第 1 章

让自己习惯 JavaScript

JavaScript 最初设计令人感觉亲切。由于其语法让人联想到 Java，并且具有许多脚本语言的共同特性（如函数、数组、字典和正则表达式），因此，具有少量编程经验的人也能够快速学习 JavaScript。新手程序员几乎不需要培训就可以开始编写程序，这要归功于 JavaScript 语言提供的为数不多的核心概念。

虽然 JavaScript 是如此的平易近人，但是精通这门语言需要更多的时间，需要更深入地理解它的语义、特性以及最有效的习惯用法。本书每个章节都涵盖了高效 JavaScript 编程的不同主题。第 1 章主要讲述一些最基本的主题。

第 1 条：了解你使用的 JavaScript 版本

像大多数成功的技术一样，JavaScript 已经发展了一段时间。最初 JavaScript 作为 Java 在交互式网页编程方面的补充而推向市场，但它最终完全取代了 Java 成为主流的 Web 编程语言。JavaScript 的普及使得其于 1997 年正式成为国际标准，其官方名称为 ECMAScript[⊖]。目前许多 JavaScript 的竞争实现都提供了 ECMAScript 标准的各种版本的一致性。

1999 年定稿的第 3 版 ECMAScript 标准（通常简称为 ES3），目前仍是最广泛采用的 JavaScript 版本。下一个有重大改进的标准是 2009 年发布的第 5 版，即 ES5。ES5 引入了一些新的特性，并且标准化了一些受到广泛支持但之前未规范的特性。由于 ES5 目前还未得到广泛支持，所以我会适时指出本书中的条款或建议是否特定于 ES5。

除了 ECMAScript 标准存在多个版本之外，还存在一些 JavaScript 实现支持非标准特性，而其他的 JavaScript 实现却并不支持这些特性的情况。例如，许多 JavaScript 引擎支持 const

⊖ ECMAScript 是一种由欧洲计算机制造商协会（ECMA）通过 ECMA-262 标准化的脚本程序设计语言。——译者注

关键字定义变量，但 ECMAScript 标准并没有定义任何关于 const 关键字的语义和行为。此外，在不同的实现之间，const 关键字的行为也不一样。在某些情况下，const 关键字修饰的变量不能被更新。

```
const PI = 3.141592653589793;
PI = "modified!";
PI; // 3.141592653589793
```

而其他的实现只是简单地将 const 视为 var 的代名词。

```
const PI = 3.141592653589793;
PI = "modified!";
PI; // "modified!"
```

由于 JavaScript 历史悠久且实现多样化，因此我们很难确定哪些特性在哪些平台上是可用的。而令事态更加严峻的事实是 JavaScript 的主要生态系统——Web 浏览器，它并不支持让程序员指定某个 JavaScript 的版本来执行代码。由于最终用户可能使用不同 Web 浏览器的不同版本，因此，我们必须精心地编写 Web 程序，使得其在所有的浏览器上始终工作如一。

另外，JavaScript 并不只是针对客户端 Web 编程。JavaScript 的其他应用包括服务器端程序、浏览器扩展以及针对移动和桌面应用程序的脚本。某些情况下你可能需要一个特定的 JavaScript 版本。对于这些情况，利用特定平台支持的 JavaScript 特定实现的额外特性是有意义的。

本书主要关注的是 JavaScript 的标准特性，但是也会讨论一些广泛支持的非标准特性。当涉及新标准特性或非标准特性时，了解你的应用程序运行环境是否支持这些特性是至关重要的。否则，你可能会面临这样的困境——应用程序在你自己的计算机或者测试环境上运行良好，但是将它部署在不同的产品环境中时却无法运行。例如，const 关键字在支持非标准特性的 JavaScript 引擎上测试时运行良好，但是，当将它部署在不识别 const 关键字的 Web 浏览器上时就会出现语法错误。

ES5 引入了另一种版本控制的考量——严格模式（strict mode）。此特性允许你选择在受限制的 JavaScript 版本中禁止使用一些 JavaScript 语言中问题较多或易于出错的特性。由于其语法设计向后兼容，因此即使在那些没有实现严格模式检查的环境中仍然可以执行严格代码[⊖]（strict code）。在程序中启用严格模式的方式是在程序的最开始增加一个特定的字符串字面量（literal）。

```
"use strict";
```

同样，你也可以在函数体的开始处加入这句指令以启用该函数的严格模式。

⊖ 严格代码是指期望运行于严格模式下的代码。——译者注

```
function f(x) {
    "use strict";
    // ...
}
```

使用字符串字面量作为指令语法看起来有点怪异，但它的好处是向后兼容。由于解释执行字符串字面量并没有任何副作用，所以 ES3 引擎执行这条指令是无伤大雅的。ES3 引擎解释执行该字符串，然后立即丢弃其值。这使得编写的严格模式的代码可以运行在旧的 JavaScript 引擎上，但有一个重要的限制：旧的引擎不会进行任何的严格模式检查。如果你没有在 ES5 环境中做过测试，那么，编写的代码运行于 ES5 环境中就很容易出错。

```
function f(x) {
    "use strict";
    var arguments = []; // error: redefinition of arguments
    // ...
}
```

在严格模式下，不允许重定义 arguments 变量，但没有实现严格模式检查的环境会接受这段代码。然而，这段代码部署在实现 ES5 的产品环境中将导致程序出错。所以，你应该总是在完全兼容 ES5 的环境中测试严格代码。

“use strict”指令只有在脚本或函数的顶部才能生效，这也是使用严格模式的一个陷阱。这样，脚本连接变得颇为敏感。对于一些大型的应用软件，在开发中使用多个独立的文件，然而部署到产品环境时却需要连接成一个单一的文件。例如，想将一个文件运行于严格模式下：

```
// file1.js
"use strict";
function f() {
    // ...
}
// ...
```

而另一个文件不是运行于严格模式下：

```
// file2.js
// no strict-mode directive
function g() {
    var arguments = [];
    // ...
}
// ...
```

我们怎样才能正确地连接这两个文件呢？如果我们以 file1.js 文件开始，那么连接后的代码运行于严格模式下：

```
// file1.js
"use strict";
function f() {
    // ...
}
// ...
// file2.js
// no strict-mode directive
function f() {
    var arguments = []; // error: redefinition of arguments
    // ...
}
// ...
```

如果我们以 file2.js 文件开始，那么连接后的代码运行于非严格模式下：

```
// file2.js
// no strict-mode directive
function g() {
    var arguments = [];
    // ...
}
// ...
// file1.js

"use strict";
function f() { // no longer strict
    // ...
}
// ...
```

在自己的项目中，你可以坚持只使用“严格模式”或只使用“非严格模式”的策略，但如果你要编写健壮的代码应对各种各样的代码连接，你有两个可选的方案。

第一个解决方案是不要将进行严格模式检查的文件和不进行严格模式检查的文件连接起来。这可能是最简单的解决方案，但它无疑会限制你对应用程序或库的文件结构的控制力。在最好的情况下，你至少要部署两个独立的文件。一个包含所有期望进行严格检查的文件，另一个则包含所有无须进行严格检查的文件。

第二个解决方案是通过将其自身包裹在立即调用的函数表达式（Immediately Invoked Function Expression, IIFE）中的方式连接多个文件。第 13 条将对立即调用的函数表达式进行深入的讲解。总之，将每个文件的内容包裹在一个立即调用的函数中，即使在不同的模式下，它们都将被独立地解释执行。基于此方案，上面例子的连接版本如下：

```
// no strict-mode directive
(function() {
```

```
    // file1.js
    "use strict";
    function f() {
        // ...
    }
    // ...
})();
(function() {
    // file2.js
    // no strict-mode directive
    function f() {
        var arguments = [];
        // ...
    }
    // ...
})();
```

由于每个文件的内容被放置在一个单独的作用域中，所以使用严格模式指令（或者不使用严格模式指令）只影响本文件的内容。但是这种方式会导致这些文件的内容不会在全局作用域内解释。例如，var 和 function 声明的变量不会被视为全局变量（更多关于全局概念的内容参见第 8 条）。这恰好与流行的模块系统（module system）类似，模块系统通过自动地将每个模块的内容放置在单独的函数中的方式来管理文件和依赖。由于所有文件都放置在局部作用域内，所以每个文件都可以自行决定是否要使用严格模式。

编写文件使其在两种模式下行为一致。想要编写一个库，使其可以工作在尽可能多的环境中，你不能假设库文件会被脚本连接工具置于一个函数中，也不能假设客户端的代码库是否处于严格模式或者非严格模式。要想构建代码以获得最大的兼容性，最简单的方法是在严格模式下编写代码，并显式地将代码内容包裹在本地启用了严格模式的函数中。这种方式类似于前面描述的方案——将每个文件的内容包裹在一个立即调用的函数表达式中，但在这种情况下，你是自己编写立即调用的函数表达式并且显式地选择严格模式，而不是采用脚本连接工具或模块系统帮你实现。

```
(function() {
    "use strict";
    function f() {
        // ...
    }
    // ...
})();
```

要注意的是，无论这段代码是在严格模式还是在非严格模式的环境中连接的，它都被视为是严格的。相比之下，即使一个函数没有选择严格模式，如果它连接在严格代码之后，它

仍被视为是严格的。所以，为了达到更为普遍的兼容性，建议在严格模式下编写代码。

提示

- 决定你的应用程序支持 JavaScript 的哪些版本 。
- 确保你使用的任何 JavaScript 的特性对于应用程序将要运行的所有环境都是支持的。
- 总是在执行严格模式检查的环境中测试严格代码。
- 当心连接那些在不同严格模式下有不同预期的脚本。

第 2 条：理解 JavaScript 的浮点数

大多数编程语言都有几种数值型数据类型，但是 JavaScript 却只有一种。你可以使用 typeof 运算符查看数字的类型。不管是整数还是浮点数，JavaScript 都将它们简单地归类为数字。

```
typeof 17;   // "number"
typeof 98.6; // "number"
typeof -2.1; // "number"
```

事实上，JavaScript 中所有的数字都是双精度浮点数。这是由 IEEE[⊖]754 标准制定的 64 位编码数字——即“ doubles”。如果这一事实使你疑惑 JavaScript 是如何表示整数的，请记住，双精度浮点数能完美地表示高达 53 位精度的整数。从 –9 007 199 254 740 992（-2^{53}）到 9 007 199 254 740 992（2^{53}）的所有整数都是有效的双精度浮点数。因此，尽管 JavaScript 中缺少明显的整数类型，但是完全可以进行整数运算。

大多数的算术运算符可以使用整数、实数或两者的组合进行计算。

```
0.1 * 1.9  // 0.19
-99 + 100; // 1
21 - 12.3; // 8.7
2.5 / 5;   // 0.5
21 % 8;    // 5
```

然而位算术运算符比较特殊。JavaScript 不会直接将操作数作为浮点数进行运算，而是会将其隐式地转换为 32 位整数后进行运算。（确切地说，它们被转换为 32 位大端（big-endian）的 2 的补码表示的整数。）以按位或运算表达式为例：

```
8 | 1; // 9
```

看似简单的表达式实际上需要几个步骤来完成运算。如前所述，JavaScript 中的数字 8 和 1 都是双精度浮点数。但是它们也可以表示成 32 位整数，即 32 位 0、1 的序列。整数 8 表示

⊖ 美国电气和电子工程师协会（Institute of Electrical and Electronics Engineers）是一个国际性的电子技术与信息科学工程师的协会，是世界上最大的专业技术组织之一。——译者注

为 32 位二进制序列如下所示：

```
00000000000000000000000000001000
```

你自己也可以使用数字类型的 toString 方法来查看：

```
(8).toString(2); // "1000"
```

toString 方法的参数指定了其转换基数，此例子以基数 2（即二进制）表示。结果值省略了左端多余的 0（位），因为它们并不影响最终值。

整数 1 表示为 32 位二进制如下所示：

```
00000000000000000000000000000001
```

按位或运算表达式合并两个比特序列。只要参与运算的两位比特中任意一位为 1，运算结果的该位就为 1。以位模式表示的结果如下：

```
00000000000000000000000000001001
```

这个序列表示整数 9。你可以使用标准的库函数 parseInt 验证，同样以 2 作为基数：

```
parseInt("1001", 2); // 9
```

（同样，前导 0 位是不必要的，因为它们并不影响运算结果。）

所有位运算符的工作方式都是相同的。它们将操作数转换为整数，然后使用整数位模式进行运算，最后将结果转换为标准的 JavaScript 浮点数。一般情况下，JavaScript 引擎需要做些额外的工作来进行这些转换。因为数字是以浮点数存储的，必须将其转换为整数，然后再转换回浮点数。然而，在某些情况下，算术表达式甚至变量只能使用整数参与运算，优化编译器有时候可以推断出这些情形而在内部将数字以整数的方式存储以避免多余的转换。

关于浮点数的最后警示是，你应该对它们保持时刻警惕。浮点数看似熟悉，但是它们是出了名的不精确。甚至一些看起来最简单的算术运算都会产生不正确的结果。

```
0.1 + 0.2; // 0.30000000000000004
```

尽管 64 位的精度已经相当高了，但是双精度浮点数也只能表示一组有限的数字，而不能表示所有的实数集。浮点运算只能产生近似的结果，四舍五入到最接近的可表示的实数。当你执行一系列的运算，随着舍入误差的积累，运算结果会越来越不精确。舍入也会使我们通常所期望的算术运算定律产生一些出人意料的偏差。例如，实数满足结合律，这意味着，对于任意的实数 x，y，z，总是满足 $(x + y) + z = x + (y + z)$。

然而，对于浮点数来说，却并不总是这样。

```
(0.1 + 0.2) + 0.3; // 0.6000000000000001
0.1 + (0.2 + 0.3); // 0.6
```

浮点数权衡了精度和性能。当我们关心精度时，要小心浮点数的局限性。一个有效的解

决方法是尽可能地采用整数值运算，因为整数在表示时不需要舍入。当进行货币相关的计算时，程序员通常会按比例将数值转换为最小的货币单位来表示再进行计算，这样就可以以整数进行计算。例如，如果上面的计算是以美元为单位，那么，我们可以将其转换为整数表示的美分进行计算。

```
(10 + 20) + 30; // 60
10 + (20 + 30); // 60
```

对于整数运算，你不必担心舍入误差，但是你还是要当心所有的计算只适用于 -2^{53} ～ 2^{53} 的整数。

提示

- JavaScript 的数字都是双精度的浮点数。
- JavaScript 中的整数仅仅是双精度浮点数的一个子集，而不是一个单独的数据类型。
- 位运算符将数字视为 32 位的有符号整数。
- 当心浮点运算中的精度陷阱。

第 3 条：当心隐式的强制转换

JavaScript 对类型错误出奇宽容。许多语言都认为表达式

```
3 + true; // 4
```

是错误的，因为布尔表达式（如 true）与算术运算是不兼容的。在静态类型语言中，含有类似这样表达式的程序甚至不会被允许运行。在一些动态类型语言中，含有类似这样表达式的程序可以运行，但是会抛出一个异常。然而，JavaScript 不仅允许程序运行，而且还会顺利地产生结果 4！

在 JavaScript 中有一些极少数的情况，提供错误的类型会产生一个即时错误。例如，调用一个非函数对象 (nonfunction) 或试图选择 null 的属性。

```
"hello"(1); // error: not a function
null.x;     // error: cannot read property 'x' of null
```

但是在大多数情况下，JavaScript 不会抛出一个错误，而是按照多种多样的自动转换协议将值强制转换为期望的类型。例如，算术运算符 -、*、/ 和 % 在计算之前都会尝试将其参数转换为数字。而运算符 + 更为微妙，因为它既重载了数字相加，又重载了字符串连接操作。具体是数字相加还是字符串连接，这取决于其参数的类型。

```
2 + 3;              // 5
"hello" + " world"; // "hello world"
```

接下来，合并一个数字和一个字符串会发生什么呢？ JavaScript 打破了这一束缚，它更偏爱字符串，将数字转换为字符串。

```
"2" + 3; // "23"
2 + "3"; // "23"
```

类似这样的混合表达式有时令人困惑，因为 JavaScript 对操作顺序是敏感的。例如，表达式：

```
1 + 2 + "3";    // "33"
```

由于加法运算是自左结合的（即左结合律），因此，它等同于下面的表达式：

```
(1 + 2) + "3";  // "33"
```

与此相反，表达式：

```
1 + "2" + 3;    // "123"
```

的计算结果为字符串“123”。左结合律相当于是将表达式左侧的加法运算包裹在括号中。

```
(1 + "2") + 3;  // "123"
```

位运算符不仅会将操作数转换为数字，而且还会将操作数转换为 32 位整数（表示的数字的子集）。我们在第 2 条已经讨论过。这些运算符包括位算术运算符（~、&、^ 和 |）以及移位运算符（<<、>> 和 >>>）。

这些强制转换十分方便。例如，来自用户输入、文本文件或者网络流的字符串都将被自动转换。

```
"17" * 3;  // 51
"8" | "1"; // 9
```

但是强制转换也会隐藏错误。结果为 null 的变量在算术运算中不会导致失败，而是被隐式地转换为 0；一个未定义的变量将被转换为特殊的浮点数值 NaN（自相矛盾地命名为“not a number”。谴责 IEEE 浮点数标准！）。这些强制转换不是立即抛出一个异常，而是继续运算，往往导致一些令人困惑和不可预测的结果。无奈的是，即便是测试 NaN 值也是异常困难的。这有两个原因。第一，JavaScript 遵循了 IEEE 浮点数标准令人头痛的要求——NaN 不等于其本身。因此，测试一个值是否等于 NaN 根本行不通。

```
var x = NaN;
x === NaN;    // false
```

另外，标准的库函数 isNaN 也不是很可靠，因为它带有自己的隐式强制转换，在测试其参数之前，会将参数转换为数字（isNaN 函数的一个更精确的名称可能是 coercesToNaN）。如果你已经知道一个值是数字，你可以使用 isNaN 函数测试它是否是 NaN。

```
isNaN(NaN); // true
```

但是对于其他绝对不是 NaN，但会被强制转换为 NaN 的值，使用 isNaN 方法是无法区分的。

```
isNaN("foo");               // true
isNaN(undefined);           // true
isNaN({});                  // true
isNaN({ valueOf: "foo" }); // true
```

幸运的是，有一个既简单又可靠的习惯用法用于测试 NaN，虽然稍微有点不直观。由于 NaN 是 JavaScript 中唯一一个不等于其自身的值，因此，你可以随时通过检查一个值是否等于其自身的方式来测试该值是否是 NaN。

```
var a = NaN;
a !== a;                    // true
var b = "foo";
b !== b;                    // false
var c = undefined;
c !== c;                    // false
var d = {};
d !== d;                    // false
var e = { valueOf: "foo" };
e !== e;                    // false
```

你也可以将这种模式抽象为一个清晰命名的实用工具函数。

```
function isReallyNaN(x) {
    return x !== x;
}
```

其实测试一个值是否与其自身相等是非常简洁的，通常没有必要借助于一个辅助函数，但关键在于识别和理解。

隐式的强制转换使得调试一个出问题的程序变得令人异常沮丧，因为它掩盖了错误，使错误更难以诊断。当一个计算出了问题，最好的调试方式是检查这个计算的中间结果，回到出错前的“最后一点”。在那里，你可以检查每个操作的参数，查看错误类型的参数。根据错误的不同，它可能是一个逻辑错误（如使用了错误的算术运算符），也可能是一个类型错误（如传入了一个 undefined 的值而不是数字）。

对象也可以被强制转换为原始值。最常见的用法是转换为字符串。

```
"the Math object: " + Math; // "the Math object: [object Math]"
"the JSON object: " + JSON; // "the JSON object: [object JSON]"
```

对象通过隐式地调用其自身的 toString 方法转换为字符串。你可以调用对象的 toString 方法进行测试。

```
Math.toString(); // "[object Math]"
JSON.toString(); // "[object JSON]"
```

类似地，对象也可以通过其 valueOf 方法转换为数字。通过定义类似下面这些方法，你可以控制对象的类型转换。

```
"J" + { toString: function() { return "S"; } }; // "JS"
2 * { valueOf: function() { return 3; } };      // 6
```

再一次，当你认识到运算符 + 被重载来实现字符串连接和加法时，事情变得棘手起来。特别是，当一个对象同时包含 toString 和 valueOf 方法时，运算符 + 应该调用哪个方法并不明显——做字符串连接还是加法应该根据参数的类型，但是存在隐式的强制转换，因此类型并不是显而易见！ JavaScript 通过盲目地选择 valueOf 方法而不是 toString 方法来解决这种含糊的情况。但是，这就意味着如果有人打算对一个对象执行字符串连接操作，那么产生的行为将会出乎意料。

```
var obj = {
    toString: function() {
        return "[object MyObject]";
    },
    valueOf: function() {
        return 17;
    }
};
"object: " + obj; // "object: 17"
```

这个例子的说明，valueOf 方法才真正是为那些代表数值的对象（如 Number 对象）而设计的。对于这些对象，toString 和 valueOf 方法应返回一致的结果（相同数字的字符串或数值表示），因此，不管是对象的连接还是对象的相加，重载的运算符 + 总是一致的行为。一般情况下，字符串的强制转换远比数字的强制转换更常见、更有用。最好避免使用 valueOf 方法，除非对象的确是一个数字的抽象，并且 obj.toString() 能产生一个 obj.valueOf() 的字符串表示。

最后一种强制转换有时称为真值运算（truthiness）。if、|| 和 && 等运算符逻辑上需要布尔值作为操作参数，但实际上可以接受任何值。JavaScript 按照简单的隐式强制转换规则将值解释为布尔值。大多数的 JavaScript 值都为真值（truthy），也就是能隐式地转换为 true。对于字符串和数字以外的其他对象，真值运算不会隐式调用任何强制转换方法。JavaScript 中有 7 个假值：false、0、−0、""、NaN、null 和 undefined。其他所有的值都为真值。由于数字和字符串可能为假值，因此，使用真值运算检查函数参数或者对象属性是否已定义不是绝对安全的。例如，一个带有默认值的接受可选参数的函数：

```
function point(x, y) {
    if (!x) {
        x = 320;
    }

     if (!y) {
         y = 240;
     }
     return { x: x, y: y };
 }
```

此函数忽略任何为假值的参数，包括 0：

```
point(0, 0); // { x: 320, y: 240 }
```

检查参数是否为 undefined 更为严格的方式是使用 typeof。

```
function point(x, y) {
    if (typeof x === "undefined") {
        x = 320;
    }
    if (typeof y === "undefined") {
        y = 240;
    }
    return { x: x, y: y };
}
```

此版本的 point 函数可以正确地识别 0 和 undefined。

```
point();     // { x: 320, y: 240 }
point(0, 0); // { x: 0, y: 0 }
```

另一种方式是与 undefined 进行比较。

```
if (x === undefined) { ... }
```

第 54 条将讨论针对库和 API 设计的真值运算测试的影响。

提示

- 类型错误可能被隐式的强制转换所隐藏。
- 重载的运算符 + 是进行加法运算还是字符串连接操作取决于其参数类型。
- 对象通过 valueOf 方法强制转换为数字，通过 toString 方法强制转换为字符串。
- 具有 valueOf 方法的对象应该实现 toString 方法，返回一个 valueOf 方法产生的数字的字符串表示。
- 测试一个值是否为未定义的值，应该使用 typeof 或者与 undefined 进行比较而不是使用真值运算。

第 4 条：原始类型优于封装对象

除了对象之外，JavaScript 有 5 个原始值类型：布尔值、数字、字符串、null 和 undefined。（令人困惑的是，对 null 类型进行 typeof 操作得到的结果为“ object”，然而，ECMAScript 标准描述其为一个独特的类型。）同时，标准库提供了构造函数来封装布尔值、数字和字符串作为对象。你可以创建一个 String 对象，该对象封装了一个字符串值。

```
var s = new String("hello");
```

在某些方面，String 对象的行为与其封装的字符串值类似。你可以通过将它与另一个值连接来创建字符串。

```
s + " world"; // "hello world"
```

你也可以提取其索引的子字符串。

```
s[4]; // "o"
```

但是不同于原始的字符串，String 对象是一个真正的对象。

```
typeof "hello"; // "string"
typeof s;       // "object"
```

这是一个重要的区别，因为这意味着你不能使用内置的操作符来比较两个截然不同的 String 对象的内容。

```
var s1 = new String("hello");
var s2 = new String("hello");
s1 === s2; // false
```

由于每个 String 对象都是一个单独的对象，其总是只等于自身。对于非严格相等运算符，结果同样如此。

```
s1 == s2; // false
```

由于这些封装的行为并不十分正确，所以用处不大。其存在的主要理由是它们的实用方法。结合另外的隐式强制转换，JavaScript 使得我们可以方便地使用这些实用方法因为这里有另一个隐式转换：当对原始值提取属性和进行方法调用时，它表现得就像已经使用了对应的对象类型封装了该值一样。例如，String 的原型对象有一个 toUpperCase 方法，可以将字符串转换为大写。你可以对原始字符串值调用这个方法。

```
"hello".toUpperCase(); // "HELLO"
```

这种隐式封装的一个奇怪后果是你可以对原始值设置属性，但是对其丝毫没有影响。

```
"hello".someProperty = 17;
"hello".someProperty; // undefined
```

因为每次隐式封装都会产生一个新的 String 对象，更新第一个封装对象并不会造成持久的影响。对原始值设置属性的确是没有意义的，但是觉察到这种行为是值得的。事实证明，这是 JavaScript 隐藏类型错误的又一种情形。本来你想给一个对象设置属性，但没注意其实它是个原始值，程序只是忽略更新而继续运行。这容易导致一些难以发现的错误，并且难以诊断。

提示

- 当做相等比较时，原始类型的封装对象与其原始值行为不一样。
- 获取和设置原始类型值的属性会隐式地创建封装对象。

第 5 条：避免对混合类型使用 == 运算符

你认为下面表达式的值是什么？

```
"1.0e0" == { valueOf: function() { return true; } };
```

对这两个看似无关的值使用 == 运算符实际上是相等的。就像第 3 条描述的隐式强制转换一样，在比较之前，它们都被转换为数字。字符串“1.0e0”被解析为数字 1，而匿名对象也通过调用其自身的 valueOf 方法得到结果 true，然后再转换为数字，得到 1。

很容易使用这些强制转换完成一些工作。例如，从一个 Web 表单读取一个字段并与一个数字进行比较。

```
var today = new Date();

if (form.month.value == (today.getMonth() + 1) &&
    form.day.value == today.getDate()) {
    // happy birthday!
    // ...
}
```

但实际上，它只是显式地使用 Number 函数或者一元运算符 + 将值转换为数字。

```
var today = new Date();

if (+form.month.value == (today.getMonth() + 1) &&
    +form.day.value == today.getDate()) {
    // happy birthday!
    // ...
}
```

上面这段代码更加清晰，因为它向读者传达了代码到底在做什么样的转换，而不要求读者记住这些转换规则。一个更好的替代方法是使用严格相等运算符。

```
var today = new Date();

if (+form.month.value === (today.getMonth() + 1) && // strict
    +form.day.value === today.getDate()) {          // strict
    // happy birthday!
    // ...
}
```

当两个参数属于同一类型时，== 和 === 运算符的行为是没有区别的。因此，如果你知道参数属于同一类型，那么，== 和 === 运算符可以互换。但最好使用严格相等运算符，因为读者会非常清晰地知道：在比较操作中并没有涉及任何转换。否则，你需要读者准确地记住这些强制转换规则以解读代码的行为。

事实上，这些强制转换规则一点也不明显。表 1.1 包含了 == 运算符针对不同类型参数的强制转换规则。这些规则具有对称性。例如，第一条规则既适用于 null == undefined，也适用于 undefined == null。在很多时候，这些转换都试图产生数字。但当它们处理对象时会变得难以捉摸。操作符试图将对象转换为原始值，可通过调用对象的 valueOf 和 toString 方法而实现。更令人难以捉摸的是，Date 对象以相反的顺序尝试调用这两个方法。

表 1.1 == 运算符的强制转换规则

参数类型 1	参数类型 2	强制转换
null	undefined	不转换，总是返回 true
null 或 undefined	其他任何非 null 或 undefined 的类型	不转换，总是返回 false
原始类型：string、number 或 boolean	Date 对象	将原始类型转换为数字；将 Date 对象转换为原始类型（优先尝试 toString 方法，再尝试 valueOf 方法）
原始类型：string、number 或 boolean	非 Date 对象	将原始类型转换为数字；将非 Date 对象转换为原始类型（优先尝试 valueOf 方法，再尝试 toString 方法）
原始类型：string、number 或 boolean	原始类型：string、number 或 boolean	将原始类型转换为数字

== 运算符将数据以不同的表现呈现出来，这种纠错有时称为“照我的意思去做”（do what I mean）的语义。但计算机并不能真正地了解你的心思。世界上有太多的数据表现形式，JavaScript 需要知道你使用的是哪种。例如，你可能希望你能将一个包含日期的字符串和一个 Date 对象进行比较。

```
var date = new Date("1999/12/31");
date == "1999/12/31"; // false
```

这个例子失败是因为 Date 对象被转换成一种不同格式的字符串，而不是本例所采用的格式。

```
date.toString(); // "Fri Dec 31 1999 00:00:00 GMT-0800 (PST)"
```

但是，这种错误是一个更普遍的强制转换误解的“症状”。== 运算符并不能推断和统一所有的数据格式。它需要你和读者都能理解其微妙的强制转换规则。更好的策略是显式自定义应用程序转换的逻辑，并使用严格相等运算符。

```
function toYMD(date) {
    var y = date.getYear() + 1900, // year is 1900-indexed
        m = date.getMonth() + 1,   // month is 0-indexed
        d = date.getDate();
    return y
         + "/" + (m < 10 ? "0" + m : m)
         + "/" + (d < 10 ? "0" + d : d);
}
toYMD(date) === "1999/12/31"; // true
```

显式地定义转换的逻辑能确保你不会混淆 == 运算符的强制转换规则，而且免除了读者不得不查找或记住这些规则的麻烦。

提示

- 当参数类型不同时，== 运算符应用了一套难以理解的隐式强制转换规则。
- 使用 === 运算符，使读者不需要涉及任何的隐式强制转换就能明白你的比较运算。
- 当比较不同类型的值时，使用你自己的显式强制转换使程序的行为更清晰。

第 6 条：了解分号插入的局限

JavaScript 的一个便利是能够离开语句结束分号工作。删除分号后，结果变得轻量而优雅。

```
function Point(x, y) {
    this.x = x || 0
    this.y = y || 0
}

Point.prototype.isOrigin = function() {
    return this.x === 0 && this.y === 0
}
```

上面的代码能工作多亏 JavaScript 的自动分号插入（automatic semicolon insertion）技术，它是一种程序解析技术。它能推断出某些上下文中省略的分号，然后有效地自动地将分号“插入”到程序中。ECMAScript 标准细心地制定了分号插入机制，因此，可选分号可以在不同的 JavaScript 引擎之间移植。

但是同第 3 条和第 5 条的隐式强制转换一样，分号插入也有其陷阱，你根本不能避免学习其规则。即使你从来不省略分号，受分号插入的影响，JavaScript 语法也有一些额外的限制。好消息是，一旦你学会分号插入的规则，你会发现你能从删除不必要的分号的痛苦中解脱出来。

分号插入的第一条规则：

分号仅在 } 标记之前、一个或多个换行之后和程序输入的结尾被插入。

换句话说，你只能在一行、一个代码块和一段程序结束的地方省略分号。因此，下面的函数定义是合法的。

```
function square(x) {
    var n = +x
    return n * n
}
function area(r) { r = +r; return Math.PI * r * r }
function add1(x) { return x + 1 }
```

但是，下面这个却不合法。

```
function area(r) { r = +r return Math.PI * r * r } // error
```

分号插入的第二条规则：

分号仅在随后的输入标记不能解析时插入。

换句话说，分号插入是一种错误校正机制。下面这段代码作为一个简单的例子。

```
a = b
(f());
```

能正确地解析为一条单独的语句，等价于：

```
 a = b(f());
```

也就是说，没有分号插入。与此相反，下面这段代码：

```
a = b
f();
```

被解析为两条独立的语句，因为

```
a = b f();
```

解析有误。

这条规则有一个不幸的影响：你总是要注意下一条语句的开始，从而发现你是否能合法地省略分号。如果某条语句的下一行的初始标记不能被解析为一个语句的延续，那么，你不能省略该条语句的分号。

有 5 个明确有问题的字符需要密切注意：(、[、+、−、和 /。每一个字符都能作为一个表

达式运算符或者一条语句的前缀，这依赖于具体上下文。因此，要小心提防那些以表达式结束的语句，就像上面的赋值语句一样。如果下一行以这 5 个有问题的字符之一开始，那么不会插入分号。到目前为止，最常见的情况是以一个括号开始，就像上面的例子。另一种常见的情况是数组字面量。

```
a = b
["r", "g", "b"].forEach(function(key) {
    background[key] = foreground[key] / 2;
});
```

这看起来像两条语句。一条赋值语句，紧接着一条按序对字符“r”、“g”和“b”调用函数的语句。但是由于该语句以“[”开始，它被解析为一条语句，等价于：

```
a = b["r", "g", "b"].forEach(function(key) {
    background[key] = foreground[key] / 2;
});
```

这个中括号表达式看起来有点怪，请记住 JavaScript 允许逗号分隔表达式。逗号分隔表达式从左至右依次执行，并返回最后一个表达式的值。对于该例子，它返回字符“b”。

+、- 和 / 字符出现在语句开始并不常见，但也不是闻所未闻。字符“/”有一种非常微妙的情况：它出现在语句的开始实际上不是一个入口标记，而是作为正则表达式标记的开始。

```
/Error/i.test(str) && fail();
```

该语句使用一个不区分大小写的正则表达式 /Error/i 来匹配字符串。如果找到一个匹配，就会调用 fail 函数。但是，如果这段出现在一个未终止的赋值语句之后，例如：

```
a = b
/Error/i.test(str) && fail();
```

那么，这段代码会被解析为一条语句，等价于：

```
a = b / Error / i.test(str) && fail();
```

换句话说，初始的 / 字符被解析为除法运算符！

想省略分号时，有经验的 JavaScript 程序员会在该语句的后面跟一个声明，以保证该语句不会被错误地解析。在重构代码时，他们也会非常小心。例如，一个完全正确的程序，省略了 3 个可推断的分号 。

```
a = b     // semicolon inferred
var x     // semicolon inferred
(f())     // semicolon inferred
```

有可能被出人意料地改成只有两个可推断分号的程序。

```
var x     // semicolon inferred
a = b     // no semicolon inferred
(f())     // semicolon inferred
```

即使把 var 语句提前，这两段程序也应该是等价的（变量作用域的详细信息，请参见第 12 条），但事实是，b 后面跟着一个括号，程序被错误地解析为：

```
var x;
a = b(f());
```

其结果是你总需要注意省略分号，并且检查接下来的一行开始的标记是否会禁用自动插入分号。或者，你也可以采用在 (、[、+、- 和 / 字符的开始前置一个额外的分号语句的方法。例如，前面的例子可以改为下面的代码以保护括号中的函数调用。

```
a = b    // semicolon inferred
var x    // semicolon on next line
;(f())   // semicolon inferred
```

现在，把 var 声明语句移至行首是安全的，而不用担心改变这段程序。

```
var x    // semicolon inferred
a = b    // semicolon on next line
;(f())   // semicolon inferred
```

另一个常见的情况是，省略分号可能导致脚本连接问题（参见第 1 条）。每个文件可能由大量的函数调用表达式组成（参见第 13 条更多关于立即调用的函数表达式的信息）。

```
// file1.js
(function() {
    // ...
})()

// file2.js
(function() {
    // ...
})()
```

当每个文件作为一个单独的程序加载时，分号能自动地插入到末尾，将函数调用转变为一条语句。但是，当这些文件以下面的方式进行连接时：

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

结果被视为一条单一的语句，等价于：

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

结果是：省略语句的分号不仅需要当心当前文件的下一个标记，而且还需要当心脚本连接后可能出现在语句之后的任一标记。类似上述方法，你可以防御性地为每个文件前缀一个额外的分号以保护脚本免受粗心连接的影响。如果文件最开始的语句以这 5 个脆弱的字符 (、[、+、− 和 / 开头，你就应该这么做。

```
// file1.js
;(function() {
    // ...
})()

// file2.js
;(function() {
    // ...
})()
```

这会确保即使前一个文件忽略了最后的分号，合并后的结果仍然会被视为单独的语句。

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

当然，如果脚本连接程序能够自动地在文件之间增加额外的分号是更好的。但并不是所有的脚本连接工具都写得很好，因此，最安全的选择是防御性地增加分号。

此时，你可能会认为，“这是多余的担心。我从来就不省略分号，我会没事儿的。”事实并不是这样。也有一些情况，尽管不会出现解析错误，JavaScript 仍会强制地插入分号。这就是所谓的 JavaScript 语法限制产生式（restricted production），它不允许在两个字符之间出现换行。最危险的情况是 return 语句，在 return 关键字和其可选参数之间一定不能包含换行符。因此，语句

```
return { };
```

返回一个新对象，而下面这段代码

```
return
{ };
```

被解析为 3 条单独的语句，等价于：

```
return;
{ }
;
```

换句话说，return 关键字后的换行会强制自动地插入分号。该段代码被解析为不带参数的 return 语句，后接一个空的代码块和一条空语句。其他的限制产生式包括：

❑ throw 语句

❑ 带有显式标签的 break 或 continue 语句

❑ 后置自增或自减运算符

最后一条规则是为了消除如下代码的歧义：

```
a
++
b
```

因为自增运算符既可以作为前置运算符也可以作为后置运算符，但是，后者不能出现在换行之前。这段代码被解析为：

```
a; ++b;
```

第三条也是最后一条分号插入规则：

分号不会作为分隔符在 for 循环空语句的头部被自动插入。

这就意味着你必须在 for 循环头部显式地包含分号。否则，类似下面的代码

```
for (var i = 0, total = 1 // parse error
     i < n
     i++) {
    total *= i
}
```

将会导致解析错误。空循环体的 while 循环同样也需要显式的分号。否则，省略分号也会导致解析错误：

```
function infiniteLoop() { while (true) } // parse error
```

因此，这就是一种需要分号的情况。

```
function infiniteLoop() { while (true); }
```

提示

❑ 仅在“}”标记之前、一行的结束和程序的结束处推导分号。

❑ 仅在紧接着的标记不能被解析的时候推导分号。

❑ 在以 (、[、+、- 或 / 字符开头的语句前绝不能省略分号。

❑ 当脚本连接的时候，在脚本之间显式地插入分号。

❑ 在 return、throw、break、continue、++ 或 -- 的参数之前绝不能换行。

❑ 分号不能作为 for 循环的头部或空语句的分隔符而被推导出。

第 7 条：视字符串为 16 位的代码单元序列

Unicode 有一个声誉，就是其复杂性。尽管字符串无处不在，大多数程序员还是抱着乐

观的态度避免学习 Unicode。但是在概念层面，它没有什么可害怕的。Unicode 的基础非常简单。它为世界上所有的文字系统的每个字符单位分配了一个唯一的整数，该整数介于 0 和 1 114 111 之间，在 Unicode 术语中称为代码点（code point）。Unicode 与其他字符编码几乎没有任何不同（例如，ASCII）。然而不同的是，ASCII 将每个索引映射为唯一的二进制表示，但 Unicode 允许多个不同二进制编码的代码点。不同的编码在要求存储的字符串数量和操作速度（如索引到某个字符串）之间进行权衡。目前有多种 Unicode 的编码标准，最流行的几个是：UTF-8、UTF-16 和 UTF-32。

进一步使情况复杂的是，Unicode 的设计师根据历史的数据，错误估算了代码点的容量范围。人们起初认为 Unicode 最多只需要 2^{16} 个代码点，所以产生了 UCS-2，其为 16 位编码的原始标准。这是一个特别有吸引力的选择。由于每个代码点可以容纳一个 16 位的数字，所以简单的方法就是将代码点与其编码元素一对一地映射起来，这称为一个代码单元（code unit）。也就是说，UCS-2 是由独立的 16 位的代码单元组成的，每个代码单元对应一个单独的 Unicode 代码点。这种编码方式的主要好处在于索引字符串是一种代价小的、固定时间的操作。获取某个字符串的第 n 个代码点只是简单地选取数组的第 n 个 16 位元素。图 1.1 显示了一个字符串例子。这些字符仅由最初的 16 位范围中的代码点组成。正如你看到的一样，对于 Unicode 的字符串，编码元素和代码点能完全的匹配。

其结果是，当时许多平台都采用 16 位编码的字符串。Java 便是其中之一，JavaScript 也紧随其后，所以 JavaScript 字符串的每个元素都是一个 16 位的值。现在，如果 Unicode 还是保持 20 世纪 90 年代初的做法，那么 JavaScript 字符串的每个元素仍然对应一个单独的代码点。

16 位的范围是相当大的，囊括了世界上的大多数文字系统，这比 ASCII 或其无数的历史替代者都要多。即便如此，Unicode 也需要及时扩大其最初的范围，标准从当时的 2^{16} 扩展到了超过 2^{20} 个代码点。新增加的范围被组织为 17 个大小为 2^{16} 代码点的子范围。第一个子范围，称为基本多文种平面（Basic Multilingual Plane, BMP），包含最初的 2^{16} 个代码点。余下的 16 个范围称为辅助平面（supplementary plane）。

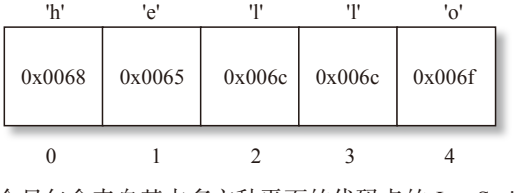

图 1.1 一个只包含来自基本多文种平面的代码点的 JavaScript 字符串

一旦代码点的范围扩展，UCS-2 就变得过时了。它需要通过扩展来表示这些附加的代码点。其替代者 UTF-16 与之类似，但 UTF-16 采用代理对表示附加的代码点。一对 16 位的代

码单元共同编码一个等于或大于 2^{16} 的代码点。例如，分配给高音谱号的音乐符号（“𝄞”）的代码点为 U+1D11E（代码点数 119 070 的 Unicode 的惯用 16 进制写法）。其由 UTF-16 格式的代码单元 0xd834 和 0xddle 共同表示。可以通过合并这两个代码单元选择的位来对这个代码点进行解码。（巧妙的是，这种编码保证了这些代理对绝不会与有效的 BMP 代码点混淆，因此，甚至从字符串中间的某个位置进行搜索，你也可以随时识别一个代理对。）在图 1.2 中你可以看到一个含有代理对的字符串的例子。该字符串的第一个代码点需要一个代理对，从而导致了代码单元的索引与代码点的索引不同。

由于 UTF-16 的每个代码点编码需要一个或两个 16 位的代码单元，因此 UTF-16 是一种可变长度的编码。长度为 n 的字符串在内存中的大小变化基于该字符串特定的代码点。此外，查找字符串的第 n 个代码点不再是一个固定时间的操作，因为它一般需要从字符串的开始处进行搜索。

但是当 Unicode 扩大规模时，JavaScript 已经采用了 16 位的字符串元素。字符串属性和方法（如 length、charAt 和 charCodeAt）都是基于代码单元层级，而不是代码点层级。所以每当字符串包含辅助平面中的代码点时，JavaScript 将每个代码点表示为两个元素而不是一个（一对 UTF-16 代理对的代码点）。简单地说，一个 JavaScript 字符串的元素是一个 16 位的代码单元。

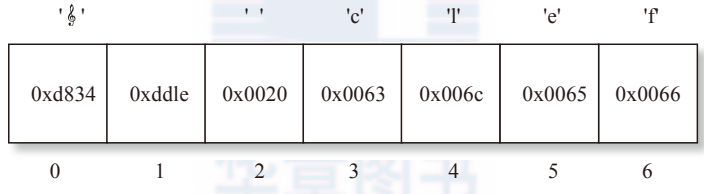

图 1.2　一个包含来自辅助平面的代码点的 JavaScript 字符串

JavaScript 引擎可以在内部优化字符串内容的存储。但是考虑到字符串的属性和方法，字符串表现得就像 UTF-16 的代码单元序列。正如图 1.2 中的字符串，尽管事实上只包含 6 个代码点，但是 JavaScript 报告它的长度为 7。

```
"𝄞 clef".length; // 7
"G clef".length; // 6
```

提取该字符串的某个字符得到的是代码单元，而不是代码点。

```
"𝄞 clef".charCodeAt(0);     // 55348 (0xd834)
"𝄞 clef".charCodeAt(1);     // 56606 (0xddle)
"𝄞 clef".charAt(1) === " "; // false
"𝄞 clef".charAt(2) === " "; // true
```

类似地，正则表达式也工作于代码单元层级。其单字符模式（“.”）匹配一个单一的代

码单元。

```
/^.$/.test("𝄞");  // false
/^..$/.test("𝄞"); // true
```

这种状况意味着应用程序同 Unicode 的整个范围一起工作必须更加仔细。应用程序不能信赖字符串方法、长度值、索引查找或者许多正则表达式模式。如果你使用除 BMP 之外的代码点，那么求助于一些支持代码点的库是个好主意。正确地获取编码和解码的细节是相当棘手的，所以最好使用一个现存的库，而不是自己实现这些逻辑。

虽然 JavaScript 内置的字符串数据类型工作于代码单元层级，但这并不能阻止一些 API 意识到代码点和代理对。事实上，一些标准的 ECMAScript 库正确地处理了代理对，例如 URI 操作函数：sendcodeURI、decodeURI、encodeURIComponent 和 decodeURIComponent。每当一个 JavaScript 环境提供一个库操作字符串（例如，操作一个 Web 页面的内容或者执行关于字符串的 I/O 操作），你都需要查阅这些库文档，看它们如何处理 Unicode 代码点的整个范围。

提示

- JavaScript 字符串由 16 位的代码单元组成，而不是由 Unicode 代码点组成。
- JavaScript 使用两个代码单元表示 2^{16} 及其以上的 Unicode 代码点。这两个代码单元被称为代理对。
- 代理对甩开了字符串元素计数，length、charAt、charCodeAt 方法以及正则表达式模式（例如“.”）受到了影响。
- 使用第三方的库编写可识别代码点的字符串操作。
- 每当你使用一个含有字符串操作的库时，你都需要查阅该库文档，看它如何处理代码点的整个范围。

第 2 章

变量作用域

作用域对于程序员来说就像氧气。它无处不在，甚至，你往往不会去想它。但当它被污染时，你会感觉到窒息。

好消息是 JavaScript 核心的作用域规则很简单。其作用域规则被精心设计，且强大得令人难以置信。但也有一些例外情况。有效地使用 JavaScript 需要掌握变量作用域的一些基本概念，并了解一些可能导致难以捉摸的、令人讨厌的问题的极端情况。

第 8 条：尽量少用全局对象

在 JavaScript 中很容易在全局命名空间中创建变量。创建全局变量毫不费力，因为它不需要任何形式的声明，而且能被整个程序的所有代码自动地访问。这种便利很容易诱惑初学者。然而经验丰富的程序员都知道应该避免使用全局变量。定义全局变量会污染共享的公共命名空间，并可能导致意外的命名冲突。全局变量不利于模块化，因为它会导致程序中独立组件间的不必要耦合。虽然“先这样写，以后再调整”（“code now and organize later”）可能比较方便，但优秀的程序员会不断地留意程序的结构、持续地归类相关的功能以及分离不相关的组件，并将这些行为作为编程过程中的一部分。

由于全局命名空间是 JavaScript 程序中独立的组件进行交互的唯一途径，因此，利用全局命名空间的情况是不可避免的。组件或程序库不得不定义一些全局变量，以便程序中的其他部分使用。否则，最好尽量使用局部变量。当然可以写一个只使用全局变量而不使用其他变量的程序，但那是自寻烦恼。即使在很简单的函数中将临时变量定义为全局的，我们都会担心是否有任何其他的代码可能会使用相同的变量名。

```
var i, n, sum; // globals
function averageScore(players) {
```

```
    sum = 0;
    for (i = 0, n = players.length; i < n; i++) {
        sum += score(players[i]);
    }
    return sum / n;
}
```

如果 score 函数出于自身的目的使用了任何同名的全局变量，averageScore 函数的定义将出现问题。

```
var i, n, sum; // same globals as averageScore!
function score(player) {
    sum = 0;
    for (i = 0, n = player.levels.length; i < n; i++) {
        sum += player.levels[i].score;
    }
    return sum;
}
```

答案是保持这些变量为局部变量，仅将其作为需要使用它们的代码的一部分。

```
function averageScore(players) {
    var i, n, sum;
    sum = 0;
    for (i = 0, n = players.length; i < n; i++) {
        sum += score(players[i]);
    }
    return sum / n;
}

function score(player) {
    var i, n, sum;
    sum = 0;
    for (i = 0, n = player.levels.length; i < n; i++) {
        sum += player.levels[i].score;
    }
    return sum;
}
```

JavaScript 的全局命名空间也被暴露为在程序全局作用域中可以访问的全局对象，该对象作为 this 关键字的初始值。在 Web 浏览器中，全局对象被绑定到全局的 window 变量。添加或修改全局变量会自动更新全局对象。

```
this.foo; // undefined
foo = "global foo";
this.foo; // "global foo"
```

类似地，更新全局对象也会自动地更新全局命名空间：

```
var foo = "global foo";
this.foo = "changed";
foo; // "changed"
```

这意味着你创建一个全局变量有两种方法可供挑选。你可以在全局作用域内使用 var 声明它，或者将其加入到全局对象中。无论使用哪种方法都行，但是 var 声明的好处是更能清晰地表达全局变量在程序范围中的影响。鉴于引用未绑定的变量会导致运行时错误，因此，保持作用域清晰和简洁会使代码的使用者更容易理解程序声明了哪些全局变量。

虽然最好限制使用全局对象，但是它确实提供了一个不可或缺的特别用途。由于全局对象提供了全局环境的动态反应机制，所以可以使用它查询一个运行环境，检测在这个平台下哪些特性是可用的。例如，ES5 引入了一个全局的 JSON 对象来读写 JSON 格式的数据。将代码部署到一个不确定是否提供了 JSON 对象的环境时的一个权宜之计是，你可以测试这个全局对象是否存在并提供一个替代实现。

```
if (!this.JSON) {
    this.JSON = {
        parse: ...,
        stringify: ...
    };
}
```

如果你已经提供了 JSON 的实现，你当然可以简单无条件地使用自己的实现。但是由宿主环境提供的内置实现几乎总是更合适的。因为它们按照一定的标准对正确性和一致性进行了严格检查，并且普遍来说比第三方实现提供了更好的性能。

特性检测技术在 Web 浏览器中特别重要，因为在各种各样的浏览器和浏览器版本中可能会执行同样的代码。特性检测是一种使得程序在平台特性集合的变化中依旧健壮的相对简单的方法。这种技术也适用于其他地方。例如，此技术使得在浏览器和 JavaScript 服务器环境中共享程序库成为可能。

提示

- 避免声明全局变量。
- 尽量声明局部变量。
- 避免对全局对象添加属性。
- 使用全局对象来做平台特性检测。

第 9 条：始终声明局部变量

如果存在比全局变量更麻烦的事，那就是意外的全局变量。遗憾的是，JavaScript 的变

量赋值规则使得意外地创建全局变量太容易了。程序中给一个未绑定的变量赋值将会简单地创建一个新的全局变量并赋值给它，而不是引发错误。这意味着，如果忘记将变量声明为局部变量，那么该变量将会被隐式地转变为全局变量。

```
function swap(a, i, j) {
    temp = a[i]; // global
    a[i] = a[j];
    a[j] = temp;
}
```

尽管该程序没有使用 var 声明 temp 变量，但是执行是不会出错的，只是会导致意外地创建一个全局变量。正确的实现应该使用 var 声明 temp 变量。

```
function swap(a, i, j) {
    var temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

故意地创建全局变量是不好的风格，而意外地创建全局变量将是彻头彻尾的灾难。正因如此，许多程序员使用 lint 工具检查程序源代码中的不好风格和潜在的错误。该工具通常具有报告未绑定变量使用情况的功能。通常情况下，lint 工具使用用户提供的一套已知的全局变量（例如，期望存在于宿主环境中的或在单独文件中定义的全局变量）检查未声明的变量，然后报告出所有既没有在列表中提供的又没有在程序中声明的引用或赋值变量。花一些时间去探索什么样的工具对 JavaScript 可用是值得的。将自动检查一些常见的错误（例如，意外的全局变量）整合到开发过程中可能会成为救命稻草。

提示

- 始终使用 var 声明新的局部变量。
- 考虑使用 lint 工具帮助检查未绑定的变量。

第 10 条：避免使用 with

悲催的 with 特性。在 JavaScript 中可能没有比它更令人诟病的特性了。然而，with 语句是罪有应得。它提供的任何“便利”，都更让其变得不可靠和低效率。

with 语句的动机是可以理解的。程序经常需要对单个对象依次调用一系列方法。使用 with 语句可以很方便地避免对对象的重复引用：

```
function status(info) {
    var widget = new Widget();
    with (widget) {
```

```
            setBackground("blue");
            setForeground("white");
            setText("Status: " + info); // ambiguous reference
            show();
        }
    }
```

使用 with 语句从模块对象中“导入”（import）变量也是很有诱惑力的。

```
function f(x, y) {
    with (Math) {
        return min(round(x), sqrt(y)); // ambiguous references
    }
}
```

在这两种情况下，使用 with 语句使得提取对象的属性，并将这些属性绑定到块的局部变量中变得非常诱人且容易。

这些例子看起来很有吸引力，但它实际没做它应该做的事。请注意这两个例子有两种不同类型的变量。一种是我们希望引用 with 对象的属性的变量，如 setBackground、round 以及 sqrt。另一种是我们希望引用外部变量绑定的变量，如 info、x 和 y。但其实在语法上并没有区分这两种类型的变量。它们都只是看起来像变量。

事实上，JavaScript 对待所有的变量都是相同的。JavaScript 从最内层的作用域开始向外查找变量。with 语句对待一个对象犹如该对象代表一个变量作用域，因此，在 with 代码块的内部，变量查找从搜索给定的变量名的属性开始。如果在这个对象中没有找到该属性，则继续在外部作用域中搜索。

图 2.1 显示了当执行 with 语句的代码时 ,status 函数的作用域在 JavaScript 引擎中的内部表示图。在 ES5 规范中这称为词法环境 (在旧版本标准中称为作用域链)。该词法环境的最内层作用域由 widget 对象提供。接下来的作用域用来绑定该函数的局部变量 info 和 widget。接下来一层绑定到 status 函数。注意在一个正常的作用域中，会有与局部作用域中的变量同样多的作用域绑定存储在与之对应的环境层级中。但是对于 with 作用域，绑定集合依赖于碰巧在给定时间点时的对象。

我们有多大的信心确信在提供给 with 的对象中可以找到哪些属性，或者找不到哪些属性？ with 块中的每个外部变量的引用都隐式地假设在 with 对象（以及它的任何原型对象）中没有同名的属性。而在程序的其他地方创建或修改 with 对象或其原型对象不一定会遵循这样的假设。JavaScript 引擎当然不会读取局部代码来获取你使用了哪些局部变量。

变量作用域和对象命名空间之间的冲突使得 with 代码块异常脆弱。例如，如果上述例子的 with 对象获得了一个名为 info 的属性，status 函数的行为将被立即改变。status 函数将使用这个属性而不是 status 函数的 info 参数。这种情况可能发生在源代码的演化中。例如，程

序员决定所有的 widget 对象都应该有一个 info 属性。更糟糕的是，有时会给 Widget 的原型对象在运行时加入 info 属性，这将导致 status 函数变得不可预测。

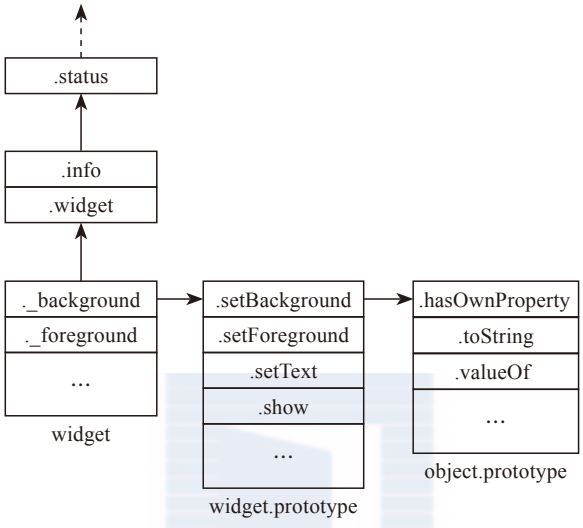

图 2.1 status 函数的词法环境（又称作用域链）

```
status("connecting"); // Status: connecting
Widget.prototype.info = "[[widget info]]";
status("connected");  // Status: [[widget info]]
```

同样，如果某人添加名为 x 或 y 的属性到 Math 对象上，那么上述例子中的 f 函数也会出错。

```
Math.x = 0;
Math.y = 0;
f(2, 9); // 0
```

可能不会有人给 Math 添加 x 和 y 属性。但总是很难预测一个特定的对象是否已被修改，或是否可能拥有你不知道的属性。而事实证明，人力不可预测的特性对于优化编译器同样不可预测。通常情况下，JavaScript 作用域可被表示为高效的内部数据结构，变量查找会非常快速。但是由于 with 代码块需要搜索对象的原型链来查找 with 代码块里的所有变量，因此，其运行速度远远低于一般的代码块。

在 JavaScript 中没有单个特性能作为一个更好的选择直接替代 with 语句。在某些情况下，最好的替代方法是简单地将对象绑定到一个简短的变量名上。

```
function status(info) {
    var w = new Widget();
    w.setBackground("blue");
    w.setForeground("white");
```

```
    w.addText("Status: " + info);
    w.show();
}
```

该版本的行为更具可预测性。没有任何变量引用对于 w 对象的内容是敏感的。所以即使一些代码修改了 Widget 的原型对象，status 函数的行为依旧与预期一致。

```
status("connecting"); // Status: connecting
Widget.prototype.info = "[[widget info]]";
status("connected");  // Status: connected
```

在其他情况下，最好的方法是将局部变量显式地绑定到相关的属性上。

```
function f(x, y) {
    var min = Math.min, round = Math.round, sqrt = Math.sqrt;
    return min(round(x), sqrt(y));
}
```

再次，一旦消除 with 语句，函数的行为变得可以预测。

```
Math.x = 0;
Math.y = 0;
f(2, 9); // 2
```

提示

- 避免使用 with 语句。
- 使用简短的变量名代替重复访问的对象。
- 显式地绑定局部变量到对象属性上，而不要使用 with 语句隐式地绑定它们。

第 11 条：熟练掌握闭包

对于那些使用不支持闭包特性的编程语言的程序员来说，闭包可能是一个陌生的概念。初看起来，它们似乎令人生畏。但请放心，付出努力掌握闭包将会给你带来超值的回报。

幸运的是，闭包真没有什么可害怕的。理解闭包只需要学会三个基本的事实。第一个事实：JavaScript 允许你引用在当前函数以外定义的变量。

```
function makeSandwich() {
    var magicIngredient = "peanut butter";
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make("jelly");
}
makeSandwich(); // "peanut butter and jelly"
```

请注意内部的 make 函数是如何引用定义在外部 makeSandwich 函数内的 magicIngredient 变量的。

第二个事实：即使外部函数已经返回，当前函数仍然可以引用在外部函数所定义的变量。如果这听起来让人难以置信，请记住，JavaScript 的函数是第一类（first-class）对象（请参阅第 19 条）。这意味着，你可以返回一个内部函数，并在稍后调用它。

```
function sandwichMaker() {
    var magicIngredient = "peanut butter";
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make;
}
var f = sandwichMaker();
f("jelly");        // "peanut butter and jelly"

f("bananas");      // "peanut butter and bananas"
f("marshmallows"); // "peanut butter and marshmallows"
```

这与第一个例子几乎完全相同。不同的是，不是在外部的 sandwichMaker 函数中立即调用 make（"jelly"），而是返回 make 函数本身。因此，f 的值为内部的 make 函数，调用 f 实际上是调用 make 函数。但即使 sandwichMaker 函数已经返回，make 函数仍能记住 magicIngredient 的值。

这是如何工作的？答案是：JavaScript 的函数值包含了比调用它们时执行所需要的代码还要多的信息。而且，JavaScript 函数值还在内部存储它们可能会引用的定义在其封闭作用域的变量。那些在其所涵盖的作用域内跟踪变量的函数被称为闭包。make 函数就是一个闭包，其代码引用了两个外部变量：magicIngredient 和 filling。每当 make 函数被调用时，其代码都能引用到这两个变量，因为该闭包存储了这两个变量。

函数可以引用在其作用域内的任何变量，包括参数和外部函数变量。我们可以利用这点来编写更加通用的 sandwichMaker 函数。

```
function sandwichMaker(magicIngredient) {
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make;
}
var hamAnd = sandwichMaker("ham");
hamAnd("cheese");        // "ham and cheese"
hamAnd("mustard");       // "ham and mustard"
var turkeyAnd = sandwichMaker("turkey");
```

```
turkeyAnd("Swiss");       // "turkey and Swiss"
turkeyAnd("Provolone");   // "turkey and Provolone"
```

该例子创建了 hamAnd 和 turkeyAnd 两个完全不同的函数。尽管它们都是由相同的 make 函数定义的，但是它们是两个截然不同的对象。第一个函数的 magicIngredient 的值为 "ham"，而第二个函数的 magicIngredient 的值为 "turkey"。

闭包是 JavaScript 最优雅、最有表现力的特性之一，也是许多惯用法的核心。JavaScript 甚至还提供了一种更为方便地构建闭包的字面量语法——函数表达式。

```
function sandwichMaker(magicIngredient) {
    return function(filling) {
        return magicIngredient + " and " + filling;
    };
}
```

请注意，该函数表达式是匿名的。由于我们只需要其能产生一个新的函数值，而不打算在局部调用它，因此根本没有必要给该函数命名。函数表达式也可以有名称（请参阅第 14 条）。

学习闭包的第三个也是最后一个事实：闭包可以更新外部变量的值。实际上，闭包存储的是外部变量的引用，而不是它们的值的副本。因此，对于任何具有访问这些外部变量的闭包，都可以进行更新。一个简单的惯用法 box 对象说明了这一切。它存储了一个可读写的内部值。

```
function box() {
    var val = undefined;
    return {
        set: function(newVal) { val = newVal; },
        get: function() { return val; },
        type: function() { return typeof val; }
    };
}
var b = box();
b.type(); // "undefined"
b.set(98.6);
b.get();  // 98.6
b.type(); // "number"
```

该例子产生了一个包含三个闭包的对象。这三个闭包是 set、get 和 type 属性。它们都共享访问 val 变量。set 闭包更新 val 的值，随后调用 get 和 type 查看更新的结果。

提示

- 函数可以引用定义在其外部作用域的变量。
- 闭包比创建它们的函数有更长的生命周期。

❑ 闭包在内部存储其外部变量的引用，并能读写这些变量。

第 12 条：理解变量声明提升

JavaScript 支持词法作用域（lexical scoping），即除了极少的例外，对变量 foo 的引用会被绑定到声明 foo 变量最近的作用域中。但是，JavaScript 不支持块级作用域，即变量定义的作用域并不是离其最近的封闭语句或代码块，而是包含它们的函数。

不明白 JavaScript 的这一特性将会导致一些微妙的 Bug，例如：

```
function isWinner(player, others) {
    var highest = 0;
    for (var i = 0, n = others.length; i < n; i++) {
        var player = others[i];
        if (player.score > highest) {
            highest = player.score;
        }
    }
    return player.score > highest;
}
```

该程序在 for 循环体内声明了一个局部变量 player。但是由于 JavaScript 中变量是函数级作用域 (function-scoped)，而不是块级作用域，所以在内部声明的 player 变量只是简单地重声明了一个已经存在于作用域内的变量（即参数 player）。该循环的每次迭代都会重写同一变量。因此，return 语句将 player 看作 others 的最后一个元素，而不是此函数最初的 player 参数。

理解 JavaScript 变量声明行为的一个好办法是把变量声明看作由两部分组成，即声明和赋值。JavaScript 隐式地提升 (hoists) 声明部分到封闭函数的顶部，而将赋值留在原地。换句话说，变量的作用域是整个函数，但仅在 var 语句出现的位置进行赋值。图 2.2 提供了变量声明提升的可视化图。

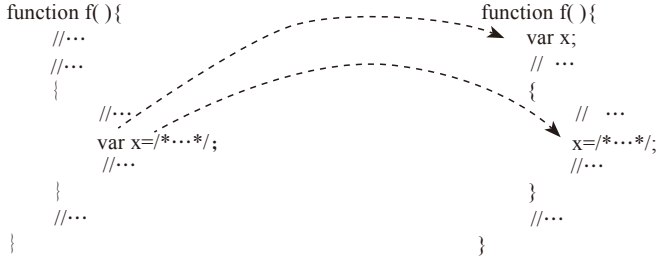

图 2.2 变量声明提升

变量声明提升也可能导致变量重声明的混淆。在同一函数中多次声明相同变量是合法

的。这在写多个循环时会经常出现。

```
function trimSections(header, body, footer) {
    for (var i = 0, n = header.length; i < n; i++) {
        header[i] = header[i].trim();
    }

    for (var i = 0, n = body.length; i < n; i++) {
        body[i] = body[i].trim();
    }
    for (var i = 0, n = footer.length; i < n; i++) {
        footer[i] = footer[i].trim();
    }
}
```

trimSections 函数好像声明了 6 个局部变量（3 个变量 i,3 个变量 n），但经过变量声明提升后其实只声明了 2 个。换句话说，经过变量声明提升后，trimSections 函数等同于下面这个重写的版本。

```
    for (var i = 0, n = body.length; i < n; i++) {
        body[i] = body[i].trim();
    }
    for (var i = 0, n = footer.length; i < n; i++) {
        footer[i] = footer[i].trim();
    }
}
```

因为重声明会导致截然不同的变量展现，一些程序员喜欢通过有效地手动提升变量将所有的 var 声明放置在函数的顶部，从而避免歧义。无论你是否喜欢这种风格，重要的是，不管是写代码还是读代码，都要理解 JavaScript 的作用域规则 。

JavaScript 没有块级作用域的一个例外恰好是其异常处理。try...catch 语句将捕获的异常绑定到一个变量，该变量的作用域只是 catch 语句块。

```
function test() {
    var x = "var", result = [];
    result.push(x);
    try {
        throw "exception";
    } catch (x) {
        x = "catch";
    }
    result.push(x);
    return result;
}
test(); // ["var", "var"]
```

提示

- 在代码块中的变量声明会被隐式地提升到封闭函数的顶部。
- 重声明变量被视为单个变量。
- 考虑手动提升局部变量的声明，从而避免混淆。

第 13 条：使用立即调用的函数表达式创建局部作用域

这段程序（Bug 程序）输出什么？

```
function wrapElements(a) {
    var result = [], i, n;
    for (i = 0, n = a.length; i < n; i++) {
        result[i] = function() { return a[i]; };
    }
    return result;
}

var wrapped = wrapElements([10, 20, 30, 40, 50]);
var f = wrapped[0];
f(); // ?
```

程序员可能希望这段程序输出 10，但实际上它输出 undefined 值。

搞清楚该例子的方法是理解绑定与赋值的区别。在运行时进入一个作用域，JavaScript 会为每一个绑定到该作用域的变量在内存中分配一个“槽”（slot）。wrapElements 函数绑定了三个局部变量：result、i 和 n。因此，当它被调用时，wrapElements 函数会为这三个变量分配“槽”。在循环的每次迭代中，循环体都会为嵌套函数分配一个闭包。该程序的 Bug 在于这样一个事实：程序员似乎期望该函数存储的是嵌套函数创建时变量 i 的值。但事实上，它存储的是变量 i 的引用。由于每次函数创建后变量 i 的值都发生了变化，因此内部函数最终看到的是变量 i 最后的值。值得注意的是，闭包存储的是其外部变量的引用而不是值。

所以，所有由 wrapElements 函数创建的闭包都引用在循环之前创建的变量 i 的同一个共享“槽”。由于每次循环迭代都递增变量 i 直到运行到数组结束，因此，这时候其实当我们调用其中任何一个闭包时，它都会查找数组的索引 5 并返回 undefined 值。

请注意，即使我们把 var 声明置于 for 循环的头部，wrapElements 函数的表现也完全一样。

```
function wrapElements(a) {
    var result = [];
    for (var i = 0, n = a.length; i < n; i++) {
        result[i] = function() { return a[i]; };
```

```
    }
    return result;
}

var wrapped = wrapElements([10, 20, 30, 40, 50]);
var f = wrapped[0];
f(); // undefined
```

这个版本看起来更具欺骗性，因为 var 声明出现在了循环体中。但一如既往，变量声明会被提升到循环的上方。再一次，变量 i 只被分配了一个“槽”。

解决的办法是通过创建一个嵌套函数并立即调用它来强制创建一个局部作用域。

```
function wrapElements(a) {
    var result = [];
    for (var i = 0, n = a.length; i < n; i++) {

      (function() {
          var j = i;
          result[i] = function() { return a[j]; };
      })();
    }
    return result;
}
```

这种技术被称为立即调用的函数表达式，或 IIFE（发音为“iffy”）。它是一种不可或缺的解决 JavaScript 缺少块级作用域的方法。另一种变种是将作为形参的局部变量绑定到 IIFE 并将其值作为实参传入。

```
function wrapElements(a) {
    var result = [];
    for (var i = 0, n = a.length; i < n; i++) {
      (function(j) {
          result[i] = function() { return a[j]; };
      })(i);
    }
    return result;
}
```

然而，使用 IIFE 来创建局部作用域要小心，因为在函数中包裹代码块可能会导致代码块发生一些微妙的变化。首先，代码块不能包含任何跳出块的 break 语句和 continue 语句。因为在函数外使用 break 或 continue 是不合法的。其次，如果代码块引用了 this 或特别的 arguments 变量，IIFE 将会改变它们的含义。第 3 章将讨论与 this 和 arguments 变量一起工作的技术。

提示

- 理解绑定与赋值的区别。
- 闭包通过引用而不是值捕获它们的外部变量。
- 使用立即调用的函数表达式（IIFE）来创建局部作用域。
- 当心在立即调用的函数表达式中包裹代码块可能改变其行为的情形。

第 14 条：当心命名函数表达式笨拙的作用域

JavaScript 函数无论放在何处看起来似乎都是一样的，但是根据上下文其含义会发生变化。请看以下代码片段。

```
function double(x) { return x * 2; }
```

这段代码可以是一个函数声明，也可以是一个命名函数表达式(named function expression)，这取决于它出现的地方。这个声明是如此熟悉，它定义一个函数并绑定到当前作用域的一个变量。例如，在程序的最顶层，以上的声明将创建一个名为 double 的全局函数。但是同一段函数代码也可以作为一个表达式，它可以有截然不同的含义。例如：

```
var f = function double(x) { return x * 2; };
```

根据 ECMAScript 规范，此语句将该函数绑定到变量 f，而不是变量 double。当然，给函数表达式命名并不是必要的。我们可以使用匿名的函数表达式形式：

```
var f = function(x) { return x * 2; };
```

匿名和命名函数表达式的官方区别在于后者会绑定到与其函数名相同的变量上，该变量将作为该函数内的一个局部变量。这可以用来写递归函数表达式。

```
var f = function find(tree, key) {
    if (!tree) {
        return null;
    }
    if (tree.key === key) {
        return tree.value;
    }
    return find(tree.left, key) ||
           find(tree.right, key);
};
```

注意，变量 find 的作用域只在其自身函数中。不像函数声明，命名函数表达式不能通过其内部的函数名在外部被引用。

```
find(myTree, "foo"); // error: find is not defined
```

使用命名函数表达式进行递归似乎没有必要，因为使用外部作用域的函数名也可达到同样的效果：

```
var f = function(tree, key) {
    if (!tree) {
        return null;
    }
    if (tree.key === key) {
        return tree.value;
    }
    return f(tree.left, key) ||
           f(tree.right, key);
};
```

或者我们可以只使用一个声明。

```
function find(tree, key) {
    if (!tree) {
        return null;
    }
    if (tree.key === key) {
        return tree.value;
    }
    return find(tree.left, key) ||
           find(tree.right, key);
}
var f = find;
```

命名函数表达式真正的用处是进行调试。大多数现代的 JavaScript 环境都提供对 Error 对象的栈跟踪功能。在栈跟踪中，函数表达式的名称通常作为其入口使用。用于检查栈的设备调试器对命名函数表达式有类似的使用。

遗憾的是，命名函数表达式是作用域和兼容性问题臭名昭著的来源。这要归结于在 ECMAScript 规范的历史中很不幸的错误以及流行的 JavaScript 引擎中的 Bug。规范的错误在 ES3 中已经存在，JavaScript 引擎被要求将命名函数表达式的作用域表示为一个对象，这有点像有问题的 with 结构。该作用域对象只含有单个属性，该属性将函数名和函数自身绑定起来。该作用域对象也继承了 Object.prototype 的属性。这意味着仅仅是给函数表达式命名也会将 Object.prototype 中的所有属性引入到作用域中。结果可能会出人意料：

```
var constructor = function() { return null; };
var f = function f() {
    return constructor();
};
f(); // {} (in ES3 environments)
```

该程序看起来会产生 null, 但其实会产生一个新的对象。因为命名函数表达式在其作用

域内继承了 Object.prototype.constructor（即 Object 的构造函数）。就像 with 语句一样，这个作用域会因 Object.prototype 的动态改变而受到影响。程序的一部分可能添加或删除 Object.prototype 属性，命名函数表达式中的所有变量都会受到影响。

幸运的是，ES5 修正了这个错误。但是一些 JavaScript 环境仍然使用过时的对象作用域。更糟的是，有些环境甚至更不符合标准，而且甚至对匿名函数表达式使用对象作为作用域。即使删除上述例子中的函数表达式名也会产生一个对象，而不是预期结果 null.

```
var constructor = function() { return null; };
var f = function() {
    return constructor();
};
f(); // {} (in nonconformant environments)
```

在系统中避免对象污染函数表达式作用域的最好方式是避免任何时候在 Object.prototype 中添加属性，以及避免使用任何与标准 Object.protoype 属性同名的局部变量。

在流行的 JavaScript 引擎中的另一个缺陷是对命名函数表达式的声明进行提升。例如：

```
var f = function g() { return 17; };
g(); // 17 (in nonconformant environments)
```

需要明确的是，这是不符合标准的行为。更糟的是，一些 JavaScript 环境甚至把 f 和 g 这两个函数作为不同的对象，从而导致不必要的内存分配。这种行为的一个合理的解决办法是创建一个与函数表达式同名的局部变量并赋值为 null。

```
var f = function g() { return 17; };
var g = null;
```

即使在没有错误地提升函数表达式声明的环境中，使用 var 重声明变量能确保仍然会绑定变量 g。设置变量 g 为 null 能确保重复的函数可以被垃圾回收。

当然可以得出合理的结论：命名函数表达式由于会导致很多问题，所以并不值得使用。一个不太严肃的回应是在开发阶段使用命名函数表达式用作调试，在发布前通过预处理程序将所有的函数表达式转为匿名的。但有一条是肯定的，你应当总是明确发布的平台（请参阅第 1 条）。你可能做的最糟的事情是为了支持那些甚至没有必要支持的平台将代码弄得一团糟。

提示

- 在 Error 对象和调试器中使用命名函数表达式改进栈跟踪。
- 在 ES3 和有问题的 JavaScript 环境中谨记函数表达式作用域会被 Object.prototype 污染。
- 谨记在错误百出的 JavaScript 环境中会提升命名函数表达式声明，并导致命名函数表

达式的重复存储。

- ❑ 考虑避免使用命名函数表达式或在发布前删除函数名。
- ❑ 如果你将代码发布到正确实现的 ES5 环境中，那么你没有什么好担心的。

第 15 条：当心局部块函数声明笨拙的作用域

我们继续讨论关于上下文敏感的传奇故事：嵌套函数声明。当你知道没有标准的方法在局部块里声明函数时，你可能会感到惊讶。然而现在，这是完全合法的，而且人们习惯于在另一个函数的顶部嵌套函数声明。

```
function f() { return "global"; }

function test(x) {
    function f() { return "local"; }

    var result = [];
    if (x) {
        result.push(f());
    }

    result.push(f());
    return result;
}

test(true);  // ["local", "local"]
test(false); // ["local"]
```

然而，如果我们把函数 f 移到局部块里，那么，将产生一个完全不同的情形。

```
function f() { return "global"; }

function test(x) {
    var result = [];
    if (x) {
        function f() { return "local"; } // block-local

        result.push(f());
    }
    result.push(f());
    return result;
}

test(true);  // ?
test(false); // ?
```

由于内部的函数f出现在if语句块中，因此你可能认为第一次调用test产生数组["local", "global"]，第二次调用产生数组["global"]。但是要记住JavaScript没有块级作用域，所以内部函数f的作用域应该是整个test函数。第二个例子的合理猜测是["local", "local"]和["local"]。而事实上，一些JavaScript环境的确如此行事。但并不是所有的JavaScript环境都这样。其他一些环境在运行时根据包含函数f的块是否被执行来有条件地绑定函数f。(不仅使代码更难理解，而且还致使性能降低。这与with语句没什么不同。)

关于这一点ECMAScript标准说了什么呢？令人惊讶的是，几乎没有。直到ES5，JavaScript标准才承认局部块函数声明的存在。官方指定函数声明只能出现在其他函数或者程序的最外层。ES5甚至建议将在非标准环境的函数声明转变成警告或错误。一些流行的JavaScript实现在严格模式下将这类函数报告为错误(具有局部块函数声明的处于严格模式下的程序将报告一个语法错误)。这有助于检测出不可移植的代码，并为未来的标准版本在给局部块函数声明指定更明智和可移植的语义开辟了一条路。

在此期间，编写可移植的函数的最好方式是始终避免将函数声明置于局部块或子语句中。如果你想编写嵌套函数声明，应该将它置于其父函数的最外层，正如最开始的示例所示。另外，如果你需要有条件地选择函数，最好的办法是使用var声明和函数表达式来实现。

```
function f() { return "global"; }

function test(x) {
    var g = f, result = [];
    if (x) {
        g = function() { return "local"; }

        result.push(g());
    }
    result.push(g());
    return result;
}
```

这消除了内部变量(重命名为g)作用域的神秘性。它无条件地作为局部变量被绑定，而仅仅只有赋值语句是有条件的。结果很明确，该函数完全可移植。

提示

- 始终将函数声明置于程序或被包含的函数的最外层以避免不可移植的行为。
- 使用var声明和有条件的赋值语句替代有条件的函数声明。

第 16 条：避免使用 eval 创建局部变量

JavaScript 的 eval 函数是一个令人难以置信的强大、灵活的工具。强大的工具容易被滥用，所以了解它们是值得的。错误使用 eval 函数的最简单的方式之一是允许它干扰作用域。

调用 eval 函数会将其参数作为 JavaScript 程序进行解释。但是该程序运行于调用者的局部作用域中。嵌入到程序的全局变量会被创建为调用程序的局部变量。

```
function test(x) {
    eval("var y = x;"); // dynamic binding
    return y;
}
test("hello"); // "hello"
```

这个例子看起来很清晰，但此 var 声明语句与将其直接放置在 test 函数体中的行为略有不同。只有当 eval 函数被调用时此 var 声明语句才会被调用。只有当条件语句被执行时，放置在该条件语句中的 eval 函数才会将其变量加入到作用域中。

```
var y = "global";
function test(x) {
    if (x) {
        eval("var y = 'local';"); // dynamic binding
    }
    return y;
}
test(true);  // "local"
test(false); // "global"
```

基于作用域决定程序的动态行为通常是个坏主意。导致的结果是，即使想简单地理解变量是如何绑定的都需要了解程序执行的细节。当源代码将未在局部作用域内定义的变量传递给 eval 函数时，程序将变得特别棘手：

```
var y = "global";
function test(src) {
    eval(src); // may dynamically bind
    return y;

}
test("var y = 'local';"); // "local"
test("var z = 'local';"); // "global"
```

这段代码很脆弱，也不安全。它赋予了外部调用者能改变 test 函数内部作用域的能力。期望 eval 函数能修改自身包含的作用域对 ES5 严格模式的兼容性也是不可靠的 。ES5 严格模式将 eval 函数运行在一个嵌套的作用域中以防止这种污染。保证 eval 函数不影响外部作用域的一个简单方法是在一个明确的嵌套作用域中运行它。

```
var y = "global";
function test(src) {
    (function() { eval(src); })();
    return y;
}

test("var y = 'local';"); // "global"
test("var z = 'local';"); // "global"
```

提示

- 避免使用 eval 函数创建的变量污染调用者的作用域。
- 如果 eval 函数代码可能创建全局变量，将此调用封装到嵌套的函数中以防止作用域污染。

第 17 条：间接调用 eval 函数优于直接调用

eval 函数有一个秘密武器：它不仅仅是一个函数。

大多数函数只能访问定义它们所在的作用域，而不能访问除此之外的作用域。然而，eval 函数具有访问调用它那时的整个作用域的能力。这是非常强大的能力。当编译器编写者首次设法优化 JavaScript 时，他们发现 eval 函数很难高效地调用任何一个函数，因为一旦被调用的函数是 eval 函数，那么每个函数调用都需要确保在运行时整个作用域对 eval 函数是可访问的。

作为一种折中的解决方案，语言标准演化出了辨别两种不同的调用 eval 函数的方法。函数调用涉及 eval 标识符，被认为是一种“直接”调用 eval 函数的方式。

```
var x = "global";
function test() {
    var x = "local";
    return eval("x"); // direct eval
}
test(); // "local"
```

在这种情况下，编译器需要确保被执行的程序具有完全访问调用者局部作用域的权限。其他调用 eval 函数的方式被认为是“间接”的。这些方式在全局作用域内对 eval 函数的参数求值。例如，绑定 eval 函数到另一个变量名，通过该变量名调用函数会使代码失去对所有局部作用域的访问能力。

```
var x = "global";
function test() {
    var x = "local";
```

```
    var f = eval;
    return f("x"); // indirect eval
}
test(); // "global"
```

直接调用 eval 函数的确切的定义取决于 ECMAScript 标准相当特殊的规范语言。在实践中，唯一能够产生直接调用 eval 函数的语法是可能被（许多的）括号包裹的名称为 eval 的变量。编写间接调用 eval 函数的一种简洁方式是使用表达式序列运算符（,）和一个明显毫无意义的数字字面量。

```
(0,eval)(src);
```

这个奇形怪状的函数调用是如何工作的呢？数字字面量 0 被求值但其值被忽略掉了，括号表示的序列表达式产生的结果是 eval 函数。因此，(0,eval) 的行为几乎与简单的 eval 函数标识符完全一致，一个重要的区别在于整个调用表达式被视为是一种间接调用 eval 函数的方式。

直接调用 eval 函数的能力可能很容易被滥用。例如，对一个来自网络的源字符串进行求值，可能会暴露其内部细节给一些未受信者。第 16 条探讨了使用 eval 函数动态创建局部变量的危害。这些危害只可能与直接调用 eval 函数相关。此外，直接调用 eval 函数性能上的损耗也是相当高昂的。通常情况下，你要承担直接调用 eval 函数导致其包含的函数以及所有直到程序最外层的函数运行相当缓慢的风险。

出于某些原因偶尔也需要使用直接调用 eval 函数。但是，除非有一个检查局部作用域的特别能力的明确需求，否则应当使用更不容易滥用、更廉价的间接调用 eval 函数的方式。

提示

- 将 eval 函数同一个毫无意义的字面量包裹在序列表达式中以达到强制使用间接调用 eval 函数的目的。
- 尽可能间接调用 eval 函数，而不要直接调用 eval 函数。