附录A
ES.Next

从 ECMAScript 2015（ES6）开始，TC39 委员会改为每年发布一版新 ECMAScript 规范。这样各个提案可以独立发展，每年所有达到成熟阶段的提案会被打包发布到新一版标准中。不过，打包多少特性并不重要，主要取决于浏览器厂商实现的情况。一旦提案进入第 4 阶段（stage 4），其内容就不会更改，并通常会包含在下一版 ECMAScript 规范中，浏览器就会着手根据自己的计划实现提案的特性。

本附录将讨论一些可能会被纳入之后 ECMAScript 版本中的处于后期阶段的提案。

> **注意** 本附录中介绍的特性相对较新，往往只有最新版本的浏览器才支持。使用这些特性之前，请参考 Can I Use 网站确定支持相应特性的浏览器及其版本。

A.1 分组同步可迭代对象

`Map.groupBy()`静态方法用于把一个可迭代对象的元素分组为映射的条目，条目的键由回调函数决定。比如，下面的例子将一个数组的元素按照数值的符号分组并得到一个映射：

```
Map.groupBy([2, -3, 1, 0, -5, 6, 7], x => Math.sign(x))
```

以上代码会得到类似下面的映射：

```
new Map()
  .set(0, [0])
  .set(-1, [-3, -5])
  .set(1, [2, 1, 6, 7])
```

此外，还有一个静态方法 `Object.groupBy()`，它会产生对象而非映射：

```
Object.groupBy([2, -3, 1, 0, -5, 6, 7], x => Math.sign(x))
```

以上代码产生的对象如下：

```
{
  '0': [0],
  '1': [2, 1, 6, 7],
  '-1': [-3, -5]
}
```

`Map.groupBy()`和 `Object.groupBy()`方法都接受两个参数，第一个参数是可迭代对象，第二个参数是回调函数。回调函数针对可迭代对象的每个元素都会执行一次，返回字符串或符号，这个字符串或符号表示元素的分组，并用作上述两个静态方法返回的映射或对象的键。在返回的映射或对象中，每个键对应一个数组，数组中包含同一分组的元素。

A.2 `Promise.withResolvers()`

`Promise.withResolvers()` 静态方法返回一个对象，对象中包含一个新的期约对象和两个函数，这两个函数分别用于解决和拒绝新期约对象，与通过 `Promise()` 构造函数创建新期约对象时传给执行器函数的两个参数对应。

```
const { promise, resolve, reject } = Promise.withResolvers();
```

以上代码与下面的代码是等价的，但更简洁，也省去了使用 `let` 声明的麻烦：

```
let resolve, reject;
const promise = new Promise((res, rej) => {
  resolve = res;
  reject = rej;
});
```

`Promise.withResolvers()` 提供了一种新的创建期约对象的方式，采用这种方式可以在同一作用域中得到新的期约对象，以及用于解决和拒绝该期约的函数。这样，我们就不再局限于只能够在执行器函数中使用解决或拒绝期约的函数。

这个新方法可以支持更高级的用法，比如在可重复发生的事件中重用解决和拒绝函数，特别是在基于流和队列的用例中。相比在执行器函数中嵌套代码，使用这种新方式编写的代码也会更直观。

A.3 正则表达式的新标志：`/v`

正则表达式标志 `/v` 是对 `/u` 的升级（在字母表中，v 位于 u 后面），`/u` 让引擎将模式视为一系列 Unicode 码点，而 `/v` 在此基础上则支持更多 Unicode 特性。

由于 `/v` 和 `/u` 标志解释正则表达式的方式不兼容，所以这两个标志同时出现会导致 `SyntaxError`。使用 `/v` 标志时，除了 `/u` 的特性，还额外支持以下特性。

- `\p` 转义序列除了匹配字符，还可以额外用于匹配字符串属性。
- 字符类语法升级为允许插入、联合和相减的语法，同时也能匹配多个 Unicode 字符。
- 字符类取反语法 `[^...]` 会构造一个取反字符类，而不是对匹配结果取反，避免不区分大小写匹配中的某些令人困惑的行为。

要了解更多关于 `/v` 标志的内容，可以参考 MDN 的介绍。

A.4 `ArrayBuffer` 和 `SharedArrayBuffer` 的新特性

`ArrayBuffer` 可以就地缩放大小：

```
const buf = new ArrayBuffer(2, {maxByteLength: 4});
const typedArray = new Uint8Array(buf, 2);
// typedArray.length === 0
buf.resize(4);
// typedArray.length === 2
```

另外，`ArrayBuffer` 也新增了一个 `.transfer()` 方法，用于传送 `ArrayBuffer` 对象。

`SharedArrayBuffer` 也支持调整大小，但只能增长，不能收缩。另外，因为本身不可以传送，所以 `SharedArrayBuffer` 没有 `.transfer()` 方法。

A.5　保证字符串格式良好

字符串实例增加了两个方法，用于保证字符串格式良好。

- `isWellFormed()`：检查字符串是否格式良好，且不包含残缺的代理对。相比于自定义实现，这个方法性能更好，因为引擎可以直接访问字符串的内部表示。
- `toWellFormed()`：返回字符串的副本，但会将其中包含的残缺代理对（如果有）替换成码点`0xFFFD`（Unicode 替换字符串），从而保证结果字符串格式良好，可用于需要格式良好的字符串作为输入的函数，比如`encodeURI()`。

A.6　`Atomics.waitAsync()`

`Atomics.waitAsync()`方法用于异步等待对共享内存的修改，返回一个期约对象。语法如下：

```
Atomics.waitAsync(typedArray, index, value)
Atomics.waitAsync(typedArray, index, value, timeout)
```

要了解更多关于`Atomics.waitAsync()`方法的内容，可以参考 MDN 的介绍。

A.7　从后向前查找的数组方法

这个提案给数组和定型数组添加了`.findLast()`和`.findLastIndex()`方法，分别对应于原有的`.find()`和`.findIndex()`方法。这一对新方法与之前对应方法唯一的区别，就是它们会返回最后一个而不是第一个匹配项。换句话说，`.findLast()`等价于`.reverse().find()`，但是不需要先反转数组了。下面的代码对比演示了这两个新方法：

```
const hasFoo = (string) => string.includes("foo");
const strings = ["hello", "foo", "world", "foobar", "baz"];

console.log(strings.find(hasFoo));
// "foo"

console.log(strings.findIndex(hasFoo));
// 1

console.log(strings.findLast(hasFoo));
// "foobar"

console.log(strings.findLastIndex(hasFoo));
// 3
```

A.8　Hashbang/Shebang 语法

Hashbang，也称为 shebang，指的是位于可执行脚本开头的一组字符，用来定义运行程序的解释器。类 Unix 平台使用它来指定把脚本传给哪个解释器去执行。来看看下面的示例脚本，路径为/home/myusername/scripts/myscript.js：

```
#!/usr/bin/env node

console.log("Hello, world!");
```

这里的代码告诉程序加载器使用/usr/bin/env node 来运行代码，并且将/home/myusername/scripts/myscript.js 作为第一个参数。

在本书写作时，JavaScript 引擎不会自动剥离 hashbang。在程序加载器执行 JavaScript 程序时，宿主必须先剥离 hashbang 以生成有效源代码，再把它传给执行引擎。这个提案把宿主的剥离动作移交给了执行引擎。

A.9　符号作为 WeakMap 键

在本书写作时，WeakMap 的键只能是对象，这可能会限制那些既想使用唯一值作为键，又希望键可以被垃圾回收的开发者。这个建议扩展了 WeakMap，允许使用 Symbol 作为键：

```
const wm = new WeakMap();

const key = Symbol('Key for weak map entry');
const data = {
  // ...
};

myWeakMap.set(key, data);
```

由于对象具有相同的身份行为，因此被用作 WeakMap 的键：只有通过访问原始对象才能验证对象的身份，且在严格比较中没有新对象能与已存在的对象匹配。然而，Symbol 在某些使用场景中可能更合适，这个提案旨在支持该能力以增强 WeakMap 的灵活性。

A.10　先复制再修改数组

这个建议给数组和定型数组添加了.toReversed()、.toSorted()和.with()方法，另外还只给数组添加了.toSpliced()方法。与数组和定型数组已有的就地修改数组的方法一一对应，这些新方法会**先创建数组的一个浅拷贝**，然后修改复制后的数组。下面的代码演示了区别所在：

```
// 这样执行的是就地反转
myArray.reverse();

// 这样是先创建一个副本，然后反转副本
[...myArray].reverse();

// 这样也是先创建副本，然后反转副本
myArray.toReversed()
```

- .toReversed()与.reverse()的行为对应。
- .toSorted(compareFn)与.sort()的行为对应。
- .toSpliced(start, deleteCount, ...items)与.splice()的行为对应。
- .with(index, value)没有对应的方法，它的行为是创建一个数组副本并用 value 替换位置在 index 处的值。

以下代码演示了这些方法。

```
const myArray = [5,7,3,9,2];

const reversedArray = myArray.toReversed();
```

```
console.log(reversedArray);
// [2,9,3,7,5]

const sortedArray = reversedArray.toSorted();
console.log(sortedArray);
// [2,3,5,7,9]

const splicedArray = sortedArray.toSpliced(1,3);
console.log(splicedArray);
// [3,5,7]

const withArray = splicedArray.with(1, 10);
console.log(withArray);
// [3,10,7]

// 原始数组不变!
console.log(myArray);
// [5,7,3,9,2]
```

附录 B
严格模式

ECMAScript 5 首次引入**严格模式**的概念。严格模式用于选择以更严格的条件检查 JavaScript 代码错误，可以应用到全局，也可以应用到函数内部。严格模式的好处是可以提早发现错误，因此可以捕获某些 ECMAScript 问题导致的编程错误。严格模式已得到所有主流浏览器支持。

B.1 选择使用

要选择使用严格模式，需要使用严格模式**编译指示**（pragma），即一个不赋值给任何变量的字符串：

```
"use strict";
```

这样一个即使在 ECMAScript 3 中也有效的字符串，可以兼容不支持严格模式的 JavaScript 引擎。支持严格模式的引擎会启用严格模式，而不支持的引擎会将这个编译指示当成一个未赋值的字符串字面量。

如果把这个编译指示应用到全局作用域，即函数外部，则整个脚本都会按照严格模式来解析。这意味着在最终会与其他脚本拼接为一个文件的脚本中添加了编译指示，会将该文件中的所有 JavaScript 置于严格模式之下。

也可以像下面这样只在一个函数内部开启严格模式：

```
function doSomething() {
  "use strict";
  // 其他代码
}
```

如果你不能控制页面中的所有脚本，那么建议只在经过测试的特定函数中启用严格模式。

B.2 类和模块

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.3 变量

严格模式下如何创建变量及何时会创建变量都会发生变化。第一个变化是不允许意外创建全局变量。在非严格模式下，以下代码可以创建全局变量：

```
// 变量未声明
// 非严格模式：创建全局变量
// 严格模式：抛出 ReferenceError
message = "Hello world!";
```

虽然这里的 message 没有前置 let 关键字，也没有明确定义为全局对象的属性，但仍然会自动创建为全局变量。在严格模式下，给未声明的变量赋值会在执行代码时抛出 ReferenceError。

相关的另一个变化是无法在变量上调用 delete。在非严格模式下允许这样，但可能会静默失败（返回 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 SyntaxError。

B.4　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: "Matt",
  name: "Alice"
};
```

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.5　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.5.1　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.5.2　`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.5.3 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.6 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.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 应用开发的基础。专用 JavaScript 库支持特定功能，只适合网站或 Web 应用的一部分。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。框架针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　Alpine.js

Alpine.js 是一款轻量级的 JavaScript 库，提供了一种富有表达性的语法，以声明的方式为 HTML 元素添加交互功能，而无须复杂的框架。它利用数据绑定和事件监听的强大功能，用最少的代码实现动态和响应式的用户界面。Alpine.js 可以与其他库和框架结合使用，也可以作为独立工具，使其成为 Web 开发者增强应用功能和用户体验的不错选择。凭借其简单直观的语法，Alpine.js 提供了较低的学习曲线，并提供了为网页添加交互性的一种快速高效的方法。

❑ 许可：MIT

C.1.5　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.1.6　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.1.7　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　实用库

JavaScript 库提供了预先编写的代码，帮助开发者加速开发过程、简化编码任务，并增强 Web 应用的功能和交互性。从 Lodash 这样的实用库到 D3.js 这样的数据可视化库，有着种类繁多且功能丰富的库可供选择。如果你打算从头构建某个功能，很可能已经有至少一个开源库能够实现它。以下将介绍一些活跃维护且流行的库。

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 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　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.2.6　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.2.7　Anime.js

Anime.js 是一个轻量级的 JavaScript 动画库，拥有简单却强大的 API。它可以操作 CSS 属性、SVG、DOM 属性和 JavaScript 对象。

❑ 许可：MIT

C.2.8　Chart.js

Chart.js 提供了一系列常用的图表类型、插件和自定义选项。除了丰富的内置图表类型，还可以使用社区维护的图表类型，也可以将几种图表类型组合成一个混合图表（基本上是在同一个画布上融合多

种图表类型）。Chart.js 高度可定制，可以通过自定义插件创建注释、缩放或拖放功能。

❑ 许可：MIT

C.2.9　Leaflet

Leaflet 是领先的开源 JavaScript 库，用于创建移动友好型交互地图。其 JavaScript 文件大小仅约 42 KB，但拥有大多数开发者所需的所有地图功能。Leaflet 在设计时注重简洁、性能和可用性。它在所有主要的桌面和移动平台上都能高效运行，可以通过众多插件进行扩展，并拥有美观、易用且文档完善的 API 以及简洁易读的源代码。

❑ 许可：BSD-2-Clause

C.2.10　Axios

Axios 是一个流行的 JavaScript 库，提供了简单的接口，用于在浏览器或 Node.js 中发起 HTTP 请求。它支持所有现代浏览器，并提供了一个与 Promise（期约）无缝配合的统一 API。Axios 可以处理多种 HTTP 请求和响应类型，包括 JSON、XML 和 FormData，并支持拦截器、请求与响应转换以及自动处理响应状态和头信息等功能。此外，Axios 内置了请求取消和错误处理功能，使其成为开发者在 JavaScript 应用中处理 HTTP 请求的可靠且灵活的选择。

❑ 许可：MIT

C.2.11　Rxjs

RxJS 是一个流行的 JavaScript 库，用于响应式编程，为管理异步数据流提供了方便的接口。它利用观察者模式表示数据流，并提供了一系列运算符，以强大的方式转换和组合这些数据流。

❑ 许可：Apache 2.0

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，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 Yarn

Yarn 是 Facebook 开发的定制包管理器，从很多方面看是 npm 的升级版。Yarn 可以通过自己的注册表访问相同的 npm 包，并且安装方式与 npm 也相同。Yarn 与 npm 的主要区别是提供了加速安装、包缓存、锁文件等功能，且提供了改进的包安全功能。

D.1.3　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。

D.2　模块加载器

模块加载器可以让项目按需从服务器获取模块，而不是一次性加载所有模块或包含所有模块的 JS 文件。ECMAScript 6 模块规范定义了浏览器原生支持动态模块加载的最终目标。因此，模块加载器作为某种腻子脚本，可以让客户端实现动态模块加载。

D.2.1　SystemJS

SystemJS 模块加载器可以在服务器上使用，也可以在客户端使用。它支持所有模块格式，包括 AMD、CommonJS、UMD 和 ES6；也支持浏览器内转译（考虑到性能，不推荐在大型项目中使用）。

D.2.2　RequireJS

RequireJS 构建于 AMD 模块规范之上，支持特别旧的浏览器。虽然 RequireJS 经实践证明很不错，但 JavaScript 社区整体上还是会抛弃 AMD 模块格式。因此不推荐在大型项目中使用 RequireJS。

D.3　模块打包器

模块打包器可以将任意格式、任意数量的模块合并为一个或多个文件，供客户端加载。模块打包器会分析应用程序的依赖图并按需排序模块。一般来说，应用程序最终只需要一个打包后的文件，但多个结果文件也是可以配置生成的。模块打包器有时候也支持打包原始或编译的 CSS 资源。最终生成的文件可以自执行，也可以多个资源拼接在一起按需执行。

D.3.1　Webpack

Webpack 拥有强大的功能和可扩展能力，是今天非常流行的打包工具。Webpack 可以绑定不同的模块类型，支持多种插件，且完全兼容大多数模板和转译库。

D.3.2　Parcel

Parcel 自动检测依赖项，省去了复杂的配置文件需求。Parcel 为许多常见的 Web 开发工作流程提供零配置设置，支持诸如 React、Vue.js 和 Angular 等流行框架。除了强大的打包功能外，Parcel 还包括诸如热模块替换、代码分割和代码压缩等功能。

D.3.3　Rollup

Rollup 在模块打包能力方面与 Browserify 类似，但内置了摇树优化功能。Rollup 可以解析应用程序

的依赖图，排除没有实际使用的模块。

D.4　编译/转译工具及静态类型系统

在代码编辑器中写的 Web 应用程序代码通常不是实际发送给浏览器的代码。开发者通常希望使用很新的 ECMAScript 特性，而这些特性未必所有浏览器都支持。此外，开发者也经常希望使用静态类型系统或特性在 ECMAScript 规范之外强化自己的代码。有很多工具可以满足上述需求。

D.4.1　Babel

Babel 是将最新 ECMAScript 规范代码编译为兼容 ECMA 版本的一个常用工具。Babel 也支持 React 的 JSX，支持各种插件，与所有主流构建工具兼容。

D.4.2　Google Closure Compiler

Google Closure Compiler 是强大的 JavaScript 编译器，能够执行各种级别的编译优化，同时也是稳健的静态类型检查系统。其类型注解要求以 JSDoc 风格编写。

D.4.3　TypeScript

微软的 TypeScript 是 JavaScript 支持类型的超集，增加了稳健的静态类型检查和主要语法增强。因为它是 JavaScript 严格的超集，所以常规 JavaScript 代码也是有效的 TypeScript 代码。TypeScript 也可以使用类型定义文件指定已有 JavaScript 库的类型信息。

D.4.4　Flow

Flow 是 Facebook 推出的简单的 JavaScript 类型注解系统，其类型语法与 TypeScript 非常相似，但除了类型声明没有增加其他语言特性。

D.5　高性能脚本工具

关于 JavaScript 的一个常见批评是运行速度慢，不适合要求很高的计算。无论这里所说的“慢”是否符合实际，毋庸置疑的是这门语言从一开始就没有考虑支持敏捷的计算。为解决性能问题，有很多项目致力于改造浏览器执行代码的方式，以便让 JavaScript 代码的速度可以接近原生代码速度，同时利用硬件优化。

D.5.1　WebAssembly

WebAssembly 项目（简称 Wasm）正在实现一门语言，该语言可以在多处执行（可移植）并以二进制语言形式存在，可以作为多种低级语言（如 C++和 Rust）的编译目标。WebAssembly 代码在浏览器的一个与 JavaScript 完全独立的虚拟机中运行，与各种浏览器 API 交互的能力极为有限。它可以与 JavaScript 和 DOM 以间接、受限的方式交互，但其更大的目标是创造一门可以在 Web 浏览器中（以及在任何地方）运行的速度极快的语言，并提供接近原生的性能和硬件加速。WebAssembly 系列规范 2019 年 12 月 5 日已成为 W3C 的正式推荐标准，是浏览器技术中非常值得期待的领域。

D.5.2　asm.js

asm.js 的理论基础是 JavaScript 编译后比硬编码 JavaScript 运行得更快。asm.js 是 JavaScript 的子集，可以作为低级语言的编译目标，并在常规浏览器或 Node.js 引擎中执行。现代 JavaScript 引擎在运行时推断类型，而 asm.js 代码通过使用词法提示将这些类型推断（及其相关操作）的计算大大降低。asm.js 广泛使用了定型数组（`TypedArray`），相比常规的 JavaScript 数组能够显著提升性能。asm.js 没有 WebAssembly 快，但通过编译显著提升了性能。

D.5.3　Emscripten 与 LLVM

虽然 Emscripten 从未在浏览器中执行，但它是重要的工具包，可以将低级代码编译为 WebAssembly 和 asm.js。Emscripten 使用 LLVM 编译器将 C、C++和 Rust 代码编译为可以直接在浏览器中运行的代码（asm.js），或者可以在浏览器虚拟机中执行的代码（WebAssembly）。

D.6　编辑器

VIM、Emacs 及其同类的文本编辑器非常优秀，但随着构建环境和项目规模逐渐复杂，编辑器最好能够自动化常见任务，如代码自动完成、文件自动格式化、自动检查代码错误、自动补足项目目录。目前有很多编辑器和 IDE 支持这些功能，既有免费的也有收费的。

D.6.1　Sublime Text

Sublime Text 是比较流行的闭源文本编辑器。它可用于开发各种语言，还提供了大量可扩展的插件，由社区来维护。Sublime Text 的性能非常突出。

- **类型**：收费

D.6.2　Atom

Atom 是 GitHub 的开源编辑器，与 Sublime Text 有很多相同的特性，如社区在蓬勃发展且拥有第三方扩展包。Atom 的性能稍差，但它在不断地提升。

- **类型**：免费

D.6.3　Brackets

Brackets 是 Adobe 的开源编辑器，与 Atom 类似。但 Brackets 是专门为 Web 开发者设计的，提供了许多非常令人印象深刻的、面向前端编码的独特功能。该编辑器还有丰富的插件。

- **类型**：免费

D.6.4　Visual Studio Code

微软的 Visual Studio Code 是基于 Electron 框架的开源代码编辑器。与其他主流编辑器一样，Visual Studio Code 是高度可扩展的。

- **类型**：免费

D.6.5　WebStorm

WebStorm 是 JetBrains 的高性能 IDE，号称终极项目开发工具包，集成了前沿的前端框架，也集成了大多数构建工具和版本控制系统。

- **类型**：免费试用；之后收费。

D.7　构建工具、自动化系统和任务运行器

把本地开发的项目目录转换为线上应用程序需要一系列步骤。每个步骤都需要细分为很多子任务，如构建和部署应用程序要涉及模块打包、编译、压缩和发布静态资源，等等。运行单元和集成测试也涉及初始化测试套件和控制无头浏览器。为了让管理和使用这些任务更容易，也出现了很多工具可以用来更高效地组织和拼接这些任务。

D.7.1　npm

虽然严格来讲 npm 并不算是一个构建工具，但它提供了一个 scripts 功能，许多开发者发现它作为任务执行器非常好用且无须额外配置。scripts 直接定义在 NodeJS 的 package.json 中。

D.7.2　Grunt

Grunt 是在 Node.js 环境下运行的任务运行器，使用配置对象声明如何执行任务。Grunt 有庞大的社区和众多插件可以支持项目构建。

D.7.3　Gulp

与 Grunt 类似，Gulp 也是在 Node.js 环境下运行的任务运行器。Gulp 使用 UNIX 风格的管道方式定义任务，每个任务表现为一个 JavaScript 函数。Gulp 也有活跃的社区和丰富的扩展。

D.8　代码检查和格式化

JavaScript 代码调试有一个问题，没有多少 IDE 可以在输入代码时提示代码错误。大多数开发者是写一段代码，然后在浏览器里刷新看看有没有错误。在部署之前验证 JavaScript 代码可以显著减少线上错误。**代码检查器**（linter）可以检查基本的语法并提供关于风格的警告。

格式化器（formatter）是一种工具，可以分析语法规则并实现自动缩进、加空格和对齐代码等操作，也可以自定义完成对文件内容的其他操作。格式化器不会破坏或修改代码或者代码的语义，因为它们可以避免做出影响代码执行的修改。

D.8.1　ESLint

ESLint 是开源的 JavaScript 代码检查器，由本书前几版的作者 Nicholas Zakas 独立开发；完全“可插拔”，以常识化规则作为默认规则，支持配置；有大型可修改和可切换的规则库，可以用来调试工具的行为。

D.8.2　Google Closure Compiler

Google Closure Compiler 内置了一个代码检查工具，可以通过命令行参数激活。这个代码检查器基于代码的抽象语法树工作，因此不会检查空格、缩进或其他不影响代码执行代码组织问题。

D.8.3　JSLint

JSLint 是 Douglas Crockford 开发的 JavaScript 验证器。JSLint 从核心层面检查语法错误，以最大限度保证跨浏览器兼容作为最低要求。（JSLint 遵循最严格的规则以确保代码最大的兼容性。）可以启动 Crockford 关于代码风格的警告，包括代码格式、使用未声明的变量，等等。JSLint 虽然是使用 JavaScript 写的，但可以通过基于 Java 的 Rhino 解释器在命令行执行，也可以通过 WScript 或其他 JavaScript 解释器执行。它的网站提供了针对每个命令行解释器的自定义版。

D.8.4　JSHint

JSHint 是 JSLint 的分支，支持对检查规则更宽泛的自定义。与 JSLint 类似，JSHint 也先检查语法错误，然后再检查有问题的代码模式。JSLint 的每项检查 JSHint 中也都有，但开发者可以更好地控制应用哪些规则。同样与 JSLint 类似，JSHint 可以使用 Rhino 在命令行中执行。

D.8.5　ClangFormat

ClangFormat 是构建在 Clang 项目的 LibFormat 库基础上的格式化工具。它使用了 Clang 格式化规则自动重新组织代码（不会改变语义结构）。ClangFormat 可以在命令行中使用，也可以集成到编辑器里。

D.9　压缩工具

JavaScript 构建过程的一个重要环节就是压缩输出，剔除多余字符。这样可以保证只将最少的字节量传输到浏览器进行解析，用户体验会更好。有不少**压缩工具**，它们的压缩率有所不同。

D.9.1　Uglify

Uglify 是可以压缩、美化和最小化 JavaScript 代码的工具包。它可以在命令行运行，可以接收极为丰富的配置选项，实现满足需求的自定义压缩。

D.9.2　Google Closure Compiler

虽然严格来讲并不是压缩工具，但 Google Closure Compiler 也在其优化工具中提供了不同级别的优化，能够缩小代码体积。

D.9.3　JSMin

JSMin 是 Douglas Crockford 用 C 语言写的一个代码压缩程序，能对 JavaScript 进行基本的压缩。它主要用于删除空格和注释，确保结果可以正确运行。JSMin 也提供了 Window 可执行文件，有 C 语言和其他语言的源代码。

D.10　单元测试

大多数 JavaScript 库会使用某种形式的单元测试来测试自己的代码，有的还会将自己的单元测试框架公之于众，供他人使用。**测试驱动开发**（TDD，Test Driven Development）是以单元测试为中心的软件开发过程。

D.10.1　Mocha

Mocha 是目前非常流行的单元测试框架，为开发单元测试提供了优秀的配置能力和可扩展性。Mocha 的测试非常灵活，顺序执行可以保证生成准确的报告且更容易调试。

D.10.2　Jest

Jest 的宗旨是快速且可靠。Jest 具备并行测试执行和智能测试监视等功能，提供了一系列内置的匹配器和断言，使编写易于阅读和理解的测试变得简单。此外，它支持对函数和模块进行模拟和监听操作，方便隔离和测试大型应用的各个组件。

D.10.3　Jasmine

Jasmine 虽然是比较老的单元测试框架，但仍非常流行。它内置了单元测试所需的一切，没有外部依赖，而且语法简单易读。

D.10.4　qUnit

qUnit 是为 jQuery 设计的单元测试框架。事实上，jQuery 本身在所有测试中都使用 qUnit。除此之外，qUnit 对 jQuery 没有依赖，可用于测试任何 JavaScript 代码。qUnit 非常简单，容易上手。

D.10.5　JsUnit

JsUnit 是早期的 JavaScript 单元测试库，不依赖任何 JavaScript 库。JsUnit 是流行的 Java 测试框架 JUnit 的端口。测试在页面中运行，可以设置为自动测试并将结果提交给服务器。JsUnit 的网站上包含示例和文档。

D.11　文档生成器

大多数 IDE 包含主语言的文档生成器。因为 JavaScript 没有官方 IDE，所以过去文档要么手动生成，要么借用其他语言的文档生成器生成。不过，目前已出现了一些面向 JavaScript 的文档生成器。

D.11.1　ESDoc

ESDoc 能够为 JavaScript 代码生成非常高级的文档页面，包括从文档页面链接到源代码的功能。ESDoc 还有一个插件库可以扩展其功能。不过，ESDoc 要求代码必须使用 ES6 模块。

D.11.2　documentation.js

documentation.js 可以处理代码中的 JSDoc 注释，自动生成 HTML、Markdown 或 JSON 格式的文档。它兼容最新版本的 ECMAScript 和所有主流构建工具，也支持 Flow 的注解。

D.11.3　Docco

按照其网站的描述，Docco 是“简单快捷”的文档生成器。这个工具的理念是以简单的方式生成描述代码的 HTML 页面。Docco 在某些情况下会出问题，但它确实是生成代码文档的极简方法。

D.11.4　JsDoc Toolkit

JsDoc Toolkit 是早期的 JavaScript 文档生成器。它要求代码中包含 Javadoc 风格的注释，然后可以基于这些注释生成 HTML 文件。可以使用预置的 JsDoc 模板或自己创建的模板来自定义生成的 HTML 页面格式。JsDoc Toolkit 是个 Java 包。