序

Kyle Simpson 是一位缜密的务实主义者。

这是我能想到的最高赞美。对我来说，这是软件开发者必须具备的最重要的品质。对，是必须而不是应该。把 JavaScript 编程语言的各个层次梳理清楚，并以含义丰富且易于理解的形式呈现出来，Kyle 在这方面的敏锐能力是首屈一指的。

“你不知道的 JavaScript”系列的读者将会了解“ ES6 及更新版本”，从最直观的到那些我们一直认为理所当然或者从未想到的微妙语义，他们将会沉浸于其间。到目前为止，“你不知道的 JavaScript”系列图书覆盖的内容是读者至少在某种程度上已经熟悉的，看到或者听说过这些主题，甚至可能实践过。 而本书这一部分将要介绍的内容只有 JavaScript 开发者社区的很少一部分人有所了解，即 ECMAScript 2015 语言规范的发展变化。

过去几年里，我看到了 Kyle 不懈努力地学习熟悉这部分内容，最终达到了只有少数专业人士所能精通的程度。在他写作这部分内容时，ECMAScript 2015 语言规范文档还没有正式发布，能考虑到这一点是多么地了不起！我所说的千真万确，我阅读了 Kyle 为这一内容所写的每一个字。我还跟踪了他的所有修改，每一次修改都是对内容的改进，为我们提供了更深层次的认识。

这些内容展示了未知的新知识，以此来触发读者的思考，使你的知识储备与工具一起进化。它还会为读者增加信心来彻底拥抱 JavaScript 编程的下一个重要时代。

——Rick Waldron (@rwaldron)，Bocoup 公司开源 Web 开发者，Ecma/TC39 jQuery 代表

第 1 章

ES? 现在与未来

在深入阅读本部分之前，你应该已经能够熟练应用（直到编写本部分时）最新标准下的 JavaScript 工作了，这个标准通常被称为 ES5（严格说是 ES5.1）。本部分中，我们将会直接介绍 ES6，同时也会扩展视野，理解 JavaScript 未来的发展方向。

如果你对自己的 JavaScript 水平还不是那么有信心的话，我强烈建议你先阅读一下本系列《你不知道的 JavaScript（上卷）》和《你不知道的 JavaScript（中卷）》中的几部分内容。

- 作用域和闭包：你知道 JavaScript 的词法作用域是基于编译器（而非解释器！）语义的吗？你能解释词法作用域和作为值的函数这两者的直接结果之一就是闭包吗？
- `this` 和对象原型：你能复述 `this` 绑定的四条基本原则吗？你是否还在用 JavaScript 的“伪”类应付了事，而没有采用更简洁的“行为委托”设计模式？你听说过**连接到其他对象的对象**（objects linked to other objects，OLOO）吗？
- 类型和语法：你了解 JavaScript 中的内置类型吗？更重要的是，你了解如何正确安全地使用类型间强制转换吗？对于 JavaScript 语法 / 句法中的微妙细节，你的熟悉程度又如何？
- 异步和性能：你还在使用回调管理异步吗？你能解释 promise 是什么以及它为什么 / 如何能够解决“回调地狱”这个问题吗？你知道如何应用生成器来使得异步代码更加清晰吗？对 JavaScript 程序和具体运算的深度优化到底由哪些方面构成？
- 起步上路：你还是编程新手或者 JavaScript 新手吗？这是你起步之时需要了解的技术发展路线（本书第一部分）。

如果你阅读了前面所有内容，并对其覆盖的主题有了足够的信心，现在是我们深入探索 JavaScript 已经到来的及更远未来将会出现的改变的时候了。

与 ES5 不同，ES6 并不仅仅是为这个语言新增一组 API。它包括一组新的语法形式，其中的一部分可能是要花些时间才能理解和熟悉的。它还包括各种各样的新的组织形式和操作各种数据类型的新的辅助 API。

对于这个语言来说，ES6 是一次激进的飞跃。即使你认为自己对 JavaScript ES5 已经相当了解，ES6 还是有大量你不知道的全新内容。所以做好准备吧！这里探讨了你需要了解的 ES6 的所有重要主题，并且简单介绍了你可能还没有意识到的未来将会出现的特性。

本部分中的所有代码假定运行环境为 ES6+。在编写本部分的时候，各种浏览器和 JavaScript 环境（比如 Node.js）对 ES6 的支持程度差异巨大，所以运行结果可能也会有所差异。

1.1 版本

JavaScript 标准的官方名称是“ECMAScript”（简称“ES”），直到最近都是用有序数字来标识版本的，例如“5”表示“第 5 版”。

最早的 JavaScript 版本是 ES1 和 ES2，它们不怎么为人所知，实现也很少。第一个流行起来的 JavaScript 版本是 ES3，它成为浏览器 IE6-8 和早前的旧版 Android 2.x 移动浏览器的 JavaScript 标准。出于某些政治原因，倒霉的 ES4 从来没有成形，这里我们不做讨论。

2009 年，ES5 正式发布（然后是 2011 年的 ES5.1），在当代浏览器（包括 Firefox、Chrome、Opera、Safari 以及许多其他类型）的进化和爆发中成为 JavaScript 广泛使用的标准。

下一个 JavaScript 版本（发布日期从 2013 年拖到 2014 年，然后又到 2015 年）标签，之前的共识显然是 ES6。

但是，在 ES6 规范发展后期，出现了这样的方案：有人建议未来的版本应该改成基于年份，比如 ES2016（也就是 ES7）来标示在 2016 年结束之前敲定的任何版本的规范。尽管有异议，但比起后来提出的方案 ES2015，很可能保持统治地位的版本命名仍是 ES6。而 ES2016 可能会采用新的基于年份的命名方案。

人们已经观察到，JavaScript 的发展步伐要比每年一个版本快得多。一旦一个想法进入标准讨论阶段，浏览器就会开始为这个特性开发原型，早期的使用者也就会开始编码进行试验了。

对于一个特性来说，通常在官方正式批准之前很久，通过早期的引擎 / 工具原型，这个特性就已经事实上标准化了。所以，把 JavaScript 未来的版本看成基于单个特性，而不是基于某一组主要特性组合为单位（就像现在）的新版本，甚至是每年一个版本（就像以后采用的形式），也是合理的。

这带来的结果是，版本标签已经不再那么重要，JavaScript 开始被看作更像是一个发展的动态的标准。对待这一结果最好的办法是别再把你的代码看作是基于版本的，比如“基于 ES6”，而是去考虑它具体支持哪些单独的特性。

1.2 transpiling

特性演变迅速也让 JavaScript 开发者遇到了问题，他们非常想要使用新特性，但同时又被现实所困，他们的网站 /app 可能需要支持并未提供这些新特性的旧版浏览器，而功能特性的快速进化使得这个问题更加严峻。

ES5 广泛应用于工业界的过程中，其典型思路是要等到绝大多数——如果不说所有——前 ES5 环境都不再需要被支持的时候，代码才采用 ES5。结果是，很多实现才开始（在编写本部分的时候）使用像 `strict` 模式这样的特性，而这早在五年以前就已经出现在 ES5 中了。

人们普遍认为，落后规范这么多年对于 JavaScript 生态系统的未来是有害的。所有负责语言发展的人士都希望，新的特性和模式一旦在标准中稳定下来，并且浏览器能够实现它们之后，就能够在开发者的代码中得到应用。

所以我们如何解决这个矛盾呢？答案是工具化（提供工具）。具体来说是一种称为 transpiling（transformation + compiling，转换 + 编译）的技术。简单地说，其思路是利用专门的工具把你的 ES6 代码转化为等价（或近似！）的可以在 ES5 环境下工作的代码。

举例来说，考虑短路属性定义（参见 2.6 节）。下面是 ES6 形式：

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

var obj = {
    foo     // 也就是foo: foo
};

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

而下面是转化后的（大致）代码：

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

var obj = {
    foo: foo
};

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

这个变换很小但能给人带来方便，使我们在同名的时候，可以把对象字面声明的 `foo: foo` 简写为 `foo`。

通常在构建过程中使用 transpiler 执行这些转换，如同执行 linting、minification，或者其他类似操作的步骤。

shim/polyfill

并非所有的 ES6 新特性都需要使用 transpiler，还有 polyfill（也称为 shim）这种模式。在可能的情况下，polyfill 会为新环境中的行为定义在旧环境中的等价行为。语法不能 polyfill，而 API 通常可以。

举例来说，`Object.is(..)` 是一个用于检查两个值严格相等的新工具，而且不像 `===` 那样在处理 `NaN` 和 `-0` 值的时候有微妙的例外情况。对 `Object.is(..)` 应用 polyfill 非常简单：

```
if (!Object.is) {
    Object.is = function(v1, v2) {
        // 检查-0
        if (v1 === 0 && v2 === 0) {
            return 1 / v1 === 1 / v2;
        }
        // 检查NaN
        if (v1 !== v1) {
            return v2 !== v2;
        }
        // 其余所有情况
        return v1 === v2;
    };
}
```

注意这个 polyfill 外层用于保护的 `if` 语句。这个细节很重要，它表示这段代码只定义了在未定义 API 的旧环境下的行为；需要覆盖已经存在的 API 的情况是非常罕见的。

这里有一组名为“ES6 Shim”的 ES6 shim 实现（https:// github.com/paulmillr/es6-shim/），你一定要把它作为一个标准放在你所有的 JavaScript 新项目中。

人们认为 JavaScript 会持续不断地发展，浏览器会逐渐地而不是以大规模突变的形式支持新特性。所以，保持 JavaScript 发展更新的最好战略就是在你的代码中引入 polyfill shim，并且在构建过程中加入 transpiler 步骤，现在就开始接受并习惯这个新现实吧。

如果还要保持现状，等着所有浏览器都支持某个特性才开始应用这个特性，那么你就已经落后了。你将遗憾地错过所有设计用于使得编写 JavaSript 更高效健壮的创新。

1.3 小结

在编写本部分时，ES6（有些人可能想要称其为 ES2015）刚刚出现，其中有很多新的技术需要学习！

但是更重要的是，你要转变你的思路以符合 JavaScript 新的发展方式。不要再像过去很多人所做的那样，等待多年直到某个正式文档投票通过（才开始应用这个新特性）。

现在，JavaScript 的新特性一旦可行就会进入浏览器，因此是早早步入正轨还是多年以后再来追赶由你自己决定。

不管未来的 JavaScript 采用何种版本标识，它的发展都会比过去要快得多。transpiler 和 shim/polyfill 都是重要的工具，它们能帮助你保持处于这个语言发展的最前沿。

如果说 JavaScript 发展的新图景中有任何要点需要了解的话，就是现在所有的 JavaScript 开发者都被强烈要求从追踪发展的状态转换为直接站在最前沿。那么，学习 ES6 就是开始的第一步！

第 2 章

语法

有一点比较奇怪的是，不管你编写 JavaScript 代码的时间有多长，都会觉得语法对你来说非常熟悉。虽然确实有一些怪异的地方，但总的来说，JavaScript 语法非常合理自然，它借鉴于其他一些语言，和它们有很多相似之处。

然而，ES6 新增了很多新的语法形式，需要我们去熟悉。本章我们将会介绍这些新的语法形式，看看它们提供了哪些新东西。

在编写本部分时，书里讨论的特性中有一些已经被各种浏览器（Firefox、Chrome 等）所支持，但还有一些只是部分实现，其他甚至完全没有实现。直接试验这些示例可能会出现各种结果。如果是这样的话，可以通过 transpiler 来试验，因为这些特性中的绝大多数都已经被此类工具所支持。

ES6Fiddle（http://www.es6fiddle.net/）是一个非常不错的、易于使用的 ES6 试验田，Babel transpiler（http://babeljs.io/repl/）的在线 REPL 也是这样。

2.1 块作用域声明

你很可能已经了解，JavaScript 中变量作用域的基本单元一直是 `function`。如果需要创建一个块作用域，最普遍的方法除了普通的函数声明之外，就是立即调用函数表达式（IIFE）。举例来说：

```
var a = 2;

(function IIFE(){
```

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

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

2.1.1 let 声明

但现在，我们可以创建绑定到任意块的声明，（不出意外地）其被称为**块作用域**（block scoping）。这意味着我们只需要一对 { .. } 就可以创建一个作用域。不像使用 var 那样声明的变量总是归属于包含函数（即全局，如果在最顶层的话）作用域，而是像下面这样使用 let：

```
var a = 2;

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

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

在 JavaScript 中使用独立的 { .. } 还不是很常见的惯用法，但总是合法的。如果开发者有其他支持块作用域语言的经验，会很容易认出这种模式。

我认为这种使用一个专门的 { .. } 块的模式是创建块作用域变量的最好方法。另外，应该总是把 let 声明放在块的最前面。如果有多个变量需要声明的话，建议只用一个 let。

从编码风格上来说，我甚至愿意把 let 和左括号 { 放在同一行，这样更明确地表明了这个块的目的只是为了声明这些变量的作用域。

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

这样看起来很奇怪，也不大可能符合其他大多数 ES6 文献中推荐的语法形式。但是我的疯狂是有原因的。

let 声明还有一个试验性（未标准化的）的形式，称为 let 块，看起来就像这样：

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

我把这种形式称为**显式**块作用域，而镜像于 var 的那种 let .. 声明形式更加**隐式**，因为它某种程度上“劫持”了所在的 { .. } 中的一切。一般来说，开发者喜欢**显式**机制胜过**隐式**机制，我认为这里也是这样。

比较前面的两个代码片段，它们是非常相似的，我认为这两种机制风格上都称得上是**显式**块作用域。不幸的是，其中最显式的 `let (..) { .. }` 形式并没有被 ES6 采用。ES6 之后可能会再次考虑这个选择，但是现在来说，我认为前者是最好的选择。

为了强调说明 `let ..` 声明的**隐式**本质，考虑下面这种用法：

```
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
}
```

不要回头去看前面的代码，快速回答：哪个（些）变量只存在于 `if` 语句内部，哪个（些）变量只存在于 `for` 循环内部？

答案：`if` 语句包含了块作用域变量 `b` 和 `c`，块作用域变量 `i` 和 `j` 存在于 `for` 循环之中。

你是否需要思考一会呢？`i` 并不在包含它的 `if` 语句作用域中，这一点是否让你吃惊呢？这种疑惑和思考，我称之为“脑力税”，来自于 `let` 机制对于我们来说不只是新的，同时也是隐式的这个事实。

作用域内部后面才出现的 `let c = ..` 声明也有隐患。和传统的 `var` 声明变量不同，不管出现在什么位置，`var` 都是归属于包含它的整个函数作用域。`let` 声明归属于块作用域，但是直到在块中出现才会被初始化。

在 `let` 声明 / 初始化之前访问 `let` 声明的变量会导致错误，而使用 `var` 的话这个顺序是无关紧要的（除了代码风格方面）。

考虑：

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

    var a;
    let b;
}
```

过早访问 let 声明的引用导致的这个 ReferenceError 严格说叫作**临时死亡区**（Temporal Dead Zone，TDZ）错误——你在访问一个已经声明但没有初始化的变量。以后我们还会遇到 TDZ 错误——ES6 中它出现了多次。另外，注意“初始化”并不要求代码中的显式赋值，比如 `let b`；这样完全是有效的。声明时没有赋值的变量会自动赋值为 undefined，所以 `let b;` 就等价于 `let b = undefined;`。不管是否显式赋值，都不能在 `let b` 语句运行之前访问 b。

最后一点：对于 TDZ 值和未声明值（以及声明过的！），typeof 结果是不同的。举例来说：

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

    // b声明了，但还处于TDZ
    if (typeof b === "undefined") {    // ReferenceError!
        // ..
    }

    // ..

    let b;
}
```

这里 a 是未声明的，所以 typeof 是检查它是否存在的唯一安全的方法。而 `typeof b` 会抛出 TDZ 错误，因为代码中在后面恰好有一个 `let b` 声明。

现在能更清楚为什么我坚持认为应该把所有的 let 声明放在其所在作用域的最前面了吧。这样就完全避免了不小心过早访问的问题。这也使得阅读这个块，以及所有块代码开头的时候能够更清楚地了解这块代码包含了哪些变量。

你的代码块（if 语句、while 循环等）不需要与块行为方式共享原来的行为方式。

完全由你来选择是否遵守这个规则来提高你的代码的明晰性，而这一点将会节省你大量的重构精力，避免自作自受。

关于 let 和块作用域的更多信息，参见本系列《你不知道的 JavaScript（上卷）》第一部分的第 3 章。

`let +for`

我建议使用 let 声明块的显式形式，唯一的例外是 let 出现在 for 循环头部的情况。原因可能有点微妙，但是我认为这是 ES6 的重要特性之一。