并非所有的新特性都是可以兼容旧环境的。有时一个特性的绝大部分可以兼容旧环境，但仍有微小的偏离。如果你要亲自进行 polyfilling 的话，一定要特别小心，确保尽可能严格地遵循标准规定。

或许更好的办法是，使用一个已有的、可信任的 polyfilling 版本，比如由 ES5-Shim（https://github.com/es-shims/es5-shim）和 ES6-Shim（https://github.com/es-shims/es6-shim）提供的版本。

2.8.2 transpiling

语言中新增的语法是无法进行 polyfilling 的。新语法在旧版 JavaScript 引擎上会抛出未识别 / 无效错误。

因此，更好的方法是，通过工具将新版代码转换为等价的旧版代码。这个过程通常被称为“transpiling”。它是由 transforming（转换）和 compiling（编译）组合而成的术语。

从本质上来说，你的源码是用新语法形式编写的，但部署在浏览器上的是编译转换后的旧语法形式。通常会在构建过程中插入 transpiler 工具，类似于代码 linter 或者 minifier。

你可能会疑惑为什么要这么麻烦地编写新语法代码，难道只是为了将它编译转换到旧版代码——为什么不直接编写旧语法代码呢？

有几点重要原因使得 transpiling 值得被关注。

- 语言中新添加的语法的设计目的是让代码更容易阅读和维护。等价的旧版本通常更加繁复。你应该编写更新、更简洁的语法，这不只是为你自己，同时也是为开发组中的所有其他成员着想。
- 如果只是为旧版本进行编译转换，对新版本应用新语法，那么你就得到了新语法浏览器性能优化的好处。这也使得浏览器开发者可以拥有更真实的代码，以便测试它们的实现和优化。
- 越早使用新语法，就可以越早在现实世界中更健壮地测试这些语法，也就可以越早地为 JavaScript 委员会（TC39）提供反馈。如果能够很早就发现问题，那么就能够在这些语言设计错误被固化前对其进行修改 / 修复。

以下是 transpiling 的一个简单示例。ES6 新增了一个名为“默认参数值”的新特性。如下所示：

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

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

很简单，对不对？但也非常有用！然而这个新语法在 ES6 前的引擎中是无效的。那么

transpiler 是如何改变这段代码，从而让其能够在旧环境下运行的呢？

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

正如你可以看到的，它会检查 `arguments[0]` 的值是否为 `void 0`（也就是 `undefined`），如果是的话就提供默认值 `2`；否则就使用传入值。

除了能够在旧版浏览器中使用更好的新语法，编译转换后的代码实际上也更好地表达了编程意图。

单看这段 ES6 版本的代码，你可能不会意识到 `undefined` 是唯一一个无法作为默认值参数显式传入的值。而编译转换后的代码就更清楚地展示了这一点。

关于 transpiler 最后要强调的重要细节是，现在应该将它看作是 JavaScript 开发生态环境和过程的一个标准部分。JavaScript 将会持续进化，比以往更快，所以每隔几个月就会添加新的语法和特性。

如果你默认使用 transpiler，只要发现新的语法有用就能够一直转换到新语法，而无需等到多年以后当前浏览器被淘汰。

有很多很棒的 transpiler 可供选择。以下是编写本部分时几个很好的选择：

- Babel（https://babeljs.io/，从 6 到 5）
  从 ES6+ 编译转换到 ES5
- Traceur（https://github.com/google/traceur-compiler）
  将 ES6、ES7 及后续版本转换到 ES5

2.9 非 JavaScript

到目前为止，我们介绍的内容都局限于 JavaScript 语言本身。而现实情况是，大多数的 JavaScript 都是编写用于在浏览器这样的环境中运行并与之交互的。严格来说，你编写的代码很大一部分并不直接由 JavaScript 控制。这听起来有点奇怪。

你将遇到的最常见的非 JavaScript 就是 DOM API。举例来说：

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

当你的代码在浏览器中运行时，变量 `document` 作为一个全局变量存在。它既不是由 JavaScript 引擎提供的，也不由 JavaScript 标准控制。它的存在形式看起来非常类似于普通的 JavaScript 对象，但实际上并不完全是这样。它是一个特殊的对象，通常被称为“宿

主对象”。

另外，`document` 上的方法 `getElementById(..)` 看起来像是一个正常的 JavaScript 函数，但它其实是浏览器的 DOM 提供的指向内置方法的一个很薄的暴露接口。在某些（新版的）浏览器中，这一层可能在 JavaScript 中，但传统的 DOM 及其行为更可能是用 C/C++ 实现的。

另一个示例是输入 / 输出 (I/O)。

广受喜爱的 `alert(..)` 会在用户浏览器窗口弹出一个消息框。`alert(..)` 是由浏览器提供给 JavaScript 程序的，而不是由 JavaScript 引擎本身提供。你发起的调用将消息发送到浏览器内部，然后由它负责绘制并显示消息框。

`console.log(..)` 也是如此；你的浏览器提供了这样的机制并将其连接到开发者工具中。

本书以及本系列主要关注 JavaScript 语言。因此你不会看到这些非 JavaScript 的机制。但你需要了解这些知识，因为你编写的每个 JavaScript 程序都需要用到它们！

2.10 小结

学习使用 JavaScript 编程的第一步就是要了解其核心机制，比如值、类型、函数闭包、`this` 以及原型。

当然，你在本部分看到的每个主题都值得更深入的学习，这也是本系列其他部分专门讲述这些概念的原因。充分理解本章中的概念和代码示例后，本系列的其他部分就等着你去真正挖掘了，希望你能对这门语言获得更深入的理解。

本部分的最后一章简单总结了本系列其他部分的主题，以及那些我们还没有讨论过的概念。

第 3 章

深入“你不知道的 JavaScript”系列

这个系列到底讲了什么？简单地说，这个系列视学习**全部的** JavaScript 为一个严肃的任务，不仅仅是这门语言中被称为“精髓”的某个子集，也不是你完成工作所需要的最小集合。

学习其他语言的认真的开发者会想要花费精力学习他们使用的主要语言的方方面面，但 JavaScript 开发者却通常不会学习这个语言的很多内容，从这个意义上来说，他们似乎是特立独行的。这并不是好事情，也不是我们应该继续放任其发展的事情。

“你不知道的 JavaScript”系列与普遍的 JavaScript 学习方法形成鲜明对比，几乎与任何其他你能读到的 JavaScript 相关书籍都有所不同。它能够让你超越自己的舒适区，对遇到的每个特性提出更深层次的“为什么”。你准备好迎接挑战了吗？

我将在本章中简单总结一下本系列其余几本图书的内容，以及如何基于这个系列最有效地构建 JavaScript 学习的基础。

3.1 作用域和闭包

参见《你不知道的 JavaScript（上卷）》第一部分。

变量的作用域到底是如何在 JavaScript 中工作的？这可能是你需要快速理解的一个最基础的事情了。对作用域只有**道听途说、模糊不清**的理解是不够的。

“作用域和闭包”这部分从批判 JavaScript 是“解释性语言”因而无法编译这一常见误解开始。事实并非如此。

JavaScript 引擎在执行前（有时是执行中！）就编译了代码。因此，通过深入理解编译器对代码的处理方式，我们可以尝试理解它是如何找到并处理变量和函数声明的。沿着这条路，我们看到了 JavaScript 变量作用域管理的常见方式——“提升”。

对“词法作用域”的关键理解是我们在上卷第一部分最后一章继续研究闭包的基础。闭包可能是 JavaScript 所有概念中最重要的一个，但如果你没有深刻了解作用域的工作原理，那么很可能就无法理解闭包。

正如我们在第 2 章中简单提到的那样，闭包的一个重要应用就是模块模式。模块模式可能是 JavaScript 所有代码组织模式中最普遍的方法；深入理解模块模式应该是你最高优先级的任务之一。

3.2 this 和对象原型

参见《你不知道的 JavaScript（上卷）》第二部分。

有关 JavaScript 流传最广、最持久的不实论点是，关键词 `this` 指向它所在的函数。这简直错得离谱。

关键词 `this` 是根据相关函数的执行方式而动态绑定的，事实证明，可以通过 4 条简单的规则理解并完全确定 `this` 绑定。

与 `this` 紧密关联的是对象原型机制，这种机制是一个属性查找链，与寻找词法作用域变量的方式类似。但在原型中进行封装，即模拟（伪造）类和（所谓“原型化的”）继承，是对 JavaScript 的另一个重大误用。

不幸的是，将类和继承的设计模式思维带入 JavaScript 的想法是你所做的最坏的事情，因为语法可能会让你迷惑不已，让你以为真的有类这样的东西存在，实际上原型机制与类的行为特性是完全相反的。

问题是，忽略这种不一致性而假装你实现的就是“继承”更好，还是学习和接受对象原型系统真实的工作方式更有益呢？后者被更合理地命名为“行为委托”。

这不只是语法偏好的问题。委托是完全不同的设计模式，也更加强大，它取代了需要类和继承的设计。但是这些判断违背了这个主题在 JavaScript 的整个生命周期的每个博文、图书和会议发言中的说法。

我对委托与继承的看法并非出自对这门语言及其语法的厌恶，而是来自对使用这个语言真实能力的期待，以及消除无休止的迷惑和沮丧的期待。

但我就原型和委托给出的解释示例要比这里的内容深入许多。如果你已经准备好重新思考对 JavaScript 的“类”和“继承”的所有认知，那么我为你提供“吃下红色药丸”（《黑客

帝国》，1999）的机会，阅读“this 和对象原型”这部分的第 4~6 章吧。

3.3 类型和语法

参见《你不知道的 JavaScript（中卷）》第一部分。

“类型和语法”这部分主要关注另一个高度争议的主题：强制类型转换。当讨论有关隐式类型转换的迷惑时，没有比这个主题更令 JavaScript 开发者烦恼的了。

到目前为止，传统的认知是，隐式类型转换是这个语言中“坏的部分”，应该不惜代价地予以避免。实际上，有些人甚至称其为这个语言的设计“缺陷”。确实，一些工具所做的所有事情就是搜索代码，抱怨你是否进行了类型转换这样的事情。

但是，类型转换是否真的这么令人迷惑、这么坏、这么危险，以至于你一使用就会毁灭自己的代码呢？

我认为并非如此。在第 1~3 章理解了类型和值到底是如何工作的后，第 4 章讨论了这个争议，并完整地解释了类型转换是如何工作的，包括所有的边边角角。

我们会看到，到底类型转换的哪些部分是出乎意料的，哪些部分在花费精力学习后则是完全可以理解的。

这不仅仅只是声称类型转换是合理的、可学习的；我想表明的是，类型转换是非常有用且被低估了的工具，**你应该在自己的代码中使用它**。在我看来，如果能够正确使用的话，类型转换不仅能够工作，而且也会让你的代码质量更高。所有的反对者和怀疑者肯定会嘲笑这样的立场，但我坚信这是提高你 JavaScript 水平的关键一点。

你只想人云亦云、随波逐流吗？还是你愿意将所有的假设放到一边，用全新的视角来观察类型转换？“类型与语法”部分将会转换你的思路。

3.4 异步和性能

参见《你不知道的 JavaScript（中卷）》第二部分。

“作用域和闭包”“this 和对象原型”以及“类型和语法”关注的都是语言的核心机制，而“异步和性能”则稍微偏重于在语言机制之上处理异步编程的模式。异步不只是对应用的性能至关重要，而且正在慢慢成为代码易写性和可维护性方面的关键因素。

“异步和性能”部分一开始明确了大量的术语和概念，如“异步”“并行”和“并发”这些概念，并深入解释了这些概念为什么适用或不适用于 JavaScript。

然后我们查看了回调这个使得异步成为可能的基本方法。但我们很快就看到，单独使用回

调完全不足以满足当代异步编程的需求。我们确定了两种只用回调编码的缺陷：**控制反转**（Inversion of Control，IoC）信任缺失和线性理解能力的缺失。

为了避免这两个主要的缺陷，ES6 引入了新的机制（实际上是模式）：promise 和生成器。

promise 是对“未来值”的与时间无关的封装，使得不管这个值是否已经可用，你都可以推导和组合使用它们。另外，通过一种可信任的、可组合的 promise 机制，分发回调它们也有效地解决了 IoC 信任问题。

生成器为 JavaScript 函数引入了一种新的执行模式，其中生成器可以暂停在 `yield` 点上，并在之后被异步继续。暂停与继续的能力使得生成器中同步的、看似连续的代码可以在后台异步执行。通过这种方式，我们解决了回调的非线性、非局部跳转引发的代码混乱问题，因而让我们的异步代码看似同步，更容易追踪。

但是，promise 和生成器的组合“暂缓”了我们最有效的异步编码模式进入 JavaScript 的日程。实际上，ES7 及更新版本中即将出现的异步的高级机制很大程度上是建立在这个基础上的。要想在异步的世界中严肃地对待程序效率，你需要非常熟悉 promise 和生成器的组合。

如果说 promise 和生成器与表达模式有关，这种模式使得我们的程序可以更加并发地运行，因此能在更短的时间内处理完毕，那么 JavaScript 还有很多其他的性能优化因素值得探讨。

第 5 章探讨了通过 Web Worker 实现程序并行和通过 SIMD 实现数据并行的主题，以及像 ASM.js 这样的底层优化技术。第 6 章从合适的测评技术角度介绍了性能优化，其中包括哪些类型的性能需要关注，哪些可以忽略。

编写高效的 JavaScript 代码意味着，你编写的代码可以打破不同浏览器和环境的壁垒，达到动态运行。这要求大量复杂而详细的计划和努力，只有这样，才能让程序从“可以运行”到“可以很好地运行”。

“异步和性能”部分的目的是，为你提供编写合理、高性能 JavaScript 代码所需要的所有工具和技巧。

3.5 ES6 及更新版本

参见本书第二部分。

不管你认为自己此时对 JavaScript 已经有了怎样的掌握，事实是 JavaScript 一直在持续发展，而且，发展的速度越来越快。这个事实几乎就是本系列精神的隐喻，你需要接受我们永远无法完全了解 JavaScript 这个事实，因为即使你掌握了所有内容，还是会出现你需要学习的新东西。

这个主题为这门语言的发展方向提供了短期和中期的展望，不只是 ES6 这样已知的方向，还有在其之后可能的方向。

本系列中的所有内容都是根据撰写时的 JavaScript 的状态编写的，即 ES6 的接受期。本系列的关注点更多在 ES5 上。现在，我们要将注意力放在 ES6、ES7 及更远的未来……

既然编写本系列时 ES6 已经接近完成，“ES6 及更新版本”部分一开始就将 ES6 中的具体内容分成了几个关键的类别，其中包括新语法、新数据结构（集合）以及新处理能力和 API。我们将介绍 ES6 的每一个新特性，详细程度有所不同，并且还会回顾本系列其他部分提到的细节。

预先列出令人兴奋的 ES6 特性：解构、默认参数值、符号、简洁方法、计算属性、箭头函数、块作用域、promise、生成器、迭代器、模块、代理、WeakMap，以及更多！啊，ES6 确实改进了不少！

“ES6 及更新版本”的前面是一个线路图，可以帮助你了解改进后的 JavaScript，这可是你在未来几年里需要编写和探索的 JavaScript。其后面简单关注了我们有把握在 JavaScript 的近期可期待的新特性。最重要的实现就是后 ES6，JavaScript 很可能是一个特性一个特性地演化，而不是一个版本一个版本地演化，这意味着这些近期将要出现的特性可能会比你想象的更快到来。

JavaScript 的未来是光明的。是时候开始学习了吧！

3.6 小结

“你不知道的 JavaScript”系列的宗旨是，所有的 JavaScript 开发者都可以，也应该学习这门伟大语言的方方面面。个人偏见、框架假定和项目的截止日期都不应该成为你从不学习和深入理解 JavaScript 的借口。

我们覆盖了这门语言中每个重要的焦点领域，完整地探索了所有你可能以为自己已经了解，却并没有完全理解的部分。

“你不知道的 JavaScript”既不是批判也不是攻击。这是一种领悟，是包括我自己在内的所有人都必须接受的事实。学习 JavaScript 不是最终目标，而是一个过程。我们还不了解 JavaScript。但我们终将做到这一点！

第二部分

ES6 及更新版本