附录A

ES2018 和 ES2019

从 ECMAScript 2015 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。

ECMAScript 2018 于 2018 年 1 月完成，主要增加了异步迭代、剩余和扩展操作符、正则表达式和期约等方面的特性。可以通过 TC39 委员会的 GitHub 仓库了解规范的最新动态。

> **注意** 本附录中介绍的特性相对较新，往往只有新近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 异步迭代

在 ECMAScript 最近发布的几个版本中，异步执行和迭代器协议是两个极其热门的主题。异步执行用于释放对执行线程的控制以执行慢操作和收回控制，而迭代器协议则涉及为任意对象定义规范顺序。异步迭代只是这两个概念在逻辑上的统一。

同步迭代器在每次调用 `next()` 时都会返回 `{ value, done }` 对象。当然，这要求确定这个对象内容的计算和资源获取在 `next()` 调用退出时必须完成，否则这些值就无法确定。在使用**同步**迭代器迭代**异步**确定的值时，主执行线程会被阻塞，以等待异步操作完成。

有了异步迭代器，这个问题就迎刃而解了。异步迭代器在每次调用 `next()` 时会提供解决为 `{ value, done }` 对象的期约。这样，执行线程可以释放并在当前这步循环完成之前执行其他任务。

A.1.1 创建并使用异步迭代器

要理解异步迭代器，最简单的办法是用它跟同步迭代器进行比较。下面代码中创建了一个简单的 `Emitter` 类，该类包含一个同步生成器函数，该函数会产生一个同步迭代器，同步迭代器输出 0~4：

```
class Emitter {
  constructor(max) {
    this.max = max;
    this.syncIdx = 0;
  }

  *[Symbol.iterator]() {
    while(this.syncIdx < this.max) {
      yield this.syncIdx++;
```

```
    }
  }
}

const emitter = new Emitter(5);

function syncCount() {
  const syncCounter = emitter[Symbol.iterator]();

  for (const x of syncCounter) {
    console.log(x);
  }
}

syncCount();
// 0
// 1
// 2
// 3
// 4
```

这个例子之所以可以运行起来，主要是因为迭代器可以立即产生下一个值。假如你不想在确定下一个产生的值时阻塞主线程执行，也可以定义异步迭代器函数，让它产生期约包装的值。

为此，要使用迭代器和生成器的异步版本。ECMAScript 2018 为此定义了 `Symbol.asyncIterator`，以便定义和调用输出期约的生成器函数。同时，这一版规范还为异步迭代器增加了 `for-await-of` 循环，用于使用异步迭代器。

相应地，前面的例子可以扩展为同时支持同步和异步迭代：

```
class Emitter {
  constructor(max) {
    this.max = max;
    this.syncIdx = 0;
    this.asyncIdx = 0;
  }

  *[Symbol.iterator]() {
    while(this.syncIdx < this.max) {
      yield this.syncIdx++;
    }
  }

  async *[Symbol.asyncIterator]() {
  // *[Symbol.asyncIterator]() {
    while(this.asyncIdx < this.max) {
      // yield new Promise((resolve) => resolve(this.asyncIdx++));
    yield this.asyncIdx++
    }
  }
}

const emitter = new Emitter(5);

function syncCount() {
  const syncCounter = emitter[Symbol.iterator]();

  for (const x of syncCounter) {
    console.log(x);
```

```
  }
}

async function asyncCount() {
  const asyncCounter = emitter[Symbol.asyncIterator]();

  for await(const x of asyncCounter) {
    console.log(x);
  }
}

syncCount();
// 0
// 1
// 2
// 3
// 4

asyncCount();
// 0
// 1
// 2
// 3
// 4
```

为了加深理解，可以把前面例子中的**同步**生成器传给 `for-await-of` 循环：

```
const emitter = new Emitter(5);

async function asyncIteratorSyncCount() {
  const syncCounter = emitter[Symbol.iterator]();

  for await(const x of syncCounter) {
    console.log(x);
  }
}

asyncIteratorSyncCount();
// 0
// 1
// 2
// 3
// 4
```

虽然这里迭代的是同步生成器产生的原始值，但 `for-await-of` 循环仍像它们被包装在期约中一样处理它们。这说明 `for-await-of` 循环可以流畅地处理同步和异步可迭代对象。但是常规 `for` 循环就不能处理异步迭代器了：

```
function syncIteratorAsyncCount() {
  const asyncCounter = emitter[Symbol.asyncIterator]();

  for (const x of asyncCounter) {
    console.log(x);
  }
}

syncIteratorAsyncCount();
// TypeError: asyncCounter is not iterable
```

关于异步迭代器，要理解的非常重要的一个概念是 Symbol.asyncIterator 符号不会改变生成器函数的行为或者消费生成器的方式。注意在前面的例子中，生成器函数加上了 async 修饰符成为异步函数，又加上了星号成为生成器函数。Symbol.asyncIterator 在这里只起一个提示的作用，告诉将来消费这个迭代器的外部结构如 for-await-of 循环，这个迭代器会返回期约对象的序列。

A.1.2　理解异步迭代器队列

当然，前面的例子是假想的，因为迭代器返回的期约都会立即解决，所以跟同步迭代器的区别很难看出来。想象一下迭代器返回的期约会在不确定的时间解决，而且它们返回的顺序是乱的。异步迭代器应该尽可能模拟同步迭代器，包括每次迭代时代码的按顺序执行。为此，异步迭代器会维护一个回调队列，以保证早期值的迭代器处理程序总是会在处理晚期值之前完成，即使后面的值早于之前的值解决。

为验证这一点，下面的例子中的异步迭代器以随机时长返回期约。异步迭代队列可以保证期约解决的顺序不会干扰迭代顺序。结果应该按顺序打印一组整数（但间隔时间随机）：

```
class Emitter {
  constructor(max) {
    this.max = max;
    this.syncIdx = 0;
    this.asyncIdx = 0;
  }

  *[Symbol.iterator]() {
    while(this.syncIdx < this.max) {
      yield this.syncIdx++;
    }
  }
  async *[Symbol.asyncIterator]() {
    while(this.asyncIdx < this.max) {
      yield new Promise((resolve) => {
        setTimeout(() => {
          resolve(this.asyncIdx++);
        }, Math.floor(Math.random() * 1000));
      });
    }
  }
}

const emitter = new Emitter(5);

function syncCount() {
  const syncCounter = emitter[Symbol.iterator]();

  for (const x of syncCounter) {
    console.log(x);
  }
}

async function asyncCount() {
  const asyncCounter = emitter[Symbol.asyncIterator]();

  for await(const x of asyncCounter) {
    console.log(x);
  }
```

```
}

syncCount();
// 0
// 1
// 2
// 3
// 4

asyncCount();
// 0
// 1
// 2
// 3
// 4
```

A.1.3　处理异步迭代器的 reject()

因为异步迭代器使用期约来包装返回值，所以必须考虑某个期约被拒绝的情况。由于异步迭代会按顺序完成，而在循环中跳过被拒绝的期间是不合理的。因此，被拒绝的期约会强制退出迭代器：

```
class Emitter {
  constructor(max) {
    this.max = max;
    this.asyncIdx = 0;
  }

  async *[Symbol.asyncIterator]() {
    while (this.asyncIdx < this.max) {
      if (this.asyncIdx < 3) {
        yield this.asyncIdx++;
      } else {
        throw 'Exited loop';
      }
    }
  }
}

const emitter = new Emitter(5);

async function asyncCount() {
  const asyncCounter = emitter[Symbol.asyncIterator]();

  for await (const x of asyncCounter) {
    console.log(x);
  }
}

asyncCount();
// 0
// 1
// 2
// Uncaught (in promise) Exited loop
```

A.1.4　使用 next()手动异步迭代

for-await-of 循环提供了两个有用的特性：一是利用异步迭代器队列保证按顺序执行，二是隐藏异步迭代器的期约。不过，使用这个循环会隐藏很多底层行为。

因为异步迭代器仍遵守迭代器协议，所以可以使用 next()逐个遍历异步可迭代对象。如前所述，next()返回的值会包含一个期约，该期约可解决为{ value, done }这样的迭代结果。这意味着必须使用期约 API 获取方法，同时也意味着可以不使用异步迭代器队列。

```
const emitter = new Emitter(5);

const asyncCounter = emitter[Symbol.asyncIterator]();

console.log(asyncCounter.next());
// Promise<{value, done}>
```

A.1.5　顶级异步循环

一般来说，包括 for-await-of 循环在内的异步行为不能出现在异步函数外部。不过，有时候可能确实需要在这样的上下文使用异步行为。为此可以通过创建异步 IIFE 来达到目的：

```
class Emitter {
  constructor(max) {
    this.max = max;
    this.asyncIdx = 0;
  }

  async *[Symbol.asyncIterator]() {
    while(this.asyncIdx < this.max) {
      yield new Promise((resolve) => resolve(this.asyncIdx++));
    }
  }
}

const emitter = new Emitter(5);

(async function() {
  const asyncCounter = emitter[Symbol.asyncIterator]();

  for await(const x of asyncCounter) {
    console.log(x);
  }
})();
// 0
// 1
// 2
// 3
// 4
```

A.1.6　实现可观察对象

异步迭代器可以耐心等待下一次迭代而不会导致计算成本，那么这也为实现可观察对象（Observable）接口提供了可能。总体上看，这涉及捕获事件，将它们封装在期约中，然后把这些事件提供给迭代器，

而处理程序可以利用这些异步迭代器。在某个事件触发时，异步迭代器的下一个期约会解决为该事件。

> **注意**　可观察对象的话题超出了本书范围，因为它们很大程度上是作为第三方库实现的。有兴趣的读者可以了解一下非常流行的 RxJS 库。

下面这个简单的例子会捕获浏览器事件的可观察流。这需要一个期约的队列，每个期约对应一个事件。该队列也会保持事件生成的顺序，对这种问题来说保持顺序也是合理的。

```
class Observable {
  constructor() {
    this.promiseQueue = [];

    // 保存用于解决队列中下一个期约的程序
    this.resolve = null;

    // 把最初的期约推到队列
    // 该期约会解决为第一个观察到的事件
    this.enqueue();
  }

  // 创建新期约，保存其解决方法
  // 再把它保存到队列中
  enqueue() {
    this.promiseQueue.push(
      new Promise((resolve) => this.resolve = resolve));
  }

  // 从队列前端移除期约
  // 并返回它
  dequeue() {
    return this.promiseQueue.shift();
  }
}
```

要利用这个期约队列，可以在这个类上定义一个异步生成器方法。该生成器可用于任何类型的事件：

```
class Observable {
  constructor() {
    this.promiseQueue = [];

    // 保存用于解决队列中下一个期约的程序
    this.resolve = null;

    // 把最初的期约推到队列
    // 该期约会解决为第一个观察到的事件
    this.enqueue();
  }

  // 创建新期约，保存其解决方法
  // 再把它保存到队列中
  enqueue() {
    this.promiseQueue.push(
      new Promise((resolve) => this.resolve = resolve));
  }

  // 从队列前端移除期约
```

```
    // 并返回它
    dequeue() {
      return this.promiseQueue.shift();
    }

    async *fromEvent (element, eventType) {
      // 在有事件生成时，用事件对象来解决队列头部的期约
      // 同时把另一个期约加入队列
      element.addEventListener(eventType, (event) => {
        this.resolve(event);
        this.enqueue();
      });

      // 每次解决队列前面的期约
      // 都会向异步迭代器返回相应的事件对象
      while (1) {
        yield await this.dequeue();
      }
    }
  }
```

这样，这个类就定义完了。接下来在 DOM 元素上定义可观察对象就很简单了。假设页面上有一个<button>元素，可以像下面这样捕获该按钮上的一系列 click 事件，然后在控制台把它们打印出来：

```
  class Observable {
    constructor() {
      this.promiseQueue = [];

      // 保存用于解决队列中下一个期约的程序
      this.resolve = null;

      // 把最初的期约推到队列
      // 该期约会解决为第一个观察到的事件
      this.enqueue();
    }

    // 创建新期约，保存其解决方法
    // 再把它保存到队列中
    enqueue() {
      this.promiseQueue.push(
        new Promise((resolve) => this.resolve = resolve));
    }

    // 从队列前端移除期约
    // 并返回它
    dequeue() {
      return this.promiseQueue.shift();
    }

    async *fromEvent (element, eventType) {
      // 在有事件生成时，用事件对象来解决 队列头部的期约
      // 同时把另一个期约加入队列
      element.addEventListener(eventType, (event) => {
        this.resolve(event);
        this.enqueue();
      });

      // 每次解决队列前面的期约
      // 都会向异步迭代器返回相应的事件对象
```

```
    while (1) {
      yield await this.dequeue();
    }
  }
}

(async function() {
  const observable = new Observable();

  const button = document.querySelector('button');
  const mouseClickIterator = observable.fromEvent(button, 'click');

  for await (const clickEvent of mouseClickIterator) {
    console.log(clickEvent);
  }
})();
```

A.2　对象字面量的剩余操作符和扩展操作符

ECMAScript 2018 将数组中的剩余操作符和扩展操作符也移植到了对象字面量。这极大地方便了对象合并和通过其他对象创建新对象。

A.2.1　剩余操作符

剩余操作符可以在解构对象时将所有剩下未指定的可枚举属性收集到一个对象中。比如：

```
const person = { name: 'Matt', age: 27, job: 'Engineer' };
const { name, ...remainingData } = person;

console.log(name); // Matt
console.log(remainingData); // { age: 27, job: 'Engineer' }
```

每个对象字面量中最多可以使用一次剩余操作符，而且必须放在最后。因为每个对象字面量只能有一个剩余操作符，所以可以嵌套剩余操作符。嵌套时，因为子属性对应的剩余操作符没有歧义，所以返回对象的内容不会重叠：

```
const person = { name: 'Matt', age: 27, job: { title: 'Engineer', level: 10 } };

const { name, job: { title, ...remainingJobData }, ...remainingPersonData } = person;

console.log(name);                  // Matt
console.log(title);                 // Engineer
console.log(remainingPersonData);   // { age: 27 }
console.log(remainingJobData);      // { level: 10 }

const { ...a, job } = person;
// SyntaxError: Rest element must be last element
```

剩余操作符在对象间执行浅复制，因此会复制对象的引用而不会克隆整个对象：

```
const person = { name: 'Matt', age: 27, job: { title: 'Engineer', level: 10 } };

const { ...remainingData } = person;

console.log(person === remainingData);          // false
console.log(person.job === remainingData.job);  // true
```

剩余操作符会复制所有自有可枚举属性，包括符号：

```
const s = Symbol();
const foo = { a: 1, [s]: 2, b: 3 }

const {a, ...remainingData} = foo;

console.log(remainingData);
// { b: 3, Symbol(): 2 }
```

A.2.2　扩展操作符

扩展操作符可以像拼接数组一样合并两个对象。应用到内部对象的扩展操作符会对所有自有可枚举属性执行浅复制到外部对象，包括符号：

```
const s = Symbol();
const foo = { a: 1 };
const bar = { [s]: 2 };

const foobar = {...foo, c: 3, ...bar};

console.log(foobar);
// { a: 1, c: 3 Symbol(): 2 }
```

扩展对象的顺序很重要，主要有两个原因。

(1) 对象跟踪插入顺序。从扩展对象复制的属性按照它们在对象字面量中列出的顺序插入。

(2) 对象会覆盖重名属性。在出现重名属性时，会使用后出现属性的值。

下面的例子演示了上述约定：

```
const foo = { a: 1 };
const bar = { b: 2 };

const foobar = {c: 3, ...bar, ...foo};

console.log(foobar);
// { c: 3, b: 2, a: 1 }

const baz = { c: 4 };

const foobarbaz = {...foo, ...bar, c: 3, ...baz};

console.log(foobarbaz);
// { a: 1, b: 2, c: 4 }
```

与剩余操作符一样，所有复制都是浅复制：

```
const foo = { a: 1 };
const bar = { b: 2, c: { d: 3 } };

const foobar = {...foo, ...bar};

console.log(foobar.c === bar.c); // true
```

A.3　`Promise.prototype.finally()`

以前，只能使用不太好看的方式处理期约退出“待定”状态，而不管后续状态如何。换句话说，通

常需要重复利用处理程序：

```
let resolveA, rejectB;

function finalHandler() {
  console.log('finished');
}

function resolveHandler(val) {
  console.log('resolved');
  finalHandler();
}

function rejectHandler(err) {
  console.log('rejected');
  finalHandler();
}

new Promise((resolve, reject) => {
  resolveA = resolve;
})
.then(resolveHandler, rejectHandler);

new Promise((resolve, reject) => {
  rejectB = reject;
})
.then(resolveHandler, rejectHandler);

resolveA();
rejectB();
// resolved
// finished
// rejected
// finished
```

有了 `Promise.prototype.finally()`，就可以统一共享的处理程序。`finally()`处理程序不传递任何参数，也不知道自己处理的期约是解决的还是拒绝的。前面的代码可以重构为如下形式：

```
let resolveA, rejectB;

function finalHandler() {
  console.log('finished');
}

function resolveHandler(val) {
  console.log('resolved');
}

function rejectHandler(err) {
  console.log('rejected');
}

new Promise((resolve, reject) => {
  resolveA = resolve;
})
.then(resolveHandler, rejectHandler)
.finally(finalHandler);
```

```
new Promise((resolve, reject) => {
  rejectB = reject;
})
.then(resolveHandler, rejectHandler)
.finally(finalHandler);

resolveA();
rejectB();
// resolved
// rejected
// finished
// finished
```

注意日志的顺序不一样了。每个 `finally()`都会创建一个新期约实例，而这个新期约会添加到浏览器的微任务队列，只有前面的处理程序执行完成才会解决。

A.4　正则表达式相关特性

ECMAScript 2018 为正则表达式增加了一些特性。

A.4.1　dotAll 标志

正则表达式中用于匹配任意字符的点（.）不匹配换行符，比如`\n` 和`\r` 或非 BMP 字符，如表情符号。

```
const text = `
foo
bar
`;

const re = /foo.bar/;

console.log(re.test(text)); // false
```

为此，规范新增了 `s` 标志（意思是单行，singleline），从而解决了这个问题：

```
const text = `
foo
bar
`;

const re = /foo.bar/s;

console.log(re.test(text)); // true
```

A.4.2　向后查找断言

正则表达式支持肯定式向前查找断言和否定式向前查找断言，可以匹配后跟指定字符串的表达式：

```
const text = 'foobar';

// 肯定式向前查找
// 断言后跟某个值，但不捕获该值
const rePositiveMatch = /foo(?=bar)/;
const rePositiveNoMatch = /foo(?=baz)/;
```

```
console.log(rePositiveMatch.exec(text));
// ["foo"]

console.log(rePositiveNoMatch.exec(text));
// null

// 否定式向前查找
// 断言后面不是某个值，但不捕获该值
const reNegativeNoMatch = /foo(?!bar)/;
const reNegativeMatch = /foo(?!baz)/;

console.log(reNegativeNoMatch.exec(text));
// null

console.log(reNegativeMatch.exec(text));
// ["foo"]
```

规范相应地增加了与这些断言对应的肯定式向后查找和否定式向后查找。向后查找与向前查找的工作原理类似，只是会检测要捕获内容之前的内容。

```
const text = 'foobar';

// 肯定式向后查找
// 断言前面是某个值，但不捕获该值
const rePositiveMatch = /(?<=foo)bar/;
const rePositiveNoMatch = /(?<=baz)bar/;

console.log(rePositiveMatch.exec(text));
// ["bar"]

console.log(rePositiveNoMatch.exec(text));
// null

// 否定式向后查找
// 断言前面不是某个值，但不捕获该值
const reNegativeNoMatch = /(?<!foo)bar/;
const reNegativeMatch = /(?<!baz)bar/;

console.log(reNegativeNoMatch.exec(text));
// null

console.log(reNegativeMatch.exec(text));
// ["bar"]
```

A.4.3　命名捕获组

多个捕获组通常是按索引来取值的，但索引没有上下文，反映不出它们包含的是什么内容：

```
const text = '2018-03-14';

const re = /(\d+)-(\d+)-(\d+)/;

console.log(re.exec(text));
// ["2018-03-14", "2018", "03", "14"]
```

为此，规范支持将捕获组与有效 JavaScript 标识符关联，这样就可以通过标识符获取捕获组的内容：

```
const text = '2018-03-14';

const re = /(?<year>\d+)-(?<month>\d+)-(?<day>\d+)/;

console.log(re.exec(text).groups);
// { year: "2018", month: "03", day: "14" }
```

A.4.4　Unicode 属性转义

Unicode 标准为每个字符都定义了属性。字符属性，涉及字符名称、类别、空格指示和字符内部定义的脚本或语言等。通过使用 Unicode 属性转义可以在正则表达式中使用这些属性。

有些属性是二进制，意味可以独立使用。比如 `Uppercase` 和 `White_Space`。有些属性是键/值对，即一个属性对应一个属性值。比如 `Script_Extensions=Greek`。

Unicode 属性列表及属性值列表参见 Unicode 网站。

Unicode 属性转义在正则表达式中可使用`\p` 表示匹配，使用`\P` 表示不匹配：

```
const pi = String.fromCharCode(0x03C0);
const linereturn = `
`;

const reWhiteSpace = /\p{White_Space}/u;
const reGreek = /\p{Script_Extensions=Greek}/u;
const reNotWhiteSpace = /\P{White_Space}/u;
const reNotGreek = /\P{Script_Extensions=Greek}/u;

console.log(reWhiteSpace.test(pi));              // false
console.log(reWhiteSpace.test(linereturn));      // true
console.log(reNotWhiteSpace.test(pi));           // true
console.log(reNotWhiteSpace.test(linereturn));   // false

console.log(reGreek.test(pi));                   // true
console.log(reGreek.test(linereturn));           // false
console.log(reNotGreek.test(pi));                // false
console.log(reNotGreek.test(linereturn));        // true
```

A.5　数组打平方法

ECMAScript 2019 在 `Array.prototype` 上增加了两个方法：`flat()`和 `flatMap()`。这两个方法为打平数组提供了便利。如果没有这两个方法，则打平数组就要使用迭代或递归。

> **注意**　`flat()`和 `flatMap()`只能用于打平嵌套数组。嵌套的可迭代对象如 `Map` 和 `Set` 不能打平。

A.5.1　`Array.prototype.flatten()`

下面是如果没有这两个新方法要打平数组的一个示例实现：

```
function flatten(sourceArray, flattenedArray = []) {
  for (const element of sourceArray) {
    if (Array.isArray(element)) {
```

```
      flatten(element, flattenedArray);
    } else {
      flattenedArray.push(element);
    }
  }
  return flattenedArray;
}

const arr = [[0], 1, 2, [3, [4, 5]], 6];

console.log(flatten(arr))
// [0, 1, 2, 3, 4, 5, 6]
```

这个例子在很多方面像一个树形数据结构：数组中每个元素都像一个子节点，非数组元素是叶节点。因此，这个例子中的输入数组是一个高度为 2 有 7 个叶节点的树。打平这个数组本质上是对叶节点的按序遍历。

有时候如果能指定打平到第几级嵌套是很有用的。比如下面这个例子，它重写了上面的版本，允许指定要打平几级：

```
function flatten(sourceArray, depth, flattenedArray = []) {
  for (const element of sourceArray) {
    if (Array.isArray(element) && depth > 0) {
      flatten(element, depth - 1, flattenedArray);
    } else {
      flattenedArray.push(element);
    }
  }

  return flattenedArray;
}


const arr = [[0], 1, 2, [3, [4, 5]], 6];

console.log(flatten(arr, 1));
// [0, 1, 2, 3, [4, 5], 6]
```

为了解决上述问题，规范增加了 `Array.prototype.flat()`方法。该方法接收 `depth` 参数（默认值为 1），返回一个对要打平 `Array` 实例的浅复制副本。下面看几个例子：

```
const arr = [[0], 1, 2, [3, [4, 5]], 6];

console.log(arr.flat(2));
// [0, 1, 2, 3, 4, 5, 6]

console.log(arr.flat());
// [0, 1, 2, 3, [4, 5], 6]
```

因为是执行浅复制，所以包含循环引用的数组在被打平时会从源数组复制值：

```
const arr = [[0], 1, 2, [3, [4, 5]], 6];

arr.push(arr);

console.log(arr.flat());
// [0, 1, 2, 3, 4, 5, 6, [0], 1, 2, [3, [4, 5]], 6]
```

A.5.2 Array.prototype.flatMap()

Array.prototype.flatMap()方法会在打平数组之前执行一次映射操作。在功能上，arr.flatMap(f)与 arr.map(f).flat()等价；但 arr.flatMap()更高效，因为浏览器只需要执行一次遍历。

flatMap()的函数签名与 map()相同。下面是一个简单的例子：

```
const arr = [[1], [3], [5]];

console.log(arr.map(([x]) => [x, x + 1]));
// [[1, 2], [3, 4], [5, 6]]

console.log(arr.flatMap(([x]) => [x, x + 1]));
// [1, 2, 3, 4, 5, 6]
```

flatMap()在非数组对象的方法返回数组时特别有用，例如字符串的 split()方法。来看下面的例子，该例子把一组输入字符串分割为单词，然后把这些单词拼接为一个单词数组：

```
const arr = ['Lorem ipsum dolor sit amet,', 'consectetur adipiscing elit.'];

console.log(arr.flatMap((x) => x.split(/[\W+]/)));
// ["Lorem", "ipsum", "dolor", "sit", "amet", "", "consectetur", "adipiscing",
"elit", ""]
```

对于上面的例子，可以利用空数组进一步过滤上一次映射后的结果，这也是一个数据处理技巧（虽然可能会有些性能损失）。下面的例子扩展了上面的例子，去掉了空字符串:

```
const arr = ['Lorem ipsum dolor sit amet,', 'consectetur adipiscing elit.'];

console.log(arr.flatMap((x) => x.split(/[\W+]/)).flatMap((x) => x || []));
// ["Lorem", "ipsum", "dolor", "sit", "amet", consectetur", "adipiscing", "elit"]
```

这里，结果中的每个空字符串首先映射到一个空数组。在打平时，这些空数组就会因为没有内容而被忽略。

> **注意**　不建议使用这个技巧。这是因为过滤每个值都要构建一个立即丢弃的 Array 实例。

A.6 Object.fromEntries()

ECMAScript 2019 又给 Object 类添加了一个静态方法 fromEntries()，用于通过键/值对数组的集合构建对象。这个方法执行与 Object.entries()方法相反的操作。来看下面的例子：

```
const obj = {
  foo: 'bar',
  baz: 'qux'
};

const objEntries = Object.entries(obj);

console.log(objEntries);
// [["foo", "bar"], ["baz", "qux"]]

console.log(Object.fromEntries(objEntries));
// { foo: "bar", baz: "qux" }
```

此静态方法接收一个可迭代对象参数，该可迭代对象可以包含任意数量的大小为 2 的可迭代对象。这个方法可以方便地将 Map 实例转换为 Object 实例，因为 Map 迭代器返回的结果与 fromEntries() 的参数恰好匹配：

```
const map = new Map().set('foo', 'bar');

console.log(Object.fromEntries(map));
// { foo: "bar" }
```

A.7　字符串修理方法

ECMAScript 2019 向 String.prototype 添加了两个新方法：trimStart() 和 trimEnd()。它们分别用于删除字符串开头和末尾的空格。这两个方法旨在取代之前的 trimLeft()和 trimRight()，因为后两个方法在从右往左书写的语言（如阿拉伯语和希伯来语）中有歧义。

在只有一个空格的情况下，这两个方法相当于执行与 padStart()和 padEnd()相反的操作。下面的例子演示了使用这两个方法删除字符串前后的空格：

```
let s = '   foo   ';

console.log(s.trimStart()); // "foo   "
console.log(s.trimEnd());   // "   foo"
```

A.8　Symbol.prototype.description

ECMAScript 2019 在 Symbol.prototype 上增加了 description 属性，用于取得可选的符号描述。以前，只能通过将符号转型为字符串来取得这个描述：

```
const s = Symbol('foo');

console.log(s.toString());
// Symbol(foo)
```

这个原型属性是只读的，可以在实例上直接取得符号的描述。如果没有描述，则默认为 undefined。

```
const s = Symbol('foo');

console.log(s.description);
// foo
```

A.9　可选的 catch 绑定

在 ECMAScript 2019 之前，try/catch 块的结构相当严格。即使不想使用捕获的错误对象，解析器也要求必须在 catch 子句中把该对象赋值给一个变量：

```
try {
  throw 'foo';
} catch (e) {
  // 发生错误了，但你不想使用错误对象
}
```

在 ECMAScript 2019 中，可以省略这个赋值，并完全忽略错误：

```
try {
  throw 'foo';
} catch {
  // 发生错误了，但你不想使用错误对象
}
```

A.10　其他新增内容

ES2019 还对现有 API 进行了其他一些调整。

- `Array.prototype.sort()`**稳定**了，意味着相同的对象在输出中不会被重新排序。
- 由于单独的 UTF-16 代理对字符不能使用 UTF-8 编码，`JSON.stringify()` 在 ES2019 以前会返回 UTF-16 码元，现在则改为返回转义序列，也是有效的 Unicode。
- ES2019 以前，U+2028 LINE SEPARATOR 和 U+2029 PARAGRAPH SEPARATOR 在 JSON 字符串中都有效，但在 ECMAScript 字符串中则无效。ES2019 实现了 ECMAScript 字符串与 JSON 字符串的兼容。
- ES2019 以前，浏览器厂商可以自由决定 `Function.prototype.toString()` 返回什么。ES2019 要求这个方法尽可能返回函数的源代码，否则返回 `{ [native code] }`。

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。

理解严格模式的规则非常重要，因为未来的 ECMAScript 会逐步强制全局使用严格模式。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，而不支持的引擎则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 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。

D.1.3　JSPM

JSPM 是使用 SystemJS 构建的包管理器，用动态模块加载。这个包管理器本身与 npm 类似，但其包仓库与注册无关。在 npm、GitHub 或自定义仓库中注册包，都可以使用 JSPM 的 CLI 安装。JSPM 不会在服务器上构建和预编译资源，而是通过 SystemJS 按需将包交付给客户端。与 Bower 类似，JSPM 也使用打平的依赖结构。

D.1.4　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 和 npm 的主要区别是提供了加速安装、包缓存、锁文件等功能，且提供了改进了包安全功能。

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。但现在，仍有很多浏览器不支持 ES6 模块加载。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　JSPM

JSPM 是构建在 SystemJS 和 ES6 模块加载器之上的包管理器。JSPM 建议的一个工作流是把所有模

块打包到一个文件，然后通过 SystemJS 加载。可以通过 JSPM CLI 使用这个功能。

D.3.3　Browserify

Browserify 是稍微有点历史但久经考验的模块打包器，支持 Node.js 的 CommonJS `require()`依赖语法。

D.3.4　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　CoffeeScript

CoffeeScript 是 ECMAScript 语法的增强版，可以直接编译为常规 JavaScript。CoffeeScript 中绝大部分是表达式，这是受到了 Ruby、Python 和 Haskell 的启发。

D.4.4　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.4.5　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 系列规范在 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　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.7.2　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.7.3　Brunch

Brunch 也是 Node.js 构建工具，旨在简化配置，方便使用。Brunch 虽然比 Gulp 和 Grunt 出现得晚，但仍有很多插件可以选择。

D.7.4　npm

npm 严格来讲不是构建工具，但它提供了脚本功能，很多项目会利用这个功能融合任务运行器。脚本是在 package.json 中定义的。

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 现在是第 3 版[①]，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.9.4　Dojo ShrinkSafe

Dojo Toolkit 团队开发的 ShrinkSafe 会使用 Rhino 先把 JavaScript 代码解析为符号流，然后再安全地压缩代码。与 JSMin 一样，ShrinkSafe 也会删除多余的空格（但不删除换行）和注释，且会更进一步将局部变量名替换为两个字符的变量名。因此结果比 JSMin 压缩后的更小，不会引入语法错误。

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　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。

---

① 编辑本书时仍是第 3 版。——编者注

D.10.3　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.10.4　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.10.5　Dojo Object Harness

Dojo Object Harness（DOH）最初是 Dojo 内部的单元测试工具，后来开放给所有人使用。与其他框架一样，DOH 的测试也是在浏览器中运行的。

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式来自定义生成的 HTML 页面格式。JsDoc Toolkit 是个 Java 包。

D.11.5　YUI Doc

YUI Doc 是 YUI 的文档生成器。该生成器是用 Python 写的，因此要求安装 Python 运行时。YUI Doc 输出的 HTML 文件中集成了基于 YUI 的自动完成部件的属性和方法搜索功能。与 JsDoc 一样，YUI Doc

要求代码中包含 Javadoc 风格的注释。可以通过修改默认 HTML 模板和关联的样式表来修改默认的 HTML 输出。

D.11.6　AjaxDoc

AjaxDoc 的目标与前面的文档生成器稍有不同。它不会为 JavaScript 代码创建 HTML 文件，而是会创建与.NET 语言（如 C#、Visual Basic）兼容的 XML 格式。这样就可以使用标准.NET 文档生成器来创建 HTML 文档。AjaxDoc 要求所有文档注释的格式与.NET 语言的文档注释格式类似。AjaxDoc 是为 ASP.NET Ajax 解决方案而创建的，但也可以用于独立的项目。