不能添加错误处理器来可靠地监听错误。

## 单独的值

根据定义，Promise只能有一个单独的完成值或一个单独的拒绝理由。在简单的例子中，这没什么大不了的，但在更精巧的场景下，你可能发现这个限制。

通常的建议是构建一个包装值（比如 object 或 array）来包含这些多个消息。这个方法好用，但是在你的Promise链的每一步上把消息包装再拆开显得十分尴尬和烦人。

## 分割值

有时你可以将这种情况当做一个信号，表示你可以/应当将问题拆分为两个或更多的Promise。

想象你有一个工具 `foo(..)`，它异步地产生两个值（`x` 和 `y`）：

```
function getY(x) {
  return new Promise( function(resolve,reject){
    setTimeout( function(){
      resolve( (3 * x) - 1 );
    }, 100 );
  } );
}

function foo(bar,baz) {
  var x = bar * baz;

  return getY( x )
    .then( function(y){
      // 将两个值包装进一个容器
      return [x,y];
    } );
}

foo( 10, 20 )
  .then( function(msgs){
    var x = msgs[0];
    var y = msgs[1];

    console.log( x, y );    // 200 599
  } );
}
```

首先，让我们重新安排一下 `foo(..)` 返回的东西，以便于我们不必再将 `x` 和 `y` 包装进一个单独的 array 值中来传送给一个Promise。相反，我们将每一个值包装进它自己的promise：

```

function foo(bar,baz) {
  var x = bar * baz;

  // 将两个promise返回
  return [
    Promise.resolve( x ),
    getY( x )
  ];
}

Promise.all(
  foo( 10, 20 )
)
.then( function(msgs){
  var x = msgs[0];
  var y = msgs[1];

  console.log( x, y );
} );

```

一个promise的 array 真的要比传递给一个单独的Promise的值的 array 要好吗？语法上，它没有太多改进。

但是这种方式更加接近于Promise的设计原理。现在它更易于在未来将 x 与 y 的计算分开，重构进两个分离的函数中。它更清晰，也允许调用端代码更灵活地安排这两个promise——这里使用了 `Promise.all([ .. ])`，但它当然不是唯一的选择——而不是将这样的细节在 `foo(..)` 内部进行抽象。

## 展开/散开参数

`var x = ..` 和 `var y = ..` 的赋值依然是一个尴尬的负担。我们可以在一个帮助工具中利用一些函数式技巧（向Reginald Braithwaite致敬，在推特上 @raganwald）：

```

function spread(fn) {
  return Function.prototype.bind( fn, null );
}

Promise.all(
  foo( 10, 20 )
)
.then(
  spread( function(x,y){
    console.log( x, y );      // 200 599
  } )
)

```

看起来好些了！当然，你可以内联这个函数式魔法来避免额外的帮助函数：

```

Promise.all(
  foo( 10, 20 )
)
.then( Function.apply.bind(
  function(x,y){
    console.log( x, y );    // 200 599
  },
  null
) );

```

这个技巧可能很整洁，但是ES6给了我们一个更好的答案：解构（destructuring）。数组的解构赋值形式看起来像这样：

```

Promise.all(
  foo( 10, 20 )
)
.then( function(msgs){
  var [x,y] = msgs;

  console.log( x, y );    // 200 599
} );

```

最棒的是，ES6提供了数组参数解构形式：

```

Promise.all(
  foo( 10, 20 )
)
.then( function([x,y]){
  console.log( x, y );    // 200 599
} );

```

我们现在已经接受了“每个Promise一个值”的准则，继续让我们把模板代码最小化！

注意：更多关于ES6解构形式的信息，参阅本系列的 *ES6与未来*。

## 单次解析

Promise的一个最固有的行为之一就是，一个Promise只能被解析一次（成功或拒绝）。对于多数异步用例来说，你仅仅取用这个值一次，所以这工作的很好。

但也有许多异步情况适用于一个不同的模型——更类似于事件和/或数据流。表面上看不清Promise能对这种用例适应的多好，如果能的话。没有基于Promise的重大抽象过程，它们完全缺乏对多个值解析的处理。

想象这样一个场景，你可能想要为响应一个刺激（比如事件）触发一系列异步处理步骤，而这实际上将会发生多次，比如按钮点击。

这可能不会像你想的那样工作：

```
// `click(..)` 绑定了一个DOM元素的 `"click"` 事件
// `request(..)` 是先前定义的支持Promise的Ajax

var p = new Promise( function(resolve,reject){
    click( "#mybtn", resolve );
} );

p.then( function(evt){
    var btnID = evt.currentTarget.id;
    return request( "http://some.url.1/?id=" + btnID );
} )
.then( function(text){
    console.log( text );
} );
```

这里的 behavior 仅能在你的应用程序只让按钮被点击一次的情况下工作。如果按钮被点击第二次，promise `p` 已经被解析了，所以第二个 `resolve(..)` 将被忽略。

相反的，你可能需要将模式反过来，在每次事件触发时创建一个全新的Promise链：

```
click( "#mybtn", function(evt){
    var btnID = evt.currentTarget.id;

    request( "http://some.url.1/?id=" + btnID )
    .then( function(text){
        console.log( text );
    } );
} );
```

这种方式会好用，为每个按钮上的 `"click"` 事件发起一个全新的Promise序列。

但是除了在事件处理器内部定义一整套Promise链看起来很丑以外，这样的设计在某种意义上违背了关注/能力分离原则（SoC）。你可能非常想在一个你的代码不同的地方定义事件处理器：你定义对事件的响应（Promise链）的地方。如果没有帮助机制，在这种模式下这么做很尴尬。

注意：这种限制的另一种表述方法是，如果我们能够构建某种能在它上面进行Promise链监听的“可监听对象（observable）”就好了。有一些库已经建立这些抽象（比如RxJS——<http://rxjs.codeplex.com/>），但是这种抽象看起来是如此的重，以至于你甚至再也看不到Promise的性质。这样的重抽象带来一个重要的问题：这些机制是否像Promise本身被设计的一样可靠。我们将会在附录B中重新讨论“观察者（Observable）”模式。

## 惰性

对于在你的代码中使用Promise而言一个实在的壁垒是，现存的所有代码都没有支持Promise。如果你有许多基于回调的代码，让代码保持相同的风格容易多了。

“一段基于动作（用回调）的代码将仍然基于动作（用回调），除非一个更聪明，具有Promise意识的开发者对它采取行动。”

Promise提供了一种不同的模式规范，如此，代码的表达方式可能会变得有一点儿不同，某些情况下，则根本不同。你不得不有意这么做，因为Promise不仅只是把那些为你服务至今的老式编码方法自然地抖落掉。

考虑一个像这样的基于回调的场景：

```
function foo(x, y, cb) {
  ajax(
    "http://some.url.1/?x=" + x + "&y=" + y,
    cb
  );
}

foo( 11, 31, function(err, text) {
  if (err) {
    console.error( err );
  }
  else {
    console.log( text );
  }
});
```

将这个基于回调的代码转换为支持Promise的代码的第一步该怎么做，是立即明确的吗？这要看你的经验。你练习的越多，它就感觉越自然。但当然，Promise没有明确告知到底怎么做——没有一个放之四海而皆准的答案——所以这要靠你的责任心。

就像我们以前讲过的，我们绝对需要一种支持Promise的Ajax工具来取代基于回调的工具，我们可以称它为`request(..)`。你可以制造自己的，正如我们已经做过的。但是不得不为每个基于回调的工具手动定义Promise相关的包装器的负担，使得你根本就不太可能选择将代码重构为Promise相关的。

Promise没有为这种限制提供直接的答案。但是大多数Promise库确实提供了帮助函数。想象一个这样的帮助函数：

```
// 填补的安全检查
if (!Promise.wrap) {
  Promise.wrap = function(fn) {
    return function() {
      var args = [].slice.call( arguments );
      return new Promise( function(resolve, reject){
        fn.apply(
          null,
          args.concat( function(err,v){
            if (err) {
              reject( err );
            }
            else {
              resolve( v );
            }
          })
        );
      });
    };
  };
}
```

好吧，这可不是一个微不足道的工具。然而，虽然他可能看起来有点儿令人生畏，但也没有你想的那么糟。它接收一个函数，这个函数期望一个错误优先风格的回调作为第一个参数，然后返回一个可以自动创建Promise并返回的新函数，然后为你替换掉回调，与Promise的完成/拒绝连接在一起。

与其浪费太多时间谈论这个 `Promise.wrap(..)` 帮助函数如何工作，还不如让我们来看看如何使用它：

```
var request = Promise.wrap( ajax );

request( "http://some.url.1/" )
.then( ... )
..
```

哇哦，真简单！

`Promise.wrap(..)` 不会生产Promise。它生产一个将会生产Promise的函数。某种意义上，一个Promise生产函数可以被看做一个“Promise工厂”。我提议将这样的东西命名为“promisory”（"Promise" + "factory"）。

这种将期望回调的函数包装为一个Promise相关的函数的行为，有时被称为“提升（lifting）”或“promise化（promisifying）”。但是除了“提升过的函数”以外，看起来没有一个标准的名词来称呼这个结果函数，所以我更喜欢“promisory”，因为我认为他更具描述性。

注意: Promisory不是一个瞎编的词。它是一个真实存在的词汇,而且它的定义是含有或载有一个promise。这正是这些函数所做的,所以这个术语匹配得简直完美!

那么, `Promise.wrap.ajax` 生产了一个我们称为 `request(..)` 的 `ajax(..)` promisory,而这个 promisory 为 Ajax 应答生产 Promise。

如果所有的函数已经都是 promisory, 我们就不需要自己制造它们, 所以额外的步骤就有点儿多余。但是至少包装模式是(通常都是)可重复的, 所以我们可以把它放进 `Promise.wrap(..)` 帮助函数中来支援我们的 promise 编码。

那么回到刚才的例子, 我们需要为 `ajax(..)` 和 `foo(..)` 都做一个 promisory。

```
// 为`ajax(..)`制造一个promisory
var request = Promise.wrap( ajax );

// 重构`foo(..)`, 但是为了代码其他部分
// 的兼容性暂且保持它对外是基于回调的
// —仅在内部使用`request(..)`的promise
function foo(x,y,cb) {
    request(
        "http://some.url.1/?x=" + x + "&y=" + y
    )
    .then(
        function fulfilled(text){
            cb( null, text );
        },
        cb
    );
}

// 现在, 为了这段代码本来的目的, 为`foo(..)`制造一个promisory
var betterFoo = Promise.wrap( foo );

// 并使用这个promisory
betterFoo( 11, 31 )
.then(
    function fulfilled(text){
        console.log( text );
    },
    function rejected(err){
        console.error( err );
    }
);
}
```

当然, 虽然我们将 `foo(..)` 重构为使用我们的新 `request(..)` promisory, 我们可以将 `foo(..)` 本身制成 promisory, 而不是保留基于会掉的实现并需要制造和使用后续的 `betterFoo(..)` promisory。这个决定只是要看 `foo(..)` 是否需要保持基于回调的形式以便于代码的其他部分兼容。

考虑这段代码：

```
// 现在，`foo(..)`也是一个promisory
// 因为它委托到`request(..)`` promisory
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

foo( 11, 31 )
.then( ... )
..
```

虽然ES6的Promise没有为这样的promisory包装提供原生的帮助函数，但是大多数库提供它们，或者你可以制造自己的。不管哪种方法，这种Promise特定的限制是可以不费太多劲儿就可以解决的（当然是和回调地狱的痛苦相比！）。

## Promise不可撤销

一旦你创建了一个Promise并给它注册了一个完成和/或拒绝处理器，就没有什么你可以从外部做的事情能停止这个进程，即使是某些其他的事情使这个任务变得毫无意义。

注意：许多Promise抽象库都提供取消Promise的功能，但这是一个非常坏的主意！许多开发者都希望Promise被原生地设计为具有外部取消能力，但问题是这将允许Promise的一个消费者/监听器影响某些其他消费者监听同一个Promise的能力。这违反了未来值得可靠性原则（外部不可变），另外就是嵌入了“远距离行为（action at a distance）”的反模式（[http://en.wikipedia.org/wiki/Action\\_at\\_a\\_distance\\_%28computer\\_programming%29](http://en.wikipedia.org/wiki/Action_at_a_distance_%28computer_programming%29)）。不管它看起来多么有用，它实际上会直接将你引回与回调地狱相同的噩梦。

考虑我们早先的Promise超时场景：

```
var p = foo( 42 );

Promise.race([
    p,
    timeoutPromise( 3000 )
])
.then(
    doSomething,
    handleError
);

p.then( function(){
    // 即使是在超时的情况下也会发生 :(
});
```

“超时”对于promise `p` 来说是外部的，所以 `p` 本身继续运行，这可能不是我们想要的。

一个选项是侵入性地定义你的解析回调：

```
var OK = true;

var p = foo( 42 );

Promise.race([
  p,
  timeoutPromise( 3000 )
  .catch( function(err){
    OK = false;
    throw err;
  })
])
.then(
  doSomething,
  handleError
);

p.then( function(){
  if (OK) {
    // 仅在没有超时的情况下发生！ :)
  }
});
```

很难看。这可以工作，但是远不理想。一般来说，你应当避免这样的场景。

但是如果你不能，这种解决方案的丑陋应当是一个线索，说明取消是一种属于在Promise之上的更高层抽象的功能。我推荐你找一个Promise抽象库来辅助你，而不是自己使用黑科技。

注意：我的 `asynquence` Promise抽象库提供了这样的抽象，还为序列提供了一个 `abort()` 能力，这一切将在附录A中讨论。

一个单独的Promise不是真正的流程控制机制（至少没有多大实际意义），而流程控制机制正是取消要表达的；这就是为什么Promise取消显得尴尬。

相比之下，一个链条的Promise集合在一起——我称之为“序列”——是一个流程控制的表达，如此在这一层面的抽象上它就适于定义取消。

没有一个单独的Promise应该是可以取消的，但是一个序列可以取消是有道理的，因为 you 不会将一个序列作为一个不可变值传来传去，就像Promise那样。

## Promise性能

这种限制既简单又复杂。

比较一下在基于回调的异步任务链和Promise链上有多少东西在动，很明显Promise有多得多的事情发生，这意味着它们自然地会更慢一点点。回想一下Promise提供的保证信任的简单列表，将它和你为了达到相同保护效果而在回调上面添加的特殊代码比较一下。

更多工作要做，更多的安全要保护，意味着Promise与赤裸裸的，不可靠的回调相比确实更慢。这些都很明显，可能很容易萦绕在你脑海中。

但是慢多少？好吧……这实际上是一个难到不可思议的问题，无法绝对，全面地回答。

坦白地说，这是一个比较苹果和橘子的问题，所以可能是问错了。你实际上应当比较的是，带有所有手动保护层的经过特殊处理的回调系统，是否比一个Promise实现要快。

如果说Promise有一种合理的性能限制，那就是它并不将可靠性保护的选项罗列出来让你选择——你总是一下得到全部。

如果我们承认Promise一般来说要比它的非Promise，不可靠的回调等价物慢一点儿——假定在有些地方你觉得你可以自己调整可靠性的缺失——难道这意味着Promise应当被全面地避免，就好像你的整个应用程序仅仅由一些可能的“必须绝对最快”的代码驱动着？

扪心自问：如果你的代码有那么合理，那么对于这样的任务，**JavaScript**是正确的选择吗？为了运行应用程序**JavaScript**可以被优化得十分高效（参见第五章和第六章）。但是在Promise提供的所有好处的光辉之下，过于沉迷它微小的性能权衡，真的合适吗？

另一个微妙的问题是Promise使所有事情都成为异步的，这意味着有些应当立即完成的（同步的）步骤也要推迟到下一个Job步骤中（参见第一章）。也就是说一个Promise任务序列要比使用回调连接的相同序列要完成的稍微慢一些是可能的。

当然，这里的问题是：这些关于性能的微小零头的潜在疏忽，和我们在本章通篇阐述的Promise带来的益处相比，还值得考虑吗？

我的观点是，在几乎所有你可能认为Promise的性能慢到了需要被考虑的情况下，完全回避Promise并将它的可靠性和组合性优化掉，实际上一种反模式。

相反地，你应当默认地在代码中广泛使用它们，然后再记录并分析你的应用程序的热（关键）路径。Promise真的是瓶颈？还是它们只是理论上慢了下来？只有在那之后，拿着实际合法的基准分析观测数据（参见第六章），再将Promise从这些关键区域中重构移除才称得上是合理与谨慎。

Promise是有一点儿慢，但作为交换你得到了很多内建的可靠性，无Zalgo的可预测性，与组合性。也许真正的限制不是它们的性能，而是你对它们的益处缺乏认识？

## 复习

Promise很牛。用它们。它们解决了肆虐在回调代码中的控制倒转问题。

它们没有摆回调，而是重新定向了这些回调的组织安排方式，是它成为一种坐落于我们和其他工具之间的可靠的中间机制。

Promise链还开始以顺序的风格定义了一种更好的（当然，还不完美）表达异步流程的方式，它帮我们的大脑更好的规划和维护异步JS代码。我们会在下一章中看到一个更好的解决这个问题的方法！

# 你不懂JS: 异步与性能

## 第四章: Generator

在第二章中，我们发现了在使用回调表达异步流程控制时的两个关键缺陷：

- 基于回调的异步与我们的大脑规划任务的各个步骤的过程不相符。
- 由于 控制倒转 回调是不可靠的，也是不可组合的。

在第三章中，我们详细地讨论了Promise如何反转回调的 控制倒转，重建了可靠性/可组合性。

现在让我们把注意力集中到用一种顺序的，看起来同步的风格来表达异步流程控制。使这一切成为可能的“魔法”是ES6的 **generator**。

### 打破运行至完成

在第一章中，我们讲解了一个JS开发者们在他们的代码中几乎永恒依仗的一个认识：一旦函数开始执行，它将运行直至完成，没有其他的代码可以在运行期间干扰它。

这看起来可能很滑稽，ES6引入了一种新型的函数，它不按照“运行至完成”的行为进行动作。这种新型的函数称为“generator（生成器）”。

为了理解它的含义，然我们看看这个例子：

```
var x = 1;

function foo() {
    x++;
    bar();           // <-- 这一行会发生什么？
    console.log("x:", x);
}

function bar() {
    x++;
}

foo();           // x: 3
```

在这个例子中，我们确信 bar() 会在 x++ 和 console.log(x) 之间运行。但如果 bar() 不在这里呢？很明显结果将是 2 而不是 3。

现在让我们来燃烧你的大脑。要是 `bar()` 不存在，但以某种方式依然可以在 `x++` 和 `console.log(x)` 语句之间运行呢？这可能吗？

在 抢占式（**preemptive**）多线程语言中，`bar()` 去“干扰”并正好在两个语句之间那一时刻运行，实质上时可能的。但JS不是抢占式的，也（还）不是多线程的。但是，如果 `foo()` 本身可以用某种办法在代码的这一部分指示一个“暂停”，那么这种“干扰”（并发）的协作形式就是可能的。

注意：我使用“协作”这个词，不仅是因为它与经典的并发术语有关联（见第一章），也因为正如你将在下一个代码段中看到的，ES6在代码中指示暂停点的语法是 `yield` ——暗示一个让出控制权的礼貌的协作。

这就是实现这种协作并发的ES6代码：

```
var x = 1;

function *foo() {
    x++;
    yield; // 暂停！
    console.log("x:", x);
}

function bar() {
    x++;
}
```

注意：你将很可能在大多数其他的JS文档/代码中看到，一个generator的声明被格式化为 `function* foo() { .. }` 而不是我在这里使用的 `function *foo() { .. }` ——唯一的区别是摆放 `*` 位置的风格。这两种形式在功能性/语法上是完全一样的，还有第三种 `function*foo() { .. }` （没空格）形式。这两种风格存在争议，但我基本上偏好 `function *foo..`，因为当我在写作中用 `*foo()` 引用一个generator时，这种形式可以匹配我写的东西。如果我只说 `foo()`，你就不会清楚地知道我是在说一个generator还是一个一般的函数。这纯粹是一个风格偏好的问题。

现在，我们该如何运行上面的代码，使 `bar()` 在 `yield` 那一点取代 `*foo()` 的执行？

```
// 构建一个迭代器`it`来控制generator
var it = foo();

// 在这里开始`foo()`！
it.next();
x;                      // 2
bar();
x;                      // 3
it.next();               // x: 3
```

好了，这两段代码中有不少新的，可能使人困惑的东西，所以我们得跋涉好一段了。在我们用ES6的generator来讲解不同的机制/语法之前，让我们过一遍这个行为的流程：

1. `it = foo()` 操作不会执行 `*foo()` generator，它只不过构建了一个用来控制它执行的迭代器 (`iterator`)。我们一会更多地讨论 迭代器。
2. 第一个 `it.next()` 启动了 `*foo()` generator，并且运行 `*foo()` 第一行上的 `x++`。
3. `*foo()` 在 `yield` 语句处暂停，就在这时第一个 `it.next()` 调用结束。在这个时刻，`*foo()` 依然运行而且是活动的，但是处于暂停状态。
4. 我们观察 `x` 的值，现在它是 `2`。
5. 我们调用 `bar()`，它再一次用 `x++` 递增 `x`。
6. 我们再一次观察 `x` 的值，现在它是 `3`。
7. 最后的 `it.next()` 调用使 `*foo()` generator从它暂停的地方继续运行，而后运行使用 `x` 的当前值 `3` 的 `console.log(...)` 语句。

清楚的是，`*foo()` 启动了，但没有运行到底——它停在 `yield`。我们稍后继续 `*foo()`，让它完成，但这甚至不是必须的。

所以，一个generator是一种函数，它可以开始和停止一次或多次，甚至没必要一定要完成。虽然为什么它很强大看起来不那么明显，但正如我们将要在本章剩下的部分将要讲到的，它是我们用于在我们的代码中构建“generator异步流程控制”模式的基础构件之一。

## 输入和输出

一个generator函数是一种带有我们刚才提到的新型处理模型的函数。但它仍然是一个函数，这意味着依旧有一些不变的基本原则——即，它依然接收参数（也就是“输入”），而且它依然返回一个值（也就是“输出”）：

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

var it = foo( 6, 7 );

var res = it.next();

res.value;           // 42
```

我们将 `6` 和 `7` 分别作为参数 `x` 和 `y` 传递给 `*foo(..)`。而 `*foo(..)` 将值 `42` 返回给调用端代码。

现在我们可以看到发生器的调用和一般函数的调用的一个不同之处了。`foo(6,7)` 显然看起来很熟悉。但微妙的是，`*foo(..)` generator不会像一个函数那样实际运行起来。

相反，我们只是创建了迭代器对象，将它赋值给变量 `it`，来控制 `*foo(..)` generator。当我们调用 `it.next()` 时，它指示 `*foo(..)` generator 从现在的位置向前推进，直到下一个 `yield` 或者 generator 的最后。

`next(..)` 调用的结果是一个带有 `value` 属性的对象，它持有从 `*foo(..)` 返回的任何值（如果有的话）。换句话说，`yield` 导致在 generator 运行期间，一个值被从中发送出来，有点儿像一个中间的 `return`。

但是，为什么我们需要这个完全间接的迭代器对象来控制 generator 还不清楚。我们回头会讨论它的，我保证。

## 迭代通信

generator 除了接收参数和拥有返回值，它们还内建有更强大，更吸引人的输入/输出消息能力，这是通过使用 `yield` 和 `next(..)` 实现的。

考虑下面的代码：

```
function *foo(x) {
  var y = x * (yield);
  return y;
}

var it = foo( 6 );

// 开始`foo(..)`
it.next();

var res = it.next( 7 );

res.value;           // 42
```

首先，我们将 `6` 作为参数 `x` 传入。之后我们调用 `it.next()`，它启动了 `*foo(..)`。

在 `*foo(..)` 内部，`var y = x ..` 语句开始被处理，但它运行到了一个 `yield` 表达式。就在这时，它暂停了 `*foo(..)`（就在赋值语句的中间！），而且请求调用端代码为 `yield` 表达式提供一个结果值。接下来，我们调用 `it.next(7)`，将 `7` 这个值传回去作为暂停的 `yield` 表达式的结果。

所以，在这个时候，赋值语句实质上是 `var y = 6 * 7`。现在，`return y` 将值 `42` 作为结果返回给 `it.next(7)` 调用。

注意一个非常重要，而且即便是对于老练的 JS 开发者也非常容易犯糊涂的事情：根据你的角度，在 `yield` 和 `next(..)` 调用之间存在着错位。一般来说，你所拥有的 `next(..)` 调用的数量，会比你所拥有的 `yield` 语句的数量多一个——前面的代码段中有一个 `yield` 和两个 `next(..)` 调用。

为什么会有这样的错位？

因为第一个 `next(..)` 总是启动一个**generator**，然后运行至第一个 `yield`。但是第二个 `next(..)` 调用满足了第一个暂停的 `yield` 表达式，而第三个 `next(..)` 将满足第二个 `yield`，如此反复。

两个疑问的故事

实际上，你主要考虑的是哪部分代码会影响你是否感知到错位。

仅考虑**generator**代码：

```
var y = x * (yield);
return y;
```

这第一个 `yield` 基本上是在问一个问题：“我应该在这里插入什么值？”

谁来回答这个问题？好吧，第一个 `next()` 在这个时候已经为了启动**generator**而运行过了，所以很明显它不能回答这个问题。所以，第二个 `next(..)` 调用必须回答由第一个 `yield` 提出的问题。

看到错位了吧——第二个对第一个？

但是让我们反转一下我们的角度。让我们不从**generator**的角度看问题，而从迭代器的角度看。

为了恰当地描述这种角度，我们还需要解释一下，消息可以双向发送——`yield ..` 作为表达式可以发送消息来应答 `next(..)` 调用，而 `next(..)` 可以发送值给暂停的 `yield` 表达式。考虑一下这段稍稍调整过的代码：

```
function *foo(x) {
  var y = x * (yield "Hello"); // <-- 让出一个值！
  return y;
}

var it = foo( 6 );

var res = it.next(); // 第一个`next()`，不传递任何东西
res.value;           // "Hello"

res = it.next( 7 ); // 传递`7`给等待中的`yield`
res.value;           // 42
```

`yield ..` 和 `next(..)` 一起成对地在**generator**运行期间构成了一个双向消息传递系统。

那么，如果只看迭代器代码：

```

var res = it.next();      // 第一个`next()`，不传递任何东西
res.value;                // "Hello"

res = it.next( 7 );       // 传递`7`给等待中的`yield`
res.value;                // 42

```

注意：我们没有传递任何值给第一个 `next()` 调用，而且是故意的。只有一个暂停的 `yield` 才能接收这样一个被 `next(..)` 传递的值，但是当我们调用第一个 `next()` 时，在 generator 的最开始并没有任何暂停的 `yield` 可以接收这样的值。语言规范和所有兼容此语言规范的浏览器只会无声地丢弃任何传入第一个 `next()` 的东西。传递这样的值是一个坏主意，因为你只不过创建了一些令人困惑的无声“失败”的代码。所以，记得总是用一个无参数的 `next()` 来启动 generator。

第一个 `next()` 调用（没有任何参数的）基本上是在问一个问题：“`*foo(..) generator` 将要给我的下一个值是什么？”，谁来回答这个问题？第一个 `yield` 表达式。

看到了？这里没有错位。

根据你认为是谁在问问题，在 `yield` 和 `next(..)` 之间的错位既存在又不存在。

但等一下！跟 `yield` 语句的数量比起来，还有一个额外的 `next()`。那么，这个最后的 `it.next(7)` 调用又一次在询问 generator 下一个产生的值是什么。但是没有 `yield` 语句剩下可以回答了，不是吗？那么谁来回答？

`return` 语句回答这个问题！

而且如果在你的 generator 中没有 `return`——比起一般的函数，generator 中的 `return` 当然不再是必须的——总会有一个假定/隐式的 `return;`（也就是 `return undefined;`），它默认的目的就是回答由最后的 `it.next(7)` 调用提出的问题。

这些问题与回答——用 `yield` 和 `next(..)` 进行双向消息传递——十分强大，但还是看不出来这些机制与异步流程控制有什么联系。我们正在接近真相！

## 多迭代器

从语法使用上来看，当你用一个迭代器来控制 generator 时，你正在控制声明的 generator 函数本身。但这里有一个容易忽视的微妙细节：每当你构建一个迭代器，你都隐含地构建了一个将由这个迭代器控制的 generator 的实例。

你可以让同一个 generator 的多个实例同时运行，它们甚至可以互动：

```

function *foo() {
  var x = yield 2;
  z++;
  var y = yield (x * z);
  console.log( x, y, z );
}

var z = 1;

var it1 = foo();
var it2 = foo();

var val1 = it1.next().value;           // 2 <-- 让出2
var val2 = it2.next().value;           // 2 <-- 让出2

val1 = it1.next( val2 * 10 ).value;    // 40 <-- x:20, z:2
val2 = it2.next( val1 * 5 ).value;     // 600 <-- x:200, z:3

it1.next( val2 / 2 );                // y:300
// 20 300 3
it2.next( val1 / 4 );                // y:10
// 200 10 3

```

警告：同一个generator的多个并发运行实例的最常见的用法，不是这样的互动，而是generator在没有输入的情况下，从一些连接着的独立资源中产生它自己的值。我们将在下一节中更多地讨论产生值。

让我们简单地走一遍这个处理过程：

1. 两个 `*foo()` 在同时启动，而且两个 `next()` 都分别从 `yield 2` 语句中得到了 `2` 的 `value`。
2. `val2 * 10` 就是 `2 * 10`，它被发送到第一个generator实例 `it1`，所以 `x` 得到值 `20`。`z` 将 `1` 递增至 `2`，然后 `20 * 2` 被 `yield` 出来，将 `val1` 设置为 `40`。
3. `val1 * 5` 就是 `40 * 5`，它被发送到第二个generator实例 `it2` 中，所以 `x` 得到值 `200`。`z` 又一次递增，从 `2` 到 `3`，然后 `200 * 3` 被 `yield` 出来，将 `val2` 设置为 `600`。
4. `val2 / 2` 就是 `600 / 2`，它被发送到第一个generator实例 `it1`，所以 `y` 得到值 `300`，然后分别为它的 `x y z` 值打印出 `20 300 3`。
5. `val1 / 4` 就是 `40 / 4`，它被发送到第一个generator实例 `it2`，所以 `y` 得到值 `10`，然后分别为它的 `x y z` 值打印出 `200 10 3`。

这是在你脑海中跑过的一个“有趣”的例子。你还能保持清醒？

## 穿插

回想第一章中“运行至完成”一节的这个场景：

```

var a = 1;
var b = 2;

function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}

```

使用普通的JS函数，当然要么是 `foo()` 可以首先运行完成，要么是 `bar()` 可以首先运行至完成，但是 `foo()` 不可能与 `bar()` 穿插它的独立语句。所以，前面这段代码只有两个可能的结果。

然而，使用generator，明确地穿插（甚至是在语句中间！）是可能的：

```

var a = 1;
var b = 2;

function *foo() {
    a++;
    yield;
    b = b * a;
    a = (yield b) + 3;
}

function *bar() {
    b--;
    yield;
    a = (yield 8) + b;
    b = a * (yield 2);
}

```

根据迭代器控制 `*foo()` 与 `*bar()` 分别以什么样的顺序被调用，前面这段代码可以产生几种不同的结果。换句话说，通过两个generator在同一个共享的变量上穿插，我们实际上可以展示（以一种模拟的方式）在第一章中讨论的，理论上的“线程的竞争状态”环境。

首先，让我们制造一个称为 `step(...)` 的帮助函数，让它控制迭代器：

```

function step(gen) {
  var it = gen();
  var last;

  return function() {
    // 不论`yield`出什么，只管在下一次时直接把它塞回去！
    last = it.next( last ).value;
  };
}

```

`step(..)` 初始化一个 generator 来创建它的 `it` 迭代器，然后它返回一个函数，每次这个函数被调用时，都将迭代器向前推一步。另外，前一个被 `yield` 出来的值将被直接发给下一步。所以，`yield 8` 将变成 `8` 而 `yield b` 将成为 `b`（不管它在 `yield` 时是什么值）。

现在，为了好玩儿，让我们做一些实验，来看看将这些 `*foo()` 与 `*bar()` 的不同块儿穿插时的效果。我们从一个无聊的基本情况开始，保证 `*foo()` 在 `*bar()` 之前全部完成（就像我们在第一章中做的那样）：

```

// 确保重置了`a`和`b`
a = 1;
b = 2;

var s1 = step( foo );
var s2 = step( bar );

// 首先完全运行`*foo()`
s1();
s1();
s1();

// 现在运行`*bar()`
s2();
s2();
s2();
s2();

console.log( a, b );      // 11 22

```

最终结果是 `11` 和 `22`，就像第一章的版本那样。现在让我们把顺序混合穿插，来看看它如何改变 `a` 与 `b` 的值。

```
// 确保重置了`a`和`b`
a = 1;
b = 2;

var s1 = step( foo );
var s2 = step( bar );

s2();      // b--;
s2();      // 让出 8
s1();      // a++;
s2();      // a = 8 + b;
           // 让出 2
s1();      // b = b * a;
           // 让出 b
s1();      // a = b + 3;
s2();      // b = a * 2;
```

在我告诉你结果之前，你能指出在前面的程序运行之后 `a` 和 `b` 的值是什么吗？不要作弊！

```
console.log( a, b );    // 12 18
```

注意：作为留给读者的练习，试试通过重新安排 `s1()` 和 `s2()` 调用的顺序，看看你能得到多少种结果组合。别忘了你总是需要三个 `s1()` 调用和四个 `s2()` 调用。至于为什么，回想一下刚才关于使用 `yield` 匹配 `next()` 的讨论。

当然，你几乎不会想有意制造这种水平的，令人糊涂的穿插，因为他创建了非常难理解的代码。但是这个练习很有趣，而且对于理解多个generator如何并发地运行在相同的共享作用域来说很有教育意义，因为会有一些地方这种能力十分有用。

我们会在本章末尾更详细地讨论generator并发。

## 生成值

在前一节中，我们提到了一个generator的有趣用法，作为一种生产值的方式。这不是我们本章主要关注的，但如果我们在不在这里讲一下基本我们会想念它的，特别是因为这种用法实质上是它的名称的由来：生成器。

我们将要稍稍深入一下迭代器的话题，但我们会绕回到它们如何与generator关联，并使用generator来生成值。

## 发生器与迭代器

想象你正在生产一系列的值，它们中的每一个都与前一个值有可定义的关系。为此，你将需要一个有状态的发生器来记住上一个给出的值。

你可以用函数闭包（参加本系列的作用域与闭包）来直接地实现这样的东西：

```
var gimmeSomething = (function(){
    var nextVal;

    return function(){
        if (nextVal === undefined) {
            nextVal = 1;
        }
        else {
            nextVal = (3 * nextVal) + 6;
        }

        return nextVal;
    };
})();

gimmeSomething();           // 1
gimmeSomething();           // 9
gimmeSomething();           // 33
gimmeSomething();           // 105
```

注意：这里 `nextVal` 的计算逻辑已经被简化了，但从概念上讲，直到下一次 `gimmeSomething()` 调用发生之前，我们不想计算下一个值（也就是 `nextVal`），因为一般对于持久性更强的，或者比简单的 `number` 更有限的资源的发生器来说，那可能是一种资源泄漏的设计。

生成随意的数字序列不是一个很真实的例子。但是如果你从一个数据源中生成记录呢？你可以想象很多相同的代码。

事实上，这种任务是一种非常常见的设计模式，通常用迭代器解决。一个 `迭代器` 是一个明确定义的接口，用来逐个通过一系列从发生器得到的值。迭代器的JS接口，和大多数语言一样，是在你每次想从发生器中得到下一个值时调用的 `next()`。

我们可以为我们的数字序列发生器实现标准的 `迭代器`；

```

var something = (function(){
  var nextVal;

  return {
    // `for..of` 循环需要这个
    [Symbol.iterator]: function(){ return this; },

    // 标准的迭代器接口方法
    next: function(){
      if (nextVal === undefined) {
        nextVal = 1;
      }
      else {
        nextVal = (3 * nextVal) + 6;
      }

      return { done:false, value:nextVal };
    }
  };
})();

something.next().value;          // 1
something.next().value;          // 9
something.next().value;          // 33
something.next().value;          // 105

```

注意： 我们将在“Iterables”一节中讲解为什么我们在这个代码段中需要 `[Symbol.iterator]`：

.. 这一部分。在语法上讲，两个ES6特性在发挥作用。首先，`[ .. ]` 语法称为一个 计算型属性名（参见本系列的 `this`与对象原型）。它是一种字面对象定义方法，用来指定一个表达式并使用这个表达式的结果作为属性名。另一个，`Symbol.iterator` 是ES6预定义的特殊 `Symbol` 值。

`next()` 调用返回一个对象，它带有两个属性：`done` 是一个 `boolean` 值表示 迭代器 的完成状态；`value` 持有迭代的值。

ES6还增加了 `for..of` 循环，它意味着一个标准的 迭代器 可以使用原生的循环语法来自动地被消费：

```

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

  // 不要让循环永无休止！
  if (v > 500) {
    break;
  }
}

// 1 9 33 105 321 969

```

注意：因为我们的 `something` 迭代器总是返回 `done:false`，这个 `for..of` 循环将会永远运行，这就是为什么我们条件性地放进一个 `break`。对于迭代器来说永不终结是完全没有问题的，但是也有一些情况 迭代器 将运行在有限的值的集合上，而最终返回 `done:true`。

`for..of` 循环为每一次迭代自动调用 `next()` ——他不会给 `next()` 传入任何值——而且他将会在收到一个 `done:true` 时自动终结。这对于在一个集合的数据中进行循环十分方便。

当然，你可以手动循环一个迭代器，调用 `next()` 并检查 `done:true` 条件来知道什么时候停止：

```
for (
  var ret;
  (ret = something.next()) && !ret.done;
) {
  console.log( ret.value );

  // 不要让循环永无休止！
  if (ret.value > 500) {
    break;
  }
}

// 1 9 33 105 321 969
```

注意：这种手动的 `for` 方式当然要比ES6的 `for..of` 循环语法难看，但它的好处是它提供给你一个机会，在有必要时传值给 `next(..)` 调用。

除了制造你自己的 迭代器 之外，许多JS中（就ES6来说）内建的数据结构，比如 `array`，也有默认的 迭代器：

```
var a = [1,3,5,7,9];

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

// 1 3 5 7 9
```

`for..of` 循环向 `a` 要求它的迭代器，并自动使用它迭代 `a` 的值。

注意：看起来像是一个ES6的奇怪省略，普通的 `object` 有意地不带有像 `array` 那样的默认迭代器。原因比我们要在这里讲的深刻得多。如果你想要的只是迭代一个对象的属性（不特别保证顺序），`Object.keys(..)` 返回一个 `array`，它可以像 `for (var k of Object.keys(obj)) { .. }` 这样使用。像这样用 `for..of` 循环一个对象上的键，与用 `for..in` 循环内很相似，除了在 `for..in` 中会包含 `[[Prototype]]` 链的属性，而 `Object.keys(..)` 不会（参见本系列的 `this` 与 对象原型）。

## Iterables

在我们运行的例子中的 `something` 对象被称为一个 **迭代器**，因为它的接口中有 `next()` 方法。但一个紧密关联的术语是 **iterable**，它指包含有一个可以迭代它所有值的迭代器的对象。

在ES6中，从一个 **iterable** 中取得一个 **迭代器** 的方法是，**iterable** 上必须有一个函数，它的名称是特殊的ES6符号值 `Symbol.iterator`。当这个函数被调用时，它就会返回一个 **迭代器**。虽然不是必须的，但一般来说每次调用应当返回一个全新的 **迭代器**。

前一个代码段的 `a` 就是一个 **iterable**。`for..of` 循环自动地调用它的 `Symbol.iterator` 函数来构建一个 **迭代器**。我们当然可以手动地调用这个函数，然后使用它返回的 `iterator`：

```
var a = [1, 3, 5, 7, 9];

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

it.next().value;    // 1
it.next().value;    // 3
it.next().value;    // 5
..
```

在前面定义 `something` 的代码段中，你可能已经注意到了这一行：

```
[Symbol.iterator]: function(){ return this; }
```

这段有点让人困惑的代码制造了 `something` 值——`something` 迭代器的接口——也是一个 **iterable**；现在它既是一个 **iterable** 也是一个 **迭代器**。然后，我们把 `something` 传递给 `for..of` 循环：

```
for (var v of something) {
  ..
}
```

`for..of` 循环期待 `something` 是一个 **iterable**，所以它会寻找并调用它的 `Symbol.iterator` 函数。我们将这个函数定义为简单地 `return this`，所以它将自己给出，而 `for..of` 不会知道这些。

## Generator迭代器

带着 **迭代器** 的背景知识，让我们把注意力移回generator。一个generator可以被看做一个值的发生器，我们通过一个 **迭代器** 接口的 `next()` 调用每次从中抽取一个值。

所以，一个generator本身在技术上讲并不是一个 **iterable**，虽然很相似——当你执行 generator时，你就得到一个 **迭代器**：

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

var it = foo();
```

我们可以用generator实现早前的 something 无限数字序列发生器，就像这样：

```
function *something() {
  var nextVal;

  while (true) {
    if (nextVal === undefined) {
      nextVal = 1;
    }
    else {
      nextVal = (3 * nextVal) + 6;
    }

    yield nextVal;
  }
}
```

注意：在一个真实的JS程序中含有一个 `while..true` 循环通常是一件非常不好的事情，至少如果它没有一个 `break` 或 `return` 语句，那么它就很可能永远运行，并同步地，阻塞/锁定浏览器UI。然而，在generator中，如果这样的循环含有一个 `yield`，那它就是完全没有问题的，因为generator将在每次迭代后暂停，`yield` 回主程序和/或事件轮询队列。说的明白点儿，“generator把 `while..true` 带回到JS编程中了！”

这变得相当干净和简单点儿了，对吧？因为generator会暂停在每个 `yield`，`*something()` 函数的状态（作用域）被保持着，这意味着没有必要用闭包的模板代码来跨调用保留变量的状态了。

不仅是更简单的代码——我们不必自己制造 迭代器 接口了——它实际上是更合理的代码，因为它更清晰地表达了意图。比如，`while..true` 循环告诉我们这个generator将要永远运行——只要我们一直向它请求，它就一直产生值。

现在我们可以在 `for..of` 循环中使用新得发亮的 `*something()` generator了，而且你会看到它工作起来基本一模一样：

```

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

    // 不要让循环永无休止 !
    if (v > 500) {
        break;
    }
}

// 1 9 33 105 321 969

```

不要跳过 `for (var v of something()) ...` ! 我们不仅仅像之前的例子那样将 `something` 作为一个值引用了，而是调用 `*something()` generator 来得到它的迭代器，并交给 `for..of` 使用。

如果你仔细观察，在这个generator和循环的互动中，你可能会有两个疑问：

- 为什么我们不能说 `for (var v of something) ...` ? 因为这个 `something` 是一个 generator，而不是一个 `iterable`。我们不得不调用 `something()` 来构建一个发生器给 `for..of`，以便它可以迭代。
- `something()` 调用创建一个迭代器，但是 `for..of` 想要一个 `iterable`，对吧？对，generator 的迭代器上也有一个 `Symbol.iterator` 函数，这个函数基本上就是 `return this`，就像我们刚才定义的 `something iterable`。换句话说 generator 的迭代器也是一个 `iterable` !

## 停止Generator

在前一个例子中，看起来在循环的 `break` 被调用后，`*something()` generator 的迭代器实例基本上被留在了一个永远挂起的状态。

但是这里有一个隐藏的行为为你处理这件事。`for..of` 循环的“异常完成”（“提前终结”等等）——一般是由 `break`，`return`，或未捕捉的异常导致的——会向 generator 的迭代器发送一个信号，以使它终结。

注意：技术上讲，`for..of` 循环也会在循环正常完成时向迭代器发送这个信号。对于 generator 来说，这实质上是一个无实际意义的操作，因为 generator 的迭代器要首先完成，`for..of` 循环才能完成。然而，自定义的迭代器可能会希望从 `for..of` 循环的消费者那里得到另外的信号。

虽然一个 `for..of` 循环将会自动发送这种信号，你可能会希望手动发送信号给一个迭代器；你可以通过调用 `return(..)` 来这么做。

如果你在 generator 内部指定一个 `try..finally` 从句，它将总是被执行，即便是 generator 从外部被完成。这在你需要进行资源清理时很有用（数据库连接等）：

```

function *something() {
  try {
    var nextVal;

    while (true) {
      if (nextVal === undefined) {
        nextVal = 1;
      }
      else {
        nextVal = (3 * nextVal) + 6;
      }

      yield nextVal;
    }
  }
  // 清理用的从句
  finally {
    console.log("cleaning up!");
  }
}

```

前面那个在 `for..of` 中带有 `break` 的例子将会触发 `finally` 从句。但是你可以用 `return(..)` 从外部来手动终结generator的迭代器实例：

```

var it = something();
for (var v of it) {
  console.log(v);

  // 不要让循环永无休止！
  if (v > 500) {
    console.log(
      // 使generator得迭代器完成
      it.return("Hello World").value
    );
    // 这里不需要`break`
  }
}

// 1 9 33 105 321 969
// cleaning up!
// Hello World

```

当我们调用 `it.return(..)` 时，它会立即终结generator，从而运行 `finally` 从句。而且，它会将返回的 `value` 设置为你传入 `return(..)` 的任何东西，这就是 `Hello World` 如何立即返回来的。我们现在也不必再包含一个 `break`，因为generator的迭代器会被设置为 `done:true`，所以 `for..of` 循环会在下一次迭代时终结。

generator的命名大部分源自于这种 消费生产的值 的用法。但要重申的是，这只是generator的用法之一，而且坦白的说，在这本书的背景下这甚至不是我们主要关注的。

但是现在我们更加全面地了解它们的机制是如何工作的，我们接下来可以将注意力转向 generator 如何实施于异步并发。

## 异步地迭代 Generator

generator 要怎样处理异步编码模式，解决回调和类似的问题？让我们开始回答这个问题。

我们应当重温一下第三章的一个场景。回想一下这个回调方式：

```
function foo(x, y, cb) {
  ajax(
    "http://some.url.1/?x=" + x + "&y=" + y,
    cb
  );
}

foo( 11, 31, function(err, text) {
  if (err) {
    console.error( err );
  }
  else {
    console.log( text );
  }
});
```

如果我们想用 generator 表示相同的任务流控制，我们可以：

```

function foo(x,y) {
  ajax(
    "http://some.url.1/?x=" + x + "&y=" + y,
    function(err,data){
      if (err) {
        // 向`*main()`中扔进一个错误
        it.throw( err );
      }
      else {
        // 使用收到的`data`来继续`*main()`
        it.next( data );
      }
    }
  );
}

function *main() {
  try {
    var text = yield foo( 11, 31 );
    console.log( text );
  }
  catch (err) {
    console.error( err );
  }
}

var it = main();

// 使一切开始运行！
it.next();

```

一眼看上去，这个代码段要比以前的回调代码更长，而且也许看起来更复杂。但不要让这种印象误导你。**generator**的代码段实际上要好太多了！但是这里有很多我们需要讲解的。

首先，让我们看看代码的这一部分，也是最重要的部分：

```

var text = yield foo( 11, 31 );
console.log( text );

```

花一点时间考虑一下这段代码如何工作。我们调用了一个普通的函数 `foo(..)`，而且我们显然可以从Ajax调用那里得到 `text`，即便它是异步的。

这怎么可能？如果你回忆一下第一章的最开始，我们有一个几乎完全一样的代码：

```

var data = ajax( "...url 1..." );
console.log( data );

```

但是这段代码不好用！你能发现不同吗？它就是在**generator**中使用的 `yield`。

这就是魔法发生的地方！是它允许我们拥有一个看起来是阻塞的，同步的，但实际上不会阻塞整个程序的代码；它仅仅暂停/阻塞在generator本身的代码。

在 `yield foo(11, 31)` 中，首先 `foo(11, 31)` 调用被发起，它什么也不返回（也就是 `undefined`），所以我们发起了数据请求，然后我们实际上做的是 `yield undefined`。这没问题，因为这段代码现在没有依赖 `yield` 的值来做任何有趣的事。我们在本章稍后再重新讨论这个问题。

在这里，我们没有将 `yield` 作为消息传递的工具，只是作为进行暂停/阻塞的流程控制的工具。实际上，它会传递消息，但是只是单向的，在generator被继续运行之后。

那么，generator暂停在了 `yield`，它实质上再问一个问题，“我该将什么值返回并赋给变量 `text`？”谁来回答这个问题？

看一下 `foo(..)`。如果Ajax请求成功，我们调用：

```
it.next( data );
```

这将使generator使用应答数据继续运行，这意味着我们暂停的 `yield` 表达式直接收到这个值，然后因为它重新开始以运行generator代码，所以这个值被赋给本地变量 `text`。

很酷吧？

退一步考虑一下它的意义。我们在generator内部的代码看起来完全是同步的（除了 `yield` 关键字本身），但隐藏在幕后的是，在 `foo(..)` 内部，操作可以完全是异步的。

这很伟大！这几乎完美地解决了我们前面遇到的问题：回调不能像我们的大脑可以关联的那样，以一种顺序，同步的风格表达异步处理。

实质上，我们将异步处理作为实现细节抽象出去，以至于我们可以同步地/顺序地推理我们的流程控制：“发起Ajax请求，然后在它完成之后打印应答。”当然，我们仅仅在这个流程控制中表达了两个步骤，但同样的能力可以无边界地延伸，让我们需要表达多少步骤，就表达多少。

提示：这是一个如此重要的认识，为了充分理解，现在回过头去再把最后三段读一遍！

## 同步错误处理

但是前面的generator代码会让出更多的好处给我们。让我们把注意力移到generator内部的 `try..catch` 上：

```

try {
  var text = yield foo( 11, 31 );
  console.log( text );
}
catch (err) {
  console.error( err );
}

```

这是怎么工作的？`foo(..)` 调用是异步完成的，`try..catch` 不是无法捕捉异步错误吗？就像我们在第三章中看到的？

我们已经看到了 `yield` 如何让赋值语句暂停，来等待 `foo(..)` 去完成，以至于完成的响应可以被赋予 `text`。牛X的是，`yield` 暂停还允许generator来 `catch` 一个错误。我们在前面的例子，我们用这一部分代码将这个错误抛出到generator中：

```

if (err) {
  // 向`*main()`中扔进一个错误
  it.throw( err );
}

```

generator的 `yield` 暂停特性不仅意味着我们可以从异步的函数调用那里得到看起来同步的 `return` 值，还意味着我们可以同步地捕获这些异步函数调用的错误！

那么我们看到了，我们可以将错误 抛入 generator，但是将错误 抛出一个generator呢？和你期望的一样：

```

function *main() {
  var x = yield "Hello World";

  yield x.toLowerCase();    // 引发一个异常！
}

var it = main();

it.next().value;           // Hello World

try {
  it.next( 42 );
}
catch (err) {
  console.error( err );   // TypeError
}

```

当然，我们本可以用 `throw ..` 手动地抛出一个错误，而不是制造一个异常。

我们甚至可以 `catch` 我们 `throw(...)` 进generator的同一个错误，实质上给了generator一个机会来处理它，但如果generator没处理，那么迭代器代码必须处理它：

```
function *main() {
  var x = yield "Hello World";

  // 永远不会跑到这里
  console.log( x );
}

var it = main();

it.next();

try {
  // `*main()` 会处理这个错误吗？我们走着瞧！
  it.throw( "Oops" );
}
catch (err) {
  // 不，它没处理！
  console.error( err );           // Oops
}
```

使用异步代码的，看似同步的错误处理（通过 `try..catch`）在可读性和可推理性上大获全胜。

## Generators + Promises

在我们前面的讨论中，我们展示了generator如何可以异步地迭代，这是一个用顺序的可推理性来取代混乱如面条的回调的一个巨大进步。但我们丢掉了两个非常重要的东西：Promise的可靠性和可组合性（见第三章）！

别担心——我们会把它们拿回来。在ES6的世界中最棒的就是将generator（看似同步的异步代码）与Promise（可靠性和可组合性）组合起来。

但怎么做呢？

回想一下第三章中我们基于Promise的方式运行Ajax的例子：

```

function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

foo( 11, 31 )
.then(
    function(text){
        console.log( text );
    },
    function(err){
        console.error( err );
    }
);

```

在我们早先的运行Ajax的例子的generator代码中，`foo(..)` 什么也不返回（`undefined`），而且我们的迭代器控制代码也不关心 `yield` 的值。

但这里的Promise相关的`foo(..)` 在发起Ajax调用后返回一个promise。这暗示着我们可以用`foo(..)` 构建一个promise，然后从generator中 `yield` 出来，而后 迭代器 控制代码将可以收到这个promise。

那么 迭代器 应当对promise做什么？

它应当监听promise的解析（完成或拒绝），然后要么使用完成消息继续运行generator，要么使用拒绝理由向generator抛出错误。

让我重复一遍，因为它如此重要。发挥Promise和generator的最大功效的自然方法是 `yield` 一个Promise，并将这个Promise连接到generator的 迭代器 的控制端。

让我们试一下！首先，我们将Promise相关的`foo(..)` 与generator `*main()` 放在一起：

```

function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

function *main() {
    try {
        var text = yield foo( 11, 31 );
        console.log( text );
    }
    catch (err) {
        console.error( err );
    }
}

```

在这个重构中最强大的启示是，`*main()` 内部的代码更本就没变！在generator内部，无论什么样的值被 `yield` 出去都是一个不可见的实现细节，所以我们甚至不会察觉它发生了，也不用担心它。

那么我们现在如何运行 `*main()`？我们还有一些管道的实现工作要做，接收并连接 `yield` 的 promise，使它能够根据解析来继续运行generator。我们从手动这么做开始：

```
var it = main();

var p = it.next().value;

// 等待`p` promise解析
p.then(
  function(text){
    it.next( text );
  },
  function(err){
    it.throw( err );
  }
);
```

其实，根本不费事，对吧？

这段代码应当看起来与我们早前做的很相似：手动地连接被错误优先的回调控制的 generator。与 `if (err) { it.throw(..)` 不同的是，promise已经为我们分割为完成（成功）与拒绝（失败），否则迭代器控制是完全相同的。

现在，我们已经掩盖了一些重要的细节。

最重要的是，我们利用了这样一个事实：我们知道 `*main()` 里面只有一个Promise相关的步骤。如果我们想要能用Promise驱动一个generator而不管它有多少步骤呢？我们当然不想为每一个generator手动编写一个不同的Promise链！要是有这样一种方法该多好：可以重复（也就是“循环”）迭代的控制，而且每次一有Promise出来，就在继续之前等待它的解析。

另外，如果generator在 `it.next()` 调用期间抛出一个错误怎么办？我们是该退出，还是应该 `catch` 它并把它送回去？相似地，要是我们 `it.throw(..)` 一个Promise拒绝给generator，但是没有被处理，又直接回来了呢？

## 带有Promise的Generator运行器

你在这条路上探索得越远，你就越能感到，“哇，要是有一些工具能帮我做这些就好了。”而且你绝对是对的。这是一种如此重要的模式，而且你不想把它弄错（或者因为一遍又一遍地重复它而把自己累死），所以你最好的选择是把赌注压在一个工具上，而它以我们将要描述的方式使用这种特定设计的工具来运行 `yield` Promise的generator。

有几种Promise抽象库提供了这样的工具，包括我的 `asynquence` 库和它的 `runner(..)`，我们将在本书的附录A中讨论它。

但看在学习和讲解的份儿上，让我们定义我们自己的名为 `run(..)` 的独立工具：

```
// 感谢Benjamin Gruenbaum (@benjammingr在GitHub)在此做出的巨大改进！
function run(gen) {
    var args = [].slice.call( arguments, 1), it;

    // 在当前的上下文环境中初始化generator
    it = gen.apply( this, args );

    // 为generator的完成返回一个promise
    return Promise.resolve()
        .then( function handleNext(value){
            // 运行至下一个让出的值
            var next = it.next( value );

            return (function handleResult(next){
                // generator已经完成运行了？
                if (next.done) {
                    return next.value;
                }
                // 否则继续执行
                else {
                    return Promise.resolve( next.value )
                        .then(
                            // 在成功的情况下继续异步循环，将解析的值送回generator
                            handleNext,
                            // 如果`value`是一个拒绝的promise，就将错误传播回generator自己的
                            // 错误处理
                            function handleErr(err) {
                                return Promise.resolve(
                                    it.throw( err )
                                )
                                .then( handleResult );
                            }
                        );
                }
            })(next);
        } );
}
```

如你所见，它可能比你想要自己编写的东西复杂得多，特别是你将不会想为每个你使用的 `generator` 重复这段代码。所以，一个帮助工具/库绝对是可行的。虽然，我鼓励你花几分钟时间研究一下这点代码，以便对如何管理`generator+Promise`交涉得到更好的感觉。

你如何在我们正在讨论的Ajax例子中将 `run(..)` 和 `*main()` 一起使用呢？

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

run( main );
```

就是这样！按照我们连接 `run(..)` 的方式，它将自动地，异步地推进你传入的generator，直到完成。

注意：我们定义的 `run(..)` 返回一个promise，它被连接成一旦generator完成就立即解析，或者收到一个未捕获的异常，而generator没有处理它。我们没有在这里展示这种能力，但我们会在这章稍后回到这个话题。

## ES7: `async` 和 `await` ?

前面的模式——generator让出一个Promise，然后这个Promise控制generator的迭代器向前推进至它完成——是一个如此强大和有用的方法，如果我们能不通过乱七八糟的帮助工具库（也就是 `run(..)`）来使用它就更好了。

在这方面可能有一些好消息。在写作这本书的时候，后ES6，ES7化的时间表上已经出现了草案，对这个问题提供早期但强大的附加语法支持。显然，现在还太早而不能保证其细节，但是有相当大的可能性它将蜕变为类似于下面的东西：

```
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

async function main() {
    try {
        var text = await foo( 11, 31 );
        console.log( text );
    }
    catch (err) {
        console.error( err );
    }
}

main();
```

如你所见，这里没有 `run(..)` 调用（意味着不需要工具库！）来驱动和调用 `main()` ——它仅仅像一个普通函数那样被调用。另外，`main()` 不再作为一个generator函数声明；它是一种新型的函数：`async function`。而最后，与 `yield` 一个Promise相反，我们 `await` 它解析。

如果你 `await` 一个Promise，`async function` 会自动地知道做什么——它会暂停这个函数（就像使用`generator`那样）直到Promise解析。我们没有在这个代码段中展示，但是调用一个像 `main()` 这样的异步函数将自动地返回一个promise，它会在函数完全完成时被解析。

**提示：** `async / await` 的语法应该对拥有C#经验的读者看起来非常熟悉，因为它们基本上是一样的。

这个草案实质上是为我们已经衍生出的模式进行代码化的支持，成为一种语法机制：用看似同步的流程控制代码与Promise组合。将两个世界的最好部分组合，来有效解决我们用回调遇到的几乎所有主要问题。

这样的ES7化草案已经存在，并且有了早期的支持和热忱的拥护。这一事实为这种异步模式在未来的重要性上信心满满地投了有力的一票。

## Generator中的Promise并发

至此，所有我们展示过的是一种使用Promise+generator的单步异步流程。但是现实世界的代码将总是有许多异步步骤。

如果你不小心，`generator`看似同步的风格也许会蒙蔽你，使你在如何构造你的异步并发上感到自满，导致性能次优的模式。那么我们想花一点时间来探索一下其他选项。

想象一个场景，你需要从两个不同的数据源取得数据，然后将这些应答组合来发起第三个请求，最后打印出最终的应答。我们在第三章中用Promise探索过类似的场景，但这次让我们在`generator`的环境下考虑它。

你的第一直觉可能是像这样的东西：

```
function *foo() {
  var r1 = yield request( "http://some.url.1" );
  var r2 = yield request( "http://some.url.2" );

  var r3 = yield request(
    "http://some.url.3/?v=" + r1 + "," + r2
  );

  console.log( r3 );
}

// 使用刚才定义的`run(..)`工具
run( foo );
```

这段代码可以工作，但在我们特定的这个场景中，它不是最优的。你能发现为什么吗？

因为 `r1` 和 `r2` 请求可以——而且为了性能的原因，应该——并发运行，但在这段代码中它们将顺序地运行；直到 `"http://some.url.1"` 请求完成之前，`"http://some.url.2"` URL 不会被 Ajax 取得。这两个请求是独立的，所以性能更好的方式可能是让它们同时运行。

但是使用 generator 和 `yield`，到底应该怎么做？我们知道 `yield` 在代码中只是一个单独的暂停点，所以你根本不能再同一时刻做两次暂停。

最自然和有效的答案是基于 Promise 的异步流程，特别是因为它们的时间无关的状态管理能力（参见第三章的“未来的值”）。

最简单的方式：

```
function *foo() {
  // 使两个请求“并行”
  var p1 = request( "http://some.url.1" );
  var p2 = request( "http://some.url.2" );

  // 等待两个promise都被解析
  var r1 = yield p1;
  var r2 = yield p2;

  var r3 = yield request(
    "http://some.url.3/?v=" + r1 + "," + r2
  );

  console.log( r3 );
}

// 使用刚才定义的`run(..)`工具
run( foo );
```

为什么这与前一个代码段不同？看看 `yield` 在哪里和不在哪里。`p1` 和 `p2` 是并发地（也就是“并行”）发起的 Ajax 请求 promise。它们哪一个先完成都不要紧，因为 promise 会一直保持它们的解析状态。

然后我们使用两个连续的 `yield` 语句等待并从 promise 中取得解析值（分别取到 `r1` 和 `r2` 中）。如果 `p1` 首先解析，`yield p1` 会首先继续执行然后等待 `yield p2` 继续执行。如果 `p2` 首先解析，它将会耐心地保持解析值知道被请求，但是 `yield p1` 将会首先停住，直到 `p1` 解析。

不管是哪一种情况，`p1` 和 `p2` 都将并发地运行，并且在 `r3 = yield request..` Ajax 请求发起之前，都必须完成，无论以哪种顺序。

如果这种流程控制处理模型听起来很熟悉，那是因为它基本上和我们在第三章中介绍的，因 `Promise.all([ .. ])` 工具成为可能的“门”模式是相同的。所以，我们也可以像这样表达这种流程控制：

```

function *foo() {
  // 使两个请求“并行”并等待两个promise都被解析
  var results = yield Promise.all([
    request( "http://some.url.1" ),
    request( "http://some.url.2" )
  ]);

  var r1 = results[0];
  var r2 = results[1];

  var r3 = yield request(
    "http://some.url.3/?v=" + r1 + "," + r2
  );

  console.log( r3 );
}

// 使用前面定义的`run(..)`工具
run( foo );

```

注意：就像我们在第三章中讨论的，我们甚至可以用ES6解构赋值来把 `var r1 = .. var r2 = ..` 赋值简写为 `var [r1,r2] = results`。

换句话说，在generator+Promise的方式中，Promise所有的并发能力都是可用的。所以在任何地方，如果你需要比“这个然后那个”要复杂的顺序异步流程步骤时，Promise都可能是最佳选择。

## Promises，隐藏起来

作为代码风格的警告要说一句，要小心你在你的**generator**内部包含了多少Promise逻辑。以我们描述过的方式在异步性上使用**generator**的全部意义，是要创建简单，顺序，看似同步的代码，并尽可能多地将异步性细节隐藏在这些代码之外。

比如，这可能是一种更干净的方式：

```
// 注意：这是一个普通函数，不是generator
function bar(url1,url2) {
    return Promise.all([
        request( url1 ),
        request( url2 )
    ]);
}

function *foo() {
    // 将基于Promise的并发细节隐藏在`bar(..)`内部
    var results = yield bar(
        "http://some.url.1",
        "http://some.url.2"
    );

    var r1 = results[0];
    var r2 = results[1];

    var r3 = yield request(
        "http://some.url.3/?v=" + r1 + "," + r2
    );

    console.log( r3 );
}

// 使用刚才定义的`run(..)`工具
run( foo );
```

在 `*foo()` 内部，它更干净更清晰地表达了我们要做的事情：我们要求 `bar(..)` 给我们一些 `results`，而我们将用 `yield` 等待它的发生。我们不必关心在底层一个 `Promise.all([ .. ])` 的 `Promise` 组合将被用来完成任务。

我们将异步性，特别是**Promise**，作为一种实现细节。

如果你要做一种精巧的序列流控制，那么将你的**Promise**逻辑隐藏在一个仅仅从你的 generator 中调用的函数里特别有用。举个例子：

```
function bar() {
    return Promise.all([
        baz( ... )
        .then( ... ),
        Promise.race( [ ... ] )
    ])
    .then( ... )
}
```

有时候这种逻辑是必须的，而如果你直接把它扔在你的generator内部，你就违背了大多数你使用generator的初衷。我们应当有意地将这样的细节从generator代码中抽象出去，以使它们不会搞乱更高层的任务表达。

在创建功能强与性能好的代码之上，你还应当努力使代码尽可能地容易推理论和维护。

注意：对于编程来说，抽象不总是一种健康的东西——许多时候它可能在得到简洁的同时增加复杂性。但是在这种情况下，我相信你的generator+Promise异步代码要比其他的选择健康得多。虽然有所有这些建议，你仍然要注意你的特殊情况，并为你和你的团队做出合适的决策。

## Generator 委托

在上一节中，我们展示了从generator内部调用普通函数，和它如何作为一种有用的技术来将实现细节（比如异步Promise流程）抽象出去。但是为这样的任务使用普通函数的缺陷是，它必须按照普通函数的规则行动，也就是说它不能像generator那样用 `yield` 来暂停自己。

在你身上可能发生这样的事情：你可能会试着使用我们的 `run(..)` 帮助函数，从一个 generator 中调用另一个 generator。比如：

```
function *foo() {
  var r2 = yield request( "http://some.url.2" );
  var r3 = yield request( "http://some.url.3/?v=" + r2 );

  return r3;
}

function *bar() {
  var r1 = yield request( "http://some.url.1" );

  // 通过`run(..)`“委托”到`*foo()``
  var r3 = yield run( foo );

  console.log( r3 );
}

run( bar );
```

通过再一次使用我们的 `run(..)` 工具，我们在 `*bar()` 内部运行 `*foo()`。我们利用了这样一个事实：我们早先定义的 `run(..)` 返回一个promise，这个promise在generator运行至完成时才解析（或发生错误），所以如果我们从一个 `run(..)` 调用中 `yield` 出一个promise给另一个 `run(..)`，它就会自动暂停 `*bar()` 直到 `*foo()` 完成。

但这里有一个更好的办法将 `*foo()` 调用整合进 `*bar()`，它称为 `yield` 委托。`yield` 委托的特殊语法是：`yield * __`（注意额外的 `*`）。让它在我们前面的例子中工作之前，让我们看一个更简单的场景：

```
function *foo() {
  console.log(`'*foo()'` starting);
  yield 3;
  yield 4;
  console.log(`'*foo()'` finished);
}

function *bar() {
  yield 1;
  yield 2;
  yield *foo(); // `yield`-delegation!
  yield 5;
}

var it = bar();

it.next().value; // 1
it.next().value; // 2
it.next().value; // '*foo()'` starting
                 // 3
it.next().value; // 4
it.next().value; // '*foo()'` finished
                 // 5
```

注意：在本章早前的一个注意点中，我解释了为什么我偏好 `function *foo() ..` 而不是 `function* foo() ..`，相似地，我也偏好——与关于这个话题的其他大多数文档不同——说 `yield *foo()` 而不是 `yield* foo()`。`*` 的摆放是纯粹的风格问题，而且要看你的最佳判断。但我发现保持统一风格很吸引人。

### `yield *foo()` 委托是如何工作的？

首先，正如我们看到过的那样，调用 `foo()` 创建了一个迭代器。然后，`yield *` 将（当前 `*bar()` generator 的）迭代器的控制委托/传递给这另一个 `*foo()` 迭代器。

那么，前两个 `it.next()` 调用控制着 `*bar()`，但当我们发起第三个 `it.next()` 调用时，`*foo()` 就启动了，而且这时我们控制的是 `*foo()` 而非 `*bar()`。这就是为什么它称为委托——`*bar()` 将它的迭代控制委托给 `*foo()`。

只要 `it` 迭代器的控制耗尽了整个 `*foo()` 迭代器，它就会自动地将控制返回到 `*bar()`。

那么现在回到前面的三个顺序Ajax请求的例子：

```

function *foo() {
  var r2 = yield request( "http://some.url.2" );
  var r3 = yield request( "http://some.url.3/?v=" + r2 );

  return r3;
}

function *bar() {
  var r1 = yield request( "http://some.url.1" );

  // 通过`run(..)`“委托”到`*foo()``
  var r3 = yield *foo();

  console.log( r3 );
}

run( bar );

```

这个代码段和前面使用的版本的唯一区别是，使用了 `yield *foo()` 而不是前面的 `yield run(foo)`。

注意：`yield *` 让出了迭代控制，不是generator控制；当你调用 `*foo()` generator时，你就 `yield` 委托给它的迭代器。但你实际上可以 `yield` 委托给任何迭代器；`yield *` `[1,2,3]` 将会消费默认的 `[1,2,3]` 数组值迭代器。

## 为什么委托？

`yield` 委托的目的很大程度上是为了代码组织，而且这种方式是与普通函数调用对称的。

想象两个分别提供了 `foo()` 和 `bar()` 方法的模块，其中 `bar()` 调用 `foo()`。它们俩分开的原因一般是由于为了程序将它们作为分离的程序来调用而进行的恰当组织。例如，可能会有一些情况 `foo()` 需要被独立调用，而其他地方 `bar()` 来调用 `foo()`。

由于这些完全相同的原因，将generator分开可以增强程序的可读性，可维护性，与可调试性。从这个角度讲，`yield *` 是一种快捷的语法，用来在 `*bar()` 内部手动地迭代 `*foo()` 的步骤。

如果 `*foo()` 中的步骤是异步的，这样的手动方式可能会特别复杂，这就是为什么你可能会需要那个 `run(..)` 工具来做它。正如我们已经展示的，`yield *foo()` 消灭了使用 `run(..)` 工具的子实例（比如 `run(foo)`）的需要。

## 委托消息

你可能想知道，这种 `yield` 委托在除了与 迭代器 控制一起工作以外，是如何与双向消息传递一起工作的。仔细查看下面这些通过 `yield` 委托进进出出的消息流：

```

function *foo() {
  console.log( "inside `*foo()`:", yield "B" );

  console.log( "inside `*foo()`:", yield "C" );

  return "D";
}

function *bar() {
  console.log( "inside `*bar()`:", yield "A" );

  // `yield`-委托!
  console.log( "inside `*bar()`:", yield *foo() );

  console.log( "inside `*bar()`:", yield "E" );

  return "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// inside `*bar()`: 1
// outside: B

console.log( "outside:", it.next( 2 ).value );
// inside `*foo()`: 2
// outside: C

console.log( "outside:", it.next( 3 ).value );
// inside `*foo()`: 3
// inside `*bar()`: D
// outside: E

console.log( "outside:", it.next( 4 ).value );
// inside `*bar()`: 4
// outside: F

```

特别注意一下 `it.next(3)` 调用之后的处理步骤：

1. 值 `3` 被传入（通过 `*bar` 里的 `yield` 委托）在 `*foo()` 内部等待中的 `yield "C"` 表达式。
2. 然后 `*foo()` 调用 `return "D"`，但是这个值不会一路返回到外面的 `it.next(3)` 调用。
3. 相反地，值 `"D"` 作为结果被发送到在 `*bar()` 内部等待中的 `yield *foo()` 表达式——这个 `yield` 委托表达式实质上在 `*foo()` 被耗尽之前一直被暂停着。所以 `"D"` 被送到 `*bar()` 内部来让它打印。
4. `yield "E"` 在 `*bar()` 内部被调用，而且值 `"E"` 被让出到外部作为 `it.next(3)` 调用的结果。

果。

从外部迭代器（`it`）的角度来看，在初始的generator和被委托的generator之间的控制没有任何区别。

事实上，`yield` 委托甚至不必指向另一个generator；它可以仅被指向一个非generator的，一般的 `iterable`。比如：

```
function *bar() {
  console.log( "inside `*bar()`:", yield "A" );

  // `yield`-委托至一个非generator
  console.log( "inside `*bar()`:", yield *[ "B", "C", "D" ] );

  console.log( "inside `*bar()`:", yield "E" );

  return "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// inside `*bar()`: 1
// outside: B

console.log( "outside:", it.next( 2 ).value );
// outside: C

console.log( "outside:", it.next( 3 ).value );
// outside: D

console.log( "outside:", it.next( 4 ).value );
// inside `*bar()`: undefined
// outside: E

console.log( "outside:", it.next( 5 ).value );
// inside `*bar()`: 5
// outside: F
```

注意这个例子与前一个之间，被接收/报告的消息的不同之处。

最惊人的是，默认的 `array` 迭代器不关心任何通过 `next(..)` 调用被发送的消息，所以值 `2`，`3`，与 `4` 实质上被忽略了。另外，因为这个迭代器没有明确的 `return` 值（不像前面使用的 `*foo()`），所以 `yield *` 表达式在它完成时得到一个 `undefined`。

## 异常也委托！

与 `yield` 委托在两个方向上透明地传递消息的方式相同，错误/异常也在双向传递：

```

function *foo() {
  try {
    yield "B";
  }
  catch (err) {
    console.log( "error caught inside `*foo()`:", err );
  }

  yield "C";

  throw "D";
}

function *bar() {
  yield "A";

  try {
    yield *foo();
  }
  catch (err) {
    console.log( "error caught inside `*bar()`:", err );
  }

  yield "E";

  yield *baz();

  // note: can't get here!
  yield "G";
}

function *baz() {
  throw "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// outside: B

console.log( "outside:", it.throw( 2 ).value );
// error caught inside `*foo()`: 2
// outside: C

console.log( "outside:", it.next( 3 ).value );
// error caught inside `*bar()`: D
// outside: E

```

```

try {
    console.log( "outside:", it.next( 4 ).value );
}
catch (err) {
    console.log( "error caught outside:", err );
}
// error caught outside: F

```

在这段代码中有一些事情要注意：

1. 但我们调用 `it.throw(2)` 时，它发送一个错误消息 `2` 到 `*bar()`，而 `*bar()` 将它委托至 `*foo()`，然后 `*foo()` 来 `catch` 它并平静地处理。之后，`yield "C"` 把 `"C"` 作为返回的 `value` 发送回 `it.throw(2)` 调用。
2. 接下来值 `"D"` 被从 `*foo()` 内部 `throw` 出来并传播到 `*bar()`，`*bar()` 会 `catch` 它并平静地处理。然后 `yield "E"` 把 `"E"` 作为返回的 `value` 发送回 `it.next(3)` 调用。
3. 接下来，一个异常从 `*baz()` 中 `throw` 出来，而没有被 `*bar()` 捕获——我们没在外面 `catch` 它——所以 `*baz()` 和 `*bar()` 都被设置为完成状态。这段代码结束后，即便有后续的 `next(..)` 调用，你也不会得到值 `"G"`——它们的 `value` 将返回 `undefined`。

## 异步委托

最后让我们回到早先的多个顺序Ajax请求的例子，使用 `yield` 委托：

```

function *foo() {
    var r2 = yield request( "http://some.url.2" );
    var r3 = yield request( "http://some.url.3/?v=" + r2 );

    return r3;
}

function *bar() {
    var r1 = yield request( "http://some.url.1" );

    var r3 = yield *foo();

    console.log( r3 );
}

run( bar );

```

在 `*bar()` 内部，与调用 `yield run(foo)` 不同的是，我们调用 `yield *foo()` 就可以了。

在前一个版本的这个例子中，Promise机制（通过 `run(..)` 控制的）被用于将值从 `*foo()` 中的 `return r3` 传递到 `*bar()` 内部的本地变量 `r3`。现在，这个值通过 `yield *` 机制直接返回。

除此以外，它们的行为是一样的。

## “递归”委托

当然，`yield` 委托可以一直持续委托下去，你想连接多少步骤就连接多少。你甚至可以在具有异步能力的`generator`上“递归”使用`yield` 委托——一个`yield` 委托至自己的`generator`：

```
function *foo(val) {
  if (val > 1) {
    // 递归委托
    val = yield *foo( val - 1 );
  }

  return yield request( "http://some.url/?v=" + val );
}

function *bar() {
  var r1 = yield *foo( 3 );
  console.log( r1 );
}

run( bar );
```

注意：我们的`run(..)`工具本可以用`run( foo, 3 )`来调用，因为它支持用额外传递的参数来进行`generator`的初始化。然而，为了在这里高调展示`yield *`的灵活性，我们使用了无参数的`*bar()`。

这段代码之后的处理步骤是什么？坚持住，它的细节要描述起来可是十分错综复杂：

1. `run(bar)` 启动了`*bar()` generator。
2. `foo(3)` 为`*foo(..)` 创建了迭代器并传递`3`作为它的`val`参数。
3. 因为`3 > 1`，`foo(2)` 创建了另一个迭代器并传递`2`作为它的`val`参数。
4. 因为`2 > 1`，`foo(1)` 又创建了另一个迭代器并传递`1`作为它的`val`参数。
5. `1 > 1` 是`false`，所以我们接下来用值`1`调用`request(..)`，并得到一个代表第一个 Ajax 调用的`promise`。
6. 这个`promise`被`yield`出来，回到`*foo(2)` generator 实例。
7. `yield *` 将这个`promise`传出并回到`*foo(3)` 生成`generator`。另一个`yield *`把这个`promise`传出到`*bar()` generator 实例。而又有另一个`yield *`把这个`promise`传出到`run(..)` 工具，而它将会等待这个`promise`（第一个 Ajax 请求）再处理。
8. 当这个`promise`解析时，它的完成消息会被发送以继续`*bar()`，`*bar()` 通过`yield *`把消息传递进`*foo(3)` 实例，`*foo(3)` 实例通过`yield *`把消息传递进`*foo(2)` generator 实例，`*foo(2)` 实例通过`yield *`把消息传给那个在`*foo(3)` generator 实例中等待的一般的`yield`。
9. 这第一个 Ajax 调用的应答现在立即从`*foo(3)` generator 实例中被`return`，作

为 `*foo(2)` 实例中 `yield *` 表达式的结果发送回来，并赋值给本地 `val` 变量。

10. `*foo(2)` 内部，第二个Ajax请求用 `request(..)` 发起，它的promise被 `yield` 回到 `*foo(1)` 实例，然后一路 `yield *` 传播到 `run(..)`（回到第7步）。当promise解析时，第二个Ajax应答一路传播回到 `*foo(2)` generator实例，并赋值到他本地的 `val` 变量。
11. 最终，第三个Ajax请求用 `request(..)` 发起，它的promise走出到 `run(..)`，然后它的解析值一路返回，最后被 `return` 到在 `*bar()` 中等待的 `yield *` 表达式。

天！许多疯狂的头脑杂技，对吧？你可能想要把它通读几遍，然后抓点儿零食放松一下大脑！

## Generator并发

正如我们在第一章和本章早先讨论过的，另个同时运行的“进程”可以协作地穿插它们的操作，而且许多时候这可以产生非常强大的异步表达式。

坦白地说，我们前面关于多个generator并发穿插的例子，展示了这真的容易让人糊涂。但我们也受到了启发，有些地方这种能力十分有用。

回想我们在第一章中看过的场景，两个不同但同时的Ajax应答处理需要互相协调，来确保数据通信不是竞合状态。我们这样把应答分别放在 `res` 数组的不同位置中：

```
function response(data) {
  if (data.url == "http://some.url.1") {
    res[0] = data;
  }
  else if (data.url == "http://some.url.2") {
    res[1] = data;
  }
}
```

但是我们如何在这种场景下使用多generator呢？

```
// `request(..)` 是一个基于Promise的Ajax工具

var res = [];

function *reqData(url) {
  res.push(
    yield request( url )
  );
}
```

注意：我们将在这里使用两个 `*reqData(..)` generator 的实例，但是这和分别使用两个不同 `generator` 的一个实例没有区别；这两种方式在道理上完全一样的。我们过一会儿就会看到两个 `generator` 的协调操作。

与不得不将 `res[0]` 和 `res[1]` 赋值手动排序不同，我们将使用协调过的顺序，让 `res.push(..)` 以可预见的顺序恰当地将值放在预期的位置。如此被表达的逻辑会让人感觉更干净。

但是我们将如何实际安排这种互动呢？首先，让我们手动实现它：

```
var it1 = reqData( "http://some.url.1" );
var it2 = reqData( "http://some.url.2" );

var p1 = it1.next().value;
var p2 = it2.next().value;

p1
.then( function(data){
  it1.next( data );
  return p2;
} )
.then( function(data){
  it2.next( data );
} );
```

`*reqData(..)` 的两个实例都开始发起它们的 Ajax 请求，然后用 `yield` 暂停。之后我们再 `p1` 解析时继续运行第一个实例，而后来的 `p2` 的解析将会重启第二个实例。以这种方式，我们使用 `Promise` 的安排来确保 `res[0]` 将持有第一个应答，而 `res[1]` 持有第二个应答。

但坦白地说，这是可怕的手动，而且它没有真正让 `generator` 组织它们自己，而那才是真正的力量。让我们用不同的方法试一下：

```
// `request(..)` 是一个基于Promise的Ajax工具

var res = [];

function *reqData(url) {
  var data = yield request( url );

  // 传递控制权
  yield;

  res.push( data );
}

var it1 = reqData( "http://some.url.1" );
var it2 = reqData( "http://some.url.2" );

var p1 = it1.next().value;
var p2 = it2.next().value;

p1.then( function(data){
  it1.next( data );
} );

p2.then( function(data){
  it2.next( data );
} );

Promise.all( [p1,p2] )
.then( function(){
  it1.next();
  it2.next();
} );
}
```

好的，这看起来好些了（虽然仍然是手动），因为现在两个 `*reqData(..)` 的实例真正地并发运行了，而且（至少是在第一部分）是独立的。

在前一个代码段中，第二个实例在第一个实例完全完成之前没有给出它的数据。但是这里，只要它们的应答一返回这两个实例就立即分别收到他们的数据，然后每个实例调用另一个 `yield` 来传送控制。最后我们在 `Promise.all([ .. ])` 的处理器中选择用什么样的顺序继续它们。

可能不太明显的是，这种方式因其对称性启发了一种可复用工具的简单形式。让我们想象使用一个称为 `runAll(..)` 的工具：

```
// `request(..)` 是一个基于Promise的Ajax工具

var res = [];

runAll(
  function*(){
    var p1 = request( "http://some.url.1" );

    // 传递控制权
    yield;

    res.push( yield p1 );
  },
  function*(){
    var p2 = request( "http://some.url.2" );

    // 传递控制权
    yield;

    res.push( yield p2 );
  }
);

```

注意：我们没有包含 `runAll(..)` 的实现代码，不仅因为它长得无法行文，也因为它是一个我们已经在先前的 `run(..)` 中实现的逻辑的扩展。所以，作为留给读者的一个很好的补充性练习，请你自己动手改进 `run(..)` 的代码，来使它像想象中的 `runAll(..)` 那样工作。另外，我的 `asynquence` 库提供了一个前面提到过的 `runner(..)` 工具，它内建了这种能力，我们将在本书的附录A中讨论它。

这是 `runAll(..)` 内部的处理将如何操作：

1. 第一个generator得到一个代表从 `"http://some.url.1"` 来的Ajax应答，然后将控制权 `yield` 回到 `runAll(..)` 工具。
2. 第二个generator运行，并对 `"http://some.url.2"` 做相同的事，将控制权 `yield` 回到 `runAll(..)` 工具。
3. 第一个generator继续，然后 `yield` 出他的promise `p1`。在这种情况下 `runAll(..)` 工具和我们前面的 `run(..)` 做同样的事，它等待promise解析，然后继续这同一个generator（没有控制传递！）。当 `p1` 解析时，`runAll(..)` 使用解析值再一次继续第一个generator，而后 `res[0]` 得到它的值。在第一个generator完成之后，有一个隐式的控制权传递。
4. 第二个generator继续，`yield` 出它的promise `p2`，并等待它的解析。一旦 `p2` 解析，`runAll(..)` 使用这个解析值继续第二个generator，于是 `res[1]` 被设置。

在这个例子中，我们使用了一个称为 `res` 的外部变量来保存两个不同的Ajax应答的结果——这是我们的并发协调。

但是这样做可能十分有帮助：进一步扩展 `runAll(..)` 使它为多个generator实例提供 分享的 内部的变量作用域，比如一个我们将在下面称为 `data` 的空对象。另外，它可以接收 被 `yield` 的非Promise值，并把它们交给下一个generator。

考虑这段代码：

```
// `request(..)` 是一个基于Promise的Ajax工具

runAll(
  function*(data){
    data.res = [];

    // 传递控制权 (并传递消息)
    var url1 = yield "http://some.url.2";

    var p1 = request( url1 ); // "http://some.url.1"

    // 传递控制权
    yield;

    data.res.push( yield p1 );
  },
  function*(data){
    // 传递控制权 (并传递消息)
    var url2 = yield "http://some.url.1";

    var p2 = request( url2 ); // "http://some.url.2"

    // 传递控制权
    yield;

    data.res.push( yield p2 );
  }
);
```

在这个公式中，两个generator不仅协调控制传递，实际上还互相通信：通过 `data.res`，和交换 `url1` 与 `url2` 的值的 `yield` 消息。这强大到不可思议！

这样的认识也是一种更为精巧的称为CSP（Communicating Sequential Processes——通信顺序处理）的异步技术的概念基础，我们将在本书的附录B中讨论它。

## Thunks

至此，我们都假定从一个generator中 `yield` 一个Promise——让这个Promise使用 像 `run(..)` 这样的帮助工具来推进generator——是管理使用generator的异步处理的最佳方法。明白地说，它是的。

但是我们跳过了一个被轻度广泛使用的模式，为了完整性我们将简单地看一看它。

在一般的计算机科学中，有一种老旧的前JS时代的概念，称为“thunk”。我们不在这里赘述它的历史，一个狭隘的表达是，thunk是一个JS函数——没有任何参数——它连接并调用另一个函数。

换句话讲，你用一个函数定义包装函数调用——带着它需要的所有参数——来推迟这个调用的执行，而这个包装用的函数就是thunk。当你稍后执行thunk时，你最终会调用那个原始的函数。

举个例子：

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

function fooThunk() {
    return foo( 3, 4 );
}

// 稍后

console.log( fooThunk() ); // 7
```

所以，一个同步的thunk是十分直白的。但是一个异步的thunk呢？我们实质上可以扩展这个狭隘的thunk定义，让它接收一个回调。

考虑这段代码：

```
function foo(x, y, cb) {
    setTimeout( function(){
        cb( x + y );
    }, 1000 );
}

function fooThunk(cb) {
    foo( 3, 4, cb );
}

// 稍后

fooThunk( function(sum){
    console.log( sum ); // 7
} );
```

如你所见，`fooThunk(..)` 仅需要一个 `cb(..)` 参数，因为它已经预先制定了值 `3` 和 `4`（分别为 `x` 和 `y`）并准备传递给 `foo(..)`。一个thunk只是在外面耐心地等待着它开始工作所需的最后一部分信息：回调。

但是你不会想要手动制造thunk。那么，让我们发明一个工具来为我们进行这种包装。

考虑这段代码：

```
function thunkify(fn) {
  var args = [].slice.call( arguments, 1 );
  return function(cb) {
    args.push( cb );
    return fn.apply( null, args );
  };
}

var fooThunk = thunkify( foo, 3, 4 );

// 稍后

fooThunk( function(sum) {
  console.log( sum );           // 7
} );
```

提示：这里我们假定原始的（`foo(..)`）函数签名希望它的回调的位置在最后，而其它的参数在这之前。这是一个异步JS函数的相当普遍的“标准”。你可以称它为“回调后置风格”。如果因为某些原因你需要处理“回调优先风格”的签名，你只需要制造一个使用`args.unshift(..)`而非`args.push(..)`的工具。

前面的`thunkify(..)`公式接收`foo(..)`函数的引用，和任何它所需的参数，并返回thunk本身（`fooThunk(..)`）。然而，这并不是你将在JS中发现的thunk的典型表达方式。

与`thunkify(..)`制造thunk本身相反，典型的——可能有点儿让人困惑的——`thunkify(..)`工具将产生一个制造thunk的函数。

额...是的。

考虑这段代码：

```
function thunkify(fn) {
  return function() {
    var args = [].slice.call( arguments );
    return function(cb) {
      args.push( cb );
      return fn.apply( null, args );
    };
  };
}
```

这里主要的不同之处是有一个额外的`return function() { .. }`。这是它在用法上的不同：

```

var whatIsThis = thunkify( foo );

var fooThunk = whatIsThis( 3, 4 );

// 稍后

fooThunk( function(sum) {
    console.log( sum );           // 7
} );

```

明显地，这段代码隐含的最大的问题是，`whatIsThis` 叫什么合适？它不是`thunk`，它是一个从 `foo(..)` 调用生产`thunk`的东西。它是一种“`thunk`”的“工厂”。而且看起来没有任何标准的意见来命名这种东西。

所以，我的提议是“`thunkory`”（“`thunk`” + “`factory`”）。于是，`thunkify(..)` 制造了一个 `thunkory`，而一个 `thunkory` 制造 `thunks`。这个道理与第三章中我的“`promisory`”提议是对称的：

```

var fooThunkory = thunkify( foo );

var fooThunk1 = fooThunkory( 3, 4 );
var fooThunk2 = fooThunkory( 5, 6 );

// 稍后

fooThunk1( function(sum) {
    console.log( sum );           // 7
} );

fooThunk2( function(sum) {
    console.log( sum );           // 11
} );

```

注意：这个例子中的 `foo(..)` 期望的回调不是“错误优先风格”。当然，“错误优先风格”更常见。如果 `foo(..)` 有某种合理的错误发生机制，我们可以改变而使它期望并使用一个错误优先的回调。后续的 `thunkify(..)` 不会关心回调被预想成什么样。用法的唯一区别是 `fooThunk1(function(err,sum){...})`。

暴露出`thunkory`方法——而不是像早先的 `thunkify(..)` 那样将中间步骤隐藏起来——可能看起来像是没必要的混乱。但是一般来讲，在你的程序一开始就制造一些`thunkory`来包装既存 API 的方法是十分有用的，然后你就可以在你需要`thunk`的时候传递并调用这些`thunkory`。这两个区别开的步骤保证了功能上更干净的分离。

来展示一下的话：

```
// 更干净：
var fooThunkory = thunkify( foo );

var fooThunk1 = fooThunkory( 3, 4 );
var fooThunk2 = fooThunkory( 5, 6 );

// 而这个不干净：
var fooThunk1 = thunkify( foo, 3, 4 );
var fooThunk2 = thunkify( foo, 5, 6 );
```

不管你是否愿意明确对付，( `fooThunk1(..)` 和 `fooThunk2(..)` ) 的用法还是一样的。

## s/promise/thunk/

那么所有这些的东西与generator有什么关系？

一般性地比较一下和：它们是不能直接互换的，因为它们在行为上不是等价的。比起单纯的，Promise可用性更广泛，而且更可靠。

但从另一种意义上讲，它们都可以被看作是对一个值的请求，这个请求可能被异步地应答。

回忆第三章，我们定义了一个工具来promise化一个函数，我们称之为 `Promise.wrap(..)` —— 我们本来也可以叫它 `promisify(..)` 的！这个Promise化包装工具不会生产Promise；它生产那些继而生产Promise的promisories。这和我们当前讨论的和是完全对称的。

为了描绘这种对称性，让我们首先将 `foo(..)` 的例子改为假定一个“错误优先风格”回调的形式：

```
function foo(x, y, cb) {
  setTimeout( function(){
    // 假定 `cb(..)` 是“错误优先风格”
    cb( null, x + y );
  }, 1000 );
}
```

现在，我们将比较 `thunkify(..)` 和 `promisify(..)` (也就是第三章的 `Promise.wrap(..)` )：

```
// 对称的：构建问题的回答者
var fooThunkory = thunkify( foo );
var fooPromisory = promisify( foo );

// 对称的：提出问题
var fooThunk = fooThunkory( 3, 4 );
var fooPromise = fooPromisory( 3, 4 );

// 取得 thunk 的回答
fooThunk( function(err,sum){
  if (err) {
    console.error( err );
  }
  else {
    console.log( sum );           // 7
  }
} );

// 取得 promise 的回答
fooPromise
.then(
  function(sum){
    console.log( sum );           // 7
  },
  function(err){
    console.error( err );
  }
);

```

thunkory和promisory实质上都是在问一个问题（一个值），thunk的 fooThunk 和promise 的 fooPromise 分别代表这个问题的未来的答案。这样看来，对称性就清楚了。

带着这个视角，我们可以看到为了异步而 `yield Promise` 的generator，也可以为异步而 `yield thunk`。我们需要的只是一个更聪明的 `run(...)` 工具（就像以前一样），它不仅可以寻找并连接一个被 `yield` 的Promise，而且可以给一个被 `yield` 的thunk提供回调。

考慮这段代码：

```
function *foo() {
  var val = yield request( "http://some.url.1" );
  console.log( val );
}

run( foo );

```

在这个例子中，`request(..)` 既可以是一个返回一个promise的promisory，也可以是一个返回一个thunk的thunkory。从generator的内部代码逻辑的角度看，我们不关心这个实现细节，这就它强大的地方！

所以，`request(..)` 可以使以下任何一种形式：

```
// promisory `request(..)` (见第三章)
var request = Promise.wrap( ajax );

// vs.

// thunkory `request(..)`
var request = thunkify( ajax );
```

最后，作为一个让我们早先的 `run(..)` 工具支持thunk的补丁，我们可能会需要这样的逻辑：

```
// ...
// 我们收到了一个回调吗？
else if (typeof next.value == "function") {
    return new Promise( function(resolve,reject){
        // 使用一个错误优先回调调用thunk
        next.value( function(err,msg) {
            if (err) {
                reject( err );
            }
            else {
                resolve( msg );
            }
        } );
    } )
    .then(
        handleNext,
        function handleErr(err) {
            return Promise.resolve(
                it.throw( err )
            )
            .then( handleResult );
        }
    );
}
```

现在，我们generator既可以调用promisory来 `yield Promise`，也可以调用thunkory来 `yield thunk`，而不论那种情况，`run(..)` 都将处理这个值并等待它的完成，以继续generator。

在对称性上，这两个方式是看起来相同的。然而，我们应当指出这仅仅从Promise或thunk表示延续generator的未来值的角度讲是成立的。

从更高的角度讲，与Promise被设计成的那样不同，thunk没有提供，它们本身也几乎没有任  
何可靠性和可组合性的保证。在这种特定的generator异步模式下使用一个thunk作为Promise的替代品是可以工作的，但与Promise提供的所有好处相比，这应当被看做是一种次理想的方法。

如果你有选择，那就偏向 `yield pr` 而非 `yield th`。但是使 `run(..)` 工具可以处理两种类型的值本身没有什么问题。

注意：在我们将要在附录A中讨论的，我的 `asynquence` 库中的 `runner(..)` 工具，可以处理 `yield` 的Promise，thunk和 `asynquence` 序列。

## 前ES6时代的Generator

我希望你已经被说服了，`generator`是一个异步编程工具箱里的非常重要的增强工具。但它是ES6中的新语法，这意味着你不能像填补Promise（它只是新的API）那样填补`generator`。那么如果我们不能奢望忽略前ES6时代的浏览器，我们该如何将`generator`带到浏览器中呢？

对所有ES6中的新语法的扩展，有一些工具——称呼他们最常见的名词是转译器（`transpilers`），也就是转换编译器（`trans-compilers`）——它们会拿起你的ES6语法，并转换为前ES6时代的等价代码（但是明显地变难看了！）。所以，`generator`可以被转译为具有相同行为但可以在ES5或以下版本进行工作的代码。

但是是怎么做到的？`yield`的“魔法”听起来不像是那么容易转译的。在我们早先的基于闭包的迭代器例子中，实际上提示了一种解决方法。

## 手动变形

在我们讨论转译器之前，让我们延伸一下，在`generator`的情况下如何手动转译。这不仅是一个学院派的练习，因为这样做实际上可以帮助我们进一步理解它们如何工作。

考虑这段代码：

```
// `request(..)` 是一个支持Promise的Ajax工具

function *foo(url) {
  try {
    console.log("requesting:", url);
    var val = yield request(url);
    console.log(val);
  }
  catch (err) {
    console.log("Oops:", err);
    return false;
  }
}

var it = foo("http://some.url.1");
```

第一个要注意的事情是，我们仍然需要一个可以被调用的普通的 `foo()` 函数，而且它仍然需要返回一个 迭代器。那么让我们来画出非`generator`的变形草图：

```

function foo(url) {
    // ...
    // 制造并返回 iterator
    return {
        next: function(v) {
            // ...
        },
        throw: function(e) {
            // ...
        }
    };
}

var it = foo( "http://some.url.1" );

```

下一个需要注意的地方是，generator通过挂起它的作用域/状态来施展它的“魔法”，但我们可以用函数闭包来模拟。为了理解如何写出这样的代码，我们将先用状态值注释generator不同的部分：

```

// `request(..)` 是一个支持Promise的Ajax工具

function *foo(url) {
    // 状态 *1*
    try {
        console.log( "requesting:", url );
        var TMP1 = request( url );

        // 状态 *2*
        var val = yield TMP1;
        console.log( val );
    }
    catch (err) {
        // 状态 *3*
        console.log( "Oops:", err );
        return false;
    }
}

```

注意：为了更准确地讲解，我们使用 `TMP1` 变量将 `val = yield request..` 语句分割为两部分。`request(..)` 发生在状态 `*1*`，而将完成值赋给 `val` 发生在状态 `*2*`。在我们将代码转换为非generator的等价物后，我们就可以摆脱中间的 `TMP1`。

换句话说，`*1*` 是初始状态，`*2*` 是 `request(..)` 成功的状态，`*3*` 是 `request(..)` 失败的状态。你可能会想象额外的 `yield` 步骤将如何编码为额外的状态。

回到我们被转译的generator，让我们在这个闭包中定义一个变量 `state`，用它来追踪状态：

```
function foo(url) {
  // 管理 generator 状态
  var state;

  // ...
}
```

现在，让我们在闭包内部定义一个称为 `process(..)` 的内部函数，它用 `switch` 语句来处理各种状态。

```
// `request(..)` 是一个支持Promise的Ajax工具

function foo(url) {
  // 管理 generator 状态
  var state;

  // generator- 范围的变量声明
  var val;

  function process(v) {
    switch (state) {
      case 1:
        console.log("requesting:", url);
        return request(url);
      case 2:
        val = v;
        console.log(val);
        return;
      case 3:
        var err = v;
        console.log("Oops:", err);
        return false;
    }
  }
}

// ...
}
```

在我们的generator中每种状态都在 `switch` 语句中有它自己的 `case`。每当我们需要处理一个新状态时，`process(..)` 就会被调用。我们一会就回来讨论它如何工作。

对任何generator范围的变量声明（`val`），我们将它们移动到 `process(..)` 外面的 `var` 声明中，这样它们就可以在 `process(..)` 的多次调用中存活下来。但是“块儿作用域”的 `err` 变量仅在 \*3\* 状态下需要，所以我们将它留在原处。

在状态 \*1\*，与 `yield request(..)` 相反，我们 `return request(..)`。在终结状态 \*2\*，没有明确的 `return`，所以我们仅仅 `return;` 也就是 `return undefined`。在终结状态 \*3\*，有一个 `return false`，我们保留它。

现在我们需要定义迭代器函数的代码，以便人们恰当地调用 `process(..)`：

```

function foo(url) {
    // 管理 generator 状态
    var state;

    // generator- 范围的变量声明
    var val;

    function process(v) {
        switch (state) {
            case 1:
                console.log("requesting:", url);
                return request(url);
            case 2:
                val = v;
                console.log(val);
                return;
            case 3:
                var err = v;
                console.log("Oops:", err);
                return false;
        }
    }

    // 制造并返回 iterator
    return {
        next: function(v) {
            // 初始状态
            if (!state) {
                state = 1;
                return {
                    done: false,
                    value: process()
                };
            }
            // 成功地让出继续值
            else if (state == 1) {
                state = 2;
                return {
                    done: true,
                    value: process(v)
                };
            }
            // generator 已经完成了
            else {
                return {
                    done: true,

```

```

        value: undefined
    );
}
},
"throw": function(e) {
    // 在状态 *1* 中，有唯一明确的错误处理
    if (state == 1) {
        state = 3;
        return {
            done: true,
            value: process( e )
        };
    }
    // 否则，是一个不会被处理的错误，所以我们仅仅把它扔回去
    else {
        throw e;
    }
}
);
}
}

```

这段代码如何工作？

1. 第一个对迭代器的 `next()` 调用将把 generator 从未初始化的状态移动到状态 `1`，然后调用 `process()` 来处理这个状态。`request(..)` 的返回值是一个代表 Ajax 应答的 `promise`，它作为 `value` 属性从 `next()` 调用被返回。
2. 如果 Ajax 请求成功，第二个 `next(..)` 调用应当送进 Ajax 的应答值，它将我们的状态移动到 `2`。`process(..)` 再次被调用（这次它被传入 Ajax 应答的值），而从 `next(..)` 返回的 `value` 属性将是 `undefined`。
3. 然而，如果 Ajax 请求失败，应当用错误调用 `throw(..)`，它将状态从 `1` 移动到 `3`（而不是 `2`）。`process(..)` 再一次被调用，这次被传入了错误的值。这个 `case` 返回 `false`，所以 `false` 作为 `throw(..)` 调用返回的 `value` 属性。

从外面看——也就是仅仅与迭代器互动——这个普通的 `foo(..)` 函数与 `*foo(..)` generator 的工作方式是一样的。所以我们有效地将 ES6 generator“转译”为前 ES6 可兼容的！

然后我们就可以手动初始化我们的 generator 并控制它的迭代器——调用 `var it = foo(..)` 和 `it.next(..)` 等等——或更好地，我们可以将它传递给我们先前定义的 `run(..)` 工具，比如 `run(foo, "..")`。

## 自动转译

前面的练习——手动编写从 ES6 generator 到前 ES6 的等价物的变形过程——教会了我们 generator 在概念上是如何工作的。但是这种变形真的是错综复杂，而且不能很好地移植到我们代码中的其他 generator 上。手动做这些工作是不切实际的，而且将会把 generator 的好处完全抵消掉。

但幸运的是，已经存在几种工具可以自动地将ES6 generator转换为我们在前一节延伸出的东西。它们不仅帮我们做力气活儿，还可以处理几种我们敷衍而过的情况。

一个这样的工具是regenerator (<https://facebook.github.io/regenerator/>)，由Facebook的聪明伙伴们开发的。

如果我们用regenerator来转译我们前面的generator，这就是产生的代码（在编写本文时）：

```
// `request(..)` 是一个支持Promise的Ajax工具

var foo = regeneratorRuntime.mark(function foo(url) {
  var val;

  return regeneratorRuntime.wrap(function foo$(context$1$0) {
    while (1) switch (context$1$0.prev = context$1$0.next) {
      case 0:
        context$1$0.prev = 0;
        console.log("requesting:", url);
        context$1$0.next = 4;
        return request(url);
      case 4:
        val = context$1$0.sent;
        console.log(val);
        context$1$0.next = 12;
        break;
      case 8:
        context$1$0.prev = 8;
        context$1$0.t0 = context$1$0.catch(0);
        console.log("Oops:", context$1$0.t0);
        return context$1$0.abrupt("return", false);
      case 12:
      case "end":
        return context$1$0.stop();
    }
  }, foo, this, [[0, 8]]);
});
```

这和我们的手动推导有明显的相似性，比如 `switch / case` 语句，而且我们甚至可以看到，`val` 被拉到了闭包外面，正如我们做的那样。

当然，一个代价是这个generator的转译需要一个帮助工具库 `regeneratorRuntime`，它持有全部管理一个普通generator/迭代器所需的可复用逻辑。它的许多模板代码看起来和我们的版本不同，但即便如此，概念还是可以看到的，比如使用 `context$1$0.next = 4` 追踪generator的下一个状态。

主要的结论是，generator不仅限于ES6+的环境中才有用。一旦你理解了它的概念，你可以在你的所有代码中利用他们，并使用工具将代码变形为旧环境兼容的。

这比使用 `Promise API` 的填补来实现前ES6的Promise要做更多的工作，但是努力完全是值得的，因为对于以一种可推理的，合理的，看似同步的顺序风格来表达异步流程控制来说，generator实在是好太多了。

一旦你适应了generator，你将永远不会回到面条般的回调地狱了！

## 复习

generator是一种ES6的新函数类型，它不像普通函数那样运行至完成。相反，generator可以暂停在一种中间完成状态（完整地保留它的状态），而且它可以从暂停的地方重新开始。

这种暂停/继续的互换是一种协作而非抢占，这意味着generator拥有的唯一能力是使用 `yield` 关键字暂停它自己，而且控制这个generator的迭代器拥有的唯一能力是继续这个generator（通过 `next(..)`）。

`yield / next(..)` 的对偶不仅是一种控制机制，它实际上是一种双向消息传递机制。一个 `yield ..` 表达式实质上为了等待一个值而暂停，而下一个 `next(..)` 调用将把值（或隐含的 `undefined`）传递回这个暂停的 `yield` 表达式。

与异步流程控制关联的generator的主要好处是，在一个generator内部的代码以一种自然的同步/顺序风格表达一个任务的各个步骤的序列。这其中的技巧是我们实质上将潜在的异步处理隐藏在 `yield` 关键字的后面——将异步处理移动到控制generator的迭代器代码中。

换句话说，generator为异步代码保留了顺序的，同步的，阻塞的代码模式，这允许我们的大脑更自然地推理代码，解决了基于回调的异步产生的两个关键问题中的一个。

# 你不懂JS: 异步与性能

## 第五章: 程序性能

这本书至此一直是关于如何更有效地利用异步模式。但是我们还没有直接解释为什么异步对于JS如此重要。最明显明确的理由就是性能。

举个例子，如果你要发起两个Ajax请求，而且他们是相互独立的，但你在进行下一个任务之前需要等到他们全部完成，你就有两种选择来对这种互动建立模型：顺序和并发。

你可以发起第一个请求并等到它完成再发起第二个请求。或者，就像我们在promise和generator中看到的那样，你可以“并列地”发起两个请求，并在继续下一步之前让一个“门”等待它们全部完成。

显然，后者要比前者性能更好。而更好的性能一般都会带来更好的用户体验。

异步（并发穿插）甚至可能仅仅增强高性能的印象，即便整个程序依然要用相同的时间才完成。用户对性能的印象意味着一切——如果不能再多的话！——和实际可测量的性能一样重要。

现在，我们想超越局部的异步模式，转而在程序级别的水平上讨论一些宏观的性能细节。

注意：你可能会想知道关于微性能问题，比如 `a++` 与 `++a` 哪个更快。我们会在下一章“基准分析与调优”中讨论这类性能细节。

## Web Workers

如果你有一些处理密集型的任务，但你不想让它们在主线程上运行（那样会使浏览器/UI变慢），你可能会希望JavaScript可以以多线程的方式操作。

在第一章中，我们详细地谈到了关于JavaScript如何是单线程的。那仍然是成立的。但是单线程不是组织你程序运行的唯一方法。

想象将你的程序分割成两块儿，在UI主线程上运行其中的一块儿，而在一个完全分离的线程上运行另一块儿。

这样的结构会引发什么我们需要关心的问题？

其一，你会想知道运行在一个分离的线程上是否意味着它在并行运行（在多CPU/内核的系统上），如此在第二个线程上长时间运行的处理将不会阻塞主程序线程。否则，“虚拟线程”所带来的好处，不会比我们已经在异步并发的JS中得到的更多。

而且你会想知道这两块儿程序是否访问共享的作用域/资源。如果是，那么你就要对付多线程语言（Java，C++等等）的所有问题，比如协作式或抢占式锁定（互斥，等）。这是很多额外的工作，而且不应当轻易着手。

换一个角度，如果这两块儿程序不能共享作用域/资源，你会想知道它们将如何“通信”。

所有这些我们需要考虑的问题，指引我们探索一个在近HTML5时代被加入web平台的特性，称为“Web Worker”。这是一个浏览器（也就是宿主环境）特性，而且几乎和JS语言本身没有任何关系。也就是说，JavaScript当前并没有任何特性可以支持多线程运行。

但是一个像你的浏览器那样的环境可以很容易地提供多个JavaScript引擎实例，每个都在自己的线程上，并允许你在每个线程上运行不同的程序。你的程序中分离的线程块儿中的每一个都称为一个“（Web）Worker”。这种并行机制叫做“任务并行机制”，它强调将你的程序分割成块儿来并行运行。

在你的主JS程序（或另一个Worker）中，你可以这样初始化一个Worker：

```
var w1 = new Worker( "http://some.url.1/mycoolworker.js" );
```

这个URL应当指向JS文件的位置（不是一个HTML网页！），它将会被加载到一个Worker。然后浏览器会启动一个分离的线程，让这个文件在这个线程上作为独立的程序运行。

注意：这种用这样的URL创建的Worker称为“专用（Dedicated）Worker”。但与提供一个外部文件的URL不同的是，你也可以通过提供一个Blob URL（另一个HTML5特性）来创建一个“内联（Inline）Worker”；它实质上是一个存储在单一（二进制）值中的内联文件。但是，Blob超出了我们要在这里讨论的范围。

Worker不会相互，或者与主程序共享任何作用域或资源——那会将所有的多线程编程的噩梦带到我们面前——取而代之的是一种连接它们的基本事件消息机制。

w1 Worker对象是一个事件监听器和触发器，它允许你监听Worker发出的事件也允许你向Worker发送事件。

这是如何监听事件（实际上，是固定的 "message" 事件）：

```
w1.addEventListener( "message", function(evt){
    // evt.data
});
```

而且你可以发送 "message" 事件给Worker：

```
w1.postMessage( "something cool to say" );
```

在Worker内部，消息是完全对称的：

```
// "mycoolworker.js"

addEventListener( "message", function(evt){
    // evt.data
} );

postMessage( "a really cool reply" );
```

要注意的是，一个专用Worker与它创建的程序是一对一的关系。也就是，“message”事件不需要消除任何歧义，因为我们可以确定它只可能来自于这种一对一关系——不是从Worker来的，就是从主页面来的。

通常主页面的程序会创建Worker，但是一个Worker可以根据需要初始化它自己的子Worker——称为subworker。有时将这样的细节委托给一个“主”Worker十分有用，它可以生成其他Worker来处理任务的一部分。不幸的是，在本书写作的时候，Chrome还没有支持subworker，然而Firefox支持。

要从创建一个Worker的程序中立即杀死它，可以在Worker对象（就像前一个代码段中的`w1`）上调用`terminate()`。突然终结一个Worker线程不会给它任何机会结束它的任务，或清理任何资源。这和你关闭浏览器的标签页来杀死一个页面相似。

如果你在浏览器中有两个或多个页面（或者打开同一个页面的多个标签页！），试着从同一个文件URL中创建Worker，实际上最终结果是完全分离的Worker。待一会儿我们就会讨论“共享”Worker的方法。

注意：看起来一个恶意的或者是呆头呆脑的JS程序可以很容易地通过在系统上生成数百个Worker来发起拒绝服务攻击（Dos攻击），看起来每个Worker都在自己的线程上。虽然一个Worker将会在存在于一个分离的线程上是有某种保证的，但这种保证不是没有限制的。系统可以自由决定有多少实际的线程/CPU/内核要去创建。没有办法预测或保证你能访问多少，虽然很多人假定它至少和可用的CPU/内核数一样多。我认为最安全的臆测是，除了主UI线程外至少有一个线程，仅此而已。

## Worker 环境

在Worker内部，你不能访问主程序的任何资源。这意味着你不能访问它的任何全局变量，你也不能访问页面的DOM或其他资源。记住：它是一个完全分离的线程。

然而，你可以实施网络操作（Ajax，WebSocket）和设置定时器。另外，Worker可以访问它自己的几个重要全局变量/特性的拷贝，包括`navigator`，`location`，`JSON`，和`applicationCache`。

你还可以使用`importScripts(..)`加载额外的JS脚本到你的Worker中：

```
// 在Worker内部
importScripts( "foo.js", "bar.js" );
```

这些脚本会被同步地加载，这意味着在文件完成加载和运行之前，`importScripts(..)` 调用会阻塞Worker的执行。

注意：还有一些关于暴露 `<canvas>` API给Worker的讨论，其中包括使`canvas`成为`Transferable`的（见“数据传送”一节），这将允许Worker来实施一些精细的脱线程图形处理，在高性能的游戏（WebGL）和其他类似应用中可能很有用。虽然这在任何浏览器中都还不存在，但是很有可能在近未来发生。

Web Worker的常见用途是什么？

- 处理密集型的数学计算
- 大数据集合的排序
- 数据操作（压缩，音频分析，图像像素操作等等）
- 高流量网络通信

## 数据传送

你可能注意到了这些用途中的大多数的一个共同性质，就是它们要求使用事件机制穿越线程间的壁垒来传递大量的信息，也许是双向的。

在Worker的早期，将所有数据序列化为字符串是唯一的选择。除了在两个方向上进行序列化时速度上变慢了，另外一个主要缺点是，数据是被拷贝的，这意味着内存用量翻了一倍（以及在后续垃圾回收上的流失）。

谢天谢地，现在我们有了几个更好的选择。

如果你传递一个对象，在另一端一个所谓的“结构化克隆算法（Structured Cloning Algorithm）”（[https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/The\\_structured\\_clone\\_algorithm](https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/The_structured_clone_algorithm)）会用于拷贝/复制这个对象。这个算法相当精巧，甚至可以处理带有循环引用的对象复制。`to-string/from-string`的性能劣化没有了，但用这种方式我们依然面对着内存用量的翻倍。IE10以上版本，和其他主流浏览器都对此有支持。

一个更好的选择，特别是对大的数据集合而言，是“Transferable对象”（<http://updates.html5rocks.com/2011/12/Transferable-Objects-Lightning-Fast>）。它使对象的“所有权”被传送，而对象本身没动。一旦你传送一个对象给Worker，它在原来的位置就空了出来或者不可访问——这消除了共享作用域的多线程编程中的灾难。当然，所有权的传送可以双向进行。

选择使用 Transferable 对象不需要你做太多；任何实现了 Transferable 接口（<https://developer.mozilla.org/en-US/docs/Web/API/Transferable>）的数据结构都将自动地以这种方式传递（Firefox 和 Chrome 支持此特性）。

举个例子，有类型的数组如 `Uint8Array`（见本系列的 *ES6 与未来*）是一个“Transferables”。这是你如何用 `postMessage(...)` 来传送一个 Transferable 对象：

```
// `foo` 是一个 `Uint8Array`

postMessage( foo.buffer, [ foo.buffer ] );
```

第一个参数是未经加工的缓冲，而第二个参数是要传送的内容的列表。

不支持 Transferable 对象的浏览器简单地降级到结构化克隆，这意味着性能上的降低，而不是彻底的特性失灵。

## 共享的 Workers

如果你的网站或应用允许多个标签页加载同一个网页（一个常见的特性），你也许非常想通过防止复制专用 Worker 来降低系统资源的使用量；这方面最常见的资源限制是网络套接字链接，因为浏览器限制同时连接到一个服务器的连接数量。当然，限制从客户端来的链接数也缓和了你的服务器资源需求。

在这种情况下，创建一个单独的中心化 Worker，让你的网站或应用的所有网页实例可以 共享 它是十分有用的。

这称为 `SharedWorker`，你会这样创建它（仅有 Firefox 与 Chrome 支持此特性）：

```
var w1 = new SharedWorker( "http://some.url.1/mycoolworker.js" );
```

因为一个共享 Worker 可以连接或被连接到你的网站上的多个程序实例或网页，Worker 需要一个方法来知道消息来自哪个程序。这种唯一的标识称为“端口（port）”——联想网络套接字端口。所以调用端程序必须使用 Worker 的 `port` 对象来通信：

```
w1.port.addEventListener( "message", handleMessages );

// ...

w1.port.postMessage( "something cool" );
```

另外，端口连接必须被初始化，就像这样：

```
w1.port.start();
```

在共享Worker内部，一个额外的事件必须被处理：“connect”。这个事件为这个特定的连接提供端口 object。保持多个分离的连接最简单的方法是在 port 上使用闭包，就像下面展示的那样，同时在 “connect” 事件的处理器内部定义这个连接的事件监听与传送：

```
// 在共享Worker的内部
addEventListener( "connect", function(evt){
    // 为这个连接分配的端口
    var port = evt.ports[0];

    port.addEventListener( "message", function(evt){
        // ...
        port.postMessage( ... );
        // ...
    } );

    // 初始化端口连接
    port.start();
} );
```

除了这点不同，共享与专用Worker的功能和语义是一样的。

注意：如果在一个端口的连接终结时还有其他端口的连接存活着的话，共享Worker也会存活下来，而专用Worker会在与初始化它的程序间接终结时终结。

## 填补 Web Workers

对于并行运行的JS程序在性能考量上，Web Worker十分吸引人。然而，你的代码可能运行在对此缺乏支持的老版本浏览器上。因为Worker是一个API而不是语法，所以在某种程度上它们可以被填补。

如果浏览器不支持Worker，那就根本没有办法从性能的角度来模拟多线程。Iframe通常被认为可以提供并行环境，但在所有的现代浏览器中它们实际上和主页运行在同一个线程上，所以用它们来模拟并行机制是不够的。

正如我们在第一章中详细讨论的，JS的异步能力（不是并行机制）来自于事件轮询队列，所以你可以用计时器（`setTimeout(..)` 等等）来强制模拟的Worker是异步的。然后你只需要提供Worker API的填补就行了。这里有一份列表

（<https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills#web-workers>），但坦白地说它们看起来都不怎么样。

我在这里（<https://gist.github.com/getify/1b26accb1a09aa53ad25>）写了一个填补‘Worker’的轮廓。它很基础，但应该满足了简单的‘Worker’支持，它的双向信息传递可以正确工作，还有“onerror”处理。你可能会扩展它来支持更多特性，比如`terminate()`或模拟共享Worker，只要你觉得合适。

注意：你不能模拟同步阻塞，所以这个填补不允许使用 `importScripts(..)`。另一个选择可能是转换并传递Worker的代码（一旦Ajax加载后），来重写一个 `importScripts(..)` 填补的一些异步形式，也许使用一个promise相关的接口。

## SIMD

一个指令，多个数据（SIMD）是一种“数据并行机制”形式，与Web Worker的“任务并行机制”相对应，因为他强调的不是程序逻辑的块儿被并行化，而是多个字节的数据被并行地处理。

使用SIMD，线程不提供并行机制。相反，现代CPU用数字的“向量”提供SIMD能力——想想：指定类型的数组——还有可以在所有这些数字上并行操作的指令；这些是利用底层操作的指令级别的并行机制。

使SIMD能力包含在JavaScript中的努力主要是由Intel带头的（<https://01.org/node/1495>），名义上是Mohammad Haghigat（在本书写作的时候），与Firefox和Chrome团队合作。SIMD处于早期标准化阶段，而且很有可能被加入未来版本的JavaScript中，很可能在ES7的时间框架内。

SIMD JavaScript提议向JS代码暴露短向量类型与API，它们在SIMD可用的系统中将操作直接映射为CPU指令的等价物，同时在非SIMD系统中退回到非并行化操作的“shim”。

对于数据密集型的应用程序（信号分析，对图形的矩阵操作等等）来说，这种并行数学处理在性能上的优势是十分明显的！

在本书写作时，SIMD API的早期提案形式看起来像这样：

```
var v1 = SIMD.float32x4( 3.14159, 21.0, 32.3, 55.55 );
var v2 = SIMD.float32x4( 2.1, 3.2, 4.3, 5.4 );

var v3 = SIMD.int32x4( 10, 101, 1001, 10001 );
var v4 = SIMD.int32x4( 10, 20, 30, 40 );

SIMD.float32x4.mul( v1, v2 );    // [ 6.597339, 67.2, 138.89, 299.97 ]
SIMD.int32x4.add( v3, v4 );      // [ 20, 121, 1031, 10041 ]
```

这里展示了两种不同的向量数据类型，32位浮点数和32位整数。你可以看到这些向量正好被设置为4个32位元素，这与大多数CPU中可用的SIMD向量的大小（128位）相匹配。在未来我们看到一个 `x8`（或更大！）版本的这些API也是可能的。

除了 `mul()` 和 `add()`，许多其他操作也很可能被加入，比

如 `sub()`，`div()`，`abs()`，`neg()`，`sqrt()`，`reciprocal()`，`reciprocalSqrt()`（算数运算），`shuffle()`（重拍向量元素），`and()`，`or()`，`xor()`，`not()`（逻辑运

算) , `equal()` , `greaterThan()` , `lessThan()` (比较运算) , `shiftLeft()` , `shiftRightLogical()` , `shiftRightArithmetic()` (轮换) , `fromFloat32x4()` , 和 `fromInt32x4()` (变换) 。

注意：这里有一个**SIMD**功能的官方“填补”（很有希望，预期的，着眼未来的填补）([https://github.com/johnmccutchan/ecmascript\\_simd](https://github.com/johnmccutchan/ecmascript_simd))，它描述了许多比我们在这一节中没有讲到的许多计划中的**SIMD**功能。

## asm.js

“**asm.js**” (<http://asmjs.org/>) 是可以被高度优化的JavaScript语言子集的标志。通过小心地回避那些特定的很难优化的（垃圾回收，强制转换，等等）机制和模式，**asm.js**风格的代码可以被JS引擎识别，而且用主动地底层优化进行特殊的处理。

与本章中讨论的其他性能优化机制不同的是，**asm.js**没必须要是必须被JS语言规范所采纳的东西。确实有一个**asm.js**规范 (<http://asmjs.org/spec/latest/>)，但它主要是追踪一组关于优化的候选对象的推论，而不是JS引擎的需求。

目前还没有新的语法被提案。取而代之的是，**asm.js**建议了一些方法，用来识别那些符合**asm.js**规则的既存标准JS语法，并且让引擎相应地实现它们自己的优化功能。

关于**asm.js**应当如何在程序中活动的问题，在浏览器生产商之间存在一些争议。早期版本的**asm.js**实验中，要求一个 `"use asm";` 编译附注（与**strict**模式的 `"use strict";` 类似）来帮助JS引擎来寻找**asm.js**优化的机会和提示。另一些人则断言**asm.js**应当只是一组启发式算法，让引擎自动地识别而不用作者做任何额外的事情，这意味着理论上既存的程序可以在不用做任何特殊的事情的情况下从**asm.js**优化中获益。

## 如何使用 **asm.js** 进行优化

关于**asm.js**需要理解的第一件事情是类型和强制转换。如果JS引擎不得不在变量的操作期间一直追踪一个变量内的值的类型，以便于在必要时它可以处理强制转换，那么就会有许多额外的工作使程序处于次优化状态。

注意：为了说明的目的，我们将在这里使用**asm.js**风格的代码，但要意识到的是你手写这些代码的情况不是很常见。**asm.js**的本意更多的是作为其他工具的编译目标，比如 **Emscripten** (<https://github.com/kripken/emscripten/wiki>)。当然你写自己的**asm.js**代码也是可能的，但是这通常不是一个好主意，因为那样的代码非常底层，而这意味着它会非常耗时而且易错。尽管如此，也会有情况使你想要为了**asm.js**优化的目的手动调整代码。

这里有一些“技巧”，你可以使用它们来提示支持**asm.js**的JS引擎变量/操作预期的类型是什么，以便于它可以跳过那些强制转换追踪的步骤。

举个例子：

```
var a = 42;
// ...
var b = a;
```

在这个程序中，赋值 `b = a` 在变量中留下了类型分歧的问题。然而，它可以写成这样：

```
var a = 42;
// ...
var b = a | 0;
```

这里，我们与值 `0` 一起使用了 `|`（“二进制或”），虽然它对值没有任何影响，但它确保这个值是一个32位整数。这段代码在普通的JS引擎中可以工作，但是当它运行在支持asm.js的JS引擎上时，它可以表示 `b` 应当总是被作为32位整数来对待，所以强制转换追踪可以被跳过。

类似地，两个变量之间的加法操作可以被限定为性能更好的整数加法（而不是浮点数）：

```
(a + b) | 0
```

再一次，支持asm.js的JS引擎可以看到这个提示，并推断 `+` 操作应当是一个32位整数加法，因为不论怎样整个表达式的最终结果都将自动是32位整数。

## asm.js 模块

在JS中最拖性能后腿的东西之一是关于内存分配，垃圾回收，与作用域访问。asm.js对于这些问题建一个的一个方法是，声明一个更加正式的asm.js“模块”——不要和ES6模块搞混；参见本系列的 *ES6与未来*。

对于一个asm.js模块，你需要明确传入一个被严格遵循的名称空间——在规范中以 `stdlib` 引用，因为它应当代表需要的标准库——来引入需要的符号，而不是通过词法作用域来使用全局对象。在最基本的情况下，`window` 对象就是一个可接受的用于asm.js模块的 `stdlib` 对象，但是你可能应该构建一个更加被严格限制的对象。

你还必须定义一个“堆（heap）”——这只是一个别致的词汇，它表示在内存中被保留的位置，变量不必要求内存分配或释放已使用内存就可以使用——并将它传入，这样asm.js模块就不必做任何导致内存流失的事情；它可以使用提前保留的空间。

一个“堆”就像一个有类型的 `ArrayBuffer`，比如：

```
var heap = new ArrayBuffer( 0x10000 ); // 64k 的堆
```

使用这个提前保留的64k的二进制空间，一个`asm.js`模块可以在这个缓冲区中存储或读取值，而不受任何内存分配与垃圾回收的性能损耗。比如，`heap`缓冲区可以在模块内部用于备份一个64位浮点数值的数组，像这样：

```
var arr = new Float64Array( heap );
```

好了，让我制作一个`asm.js`风格模块的快速，愚蠢的例子来描述这些东西是如何联系在一起的。我们将定义一个`foo(..)`，它为一个范围接收一个开始位置（`x`）和一个终止位置（`y`），并且计算这个范围内所有相邻的数字的积，然后最终计算这些值的平均值：

```

function fooASM(stdlib,foreign,heap) {
    "use asm";

    var arr = new stdlib.Int32Array( heap );

    function foo(x,y) {
        x = x | 0;
        y = y | 0;

        var i = 0;
        var p = 0;
        var sum = 0;
        var count = ((y|0) - (x|0)) | 0;

        // 计算范围内所有相邻的数字的积
        for (i = x | 0;
             (i | 0) < (y | 0);
             p = (p + 8) | 0, i = (i + 1) | 0
        ) {
            // 存储结果
            arr[ p >> 3 ] = (i * (i + 1)) | 0;
        }

        // 计算所有中间值的平均值
        for (i = 0, p = 0;
             (i | 0) < (count | 0);
             p = (p + 8) | 0, i = (i + 1) | 0
        ) {
            sum = (sum + arr[ p >> 3 ]) | 0;
        }

        return +(sum / count);
    }

    return {
        foo: foo
    };
}

var heap = new ArrayBuffer( 0x1000 );
var foo = fooASM( window, null, heap ).foo;

foo( 10, 20 );           // 233

```

注意：这个asm.js例子是为了演示的目的手动编写的，所以它与那些支持asm.js的编译工具生产的代码的表现不同。但是它展示了asm.js代码的典型性质，特别是类型提示与为了临时变量存储而使用 `heap` 缓冲。

第一个 `fooASM(..)` 调用用它的 `heap` 分配区建立了我们的 `asm.js` 模块。结果是一个我们可以调用任意多次的 `foo(..)` 函数。这些调用应当会被支持 `asm.js` 的 JS 引擎特别优化。重要的是，前面的代码完全是标准 JS，而且会在非 `asm.js` 引擎中工作的很好（但没有特别优化）。

很明显，使 `asm.js` 代码可优化的各种限制降低了广泛使用这种代码的可能性。对于任意给出的 JS 程序，`asm.js` 没有必要为成为一个一般化的优化集合。相反，它的本意是提供针对一种处理特定任务——如密集数学操作（那些用于游戏中图形处理的）——的优化方法。

## 复习

本书的前四章基于这样的前提：异步编码模式给了你编写更高效代码的能力，这通常是一个非常重要的改进。但是异步行为也就能帮你这么多，因为它在基础上仍然使用一个单独的事件轮询线程。

所以在这一章我们涵盖了几种程序级别的机制来进一步提升性能。

`Web Worker` 让你在一个分离的线程上运行一个 JS 文件（也就是程序），使用异步事件在线程之间传递消息。对于将长时间运行或资源密集型任务挂载到一个不同线程，从而让主 UI 线程保持相应来说，它们非常棒。

`SIMD` 提议将 CPU 级别的并行数学操作映射到 JavaScript API 上来提供高性能数据并行操作，比如在大数据集合上进行数字处理。

最后，`asm.js` 描述了一个 JavaScript 的小的子集，它回避了 JS 中不易优化的部分（比如垃圾回收与强制转换）并让 JS 引擎通过主动优化识别并运行这样的代码。`asm.js` 可以手动编写，但是极其麻烦且易错，就像手动编写汇编语言。相反，`asm.js` 的主要意图是作为一个从其他高度优化的程序语言交叉编译来的目标——例如，

`Emscripten` (<https://github.com/kripken/emscripten/wiki>) 可以将 C/C++ 转译为 JavaScript。

虽然在本章没有明确地提及，在很早以前的有关 JavaScript 的讨论中存在着更激进的想法，包括近似地直接多线程功能（不仅仅是隐藏在数据结构 API 后面）。无论这是否会明确地发生，还是我们将看到更多并行机制偷偷潜入 JS，但是在 JS 中发生更多程序级别优化的未来是可以确定的。

# 你不懂JS: 异步与性能

## 第六章: 基准分析与调优

本书的前四章都是关于代码模式（异步与同步）的性能，而第五章是关于宏观的程序结构层面的性能，本章从微观层面继续性能的话题，关注的焦点在一个表达式/语句上。

好奇心的一个最常见的领域——确实，一些开发者十分痴迷于此——是分析和测试如何写一行或一块儿代码的各种选项，看哪一个更快。

我们将会看到这些问题中的一些，但重要的是要理解从最开始这一章就不是为了满足对微性能调优的痴迷，比如某种给定的JS引擎运行 `++a` 是否要比运行 `a++` 快。这一章更重要的目标是，搞清楚哪种JS性能要紧而哪种不要紧，和如何指出这种不同。

但在我们达到目的之前，我们需要探索一下如何最准确和最可靠地测试JS性能，因为有太多的误解和谜题充斥着我们集体主义崇拜的知识库。我们需要将这些垃圾筛出去以便找到清晰的答案。

### 基准分析 (Benchmarking)

好了，是时候开始消除一些误解了。我敢打赌，最广大的JS开发者们，如果被问到如何测量一个特定操作的速度（执行时间），将会一头扎进这样的东西：

```
var start = (new Date()).getTime();      // 或者`Date.now()`
// 做一些操作

var end = (new Date()).getTime();

console.log( "Duration:", (end - start) );
```

如果这大致就是你想到的，请举手。是的，我就知道你会这么想。这个方式有许多错误，但是别难过；我们都这么干过。

这种测量到底告诉了你什么？对于当前的操作的执行时间来说，理解它告诉了你什么和没告诉你什么是学习如何正确测量JavaScript的性能的关键。

如果持续的时间报告为 `0`，你也许会试图认为它花的时间少于1毫秒。但是这不是非常准确。一些平台不能精确到毫秒，反而是在更大的时间单位上更新计时器。举个例子，老版本的windows（IE也是如此）只有15毫秒的精确度，这意味着要得到与 `0` 不同的报告，操作就必须

须至少要花这么长时间！

另外，不管被报告的持续时间是多少，你唯一真实知道的是，操作在当前这一次运行中大概花了这么长时间。你几乎没有信心说它将总是以这个速度运行。你不知道引擎或系统是否在就在那个确切的时刻进行了干扰，而在其他的时候这个操作可能会运行的快一些。

要是持续的时间报告为 4 呢？你确信它花了大概4毫秒？不，它可能没花那么长时间，而且在取得 `start` 或 `end` 时间戳时会有一些其他的延迟。

更麻烦的是，你也不知道这个操作测试所在的环境是不是过于优化了。这样的情况是有可能的：JS引擎找到了一个办法来优化你的测试用例，但是在更真实的程序中这样的优化将会被稀释或者根本不可能，如此这个操作将会比你测试时运行的慢。

那么...我们知道什么？不幸的是，在这种状态下，我们几乎什么都不知道自己。可信度如此低的东西甚至不够你建立自己的判断。你的“基准分析”基本没用。更糟的是，它隐含的这种不成立的可信度很危险，不仅是对你，而且对其他人也一样：认为导致这些结果的条件不重要。

## 重复

“好的，”你说，“在它周围放一个循环，让整个测试需要的时间长一些。”如果你重复一个操作 100 次，而整个循环在报告上说总共花了 137ms，那么你可以除以 100 并得到每次操作平均持续时间 1.37ms，对吧？

其实，不确切。

对于你打算在你的整个应用程序范围内推广的操作的性能，仅靠一个直白的数据上的平均做出判断绝对是不够的。在一百次迭代中，即使是几个极端值（或高或低）就可以歪曲平均值，而后当你反复实施这个结论时，你就更进一步扩大了这种歪曲。

与仅仅运行固定次数的迭代不同，你可以选择将测试的循环运行一个特定长的时间。那可能更可靠，但是你如何决定运行多长时间？你可能会猜它应该是你的操作运行一次所需时间的倍数。错。

实际上，循环持续的时间应当基于你使用的计时器的精度，具体地将不精确的可能性最小化。你的计时器精度越低，你就需要运行更长时间来确保你将错误的概率最小化了。一个 15ms 的计时器对于精确的基准分析来说太差劲儿了；为了把它的不确定性（也就是“错误率”）最小化到低于 1%，你需要将测试的迭代循环运行 750ms。一个 1ms 的计时器只需要一个循环运行 50ms 就可以得到相同的可信度。

但，这只是一个样本。为了确信你排除了歪曲结果的因素，你将会想要许多样本来求平均值。你还会想要明白最差的样本有多慢，最佳的样本有多快，最差与最佳的情况相差多少等等。你想知道的不仅是一个数字告诉你某个东西跑的多块，而且还需要一个关于这个数字有多可信的量化表达。

另外，你可能想要组合这些不同的技术（还有其他的），以便于你可以在所有这些可能的方式中找到最佳的平衡。

这一切只不过是开始所需的最低限度的认识。如果你曾经使用比我刚才几句话带过的东西更不严谨的方式进行基准分析，那么...“你不懂：正确的基准分析”。

## Benchmark.js

任何有用而且可靠的基准分析应当基于统计学上的实践。我不是要在这里写一章统计学，所以我会带过一些名词：标准差，方差，误差边际。如果你不知道这些名词意味着什么——我在大学上过统计学课程，而我依然对他们有点儿晕——那么实际上你没有资格去写你自己的基准分析逻辑。

幸运的是，一些像John-David Dalton和Mathias Bynens这样的聪明家伙明白这些概念，并且写了一个统计学上的基准分析工具，称为Benchmark.js (<http://benchmarkjs.com/>)。所以我可以简单地说：“用这个工具就行了。”来终结这个悬念。

我不会重复他们的整个文档来讲解Benchmark.js如何工作；他们有很棒的API文档 (<http://benchmarkjs.com/docs>) 你可以阅读。另外这里还有一些了不起的文章 (<http://calendar.perfplanet.com/2010/bulletproof-javascript-benchmarks/>) (<http://monsur.hossa.in/2012/12/11/benchmarkjs.html>) 讲解细节与方法学。

但是为了快速演示一下，这是你如何用Benchmark.js来运行一个快速的性能测试：

```
function foo() {
    // 需要测试的操作
}

var bench = new Benchmark(
    "foo test",           // 测试的名称
    foo,                  // 要测试的函数（仅仅是内容）
    {
        // ...
        // 额外的选项（参见文档）
    }
);

bench.hz;                // 每秒钟执行的操作数
bench.stats.moe;         // 误差边际
bench.stats.variance;    // 所有样本上的方差
// ...
```

比起我在这里的窥豹一斑，关于使用Benchmark.js还有许多需要学习的东西。不过重点是，为了给一段给定的JavaScript代码建立一个公平，可靠，并且合法的性能基准分析，Benchmark.js包揽了所有的复杂性。如果你想要试着对你的代码进行测试和基准分析，这个库应当是你第一个想到的地方。

我们在这里展示的是测试一个单独操作X的用法，但是相当常见的情况是你想要用X和Y进行比较。这可以通过简单地在一个“Suite”（一个Benchmark.js的组织特性）中建立两个测试来很容易做到。然后，你对照地运行它们，然后比较统计结果来对为什么X或Y更快做出论断。

Benchmark.js理所当然地可以被用于在浏览器中测试JavaScript（参见本章稍后的“jsPerf.com”一节），但它也可以运行在非浏览器环境中（Node.js等等）。

一个很大程度上没有触及的Benchmark.js的潜在用例是，在你的Dev或QA环境中针对你的应用程序的JavaScript的关键路径运行自动化的性能回归测试。与在部署之前你可能运行单元测试的方式相似，你也可以将性能与前一次基准分析进行比较，来观测你是否改进或恶化了应用程序性能。

## Setup/Teardown

在前一个代码段中，我们略过了“额外选项（extra options）”`{ .. }`对象。但是这里有两个我们应当讨论的选项 `setup` 和 `teardown`。

这两个选项让你定义在你的测试用例开始运行前和运行后被调用的函数。

一个需要理解的极其重要的事情是，你的 `setup` 和 `teardown` 代码 不会为每一次测试迭代而运行。考虑它的最佳方式是，存在一个外部循环（重复的轮回），和一个内部循环（重复的测试迭代）。`setup` 和 `teardown` 会在每个 外部循环（也就是轮回）迭代的开始和末尾运行，但不是在内部循环。

为什么这很重要？让我们想象你有一个看起来像这样的测试用例：

```
a = a + "w";
b = a.charAt( 1 );
```

然后，你这样建立你的测试 `setup`：

```
var a = "x";
```

你的意图可能是相信对每一次测试迭代 `a` 都以值 "x" 开始。

但它不是！它使 `a` 在每一次测试轮回中以 "x" 开始，而后你的反复的`+ "w"` 连接将使 `a` 的值越来越大，即便你永远唯一访问的是位于位置 1 的字符 "w"。

当你想利用副作用来改变某些东西比如DOM，向它追加一个子元素时，这种意外经常会咬到你。你可能认为的父元素每次都被设置为空，但他实际上被追加了许多元素，而这可能会显著地歪曲你的测试结果。

## 上下文为王

不要忘了检查一个指定的性能基准分析的上下文环境，特别是在X与Y之间进行比较时。仅仅因为你的测试显示X比Y速度快，并不意味着“X比Y快”这个结论是实际上有意义的。

举个例子，让我们假定一个性能测试显示出X每秒可以运行1千万次操作，而Y每秒运行8百万次。你可以声称Y比X慢20%，而且在数学上你是对的，但是你的断言并不向像你认为的那么有用。

让我们更加苛刻地考虑这个测试结果：每秒1千万次操作就是每毫秒1万次操作，就是每微秒10次操作。换句话说，一次操作要花0.1毫秒，或者100纳秒。很难体会100纳秒到底有多小，可以这样比较一下，通常认为人类的眼睛一般不能分辨小于100毫秒的变化，而这要比X操作的100纳秒的速度慢100万倍。

即便最近的科学显示，大脑可能的最快处理速度是13毫秒（比先前的论断快大约8倍），这意味着X的运行速度依然要比人类大脑可以感知事情的发生要快12万5千倍。X运行的非常，非常快。

但更重要的是，让我们来谈谈X与Y之间的不同，每秒2百万次的差。如果X花100纳秒，而Y花80纳秒，差就是20纳秒，也就是人类大脑可以感知的间隔的65万分之一。

我要说什么？这种性能上的差别根本就一点儿都不重要！

但是等一下，如果这种操作将要一个接一个地发生许多次呢？那么差异就会累加起来，对吧？

好的，那么我们就要问，操作X有多大可能性将要一次又一次，一个接一个地运行，而且为了人类大脑能够感知的一线希望而不得不发生65万次。而且，它不得不在一个紧凑的循环中发生5百万到1千万次，才能接近于有意义。

虽然你们之中的计算机科学家会反对说这是可能的，但是你们之中的现实主义者们应当对这究竟有多大可能性进行可行性检查。即使在极其稀少的偶然中这有实际意义，但是在绝大多数情况下它没有。

你们大量的针对微小操作的基准分析结果——比如 `++x` 对 `x++` 的神话——完全是伪命题，只不过是用来支持在性能的基准上X应当取代Y的结论。

## 引擎优化

你根本无法可靠地这样推断：如果在你的独立测试中X要比Y快10微妙，这意味着X总是比Y快所以应当总是被使用。这不是性能的工作方式。它要复杂太多了。

举个例子，让我们想象（纯粹地假想）你在测试某些行为的微观性能，比如比较：

```

var twelve = "12";
var foo = "foo";

// 测试 1
var X1 = parseInt( twelve );
var X2 = parseInt( foo );

// 测试 2
var Y1 = Number( twelve );
var Y2 = Number( foo );

```

如果你明白与 `Number(..)` 比起来 `parseInt(..)` 做了什么，你可能会在直觉上认为 `parseInt(..)` 潜在地有“更多工作”要做，特别是在 `foo` 的测试用例下。或者你可能在直觉上认为在 `foo` 的测试用例下它们应当有同样多的工作要做，因为它们俩应当能够在第一个字符 "`f`" 处停下。

哪一种直觉正确？老实说我不知道。但是我会制造一个与你的直觉无关的测试用例。当你测试它的时候结果会是什么？我又一次在这里制造一个纯粹的假想，我们没实际上尝试过，我也不关心。

让我们假装 `x` 与 `y` 的测试结果在统计上是相同的。那么你关于 "`f`" 字符上发生的事情的直觉得到确认了吗？没有。

在我们的假想中可能发生这样的事情：引擎可能会识别出变量 `twelve` 和 `foo` 在每个测试中仅被使用了一次，因此它可能会决定要内联这些值。然后它可能发现 `Number("12")` 可以替换为 `12`。而且也许在 `parseInt(..)` 上得到相同的结论，也许不会。

或者一个引擎的死代码移除启发式算法会搅和进来，而且它发现变量 `x` 和 `y` 都没有被使用，所以声明它们是没有意义的，所以最终在任一个测试中都不做任何事情。

而且所有这些都只是关于一个单独测试运行的假设而言的。比我们在这里用直觉想象的，现代的引擎复杂得更加难以置信。它们会使用所有的招数，比如追踪并记录一段代码在一段很短的时间内的行为，或者使用一组特别限定的输入。

如果引擎由于固定的输入而用特定的方法进行了优化，但是在你的真实的程序中你给出了更多种类的输入，以至于优化机制决定使用不同的方式呢（或者根本不优化！）？或者如果因为引擎看到代码被基准分析工具运行了成千上万次而进行了优化，但在你的真实程序中它将仅会运行大约 100 次，而在这些条件下引擎认定优化不值得呢？

所有这些我们刚刚假想的优化措施可能会发生在我们的被限定的测试中，但在更复杂的程序中引擎可能不会那么做（由于种种原因）。或者正相反——引擎可能不会优化这样不起眼的代码，但是可能会更倾向于在系统已经被一个更精巧的程序消耗后更加积极地优化。

我想要说的是，你不能确切地知道这背后究竟发生了什么。你能搜罗的所有猜测和假想几乎不会提炼成任何坚实的依据。

难道这意味着你不能真正地做有用的测试了吗？绝对不是！

这可以归结为测试不真实的代码会给你不真实的结果。在尽可能的情况下，你应当测试真实的，有意义的代码段，并且在最接近你实际能够期望的真实条件下进行。只有这样你得到的结果才有机会模拟现实。

像 `++x` 和 `x++` 这样的微观基准分析简直和伪命题一模一样，我们也许应该直接认为它就是。

## jsPerf.com

虽然Benchmark.js对于在你使用的任何JS环境中测试代码性能很有用，但是如果你需要从许多不同的环境（桌面浏览器，移动设备等）汇总测试结果并期望得到可靠的测试结论，它就显得能力不足。

举例来说，Chrome在高端的桌面电脑上与Chrome移动版在智能手机上的表现就大相径庭。而一个充满电的智能手机与一个只剩2%电量，设备开始降低无线电和处理器的能源供应的智能手机的表现也完全不同。

如果在横跨多于一种环境的情况下，你想在任何合理的意义上宣称“X比Y快”，那么你就需要实际测试尽可能多的真实世界的环境。只因为Chrome执行某种X操作比Y快并不意味着所有的浏览器都是这样。而且你还可能想要根据你的用户的人口统计交叉参照多种浏览器测试运行的结果。

有一个为此目的而生的牛X网站，称为jsPerf (<http://jsperf.com>)。它使用我们前面提到的 Benchmark.js 库来运行统计上正确且可靠的测试，并且可以让测试运行在一个你可交给其他人的公开URL上。

每当一个测试运行后，其结果都被收集并与这个测试一起保存，同时累积的测试结果将在网页上被绘制成图供所有人阅览。

当在这个网站上创建测试时，你一开始有两个测试用例可以填写，但你可以根据需要添加任意多个。你还可以建立在每次测试轮回开始时运行的 `setup` 代码，和在每次测试轮回结束前运行的 `teardown` 代码。

注意：一个只做一个测试用例（如果你只对一个方案进行基准分析而不是相互对照）的技巧是，在第一次创建时使用输入框的占位提示文本填写第二个测试输入框，之后编辑这个测试并将第二个测试留为空白，这样它就会被删除。你可以稍后添加更多测试用例。

你可以顶一个页面的初始配置（引入库文件，定义工具函数，声明变量，等等）。如有需要这里也有选项可以定义 `setup` 和 `teardown` 行为——参照前面关于 Benchmark.js 的讨论中的“Setup/Tear down”一节。

## 可行性检查

jsPerf是一个奇妙的资源，但它上面有许多公开的糟糕测试，当你分析它们时会发现，由于在本章目前为止罗列的各种原因，它们有很大的漏洞或者是伪命题。

考虑：

```
// 用例 1
var x = [];
for (var i=0; i<10; i++) {
    x[i] = "x";
}

// 用例 2
var x = [];
for (var i=0; i<10; i++) {
    x[x.length] = "x";
}

// 用例 3
var x = [];
for (var i=0; i<10; i++) {
    x.push("x");
}
```

关于这个测试场景有一些现象值得我们深思：

- 开发者们在测试用例中加入自己的循环极其常见，而他们忘记了Benchmark.js已经做了你所需要的所有反复。这些测试用例中的`for`循环有很大的可能是完全不必要的噪音。
- 在每一个测试用例中都包含了`x`的声明与初始化，似乎是不必要的。回想早前如果`x = []`存在于`setup`代码中，它实际上不会在每一次测试迭代前执行，而是在每一个轮回的开始执行一次。这意味着`x`将会持续地增长到非常大，而不仅是`for`循环中暗示的大小`10`。

那么这是有意确保测试仅被限制在很小的数组上（大小为`10`）来观察JS引擎如何动作？这可能是有意的，但如果是，你就不得不考虑它是否过于关注内微妙的部实现细节了。

另一方面，这个测试的意图包含数组实际上会增长到非常大的情况吗？JS引擎对大数组的行为与真实世界中预期的用法相比有意义且正确吗？

- 它的意图是要找出`x.length`或`x.push(..)`在数组`x`的追加操作上拖慢了多少性能吗？好吧，这可能是一个合法的测试。但再一次，`push(..)`是一个函数调用，所以它理所当然地要比`[..]`访问慢。可以说，用例1与用例2比用例3更合理。

这里有另一个展示苹果比橘子的常见漏洞的例子：

```
// 用例 1
var x = ["John", "Albert", "Sue", "Frank", "Bob"];
x.sort();

// 用例 2
var x = ["John", "Albert", "Sue", "Frank", "Bob"];
x.sort( function mySort(a,b){
    if (a < b) return -1;
    if (a > b) return 1;
    return 0;
} );
```

这里，明显的意图是要找出自定义的 `mySort(..)` 比较器比内建的默认比较器慢多少。但是通过将函数 `mySort(..)` 作为内联的函数表达式生命，你就创建了一个不合理的/伪命题的测试。这里，第二个测试用例不仅测试用户自定义的JS函数，而且它还测试为每一个迭代创建一个新的函数表达式。

不知这会不会吓到你，如果你运行一个相似的测试，但是将它更改为比较内联函数表达式与预先声明的函数，内联函数表达式的创建可能要慢2%到20%！

除非你的测试的意图就是要考虑内联函数表达式创建的“成本”，一个更好/更合理的测试是将 `mySort(..)` 的声明放在页面的 `setup` 中——不要放在测试的 `setup` 中，因为这会为每次轮回进行不必要的重复声明——然后简单地在测试用例中通过名称引用它：`x.sort(mySort)`。

基于前一个例子，另一种造成苹果比橘子场景的陷阱是，不透明地对一个测试用例回避或添加“额外的工作”：

```
// 用例 1
var x = [12, -14, 0, 3, 18, 0, 2.9];
x.sort();

// 用例 2
var x = [12, -14, 0, 3, 18, 0, 2.9];
x.sort( function mySort(a,b){
    return a - b;
} );
```

将先前提到的内联函数表达式陷阱放在一边不谈，第二个用例的 `mySort(..)` 可以在这里工作是因为你给它提供了一组数字，而在字符串的情况下肯定会失败。第一个用例不会扔出错误，但是它的实际行为将会不同而且会有不同的结果！这应当很明显，但是：两个测试用例之间结果的不同，几乎可以否定了整个测试的合法性！

但是除了结果的不同，在这个用例中，内建的 `sort(..)` 比较器实际上要比 `mySort()` 做了更多“额外的工作”，内建的比较器将被比较的值转换为字符串，然后进行字典顺序的比较。这样第一个代码段的结果为 `[-14, 0, 0, 12, 18, 2.9, 3]` 而第二段代码的结果为 `[-14, 0, 0, 2.9, 3, 12, 18]`（就测试的意图来讲可能更准确）。

所以这个测试是不合理的，因为它的两个测试用例实际上没有做相同的工作。你得到的任何结果都将是伪命题。

这些同样的陷阱可以微妙的多：

```
// 用例 1
var x = false;
var y = x ? 1 : 2;

// 用例 2
var x;
var y = x ? 1 : 2;
```

这里的意图可能是要测试如果 `x` 表达式不是 Boolean 的情况下，`? :` 操作符将要进行的 Boolean 转换对性能的影响（参见本系列的 [类型与文法](#)）。那么，根据在第二个用例中将会有额外的工作进行转换的事实，你看起来没问题。

微妙的问题呢？你在第一个测试用例中设定了 `x` 的值，而没在另一个中设置，那么你实际上在第一个用例中做了在第二个用例中没做的工作。为了消灭任何潜在的扭曲（尽管很微小），可以这样：

```
// 用例 1
var x = false;
var y = x ? 1 : 2;

// 用例 2
var x = undefined;
var y = x ? 1 : 2;
```

现在两个用例都有一个赋值了，这样你想要测试的东西——`x` 的转换或者不转换——会更加正确的被隔离并测试。

## 编写好的测试

来看看我能否清晰地表达我想在这里申明的更重要的事情。

好的测试作者需要细心地分析性地思考两个测试用例之间存在什么样的差别，和它们之间的差别是否是有意的或无意的。

有意的差别当然是正常的，但是产生歪曲结果的无意的差异实在太容易了。你不得不非常非常小心地回避这种歪曲。另外，你可能预期一个差异，但是你的意图是什么对于你的测试的其他读者来讲不那么明显，所以他们可能会错误地怀疑（或者相信！）你的测试。你如何搞定这个呢？

编写更好，更清晰的测试。另外，花些时间用文档确切地记录下你的测试意图是什么（使用jsPerf.com的“Description”字段，或/和代码注释），即使是微小的细节。明确地表示有意的差别，这将帮助其他人和未来的你自己更好地找出那些可能歪曲测试结果的无意的差别。

将与你的测试无关的东西隔离开来，通过在页面或测试的setup设置中预先声明它们，使它们位于测试计时部分的外面。

与将你的真实代码限制在很小的一块，并脱离上下文环境来进行基准分析相比，测试与基准分析在它们包含更大的上下文环境（但仍然有意义）时表现更好。这些测试将会趋向于运行得更慢，这意味着你发现的任何差别都在上下文环境中更有意义。

## 微观性能

好了，直至现在我们一直围绕着微观性能的问题跳舞，并且一般上不赞成痴迷于它们。我想花一点儿时间直接解决它们。

当你考虑对你的代码进行性能基准分析时，第一件需要习惯的事情就是你写的代码不总是引擎实际运行的代码。我们在第一章中讨论编译器的语句重排时简单地看过这个话题，但是这里我们将要说明编译器能有时决定运行与你编写的不同的代码，不仅是不同的顺序，而是不同的替代品。

让我们考虑这段代码：

```
var foo = 41;

(function(){
  (function(){
    (function(baz){
      var bar = foo + baz;
      // ...
    })(1);
  })();
})();
```

你也许会认为在最里面的函数的 `foo` 引用需要做一个三层作用域查询。我们在这个系列丛书的作用域与闭包一卷中涵盖了词法作用域如何工作，而事实上编译器通常缓存这样的查询，以至于从不同的作用域引用 `foo` 不会实质上“花费”任何额外的东西。

但是这里有些更深刻的东西需要思考。如果编译器认识到 `foo` 除了这一个位置外没有被任何其他地方引用，进而注意到它的值除了这里的 `41` 外没有任何变化会怎么样呢？

JS编译器能够决定干脆完全移除 `foo` 变量，并内联它的值是可能和可接受的，比如这样：

```
(function(){
  (function(){
    (function(baz){
      var bar = 41 + baz;
      // ...
    })(1);
  })();
})();
```

注意：当然，编译器可能也会对这里的 `baz` 变量进行相似的分析和重写。

但你开始将你的JS代码作为一种告诉引擎去做什么的提示或建议来考虑，而不是一种字面上的需求，你就会理解许多对零碎的语法细节的痴迷几乎是毫无根据的。

另一个例子：

```
function factorial(n) {
  if (n < 2) return 1;
  return n * factorial( n - 1 );
}

factorial( 5 );           // 120
```

啊，一个老式的“阶乘”算法！你可能会认为JS引擎将会原封不动地运行这段代码。老实说，它可能会——但我不是很确定。

但作为一段轶事，用C语言表达同样的代码并使用先进的优化处理进行编译时，将会导致编译器认为 `factorial(5)` 调用可以被替换为常数值 `120`，完全消除这个函数以及调用！

另外，一些引擎有一种称为“递归展开（unrolling recursion）”的行为，它会意识到你表达的递归实际上可以用循环“更容易”（也就是更优化地）地完成。前面的代码可能会被JS引擎重写为：

```
function factorial(n) {
  if (n < 2) return 1;

  var res = 1;
  for (var i=n; i>1; i--) {
    res *= i;
  }
  return res;
}

factorial( 5 );           // 120
```

现在，让我们想象在前一个片段中你曾经担心 `n * factorial(n-1)` 或 `n *= factorial(--n)` 哪一个运行的更快。也许你甚至做了性能基准分析来试着找出哪个更好。但是你忽略了一个事实，就是在更大的上下文环境中，引擎也许不会运行任何一行代码，因为它可能展开了递归！

说到 `--`，`--n` 与 `n--` 的对比，经常被认为可以通过选择 `--n` 的版本进行优化，因为理论上在汇编语言层面的处理上，它要做的努力少一些。

在现代的JavaScript中这种痴迷基本上是没道理的。这种事情应当留给引擎来处理。你应该编写最合理的代码。比较这三个 `for` 循环：

```
// 方式 1
for (var i=0; i<10; i++) {
    console.log( i );
}

// 方式 2
for (var i=0; i<10; ++i) {
    console.log( i );
}

// 方式 3
for (var i=-1; ++i<10; ) {
    console.log( i );
}
```

就算你有一些理论支持第二或第三种选择要比第一种的性能好那么一点点，充其量只能算是可疑，第三个循环更加使人困惑，因为为了使提前递增的 `++i` 被使用，你不得不让 `i` 从 `-1` 开始来计算。而第一个与第二个选择之间的区别实际上无关紧要。

这样的事情是完全有可能的：JS引擎也许看到一个 `i++` 被使用的地方，并意识到它可以安全地替换为等价的 `++i`，这意味着你决定挑选它们中的哪一个所花的时间完全被浪费了，而且这么做的产出毫无意义。

这是另外一个常见的愚蠢的痴迷于微观性能的例子：

```
var x = [ .. ];

// 方式 1
for (var i=0; i < x.length; i++) {
    // ..
}

// 方式 2
for (var i=0, len = x.length; i < len; i++) {
    // ..
}
```

这里的理论是，你应当在变量 `len` 中缓存数组 `x` 的长度，因为从表面上看它不会改变，来避免在循环的每一次迭代中都查询 `x.length` 所花的开销。

如果你围绕 `x.length` 的用法进行性能基准分析，与将它缓存在变量 `len` 中的用法进行比较，你会发现虽然理论听起来不错，但是在实践中任何测量出的差异都是在统计学上完全没有意义的。

事实上，在像v8这样的引擎中，可以看到(<http://mrale.ph/blog/2014/12/24/array-length-caching.html>)通过提前缓存长度而不是让引擎帮你处理它会使事情稍稍恶化。不要尝试在聪明上战胜你的JavaScript引擎，当它来到性能优化的地方时你可能会输给它。

## 不是所有的引擎都一样

在各种浏览器中的不同JS引擎可以称为“规范兼容的”，虽然各自有完全不同的方式处理代码。JS语言规范不要求与性能相关的任何事情——除了将在本章稍后将要讲解的ES6“尾部调用优化（Tail Call Optimization）”。

引擎可以自由决定哪一个操作将会受到它的关注而被优化，也许代价是在另一种操作上的性能降低一些。要为一种操作找到一种在所有的浏览器中总是运行的更快的方式是非常不现实的。

在JS开发者社区的一些人发起了一项运动，特别是那些使用Node.js工作的人，去分析v8 JavaScript引擎的具体内部实现细节，并决定如何编写定制的JS代码来最大限度的利用v8的工作方式。通过这样的努力你实际上可以在性能优化上达到惊人的高度，所以这种努力的收益可能十分高。

一些针对v8的经常被引用的例子是

(<https://github.com/petkaantonov/bluebird/wiki/Optimization-killers>)：

- 不要将 `arguments` 变量从一个函数传递到任何其他函数中，因为这样的“泄露”放慢了函数实现。
- 将一个 `try..catch` 隔离到它自己的函数中。浏览器在优化任何含有 `try..catch` 的函数时都会苦苦挣扎，所以将这样的结构移动到它自己的函数中意味着你持有不可优化的危害的同时，让其周围的代码是可以优化的。

但与其聚焦在这些具体的窍门上，不如让我们在一般意义上对v8专用的优化方式进行一下合理性检验。

你真的在编写仅仅需要在一种JS引擎上运行的代码吗？即便你的代码当前是完全为了Node.js，那么假设v8将总是被使用的JS引擎可靠吗？从现在开始的几年以后的某一天，你有没有可能会选择除了Node.js之外的另一种服务器端JS平台来运行你的程序？如果你以前所做的优化现在在新的引擎上成为了执行这种操作的很慢的方式怎么办？

或者如果你的代码总是在v8上运行，但是v8在某个时点决定改变一组操作的工作方式，是的曾经快的现在变慢了，曾经慢的变快了呢？

这些场景也都不只是理论上的。曾经，将多个字符串值放在一个数组中然后在这个数组上调用 `join("")` 来连接这些值，要比仅使用 `+` 直接连接这些值要快。这件事的历史原因很微妙，但它与字符串值如何被存储和在内存中如何管理的内部实现细节有关。

结果，当时在业界广泛传播的“最佳实践”建议开发者们总是使用数组 `join(..)` 的方式。而且有许多人遵循了。

但是，某一天，JS引擎改变了内部管理字符串的方式，而且特别在 `+` 连接上做了优化。他们并没有放慢 `join(..)`，但是他们在帮助 `+` 用法上做了更多的努力，因为它依然十分普遍。

注意：某些特定方法的标准化和优化的实施，很大程度上决定于它被使用的广泛程度。这经常（隐喻地）称为“*paving the cowpath*”（不提前做好方案，而是等到事情发生了再去应对）。

一旦处理字符串和连接的新方式定型，所有在世界上运行的，使用数组 `join(..)` 来连接字符串的代码都不幸地变成了次优的方式。

另一个例子：曾经，Opera浏览器在如何处理基本包装对象的封箱/拆箱（参见本系列的类型与文法）上与其他浏览器不同。因此他们给开发者的建议是，如果一个原生 `string` 值的属性（如 `length`）或方法（如 `charAt(..)`）需要被访问，就使用一个 `String` 对象取代它。这个建议也许对那时的Opera是正确的，但是对于同时代的其他浏览器来说简直就是完全相反的，因为它们都对原生 `string` 进行了专门的优化，而不是对它们的包装对象。

我认为即使是对今天的代码，这种种陷阱即便可能性不高，至少也是可能的。所以对于在我的JS代码中单纯地根据引擎的实现细节来进行大范围的优化这件事来说我会非常小心，特别是如果这些细节仅对一种引擎成立时。

反过来也有一些事情需要警惕：你不应当为了绕过某一种引擎难于处理的地方而改变一块代码。

历史上，IE是导致许多这种挫折的领头羊，在老版本的IE中曾经有许多场景，在当时的其他主流浏览器中看起来没有太多麻烦的性能方面苦苦挣扎。我们刚刚讨论的字符串连接在IE6和IE7的年代就是一个真实的问题，那时候使用 `join(..)` 就可能要比使用 `+` 能得到更好的性能。

不过为了一种浏览器的性能问题而使用一种很有可能在其他所有浏览器上是次优的编码方式，很难说是正当的。即便这种浏览器占有了你的网站用户的很大市场份额，编写恰当的代码并仰仗浏览器最终在更好的优化机制上更新自己可能更实际。

“没什么是比暂时的黑科技更永恒的。”你现在为了绕过一些性能的Bug而编写的代码可能要比这个Bug在浏览器中存在的时间长的多。

在那个浏览器每五年才更新一次的年代，这是个很难做的决定。但是如今，所有的浏览器都在快速地更新（虽然移动端的世界还有些滞后），而且它们都在竞争而使得web优化特性变得越来越好。

如果你真的碰到了一个浏览器有其他浏览器没有的性能瑕疵，那么就确保用你一切可用的手段来报告它。绝大多数浏览器都有为此而公开的Bug追迹系统。

提示：我只建议，如果一个在某种浏览器中的性能问题真的是极端搅局的问题时才绕过它，而不是仅仅因为它使人厌烦或沮丧。而且我会非常小心地检查这种性能黑科技有没有在其他浏览器中产生负面影响。

## 大局

与担心所有这些微观性能的细节相反，我们应但关注大局类型的优化。

你怎么知道什么东西是不是大局的？你首先必须理解你的代码是否运行在关键路径上。如果它没在关键路径上，你的优化可能就没有太大价值。

“这是过早的优化！”你听过这种训诫吗？它源自Donald Knuth的一段著名的话：“过早的优化是万恶之源。”。许多开发者都引用这段话来说明大多数优化都是“过早”的而且是一种精力的浪费。事实是，像往常一样，更加微妙。

这是Knuth在语境中的原话：

程序员们浪费了大量的时间考虑，或者担心，他们的程序中的不关键部分的速度，而在考虑调试和维护时这些在效率上的企图实际上有很强大的负面影响。我们应当忘记微小的效率，可以说在大概97%的情况下：过早的优化是万恶之源。然而我们不应该忽略那关键的3%中的机会。[强调]

([http://web.archive.org/web/20130731202547/http://pplab.snu.ac.kr/courses/adv\\_pl05/papers/p261-knuth.pdf](http://web.archive.org/web/20130731202547/http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf), Computing Surveys, Vol 6, No 4, December 1974)

我相信这样转述Knuth的意思是合理的：“非关键路径的优化是万恶之源。”所以问题的关键是弄清楚你的代码是否在关键路径上——你因该优化它！——或者不。

我甚至可以激进地这么说：没有花在优化关键路径上的时间是浪费的，不管它的效果多么微小。没有花在优化非关键路径上的时间是合理的，不管它的效果多大。

如果你的代码在关键路径上，比如将要一次又一次被运行的“热”代码块儿，或者在用户将要注意到的UX关键位置，比如循环动画或者CSS样式更新，那么你应当不遗余力地进行有意义的，可测量的重大优化。

举个例子，考虑一个动画循环的关键路径，它需要将一个字符串值转换为一个数字。这当然有多种方法做到，但是哪一个是最快的呢？

```

var x = "42";      // 需要数字 `42` 

// 选择1：让隐式强制转换自动完成工作
var y = x / 2;

// 选择2：使用`parseInt(..)`
var y = parseInt( x, 0 ) / 2;

// 选择3：使用`Number(..)`
var y = Number( x ) / 2;

// 选择4：使用`+`二元操作符
var y = +x / 2;

// 选择5：使用`|`二元操作符
var y = (x | 0) / 2;

```

注意：我将这个问题留作给读者们的练习，如果你对这些选择之间性能上的微小区别感兴趣的话，可以做一个测试。

当你考虑这些不同的选择时，就像人们说的，“有一个和其他的不一样。”`parseInt(..)` 可以工作，但它做的事情多的多——它会解析字符串而不是转换它。你可能会正确地猜想 `parseInt(..)` 是一个更慢的选择，而你可能应当避免使用它。

当然，如果 `x` 可能是一个需要被解析的值，比如 `"42px"`（比如CSS样式查询），那么 `parseInt(..)` 确实是唯一合适的选择！

`Number(..)` 也是一个函数调用。从行为的角度讲，它与 `+` 二元操作符是相同的，但它事实上可能慢一点儿，需要更多的机器指令运转来执行这个函数。当然，JS引擎也可能识别出了这种行为上的对称性，而仅仅为你处理 `Number(..)` 行为的内联形式（也就是 `+x`）！

但是要记住，痴迷于 `+x` 和 `x | 0` 的比较在大多数情况下都是浪费精力。这是一个微观性能问题，而且你不应该让它使你的程序的可读性降低。

虽然你的程序的关键路径性能非常重要，但它不是唯一的因素。在几种性能上大体相似的选择中，可读性应当是另一个重要的考量。

## 尾部调用优化 (TCO)

正如我们早前简单提到的，ES6包含了一个冒险进入性能世界的具体需求。它是关于在函数调用时可能发生的一种具体的优化形式：尾部调用优化 (TCO)。

简单地说，一个“尾部调用”是一个出现在另一个函数“尾部”的函数调用，于是在这个调用完成后，就没有其他的事情要做了（除了也许要返回结果值）。

例如，这是一个带有尾部调用的非递归形式：

```

function foo(x) {
    return x;
}

function bar(y) {
    return foo( y + 1 );      // 尾部调用
}

function baz() {
    return 1 + bar( 40 );    // 不是尾部调用
}

baz();                      // 42

```

`foo(y+1)` 是一个在 `bar(..)` 中的尾部调用，因为在 `foo(..)` 完成之后，`bar(..)` 也即而完成，除了在这里需要返回 `foo(..)` 调用的结果。然而，`bar(40)` 不是一个尾部调用，因为在它完成后，在 `baz()` 能返回它的结果前，这个结果必须被加1。

不过于深入本质细节而简单地说，调用一个新函数需要保留额外的内存来管理调用栈，它称为一个“栈帧（stack frame）”。所以前面的代码段通常需要同时为 `baz()`，`bar(..)`，和 `foo(..)` 都准备一个栈帧。

然而，如果一个支持TCO的引擎可以认识到 `foo(y+1)` 调用位于尾部位置 意味着 `bar(..)` 基本上完成了，那么当调用 `foo(..)` 时，它就并没有必要创建一个新的栈帧，而是可以重复利用既存的 `bar(..)` 的栈帧。这不仅更快，而且也更节省内存。

在一个简单的代码段中，这种优化机制没什么大不了的，但是当对付递归，特别是当递归会造成成百上千的栈帧时，它就变成了相当有用的技术。引擎可以使用TCO在一个栈帧内完成所有调用！

在JS中递归是一个令人不安的话题，因为没有TCO，引擎就不得不实现一个随意的（而且各不相同的）限制，规定它们允许递归栈能有多深，来防止内存耗尽。使用TCO，带有尾部位置调用的递归函数实质上可以没有边界地运行，因为从没有额外的内存使用！

考虑前面的递归 `factorial(..)`，但是将它重写为对TCO友好的：

```

function factorial(n) {
    function fact(n,res) {
        if (n < 2) return res;

        return fact( n - 1, n * res );
    }

    return fact( n, 1 );
}

factorial( 5 );           // 120

```

这个版本的 `factorial(..)` 仍然是递归的，而且它还是可以进行TCO优化的，因为两个内部的 `fact(..)` 调用都在尾部位置。

注意：一个需要注意的重点是，TCO尽在尾部调用实际存在时才会实施。如果你没用尾部调用编写递归函数，性能机制将仍然退回到普通的栈帧分配，而且引擎对于这样的递归的调用栈限制依然有效。许多递归函数可以像我们刚刚展示的 `factorial(..)` 那样重写，但是要小心处理细节。

ES6要求各个引擎实现TCO而不是留给它们自行考虑的原因之一是，由于对调用栈限制的恐惧，缺少TCO实际上趋向于减少特定的算法在JS中使用递归实现的机会。

如果无论什么情况下引擎缺少TCO只是安静地退化到性能差一些的方式上，那么它可能不会是ES6需要要求的东西。但是因为缺乏TCO可能会实际上使特定的程序不现实，所以与其说它只是一种隐藏的实现细节，不如说它是一个重要的语言特性更合适。

ES6保证，从现在开始，JS开发者们能够在所有兼容ES6+的浏览器上信赖这种优化机制。这是JS性能的一个胜利！

## 复习

有效地对一段代码进行性能基准分析，特别是将它与同样代码的另一种写法相比较来看哪一种方式更快，需要小心地关注细节。

与其运行你自己的统计学上合法的基准分析逻辑，不如使用Benchmark.js库，它会为你搞定。但要小心你如何编写测试，因为太容易构建一个看起来合法但实际上有漏洞的测试了——即使是一个微小的区别也会使结果歪曲到完全不可靠。

尽可能多地从不同的环境中得到尽可能多的测试结果来消除硬件/设备偏差很重要。

jsPerf.com是一个用于大众外包性能基准分析测试的神奇网站。

许多常见的性能测试不幸地痴迷于无关紧要的微观性能细节，比如比较 `x++` 和 `++x`。编写好的测试意味着理解如何聚焦大局上关注的问题，比如在关键路径上优化，和避免落入不同JS引擎的实现细节的陷阱。

尾部调用优化（TCO）是一个ES6要求的优化机制，它会使一些以前在JS中不可能的递归模式变得可能。TCO允许一个位于另一个函数的尾部位置的函数调用不需要额外的资源就可以执行，这意味着引擎不再需要对递归算法的调用栈深度设置一个随意的限制了。

# 你不懂JS：异步与性能

## 附录A: **asynquence** 库

第一章和第二章相当详细地探讨了常见的异步编程模式，以及如何通过回调解决它们。但我们也看到了为什么回调在处理能力上有着致命的缺陷，这将我们带到了第三章和第四章，Promise 与 Generator 为你的异步流程构建提供了一个更加坚实，可信，以及可推理的基础。

我在这本书中好几次提到我自己的异步库 **asynquence** (<http://github.com/getify/asynquence>) —— “`async`” + “`sequence`” = “`asynquence`”，现在我想简要讲解一下它的工作原理，以及它的独特设计为什么很重要和很有用。

在下一篇附录中，我们将要探索一些高级的异步模式，但为了它们的可用性能够使人接受你可能需要一个库。我们将使用 **asynquence** 来表达这些模式，所以你会想首先在这里花一点时间来了解这个库。

**asynquence** 绝对不是优秀异步编码的唯一选择；在这方面当然有许多了不起的库。但是 **asynquence** 提供了一种独特的视角——通过将这些模式中最好的部分组合进一个单独的库，另外它基于一个基本的抽象：（异步）序列。

我的前提是，精巧的JS程序经常或多或少地需要将各种不同的异步模式交织在一起，而且这通常是完全依靠每个开发者自己去搞清楚的。与其引入关注于异步流程的不同方面的两个或更多的库，**asynquence** 将它们统一为各种序列步骤，成为单独一个需要学习和部署的核心库。

我相信 **asynquence** 有足够高的价值可以使 Promise 风格的异步流程控制编程变得超级容易完成，这就是我们为什么会在那里单单关注这个库。

开始之前，我将讲解 **asynquence** 背后的设计原则，然后我们将使用代码示例来展示它的API如何工作。

## 序列，抽象设计

对 **asynquence** 的理解开始于对一个基础抽象的理解：对于一个任务的任何一系列步骤来说，无论它们是同步的还是异步的，都可以被综合地考虑为一个“序列（`sequence`）”。换句话说，一个序列是一个容器，它代表一个任务，并由一个个完成这个任务的独立的（可能是异步的）步骤组成。

在这个序列中的每一个步骤都处于一个 `Promise`（见第三章）的控制之下。也就是你向一个序列添加的每一个步骤都隐含地创建了一个 `Promise`，它被链接到这个序列的末尾。由于 `Promise` 的语义，在一个序列中的每一个步骤的推进都是异步的，即使你同步地完成这个步骤。

另外，一个序列将总是一步步线性地进行，也就是步骤2总是发生在步骤1完成之后，如此类推。

当然，一个新的序列可以从既存的序列中分支出来，也就是分支仅在主序列在流程中到达那一点时发生。序列还可以用各种方式组合，包括使一个序列在流程中的一个特定的位置汇合另一个序列。

一个序列与 `Promise` 链有些相像。但是，在 `Promise` 链中，不存在一个可以引用整个链条的“把手”可以抓住。不管你持有哪一个 `Promise` 的引用，它都表示链条中当前的步骤外加挂载在它后面的其他步骤。实质上，你无法持有一个 `Promise` 链条的引用，除非你持有链条中第一个 `Promise` 的引用。

许多情况表明，持有一个综合地指向整个序列的引用是十分有用的。这些情况中最重要的一种就是序列的退出/取消。正如我们在第三章中展开谈过的那样，`Promise` 本身绝不应当是可以取消的，因为这违反了一个基本设计规则：外部不可变性。

但是序列没有这样的不可变性设计原则，这主要是由于序列不会作为需要不可变语义的未来值的容器被传递。所以序列是一个处理退出/取消行为的恰当的抽象层面。`asynquence` 序列可以在任何时候 `abort()`，而且这个序列将会停止在那一点而不会因为任何原因继续下去。

为了流程控制，还有许多理由首选序列的抽象而非 `Promise` 链。

首先，`Promise` 链是一个更加手动的处理——一旦你开始在你的程序中大面积地创建和链接 `Promise`，这种处理可能会变得相当烦冗——在那些使用 `Promise` 相当恰当的地方，这种烦冗会降低效率而使得开发者不愿使用 `Promise`。

抽象意味着减少模板代码和烦冗，所以序列抽象是这个问题的一个好的解决方案。使用 `Promise`，你关注的是个别的步骤，而且不太会假定你将延续这个链条。而序列采用相反的方式，它假定序列将会无限地持续添加更多步骤。

当你开始考虑更高阶的 `Promise` 模式时（除了 `race(..)` 和 `all(..)` 以外），这种抽象复杂性的降低特别强大。

例如，在一个序列的中间，你可能想表达一个在概念上类似于 `try..catch` 的步骤，它的结果将总是成功，不管是意料之中的主线上的成功解析，还是为被捕获的错误提供一个正面的非错误信号。或者，你可能想表达一个类似于 `retry/until` 循环的步骤，它不停地尝试相同的步骤直到成功为止。

仅仅使用基本的 `Promise`，这类抽象不是很容易表达，而且在一个既存的 `Promise` 链的中间这样做不好看。但如果你将你的想法抽象为一个序列，并将一个步骤考虑为一个 `Promise` 的包装，这个包装可以隐藏这样的细节，它就可以使你以最合理的方式考虑流程控制，而不必关心细节。

第二，也许是更重要的，将异步流程控制考虑为一个序列中的步骤，允许你将这样的细节抽象出去——每一个步骤中引入了哪一种异步性。在这种抽象之下，一个 `Promise` 将总是控制着步骤，但在抽象之上，这个步骤可以看起来像一个延续回调（简单的默认值），或者一个真正的 `Promise`，或者一个运行至完成的 `Generator`，或者... 希望你明白我的意思。

第三，序列可以通容易地被调整来适应于不同的思考模式，比如基于事件的，基于流的，或者基于相应式的编码。`asynquence` 提供了一种我称为“响应式序列”的模式（我们稍后讲解），它是 RxJS (“Reactive Extensions”) 中“响应式可监听”思想的变种，允许重复的事件每次触发一个新的序列实例。`Promise` 是一次性的，所以单独使用 `Promise` 来表达重复的异步性十分尴尬。

在一种我称为“可迭代序列”的模式中，另一种思考模式反转了解析/控制能力。与每一个步骤在内部控制它自己的完成（并因此推进这个序列）不同，序列被反转为通过一个外部迭代器来进行推进控制，而且在这个可迭代序列中的每一步仅仅应答 `next(..)` 迭代器 控制。

在本附录的剩余部分，我们将探索所有这些不同的种类，所以如果我们刚才的步伐太快也不要担心。

要点是，对于复杂的异步处理来说，序列是一个要比单纯的 `Promise` (`Promise`链) 或单纯的 `Generator` 更加强大与合理的抽象，而 `asynquence` 被设计为使用恰当层面的语法糖来表达这种抽象，使得异步编程变得更加易于理解和更加令人愉快。

## asynquence API

首先，你创建一个序列（一个 `asynquence` 实例）的方法是使用 `ASQ(..)` 函数。一个不带参数的 `ASQ()` 调用会创建一个空的初始序列，而向 `ASQ(..)` 传递一个或多个值或函数的话，它会使用每个参数值代表序列的初始步骤来创建序列。

注意：为了这里所有的代码示例，我将使用 `asynquence` 在浏览器全局作用域中的顶层标识符：`ASQ`。如果你通过一个模块系统（在浏览器或服务器中）引入并使用 `asynquence`，你当然可以定义自己喜欢的符号，`asynquence` 不会关心这些！

许多在这里讨论的API方法都内建于 `asynquence` 的核心部分，而其他的API是通过引入可选的“contrib”插件包提供的。要知道一个方法是内建的还是通过插件定义的，可以参见 `asynquence` 的文档：<http://github.com/getify/asynquence>

### 步骤

如果一个函数代表序列中的一个普通步骤，那么这个函数会被这样调用：第一个参数是延续回调，而任何后续参数都是从前一个步骤中传递下来的消息。在延续回调被调用之前，这个步骤将不会完成。一旦延续回调被调用，你传递给它的任何参数值都会作为序列下一个步骤中的消息被发送。

要向一个序列添加额外的普通步骤，调用 `then(..)`（它实质上与 `ASQ(..)` 调用的语义完全相同）：

```
ASQ(
  // 步骤 1
  function(done){
    setTimeout( function(){
      done( "Hello" );
    }, 100 );
  },
  // 步骤 2
  function(done,greeting) {
    setTimeout( function(){
      done( greeting + " World" );
    }, 100 );
  }
)
// 步骤 3
.then( function(done,msg){
  setTimeout( function(){
    done( msg.toUpperCase() );
  }, 100 );
} )
// 步骤 4
.then( function(done,msg){
  console.log( msg );           // HELLO WORLD
} );
```

注意：虽然 `then(..)` 这个名称与原生的 Promise API 完全一样，但是这个 `then(..)` 的含义是不同的。你可以传递任意多或者任意少的函数或值给 `then(..)`，而它们中的每一个都被看作是一个分离的步骤。这里与完成/拒绝语义的双回调毫不相干。

在 Promise 中，可以把一个 Promise 与下一个你在 `then(..)` 的完成处理器中创建并 `return` 的 Promise 链接。与此不同的是，在 `asynquence` 中，你所需要做的一切就是调用延续回调——我总是称之为 `done()`，但你可以起任何适合你的名字——并将完成的消息作为参数值选择性地传递给它。

通过 `then(..)` 定义的每一个步骤都被认为是异步的。如果你有一个同步的步骤，你可以立即调用 `done(..)`，或者使用更简单的 `val(..)` 步骤帮助函数：

```
// 步骤 1 (同步)
ASQ( function(done){
    done( "Hello" );      // 手动同步
} )
// 步骤 2 (同步)
.val( function(greeting){
    return greeting + " World";
} )
// 步骤 3 (异步)
.then( function(done,msg){
    setTimeout( function(){
        done( msg.toUpperCase() );
    }, 100 );
} )
// 步骤 4 (同步)
.val( function(msg){
    console.log( msg );
} );
```

如你所见，`val(..)` 调用的步骤不会收到一个延续回调，因为这部分已经为你做好了——而且参数列表作为一个结果显得不那么凌乱了！要向下一个步骤发送消息，你简单地使用 `return`。

将 `val(..)` 考虑为表示一个同步的“仅含有值”的步骤，它对同步的值操作，比如 `logging` 之类，非常有用。

## 错误

与 `Promise` 相比 `asynquence` 的一个重要的不同之处是错误处理。

在 `Promise` 链条中，每个 `Promise`（步骤）都可以拥有自己独立的错误，而每个后续的步骤都有能力处理或不处理这个错误。这种语义（再一次）主要来自于对每个单独的 `Promise` 的关注，而非对整个链条（序列）的关注。

我相信，在大多数情况下，一个位于序列中某一部分的错误通常是不可恢复的，所以序列中后续的步骤毫无意义而应当被跳过。所以，默认情况下，在一个序列的任意一个步骤中的错误会将整个序列置于错误模式，而剩下的普通步骤将会被忽略。

如果你确实需要一个错误可以被恢复的步骤，有几个不同的API可以适应这种情况，比如 `try(..)` —— 先前提到过的，有些像 `try..catch` 的步骤 —— 或者 `until(..)` —— 一个重试循环，它持续地尝试一个步骤直到它成功或你手动地 `break()` 这个循环。`asynquence` 甚至拥有 `pThen(..)` 和 `pCatch(..)` 方法，它们的工作方式与普通的 `Promise` 的 `then(..)` 和 `catch(..)`（见第三章）完全相同，所以如果你选择这么做，你就可以进行本地化的序列中错误处理。

重点是，你同时拥有两个选项，但是在我的经验中更常见的是默认情况。使用 `Promise`，要使一个步骤的链条在错误发生时一次性忽略所有步骤，你不得不小心不要在任何步骤中注册拒绝处理器；否则，这个错误会被视为处理过而被吞掉，而序列可能仍会继续下去（也许不是意料之中的）。要恰当且可靠地处理这种期待的行为有点儿尴尬。

要注册一个序列错误通知处理器，`asynquence` 提供了一个 `or(..)` 序列方法，它还有一个别名叫做 `onerror(..)`。你可以在序列的任何位置调用这个方法，而且你可以注册任意多的处理器。这使得让多个不同的消费者监听一个序列是否失败变得很容易；从这个角度讲，它有点儿像一个错误事件处理器。

正如使用 `Promise` 那样，所有JS异常都会变为序列错误，或者你可以通过编程来发生一个序列错误：

```
var sq = ASQ( function(done){
    setTimeout( function(){
        // 为序列发出一个错误
        done.fail( "Oops" );
    }, 100 );
} )
.then( function(done){
    // 永远不会到达这里
} )
.or( function(err){
    console.log( err );           // Oops
} )
.then( function(done){
    // 也不会到达这里
} );
// 稍后

sq.or( function(err){
    console.log( err );           // Oops
} );
```

`asynquence` 与原生的 `Promise` 相比，在错误处理上另一个重要的不同就是“未处理异常”的默认行为。正如我们在第三章中以相当的篇幅讨论过的，一个没有被注册拒绝处理器的 `Promise` 如果被拒绝的话，将会无声地保持（也就是吞掉）那个错误；你不得不总是想着要用一个最后的 `catch(..)` 来终结一个链条。

在 `asynquence` 中，这种假设被颠倒过来了。

如果一个错误在序列上发生，而且 在那个时刻 它没有被注册错误处理器，那么这个错误会被报告至 `console`。换言之，未处理的的拒绝将总是默认地被报告，因此不会被吞掉或丢掉。

为了防止重复的噪音，只要你向一个序列注册一个错误处理器，它就会使这个序列从这样的报告中退出。

事实上有许多情况你想要创建这样一个序列，它可能会在你有机会注册处理器之前就进入错误状态。这不常见，但可能时不时地发生。

在这样的情况下，你也可以通过在序列上调用 `defer()` 来使一个序列实例从错误报告中退出。你应当仅在自己确信不会最终处理这样的错误时，才决定从报告中退出：

```
var sq1 = ASQ( function(done){
    doesnt.Exist();           // 将会向控制台抛出异常
} );

var sq2 = ASQ( function(done){
    doesnt.Exist();           // 仅仅会抛出一个序列错误
} )
// 错误报告中的退出
.defer();

setTimeout( function(){
    sq1.or( function(err){
        console.log( err );    // ReferenceError
    } );
    sq2.or( function(err){
        console.log( err );    // ReferenceError
    } );
}, 100 );
// ReferenceError (来自sq1)
```

这是一种比 `Promise` 本身拥有的更好的错误处理行为，因为它是一个成功的深渊，而不是一个失败的深渊（参见第三章）。

注意：如果一个序列被导入（也就是被汇合入）另一个序列——完整的描述参见“组合序列”——之后源序列从错误报告中退出，那么就必须考虑目标序列是否进行错误报告。

## 并行步骤

在你的序列中不是所有的步骤都将只拥有一个（异步）任务去执行；有些将会需要“并行”（并发地）执行多个步骤。在一个序列中，一个并发地处理多个子步骤的步骤称为一个 `gate(..)`——如果你喜欢的话它还有一个别名 `all(..)`——而且它与原生的 `Promise.all([..])` 是对称的。

如果在 `gate(..)` 中的所有步骤都成功地完成了，那么所有成功的消息都将被传递到下一个序列步骤中。如果它们中的任何一个产生了一个错误，那么整个序列会立即进入错误状态。

考虑如下代码：

```

ASQ( function(done){
    setTimeout( done, 100 );
} )
.gate(
    function(done){
        setTimeout( function(){
            done( "Hello" );
        }, 100 );
    },
    function(done){
        setTimeout( function(){
            done( "World", "!" );
        }, 100 );
    }
)
.val( function(msg1,msg2){
    console.log( msg1 ); // Hello
    console.log( msg2 ); // [ "World", "!" ]
} );

```

为了展示差异，让我们把这个例子与原生 Promise 比较一下：

```

new Promise( function(resolve,reject){
    setTimeout( resolve, 100 );
} )
.then( function(){
    return Promise.all( [
        new Promise( function(resolve,reject){
            setTimeout( function(){
                resolve( "Hello" );
            }, 100 );
        } ),
        new Promise( function(resolve,reject){
            setTimeout( function(){
                // 注意：这里我们需要一个 []
                resolve( [ "World", "!" ] );
            }, 100 );
        } )
    ] );
} )
.then( function(msgs){
    console.log( msgs[0] ); // Hello
    console.log( msgs[1] ); // [ "World", "!" ]
} );

```

讨厌。Promise 需要多得多的模板代码来表达相同的异步流程控制。这个例子很好地说明了为什么 *asynquence* API 和抽象使得对付 Promise 步骤容易多了。你的异步流程越复杂，它的改进程度就越高。

## 各种步骤

关于 `asynquence` 的 `gate(..)` 步骤类型，有好几种不同的 `contrib` 插件可能十分有用：

- `any(..)` 很像 `gate(..)`，除了为了继续主序列，只需要有一个环节最终必须成功。
- `first(..)` 很像 `any(..)`，除了只要有任何一个环节成功，主序列就会继续（忽略任何其余环节产生的后续结果）。
- `race(..)`（与 `Promise.race(..)` 对称）很像 `first(..)`，除了主序列会在任何环节完成时（不管成功还是失败）立即继续。
- `last(..)` 很像 `any(..)`，除了只有最后一个环节成功完成时才会把它的消息发送给主序列。
- `none(..)` 是 `gate(..)` 的反义：主序列仅在所有环节失败时才会继续（将所有环节的错误消息作为成功消息传递，或者反之）。

让我们首先定义一些帮助函数来使示例清晰一些：

```
function success1(done) {
  setTimeout( function(){
    done( 1 );
  }, 100 );
}

function success2(done) {
  setTimeout( function(){
    done( 2 );
  }, 100 );
}

function failure3(done) {
  setTimeout( function(){
    done.fail( 3 );
  }, 100 );
}

function output(msg) {
  console.log( msg );
}
```

现在，让我们展示一些这些 `gate(..)` 步骤的变种：

```

ASQ().race(
  failure3,
  success1
)
.or( output );           // 3

ASQ().any(
  success1,
  failure3,
  success2
)
.val( function(){
  var args = [].slice.call( arguments );
  console.log(
    args      // [ 1, undefined, 2 ]
  );
} );
}

ASQ().first(
  failure3,
  success1,
  success2
)
.val( output );           // 1

ASQ().last(
  failure3,
  success1,
  success2
)
.val( output );           // 2

ASQ().none(
  failure3
)
.val( output )           // 3
.none(
  failure3
  success1
)
.or( output );           // 1

```

另一个步骤种类是 `map(..)`，它让你将一个数组的元素异步地映射为不同的值，而且在所有映射完成之前步骤不会前进。`map(..)` 与 `gate(..)` 十分相似，除了它从一个数组，而非从一个指定的分离函数那里得到初始值，而且你定义一个函数回调来操作每一个值：

```

function double(x,done) {
    setTimeout( function(){
        done( x * 2 );
    }, 100 );
}

ASQ().map( [1,2,3], double )
.val( output );           // [2,4,6]

```

另外，`map(..)` 可以前一步骤传递来的消息中收到它的两个参数（数组或者回调）：

```

function plusOne(x,done) {
    setTimeout( function(){
        done( x + 1 );
    }, 100 );
}

ASQ( [1,2,3] )
.map( double )           // 收到消息`[1,2,3]`
.map( plusOne )          // 收到消息`[2,4,6]`
.val( output );          // [3,5,7]

```

另一个种类是 `waterfall(..)`，它有些像混合了 `gate(..)` 的消息收集行为与 `then(..)` 的序列化处理。

步骤1首先被执行，然后来自步骤1的成功消息被传递给步骤2，然后两个成功消息走到步骤3，然后所有三个成功消息走到步骤4，如此继续，这样消息被某种程度上收集并从“瀑布”上倾泻而下。

考虑如下代码：

```

function double(done) {
  var args = [].slice.call( arguments, 1 );
  console.log( args );

  setTimeout( function(){
    done( args[args.length - 1] * 2 );
  }, 100 );
}

ASQ( 3 )
.waterval(
  double,           // [ 3 ]
  double,           // [ 6 ]
  double,           // [ 6, 12 ]
  double           // [ 6, 12, 24 ]
)
.val( function(){
  var args = [].slice.call( arguments );
  console.log( args ); // [ 6, 12, 24, 48 ]
} );

```

如果在“瀑布”的任何一点发生错误，那么整个序列就会立即进入错误状态。

## 容错

有时你想在步骤一级管理错误，而不一定让它们使整个序列成为错误状态。`asynquence` 为此提供了两种步骤类型。

`try(..)` 尝试一个步骤，如果它成功，序列就会正常继续，但如果这个步骤失败了，失败的状态会转换成格式为 `{ catch: ... }` 的成功消息，它的值由错误消息填充：

```

ASQ()
.try( success1 )
.val( output )           // 1
.try( failure3 )
.val( output )           // { catch: 3 }
.or( function(err){
  // 永远不会到达这里
} );

```

你还可以使用 `until(..)` 构建一个重试循环，它尝试一个步骤，如果失败，就会在下一个事件轮询的 `tick` 中重试这个步骤，如此继续。

这种重试循环可以无限延续下去，但如果你想要从循环中跳出来，你可以在完成触发器上调用 `break()` 标志方法，它将主序列置为错误状态：

```

var count = 0;

ASQ( 3 )
.until( double )
.val( output ) // 6
.until( function(done){
    count++;

    setTimeout( function(){
        if (count < 5) {
            done.fail();
        }
        else {
            // 跳出 `until(..)` 重试循环
            done.break( "Oops" );
        }
    }, 100 );
} )
.or( output ); // Oops

```

## Promise 式的步骤

如果你喜欢在你的序列中内联 Promise 风格的语义，比如 Promise 的 `then(..)` 和 `catch(..)`（见第三章），你可以使用 `pThen` 和 `pCatch` 插件：

```

ASQ( 21 )
.pThen( function(msg){
    return msg * 2;
} )
.pThen( output ) // 42
.pThen( function(){
    // 抛出一个异常
    doesnt.Exist();
} )
.pCatch( function(err){
    // 捕获这个异常（拒绝）
    console.log( err ); // ReferenceError
} )
.val( function(){
    // 主旋律回到正常状态，
    // 因为前一个异常已经被
    // `pCatch(..)` 捕获了
} );

```

`pThen(..)` 和 `pCatch(..)` 被设计为运行在序列中，但好像在普通的 Promise 链中动作。这样，你就可以在传递给 `pThen(..)` 的“完成”处理器中解析纯粹的 Promise 或者 `asynquence` 序列。

## 序列分支

一个有关 `Promise` 的可能十分有用的特性是，你可以在同一个 `Promise` 上添附多个 `then(...)` 处理器，这实质上在这个 `Promise` 的流程上创建了“分支”：

```
var p = Promise.resolve( 21 );

// (从`p`开始的) 分支 1
p.then( function(msg){
  return msg * 2;
} )
.then( function(msg){
  console.log( msg );           // 42
} )

// (从`p`开始的) 分支 2
p.then( function(msg){
  console.log( msg );           // 21
} );
```

使用 `asynquence` 的 `fork()` 可以很容易地进行同样的“分支”：

```
var sq = ASQ(...).then(...).then(...);

var sq2 = sq.fork();

// 分支 1
sq.then(...);

// 分支 2
sq2.then(...);
```

## 组合序列

与 `fork()` 相反的是，你可以通过将一个序列汇合进另一个来组合两个序列，使用 `seq(..)` 实例方法：

```

var sq = ASQ( function(done){
    setTimeout( function(){
        done( "Hello World" );
    }, 200 );
} );

ASQ( function(done){
    setTimeout( done, 100 );
} )
// 将序列 `sq` 汇合进这个系列
.seq( sq )
.val( function(msg){
    console.log( msg );           // Hello World
} )

```

`seq(..)` 可以像这里展示的那样接收一个序列本身，或者接收一个函数。如果是一个函数，那么它会期待这个函数被调用时返回一个序列，所以前面的代码可以这样写：

```

// ...
.seq( function(){
    return sq;
} )
// ...

```

另外，这个步骤还可以使用 `pipe(..)` 来完成：

```

// ...
.then( function(done){
    // 将 `sq` 导入延续回调 `done`
    sq.pipe( done );
} )
// ...

```

当一个序列被汇合时，它的成功消息流和错误消息流都会被导入。

注意：正如早先的注意事项中提到过的，导入会使源序列从错误报告中退出，但不会影响目标序列的错误报告状态。

## 值与错误序列

如果一个序列的任意一个步骤只是一个普通值，那么这个值就会被映射到这个步骤的完成消息中：

```
var sq = ASQ( 42 );

sq.val( function(msg){
    console.log( msg );           // 42
} );
```

如果你想制造一个自动出错的序列：

```
var sq = ASQ.failed( "Oops" );

ASQ()
.seq( sq )
.val( function(msg){
    // 不会到达这里
} )
.or( function(err){
    console.log( err );          // Oops
} );
```

你还可能想要自动地创建一个延迟的值或者延迟的错误序列。使用 `after` 和 `failAfter` `contrib` 插件，这很容易：

```
var sq1 = ASQ.after( 100, "Hello", "World" );
var sq2 = ASQ.failAfter( 100, "Oops" );

sq1.val( function(msg1,msg2){
    console.log( msg1, msg2 );      // Hello World
} );

sq2.or( function(err){
    console.log( err );            // Oops
} );
```

你还可以使用 `after'(..)` 在一个序列的中间插入一个延迟：

```
ASQ( 42 )
// 在这个序列中插入一个延迟
.after( 100 )
.val( function(msg){
    console.log( msg );           // 42
} );
```

## Promises 与回调

我认为 `asynquence` 序列在原生的 `Promise` 之上提供了许多价值，而且你会发现在很大程度上它在抽象层面上使用起来更舒适更强大。然而，将 `asynquence` 与其他非 `asynquence` 代码进行整合将是不可避免的现实。

使用 `promise(..)` 实例方法，你可以很容易地将一个 `Promise`（也就是 `thenable` —— 见第三章）汇合进一个序列：

```
var p = Promise.resolve( 42 );

ASQ()
.promise( p )           // 本可以写做：`function(){ return p; }`
.val( function(msg){
    console.log( msg );   // 42
} );
```

要向相反的方向走，从一个序列的特定步骤中分支/出让一个 `Promise`，使用 `toPromise` `contrib` 插件：

```
var sq = ASQ.after( 100, "Hello World" );

sq.toPromise()
// 现在这是一个标准的 promise 链了
.then( function(msg){
    return msg.toUpperCase();
} )
.then( function(msg){
    console.log( msg );      // HELLO WORLD
} );
```

有好几种帮助设施可以在使用回调的系统中适配 `asynquence`。要从你的序列中自动地生成一个“错误优先风格”回调，来接入一个面向回调的工具，使用 `errfcb`：

```
var sq = ASQ( function(done){
    // 注意：这里期待“错误优先风格”的回调
    someAsyncFuncWithCB( 1, 2, done.errfcb )
} )
.val( function(msg){
    // ..
} )
.or( function(err){
    // ..
} );

// 注意：这里期待“错误优先风格”的回调
anotherAsyncFuncWithCB( 1, 2, sq.errfcb() );
```

你还可能想要创建一个工具的序列包装版本——与第三章的“promisory”和第四章的“thunkory”相比较——**asynquence** 为此提供了 `ASQ.wrap(..)`：

```
var coolUtility = ASQ.wrap( someAsyncFuncWithCB );

coolUtility( 1, 2 )
  .val( function(msg){
    // ..
  } )
  .or( function(err){
    // ..
  } );
}
```

注意：为了清晰（和有趣！），让我们为来自 `ASQ.wrap(..)` 的产生序列的函数杜撰另一个名词，就像这里的 `coolUtility`。我提议“**sequory**”（“sequence” + “factory”）。

## 可迭代序列

一个序列普通的范例是，每一个步骤都负责完成它自己，进而推进这个序列。Promise 就是这样工作的。

不幸的是，有时你需要从外部控制一个 Promise/步骤，而这会导致尴尬的“能力抽取”。

考虑这个 Promise 的例子：

```
var domready = new Promise( function(resolve,reject){
  // 不想把这个放在这里，因为在逻辑上
  // 它属于代码的另一部分
  document.addEventListener( "DOMContentLoaded", resolve );
} );

// ...

domready.then( function(){
  // DOM 准备好了！
} );
}
```

关于 Promise 的“能力抽取”范模式看起来像这样：

```

var ready;

var domready = new Promise( function(resolve,reject){
    // 抽取 `resolve()` 能力
    ready = resolve;
} );

// ...

domready.then( function(){
    // DOM 准备好了 !
} );

// ...

document.addEventListener( "DOMContentLoaded", ready );

```

注意：在我看来，这种反模式是一种尴尬的代码风格，但有些开发者喜欢，我不能理解其中的原因。

**asynquence** 提供一种我称为“可迭代序列”的反转序列类型，它将控制能力外部化（它在 `domready` 这样的情况下十分有用）：

```

// 注意：这里`domready`是一个控制序列的 *迭代器*
var domready = ASQ.iterable();

// ...

domready.val( function(){
    // DOM 准备好了 !
} );

// ...

document.addEventListener( "DOMContentLoaded", domready.next );

```

与我们在这个场景中看到的东西比起来，可迭代序列还有很多内容。我们将在附录B中回过头来讨论它们。

## 运行 Generator

在第四章中，我们衍生了一种称为 `run(..)` 的工具，它可以将 `generator` 运行至完成，监听被 `yield` 的 `Promise` 并使用它们来异步推进 `generator`。**asynquence** 正好有一个这样的内建工具，称为 `runner(..)`。

为了展示，然我们首先建立一些帮助函数：

```

function doublePr(x) {
    return new Promise( function(resolve,reject){
        setTimeout( function(){
            resolve( x * 2 );
        }, 100 );
    } );
}

function doubleSeq(x) {
    return ASQ( function(done){
        setTimeout( function(){
            done( x * 2 )
        }, 100 );
    } );
}

```

现在，我们可以在一个序列的中间使用 `runner(..)` 作为一个步骤：

```

ASQ( 10, 11 )
.runner( function*(token){
    var x = token.messages[0] + token.messages[1];

    // yield 一个真正的 promise
    x = yield doublePr( x );

    // yield 一个序列
    x = yield doubleSeq( x );

    return x;
} )
.val( function(msg){
    console.log( msg );           // 84
} );

```

## 包装过的 Generator

你还可以创建自包装的 generator——也就是一个普通函数，运行你指定的 generator 并为它的完成返回一个序列——通过 `ASQ.wrap(..)` 包装它：

```

var foo = ASQ.wrap( function*(token){
  var x = token.messages[0] + token.messages[1];

  // yield 一个真正的 promise
  x = yield doublePr( x );

  // yield 一个序列
  x = yield doubleSeq( x );

  return x;
}, { gen: true } );

// ...

foo( 8, 9 )
.val( function(msg){
  console.log( msg );           // 68
} );

```

`runner(..)` 还能做很多很牛的事情，我们会在附录B中回过头来讨论它。

## 复习

`asynquence` 是一个在 `Promise` 之上的简单抽象——一个序列是一系列（异步）步骤，它的目标是使各种异步模式更加容易使用，而在功能上没有任何妥协。

在 `asynquence` 的核心API与它的 `contrib` 插件中，除了我们在这篇附录中看到的内容以外还有其他的好东西，我们把对这些剩余功能的探索作为练习留给读者。

现在你看到了 `asynquence` 的实质与精神。关键点是，一个序列由许多步骤组成，而这些步骤可以使许多不同种类的 `Promise`，或者它们可以是一个 `generator` 运行器，或者... 选择由你来决定，你有完全的自由为你的任务采用恰当的任何异步流程控制逻辑。

如果你能理解这些 `asynquence` 代码段，那么你现在就可以相当快地学会这个库；它实际上没有那么难学！

如果你依然对它如何（或为什么！）工作感到模糊，那么在进入下一篇附录之前，你将会想要多花一点时间去查看前面的例子，并亲自把玩一下 `asynquence`。附录B将在几种更高级更强大的异步模式中使用 `asynquence`。

# 你不懂JS：异步与性能

## 附录B：高级异步模式

为了了解主要基于 Promise 与 Generator 的面向序列异步流程控制，附录A介绍了 `asynquence` 库。

现在我们将要探索其他建立在既存理解与功能之上的高级异步模式，并看看 `asynquence` 是如何在不需要许多分离的库的情况下，使得这些精巧的异步技术与我们的程序进行混合与匹配的。

### 可迭代序列

我们在前一篇附录中介绍过 `asynquence` 的可迭代序列，我们将更加详细地重温它们。

为了复习，回忆一下：

```
var domready = ASQ.iterable();

// ...

domready.val( function(){
  // DOM 准备好了
} );

// ...

document.addEventListener( "DOMContentLoaded", domready.next );
```

现在，让我们定义将一个多步骤序列定义为一个可迭代序列：

```

var steps = ASQ iterable();

steps
  .then( function STEP1(x){
    return x * 2;
  })
  .then( function STEP2(x){
    return x + 3;
  })
  .then( function STEP3(x){
    return x * 4;
  });

steps.next().value;      // 16
steps.next().value;      // 19
steps.next().value;      // 76
steps.next().done;       // true

```

如你所见，一个可迭代序列是一个标准兼容的 *iterator*（见第四章）。所以，就像一个 generator（或其他任何 可迭代对象）那样，它是可以使用ES6 `for..of` 循环进行迭代的，

```

var steps = ASQ iterable();

steps
  .then( function STEP1(){ return 2; } )
  .then( function STEP2(){ return 4; } )
  .then( function STEP3(){ return 6; } )
  .then( function STEP4(){ return 8; } )
  .then( function STEP5(){ return 10; } );

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

// 2 4 6 8 10

```

除了在前一篇附录中展示的事件触发的例子之外，可迭代序列的有趣之处还因为它们实质上可以被视为 generator 和 Promise 链的替代品，但具备更多灵活性。

考虑一个多 Ajax 请求的例子——我们已经在第三章和第四章中看到过同样的场景，分别使用一个 Promise 链和一个 generator——表达为一个可迭代序列：

```
// 兼容序列的 ajax
var request = ASQ.wrap( ajax );

ASQ( "http://some.url.1" )
.runner(
  ASQ iterable()

  .then( function STEP1(token){
    var url = token.messages[0];
    return request( url );
  } )

  .then( function STEP2(resp){
    return ASQ().gate(
      request( "http://some.url.2/?v=" + resp ),
      request( "http://some.url.3/?v=" + resp )
    );
  } )

  .then( function STEP3(r1,r2){ return r1 + r2; } )
)
.val( function(msg){
  console.log( msg );
} );
```

可迭代序列表达了一系列顺序的（同步的或异步的）步骤，它看起来与一个 `Promise` 链极其相似——换言之，它要比单纯嵌套的回调看起来干净的多，但没有 `generator` 的基于 `yield` 的顺序化语法那么好。

但我们将可迭代序列传入 `ASQ#runner(...)`，它将可迭代序列像一个 `generator` 那样运行至完成。由于几个原因，一个可迭代序列的行为实质上与一个 `generator` 相同的事实是值得注意的：

首先，对于ES6 `generator` 的特定子集来说，可迭代对象是它的一种前ES6等价物，这意味着你既可以直接编写它们（为了在任何地方都能运行），也可以编写ES6 `generator` 并将它们转译/转换成可迭代序列（或者 `Promise` 链！）。

将一个异步运行至完成的 `generator` 考虑为一个 `Promise` 链的语法糖，是对它们之间的同构关系的一种重要认识。

在我们继续之前，我们应当注意到，前一个代码段本可以用 `asynquence` 表达为：

```

ASQ( "http://some.url.1" )
  .seq( /*STEP 1*/ request )
  .seq( function STEP2(resp){
    return ASQ().gate(
      request( "http://some.url.2/?v=" + resp ),
      request( "http://some.url.3/?v=" + resp )
    );
  } )
  .val( function STEP3(r1,r2){ return r1 + r2; } )
  .val( function(msg){
    console.log( msg );
  } );

```

进一步，步骤2本可以被表达为：

```

.gate(
  function STEP2a(done,resp) {
    request( "http://some.url.2/?v=" + resp )
    .pipe( done );
  },
  function STEP2b(done,resp) {
    request( "http://some.url.3/?v=" + resp )
    .pipe( done );
  }
)

```

那么，为什么我们要在一个简单/扁平的 *asyquence* 链看起来可以很好地工作的情况下，很麻烦地将自己的控制流在一个 `ASQ#runner(..)` 步骤中表达为一个可迭代序列呢？

因为可迭代序列的形式有一种重要的技巧可以给我们更多的力量。继续读。

## 扩展可迭代序列

Generator，普通的 *asynquence* 序列，和 Promise 链，都是被急切求值的——控制流程最初要表达的内容就是紧跟在后面的固定流程。

然而，可迭代序列是懒惰求值的，这意味着在可迭代序列执行期间，如果有需要的话你可以用更多的步骤扩展这个序列。

注意：你只能在一个可迭代序列的末尾连接，而不是在序列的中间插入。

为了熟悉这种能力，首先让我们看一个比较简单（同步）的例子：

```

function double(x) {
  x *= 2;

  // 我们应当继续扩展吗？
  if (x < 500) {
    isq.then( double );
  }

  return x;
}

// 建立单步可迭代序列
var isq = ASQ iterable().then( double );

for (var v = 10, ret;
     (ret = isq.next( v )) && !ret.done;
  ) {
  v = ret.value;
  console.log( v );
}

```

这个可迭代序列开始时只有一个定义好的步骤（`isq.then(double)`），但是这个序列会在特定条件下（`x < 500`）持续扩展自己。`asynquence` 序列和 Promise 链在技术上都可以做相似的事情，但是我们将看到它们的这种能力不足的一些原因。

这个例子意义不大，而且本可以使用一个 generator 中的 `while` 循环来表达，所以我们将考虑更精巧的情况。

例如，你可以检查一个 Ajax 请求的应答，看它是否指示需要更多的数据，你可以条件性地向可迭代序列插入更多的步骤来发起更多的请求。或者你可以条件性地在 Ajax 处理器的末尾加入一个格式化步骤。

考虑如下代码：

```

var steps = ASQ iterable()

.then( function STEP1(token){
  var url = token.messages[0].url;

  // 有额外的格式化步骤被提供吗？
  if (token.messages[0].format) {
    steps.then( token.messages[0].format );
  }

  return request( url );
} )

.then( function STEP2(resp){
  // 要为序列增加另一个Ajax请求吗？
  if (/x1/.test( resp )) {
    steps.then( function STEP5(text){
      return request(
        "http://some.url.4/?v=" + text
      );
    } );
  }
}

return ASQ().gate(
  request( "http://some.url.2/?v=" + resp ),
  request( "http://some.url.3/?v=" + resp )
);
} )

.then( function STEP3(r1,r2){ return r1 + r2; } );

```

你可以在两个地方看到我们使用 `steps.then(..)` 条件性地扩展了 `step`。为了运行这个 `steps` 可迭代序列，我们只要使用 `ASQ#runner(..)` 将它与一个 `asynquence` 序列（这里称为 `main`）链接进我们的主程序流程中：

```

var main = ASQ( {
  url: "http://some.url.1",
  format: function STEP4(text){
    return text.toUpperCase();
  }
} )
.runner( steps )
.val( function(msg){
  console.log( msg );
} );

```

`steps` 可迭代序列的灵活性可以使用一个 `generator` 来表达吗？某种意义上可以，但我们不得不以一种有些尴尬的方式重新安排逻辑：

```

function *steps(token) {
    // **步骤 1**
    var resp = yield request( token.messages[0].url );

    // **步骤 2**
    var rvals = yield ASQ().gate(
        request( "http://some.url.2/?v=" + resp ),
        request( "http://some.url.3/?v=" + resp )
    );

    // **步骤 3**
    var text = rvals[0] + rvals[1];

    // **步骤 4**
    // 有额外的格式化步骤被提供吗？
    if (token.messages[0].format) {
        text = yield token.messages[0].format( text );
    }

    // **步骤 5**
    // 要为序列增加另一个Ajax请求吗？
    if (/foobar/.test( resp )) {
        text = yield request(
            "http://some.url.4/?v=" + text
        );
    }

    return text;
}

// 注意：`*steps()` 可以向先前的 `step` 一样被相同的 `ASQ` 序列运行

```

先把我们已经知道的序列的好处，以及看起来同步的 generator 语法（见第四章）放在一边，`steps` 逻辑不得不在 `*steps()` generator 形式中重排，来假冒可扩展的可迭代序列 `steps` 的动态机制。

那么，使用 `Promise` 或者序列如何表达这种功能呢？你可以这么做：

```

var steps = something( ... )
.then( ... )
.then( function(...){
    // ...

    // 扩展这个链条，对吧？
    steps = steps.then( ... );

    // ...
})
.then( ... );

```

这里要抓住的问题很微妙但很重要。那么，考虑试着将我们的 `steps` `Promise` 链连接到我们的主程序流程中——这次使用 `Promise` 代替 `asynquence` 来表达：

```
var main = Promise.resolve( {
  url: "http://some.url.1",
  format: function STEP4(text){
    return text.toUpperCase();
  }
} )
.then( function(..){
  return steps;           // 提示！
} )
.val( function(msg){
  console.log( msg );
} );
```

现在你能发现问题吗？仔细观察！

对于序列步骤的顺序来说，这里有一个竞合状态。当你 `return steps` 时，`steps` 在那个时刻可能是原本定义好的 `promise` 链了，或者它现在可能通过 `steps = steps.then(..)` 调用正指向扩张的 `promise` 链，这要看事情以什么顺序发生。

这里有两种可能的结果：

- 如果 `steps` 仍然是原来的 `Promise` 链，一旦它稍后通过 `steps = steps.then(..)` “扩展”，这个位于链条末尾的扩展过的 `promise` 是不会被 `main` 流程考虑的，因为它已经通过这个 `steps` 链了。这就是不幸的急切求值限制。
- 如果 `steps` 已经是扩展过的 `promise` 链了，那么由于这个扩展过的 `promise` 正是 `main` 要通过的东西，所以它会如我们期望的那样工作。

第一种情况除了展示竞合状态不可容忍的明显事实，它还展示了 `promise` 链的急切求值。相比之下，我们可以很容易地扩展可迭代序列而没有这样的问题，因为可迭代序列是懒惰求值的。

你越需要自己的流程控制动态，可迭代序列就越显得强大。

提示：在 `asynquence` 的网站

(<https://github.com/getify/asynquence/blob/master/README.md#iterable-sequences>) 上可以看到更多关于可迭代序列的信息与示例。

## 事件响应式

(至少！) 从第三章看来这应当很明显：`Promise` 是你异步工具箱中的一种非常强大的工具。但它们明显缺乏处理事件流的能力，因为一个 `Promise` 只能被解析一次。而且坦白地讲，对于 `asynquence` 序列来说这也正是它的一个弱点。

考虑这样一个场景：你想要在一个特定事件每次被触发时触发一系列步骤。一个单独的 `Promise` 或序列不能表示这个事件全部的发生状况。所以，你不得不为每一个事件的发生创建一个全新的 `Promise` 链（或序列），比如：

```
listener.on( "foobar", function(data){
    // 创建一个新的事件处理 Promise 链
    new Promise( function(resolve, reject){
        // ...
    } )
    .then( ... )
    .then( ... );
});
```

在这种方式拥有我们需要的基本功能，但是对于表达我们意图中的逻辑来说远不能使人满意。两种分离的能力混杂在这个范例中：事件监听，与事件应答；而关注点分离原则恳求我们将这些能力分开。

细心的读者会发现，这个问题与我们在第二章中详细讲解过的问题是有些对称的；它是一种控制反转问题。

想象一下非反转这个范例，就像这样：

```
var observable = listener.on( "foobar" );
// 稍后
observable
.then( ... )
.then( ... );
// 在其他的地方
observable
.then( ... )
.then( ... );
```

值 `observable` 不是一个真正的 `Promise`，但你可以像监听一个 `Promise` 那样监听它，所以它们是有密切关联的。事实上，它可以被监听很多次，而且它会在每次事件（`"foobar"`）发生时都发送通知。

**提示：**我刚刚展示过的这个模式，是响应式编程（reactive programming，也称为 RP）背后的概念和动机的大幅度简化，响应式编程已经由好几种了不起的项目和语言实现/详细论述过了。RP 的一个变种是函数响应式编程（functional reactive programming，FRP），它指的是在数据流之上实施函数式编程技术（不可变性，参照完整性，等等）。“响应式”指的是随着事件的推移散布这种功能，以对事件进行应答。对此感兴趣的读者应当考虑学习“响应式可监听对象”，它源于由微软开发的神奇的“响应式扩展”库（对于 JavaScript 来说是“RxJS”，

<http://rxjs.codeplex.com/>；它可要比我刚刚展示过的东西精巧和强大太多了。另外，Andre Staltz 写过一篇出色的文章(<https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>)，用具体的例子高效地讲解了 RP。

## ES7 可监听对象

在本书写作时，有一个早期ES7提案，一种称为“**Observable**（可监听对象）”的新数据类型(<https://github.com/jhusain/asyncgenerator#introducing-observable>)，它在精神上与我们在这里讲解过的相似，但是绝对更精巧。

这种可监听对象的概念是，你在一个流上“监听”事件的方法是传入一个 **generator**——其实迭代器才是有趣的部分——它的 `next(..)` 方法会为每一个事件而调用。

你可以想象它是这样一种东西：

```
// `someEventStream` 是一个事件流，来自于鼠标点击之类

var observer = new Observer( someEventStream, function*(){
  while (var evt = yield) {
    console.log( evt );
  }
});
```

你传入的 **generator** 将会 `yield` 而暂停 `while` 循环，来等待下一个事件。添附在 **generator** 实例上的 `next(..)` 将会在每次 `someEventStream` 发布一个新事件时被调用，因此这个事件将会使用 `evt` 数据推进你的 **generator**/迭代器。

在这里的监听事件功能中，重要的是 `迭代器` 的部分，而不是 `generator`。所以从概念上讲，你实质上可以传入任何可迭代对象，包括 `ASQ iterable()` 可迭代序列。

有趣的是，还存在一些被提案的适配方案，使得从特定类型的流中构建可监听对象变得容易，例如为DOM事件提案的 `fromEvent(..)`。如果你去看看 `fromEvent(..)` 在早期ES7提案中推荐的实现方式，你会发现它与我们将要在下一节中看到的 `ASQ.react(..)` 极其相似。

当然，这些都是早期提案，所以最终脱颖而出的东西可能会在外观/行为上与这里展示的有很大的不同。但是看到在不同的库与语言提案在概念上的早期统一还是很激动人心的！

## 响应式序列

将这种可监听对象（和F/RP）的超级简要的概览作为我们的启发与动机，我们现在将展示一种“响应式可监听对象”的很小的子集的适配方案，我称之为“响应式序列”。

首先，让我们从如何创建一个可监听对象开始，使用一个称为 `react(..)` 的 `asynquence` 插件工具：

```
var observable = ASQ.react( function setup(next){
  listener.on( "foobar", next );
} );
```

现在，让我们看看如何为这个 `observable` 定义一个“响应的”序列——在F/RP中，这通常称为“监听”：

```
observable
  .seq( ... )
  .then( ... )
  .val( ... );
```

所以，你只需要通过在这个可监听对象后面进行链接就可以了。很容易，是吧？

在F/RP中，事件流经常会通过一组函数式的变形，比如 `scan(..)`，`map(..)`，`reduce(..)`，等等。使用响应式序列，每个事件会通过一个序列的新实例。让我们看一个更具体的例子：

```
ASQ.react( function setup(next){
  document.getElementById( "mybtn" )
    .addEventListener( "click", next, false );
} )
  .seq( function(evt){
    var btnID = evt.target.id;
    return request(
      "http://some.url.1/?id=" + btnID
    );
  })
  .val( function(text){
    console.log( text );
  });
});
```

响应式序列的“响应式”部分来源于分配一个或多个事件处理器来调用事件触发器（调用 `next(..)`）。

响应式序列的“序列”部分正是我们已经探索过的：每一个步骤都可以是任何合理的异步技术——延续回调，Promise 或者 generator。

一旦拟建立了一个响应式序列，只要事件被持续地触发，它就会一直初始化序列的实例。如果你想停止一个响应式序列，你可以调用 `stop()`。

如果一个响应式序列被 `stop()` 了，你可能还想注销事件处理器；为此你可以注册一个拆卸处理器：

```

var sq = ASQ.react( function setup(next,registerTeardown){
  var btn = document.getElementById( "mybtn" );

  btn.addEventListener( "click", next, false );

  // 只要`sq.stop()`被调用，它就会被调用
  registerTeardown( function(){
    btn.removeEventListener( "click", next, false );
  } );
} )
.seq( ... )
.then( ... )
.val( ... );

// 稍后
sq.stop();

```

注意：在 `setup(..)` 处理器内部的 `this` 绑定引用是 `sq` 响应式序列，所以你可以在响应式序列的定义中使用 `this` 引用，比如调用 `stop()` 之类的方法，等等。

这是一个来自 Node.js 世界的例子，使用响应式序列处理到来的HTTP请求：

```

var server = http.createServer();
server.listen(8000);

// 响应式监听
var request = ASQ.react( function setup(next,registerTeardown){
  server.addListener( "request", next );
  server.addListener( "close", this.stop );

  registerTeardown( function(){
    server.removeListener( "request", next );
    server.removeListener( "close", request.stop );
  } );
});

// 应答请求
request
.seq( pullFromDatabase )
.val( function(data,res){
  res.end( data );
} );

// 关闭 node
process.on( "SIGINT", request.stop );

```

`next(..)` 触发器还可以很容易地适配 node 流，使用 `onStream(..)` 和 `unStream(..)`：

```
ASQ.react( function setup(next){
    var fstream = fs.createReadStream( "/some/file" );

    // 将流的 "data" 事件导向 `next(..)`
    next.onStream( fstream );

    // 监听流的结束
    fstream.on( "end", function(){
        next.unStream( fstream );
    } );
} )
.seq( ... )
.then( ... )
.val( ... );
```

你还可以使用序列组合来构成多个响应式序列流：

```
var sq1 = ASQ.react( ... ).seq( ... ).then( ... );
var sq2 = ASQ.react( ... ).seq( ... ).then( ... );

var sq3 = ASQ.react( ... )
.gate(
    sq1,
    sq2
)
.then( ... );
```

这里的要点是，`ASQ.react(...)` 是一个F/RP概念的轻量级适配，使得将一个事件流与一个序列的连接成为可能，因此得名“响应式序列”。对于基本的响应式用法，响应式序列的能力通常是足够的。

注意：这里有一个使用 `ASQ.react(...)` 来管理UI状态的例子(<http://jsbin.com/rozipaki/6/edit?js,output>)，和另一个使用`ASQ.react(..)`来处理HTTP请求/应答流的例子(<https://gist.github.com/getify/bba5ec0de9d6047b720e>)。

## Generator 协程

希望第四章帮助你很好地熟悉了ES6 generator。特别地，我们将重温并更加深入“Generator并发性”的讨论。

我们想象了一个 `runAll(..)` 工具，它可以接收两个或更多的 generator 并且并发地运行它们，让它们协作地将控制权从一个 `yield` 到下一个，并带有可选的消息传递。

除了能够将一个 generator 运行至完成之外，我们在附录A中谈论过的 `AQS#runner(..)` 是一个 `runAll(..)` 概念的近似实现，它可以将多个 generator 并发地运行至完成。

那么让我们看看如何实现第四章的并发Ajax场景：

```

ASQ(
  "http://some.url.2"
)
.runner(
  function*(token){
    // 转移控制权
    yield token;

    var url1 = token.messages[0]; // "http://some.url.1"

    // 清空消息重新开始
    token.messages = [];

    var p1 = request( url1 );

    // 转移控制权
    yield token;

    token.messages.push( yield p1 );
  },
  function*(token){
    var url2 = token.messages[0]; // "http://some.url.2"

    // 传递消息并转移控制权
    token.messages[0] = "http://some.url.1";
    yield token;

    var p2 = request( url2 );

    // 移控制权
    yield token;

    token.messages.push( yield p2 );

    // 讲结果传递给下一个序列步骤
    return token.messages;
  }
)
.val( function(res){
  // `res[0]` comes from "http://some.url.1"
  // `res[1]` comes from "http://some.url.2"
} );

```

以下是 ASQ#runner(..) 和 runAll(..) 之间的主要不同：

- 每个 generator (协程) 都被提供了一个称为 `token` 的参数值，它是一个当你想要明确地将控制权传递给下一个协程时 `yield` 用的特殊值。
- `token.messages` 是一个数组，持有从前一个序列步骤中传入的任何消息。它也是一种数

据结构，你可以用来在协程之间分享消息。

- `yield` 一个 `Promise`（或序列）值不会传递控制权，但会暂停这个协程处理直到这个值准备好。
- 这个协程处理运行到最后 `return` 或 `yield` 的值将会传递给序列中的下一个步骤。

为了适应不同的用法，在 `ASQ#runner(...)` 功能的基础上包装一层帮助函数也很容易。

## 状态机

许多程序员可能很熟悉的一个例子是状态机。在一个简单包装工具的帮助下，你可以创一个易于表达的状态机处理器。

让我们想象一个这样的工具。我们称之为 `state(..)`，我们将传递给它两个参数值：一个状态值和一个处理这个状态的 `generator`。`state(..)` 将担负起创建并返回一个适配器 `generator` 的脏活，并把它传递给 `ASQ#runner(..)`。

考虑如下代码：

```
function state(val, handler) {
    // 为这个状态制造一个协程处理器
    return function*(token) {
        // 状态转换处理器
        function transition(to) {
            token.messages[0] = to;
        }

        // 设置初始状态（如果还没有设置的话）
        if (token.messages.length < 1) {
            token.messages[0] = val;
        }

        // 持续运行直到最终状态 (false)
        while (token.messages[0] !== false) {
            // 当前的状态匹配这个处理器吗？
            if (token.messages[0] === val) {
                // 委托到状态处理器
                yield *handler(transition);
            }

            // 要把控制权转移给另一个状态处理器吗？
            if (token.messages[0] !== false) {
                yield token;
            }
        }
    };
}
```

如果你仔细观察，你会发现 `state(..)` 返回了一个接收 `token` 的 `generator`，然后它建立一个 `while` 循环，这个循环会运行到状态机直到到达它的最终状态（我们随意地将它选定为 `false` 值）为止；这正是我们想要传递给 `ASQ#runner(..)` 的那种 `generator`！

我们还随意地保留了 `token.messages[0]` 值槽，放置我们的状态机将要追踪的当前状态，这意味着我们甚至可以指定初始状态，作为序列中前一个步骤传递来的值。

我们如何将 `state(..)` 帮助函数与 `ASQ#runner(..)` 一起使用呢？

```

var prevState;

ASQ(
  /* 可选的：初始状态值 */
  2
)
// 运行我们的状态机
// 转换是：2 -> 3 -> 1 -> 3 -> false
.runner(
  // 状态 `1` 处理器
  state( 1, function *stateOne(transition){
    console.log( "in state 1" );

    prevState = 1;
    yield transition( 3 );      // 前往状态 `3`
  } ),

  // 状态 `2` 处理器
  state( 2, function *stateTwo(transition){
    console.log( "in state 2" );

    prevState = 2;
    yield transition( 3 );      // 前往状态 `3`
  } ),

  // 状态 `3` 处理器
  state( 3, function *stateThree(transition){
    console.log( "in state 3" );

    if (prevState === 2) {
      prevState = 3;
      yield transition( 1 ); // 前往状态 `1`
    }
    // 完成了！
    else {
      yield "That's all folks!";

      prevState = 3;
      yield transition( false ); // 终止状态
    }
  } )
)
// 状态机运行完成，所以继续
.val( function(msg){
  console.log( msg );      // That's all folks!
} );

```

重要的是，`*stateOne(..)`，`*stateTwo(..)`，和`*stateThree(..)` generator 本身会在每次进入那种状态时被调用，它们会在你`transition(..)`到另一个值时完成。虽然没有在这里展示，但是这些状态 generator 处理器理所当然地可以通过`yield Promise/序列/thunk`来异步

地暂停。

隐藏在底层的 generator 是由 state(...) 帮助函数产生的，实际上被传递给 ASQ#runner(...) 的 generator 是持续并发运行至状态机长度的那一个，它们的每一个都协作地将控制权 yield 给下一个，如此类推。

注意：看看这个“乒乓”的例子(<http://jsbin.com/qutabu/1/edit?js,output>)，它展示了由 `ASQ#runner(..)` 驱动的 generator 的协作并发的用法。

## 通信序列化处理（CSP）

“通信序列化处理（Communicating Sequential Processes —— CSP）”是由 C. A. R. Hoare 在 1978 年的一篇学术论文(<http://dl.acm.org/citation.cfm?doid=359576.359585>) 中首先被提出的，后来在 1985 年的一本同名书籍中被描述过。CSP 描述了一种并发“进程”在处理期间进行互动（也就是“通信”）的形式方法。

你可能会回忆起我们在第一章检视过的并发“进程”，所以我们对 CSP 的探索将会建立在那种理解之上。

就像大多数计算机科学中的伟大概念一样，CSP 深深地沉浸在学术形式主义中，被表达为一种代数处理。然而，我怀疑满是符号的代数定理不会给读者带来太多实际意义，所以我们将找其他的方法将 CSP 带进我们的大脑。

我会将很多 CSP 的形式描述和证明留给 Hoare 的文章，与其他许多美妙的相关作品。取而代之的是，我们将尽可能以一种非学院派的、但愿是可以直接理解的方法，来试着简要地讲解 CSP 的思想。

### 消息传递

CSP 的核心原则是，在独立进程之间的通信/互动都必须通过正式的消息传递。也许与你的期望背道而驰，CSP 的消息传递是作为同步行为进行描述的，发送进程与接收进程都不得不为消息的传递做好准备。

这样的同步消息怎么会与 JavaScript 中的异步编程有联系？

这种联系具体来自于 ES6 generator 的性质 —— generator 被用于生产看似同步的行为，而这些行为的内部既可以是同步的也可以（更可能）是异步的。

换言之，两个或更多并发运行的 generator 可能看起来像是在互相同步地传递消息，而同时保留了系统的异步性基础，因为每个 generator 的代码都会被暂停（也就是“阻塞”）来等待一个异步动作的运行。

这是如何工作的？

想象一个称为“**A**”的 generator，它想要给 generator “**B**”发送一个消息。首先，“**A**” `yield` 出要发送给“**B**”的消息（因此暂停了“**A**”）。当“**B**”准备好并拿走这个消息时，“**A**”才会继续（解除阻塞）。

与此对称的，想象一个 generator “**A**”想要从“**B**”接收一个消息。“**A**” `yield` 出一个从“**B**”取得消息的请求（因此暂停了“**A**”），一旦“**B**”发送了一个消息，“**A**”就拿来这个消息并继续。

对于这种CSP消息传递理论来说，一个更广为人知的表达形式是 ClojureScript 的 `core.async` 库，以及 `go` 语言。它们将CSP中描述的通信语义实现为一种在进程之间打开的管道，称为“频道（channel）”。

注意：频道这个术语描述了问题的一部分，因为存在一种模式，会有多于一个的值被一次性发送到这个频道的“缓冲”中；这与你对流的认识相似。我们不会在这里深入这个问题，但是对于数据流的管理来说它可能是一个非常强大的技术。

在CSP最简单的概念中，一个我们在“**A**”和“**B**”之间建立的频道会有一个称为 `take(..)` 的阻塞方法来接收一个值，以及一个称为 `put(..)` 的阻塞方法来发送一个值。

它看起来可能像这样：

```
var ch = channel();

function *foo() {
  var msg = yield take( ch );

  console.log( msg );
}

function *bar() {
  yield put( ch, "Hello World" );

  console.log( "message sent" );
}

run( foo );
run( bar );
// Hello World
// "message sent"
```

将这种结构化的、（看似）同步的消息传递互动，与 `ASQ#runner(..)` 通过 `token.messages` 数组与协作的 `yield` 提供的、非形式化与非结构化的消息共享相比较。实质上，`yield put(..)` 是一种可以同时发送值并为了传递控制权而暂停执行的单一操作，而前一个例子中我们将这两个步骤分开实施。

另外CSP强调，你不会真正明确地“传递控制权”，而是这样设计你的并发过程：要么为了从频道中接收值而阻塞，要么为了试着向这个频道中发送值而阻塞。这种围绕着消息的发送或接收的阻塞，就是你如何在协程之间协调行为序列的方法。

注意：预先奉告：这种模式非常强大，但要习惯它有些烧脑。你可能会需要实践它一下，来习惯这种协调并发性的新的思考方式。

有好几个了不起的库已经用 JavaScript 实现了这种风格的CSP，最引人注目的是“js-csp”(<https://github.com/ubolonton/js-csp>)，由 James Long (<http://twitter.com/jlongster>)开出的分支(<https://github.com/jlongster/js-csp>)，以及他特意撰写的作品(<http://jlongster.com/Taming-the-Asynchronous-Beast-with-CSP-in-JavaScript>)。另外，关于将 ClojureScript 中 go 风格的 core.async CSP 适配到 JS generator 的话题，无论怎么夸赞 David Nolen (<http://twitter.com/swannodette>) 的许多作品很精彩都不为过 (<http://swannodette.github.io/2013/08/24/es6-generators-and-csp>)。

## asynquence 的 CSP 模拟

因为我们是在我的 `asynquence` 库的上下文环境中讨论异步模式的，你可能会对这个话题很感兴趣：我们可以很容易地在 `ASQ#runner(..)` generator 处理上增加一个模拟层，来近乎完美地移植CSP的API和行为。这个模拟层放在与 `asynquence` 一起发放的“`asynquence-contrib`”包的可选部分。

与早先的 `state(..)` 帮助函数非常类似，`ASQ.csp.go(..)` 接收一个 generator —— 用 `go/core.async` 的术语来讲，它称为一个 `goroutine` —— 并将它适配为一个可以与 `ASQ#runner(..)` 一起使用的新 generator。

与被传入一个 `token` 不同，你的 `goroutine` 接收一个创建好的频道（下面的 `ch`），这个频道会被本次运行的所有 `goroutine` 共享。你可以使用 `ASQ.csp.chan(..)` 创建更多频道（这通常十分有用）。

在CSP中，我们使用频道消息传递上的阻塞作为所有异步性的模型，而不是为了等待 Promise/序列/thunk 的完成而发生的阻塞。

所以，与 `yield` 从 `request(..)` 中返回的 `Promise` 不同的是，`request(..)` 应当返回一个频道，你从它那里 `take(..)` 一个值。换句话说，一个单值频道在这种上下文环境/用法上大致上与一个 `Promise`/序列是等价的。

让我们先制造一个兼容频道版本的 `request(..)`：

```
function request(url) {
  var ch = ASQ.csp.channel();
  ajax( url ).then( function(content){
    // `putAsync(..)` 是 `put(..)` 的另一个版本，
    // 它可以在一个 generator 的外部使用。它为操作
    // 的完成返回一个 promise。我们不在这里使用这个
    // promise，但如果有必要的话我们可以在值被
    // `taken(..)` 之后收到通知。
    ASQ.csp.putAsync( ch, content );
  } );
  return ch;
}
```

在第三章中，“**promisory**”是一个生产 Promise 的工具，第四章中“**thunkory**”是一个生产 thunk 的工具，最后，在附录A中我们发明了“**sequory**”表示一个生产序列的工具。

很自然地，我们需要为一个生产频道的工具杜撰一个对称的术语。所以就让我们不出意料地称它为“**chanory**”（“channel” + “factory”）吧。作为一个留给读者的练习，请试着亲手定义一个 `channelify(..)` 的工具，就像 `Promise.wrap(..)` / `promisify(..)`（第三章），`thunkify(..)`（第四章），和 `ASQ.wrap(..)`（附录A）一样。

先考虑这个使用 **asyquence** 风格CSP的并发Ajax的例子：

```

ASQ()
.runner(
  ASQ.csp.go( function*(ch){
    yield ASQ.csp.put( ch, "http://some.url.2" );

    var url1 = yield ASQ.csp.take( ch );
    // "http://some.url.1"

    var res1 = yield ASQ.csp.take( request( url1 ) );

    yield ASQ.csp.put( ch, res1 );
  } ),
  ASQ.csp.go( function*(ch){
    var url2 = yield ASQ.csp.take( ch );
    // "http://some.url.2"

    yield ASQ.csp.put( ch, "http://some.url.1" );

    var res2 = yield ASQ.csp.take( request( url2 ) );
    var res1 = yield ASQ.csp.take( ch );

    // 讲结果传递给序列的下一个步骤
    ch.buffer_size = 2;
    ASQ.csp.put( ch, res1 );
    ASQ.csp.put( ch, res2 );
  } )
)
.val( function(res1,res2){
  // `res1` comes from "http://some.url.1"
  // `res2` comes from "http://some.url.2"
} );

```

消息传递在两个 `goroutines` 之间进行的 URL 字符串交换是非常直接的。第一个 `goroutine` 向第一个URL发起一个Ajax请求，它的应答被放进 `ch` 频道。第二个 `goroutine` 想第二个URL发起一个Ajax请求，然后从 `ch` 频道取下第一个应答 `res1`。在这个时刻，应答 `res1` 和 `res2` 都被完成且准备好了。

如果在 `goroutine` 运行的末尾 `ch` 频道还有什么剩余价值的话，它们将被传递进序列的下一个步骤中。所以，为了从最后的 `goroutine` 中传出消息，把它们 `put(..)` 进 `ch`。就像展示的那样，为了避免最后的那些 `put(..)` 阻塞，我们通过把 `ch` 的 `buffer_size` 设置为 `2`（默认是 `0`）来将它切换到缓冲模式。

注意：更多使用 `asynquence` 风格CSP的例子可以参见[这里](https://gist.github.com/getify/e0d04f1f5aa24b1947ae)。  
(<https://gist.github.com/getify/e0d04f1f5aa24b1947ae>)。

## 复习

Promise 和 generator 为我们能够创建更加精巧和强大的异步性提供了基础构建块。

`asynquence` 拥有许多工具，用于实现的迭代序列，响应式序列（也就是“可监听对象”），并发协程，甚至 *CSP goroutines*。

将这些模式，与延续回调和 Promise 能力相组合，使得 `asynquence` 拥有了混合不同异步处理的强大功能，一切都整合进一个干净的异步流程控制抽象：序列。

# 你不懂JS：异步与性能

## 附录C：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Jake Archibald为本书作序。

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 Tregoining, 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 Hodsdon, 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 ב-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 enamorato, 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 Ziolkowski, 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 Viloslada, 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 Gharegozlu, 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。

# 你不懂JS：ES6与未来

## 目录

- 序
- 前言
- 第一章：ES？现在与未来
  - 版本
  - 转译
- 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
- 第三章：组织
  - 迭代器
  - Generators
  - 模块
  - 类
- 第四章：异步流程控制
  - Promises
  - Generators + Promises
- 第五章：集合
  - 类型化数组（`TypedArrays`）
  - Maps
  - WeakMaps
  - Sets
  - WeakSets
- 第六章：新增API

- `Array`
- `Object`
- `Math`
- `Number`
- `String`
- 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)
- 第八章：ES6以后
  - `async function`
  - `Object.observe(..)`
  - 指数操作符
  - 对象属性与 ...
  - `Array#includes(..)`
  - SIMD
- 附录A：鸣谢

# 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

# 你不懂JS：ES6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该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

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

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

ES6（有些人可能会称它为**ES2015**）在本书写作时刚刚定稿，它包含许多你需要学习的新东西！

但更重要的是，它将你的思维模式与JavaScript新的进化方式相接轨。不是仅仅为了等待某些官方文档投票通过而耗上许多年，就像以前许多人做的那样。

现在，JavaScript特性一准备好就会在浏览器中实现，由你来决定是否现在就搭上早班车，还是去玩儿代价不菲的追车游戏。

不管未来的JavaScript采用什么样的标签，它都将会以比以前快得多的速度前进。为了使你位于在这门语言前进方向上的最前列，转译和填补是不可或缺的工具。

如果说对于JavaScript的新现实有什么重要的事情需要理解，那就是所有的JS开发者都被强烈地恳求从落后的一端移动到领先的一段。而学习ES6就是这一切的开端！

# 你不懂JS：ES6与未来

## 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/) (<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

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!
    ...
  }

  ...
}

let b;
}
```

`a` 没有被声明，所以 `typeof` 是检查它是否存在唯一安全的方法。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：

```
var funcs = [];

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

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：

```
var funcs = [];

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”）循环中也以相同的方式工作。

## 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：

```
{
  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

ES6引入了一个新的 `...` 操作符，根据你在何处以及如何使用它，它一般被称作 扩散 (`spread`) 或 剩余 (`rest`) 操作符。让我们看一看：

```

function foo(x,y,z) {
    console.log( x, y, z );
}

foo( ...[1,2,3] );           // 1 2 3

```

当 `...` 在一个数组（实际上，是我们将在第三章中讲解的任何的 可迭代 对象）前面被使用时，它就将数组“扩散”为它的个别的值。

通常你将会在前面所展示的那样的代码段中看到这种用法，它将一个数组扩散为函数调用的一组参数。在这种用法中，`...` 扮演了 `apply(..)` 方法的简约语法替代品，在前ES6中我们经常这样使用 `apply(..)`：

```
foo.apply( null, [1,2,3] );           // 1 2 3
```

但 `...` 也可以在其他上下文环境中被用于扩散/展开一个值，比如在另一个数组声明内部：

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

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

在这种用法中，`...` 取代了 `concat(..)`，它在这里的行为就像 `[1].concat( a, [5] )`。

另一种 `...` 的用法常见于一种实质上相反的操作；与将值散开不同，`...` 将一组值 收集 到一个数组中。

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

foo( 1, 2, 3, 4, 5 );           // 1 2 [3,4,5]
```

这个代码段中的 `...z` 实质上是在说：“将 剩余的 参数值（如果有的话）收集到一个称为 `z` 的数组中。”因为 `x` 被赋值为 `1`，而 `y` 被赋值为 `2`，所以剩余的参数值 `3`，`4`，和 `5` 被收集进了 `z`。

当然，如果你没有任何命名参数，`...` 会收集所有的参数值：

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

foo( 1, 2, 3, 4, 5 );           // [1,2,3,4,5]
```

注意：在 `foo(..)` 函数声明中的 `...args` 经常因为你向其中收集参数的剩余部分而被称为“剩余参数”。我喜欢使用“收集”这个词，因为它描述了它做什么而不是它包含什么。

这种用法最棒的地方是，它为被废弃了很久的 `arguments` 数组——实际上它不是一个真正的数组，而是一个类数组对象——提供了一种非常稳健的替代方案。因为 `args`（无论你叫它什么——许多人喜欢叫它 `r` 或者 `rest`）是一个真正的数组，我们可以摆脱许多愚蠢的前 ES6 技巧，我们曾经通过这些技巧尽全力去使 `arguments` 变成我们可以视之为数组的东西。

考虑如下代码：

```
// 使用新的ES6方式
function foo(...args) {
  // `args`已经是一个真正的数组了

  // 丢弃`args`中的第一个元素
  args.shift();

  // 将`args`的所有内容作为参数值传给`console.log(..)`
  console.log( ...args );
}

// 使用老旧的前ES6方式
function bar() {
  // 将`arguments`转换为一个真正的数组
  var args = Array.prototype.slice.call( arguments );

  // 在末尾添加一些元素
  args.push( 4, 5 );

  // 过滤掉所有奇数
  args = args.filter( function(v){
    return v % 2 == 0;
  } );

  // 将`args`的所有内容作为参数值传给`foo(..)`
  foo.apply( null, args );
}

bar( 0, 1, 2, 3 ); // 2 4
```

在函数 `foo(..)` 声明中的 `...args` 收集参数值，而在 `console.log(..)` 调用中的 `...args` 将它们扩散开。这个例子很好地展示了 `...` 操作符平行但相反的用途。

除了在函数声明中 `...` 的用法以外，还有另一种 `...` 被用于收集值的情况，我们将在本章稍后的“太多，太少，正合适”一节中检视它。

## 默认参数值

也许在JavaScript中最常见的惯用法之一就是为函数参数设置默认值。我们多年来一直使用的方法应当看起来很熟悉：

```

function foo(x,y) {
  x = x || 11;
  y = y || 31;

  console.log( x + y );
}

foo();           // 42
foo( 5, 6 );    // 11
foo( 5 );       // 36
foo( null, 6 ); // 17

```

当然，如果你曾经用过这种模式，你就会知道它既有用又有点儿危险，例如如果你需要能够为其中一个参数传入一个可能被认为是falsy的值。考虑下面的代码：

```
foo( 0, 42 );      // 53 <-- 噢，不是42
```

为什么？因为`0`是falsy，因此`x || 11`的结果为`11`，而不是直接被传入的`0`。

为了填这个坑，一些人会像这样更加啰嗦地编写检查：

```

function foo(x,y) {
  x = (x !== undefined) ? x : 11;
  y = (y !== undefined) ? y : 31;

  console.log( x + y );
}

foo( 0, 42 );      // 42
foo( undefined, 6 ); // 17

```

当然，这意味着除了`undefined`以外的任何值都可以直接传入。然而，`undefined`将被假定是这样一种信号，“我没有传入这个值。”除非你实际需要能够传入`undefined`，它就工作得很好。

在那样的情况下，你可以通过测试参数值是否没有出现在`arguments`数组中，来看它是否实际上被省略了，也许是像这样：

```

function foo(x,y) {
  x = (0 in arguments) ? x : 11;
  y = (1 in arguments) ? y : 31;

  console.log( x + y );
}

foo( 5 );           // 36
foo( 5, undefined ); // NaN

```

但是在没有能力传入意味着“我省略了这个参数值”的任何种类的值（连 `undefined` 也不行）的情况下，你如何才能省略第一个参数值 `x` 呢？

`foo(,5)` 很诱人，但它不是合法的语法。`foo.apply(null,[,5])` 看起来应该可以实现这个技巧，但是 `apply(..)` 的奇怪之处意味着这组参数值将被视为 `[undefined,5]`，显然它没有被省略。

如果你深入调查下去，你将发现你只能通过简单地传入比“期望的”参数值个数少的参数值来省略末尾的参数值，但是你不能省略在参数值列表中间或者开头的参数值。这就是不可能。

这里有一个施用于JavaScript设计的重要原则需要记住：`undefined` 意味着 缺失。也就是，在 `undefined` 和 缺失 之间没有区别，至少是就函数参数值而言。

注意：容易令人糊涂的是，JS中有其他的地方不适用这种特殊的设计原则，比如带有空值槽的数组。更多信息参见本系列的 [类型与文法](#)。

带着所有这些认识，现在我们可以检视在ES6中新增的一种有用的好语法，来简化对丢失的参数值进行默认值的赋值。

```

function foo(x = 11, y = 31) {
  console.log( x + y );
}

foo();           // 42
foo( 5, 6 );    // 11
foo( 0, 42 );   // 42

foo( 5 );       // 36
foo( 5, undefined ); // 36 <-- `undefined` 是缺失
foo( 5, null ); // 5 <-- null 强制转换为 `0`

foo( undefined, 6 ); // 17 <-- `undefined` 是缺失
foo( null, 6 );   // 6 <-- null 强制转换为 `0`

```

注意这些结果，和它们如何暗示了与前面的方式的微妙区别和相似之处。

与常见得多的 `x || 11` 惯用法相比，在一个函数声明中的 `x = 11` 更像 `x !== undefined ? x : 11`，所以在将你的前ES6代码转换为这种ES6默认参数值语法时要多加小心。

注意：一个剩余/收集参数（参见“扩散/剩余”）不能拥有默认值。所以，虽然 `function foo(...vals=[1, 2, 3]) {` 看起来是一种迷人的能力，但它不是合法的语法。有必要的话你需要继续手动实施那种逻辑。

## 默认值表达式

函数默认值可以比像 `31` 这样的简单值复杂得多；它们可以是任何合法的表达式，甚至是函数调用：

```
function bar(val) {
  console.log("bar called!");
  return y + val;
}

function foo(x = y + 3, z = bar(x)) {
  console.log(x, z);
}

var y = 5;
foo();                                // "bar called"
                                         // 8 13
foo(10);                               // "bar called"
                                         // 10 15
y = 6;
foo(undefined, 10);                   // 9 10
```

如你所见，默认值表达式是被懒惰地求值的，这意味着他们仅在被需要时运行——也就是，当一个参数的参数值被省略或者为 `undefined`。

这是一个微妙的细节，但是在函数声明中的正式参数是在它们自己的作用域中的（将它想象为一个仅仅围绕在函数声明的`(..)`外面的一个作用域气泡），不是在函数体的作用域中。这意味着在一个默认值表达式中的标识符引用会在首先在正式参数的作用域中查找标识符，然后再查找一个外部作用域。更多信息参见本系列的作用域与闭包。

考虑如下代码：

```
var w = 1, z = 2;

function foo(x = w + 1, y = x + 1, z = z + 1) {
  console.log(x, y, z);
}

foo();                                // ReferenceError
```

在默认值表达式 `w + 1` 中的 `w` 在正式参数作用域中查找 `w`，但没有找到，所以外部作用域的 `w` 被使用了。接下来，在默认值表达式 `x + 1` 中的 `x` 在正式参数的作用域中找到了 `x`，而且走运的是 `x` 已经被初始化了，所以对 `y` 的赋值工作的很好。

然而，`z + 1` 中的 `z` 找到了一个在那个时刻还没有被初始化的参数变量 `z`，所以它绝不会试着在外部作用域中寻找 `z`。

正如我们在本章早先的“`let` 声明”一节中提到过的那样，ES6 拥有一个 TDZ，它会防止一个变量在它还没有被初始化的状态下被访问。因此，`z + 1` 默认值表达式抛出一个 `TDZ ReferenceError` 错误。

虽然对于代码的清晰度来说不见得是一个好主意，一个默认值表达式甚至可以是一个内联的函数表达式调用——通常被称为一个立即被调用的函数表达式（IIFE）：

```
function foo( x =
  (function(v){ return v + 11; })( 31 )
) {
  console.log( x );
}

foo();           // 42
```

一个IIFE（或者任何其他被执行的内联函数表达式）作为默认值表示来说很合适是非常少见的。如果你发现自己试图这么做，那么就退一步再考虑一下！

警告：如果一个IIFE试图访问标识符 `x`，而且还没有声明自己的 `x`，那么这也将是一个TDZ 错误，就像我们刚才讨论的一样。

前一个代码段的默认值表达式是一个IIFE，这是因为它是通过 `(31)` 在内联时立即被执行。如果我们去掉这一部分，赋予 `x` 的默认值将会仅仅是一个函数的引用，也许像一个默认的回调。可能有一些情况这种模式将十分有用，比如：

```
function ajax(url, cb = function(){}) {
  // ..
}

ajax( "http://some.url.1" );
```

这种情况下，我们实质上想在没有其他值被指定时，让默认的 `cb` 是一个没有操作的空函数。这个函数表达式只是一个函数引用，不是一个调用它自己（在它末尾没有调用的 `()`）以达成自己目的的函数。

从JS的早些年开始，就有一个少为人知但是十分有用的奇怪之处可供我们使用：`Function.prototype` 本身就是一个没有操作的空函数。这样，这个声明可以是 `cb = Function.prototype` 而省去内联函数表达式的创建。

## 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
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：

```
var X = 10, Y = 20;
var o = { a: X, b: Y };
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：

```
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：

```
var aa = 10, bb = 20;

var o = { x: aa, y: bb };
var { x: AA, y: BB } = o;

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 6
```

## 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：

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

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`：

```
var [,,c,d] = foo();
var { w, z } = bar();

console.log( c, z );           // 3 6
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：

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

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：

```
var a = [2,3,4];
var [ b, ...c ] = a;

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 );
}

foo( [ 1, 2 ] );           // 1 2
foo( [ 1 ] );              // 1 undefined
foo( [] );                 // undefined undefined
```

参数也可以进行对象解构：

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

foo( { y: 1, x: 2 } );      // 2 1
foo( { y: 42 } );          // undefined 42
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）：

```
function f1([ x=2, y=3, z ]) { ... }
function f2([ x, y, ...z ], w) { ... }
function f3([ x, y, ...z ], ...w) { ... }

function f4({ x: X, y }) { ... }
function f5({ x: X = 10, y = 20 }) { ... }
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：

```
function f3([ x, y, ...z ], ...w) {
    console.log( x, y, z, w );
}

f3( [] );                                // undefined undefined []
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成为 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

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

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，事实上这也可以说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 `strings` 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，一个字符串字面连标签函数（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;
  }, "");
}

var amt1 = 11.99,
  amt2 = amt1 * 1.08,
  name = "Kyle";

var text = dollabillsyall
`Thanks for your purchase, ${name}! Your
product cost was ${amt1}, which with tax
comes out to ${amt2}.`

console.log( text );
// Thanks for your purchase, Kyle! Your
// product cost was $11.99, which with tax
// 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：

```

function showraw(strings, ...values) {
  console.log( strings );
  console.log( strings.raw );
}

showraw`Hello\nWorld`;
// [ "Hello
// World" ]
// [ "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：

```

console.log(`Hello\nWorld`);
// Hello
// World

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

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引人，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` 敏感的函数会发生什么？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：

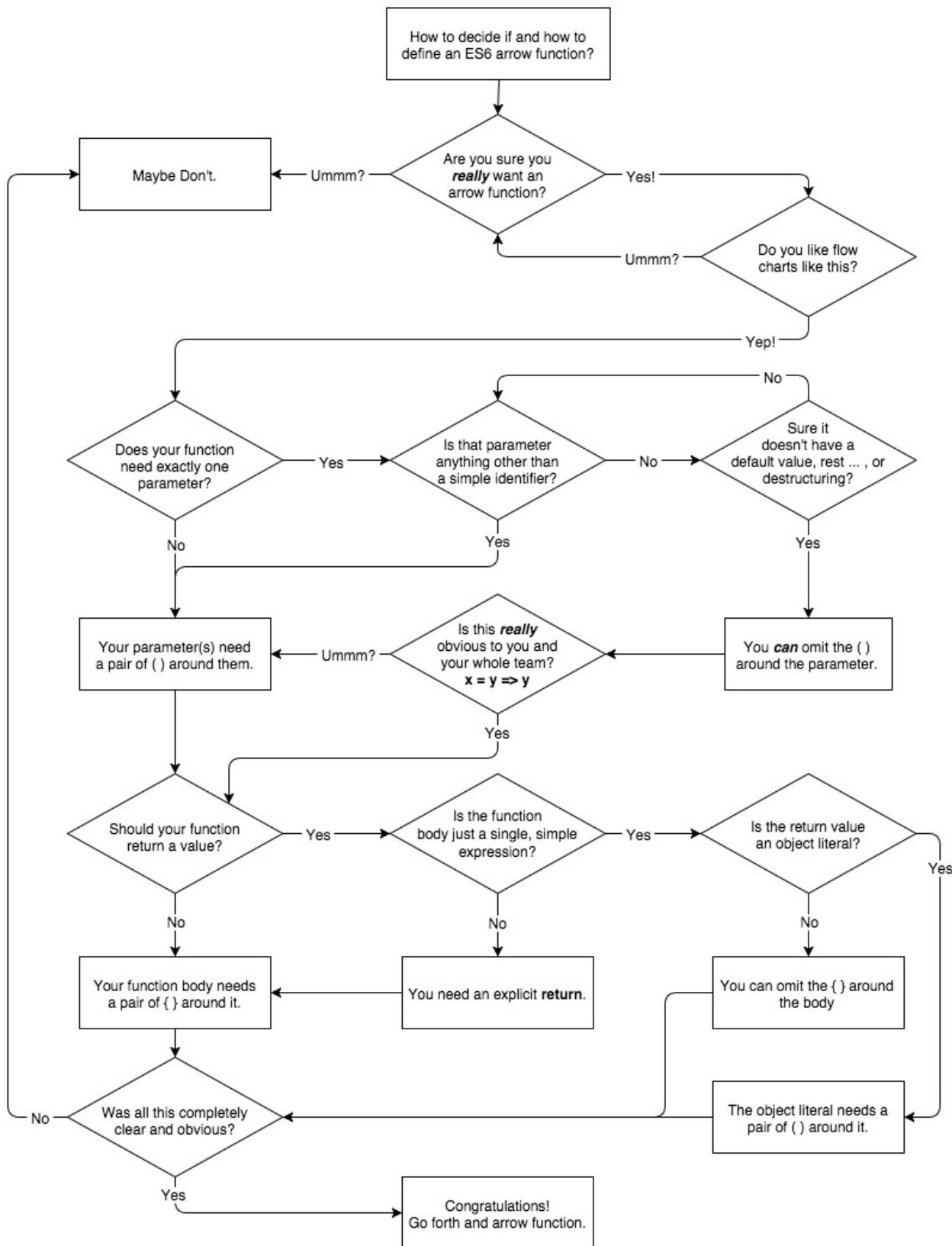
- 如果你有一个简短的，单语句内联函数表达式，它唯一的语句是某个计算后的值的 `return` 语句，并且这个函数没有在它内部制造一个 `this` 引用，并且没有自引用（递归，事件绑定/解除），并且你合理地预期这个函数绝不会变得需要 `this` 引用或自引用，那么你就可能安全地将它重构为一个 `=>` 箭头函数。
- 如果你有一个内部函数表达式，它依赖于外围函数的 `var self = this` 黑科技或者 `.bind(this)` 调用来确保正确的 `this` 绑定，那么这个内部函数表达式就可能安全地变为一个 `=>` 箭头函数。
- 如果你有一个内部函数表达式，它依赖于外围函数的类似于 `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：



## for..of 循环

伴随着我们熟知的JavaScript `for` 和 `for..in` 循环，ES6增加了一个 `for..of` 循环，它循环遍历一组由一个迭代器（*iterator*）产生的值。

你使用 `for..of` 循环遍历的值必须是一个可迭代对象 (*iterable*)，或者它必须是一个可以被强制转换/封箱（参见本系列的 [类型与文法](#)）为一个可迭代对象的值。一个可迭代对象只不过是一个可以生成迭代器的对象，然后由循环使用这个迭代器。

让我们比较 `for..of` 与 `for..in` 来展示它们的区别：

```
var a = ["a", "b", "c", "d", "e"];

for (var idx in a) {
    console.log( idx );
}

// 0 1 2 3 4

for (var val of a) {
    console.log( val );
}

// "a" "b" "c" "d" "e"
```

如你所见，`for..in` 循环遍历数组 `a` 中的键/索引，而 `for.of` 循环遍历 `a` 中的值。

这是前面代码段中 `for..of` 的前ES6版本：

```
var a = ["a", "b", "c", "d", "e"],
    k = Object.keys( a );

for (var val, i = 0; i < k.length; i++) {
    val = a[ k[i] ];
    console.log( val );
}

// "a" "b" "c" "d" "e"
```

而这是一个ES6版本的非 `for..of` 等价物，它同时展示了手动迭代一个迭代器（见第三章的“[迭代器](#)”）：

```
var a = ["a", "b", "c", "d", "e"];

for (var val, ret, it = a[Symbol.iterator]();
    (ret = it.next()) && !ret.done;
) {
    val = ret.value;
    console.log( val );
}

// "a" "b" "c" "d" "e"
```

在幕后，`for..of` 循环向可迭代对象要来一个迭代器（使用内建的 `Symbol.iterator`；参见第七章的“[通用 Symbols](#)”），然后反复调用这个迭代器并将它产生的值赋值给循环迭代的变量。

在JavaScript标准的内建值中，默認為可迭代对象的（或提供可迭代能力的）有：

- 数组
- 字符串
- Generators（见第三章）
- 集合/类型化数组（见第五章）

警告：普通对象默认是不适用于 `for..of` 循环的。因为他们没有默认的迭代器，这是有意为之的，不是一个错误。但是，我们不会进一步探究这其中微妙的原因。在第三章的“迭代器”中，我们将看到如何为我们自己的对象定义迭代器，这允许 `for..of` 遍历任何对象来得到我们定义的一组值。

这是如何遍历一个基本类型的字符串中的字符：

```
for (var c of "hello") {
    console.log( c );
}
// "h" "e" "l" "l" "o"
```

基本类型字符串 "hello" 被强制转换/封箱为等价的 `String` 对象包装器，它是默认就是一个可迭代对象。

在 `for (XYZ of ABC)...` 中，`XYZ` 子句既可以是一个赋值表达式也可以是一个声明，这与 `for` 和 `for..in` 中相同的子句一模一样。所以你可以做这样的事情：

```
var o = {};

for (o.a of [1,2,3]) {
    console.log( o.a );
}
// 1 2 3

for ({x: o.a} of [ {x: 1}, {x: 2}, {x: 3} ]) {
    console.log( o.a );
}
// 1 2 3
```

与其他的循环一样，使用 `break`，`continue`，`return`（如果是在一个函数中），以及抛出异常，`for..of` 循环可以被提前终止。在任何这些情况下，迭代器的 `return(..)` 函数（如果存在的话）都会被自动调用，以便让迭代器进行必要的清理工作。

注意：可迭代对象与迭代器的完整内容参见第三章的“迭代器”。

## 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](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}`）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）。对于出现在字符串类中的 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），否则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

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 `0052` :(
    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，以这种方式你可以用从 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

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}";
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：

```
var snowman = "\u2603";
snowman.length; // 1

var gclef = "\u260E";
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：

```
var gclef = "\u260E";

[...gclef].length; // 1
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：

```
console.log( s1 ); // "é"
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 = "õ";

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 );          // "é"
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](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 );           // "ab d"
console.log( s2 );           // "ab d"
console.log( s3 );           // "abd"

s1.charAt( 2 );              // "c"
s2.charAt( 2 );              // " "
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];        // " "
[...s2.normalize()][2];        // " "
[...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 );           // "c"
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 ) );
// "c"

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

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时，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

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] ||
      /[^\&]+?=( [^\&]+?)\b(=?=&|$)/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。这其实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

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。

# 你不懂JS：ES6与未来

## 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

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，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();

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

```

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 }

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

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

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

```

这种特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：

```

if (!Number.prototype[Symbol.iterator]) {
  Object.defineProperty(
    Number.prototype,
    Symbol.iterator,
    {
      writable: true,
      configurable: true,
      enumerable: false,
      value: function iterator(){
        var i, inc, done = false, top = +this;

        // 正向迭代还是负向迭代？
        inc = 1 * (top < 0 ? -1 : 1);

        return {
          // 使迭代器本身成为一个可迭代对象！
          [Symbol.iterator](){ return this; },

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

## 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 );
}
```

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：

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

```
var a, b;

a = 3;                      // 合法
b = 2 + a = 3;              // 不合法
b = 2 + (a = 3);            // 合法

yield 3;                     // 合法
a = 2 + yield 3;             // 不合法
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，这意味着它们会在 `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 `yield *..` 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 }

it2.next();           // { value: undefined, done: true }
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：

```

function *foo() {
    yield 1;
    yield 2;
    yield 3;
}

var it = foo();

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

try {
    it.throw( "Oops!" );
}
catch (err) {
    console.log( err );   // Exception: Oops!
}

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

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。也就是说，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：

```
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。

```
import { foo } from "foo";
foo();
```

你可以重命名被导入的绑定标识符，就像：

```
import { foo as theFooFunc } from "foo";
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：

```
import foo from "foo";
// 或者：
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";

foo.bar();
foo.x;           // 42
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：

```
export default function foo() { ... }
export function bar() { ... }
export function baz() { ... }
```

和这个 `import` :

```
import foofn, * as hello from "world";

foofn();
hello.default();
hello.bar();
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`：

```
import foofn, * as hello from "world";

foofn = 42;           // (运行时) TypeError!
hello.default = 42;   // (运行时) TypeError!
hello.bar = 42;       // (运行时) TypeError!
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：

```
foo();  
  
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：

```
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。

现在然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

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，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 `prototype` 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：

```
class ParentA {
  constructor() { this.id = "a"; }
  foo() { console.log( "ParentA:", this.id ); }
}

class ParentB {
  constructor() { this.id = "b"; }
  foo() { console.log( "ParentB:", this.id ); }
}

class ChildA extends ParentA {
  foo() {
    super.foo();
    console.log( "ChildA:", this.id );
  }
}

class ChildB extends ParentB {
  foo() {
    super.foo();
    console.log( "ChildB:", this.id );
  }
}

var a = new ChildA();
a.foo(); // ParentA: a
         // ChildA: a
var b = new ChildB();
b.foo(); // ParentB: b
         // 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

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。

# 你不懂JS：ES6与未来

## 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

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时间，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（如果存在的話）被看作是promise被成功地完成时要调用的处理器。第二个函数（如果存在的話）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

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

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！

# 你不懂JS：ES6与未来

## 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**)

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：

```
var littleEndian = (function() {
  var buffer = new ArrayBuffer( 2 );
  new DataView( buffer ).setInt16( 0, 256, true );
  return new Int16Array( buffer )[0] === 256;
})();
```

`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：

```
var buf = new ArrayBuffer( 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高度并且值都是同种“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位的值，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

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

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 };

m.set( x, y );

x = null;           // { id: 1 } 是可以GC的
y = null;           // 由于 { id: 1 } 是可以GC的，因此 { id: 2 } 也可以

m.set( z, w );

w = null;           // { id: 4 } 不可以GC

```

因此，我认为WeakMap被命名为“WeakKeyMap”更好。

## 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

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

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。

# 你不懂JS：ES6与未来

## 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

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"
};

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：

```
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：

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

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

Array.from( arrLike );
// [ 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：

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

Array.from( emptySlotsArr );
// [ 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；参见第五章）。`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];

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

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

就像 `Set` 一样，默认的 `Array` 迭代器与 `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：

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

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

## 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

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位整数乘法

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

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

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 );           // ""

"abd" .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`;
// "\abcd\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

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`cld32(..)`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。

# 你不懂JS：ES6与未来

## 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

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

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新目标是什么，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

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，或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();

c instanceof Fun;           // true
d instanceof Awesome;       // false
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：

```

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

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

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

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示来示调用 `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

(<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

(<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

(<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

(<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

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 传递到原本的目标/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.getOwnPropertySymbols(..)`，  
`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
];

obj3.baz();
// obj1.foo: obj-3
// 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` 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

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，这将极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)

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。又一个你应该用 strict 编写所有代码的理由！

这个函数调用不是在尾部：

```
"use strict";

function foo(x) {
    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：

```
"use strict";

function trampoline( res ) {
    while (typeof res == "function") {
        res = res();
    }
    return res;
}

var foo = (function(){
    function _foo(acc,x) {
        if (x <= 1) return acc;
        return function partial(){
            return _foo( (x / 2) + acc, x - 1 );
        };
    }

    return function(x) {
        return trampoline( _foo( 1, x ) );
    };
})();

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：

- 首先，我们将 `return _foo ..` 这一行包装进函数表达式 `return partial() {..}`。
- 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

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大提高了它的地位。

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！

# 你不懂JS：ES6与未来

## 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

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();

    try {
        ret = await step2( ret );
    }
    catch (err) {
        ret = await step2Failed( err );
    }

    ret = await Promise.all( [
        step3a( ret ),
        step3b( ret ),
        step3c( ret )
    ]);

    await step4( ret );
}

main()
.then(
    function fulfilled(){
        // `main()` 成功地完成了
    },
    function rejected(reason){
        // 噢，什么东西搞错了
    }
);

```

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`promise`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`promise`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(..)

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
- setPrototypeOf
- 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部分。

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 }

delete obj.b;
// { 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。

- "setPrototypeOf" 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：

```

function observer(changes){
  for (var change of changes) {
    if (change.type == "recalc") {
      change.object.c =
        change.object.oldValue +
        change.object.a +
        change.object.b;
    }
  }
}

function changeObj(a,b) {
  var notifier = Object.getNotifier( obj );

  obj.a = a * 2;
  obj.b = b * 3;

  // 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：

```
var obj = { a: 1, b: 2 };

Object.observe( obj, function observer(changes) {
    for (var change of changes) {
        if (change.type == "setPrototypeOf") {
            Object.unobserve(
                change.object, observer
            );
            break;
        }
    }
} );
```

在这个小例子中，我们监听变化事件直到我们看到 "setPrototypeOf" 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

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 `...`

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(...)

JS开发者需要执行的极其常见的一个任务就是在值的数组中搜索一个值。完成这项任务的方式曾经总是：

```
var vals = [ "foo", "bar", 42, "baz" ];

if (vals.indexOf( 42 ) >= 0) {
    // 找到了!
}
```

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：

```
var vals = [ "foo", "bar", 42, "baz" ];

if (~vals.indexOf( 42 )) {
    // 找到了!
}
```

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(..)`：

```
var vals = [ "foo", "bar", 42, "baz" ];

if (vals.includes( 42 )) {
    // 找到了!
}
```

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

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)

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。但如果不能简要地提上一句，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。ASM.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ASM.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

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！

# 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Rick Waldron为本书作序。

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 Tregoining, 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 Hodsdon, 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 ב-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 enamorato, 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 Ziolkowski, 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 Viloslada, 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 Gharegozlu, 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。