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：

图灵社区会员 aSINKz(1561821892@qq.com) 专享 尊重版权

```
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] }`。

图灵社区会员 aSINKz(1561821892@qq.com) 专享 尊重版权

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