回 false）。在严格模式下，尝试删除变量会导致错误：

```
// 删除变量
// 非严格模式：静默失败
// 严格模式：抛出 ReferenceError
let color = "red";
delete color;
```

严格模式也对变量名增加了限制。具体来说，不允许变量名为 implements、interface、let、package、private、protected、public、static 和 yield。这些是目前的保留字，可能在将来的 ECMAScript 版本中用到。如果在严格模式下使用这些名称作为变量名，则会导致语法错误。

B.3　对象

在严格模式下操作对象比在非严格模式下更容易抛出错误。严格模式倾向于在非严格模式下会静默失败的情况下抛出错误，增加了开发中提前发现错误的可能性。

首先，以下几种情况下试图操纵对象属性会引发错误。

- 给只读属性赋值会抛出 TypeError。
- 在不可配置属性上使用 delete 会抛出 TypeError。
- 给不存在的对象添加属性会抛出 TypeError。

另外，与对象相关的限制也涉及通过对象字面量声明它们。在使用对象字面量时，属性名必须唯一。例如：

```
// 两个属性重名
// 非严格模式：没有错误，第二个属性生效
// 严格模式：抛出 SyntaxError
let person = {
  name: "Nicholas",
  name: "Greg"
};
```

这里的对象字面量 person 有两个叫作 name 的属性。第二个属性在非严格模式下是最终的属性。但在严格模式下，这样写是语法错误。

> **注意**　ECMAScript 6 删除了对重名属性的这个限制，即在严格模式下重复的对象字面量属性键不会抛出错误。

B.4　函数

首先，严格模式要求命名函数参数必须唯一。看下面的例子：

```
// 命名参数重名
// 非严格模式：没有错误，只有第二个参数有效
// 严格模式：抛出 SyntaxError
function sum (num, num){
  // 函数代码
}
```

在非严格模式下，这个函数声明不会抛出错误。这样可以通过名称访问第二个 num，但只能通过 arguments 访问第一个参数。

arguments 对象在严格模式下也有一些变化。在非严格模式下，修改命名参数也会修改 arguments 对象中的值。而在严格模式下，命名参数和 arguments 是相互独立的。例如：

```
// 修改命名参数的值
// 非严格模式：arguments 会反映变化
// 严格模式：arguments 不会反映变化
function showValue(value){
  value = "Foo";
  alert(value);          // "Foo"
  alert(arguments[0]);   // 非严格模式："Foo"
                         // 严格模式："Hi"
}
showValue("Hi");
```

在这个例子中，函数 showValue()有一个命名参数 value。调用这个函数时给它传入参数"Hi"，该值会赋给 value。在函数内部，value 被修改为"Foo"。在非严格模式下，这样也会修改 arguments[0]的值，但在严格模式下则不会。

另一个变化是去掉了 arguments.callee 和 arguments.caller。在非严格模式下，它们分别引用函数本身和调用函数。在严格模式下，访问这两个属性中的任何一个都会抛出 TypeError。例如：

```
// 访问 arguments.callee
// 非严格模式：没问题
// 严格模式：抛出 TypeError
function factorial(num){
  if (num <= 1) {
    return 1;
  } else {
    return num * arguments.callee(num-1)
  }
}
let result = factorial(5);
```

类似地，读或写函数的 caller 或 callee 属性也会抛出 TypeError。因此对这个例子而言，访问 factorial.caller 和 factorial.callee 也会抛出错误。

另外，与变量一样，严格模式也限制了函数的命名，不允许函数名为 implements、interface、let、package、private、protected、public、static 和 yield。

关于函数的最后一个变化是不允许函数声明，除非它们位于脚本或函数的顶级。这意味着在 if 语句中声明的函数现在是个语法错误：

```
// 在 if 语句中声明函数
// 非严格模式：函数提升至 if 语句外部
// 严格模式：抛出 SyntaxError
if (true){
  function doSomething(){
    // ...
  }
}
```

所有浏览器在非严格模式下都支持这个语法，但在严格模式下则会抛出语法错误。

B.4.1　函数参数

ES6 增加了剩余操作符、解构操作符和默认参数，为函数组织、结构和定义参数提供了强大的支持。ECMAScript 7 增加了一条限制，要求使用任何上述先进参数特性的函数内部都不能使用严格模式，否则

会抛出错误。不过，全局严格模式还是允许的。

```
// 可以
function foo(a, b, c) {
  "use strict";
}

// 不可以
function bar(a, b, c='d') {
  "use strict";
}

// 不可以
function baz({a, b, c}) {
  "use strict";
}

// 不可以
function qux(a, b, ...c) {
  "use strict";
}
```

ES6 增加的这些新特性期待参数与函数体在相同模式下进行解析。如果允许编译指示"use strict"出现在函数体内，JavaScript 解析器就需要在解析函数参数之前先检查函数体内是否存在这个编译指示，而这会带来很多问题。为此，ES7 规范增加了这个约定，目的是让解析器在解析函数之前就确切知道该使用什么模式。

B.4.2 eval()

eval()函数在严格模式下也有变化。最大的变化是 eval()不会再在包含上下文中创建变量或函数。例如：

```
// 使用 eval()创建变量
// 非严格模式：警告框显示 10
// 严格模式：调用 alert(x)时抛出 ReferenceError
function doSomething(){
  eval("let x = 10");
  alert(x);
}
```

以上代码在非严格模式下运行时，会在 doSomething()函数内部创建局部变量 x，然后 alert()会显示这个变量的值。在严格模式下，调用 eval()不会在 doSomething()中创建变量 x，由于 x 没有声明，alert()会抛出 ReferenceError。

变量和函数可以在 eval()中声明，但它们会位于代码执行期间的一个特殊的作用域里，代码执行完毕就会销毁。因此，以下代码就不会出错：

```
"use strict";
let result = eval("let x = 10, y = 11; x + y");
alert(result);    // 21
```

这里在 eval()中声明了变量 x 和 y，将它们相加后返回得到的结果。变量 result 会包含 x 和 y 相加的结果 21，虽然 x 和 y 在调用 alert()时已经不存在了，但不影响结果的显示。

B.4.3　eval 与 arguments

严格模式明确不允许使用 eval 和 arguments 作为标识符和操作它们的值。例如：

```
// 将 eval 和 arguments 重新定义为变量
// 非严格模式：可以，没有错误
// 严格模式：抛出 SyntaxError
let eval = 10;
let arguments = "Hello world!";
```

在非严格模式下，可以重写 eval 和 arguments。在严格模式下，这样会导致语法错误。不能用它们作为标识符，这意味着下面这些情况都会抛出语法错误：

- 使用 let 声明；
- 赋予其他值；
- 修改其包含的值，如使用++；
- 用作函数名；
- 用作函数参数名；
- 在 try/catch 语句中用作异常名称。

B.5　this 强制转型

JavaScript 中最大的一个安全问题，也是最令人困惑的一个问题，就是在某些情况下 this 的值是如何确定的。使用函数的 apply()或 call()方法时，在非严格模式下 null 或 undefined 值会被强制转型为全局对象。在严格模式下，则始终以指定值作为函数 this 的值，无论指定的是什么值。例如：

```
// 访问属性
// 非严格模式：访问全局属性
// 严格模式：抛出错误，因为 this 值为 null
let color = "red";
function displayColor() {
  alert(this.color);
}
displayColor.call(null);
```

这里在调用 displayColor.call()时传入 null 作为 this 的值，在非严格模式下该函数的 this 值是全局对象。结果会显示"red"。在严格模式下，该函数的 this 值是 null，因此在访问 null 的属性时会抛出错误。

通常，函数会将其 this 的值转型为一种对象类型，这种行为经常被称为“装箱”（boxing）。这意味着原始值会转型为它们的包装对象类型。

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

foo.call();  // Window {}
foo.call(2); // Number {2}
```

在严格模式下执行以上代码时，this 的值不会再“装箱”：

```
function foo() {
  "use strict";
  console.log(this);
```

```
}

foo.call();  // undefined
foo.call(2); // 2
```

B.6　类与模块

类和模块都是 ECMAScript 6 新增的代码容器特性。在之前的 ECMAScript 版本中没有类和模块这两个概念，因此不用考虑从语法上兼容之前的 ECMAScript 版本。为此，TC39 委员会决定在 ES6 类和模块中定义的所有代码默认都处于严格模式。

对于类，这包括类声明和类表达式，构造函数、实例方法、静态方法、获取方法和设置方法都在严格模式下。对于模块，所有在其内部定义的代码都处于严格模式。

B.7　其他变化

严格模式下还有其他一些需要注意的变化。首先是消除 `with` 语句。`with` 语句改变了标识符解析时的方式，严格模式下为简单起见已去掉了这个语法。在严格模式下使用 `with` 会导致语法错误：

```
// 使用 with 语句
// 非严格模式：允许
// 严格模式：抛出 SyntaxError
with(location) {
  alert(href);
}
```

严格模式也从 JavaScript 中去掉了**八进制字面量**。八进制字面量以前导 0 开始，一直以来是很多错误的源头。在严格模式下使用八进制字面量被认为是无效语法：

```
// 使用八进制字面量
// 非严格模式：值为 8
// 严格模式：抛出 SyntaxError
let value = 010;
```

ECMAScript 5 修改了非严格模式下的 `parseInt()`，将八进制字面量当作带前导 0 的十进制字面量。例如：

```
// 在 parseInt()中使用八进制字面量
// 非严格模式：值为 8
// 严格模式：值为 10
let value = parseInt("010");
```

附录 C
JavaScript 库和框架

JavaScript 库帮助弥合浏览器之间的差异，能够简化浏览器复杂特性的使用。库主要分两种形式：**通用**和**专用**。通用 JavaScript 库支持常用的浏览器功能，可以作为网站或 Web 应用程序开发的基础。专用 JavaScript 库支持特定功能，只适合网站或 Web 应用程序的一部分。本附录会从整体上介绍这些库及其功能，并提供相关参考资源。

C.1 框架

“框架”（framework）涵盖各种不同的模式，但各自具有不同的组织形式，用于搭建复杂应用程序。使用框架可以让代码遵循一致的约定，能够灵活扩展规模和复杂性。框架对常见的任务提供了稳健的实现机制，比如组件定义及重用、控制数据流、路由，等等。

JavaScript 框架越来越多地表现为单页应用程序（SPA，Single Page Application）。SPA 使用 HTML5 浏览器历史 API，在只加载一个页面的情况下通过 URL 路由提供完整的应用程序用户界面。框架在应用程序运行期间负责管理应用程序的状态以及用户界面组件。大多数流行的 SPA 框架有坚实的开发者社区和大量第三方扩展。

C.1.1 React

React 是 Facebook 开发的框架，专注于模型–视图–控制器（MVC，Model-View-Controller）模型中的“视图”。专注的范围让它可以与其他框架或 React 扩展合作，实现 MVC 模式。React 使用单向数据流，是声明性和基于组件的，基于虚拟 DOM 高效渲染页面，提供了在 JavaScript 包含 HTML 标记的 JSX 语法。Facebook 也维护了一个 React 的补充框架，叫作 Flux。

- 许可：MIT

C.1.2 Angular

谷歌在 2010 年首次发布的 Angular 是基于模型–视图–视图模型（MVVM）架构的全功能 Web 应用程序框架。2016 年，这个项目分叉为两个分支：Angular 1.*x* 和 Angular 2。前者是最初的 AngularJS 项目，后者则是基于 ES6 语法和 TypeScript 完全重新设计的框架。这两个版本的最新发布版都是指令和基于组件的实现，两个项目都有稳健的开发者社区和第三方扩展。

- 许可：MIT

C.1.3 Vue

Vue 是类似 Angular 的全功能 Web 应用程序框架，但更加中立化。自 2014 年 Vue 发布以来，它的

开发者社区发展迅猛，很多开发者因为其高性能和易组织，同时不过于主观而选择了 Vue。

❑ 许可：MIT

C.1.4　Ember

Ember 与 Angular 非常相似，都是 MVVM 架构，并使用首选的约定来构建 Web 应用程序。2015 年发布的 2.0 版引入了很多 React 框架的行为。

❑ 许可：MIT

C.1.5　Meteor

Meteor 与前面的框架都不一样，因为它是同构的 JavaScript 框架，这意味着客户端和服务器共享一套代码。Meteor 也使用实时数据更新协议，持续从 DB 向客户端推送新数据。虽然 Meteor 是一个极为主观的框架，但好处是可以使用其稳健的开箱即用特性快速开发应用程序。

❑ 许可：MIT

C.1.6　Backbone.js

Backbone.js 是构建于 Underscore.js 之上的一个最小化 MVC 开源库，为 SPA 做了大量优化，可以方便地更新应用程序状态。

❑ 许可：MIT

C.2　通用库

通用 JavaScript 库提供适应任何需求的功能。所有通用库都致力于通过将常用功能封装为新 API，来补偿浏览器接口、弥补实现差异。其中有些 API 与原生功能相似，而另一些 API 则完全不同。通用库通常会提供与 DOM 的交互，对 Ajax 的支持，还有辅助常见任务的实用方法。

C.2.1　jQuery

jQuery 是为 JavaScript 提供函数式编程接口的开源库。该库的核心是通过 CSS 选择符匹配 DOM 元素，通过调用链，jQuery 代码看起来更像描述故事情节而不是 JavaScript 代码。这种代码风格在设计师和原型设计者中非常流行。

❑ 许可：MIT 或 GPL

C.2.2　Google Closure Library

Google Closure Library 是通用 JavaScript 工具包，与 jQuery 在很多方面都很像。这个库包含非常多的模块，涵盖底层操作和高层组件和部件。Google Closure Library 可以按需加载模块，并使用 Google Closure Compiler（附录 D 会介绍）构建。

❑ 许可：Apache 2.0

C.2.3　Underscore.js

Underscore.js 并不是严格意义上的通用库，但提供了 JavaScript 函数式编程的额外能力。它的文档

将 Underscore.js 看成 jQuery 的组件，但提供了更多底层能力，用于操作对象、数组、函数和其他 JavaScript 数据类型。

❑ 许可：MIT

C.2.4　Lodash

与 Underscore.js 一样，Lodash 也是实用库，用于扩充 JavaScript 工具包。Lodash 提供了很多操作原生类型，如数组、对象、函数和原始值的增强方法。

❑ 许可：MIT

C.2.5　Prototype

Prototype 是对常见 Web 开发任务提供简单 API 的开源库。Prototype 最初是为了 Ruby on Rails 开发者开发的，由类驱动，旨在为 JavaScript 提供类定义和继承。为此，Prototype 提供了大量的类，将常用和复杂的功能封装为简单的 API 调用。Prototype 包含在一个文件里，可以轻松地插入页面中使用。

❑ 许可：MIT 及 CC BY-SA 3.0

C.2.6　Dojo Toolkit

Dojo Toolkit 是以包系统为基础的开源库，将功能分门别类地划分为包，可以按需加载。Dojo 支持各种配置选项，几乎涵盖了使用 JavaScript 所需的一切。

❑ 许可：“新”BSD 许可或 Academic Free License 2.1

C.2.7　MooTools

MooTools 是简洁、优化的开源库，为原生 JavaScript 对象添加方法，在熟悉的接口上提供新功能。由于体积小、API 简单，MooTools 在 Web 开发者中很受欢迎。

❑ 许可：MIT

C.2.8　qooxdoo

qooxdoo 是致力于全周期支持 Web 应用程序开发的开源库。通过实现自己的类和接口，qooxdoo 创建了类似传统面向对象编程语言的模型。这个库包含完整的 GUI 工具包和编译器，用于简化前端构建过程。qooxdoo 最初是网站托管公司 1&1 的内部库，后来基于开源许可对外发布。

❑ 许可：LGPL 或 EPL

C.3　动画与特效

动画与特效是 Web 开发中越来越重要的一部分。在网站中创造流畅的动画并不容易。为此，不少库开发者已开发了包含各种动画和特效的库。前面提到的不少 JavaScript 库也包含动画特性。

C.3.1　D3

数据驱动文档（D3，Data Driven Documents）是非常流行的动画库，也是今天非常稳健和强大的

JavaScript 数据可视化工具。D3 提供了全面完整的特性，涵盖 canvas、SVG、CSS 和 HTML5 可视化。使用 D3 可以极为精准地控制最终渲染的输出。

❑ 许可：BSD

C.3.2　three.js

three.js 是当前非常流行的 WebGL 库。它提供了轻量级 API，可以实现复杂 3D 渲染与动效。

❑ 许可：MIT

C.3.3　moo.fx

moo.fx 是基于 Prototype 或 MooTools 使用的开源动画库。它的目标是尽可能小（最新版 3KB），并使开发者只写尽可能少的代码。moo.fx 默认包含 MooTools，也可以单独下载，与 Prototype 一起使用。

❑ 许可：MIT

C.3.4　Lightbox

Lightbox 是创建简单图像覆盖特效的 JavaScript 库，依赖 Prototype 和 script.aculo.us 实现特效。其基本思想是可以使用户在当前页面的一个覆盖层中查看一个图像或多个图像。可以自定义覆盖层的外观和过渡。

❑ 许可：Creative Commons Attribution 2.5

附录 D
JavaScript 工具

编写 JavaScript 代码与编写其他编程语言代码类似，都有专门的工具帮助提高开发效率。JavaScript 开发者可以使用的工具一直在增加，这些工具可以帮助开发者更容易定位问题、优化代码和部署上线。其中有些工具是在 JavaScript 中使用的，而其他工具则是在浏览器之外使用的。本附录会全面介绍这些工具，并提供相关参考资源。

> **注意** 有不少工具会在本附录中多次出现。今天的很多 JavaScript 工具是多合一的，因此适用于多个领域。

D.1 包管理

JavaScript 项目经常要使用第三方库和资源，以避免代码重复和加速开发。第三方库也称为“包”，托管在公开代码仓库中。包的形式可以是直接交付给浏览器的资源、与项目一起编译的 JavaScript 库，或者是项目开发流程中的工具。这些包总在活跃开发和不断修订，有不同的版本。JavaScript 包管理器可以管理项目依赖的包，涉及获取和安装，以及版本控制。

包管理器提供了命令行界面，用于安装和删除项目依赖。项目的配置通常存储在项目本地的配置文件中。

D.1.1 npm

npm，即 Node 包管理器（Node Package Manager），是 Node.js 运行时默认的包管理器。在 npm 仓库中发布的第三方包可以指定为项目依赖，并通过命令行本地安装。npm 仓库包含服务端和客户端 JavaScript 库。

npm 是为在服务器上使用而设计的，服务器对依赖大小并不敏感。在安装包时，npm 使用嵌套依赖树解析所有项目依赖，每个项目依赖都会安装自己的依赖。这意味着如果项目依赖三个包 A、B 和 C，而这三个包又都依赖不同版本的 D，则 npm 会安装包 D 的三个版本。

D.1.2 Bower

Bower 与 npm 在很多方面相似，包括包安装和管理 CLI，但它专注于管理要提供给客户端的包。Bower 与 npm 的一个主要区别是 Bower 使用打平的依赖结构。这意味着项目依赖会共享它们依赖的包，用户的任务是解析这些依赖。例如，如果你的项目依赖三个包 A、B 和 C，而这三个包又都依赖不同版本的 D，那你就需要找一个同时满足 A、B、C 需求的包 D。这是因为打平的依赖结构要求每个包只能安装一个版本。