JavaScript 高级程序设计笔记

Pionpill [1]

本文档为作者学习《JavaScript 高级程序设计》[2]一书时的笔记。

2022 年 12 月 16 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

[2]《Professional JavaScript for Web Developers》:Matt Frisbie 2020 年 9 月第 2 版

前言：

笔者为软件工程系在校本科生，主要用 JavaScript 做一些前端项目。本文默认读者有一定的编程基础 (至少会一门 C 语言系语言)，基础语法不再说明。

«JavaScript 高级程序设计 » 是 JavaScript 学习的一本入门及进阶书，原书中文版多达 117 万字，笔者将其分为如下几个部分：

- ECMAScript：介绍 ECMAScript(JavaScript) 的语法。
- TypeScript: 介绍 TypeScript 的语法。
- Web API：与 Web 开发相关的内容，即 BOM 与 DOM(主要是 DOM)。
- Other API：处理各类事务文件的接口。

本笔记不能代替原书，仅是对原书的一个总结归纳，笔记上只有知识点的总结，并没有详细的理解性语句，如有需要，还请购买原书。本文引用了一些 CSDN 或其他论讨的文章，有标注来源，如果原作者觉得不合适，请联系本人。

关于 Web 开发，不得不考虑浏览器兼容问题，令人欣慰的是，微软已经弃用 IE 浏览器，改用 chromium 核心的 Edge 浏览器，本文以 Chrome 浏览器为准，不再考虑任何 IE(或其他浏览器) 适配问题。

本人的编写及开发环境如下：

- IDE: VSCode 1.72
- Chrome: 91.0
- Node.js: 18.12
- OS: Window11

此外，JavaScript 是一门语法十分宽松的语言，同一种实现有非常多的写法，笔记中只强调推荐的写法，更多写法请读者用用现代编辑器代码提示功能查询。

2022 年 12 月 16 日

目录

第一部分

ECMAScript

I 基础语法

1 JavaScript 介绍

1.1 历史回顾

JavaScript 是网景公司率先设计出的，原名 LiveScript，改名是为了蹭 Java 热度。后来微软也曾开发过一款名为 JScript 的脚本语言。这两门语言是现在使用的 JavaScript 的前身。

网景和微软两个公司的 JavaScript 有许多不同之处，这段时间内造成了很多历史遗留问题。后来几个公司打造了 ECMAScript(ES) 这个新的脚本语言标准。JavaScript 才渐渐统一。

JavaScript 和 ECMAScript 基本上是同义词，但实际运用中，JavaScript 包含更宽泛的技术：

- 核心：ECMAScript
- DOM：文档对象模型
- BOM：浏览器对象模型

ECMAScript 规定了语言的语法，包括语句，关键字，全局对象等，与一般的高级语言类似。不同的浏览器支持不同版本的 ECMAScript，感谢微软在 2022 年停止了 IE 浏览器的更新，这解决了绝大部分版本兼容问题。

DOM，即文档对象模型 (Document Object Model) 是一个应用编程接口，用于在 HTML 中使用扩展的 XML。可以将其理解为网页的层次结构，所有的浏览器 DOM 都是相同的。

BOM，即浏览器对象模型 (Browser Object Model)，用于支持访问和操作浏览器的窗口。

JavaScript 不同于一般的高级程序语言，其版本并是由浏览器决定的，所以在使用之前，最好查看你的浏览器所支持的版本[1]。

1.2 HTML 中的 JavaScript

将 JavaScript 插入 HTML 的主要方法是使用 `<script>` 元素，这个元素有 8 个属性，其用法如下：

- `src`: 可选。表示包含要执行的代码的外部文件。
- `async`: 可选。表示应该立即开始下载脚本，但不能阻止其他页面动作。只对外部脚本文件有效。
- `crossorigin`: 可选。配置相关请求的 CORS(跨源资源共享) 设置。默认不使用 CORS。
  - `crossorigin="anonymous"` 配置文件请求不必设置凭据标志。

[1]现代浏览器 (Chrome，FireFox，Edge) 都对 JS 有良好的支持，无需关心版本问题。但请不要用 IE 浏览器，国产浏览器做开发。

– crossorigin="use-credentials" 设置凭据表指，意味着出站请求会包含凭据。

- defer: 可选。表示脚本可以延迟到文档全部被解析和显示之后再执行。只对外部脚本文件有效。
- integrity: 可选。允许对比接收到的资源和指定的加密签名以验证自资源完整性。
- charset: 可选。使用 src 属性指定的代码字符集。这个属性很少用。
- type: 可选，代替 language，表示代码块中脚本语言的内容类型，不同浏览器可能有不同值，一般为”text/javascript"。
- language: 废弃。历史遗留问题，不再介绍。

行内 JavaScript 代码

嵌入式 JavaScript 直接将代码放在 <script> 元素中就行：

```
<script>
    function sayHi() {
        console.log("Hi!");
    }
</script>
```

包含在 <script> 内的代码会被从上到下解释。在 <script> 元素中的代码被计算完成之前，浏览器的其余内容不会被加载，也不会被显示。

在使用行内 JavaScript 代码时，注意代码中不能出现字符串 <\script>，因为这将被识别为脚本结束标签。如果一定要使用，需要转义字符 "\"。

```
<script>
    function sayHi() {
        console.log("<\/script>");
    }
</script>
```

外部 JavaScript 代码

要包含外部文件中的 JavaScript，就必须使用 src 属性。这个属性值是一个 URL，指向包含 JavaScript 代码的文件。

```
<script src = "example.js"></script>
```

与解释行内 JavaScript 一样，在解释外部 JavaScript 文件时，页面也会阻塞 (包含下载时间)。

> **注意 1.1.** *按照惯例，外部 JavaScript 文件的扩展名是.js。但这不是必需的，因为浏览器不会检查扩展名。这就为使用服务器端脚本语言动态生成 JavaScript 代码，或在浏览器中将 JavaScript 扩展语言 (如 TypeScript) 转译为 JavaScript 提供了可能性。*

使用了 src 属性的 <script> 元素不应在标签中再包含其他 JavaScript 代码。如果有，浏

览器会直接忽略行内代码。

此外，src 属性可以包含来自外部域的 JavaScript 文件。

```
<script src="http://somewhere//example.js"></script>
```

使用外部域的脚本文件时，会产生许多安全问题，除非对齐安全性有绝对的信息，否则不要随意使用。

> **建议 1.1.** 在实际运用中，一般都会选择外部文件来书写代码，这在可维护性，缓存，兼容等方面都要优于行内代码。

标签位置

过去，所有的 <script> 元素都被放在页面的 <head> 标签内，这样做会导致许多问题。脚本文件必须从上到下一个个被执行后，页面才能开始加载。这会导致页面渲染明显延迟，现代的许多脚本文件都被放在了 <body> 标签中。

执行脚本

HTML 提供了两个属性用于推迟或异步执行脚本：

defer 属性：推迟执行

这个属性表示脚本在执行的时候不会改变页面的结构。也就是说，脚本会被延迟到整个页面解析完毕后再执行。在 <script> 元素上设置 defer 属性，相当于告诉浏览器立即下载，但延迟执行。

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title> Example </title>
    <script defer src="example1.js"> </script>
    <script defer src="example2.js"> </script>
  </head>
  <body></body>
</html>
```

虽然上述例子中的 <script> 元素包含在页面的 <head> 中，但它们会在浏览器解析到结束时才执行。

> **建议 1.2.** 由于可能会产生顺序错误，一般建议只包含一个含 *defer* 属性的外部脚本。

async 属性：异步执行

从改变脚本处理的方式上来看，async 属性与 defer 类似。不过 async 属性标记的脚本并不保证能按照它们出现的顺序来执行。

给脚本添加 async 属性的目的时告诉浏览器，不必等脚本下载和执行完后再加载页面，同样也不必等到该一部脚本下载和执行后再加载其他脚本。正因为如此，异步脚本不应该在

加载期间修改 DOM。

> **建议 1.3.** 虽然存在这个属性，但一般不推荐使用。

动态加载脚本

除了 `<script>` 标签，还有其他方式加载脚本。因为 JavaScript 可以使用 DOM API，所以通过向 DOM 中动态添加 `script` 元素同样可以加载指定的脚本。只要创建一个 `script` 元素并将其添加到 DOM 即可。

```
let script = document.createElement('script');
script.src = 'gibberish.js';
document.head.appendChild(script);
```

默认情况下，以这种方式创建的 `<script>` 元素是以异步方式加载的，相当于添加了 `async` 属性。不过这样做可能会有问题。开发者可以选择关闭 `async` 属性。

```
script.async = false;
```

以这种方式获取的资源对六拉你去预加载器是不可见的。这会严重影响它们在资源获取队列中的优先级。这可能会严重影响性能。要想让预加载器直到这些动态请求文件的存在，可以在文档头部显示声明它们。

```
<link rel="preload" href="gibberish.js">
```

`<noscript>` 元素

`<noscript>` 元素主要用于解决早期浏览器不支持 JavaScript 的问题，在以下两种情况下，将显示 <noscript> 中的内容:

- 浏览器不支持脚本。
- 浏览器对脚本的支持被关闭。

若是不满足，则不会渲染对应内容。下面是一个使用例子：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <script defer="defer" src="exmaple1.js"></script>
    <script defer="defer" src="exmaple2.js"></script>
    <title>Document</title>
  </head>
  <body>
    <noscript>
      <p>This page needs a JavaScript-enabled browser.</p>
    </noscript>
  </body>
</html>
```

2 语言基础

2.1 语法

ECMAScript 语言很大程度上借鉴了 C 语言，作为 C 语言系中的一门，它和 Java 与许多相似之处，比如区分大小写。

标识符

JavaScript 对标识符组成规定如下：

- 第一个字符必须是一个字母，下划线 (_) 或美元符号 ($)。
- 剩下的字符可以是字母，数字，下划线 (_) 或美元符号 ($)。

标识符中的字符是可扩展的 ASCII 中的字母，也可以是 Unicode 的字母字符，但不建议使用 Unicode 字符。

> **建议 2.1.** *下划线 (_) 和美元符号 ($) 往往有特殊作用，不建议在自己命名的变量中使用。*

ECMAScript 标识符使用驼峰大小写形式。

注释

ECMAScript 采用 C 语言风格的注释。

- 单行注释:

```
// 当行注释
```

- 多行注释 (块注释):

```
/*多行
注释*/
```

严格模式

ES5 增加了严格模式，用于处理之前版本 (ES3) 的一些不规范书写问题。启用严格模式需要在脚本头部加上这一行：

```
"use strict"
```

也可以单独在一个函数中执行严格模式:

```
function doSomething() {
    "use strict";
    // 函数体
}
```

严格模式会影响到 JavaScript 执行的方方面面，后文会经常出现这个概念。所有现代浏览器都支持严格模式。

语句

ECMAScript 中的语句和 Java 几乎一样。他理论上有几种宽松的写法，但并不推荐这样用。

- ECMAScript 允许省略分号[2]。
- 单条控制语句可以省略括号[3]。

2.2 关键字与保留字

JavaScript 有很多关键字，关键字不能用作标识符或属性名，具体的关键字视 ES 版本而定，这里不一一列举[4]。

此外，规范中还有一些未来的保留字，同样不能作为标识符或属性名使用。这些保留字多是其他语言中的关键字，有一定其他语言基础的人一般都不会用到。

2.3 变量

ECMAScript 变量视松散类型，意思是变量可以用于保存任何类型的数据。共有 3 个关键字可以声明变量：`var`，`let`，`const`。其中 `let`，`const` 只有在 ES6 级之后的版本才可以使用。

2.3.1 var 关键字

`var` 关键字是最原始的声明变量关键字。用法如下：

```
1 var message = "hi";
```

如果没有赋值，仅声明变量，变量会保存一个特殊值 `undefined`[5]。这里 `message` 被定义为一个保存字符串值的变量，但这样初始化变量不会将它标识为字符串类型，只是一个简单的赋值而已。

此外，虽然 ECMAScript 允许变量重新赋值，但是并不推荐这样使用。

var 声明作用域

使用 `var` 操作符定义的变量会成为包含它的函数的局部变量。在函数外是不起作用的。

[2]可能影响性能。同时降低可读性。

[3]影响不大，但建议保留。

[4]参考文章：`https://zhuanlan.zhihu.com/p/257105802`

[5]下一节讨论。

```
function test() {
    var message = "hi";
}
console.log(message)  // 出错!
```

如果我们需要在函数创建一个全局变量，只需要省略 var 关键字。

```
function test() {
    message = "hi";
}
console.log(message)  // 正确
```

警告 2.1. 虽然省略 *var* 操作符可以定义全局变量，但是不建议这么做。绝大多数面向对象的高级语言也不建议这么做。

如果要定义多个变量，可以在一条语句中用逗号分隔各个变量[6]。

var 声明提升

下面 JavaScript 代码不会报错。这是因为使用 var 关键字声明的变量会自动提升到函数作用域顶部:

```
function foo() {
    console.log(age);
    var age = 26;
}
```

上述代码等价于：

```
function foo() {
    var age;
    console.log(age);
    age = 26;
}
```

这就是所谓的” 提升”(hoist)，也就是把所有变量声明都拉到函数作用域的顶部。此外，反复多次使用 var 声明同一个变量也没有问题。

```
function foo() {
    var age = 16;
    var age = 26;
    var age = 36;
    console.log(age);
}
```

2.3.2 let 声明

let 和 var 作用差不多，设计 let 就是为了在大部分情况下取代 var。它们最明显的区别是，let 声明的范围是块作用域，而 var 是函数作用域。let 更符合一般高级语言的语法逻

[6]这种做法也不建议使用。

辑。

```
if (true) {
    var name = 'Matt';
    console.log(name);
}
console.log(name); // 正确
```

```
if (true) {
    let name = 'Matt';
    console.log(name);
}
console.log(name); // 错误
```

在这里，age 变量之所以不能在 if 块之外被引用，是因为它的作用域仅限于该块内部。块作用域是函数作用域的子集，因此适用于 var 的作用域同样适用于 let。

let 也不允许同一个块作用域中出现冗余声明。这样会报错：

```
var name;
var name;  // 出现两次，无误

let age;
let age;   // 出现两次，SyntaxError
```

此外，对声明冗余的报错，不会因混用 let 和 var 而受影响。这两个关键字声明的并不是不同类型的变量，它们只是指出变量在相关作用域如何存在。

暂时性死区

let 和 var 的另一个重要区别是：使用 let 声明的变量不会再作用域中被提升。

在解析代码时，JavaScript 引擎会注意到块后面的 let 声明，只不过在此之前不能以任何方式来引用未声明的变量。在 let 声明之前的执行瞬间被称为“暂时性死区”，在此阶段引用任何后面才声明的变量都会抛出 ReferenceError。

全局声明

与 var 关键字不同，使用 let 在全局作用域中声明的变量不会称为 windows 对象的属性(var 声明会)。

```
var name = 'Matt';
console.log(window.name); // 'Matt'

let age = 26;
console.log(window.age);  // undefined
```

不过 let 声明仍然是在全局作用域中发生的。相应变量会在页面的生命周期内存续。因此，为了避免 SyntaxError，必须确保页面不会重复声明同一个变量。

条件声明

在使用 var 声明变量时，由于声明被提升，JavaScript 引擎会自动将多余的声明在作用于顶部合并。因为 let 的作用域是块，所以不可能检查前面是否已经使用 let 声明郭同名变量，

同时也就不可能在没有声明的情况下声明它。

```
<script>
    var name = 'Pionpill';
    let age = 22;
</script>

<script>
    var name = 'Matt';
    // 没有问题，同时被作为一个提升声明来处理
    // 不需要检查之前是否声明过同名变量
    let age = 23;
    // age 之前声明过，报错
</script>
```

使用 try/catch 语句或 typeof 操作符也不能解决，因此条件快中 let 声明的作用域仅限于该块。

for 循环中的 let 声明

在 let 出现之前，for 循环定义的迭代变量会渗透到循环体外部：

```
for(var i=0; i<5; i++) {
    // 循环逻辑
}
console.log(i);   // 5
```

改用 let 后就不会出现这个问题：

```
for(let i=0; i<5; i++) {
    // 循环逻辑
}
console.log(i);   // ReferenceError
```

此外，由于 var 对应的是函数作用域，在 for 循环，for-in，for-of 循环中会出现种种意想不到的问题。

2.3.3 const 声明

const 的行为与 let 基本相同，唯一一个重要的区别是它声明的变量必须同时初始化变量，且尝试修改 const 声明的变量会报错。

此外，const 变量不可变本质上是指其引用不可变，并不是引用的对象不可变。

```
const name = "Pionpill";
name = "Matt";    // 报错

const person = {};
person.name = "Matt"; // 正确
```

此外，不能用 const 来声明迭代变量 (应为迭代变量会自增)：

```
for (const i=0; i<10; i++) {} // TypeError
```

不过，如果只想用 const 声明一个不会被修改的 `for` 循环变量，那是可以的。也就是说，每次迭代只是创建一个新变量。这对 `for-in`，`for-of` 循环特别有意义：

```
let i = 0;
for(const j=7; i<5; ++i) {
    console.log(j);
}
// 7,7,7,7,7

for(const key in {a:1,b:2}) {
    console.log(key);
}
// a,b

for (const value of [1,2,3,4,5]) {
    console.log(value);
}
// 1,2,3,4,5
```

如何声明变量

ES6 加入 `let` 和 `const` 关键字，其主要目的就是为了取代 `var` 关键字。由于 `var` 声明的变量会出现作用域，升格等问题，它已经在实践中被弃用了。至于为什么还要花大段时间讲 `var`，自然是为了看得懂一些远古代码。

至于 `const` 关键字，由于浏览器会保持它不变，无论是静态分析还是处理效率上，都要优于 `let`。当然，使用最多的还是 `let` 关键字。

总而言之，**不要使用 `var`，优先使用 `const`，`let` 次之。**

2.4 操作符

JavaScript 中的基本操作符和 C 语言系中的操作符运算是相同的，相同内容省略。此外，和 Python 一样，JavaScript 是一门十分灵活的语言，比如 `false == 0` 会返回 `true`，这种数据类型转换在操作符运算中时常会出现。限于操作符类型太多，涉及数据类型转换的操作更多，所以不一一说明，如果遇到问题，读者可通过 node.js 自行检查。

递增递减操作符

和 C/Java 语言中一样的内容省略。这个操作符往往会降低可读性，不建议使用，Python 就完全抛弃了。

ECMA 中的递增递减操作符除了对数值类型起作用，其他类型也起作用:

- 字符串，如果是有效的数值形式，则转换为数值进行计算，变量类型由此变成数值类型。
- 字符串，如果不是有效的数值形式，则转换为 `NaN`，变量类型由此变成数值类型。
- 布尔型，变成对应的 0/1 再进行计算，变量类型变成数值。

- 浮点数，加 1 或减 1。
- 对象，调用 `valueOf()` 方法取得可以操作的值。对得到的值采用上述规则。如果是 `NaN`，调用 `toString()` 并再次应用其他规则。变量类型变成数值类型。

指数操作符

ES7 新增了指数操作符，其语法和 Python 中的指数操作符相同。

```
console.log(Math.pow(3,2));  // 9
cnosole.log(3**2);           // 9
```

不止如此，ECMA 还提供了指数赋值操作符 **+:

```
let a = 3;
a **= 2;
console.log(a);   // 9
```

全等于运算

ECMA 中的相等 (==) 符号默认会转换操作数，而全等 (===) 符号则不会。

```
console.log("55" == 55);  // true
console.log("55" === 55); // false
```

与全等对应的还有不全等 (!==) 运算。

> **建议 2.2.** 多数情况下，由于等于运算涉及类型转换问题，更推荐使用全等运算，这有助于保持数据类型的完整性。

2.5 语句

省略 `if`，`while`，`for` 语句的介绍。

2.5.1 `for-in` 语句

`for-in` 语句是一种严格的迭代语句，用于枚举对象中的非符号键属性，语法如下:

```
for (property in expression) statement
```

下面是一个例子:

```
for (const propName in window) {
    document.write(propName);
}
```

这个例子使用 `for-in` 循环显示了 BOM 对象 `window` 的所有属性。这里控制语句中的 `const` 不是必须的，不过为了确保局部变量不被修改，推荐使用 `const`。

值得说明的是，ECMAScript 中对象的属性是无序的。

2.5.2 for-of 语句

for-of 语句是一种严格的迭代语句，用遍历可迭代对象的元素，语法如下:

```
for (property of expression) statement
```

下面是示例:

```
for (const el of [2,4,6,8]) {
    document.write(el);
}
```

2.5.3 标签语句

标签语句用于给语句加标签，语法如下:

```
label: statement
```

下面是一个例子:

```
start: for(lei i=0; i<count; i++) {
    console.log(i);
}
```

这个语句中的 start 是一个标签，可以在后面通过 break 或 continue 语句引用。标签语句的典型应用场景是嵌套循环。

2.5.4 with 语句

with 语句的用途是将代码作用域设置为特定的对象，其语法是:

```
with (expression) statement;
```

使用 with 语句的主要场景是针对一个对象反复操作，这时候将代码作用域设置为该对象能提供便利。

```
// 不使用 with 语句
let qs = location.search.substring(1);
let hostname = location.hostname;
let url = location.href;

// 使用 with 语句
with(location) {
    let qs = search.substring(1);
    let hostName = hostname;
    let url = href;
}
```

在 with 语句中，每个变量会首先被认为是一个局部变量。如果没有找到局部变量，则会搜索 location 对象，看它是否有同名属性。

建议 2.3. 严格模式下不允许使用 *with* 语句，由于 *with* 语句影响性能且难于调试，所以不建议使用。

3 变量，作用域与内存

3.1 原始值与引用值

原始值就是最简单的数据，引用值是由多个值构成的对象。JavaScript 中，6 个基本数据类型是原始值，变量按保存的是数据本身，其他数据类型保存的是引用。

与 Java/Python 不同的是，JavaScript 中的 String 类型是基本数据类型，是原始值，对应变量保存数据本身。

动态属性

对于引用值而言，可以随时添加，修改和删除其属性和方法。

```
let person = new Object();
person.name = "Pionpill";
```

原始值不能拥有属性，尽管尝试添加属性不会报错 (这样做没有意义)，但是调用时会报错。

注意，原始类型的初始化可以只使用原始字面量形式。如果使用的是 `new` 关键字，则 JavaScript 会创建一个 `Object` 类型的实例，但其行为类似原始值。

复制值

JavaScript 对值复制的方法同 Java 相同，原始值的复制将在内存中创建新的空间，而引用值的复制只是多加了一个引用 (指向同一块内存区域)。

传递参数

ECMAScript 中所有函数的参数都是按值传递的。也就是说在函数中会创建一个新的局部变量，是原变量的复制。

下面通过一个例子说明引用值是如何按值传递的。

```
function setName(obj) {
    obj.name = "Pionpill";
    obj = new Object();
    obj.name = "Brandon";
}

let person = new Object();
setName(person)
console.log(person.name); // "Pionpill"
```

上述例子中,最终的输出是 `"Pionpill"` 而不是 `"Brandon"`,注意我们全局变量中的 `person` 和传入函数的 (局部变量) `person` 是两个不同的变量，他们都指向一个 `Object` 实例。在第二

行，局部变量 person 修改了所指向的 Object 实例，创建 name 属性，值为 "Pionpill"。但在第 3 行，局部变量 person 创建了一个新的 Object 实例，不再指向原先的区域，因此此时再对局部变量 person 进行修改，不会影响到全局的 person 所指向的实例。

> **注意 3.1.** 上面示例代码仅因为要解释原理给出，实际作用中请不要随意在函数中改变属性或引用对象。

确定类型

我们知道 typeof 操作符可以用来确定数据类型，但是在面向对象编程中，我们往往需要知道某一实例是什么类型的对象。这时候需要用到 instanceof 操作符。

```
1 person instanceof Object; // person 是 Object 吗?
2 colors instanceof Array;  // colors 是 Array 吗?
```

> **注意 3.2.** *ECMA-262* 规定,任何实现内部 *[call]* 方法的对象都应该在 *typeof* 检测时返回 *"function"*。但并不是所有浏览器都这样做。

3.2 上下文与作用域

基础的省略。

需要指出的是，所有通过 var 定义的全局变量和函数都会成为 window 对象的属性和方法。使用 let 和 const 的顶级声明不会定义在全局上下文中，但在作用域链解析上效果是一样的。

作用域链增强

执行上下文主要有全局上下文和函数上下文两种 (eval() 调用内部存在第三种上下文)，但有其他方式来增强作用域链。某些语句会导致在作用域链前端临时添加一个上下文，这个上下文在实行代码后会被删除:

- try/catch 语句的 catch 块
- with 语句

变量声明

var，let，const 关键字以在前文进行过说明，这里不再累述。

在子块中如果声明和父块同名的变量会掩盖父级变量，也即在子块中对这一变量的调用指的都是子块中的变量。如果父块是全局作用域，则可以使用 windows.name 调用对应变量。

> **建议 3.1.** 虽然 *JavaScript* 允许父块和子块中出现同名变量，但不建议这样使用。

3.3 垃圾回收

标记清理

所有现代浏览器采用的 JavaScript 垃圾回收机制都是标记清理。当变量进入上下文时，变量会被加上存在于上下文中的标记。当变量离开上下文时，也会被加上离开上下文的标记[7]。

垃圾回收程序运行时，会标记内存中存储的所有变量。然后，它会将所有在上下文中的变量，以及被在上下文中的变量引用的变量的标记去掉。在此之后再被加上标记的变量就是待删除的了。随后垃圾回收程序做一次内存清理，销毁带标记的所有指并回收它们的内存。

> **注意 3.3.** 还有一种垃圾回收机制：引用计数。不过已经被绝大多数浏览器弃用了。但 *Python* 的垃圾回收机制采用的是这种方式，有兴趣请读者自行查阅资料。

垃圾回收机制是涉及浏览器底层的操作，了解这些底层知识对开发者并没有太多的益处，而且，开发者也无法改善这种底层操作 (需要浏览器改善)。因此，这里我省略了很多书上的理论内容。

内存管理

出于优化考虑，系统只会给浏览器分配较少的内存。因此将内存占用量保存在一个较小的值可以让页面性能更好。优化内存最好的方法就是只保存必要的数据。如果数据不需要，那么把它设置为 `null`，从而释放其引用。这也可以叫做解除引用。这个做法尤其适用于全局变量，因为局部变量在超出作用域后会自动被解除引用。

```
let person = new Object();
person.name = "Pionpill";

let person = null;    // 解除引用
```

不过要注意，解除一个值的引用并不会自动导致相关内存被回收 (可能多个变量指向同一对象)。解除引用的关键在于确保相关的值已经不在上下文。因此在下次垃圾回收时可能会将其回收。

使用 `let`，`const`

由于 `let`，`const` 关键字在声明变量时是块作用域，因此，在垃圾回收过程中变量会更加快速地被处理，而 `var` 则相对迟钝。

隐藏类

隐藏类是一种优化处理，Chrome 浏览器的 V8 引擎在解释后会采用这种优化处理，从而产生隐藏类。

简而言之，如果两个示例相同 (构造方法相同，类相同)，那么它们可能会指向同一个对象，这样就节省了开辟一个相同对象地空间。如果其中某个对象被改动了，则会智能地再创

[7] 加标记也是局部变量与全局变量不同的原因。

建一个新对象。这种处理方式和 Python 类似[8]。详细的处理过程请读者自行查阅文献，下面仅给出处理隐藏类的一些建议。

隐藏类固然有好处，但是如果使用不当，这会成为累赘，比如如果我们要创建两个不同的类实例：应该在创建时就区分它们，而不是使用相同的构造函数 (这样两个实例名会先指向同一实例，更改后再重新创建实例，步骤多余)。

```
function Article(opt) {
    this.title = "abababababa";
    this.author = opt;
}

let a1 = new Article();
let a2 = new Article("Pionpill");
```

此外，如果两个变量共享一个隐藏类，请不要使用 `delete` 语句修改类属性，这会导致生成相同的隐藏类片段，如果要删除，应该将值设为 `null`。

```
let a1 = new Article();
let a2 = new Article();

a1.author = null;
```

> **注意 3.4.** *CSDN 上鲜有隐藏类的相关文章，只有极致追求性能的开发者才会关注隐藏类的优化处理。在开发阶段无需过多关于这些内容，等优化阶段再处理。*

[8]我的另一本《Fluent Python》笔记有详细解释 Python 的这种做法。

II 数据类型

4 简单数据类型

ECMAScript 有 6 中简单数据类型 (也称原始类型)：Undefined，Null，Boolean，Number，String 和 Symbol(ES6 新增)。还有一种复杂的数据类 Object。Object 是一种无序名值对的集合。

此外，ECMAScript 不能定义自己的数据类型，所有值都可以用上述 7 中数据类型之一来表示。ECMAScript 的数据类型十分灵活，一种数据类型可以当作多种数据类型来使用。

4.1 typeof 操作符

对一个值使用 typeof 操作符会返回下列字符串之一：

- "undefined": 值未定义;
- "boolean": 布尔值;
- "string": 字符串;
- "number": 数值;
- "object": 对象 (不是函数) 或 null ;
- "function": 函数;
- "symbol": 符号;

typeof 在某些情况下返回的结果可能会让人费解。比如，调用 typeof(null) 返回的是 "object"。这是因为特殊值 null 被认为是一个对空对象的引用。

> **注意 4.1.** 严格来讲，函数在 *ECMAScript* 中被认为是对象，并不代表一种数据类型。可是，函数也有自己特殊的属性。为此，就有必要通过 *typeof* 操作符来区分函数和其他对象。

4.2 Undefined 类型

Undefined 类型只有一个值，就是特殊值 undefined。当使用 var 或 let 声明了变量但没有初始化时，就相当于给变量赋予了 undefined 值。

```
let message;
console.log(message == undefined); // true
```

我们也可以指定变量值为 undefined，与未初始化效果一样。

```
let message = undefined;
```

```
console.log(message == undefined); // true
```

> **注意 4.2.** 虽然可以显式地将 *undefined* 赋给某个值。但是不要这样做，字面量 *undefined* 主要是用于比较。而且在 *ES3* 之前没有 *undefined*。增加这个特殊值主要是用于和 *null* 区别。

与 undefined 字面意义不同的是，null 并不表示未声明变量，而表示没有引用对象。

```
let message;

console.log(message); // "undefined"
console.log(name);    // 报错
```

对于未声明的变量，只能执行一个有用的操作，就是对它调用 typeof。需要注意的是对未声明的变量和未初始化的变量调用 typeof 运算都会返回 "undefined"。

```
let message

console.log(typeof message); // "undefined"
console.log(typeof name);    // "undefined"
```

> **建议 4.1.** 都返回 *"undefined"* 往往会让人摸不着头脑，到底是那种情况。因此建议在声明变量的同时一定要进行初始化。这样，当返回 *"undefined"* 时，就会知道那是因为给定的变量尚未声明。

4.3 Null 类型

Null 类型同样只有一个值，即 null。逻辑上讲，null 值表示一个空对象指针，这也是给 typeof 传一个 null 会返回 "object" 的原因。

```
let car = null;
console.log(typeof car)  // "object"
```

在定义将来要保存对象值的变量时，建议使用 null 来初始化，不要使用其他值。这样，只要检查这个变量是不是 null 就可以直到这个变量是否在后来被重新赋予了一个对象的引用。

undefined 是由 null 派生而来的，因此 ECMA 将它们定义为表面上相等。

```
console.log(undefined == null);  // true
```

有别于 undefined ，如果要保存未来的对象，可以用 null 来进行填充。

此外，undefined 和 null 都是假值[1]。

[1]在判定语句中等价于 false。

4.4 Boolean 类型

Boolean 类型有两个字面值：true 和 false。不同于数值，true 不等于 1，false 不等于 0。

虽然布尔值只有两个，但是其他所有 ECMAScript 类型的值都有对应的布尔值的等价形式。要将一个其他类型的值转换为布尔值，可以调用特定的 Boolean() 转换函数。

表 2.1 基本数据类型与布尔型之间的转换

数据类型	转换为 true	转换为 false
Boolean	true	false
String	非空串	空串
Number	非零数值	0，NAN
Object	任意对象	null
Undefined	N/A	undefined

4.5 Number 类型

Number 类型使用 IEEE 754 格式表示整数和浮点数 (双精度浮点数)。

整型

其中整数常用的有有三种写法：十进制，八进制[2]，十六进制。其中八进制和十六进制表示法在数字前分别加 0 和 0x：

```
let intNum = 55;  // 十进制
let octNum = 070; // 八进制 56
let hexNum = 0xA; // 十六进制 10
```

如果字面量中包含的数字超出了应有的范围，会忽略前缀的零。

```
let octNum = 079; // 无效的八进制，当成 79 处理
```

> **注意 4.3.** *JavaScript 中的正零和负零是等同的。*

浮点型

要定义浮点型，数值中必须包含小数点，且小数点后至少有一个数字。虽然小数点前的数字不是必须的，但还是建议加上。

```
let floatNum1 = 1.1;
let floatNum2 = 0.1;
let floatNum2 = .1;   // 有效，但不建议这样用
```

出于优化考虑，ECMAScript 会想方设法把值转换为整数。如果值本省是整数，那么它会被自动转换为整型值。

[2]严格模式下无效。

```
let floatNum1 = 1.;   // 小数点后没有数字，当作整型1
let floatNum2 = 10.0; // 小数点后是零，当作整型10
```

此外，ECMAScript 也可以使用科学计数法，在一定情况下会将小数转换为科学计数法，其精度最高可达小数点后 17 位。

```
let floatNum1 = 3e-7; // 0.0000007
```

此外，和大多数高级语言一样，为了避免精度误差，请不要对小数运用比较运算。

值的范围

ECMAScript 可表示的最大最小值分别存储在了以下两个变量中 (大多数浏览器中值与下列相同)：

- `Number.MIN_VALUE`: 5e-324
- `Number.MAX_VALUE`: 1.797e+308

如果计算的值超出了这个范围，会自动被转换为特殊的 `Infinity`(无穷值)，负值则为 `-Infinity`。如果计算返回无穷，则该值将不能再进一步用于任何计算。可以用 `isFinite()` 函数判断是否为无穷值。

```
let result = Number.MAX_VALUE + Number.MAX_VALUE;
console.log(isFinite(result));   // false
```

此外，可以使用 `Number.NEGATIVE_INFINITY` 和 `Number.POSITIVE_INFINITY` 获取正负 `Infinity`。

NaN

`NaN` 用于表示不是数值 (Not a Number)，一般用于表示本要返回数值得操作失败了。

造成 `NaN` 得原因有很多，比如分子为零得运算。此外任何涉及 `NaN` 得操作始终返回 `NaN`。其次，`NaN` 不等于包括 `NaN` 在内得任何值。

```
console.log(NaN == NaN); // false
```

为了判断 `NaN`，JavaScript 提供了 `isNaN()` 函数。这个函数非常灵活，任何可以转化为数值类型得值都会返回 `true`:

```
console.log(isNaN(NaN));     // true
console.log(isNaN(10));      // false,10是数值
console.log(isNaN("10"));    // false,"10"可以转换为数值
console.log(isNaN("blue"));  // true,不可以转换为数值
console.log(isNaN(true));    // false,可以转换为数值1
```

数值转换

有 3 个函数可以将非数值转换为数值: `Number()`,`parseInt()`,`parseFloat()`。其中 `Number()` 是转型函数，可用于任何数据类型。后两个函数主要用于将字符串转换为数值。

`Number()` 函数基于如下规则执行转换:

- 布尔值: `true` 转为 1，`false` 转为 0。

- 数值: 直接返回。
- null: 返回 0。
- undefined: 返回 NaN。
- 字符串，有如下规则：
  - 整数字符串，包含字符前有加减号得情况，转换为一个对应的十进制数值。忽略最前面的零。
  - 浮点格式：转换为浮点数。
  - 十六进制：例如 "0xf"，转换为对应的十进制数值。
  - 空字符串：转为 0。
  - 其他情况：NaN。
- 对象: 调用 valueOf() 方法，并按照上述规则转换返回的值。如果转换结果为 NaN，则调用 toString() 方法，再按照转换字符串的规则转换。

考虑到 Number() 函数转换字符串时相对复杂且有点反常规，通常需要在得到整数时可以优先使用 parseInt() 函数。

parseInt() 函数更专注于字符串是否包含数值模式。字符串最前面的空格会被忽略，从第一个非空格字符串开始转换。如果第一个字符串不是数值字符，加号，减号。parseInt() 立即返回 NaN。这意味着空字符串也返回 NaN。如果第一个字符是数值字符，加号，减号；则继续依次检测每个字符，直到字符串末尾，或碰到非数值字符。

```
parseInt("1234blue"); // 1234
parseInt("22.5");     // 22
```

假设字符串第一个字符是数值字符，parseInt() 函数也能识别不同的整数格式 (十六进制，八进制，十进制)。如果字符以 “0” 开头，且紧跟着数值字符，在非严格模式下会被某些实现解释为八进制整数。

```
parseInt("0xA");      // 10
parseInt("070");      // 56(视浏览器而定)
```

不同的数值格式很容易混淆，因此 parseInt() 函数也接收第二个参数，用于指定进制数，例如:

```
parseInt("0xAF",16);  // 175
parseInt("AF",16);    // 175
parseInt("AF");       // NaN
```

通过第二个参数，可以极大扩展转换后获得的结果类型:

```
parseInt("10",2);     // 二进制
parseInt("10",8);     // 八进制
parseInt("10",10);    // 十进制
parseInt("10",16);    // 十六进制
```

因为不传底数参数相当于让 parseInt() 函数自己判断如何解释，所以应始终传第二个参数，这个参数多数情况下是 10。

建议 4.2. *parseInt() 函数与 Number() 函数的主要区别在于：1. 对空字符串的处理。2. 字符串后面非数值字符的处理。3. 八进制处理 (视浏览器决定)。*

4.6 String 类型

String 数据类型表示零个或多个 16 为 Unicode 字符序号。字符串可以用双引号 (")，单引号 (')，反引号 (`) 标志。但引号必须同队出现，不同类型引号不可以随意匹配。

注意，ECMAScript 中的字符串是不可变的。

转换为字符串

JavaScript 提供了两种方式转换字符串。首先是几乎所有值都有的 toString() 方法。这个方法唯一的用途就是返回当前值的字符串等价物。

toString() 方法可用于数值，布尔值，对象和字符串值 (字符串值返回自身的一个副本)。null 和 undefined 值没有 toString() 方法。

```
let age = 11;
let ageString = age.toString();     // 字符串 "11"
let found = true;
let foundString = found.toString(); // 字符串 "true"
```

多数情况下，toString() 不接收任何参数。不过，在对数值调用这个方法时，toString() 可以接收一个底数参数，即以什么底数来输出数值的字符串表示。默认为十进制 (即参数为 10)。

```
let num = 10;
console.log(num.toString()); // "10"
console.log(num.toString(2)); // "1010"
console.log(num.toString(8)); // "12"
console.log(num.toString(16)); // "a"
```

如果不确定一个值是不是 null 或 undefined ，可以使用 String() 转型函数，它始终会返回表示相应类型值的字符串，其处理规则如下:

- 如果只有 toString() 方法，则调用该方法并返回结果。
- 如果值是 null，则返回 null。
- 如果值是 undefined，则返回 undefined。

模板字面量

ES6 新增了使用模板字面量定义字符串的能力。与使用单引号或双引号不同，模板字面量保留换行字符，可以跨行定义字符串。

```
let multiLine1 = "first lin\nsecond line";
let multiLine1 = "first line
second line";

console.log(multiline1);
```

```
// first line
// second line

console.log(multiline2);
// first line
// second line
```

字符串插值

模板字面量最常用的一个特性是支持字符串插值，也就是可以在一个连续定义中插入一个或多个值。技术上讲，模板字面量不是字符串，而是一种特殊的 JavaScript 语法表达式，只不过求值后得到的是字符串。模板字面量在定义时立即求值并转换为字符串实例，任何插入的变量也会从它们最接近的作用域中取值。

字符串插值通常在 `${}` 中使用一个 JavaScript 表达式实现:

```
let value = 5;
let exponent = 'second';
// 之前的写法
let interpolatedString = value + ' to the ' + exponent + ' power is ' + (value * value);
// 现在的写法
let interpolatedString = '${value} to the ${exponent} power is ${value*value}'; // $ 写法
```

所有插入的值都会使用 `toString()` 强制转型为字符串，而且任何 JavaScript 表达式都可以用于插值。嵌套的模板字符串无需转义:

```
function capitalize(word) {
    return '${word[0].toUpperCase()} ${word.slice[1]}'
}
console.log('${capitalize('hello')}, ${capitalize('world')}!') // Hello, World!
```

模板字面量标签函数

模板字面量也支持定义标签函数，通过标签函数可以自定义插值行为。标签函数会接收被插值记号分隔后的模板和对每个表达式求值的结果。

标签函数本身是一个常规函数，通过前缀到模板字面量来应用自定义行为。标签函数接收到的参数依次是原始字符串数组和对每个表达式求值的结果。这个函数的返回值是对模板字面量求值得到的字符串。

```
let a = 6;
let b = 9;

function simpleTag(strings, aValExpression, bValExpression, sumExpression) {
    console.log(strings);
    console.log(aValExpression);
    console.log(bValExpression);
    console.log(sumExpression);
    return 'foobar';
}

let untaggedResult = '${a} + ${b} == ${a+b}';
```

```
let taggedResult = simpleTag'${a} + ${b} == ${a+b}';
// ["","+","=",""]
// 6
// 9
// 15

console.log(untaggedResult); // "6 + 9 = 15"
console.log(taggedResult);   // "foobar"
```

因为表达式参数的数量是可变的，所以通常应该使用剩余操作符将它们收集到一组数组中:

```
let a = 6;
let b = 9;

function simpleTag(strings, ...expressions) {
    console.log(strings);
    for (const expression of expressions) {
        console.log(expression);
    }
    return 'foobar';
}

let taggedResult = simpleTag'${a} + ${b} == ${a+b}';
// ["","+","=",""]
// 6
// 9
// 15

console.log(taggedResult);   // "foobar"
```

对于有 n 个插值的模板字面量，传给标签函数的表达式参数的个数始终是 n。而传给标签函数的第一个参数所包含的字符串个数则始终是 n+1。因此，如果你想把这些字符串和对表达式求值的结果拼接起来作为默认返回的字符串，可以这样做:

```
    let a = 6;
    let b = 9;

    function zipTag(strings, ...expressions) {
        return strings[0] + expressions.map((e,i) => '${e}${strings[i+1]}').join('');
    }

let taggedResult = zipTag'${a} + ${b} == ${a+b}';

console.log(taggedResult);   // "6 + 9 = 15"
```

原始字符串

使用模板字面量也可以直接获取原始的模板字面量内容 (如换行符或 Unicode 字符)，而不是被转换后的字符表示。为此，可以使用默认的 `String.raw` 标签函数:

```
console.log('\u00A9');          // ©
```

```
console.log(String.raw'\u00A9'); // \u00A9

console.log(String.raw'first line
second line')
// first line
// second line 对实际的换行符不起作用
```

4.7 Symbol 符号类型

Symbol 是 ES6 新增的数据类型。符号是原始值，是唯一的，不可变的。符号的用途是确保对象属性使用唯一标识符，不会发生属性冲突的危险。

符号的基本用法

符号要使用 Symbol() 函数初始化，也可以传入一个字符串参数作为符号的描述，将来可以通过这个字符串来调试代码。但是，这个字符串参数与符号定义或标识完全无关:

```
let firstSymbol = Symbol();
let secondSymbol = Symbol();

let thirdSymbol = Symbol("foo");
let forthSymbol = Symbol("foo");

console.log(firstSymbol == secondSymbol); // false
console.log(thirdSymbol == forthSymbol); // false

console.log(firstSymbol); // Symbol()
console.log(thirdSymbol); // Symbol(foo)
```

Symbol 函数不能与 new 关键字一起作为构造函数使用。这样做是为了避免创建符号包装对象，如果一定要做可以使用 Object() 函数:

```
let mySymbol = Symbol;
let myWrappedSymbol = Object(mySymbol);
console.log(typeof myWrappedSymbol)
```

使用全局符号注册表

如果运行时的不同部分需要共享和重用符号实例，那么可以用一个字符串作为键，在全局符号注册表中创建并重用符号。为此，需要使用 Symbol.for() 方法:

```
let fooGlobalSymbol = Symbol.for('foo');
console.log(typeof fooGlobalSymbol); // symbol
```

Symbol.for() 对每个字符串健都执行等幂操作。第一次使用某个字符串调用时，它会检查全局运行时注册表，发现不存在对应的符号，于是就会生成一个新符号实例并添加到注册表中。后续使用相同字符串的调用同样会检查注册表，大仙存在与该字符串对应的符号，然后就会返回该符号实例。

```
let fooGlobalSymbol = Symbol.for('foo');    // 创建新符号
let otherFooGlobalSymbol = Symbol.for('foo'); // 重用已有符号

console.log(fooGlobalSymbol === otherFooGlobalSymbol); // true
```

在全局注册表中定义的符号和使用 `Symbol()` 定义的符号并不等同:

```
let localSymbol = Symbol('foo');
let globalSymbol = Symbol.for('foo');

console.log(localSymbol === globalSymbol); // false
```

全局注册表中的符号必须使用字符串键来创建，因此作为参数传给 `Symbol.for()` 的任何值都会被转换为字符串。此外，注册表中使用的键同时也会被用作符号描述。

```
let emptyGlobalSymbol = Symbol.for();
console.log(emptyGlobalSymbol);  // Symbol(undefined)
```

还可以使用 `Symbol.keyFor()` 来查询全局注册表，这个方法接收符号，返回该全局符号对应的字符串键。如果查询的不是全局符号，则返回 `undefined` 。

```
let s = Symbol.for('foo');
console.log(Symbol.keyFor(s));   // foo

let s2 = Symbol('bar');
console.log(Symbol.keyFor(s2));  // undefined

Symbol.keyFor(123);              // TypeError: 123 is not a symbol
```

使用符号作为属性

凡是可以使用字符串或数值作为属性的地方，都可以使用符号。这就包括了对象字面量属性 `Object.defineProperty()/Object.defineProperties()` 定义的属性。对象字面量只能在计算性语法中使用符号作为属性。

> **注意 4.4.** *原文 P47-55 页还有更多关于 Symbol 的介绍，但本人觉得，Symbol 作为新的数据类型并不值得这么大篇幅的详细介绍，尤其是在基础篇中，故省略，读者可自行查看原文，但不查看也不会影响后续阅读。*

5 基本引用类型

在 ECMAScript 中并没有类的概念，只有引用类型。从技术上将 JavaScript 是一门 OOP 语言，但是 ECMAScript 确实类，接口这些结构。引用类型有时候也被称为对象定义，因为它们描述了自己的对象应有的属性和方法。

注意 5.1. 引用类型封装了很多方法，除了特别重要的一概省略。

5.1 Object

ES 中的对象其实就是一组数据和功能的集合。对象通过 new 操作符后跟对象类型的名称来创建。开发者可以通过创建 Object 类型的实例来创建自己的对象，然后再给对象添加属性和方法。

```
let o = new Object();
let p = new Object;   // 可行，但不推荐
```

类似 Java 中的 java.lang.Object ES 中的 Object 也是派生其他对象的基类。Object 类型的所有属性和方法在派生的对象上同样存在，每个 Object 属性都有如下属性和方法:

- constructor: 用于创建当前对象的函数。在前面例子中值是 Object() 函数。
- hasOwnProperty(propertyName): 用于判断当前对象实例上是否存在给定的属性。要检查的属性名必须是字符串或符号。
- isPrototypeOf(object): 用于判断当前对象是否为另一个对象的原型。
- propertyIsEnumerable(propertyName): 用于判断给定的属性是否可以使用 for-in 语句枚举，属性名必须是字符串。
- toLocalString(): 返回对象的字符串标识，该字符串反映对象所在的本地化执行环境。
- toString(): 返回对象的字符串表示。
- valueOf(): 返回对象对应的字符串，数值或布尔值表示，通常与 toString() 的返回值相同。

注意 5.2. 严格来讲，*ECMA* 中的对象行为不一定适合 *JavaScript* 中的其他对象，因为还有 *BOM*，*DOM* 对象。所以并不是所有对象都继承自 *Object* 对象。

创建 Object 对象有两种方式:

- 使用 new 操作符和 Object 构造函数。

```
let person = new Object();
person.name = "Nicholas";
person.age = 29;
```

- 使用对象字面量，直观方便。

```
let person = {
    name: "Nicholas",
```

```
    age: 29
};
```

注意 5.3. 在使用对象字面量表示法定义对象时，并不会实际调用 *Object* 构造函数。

为对象添加属性时，也可以使用数字等其他形式作为属性名，但最后都会转换为字符串。除了通过点访问属性，也可以通过中括号访问属性:

```
person.name = "Pionpill";
person["name"] = "Pionpill";
```

5.2 Date

要创建日期对象，就使用 new 操作符来调用 Date 构造函数：

```
let now = new Date();
```

不带参数的 `Date()` 将获取当前的时间，开发者可以根据其他构造函数创建对应的日期。

`Date.parse()` 方法接收一个表示日期的字符串参数，尝试将这个字符串转换为表示该日期的毫秒数 (可以作为 `Date()` 的参数)。ECMA 提供的几种日期格式如下:

- 月/日/年: 5/23/2019
- 月名日, 年: May 23, 2019
- 周名月名日年时: 分: 秒时区: Tue May 23 2019 00:00:00 GMT-0700
- YYYY-MM-DDTHH:mm:ss.sssZ: 2019-05-23T00:00:00

```
let someDate = new Date(Date.parse("May 23, 2019"));
let someDate = new Date("May 23, 2019"); // 语法糖: 直接传入字符串, Date() 会隐式调用
    Date.parse()
```

`Date.UTC()` 也可以返回日期的毫秒表示，但传入的参数是字符串:

```
let allFives = new Date(2005, 4, 5, 17, 55, 55); // 语法糖: 隐式调用 Date.UTC()
```

`Date` 重写了 `Object` 的几个重要的方法:

- `toLocalString()` : 返回浏览器运行的本地环境一致的时间日期。
- `toString()`: 返回带时区信息的日期时间 (上面的加强版)。
- `valueOf()`: 返回时间的毫秒表示，自世界协调时间起 (1970 年 1 月 1 日零时)。

5.3 RegExp

ECMAScript 通过 RegExp 类型支持正则表达式。

```
let expression = /pattern/flags;
```

这里包含两部分:

- **pattern**: pattern（模式）可以是任何简单或复杂的正则表达式，包括字符类、限定符、分组、向前查找和反向引用。
- **flags**: 每个正则表达式可以带零个或多个 flags（标记），用于控制正则表达式的行为。

表示匹配模式的标记如下:

- g: 全局模式，找所有，而不是找第一个就停止。
- i: 不区分大小写。
- m: 多行模式，查找到第一行文本末尾继续查找。
- y: 黏附模式，只查找从 lastIndex 开始及之后的字符串。
- u: Unicode 模式。
- s: dotAll 模式，表示元字符，匹配任何字符 (换行等)。

此外，元字符必须使用斜杠转义。

```
// 匹配所有 "at"
let pattern1 = /at/g;
// 匹配第一个 "bat" 或 "cat"
let pattern2 = /[bc]at/i;
// 匹配第一个"[bc]at", 忽略大小写
let pattern2 = /\[bc\]at/i;
```

上述写法是一种语法糖，也可以使用构造函数:

```
// 效果相同
let pattern1 = /[bc]at/i;
let pattern2 = new RegExp("[bc]at", "i");
```

在构造函数中，元字符必须经过两次转义:

```
const re1 = /cat/g;
console.log(re1); // "/cat/g"
```

5.4 包装类型

为了方便操作原始值，ECMAScript 提供了 3 中特殊的引用类型: Boolean, Number, String。我们看这样一个例子:

```
let s1 = "some text";
let s2 = s1.substring(2);
```

理论上，作为简单类型的 String 是没有方法的。那为什么会这样呢，语法糖！实际上在浏览器引擎中，涉及到 String 的代码会被这样执行:

- 创建一个 String 引用类型的实例；
- 调用实例上的特定方法；

- 销毁实例。

这种行为可以让原始值拥有对象的行为。对布尔值和数值而言，以上 3 步也会在后台发生，只不过使用的是 Boolean 和 Number 包装类型而已。

引用类型与原始值包装类型的主要区别在于对象的生命周期。在通过 new 实例化引用类型后，得到的实例会在离开作用域时被销毁，而自动创建的原始值包装对象则只存在于访问它的那行代码执行期间。这意味着不能在运行时给原始值添加属性和方法。

```
let s1 = "some text";
s1.color = "red";
console.log(s1.color); // undefined
```

Object 构造函数作为一个工厂方法，能够根据传入值的类型返回相应原始值包装类型的实例：

```
let obj = new Object("some text");
console.log(obj instanceof String); // true
```

注意，强制类型转换和 new 调用构造函数创建的东西是不一样的，虽然函数名一样:

```
let value = "25";
let number = Number(value); // 转型函数
console.log(typeof number); // "number"
let obj = new Number(value); // 构造函数
console.log(typeof obj);    // "object"
```

5.5 单例内置对象

ECMA-262 对内置对象的定义是“任何由 ECMAScript 实现提供、与宿主环境无关，并在 ECMAScript 程序开始执行时就存在的对象”。这就意味着，开发者不用显式地实例化内置对象，因为它们已经实例化好了。包括 Object、Array 和 String。另外还有两个不明显的单例内置对象：Global 和 Math。

5.5.1 Global

Global 对象是 ECMAScript 中最特别的对象，因为代码不会显式地访问它。ECMA-262 规定 Global 对象为一种兜底对象，它所针对的是不属于任何对象的属性和方法。事实上，不存在全局变量或全局函数这种东西。在全局作用域中定义的变量和函数都会变成 Global 对象的属性。本书前面介绍的函数，包括 isNaN()、isFinite()、parseInt() 和 parseFloat()，实际上都是 Global 对象的方法。

URL 编码方式

ECMAScript 提供了两个对 URI 编码的方法:

```
let uri = "http://www.wrox.com/illegal value.js#start";
// "http://www.wrox.com/illegal%20value.js#start"
```

```
console.log(encodeURI(uri));
// "http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.js%23start"
console.log(encodeURIComponent(uri));
```

同时也提供对应的解码方法:

```
let uri = "http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.js%23start";
// http%3A%2F%2Fwww.wrox.com%2Fillegal value.js%23start
console.log(decodeURI(uri));
// http:// www.wrox.com/illegal value.js#start
console.log(decodeURIComponent(uri));
```

`eval()` 方法

`eval()` 这个方法就是一个完整的 ECMAScript 解释器，它接收一个参数，即一个要执行的 ECMAScript 字符串：

```
eval("console.log('hi')");
console.log('hi');
```

当解释器发现 `eval()` 调用时，会将参数解释为实际的 ECMAScript 语句，然后将其插入到该位置。

在非严格模式下,通过 `eval()` 执行语句与没有 `eval()` 效果相同。在非严格模式下 `eval()` 内部创建的变量和函数无法被外部访问。

> **注意 5.4.** *`eval()` 方法很危险，了解即可，一般不要使用。*

`Global` 对象属性

`Global` 对象有很多属性，所有无需通过对象调用的属性都是 `Global` 对象属性，包括以下类型:

- 特殊值: undefined, NaN, Infinity
- 构造函数: Object, Array, Function, Boolean, String, Number, Date, RegExp, Symbol
- 错误类型构造函数: Error, EvalError, RangeError, ReferenceError, SyntaxError, TypeError, URIError

`window` 对象

虽然 ECMA-262 没有规定直接访问 `Global` 对象的方式，但浏览器将 `window` 对象实现为 `Global` 对象的代理。因此，所有全局作用域中声明的变量和函数都变成了 `window` 的属性。

```
var color = "red";
function sayColor() {
    console.log(window.color);
    return this;  // this 表示 Global 对象
}
window.sayColor(); // "red"
```

5.5.2 `Math`

ECMAScript 提供了 `Math` 对象作为保存数学公式、信息和计算的地方。`Math` 对象提供了一些辅助计算的属性和方法。略...

6 集合引用类型

6.1 Array 数组

数组是最常见的数据结构，ECMAScript 的数组比其他语言有几个不同:

- ECMAScript 的数组是对象。
- 数组每个槽位可以存储不同的值。
- 数组会动态扩容。

6.1.1 数组表现

创建数组的方式非常多:

```
// new 创建空数组
let colors = new Array();
// new 创建指定大小的数组
let colors = new Array(3);
// new 创建指定元素的数组
let colors = new Array("red", "blue", "yellow");
// 字面量创建数组
let colors = ["red", "blue", "yellow"];
// 字面量创建有空位的数组, 空位被视作 undefined
let colors = ["red",,,,,"blue"];
```

此外，ES6 还提供了两个创建数组的静态方法: `from()` 与 `of()`。`from()` 用于将类数组结构转换为数组实例，而 `of()` 用于将一组参数转换为数组实例，请读者自行尝试这两个方法。

`isArray()`

同一个上下文中可以使用 `instanceof` 判断数组，但是不同上下文会有不同版本的 Array 构造函数，导致 `instanceof` 判断失败，这时候可以使用静态方法 `isArray()`

```
if (Array.isArray(value)) {
    ......
}
```

6.1.2 数组作为映射，栈，队列

ECMAScript 的 Array 有如下方法，不得不怀疑 Array 到底是如何实现的:

```
const a = ["foo", "bar", "baz", "qux"]; // 因为这些方法都返回迭代器，所以可以将它们的内容
// 通过Array.from()直接转换为数组实例
const aKeys = Array.from(a.keys());     // [0, 1, 2, 3]
const aValues = Array.from(a.values()); // ["foo", "bar", "baz", "qux"]
const aEntries = Array.from(a.entries()); // [[0, "foo"], [1, "bar"], [2, "baz"], [3, "qux"]]
```

因此，可以这样写:

```
const a = ["foo", "bar", "baz", "qux"];
for (const [idx, element]of a.entries()) {
    alert(idx);
    alert(element);
}
```

此外，数组还可以像栈，队列一样操作，ECMAScript 提供了对应的方法。

```
// 栈操作
let colors = new Array(); // 创建一个数组
let count = colors.push("red", "green"); // 压入两项
let item = colors.pop();    // 弹出一项
// 队列操作
let item = colors.shift(); // 取得最后一项
```

6.1.3 谓词，迭代与归并

谓词 (predicate) 可以看作对数据的筛选，数组中的 `find()` 与 `findIndex()` 都可以使用谓词筛选数据。

```
const people = [
    {
        name: "Matt",
        age: 27
    },
    {
        name: "Nicholas",
        age: 29
    }
];
alert(people.find((element, index, array) => element.age < 28));
// {name: "Matt", age: 27}
alert(people.findIndex((element, index, array) => element.age < 28));
// 0
```

ECMAScript 为数组定义了 5 个迭代方式:

- `every()`: 对数组每一项都运行传入的函数，如果每一项函数都返回 `true`，这个方法返回 `true`。
- `some()`: 对数组每一项都运行传入的函数，如果有一项函数返回 `true`，则这个方法返回 `true`。这些方法都不改变调用它们的数组。
- `filter()`: 对数组每一项都运行传入的函数，函数返回 `true` 的项会组成数组之后返回。
- `forEach()`: 对数组每一项都运行传入的函数，没有返回值。
- `map()`: 对数组每一项都运行传入的函数，返回由每次函数调用的结果构成的数组。

ECMAScript 为数组提供了两个归并方法：`reduce()` 和 `reduceRight()`。这两个方法都会迭代数组的所有项，并在此基础上构建一个最终返回值。

6.1.4 ArrayBuffer

定型数组 (typed array) 更像我们常规认知中的数组，是申请内存后不可变的，引入它的目的是提高性能。如果熟悉其他强类型语言 (如 Java)，一定会对 Array 感到不可思议，它能同时作为列表，队列，栈，映射使用，可以存储任意类型；但方便总是有代价的：性能不高。

ArrayBuffer 是所有定型数组及视图引用的基本单位。ArrayBuffer() 是可用于在内存中分配特定数量的字节空间。

```
const buf = new ArrayBuffer(16); // 在内存中分配16字节
alert(buf.byteLength);         // 16
```

ArrayBuffer 一经创建就不能再调整大小。不过，可以使用 slice() 复制其全部或部分到一个新实例中：

```
const buf1 = new ArrayBuffer(16);
const buf2 = buf1.slice(4, 12);
alert(buf2.byteLength); // 8
```

ArrayBuffer 在申请内存时有几个注意点:

- 分配失败时会抛出错误。
- 分配内存大小有限制 Number.MAX_SAFE_INTEGER2^{53} − 1 字节。
- 会将所有二进制位初始化为 0。
- 自动 GC。

不能仅通过对 ArrayBuffer 的引用就读取或写入其内容。要读取或写入 ArrayBuffer，就必须通过视图。视图有不同的类型，但引用的都是 ArrayBuffer 中存储的二进制数据。

6.2 Map 映射

6.2.1 Map

在 ES6 以前，JavaScript 实现” 键/值” 式存储可以使用 Object 来方便高效地完成。这样做没什么问题，但 ES6 以后更推荐使用 Map 存储键值对。

Map 的构造方法有很多，比较常用的可以直接通过二维数组构造:

```
const m = new Map([
    ["key1", "val1"],
    ["key2", "val2"],
    ["key3", "val3"]
]);
```

JavaScript 的 Map 有一点与众不同：它是有序的。而 Object 作为 Map 使用时却是无序的，这两者之前有如下差异:

- Map 有序，Object 无序。

- Map 在同等内存大小下可以比 Object 多存储 50% 的数据。
- 插入性能: Map 比 Object 好一点。
- 查找性能: Object 比 Map 好一点。
- 删除性能: Map 比 Object 好一点。

6.2.2 WeakMap

WeakMap 时 Map 的子集。这里的 weak 指的是 GC 对待 “弱映射” 中键的方式。

弱映射中的键只能是 Object 或者继承自 Object 的类型，尝试使用非对象设置键会抛出 TypeError。值的类型没有限制。

WeakMap 中 “weak” 表示弱映射的键是 “弱弱地拿着” 的。意思就是，这些键不属于正式的引用，不会阻止垃圾回收。但要注意的是，弱映射中值的引用可不是 “弱弱地拿着” 的。只要键存在，键/值对就会存在于映射中，并被当作对值的引用，因此就不会被当作垃圾回收。

```
const wm = new WeakMap();
wm.set({}, "val");
```

上述代码执行完后，由于空对象不存在引用，所以它会被 GC，同时键被 GC 后，值也失去了引用，也会被 GC。

因为 WeakMap 中的键/值对任何时候都可能被销毁，所以没必要提供迭代其键/值对的能力。因为 WeakMap 实例不会妨碍垃圾回收，所以非常适合保存关联元数据。

6.3 Set 集合

Set 也是 ES6 新增的类型，Set 在很多方面都像是加强的 Map，这是因为它们的大多数 API 和行为都是共有的。

同样，可以使用数组初始化 Set。

```
const s1 = new Set(["val1", "val2", "val3"]);
```

Set 的很多特性和 Map 一致，可以存储任意类型，维护插入顺序，用于对应的弱集合 WeakSet。

III 函数与类

7 函数

7.1 定义函数的多种方式

ECMAScript 中，每个函数都是 Function 类型的实例[1]。而 Function 也有属性和方法，跟其他引用类型一样。函数名就是指向函数对象的指针，而且不一定与函数本身密切绑定。函数通常以函数声明的方式定义:

```
function sum(num1, num2) {
    return num1 + num2;
}
```

另一种定义函数的语法是函数表达式。函数表达式与函数声明几乎是等价的，但是函数表达式不会提升，也即必须先声明后调用:

```
let sum = function(num1, num2) {
    return num1 + num2;
};
```

函数表达式可以通过这种方式立刻被调用，块级作用域的语句会被立即执行:

```
(function() {
    // 块级作用域
})();
```

注意函数表达式最后是要带分号的，而函数声明则没有必要。

还有一种定义函数的方式与函数表达式很像，叫 “箭头函数”(类似 lambda 表达式):

```
let sum = (num1, num2) => {
    return num1 + num2;
};
```

最后一种定义函数的方式是使用 Function 构造函数。这个构造函数收集任意多个字符串参数，最后一个参数为函数体，之前的参数都是新函数的参数:

```
let sum = new Function("num1", "num2", "return num1 + num2");
```

这种方式并不推荐，会影响内部处理的性能。不过要记作 ECMAScript 中的函数是对象，函数名是指针。

箭头函数是 ES6 新增的语法。箭头函数和函数表达式十分相像，任何可以使用函数表达式的地方，都可以使用箭头函数。

[1] 与 Python 一样。

箭头函数的语法非常适合嵌入函数的场景:

```
let ints = [1,2,3];
console.log(ints.map(function(i) {return i+1;})); // [2,3,4]
console.log(ints.map((i) => {return i+1;})); // [2,3,4]
```

如果只有一个参数，那可以不用括号。

```
let double = (x) => {return 2*x;};
let triple = x => {return 3*x;};
```

如果函数语句十分精简，只需要一行代码，那么大括号也可以省略。省略大括号会隐式返回这些代码的值 (不需要额外的 `return` 语句)。

```
let double = x => 2 * x;
```

箭头函数虽然语法简洁，但也有很多场合不适合。箭头函数不能使用 `arguments` ， `super` 和 `new.target`，也不能用作构造函数。此外，箭头函数也没有 `prototype` 属性。

7.2 ECMAScript 的函数

函数名

因为函数名是指向函数的指针，所以它们跟其他包含对象指针的变量具有相同行为。这意味着一个函数可以拥有多个名称。

ECMAScript6 的所有函数对象都会暴露一个和只读的 `name` 属性，其中包含关于函数的信息。多数情况下，这个属性保存的就是一个函数标识符，或者说是一个字符串化的变量名。如果它是使用 `Function` 构造函数创建的，则会标识为 “anonymous”。

```
function foo() {}
let bar = function() {};
let baz = () => {};

console.log(foo.name);    // foo
console.log(bar.name);    // bar
console.log(baz.name);    // baz
console.log(() => {}.name);   // 空字符串
console.log((new Function()).name);  // anonymous
```

如果一个函数是获取函数，设置函数，或者使用 `bind()` 实例化，那么标识符前面会加上一个前缀[2]。

[2]后文会详细说明

函数参数

ECMAScript 的函数去其他大多数语言不同，它既不关心参数的类型也不关心参数的个数，甚至传入参数的个数可以超过定义函数时参数的个数。

之所以会这样，是因为 ECMAScript 函数的参数在内部表现为一个数数组。函数被调用时总会接收一个数组，但函数并不关系这个数组中有什么 (甚至什么都没有)。

事实上，在使用 `function` 关键字定义 (非箭头) 函数时，可以在函数内部访问 `arguments` 对象，从中取得传进来的每个参数值。我们可以通过中括号语法访问 `arguments` 对象中的元素 (第一个是 `arguments[0]`)。而要确定传进来多少个参数，可以访问 `arguments.length` 属性。

```
function sayHi(name, message) {
    console.log("Hello" + name + "," + message);
    }

// 等价写法
function sayHi() {
    console.log("Hello" + arguments[0] + "," + arguments[1]);
}
```

上面例子表明，ECMAScript 函数的参数只是为了方便才写出来的，并不是必须写出来。与此同时，ECMAScript 中的命名参数不会创建让之后的调用必须匹配的函数签名。这是因为根本不存在验证命名参数的机制。

如前所述，ECMAScript 的函数参数本质上是一个数组，且并不对这个数组进行校验，因此函数无法重载。

ES6 起，可以显示定义默认参数。

```
function makeKing(name = 'Henry') {
    return name;
}
```

ECMAScript 支持参数扩展，传入参数时可以通过... 将可迭代对象拆分为独立的参数:

```
function getSum() {
    let sum = 0;
    for (let i = 0; i < arguments.length; ++i) {
        sum += arguments[i];
    }
    return sum;
}
let values = [1,2,3,4];
console.log(getSum(...values)); // 10
```

同时，定义函数时... 表示多个任意个参数组成的数组:

```
function ignoreFirst(firstValue, ...values) {
    console.log(values);
}
```

```
ignoreFirst(1,2,3); // [2, 3]
```

7.3 函数内部

我们知道，JavaScript 中的函数本质上是 `Function` 类的实例，因此将 JavaScript 的函数当作对象来看，既可以在参数中使用函数，也可以定义函数闭包 (类似于 Java 内部类)，也可以拥有属性，总而言之，`Object` 能做的，他都能做。

在 ES6 后，函数内部有三个特殊的对象: `arguments`, `this`, `new.target`。

- `arguments`: 前面出现过多次，本质上是一个数组对象，包含调用函数时传入的所有参数。这个对象只有以 `function` 关键字定义函数时有用 (箭头函数没用)。`arguments` 有一个特殊的属性: `callee`，它可以指向 `arguments` 对象所在的函数，因此可以在函数内部这样调用函数本身: `arguments.callee`(严格模式下不准这样用)。
- `this`: 在标准函数和箭头函数中不同
  - 标准函数: `this` 引用的是把函数当成方法调用的上下文对象，这时候通常称其为 `this` 值（在网页的全局上下文中调用函数时，`this` 指向 `windows`）

```
window.color = 'red';
let o = { color: 'blue' };
function sayColor() {
    console.log(this.color);
}
sayColor();   // 'red'
o.sayColor = sayColor;
o.sayColor(); // 'blue'
```

  - 在箭头函数中，`this` 引用的是定义箭头函数的上下文 (更符合多数语言中 `this` 的作用)。
- `caller`: ES5 新增的函数属性，这个属性引用的是调用当前函数的函数。
- `new.target`: ECMAScript 中的函数始终可以作为构造函数实例化一个新对象，也可以作为普通函数被调用。ECMAScript 6 新增了检测函数是否使用 `new` 关键字调用的 `new.target` 属性。如果函数是正常调用的，则 `new.target` 的值是 `undefined`; 如果是使用 new 关键字调用的，则 `new.target` 将引用被调用的构造函数。

```
function King() {
    if (!new.target) {
        throw 'King must be instantiated using "new"';
    }
    console.log('King instantiated using "new"');
}
new King(); // King instantiated using "new"
King();    // Error: King must be instantiated using "new"
```

函数有两个特殊属性:

- `length`: 保存函数定义的命名参数的个数。
- `prototype`: 保存引用类型所有实例方法的地方。

函数有两个特殊方法:

- `apply()`: 接收两个参数:函数内 `this` 的值和一个参数数组。第二个参数可以是 `Array` 的实例，也可以是 `arguments` 对象。

```
function sum(num1, num2) {
    return num1 + num2;
}
function callSum1(num1, num2) {
    return sum.apply(this, arguments);
    // 传入arguments对象
}
function callSum2(num1, num2) {
    return sum.apply(this, [num1, num2]); // 传入数组
}
console.log(callSum1(10, 10)); // 20
console.log(callSum2(10, 10)); // 20
```

- `call()`: 和 `apply()` 方法作用一样，不过第二及以后参数是单个对象，而不是数组。

8 迭代器与生成器

8.1 迭代器

迭代器基础不讲，ECMAScript 的这些内置类型实现了 `Iterable` 接口: 字符串，数组，映射，集合，`arguments` 对象, `NodeList` 等 DOM 集合类型。

如果要自定义可迭代对象，写一个 `next()` 方法即可。`next()` 方法返回的是一个对象，包含 `done` 和 `value` 两个元素，其中 `done` 为 `Boolean` 表示迭代器返回成功与否，`value` 为返回的数据。

8.2 生成器

生成器是 ES6 新增的一个极为灵活的结构，拥有在一个函数块内暂停和恢复代码执行的能力。这种新能力具有深远的影响，比如，使用生成器可以自定义迭代器和实现协程。

生成器的形式是一个函数，函数名称前面加一个星号（*）表示它是一个生成器。只要是可以定义函数的地方，就可以定义生成器。

```
// 生成器函数声明
function * generatorFn() {...}
// 生成器哈桑农户表达式
let generationFn = function*() {...}
// 作为对象字面量方法的生成器函数
let foo = {
    * generatorFn() {}
}
// 作为类实例方法的生成器函数
class Foo {
    * generatorFn() {}
}
// 作为类静态方法的生成器函数
class Bar {
    static * generatorFn() {}
}
```

注意 8.1. 箭头函数不能用来定义生成器函数。

调用生成器函数会产生一个生成器对象。生成器对象一开始处于暂停执行（suspended）的状态。与迭代器相似，生成器对象也实现了 `Iterator` 接口，因此具有 `next()` 方法。调用这个方法会让生成器开始或恢复执行。

```
function* generatorFn() {}
const g = generatorFn();
console.log(g);     // generatorFn {<suspended>}
console.log(g.next); // f next() { [native code] }
```

`yield` 关键字可以让生成器停止和开始执行，也是生成器最有用的地方。生成器函数在

遇到 `yield` 关键字之前会正常执行。遇到这个关键字后，执行会停止，函数作用域的状态会被保留。停止执行的生成器函数只能通过在生成器对象上调用 `next()` 方法来恢复执行：

`yield` 关键字有点像函数的中间返回语句，它生成的值会出现在 `next()` 方法返回的对象里。通过 `yield` 关键字退出的生成器函数会处在 `done: false` 状态；通过 `return` 关键字退出的生成器函数会处于 `done: true` 状态。

```
function* generatorFn() {
    yield 'foo'; yield 'bar';
    return 'baz';
}
let generatorObject = generatorFn();
console.log(generatorObject.next()); // { done: false, value: 'foo' }
console.log(generatorObject.next()); // { done: false, value: 'bar' }
console.log(generatorObject.next()); // { done: true, value: 'baz' }
```

生成器函数内部的执行流程会针对每个生成器对象区分作用域。在一个生成器对象上调用 `next()` 不会影响其他生成器：

```
function* generatorFn() {
    yield 'foo';
    yield 'bar';
    return 'baz';
}
let generatorObject1 = generatorFn();
let generatorObject2 = generatorFn();
console.log(generatorObject1.next()); // { done: false, value: 'foo' }
console.log(generatorObject2.next()); // { done: false, value: 'foo' }
```

建议 8.1. 由于生成器在实际开发过程中用的并不多，这里仅作基本介绍。*JS* 的生成器一定程度上参考了 *python* 的生成器，有兴趣的读者请自行了解。

9 对象，类

ECMA-262 将对象定义为一组属性的无序集合。严格来说，这意味着对象就是一组没有特定顺序的值。对象的每个属性或方法都由一个名称来标识，这个名称映射到一个值。正因为如此（以及其他还未讨论的原因），可以把 ECMAScript 的对象想象成一张散列表，其中的内容就是一组名/值对，值可以是数据或者函数。

9.1 对象

创建自定义对象的通常方式是创建 `Object` 的一个新实例，然后再给它添加属性和方法：

```
// 写法一
let person = new Object();
person.name = "Nicholas";
person.age = 29;
person.job = "Software Engineer";
person.sayName = function() {
    console.log(this.name);
};
// 写法二
let person = {
    name: "Nicholas",
    age: 29,
    job: "Software Engineer",
    sayName() {
        console.log(this.name);
    }
};
```

9.1.1 属性的类型

ECMA-262 使用一些内部特性来描述属性的特征。为了将某个特性标识为内部特性，规范会用两个中括号把特性的名称括起来，比如 `[[Enumerable]]`。

属性分为两种: 数据属性和访问器属性。

数据属性

数据属性包含一个保存数据值的位置。值会从这个位置读取，也会写入到这个位置。数据属性有 4 个特性描述它们的行为。

- `[[Configuration]]`: 表示属性是否可以通过 `delete` 删除并重新定义，是否可以修改它的特性，以及是否可以把它改为访问器属性。默认为 `true`。
- `[[Enumerate]]`: 表示属性是否可以通过 `for-in` 循环返回。默认是 `true`。
- `[[Writable]]`: 表示属性的值是否可以被修改。默认是 `true`。
- `[[Value]]`: 包含属性实际的值。默认为 `undefined`。

要修改属性的默认特性，就必须使用 Object.defineProperty() 方法。这个方法接收 3 个参数：要给其添加属性的对象、属性的名称和一个描述符对象。最后一个参数，即描述符对象上的属性可以包含：configurable、enumerable、writable 和 value，跟相关特性的名称一一对应。根据要修改的特性，可以设置其中一个或多个值。比如：

```
let person = {};
Object.defineProperty(person, "name", {
    writable: false,
    value: "Nicholas"
});
console.log(person.name); // "Nicholas"
person.name = "Greg";
console.log(person.name); // "Nicholas"
```

如果 [[Configuration]] 被设置为 false，则不可以再通过 defineProperty 配置属性。

访问器属性

访问器属性不包含数据值。相反，它们包含一个获取（getter）函数和一个设置（setter）函数，不过这两个函数不是必需的。在读取访问器属性时，会调用获取函数，这个函数的责任就是返回一个有效的值。在写入访问器属性时，会调用设置函数并传入新值，这个函数必须决定对数据做出什么修改。访问器属性有 4 个特性描述它们的行为。

- [[Configuration]]: 表示属性是否可以通过 delete 删除并重新定义，是否可以修改它的特性，以及是否可以把它改为访问器属性。默认为 true。
- [[Enumerate]]: 表示属性是否可以通过 for-in 循环返回。默认是 true。
- [[Get]]: 获取函数，在读取属性时调用。默认值为 undefined。
- [[Set]]: 设置函数，在写入属性时调用。默认值为 undefined。

访问器属性是不能直接定义的，必须使用 Object.defineProperty()。

```
let book = {
    year_: 2017,
    edition: 1
};

Object.defineProperty(book, "year", {
    get() {
        return this.year_;
    },
    set(newValue) {
        if (newValue > 2017) {
            this.year_ = newValue;
            this.edition += newValue - 2017;
        }
    }
});
book.year = 2018;
console.log(book.edition); // 2
```

同时，ECMAScript 提供了 Object.defineProperties() 方法来同时定义多个属性:

```
let book = {};
Object.defineProperties(book, {
    year_: {
        value: 2017
    },
    edition: {
        value: 1
    },
    year: {
        get() {
            return this.year_;
        },
        set(newValue) {
            if (newValue > 2017) {
                this.year_ = newValue;
                this.edition += newValue - 2017;
            }
        }
    }
});
```

使用 `Object.getOwnPropertyDescriptor()` 方法可以取得指定属性的属性描述符。这个方法接收两个参数：属性所在的对象和要取得其描述符的属性名。返回值是一个对象，对于访问器属性包含 `configurable`、`enumerable`、`get` 和 `set` 属性，对于数据属性包含 `configurable`、`enumerable`、`writable` 和 `value` 属性。

ES6 也提供了访问多个属性的方法: `Object.getOwnPropertyDescriptors()`。

9.1.2 对象操作

`Object.assign()`

ES6 专门为合并对象提供了 `Object.assign()` 方法。这个方法接收一个目标对象和一个或多个源对象作为参数，然后将每个源对象中可枚举（`Object.propertyIsEnumerable()` 返回 `true`）和自有（`Object.hasOwnProperty()` 返回 `true`）属性复制到目标对象。以字符串和符号为键的属性会被复制。对每个符合条件的属性，这个方法会使用源对象上的 `[[Get]]` 取得属性的值，然后使用目标对象上的 `[[Set]]` 设置属性的值。

如果赋值期间出错，则操作会中止并退出，同时抛出错误。`Object.assign()` 没有“回滚”之前赋值的概念，因此它是一个尽力而为、可能只会完成部分复制的方法。

对象解构

ECMAScript 6 新增了对象解构语法，可以在一条语句中使用嵌套数据实现一个或多个赋值操作。简单地说，对象解构就是使用与对象匹配的结构来实现对象属性赋值。

```
let person = {
    name: 'Matt',
    age: 27
};
let { name: personName, age: personAge } = person;
```

```
console.log(personName); // Matt
console.log(personAge); // 27
```

使用解构，可以在一个类似对象字面量的结构中，声明多个变量，同时执行多个赋值操作。如果想让变量直接使用属性的名称，那么可以使用简写语法，比如：

```
let person = {
    name: 'Matt',
    age: 27
};
let { name, age, job } = person;
console.log(name); // Matt
console.log(age); // 27
console.log(job); // undefined
```

解构在内部使用函数 `ToObject()`（不能在运行时环境中直接访问）把源数据结构转换为对象。这意味着在对象解构的上下文中，原始值会被当成对象。这也意味着（根据 `ToObject()` 的定义），`null` 和 `undefined` 不能被解构，否则会抛出错误。

解构对于引用嵌套的属性或赋值目标没有限制。为此，可以通过解构来复制对象属性：

```
let person = {
    name: 'Matt',
    age: 27,
    job: {
        title: 'Software engineer'
    }
};
let personCopy = {};

({
    name: personCopy.name,
    age: personCopy.age,
    job: personCopy.job
} = person);

person.job.title = 'Hacker'
console.log(person);     // { name: 'Matt', age: 27, job: { title: 'Hacker' } }
console.log(personCopy); // { name: 'Matt', age: 27, job: { title: 'Hacker' } }
```

9.2 创建对象

9.2.1 工厂模式鱼构造函数模式

虽然使用 `Object` 构造函数或对象字面量可以方便地创建对象，但这些方式也有明显不足：创建具有同样接口的多个对象需要重复编写很多代码。

工厂模式

```
function createPerson(name, age, job) {
```

```
    let o = new Object();
    o.name = name;
    o.age = age;
    o.job = job;
    o.sayName = function() {
        console.log(this.name);
    };
    return o;
}
let person1 = createPerson("Nicholas", 29, "Software Engineer");
let person2 = createPerson("Greg", 27, "Doctor");
```

这里，函数 `createPerson()` 接收 3 个参数，根据这几个参数构建了一个包含 `Person` 信息的对象。可以用不同的参数多次调用这个函数，每次都会返回包含 3 个属性和 1 个方法的对象。这种工厂模式虽然可以解决创建多个类似对象的问题，但没有解决对象标识问题（即新创建的对象是什么类型）。

构造函数模式

前面的例子使用构造函数模式可以这样写：

```
function Person(name, age, job){
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName = function() {
        console.log(this.name);
    };
}
let person1 = new Person("Nicholas", 29, "Software Engineer");
let person2 = new Person("Greg", 27, "Doctor");
person1.sayName(); // Nicholas
person2.sayName(); // Greg
```

在这个例子中，`Person()` 构造函数代替了 `createPerson()` 工厂函数。实际上，`Person()` 内部的代码跟 `createPerson()` 基本是一样的，只是有如下区别。

- 没有显式地创建对象。
- 属性和方法直接赋值给了 `this`。
- 没有 `return`。

另外，要注意函数名 `Person` 的首字母大写了。按照惯例，构造函数名称的首字母都是要大写的，非构造函数则以小写字母开头。这是从面向对象编程语言那里借鉴的，有助于在 ECMAScript 中区分构造函数和普通函数。毕竟 ECMAScript 的构造函数就是能创建对象的函数。

要创建 `Person` 的实例，应使用 `new` 操作符。以这种方式调用构造函数会执行如下操作。

- 在内存中创建一个新对象。
- 这个新对象内部的 `[[Prototype]]` 特性被赋值为构造函数的 `prototype` 属性。
- 构造函数内部的 `this` 被赋值为这个新对象（即 `this` 指向新对象）。

- 执行构造函数内部的代码（给新对象添加属性）。
- 如果构造函数返回非空对象，则返回该对象；否则，返回刚创建的新对象。

上一个例子的最后，person1 和 person2 分别保存着 Person 的不同实例。这两个对象都有一个 constructor 属性指向 Person，如下所示：

```
console.log(person1.constructor == Person); // true
console.log(person2.constructor == Person); // true
```

在实例化时，如果不想传参数，那么构造函数后面的括号可加可不加。只要有 new 操作符，就可以调用相应的构造函数。

构造函数与普通函数唯一的区别就是调用方式不同。除此之外，构造函数也是函数。并没有把某个函数定义为构造函数的特殊语法。任何函数只要使用 new 操作符调用就是构造函数，而不使用 new 操作符调用的函数就是普通函数。

9.2.2 原型模式

每个函数都会创建一个 prototype 属性，这个属性是一个对象，包含应该由特定引用类型的实例共享的属性和方法。实际上，这个对象就是通过调用构造函数创建的对象的原型。使用原型对象的好处是，在它上面定义的属性和方法可以被对象实例共享。原来在构造函数中直接赋给对象实例的值，可以直接赋值给它们的原型，如下所示：

```
function Person() {}
Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
    console.log(this.name);
};
let person1 = new Person();
person1.sayName(); // "Nicholas"
let person2 = new Person();
person2.sayName(); // "Nicholas"
console.log(person1.sayName == person2.sayName); // true
```

理解原型

无论何时，只要创建一个函数，就会按照特定的规则为这个函数创建一个 prototype 属性（指向原型对象）。默认情况下，所有原型对象自动获得一个名为 constructor 的属性，指回与之关联的构造函数。因构造函数而异，可能会给原型对象添加其他属性和方法。

每次调用构造函数创建一个新实例，这个实例的内部 [[Prototype]] 指针就会被赋值为构造函数的原型对象。实例与构造函数原型之间有直接的联系，但实例与构造函数之间没有。

在 JavaScript 内部，和实例对象相关的有三个对象:

- 构造函数: 我们定义的，一般为首字母大写的的函数。包含如下成员:
  - prototype: 指向函数原型。

- 函数原型: 我们定义构造函数后，自动生成的，用于存储数据。包含如下成员:
  - `constructor`: 指向构造函数。
  - 构造函数定义的成员。
- 实例对象: 通过构造函数创建的实例对象，保存一个 `[[prototype]]` 属性指向函数原型。可以通过 `__proto__` 属性访问。

也就是说，我们在创建一个函数时，JavaScript 内部会创建这个函数对应的原型，函数本身只保存指向这个原型的指针，通过函数创建对象实例时本质上时通过原型创建实例。

原型的问题

原型模式弱化了向构造函数传递初始化参数的能力，会导致所有实例默认都取得相同的属性值。虽然这会带来不便，但还不是原型的最大问题。原型的最主要问题源自它的共享特性。

这对于函数来说没有什么问题，函数一般不会也不建议保存数据。而对于属性来说，则要注意不要轻易修改原型中的属性，这会导致所有使用该原型的实例对象保存的属性均被改变。

9.3 继承

9.3.1 原型链继承

我们知道，ECMAScript 是没有类的，所有对象均源自原生的 `Object` 类型。本质上，ECMAScript 的核心设计思想: 原型代替了传统语言的类与继承思想。这两者在具体实现上不同，但实现的效果类似。简而言之，ECMAScript 通过原型链实现类的继承，并且用法类似。

默认情况下，所有引用对象都继承自 `Object`，因此自定义构造函数的原型也指向 `Object` 的 `Prototype`。

确认继承关系有两种方式，一种是通过原生的 `instanceof` 操作符，另一种时通过原型的 `isPrototypeOf` 方法。

由于没有类，要通过原型实现继承，因此需要手动指定 `prototype` 来实现继承:

```
function SuperType() {
    this.property = true;
}
SuperType.prototype.getSuperValue = function() {
    return this.property;
};
function SubType() {
    this.subproperty = false;
}

// 继承SuperType
SubType.prototype = new SuperType();

// 新方法
```

```
SubType.prototype.getSubValue = function () {
    return this.subproperty;
};

// 覆盖已有的方法
SubType.prototype.getSuperValue = function () {
    return false;
};
let instance = new SubType();
console.log(instance.getSuperValue()); // false
```

9.3.2 经典继承

这样继承存在一个问题，原型包含引用值，对原型引用值得修改将影响所有与之关联的对象。为了解决这个问题，引入了“盗用构造函数”(也称“经典继承”)。

```
function SuperType() {
    this.colors = ["red", "blue", "green"];
}

function SubType() {
    // 继承SuperType
    SuperType.call(this);
}

let instance1 = new SubType();
instance1.colors.push("black");
console.log(instance1.colors); // "red,blue,green,black"
let instance2 = new SubType();
console.log(instance2.colors); // "red,blue,green"
```

通过使用 `call()`（或 `apply()`）方法，`SuperType` 构造函数在为 `SubType` 的实例创建的新对象的上下文中执行了。这相当于新的 `SubType` 对象上运行了 `SuperType()` 函数中的所有初始化代码 (不包括函数)。结果就是每个实例都会有自己的 `colors` 属性。

如果需要传参，在 `call()` 的 `this` 变量后传入其他参数即可。

盗用构造函数的主要缺点，也是使用构造函数模式自定义类型的问题：必须在构造函数中定义方法，因此函数不能重用。此外，子类也不能访问父类原型上定义的方法，因此所有类型只能使用构造函数模式。由于存在这些问题，盗用构造函数基本上也不能单独使用。

9.3.3 组合继承

组合继承（有时候也叫伪经典继承）综合了原型链和盗用构造函数，将两者的优点集中了起来。基本的思路是使用原型链继承原型上的属性和方法，而通过盗用构造函数继承实例属性。这样既可以把方法定义在原型上以实现重用，又可以让每个实例都有自己的属性。来看下面的例子：

```
function SuperType (name){
```

```
    this.name = name;
    this.colors ["red","blue","green"];
}

SuperType.prototype.sayName = function(){
    console.log(this.name);
};

function SubType(name,age){
    //继承属性
    SuperType.call(this,name);
    this.age = age;
}

//继承方法
SubType.prototype = new SuperType();
SubType.prototype.sayAge = function () {
    console.log(this.age);
}
```

调用 `call()` 方法本质上时将成员属性都执行一遍，因此 SubType 有了父类的所有属性，而 `SubType.prototype = new SuperType();` 则表示，原型链上一级为 SuperType 的原型。由于我们已经相当于写过一边成员属性，所以调用时会优先使用 SubType 的成员属性。

这是目前使用最广泛的方法。

9.4 类

从前一节可以看出，ECMAScript 通过原型实现对象继承功能写出的代码比较冗余。在 Java,Python 中简单且常用的继承在 JS 中一点都不方便。为此，ES6 提供了正宗的 `class` 关键字用于定义类，虽然本质上还是原型链。

ECMAScript 中的类与函数类似，但有以下不同:

- 无法提升。
- 作用域是块作用域，而不是函数作用域。

同样可以用两种方式创建类:

```
class Person {};
const Animal = class {};
```

构造函数

类中的 `constructor` 函数是类的构造函数，解释器在调用 `new` 操作时会调用这个函数。构造函数不是必要的，不定义的话构造函数就是空函数。

使用 `new` 调用类的构造函数会执行如下操作:

- 在内存中创建一个新对象。

- 这个新对象内部的 `[[Prototype]]` 指针被赋值为构造函数的 `prototype` 属性。
- 构造函数内部的 `this` 被赋值为这个新对象（即 `this` 指向新对象）。
- 执行构造函数内部的代码（给新对象添加属性）。
- 如果构造函数返回非空对象，则返回该对象；否则，返回刚创建的新对象。

类实例化时传入的参数会用作构造函数的参数。如果不需要参数，则类名后面的括号也是可选的：

```
class Person {
    constructor(name) {
        console.log(arguments.length);
        this.name = name || null;
    }
}
let p1 = new Person;          // 0
console.log(p1.name);         // null
let p2 = new Person();        // 0
console.log(p2.name);         // null
let p3 = new Person('Jake'); // 1
console.log(p3.name);         // Jake
```

类构造函数与构造函数的主要区别是，调用类构造函数必须使用 `new` 操作符。而普通构造函数如果不使用 `new` 调用，那么就会以全局的 `this`（通常是 `window`）作为内部对象。调用类构造函数时如果忘了使用 `new` 则会抛出错误：

类不同于函数

类本质上还是函数，可以使用函数相关的所有操作，但是类内部的成员则有所不同。

每次通过 new 调用类标识符时，都会执行类构造函数。在这个函数内部，可以为新创建的实例（`this`）添加“自有”属性。至于添加什么样的属性，则没有限制。另外，在构造函数执行完毕后，仍然可以给实例继续添加新成员。每个实例都对应一个唯一的成员对象，这意味着所有成员都不会在原型上共享。

但是，也可以通过类定义语句把类块中定义的方法作为原型方法，实现实例间共享:

```
class Person {
    constructor() {   // 添加到this的所有内容都会存在于不同的实例上
        this.locate = () => console.log('instance');
    }
    // 在类块中定义的所有内容都会定义在类的原型上
    locate() { console.log('prototype'); }
} let p = new Person();
p.locate();               // instance
Person.prototype.locate(); // prototype
```

类中可以有静态方法，但每个类只能有一个静态成员:

```
class Person {
    constructor() {   // 添加到this的所有内容都会存在于不同的实例上
        this.locate = () => console.log('instance');
    }
```

```
    // 在类块中定义的所有内容都会定义在类的原型上
    locate() {
        console.log('prototype');
    }
    // 定义在类本身上
    static locate() {
        console.log('class', this);
    }
}
```

此外，也可以像函数一样在类定义外增加函数:

```
// 在类上定义独享的函数
Person.greeting = 'My name is';
// 在原型上定义共享的函数
Person.prototype.name = 'Jake';
```

继承

有类自然有继承，ES6 同时提供了 `extends` 关键字用于类继承:

- ES6 规定 JS 是单继承。
- 提供了 `super` 方法:
  - 仅限于派生类构造方法和静态方法内部调用。
  - 在构造函数中使用 `super` 用于调用父类构造函数，将实例返回给 `this`；且 `this` 必须在 `super` 之后调用。如果没有定义构造函数，在实例化派生类时会自动调用。

抽象基类

有时候可能需要定义这样一个类，它可供其他类继承，但本身不会被实例化。虽然 ECMAScript 没有专门支持这种类的语法，但通过 `new.target` 也很容易实现。`new.target` 保存通过 `new` 关键字调用的类或函数。

```
class Vehicle {
    constructor() {
        console.log(new.target);
        if (new.target === Vehicle) {
            throw new Error('Vehicle cannot be directly instantiated');
        }
    }
}
```

第二部分

WEB API

IV BOM 与 DOM 基础

10 BOM

BOM(Browser Object Model) 是使用 JavaScript 开发 Web 应用程序的核心。BOM 提供了与网页无关的浏览器功能对象。很不幸的是，多年以来 BOM 是在缺乏规范的背景下发展起来的，许多浏览器开发商有着自己的规范，因此 BOM 像 ECMAScript 一样灵活 (乱)。

10.1 window 对象

BOM 的核心是 window 对象，表示浏览器的实例。window 对象在浏览器中有两重身份，一个是 ECMAScript 中的 Global 对象 (window 代理 Global 对象)，另一个就是浏览器窗口的 JavaScript 接口。

10.1.1 操作 window 对象

Global 作用域

因为 window 对象被复用为 ECMAScript 的 Global 对象，所以通过 var 声明的所有全局变量和函数都会变成 window 对象的属性和方法。

```
var age = 29;
alert(window.age); // 29
```

如果将 var 换成 let 则不会将变量添加给全局对象。前面已经讲过。

窗口关系

top 对象始终指向最上层 (外层) 窗口，即浏览器窗口本身。而 parent 对象则始终指向当前窗口的父窗口。

还有一个 self 对象，他是终极 window 属性，始终指向 window。实际上 self 和 window 就是同一个对象。

这些都是 window 对象的属性，因此访问 window.parent, window.top 和 window.self 都可以。这意味着可以把访问多个窗口的 window 对象串联起来。比如 window.parent.parent。

窗口位置与像素化

window 对象的位置可以通过不同的属性个方法来确定。现代浏览器提供了 parentLeft 和 screenTop 属性，用于表示窗口相对于屏幕左侧和顶部的位置，返回单位是 CSS 像素[1]。

可以使用 moveTo() 和 moveBy() 方法移动窗口。这两个方法都接收两个参数,其中 moveTo() 接收要移动到的新位置的绝对坐标 x 和 y。而 moveBy() 则接受相对当前位置在两个方向上移动的像素数 (相对位置)。

窗口大小

所有现代浏览器都支持 4 个属性：

- 返回浏览器窗口自身大小: outerWidth, outerHeight。
- 返回浏览器窗口中页面大小: innerWidth, innerHeight。

document.documentElement.clientWidth 和 document.documentElement.clientHeight 返回页面视图的宽度和高度。

浏览器窗口自身的精确尺寸不好确定，但可以确定页面视口的大小，如下所示：

```
let pageWidth = window.innerWidth,
    pageHeight = window.innerHeight;

// 如果单位不是数值(物理像素)
if (typeof pageWidth != "number") {
    // 如果是 CSS 像素，返回页面高宽
    if (document.compatMode == "CSS1Compat") {
        pageWidth = document.documentElement.clientWidth;
        pageHeight = document.documentElement.clientHeight;
    // 如果不是 CSS 像素，返回 body 元素高宽
    } else {
        pageWidth = document.body.clientWidth;
        pageHeight = document.body.clientHeight;
    }
}
```

此外，还有两个方法用于缩放窗口大小:

- window.resizeTo(x,y): 缩放到绝对大小。
- window.resizeBy(x,y): 相对缩放。

大部分浏览器都支持上诉方法，但部分浏览器则会禁用。

窗口位置

浏览器窗口尺寸通常无法满足完整显示整个页面，为此用户可以通过滚动在有限的视口中查看文档。相关的方法是 window.scroll 系方法。用法和 window.resize 类似。

[1]CSS 像素参考博客: https://blog.csdn.net/u014465934/article/details/97040694

定时器

JavaScript 在浏览器中是单线程执行的，但允许使用定时器指定在某个时间之后或每隔一段时间就执行相应的代码。`setTimeout()` 用于指定在一定时间后执行某些代码，而 `setInterval()` 用于指定每隔一段时间执行某些代码。

- `setTimeout()`: 接收两个参数：要执行的代码和在执行回调函数前等待的时间（毫秒），返回一个表示该超时排期的数值 ID。

```
let timeoutId = setTimeout(() => alert("Hello world!"), 1000);
clearTimeout(timeoutId); // 取消超时任务
```

- `setInterval()`: 接收两个参数：要执行的代码（字符串或函数），以及把下一次执行定时代码的任务添加到队列要等待的时间（毫秒）。同样返回一个 ID。

10.1.2 新窗口

`window.open()` 可以用于导航到指定 URL，也可以用于打开新浏览器窗口。这个方法接收 4 个参数：要加载的 URL、目标窗口、特性字符串和表示新窗口在浏览器历史记录中是否替代当前加载页面的布尔值。通常，调用这个方法时只传前 3 个参数，最后一个参数只有在不打开新窗口时才会使用:

```
// 与<a href="http://www.wrox.com" target="topFrame"/>相同
window.open("http://www.wrox.com/", "topFrame");
```

特征字符串是一个逗号分隔的设置字符串，用于指定新窗口包含的特性。下表列出了一些选项:

表 4.1 特征字符串

设置	值	说明
fullscreen	“yes”|“no”	新窗口是否最大化，仅“IE”支持
height	数值	新窗口高度，值不能小于 100
width	数值	新窗口宽度，值不能小于 100
left	数值	新窗口 x 坐标，不能为负
top	数值	新窗口 y 坐标，不能为负
location	“yes”|“no”	是否显示地址栏，不同显示器默认值不一样
Menubar	“yes”|“no”	表示是否显示菜单栏。默认为 “no”
resizable	“yes”|“no”	表示是否可以拖动改变新窗口大小。默认为 “no”
scrollbars	“yes”|“no”	表示是否可以在内容过长时滚动。默认为 “no”
status	“yes”|“no”	表示是否显示状态栏。不同浏览器的默认值也不一样
toolbar	“yes”|“no”	表示是否显示工具栏。默认为”no”

这些设置需要以逗号分隔的名值对形式出现，其中名值对以等号连接。

```
let wroxWin = window.open("http://www.wrox.com/",
```

```
    "wroxWindow",
    "height=400,width=400,top=10,left=10,resizable=yes");
```

`window.open()` 方法返回一个对新建窗口的引用。这个对象与普通 window 对象没有区别，只是为控制新窗口提供了方便。

新创建窗口的 `window` 对象有一个属性 `opener`，指向打开它的窗口。这个属性只在弹出窗口的最上层 `window` 对象（`top`）有定义，是指向调用 `window.open()` 打开它的窗口或窗格的指针。

```
alert(wroxWin.opener === window); // true
```

虽然新建窗口中有指向打开它的窗口的指针，但反之则不然。窗口不会跟踪记录自己打开的新窗口，因此开发者需要自己记录。

在某些浏览器中，每个标签页会运行在独立的进程中。如果一个标签页打开了另一个，而 window 对象需要跟另一个标签页通信，那么标签便不能运行在独立的进程中。在这些浏览器中，可以将新打开的标签页的 opener 属性设置为 null，表示新打开的标签页可以运行在独立的进程中。比如：

```
wroxWin.opener = null;
```

把 opener 设置为 null 表示新打开的标签页不需要与打开它的标签页通信，因此可以在独立进程中运行。这个连接一旦切断，就无法恢复了。

安全限制

由于弹出窗口被在线广告用烂了，因此很多浏览器会屏蔽弹出窗口，为此我们需要检测新窗口是否被正确弹出，最简单的方法是判断返回值:

```
let wroxWin = window.open("http://www.wrox.com", "_blank");
if (wroxWin == null){
    alert("The popup was blocked!");
}
```

但有的浏览器会让 `window.open()` 抛出错误，更好的方法是使用 `try catch` 语句:

```
let blocked = false; try {
    let wroxWin = window.open("http://www.wrox.com", "_blank");
    if (wroxWin == null) {
        blocked = true;
    }
} catch (ex) {
    blocked = true;
} if (blocked) {
    alert("The popup was blocked!");
}
```

系统对话框

使用 `alert()`、`confirm()` 和 `prompt()` 方法，可以让浏览器调用系统对话框向用户显示消息。这些对话框与浏览器中显示的网页无关，而且也不包含 HTML。它们的外观由操作系统或者浏览器决定，无法使用 CSS 设置。此外，这些对话框都是同步的模态对话框，即在它们显示的时候，代码会停止执行，在它们消失以后，代码才会恢复执行。

- `alert()`: 接收一个要显示给用户的字符串。传入的字符串会显示在一个系统对话框中。对话框只有一个“OK”（确定）按钮。如果传给 `alert()` 的参数不是一个原始字符串，则会调用这个值的 `toString()` 方法将其转换为字符串。
- `confirm()`: 和 `alert()` 类似，但会有 `Cancel` 和 `OK` 两个按钮。通过返回的 `boolean` 值判断用户按下了哪个按钮。
- `prompt()`: 在 `confirm()` 的基础上，增加了文本框，用于提示用户输入信息。
  - `prompt()` 有两个参数: 显示给用户的信息，文本框默认值。
  - 如果用户按下 `OK` 按钮，则返回文本框的内容；否则返回 `null`。

JavaScript 还可以显示另外两种对话框：`find()` 和 `print()`。这两种对话框都是异步显示的，即控制权会立即返回给脚本。用户在浏览器菜单上选择“查找”（find）和“打印”（print）时显示的就是这两种对话框。通过在 window 对象上调用 `find()` 和 `print()` 可以显示它们:

```
window.print();
window.find();
```

10.2 其他 BOM 对象

`location` 对象

`location` 是最有用的 BOM 对象之一，提供了当前窗口中加载文档的信息，以及通常的导航功能。这个对象独特的地方在于，它既是 `window` 的属性，也是 `document` 的属性。也就是说，`window.location` 和 `document.location` 指向同一个对象。

`location` 对象不仅保存着当前加载文档的信息，也保存着把 URL 解析为离散片段后能够通过属性访问的信息。比如:

- `location.host`: 服务器名以及端口号。
- `location.hostname`: 服务器名。
- `location.search`: URL 的查询字符串。这个字符串以问号开头。

查询字符串

虽然 `location.search` 能获取 URL 的查询字符串，但是 `location` 并没有日工获取单个查询内容的接口 (也提供不了)。如果我们要解析查询内容，则需要手动编写函数处理 `location.search` 返回的字符串。

URLSearchParams 提供了一组标准 API 方法，通过它们可以检查和修改查询字符串。

```
let qs = "?q=javascript&num=10";
let searchParams = new URLSearchParams(qs);
alert(searchParams.toString()); // " q=javascript&num=10"
searchParams.has("num");      // true
searchParams.get("num");      // 10
searchParams.set("page", "3");
alert(searchParams.toString()); // " q=javascript&num=10&page=3"
searchParams.delete("q");
alert(searchParams.toString()); // " num=10&page=3"
```

大多数支持 URLSearchParams 的浏览器也支持将 URLSearchParams 的实例用作可迭代对象：

```
let qs = "?q=javascript&num=10";
let searchParams = new URLSearchParams(qs);
for (let param of searchParams) {
    console.log(param);
}
// ["q", "javascript"]
// ["num", "10"]
```

修改地址

可以通过修改 location 对象修改浏览器的地址。首先，最常见的是使用 assign() 方法并传入一个 URL，如下所示：

```
location.assign("http://www.wrox.com");
```

这行代码会立即启动导航到新 URL 的操作，同时在浏览器历史记录中增加一条记录。如果给 location.href 或 window.location 设置一个 URL，也会以同一个 URL 值调用 assign() 方法。比如，下面两行代码都会执行与显式调用 assign() 一样的操作：

```
window.location = "http://www.wrox.com";
location.href = "http://www.wrox.com"; // 最常见
```

修改 location 对象的其他属性也会修改当前加载的页面。

在以前面提到的方式修改 URL 之后，浏览器历史记录中就会增加相应的记录。如果不希望增加历史记录，可以使用 replace() 方法。这个方法接收一个 URL 参数，但重新加载后不会增加历史记录。调用 replace() 之后，用户不能回到前一页。

最后一个修改地址的方法是 reload()，它能重新加载当前显示的页面。调用 reload() 而不传参数，页面会以最有效的方式重新加载。也就是说，如果页面自上次请求以来没有修改过，浏览器可能会从缓存中加载页面。如果想强制从服务器重新加载，可以给 reload() 传个 true。

脚本中位于 reload() 调用之后的代码可能执行也可能不执行，这取决于网络延迟和系统资源等因素。为此，最好把 reload() 作为最后一行代码。

navigator 对象

navigator 对象的属性通常用于确定浏览器的类型，表示用户代理的状态和标识。举几个常用的属性/方法:

- appName: 浏览器全名。
- cookieEnabled: 返回布尔值，表示是否启用了 cookie。
- deviceMemory: 返回单位为 GB 的设备内存容量。
- language: 返回浏览器的主语言。
- mediaDevices: 返回可用的媒体设备。
- plugins: 返回浏览器安装的插件数组。

最常见的用法是检查浏览器插件，plugins 数组包含以下内容:

- name: 插件名称。
- description: 插件介绍。
- filename: 插件的文件名。
- length: 由当前插件处理的 MIME 类型数量。

screen 对象

screen 对象很少用，这个对象中保存的纯粹是客户端能力信息，也就是浏览器窗口外面的客户端显示器的信息，比如像素宽度和像素高度。举几个例子:

- availHeight: 屏幕像素高度减去系统组件高度（只读）。
- colorDepth: 表示屏幕颜色的位数；多数系统是 32（只读）。
- height: 屏幕像素高度。

history 对象

history 对象表示当前窗口首次使用以来用户的导航历史记录。因为 history 是 window 的属性，所以每个 window 都有自己的 history 对象。

go() 方法可以在用户历史记录中沿任何方向导航，可以前进也可以后退。这个方法只接收一个参数，这个参数可以是一个整数，表示前进或后退多少步。

```
// 后退一页
history.go(-1);
// 前进一页
history.go(1);
// 前进两页
history.go(2);
```

go() 有两个简写方法：back() 和 forward()。顾名思义，这两个方法模拟了浏览器的后退按钮和前进按钮。

history 对象还有一个 length 属性，表示历史记录中有多个条目。

11 DOM

文档对象模型（DOM，Document Object Model）是 HTML 和 XML 文档的编程接口。DOM 表示由多层节点构成的文档，通过它开发者可以添加、删除和修改页面的各个部分。

本章与下一章将说明 DOM 最基础也是核心的用法，随着 DOM 的发展，新增了更多 DOM 相关的 API，不过掌握了核心用法，熟悉其他 API 也只是时间问题。

11.1 节点层级

任何 HTML 或 XML 文档都可以用 DOM 表示为一个由节点构成的层级结构。节点分很多类型，每种类型对应着文档中不同的信息和（或）标记，也都有自己不同的特性、数据和方法，而且与其他类型有某种关系。这些关系构成了层级，让标记可以表示为一个以特定节点为根的树形结构。

`document` 节点表示每个文档的根节点。一般的 HTML 界面中 `document` 节点的唯一子节点是 `<html>` 元素，也叫文档元素。。文档元素是文档最外层的元素，所有其他元素都存在于这个元素之内。每个文档只能有一个文档元素。

HTML 中的每段标记都可以表示为这个树形结构中的一个节点。元素节点表示 HTML 元素，属性节点表示属性，文档类型节点表示文档类型，注释节点表示注释。DOM 中总共有 12 种节点类型，这些类型都继承一种基本类型。

11.1.1 `Node` 类型

`Node` 接口时所有 DOM 节点类型都必须实现的。在 JavaScript 中，所有节点类型都继承 `Node` 类型，因此所有类型都共享相同的基本属性和方法。

每个节点都有 `nodeType` 属性，表示该节点的类型。节点类型由定义在 `Node` 类型上的 12 个数值常量表示：

- `Node.ELEMENT_NODE`: 1
- `Node.ATTRIBUTE_NODE`: 2
- `Node.TEXT_NODE`: 3
- `Node.CDATA_SECTION_NODE`: 4
- `Node.ENTITY_REFERENCE_NODE`: 5
- `Node.ENTITY_NODE`: 6
- `Node.PROCESSING_INSTRUCTION_NODE`: 7
- `Node.COMMENT_NODE`: 8
- `Node.DOCUMENT_NODE`: 9
- `Node.DOCUMENT_TYPE_NODE`: 10
- `Node.DOCUMENT_FRAGMENT_NODE`: 11
- `Node.NOTATION_NODE`: 12

并不是所有的浏览器都支持全部的节点，常用的节点类型时元素节点和文本节点。

此外，节点还有两个属性: nodeName 和 nodeValue。这两个属性的值完全取决于节点本身。

每一个节点都有一个 childNodes 属性，包含一个 NodeList 实例。NodeList 是一个类数组对象，用于存储可以按位置存取的有序节点。NodeList 的很多操作都类似于 Array，比如可以通过中括号取值，但 NodeList 并不是 Array。

NodeList 对象独特的地方在于，它其实是一个对 DOM 结构的查询，因此 DOM 结构的变化会自动地在 NodeList 中反映出来。我们通常说 NodeList 是实时的活动对象，而不是第一次访问时所获得内容的快照。

每个节点都有一个 parentNode 属性，指向其 DOM 树中的父元素。此外，childNodes 列表中的每个节点都是同一列表中其他节点的同胞节点。而使用 previousSibling 和 nextSibling 可以在这个列表的节点间导航。这个列表中第一个节点的 previousSibling 属性是 null，最后一个节点的 nextSibling 属性也是 null。

父节点和它的第一个及最后一个子节点也有专门属性：firstChild 和 lastChild 分别指向 childNodes 中的第一个和最后一个子节点。firstChild 的值始终等于 childNodes[0]。

最后还有一个所有节点都共享的关系。ownerDocument 属性是一个指向代表整个文档的文档节点的指针。

操纵节点

因为所有关系指针都是只读的，所以 DOM 又提供了一些操纵节点的方法。

最常见的是 appendChild() 用于在 childNodes 列表末尾添加节点。添加新节点会更新相关的关系指针，appendChild() 方法返回新添加的节点。

如果把文档中已经存在的节点传给 appendChild()，则这个节点会从之前的位置被转移到新位置。即使 DOM 树通过各种关系指针维系，一个节点也不会在文档中同时出现在两个或更多个地方。因此，如果调用 appendChild() 传入父元素的第一个子节点，则这个节点会成为父元素的最后一个子节点，可以这样改变 DOM 树的结构。

类似的，还有 insertBefore() 方法，接收两个参数：要插入的节点和参照节点。调用这个方法后，要插入的节点会变成参照节点的前一个同胞节点，参照节点可以为 null 表示插入为最后一个节点。

前面两个方法都不会删除任何节点，replaceChild() 方法接受两个参数: 要插入的节点和要替换的节点。要替换的节点会被返回并从文档树中完全移除，要插入的节点会取而代之。

要移除节点而不是替换节点，可以使用 removeChild() 方法。这个方法接收一个参数，即要移除的节点。被移除的节点会被返回.

此外还有两个所有节点共享的方法:

- cloneNode(): 会返回与调用它的节点一模一样的节点。有一个布尔值参数，如果为 true 则返回深复制，否则返回节点本身。复制返回的节点属于文档所有，但尚未指定

父节点，所以可称为孤儿节点（orphan）。

• normalize(): 处理文档子树中的文本节点。由于解析器实现的差异或 DOM 操作等原因，可能会出现并不包含文本的文本节点，或者文本节点之间互为同胞关系。在节点上调用 normalize() 方法会检测这个节点的所有后代，从中搜索上述两种情形。如果发现空文本节点，则将其删除；如果两个同胞节点是相邻的，则将其合并为一个文本节点。

11.1.2 Document 类型

Document 类型是 JavaScript 中表示文档节点的类型。在浏览器中，文档对象 document 是 HTMLDocument 的实例（HTMLDocument 继承 Document），表示整个 HTML 页面。document 是 window 对象的属性，因此是一个全局对象。Document 类型的节点有以下特征：

• nodeType 为 Node.DOCUMENT_NODE;

• nodeName 值为 ``#document'';

• nodeValue 值为 null;

• parentNode 值为 null;

• ownerDocument 值为 null;

• 子节点可以是 DocumentType（最多一个）、Element（最多一个）、ProcessingInstruction 或 Comment 类型。

DOM 为 Document 提供了两个访问子节点的快捷方式。第一个是 documentElement 属性，始终指向 HTML 页面中的 <html> 元素。第二个是 body 属性，直接指向 <body> 元素。因为这个元素是开发者使用最多的元素。Document 类型另一种可能的子节点是 DocumentType。<!doctype> 标签是文档中独立的部分。

文档信息

document 作为 HTMLDocument 的实例，还有一些标准 Document 对象上所没有的属性。这些属性提供浏览器所加载网页的信息。

• title: 获取 <title> 元素中的文本，可以读写，但是修改 title 属性并不会改变 <title> 元素。

• URL: 当前页面的完整 URL。

• domain: 包含页面的域名。

• reference: 包含链接到当前页面的那个页面的 URL。

其中 domain 是可以设置的，当页面中包含来自某个不同子域的窗格（<frame>）或内嵌窗格（<iframe>）时，设置 document.domain 是有用的。因为跨源通信存在安全隐患，所以不同子域的页面间无法通过 JavaScript 通信。此时，在每个页面上把 document.domain 设置为相同的值，这些页面就可以访问对方的 JavaScript 对象了。

domain 属性的设置是有限制的，只能基于当前域设置，且放松后不能再收紧:

```
// 页面来自p2p.wrox.com
document.domain = "wrox.com";  // 放松，成功
document.domain = "p2p.wrox.com"; // 收紧，错误！
```

定位元素

使用 DOM 最常见的情形可能就是获取某个或某组元素的引用，然后对它们执行某些操作。为此 Document 提供了三个方法:

- getElementById(): 接收一个参数，要获取元素的 ID, ID 区分大小写，多个则只返回第一个; 如果找到了则返回这个元素，如果没找到则返回 null。
- getElementsByTagName(): 接收一个参数，即要获取元素的标签名，返回包含零个或多个元素的 NodeList，标签名可以是 “*” 表示所有。在 HTML 文档中，这个方法返回一个 HTMLCollection 对象。
- getElementsByName(): 这个方法会返回具有给定 name 属性的所有元素，这个方法返回一个 HTMLCollection 对象。getElementsByName() 方法最常用于单选按钮，因为同一字段的单选按钮必须具有相同的 name 属性才能确保把正确的值发送给服务器。

HTMLCollection 对象还有一个额外的方法 namedItem()，可通过标签的 name 属性取得某一项的引用，当然也可以通过中括号获取。

```
<img src="myimage.gif" name="myImage">
let myImage = images.namedItem("myImage");
let myImage = images["myImage"];
```

特殊集合

document 对象上还暴露了几个特殊集合，这些集合也都是 HTMLCollection 的实例。这些集合是访问文档中公共部分的快捷方式，列举如下:

- document.anchors: 包含文档中所有带 name 属性的 <a> 元素。
- document.applets: 包含文档中所有 <applet> 元素（因为 <applet> 元素已经不建议使用，所以这个集合已经废弃）。
- document.forms: 包含文档中所有 <form> 元素。
- document.images: 包含文档中所有 <img> 元素（。
- document.links: 包含文档中所有带 href 属性的 <a> 元素。

11.1.3 Element 类型

除了 Document 类型，Element 类型就是 Web 开发中最常用的类型了。Element 表示 XML 或 HTML 元素，对外暴露出访问元素标签名、子节点和属性的能力。Element 类型的节点具有以下特征：

- nodeType 为 Node.ELEMENT_NODE;

- nodeName 值为元素的标签名;
- nodeValue 值为 null;
- parentNode 值为 Document 和 Element 对象;
- 子节点可以是 Element、Text、Comment、ProcessingInstruction、CDATASection、EntityReference 类型。

可以通过 nodeName 或 tagName 属性来获取元素的标签名，这两个属性返回同样的值。在 HTML 中，返回的值是标签名的大写，如 DIV，而 XML 中则返回相同的原标签名。

HTML 的节点包含的诸多属性如 class，title，id 等都可以用同名的 Element 变量获得并修改，同时也提供了以下方法来操作属性:

- getAttribute(): 通过属性名获取属性，属性名不区分大小写。
- setAttribute(): 设置属性值。
- removeAttribute(): 删除属性，会直接移除属性，用的很少。

Element 类型是唯一使用 attributes 属性的 DOM 节点类型。attributes 属性包含一个 NamedNodeMap 实例，是一个类似 NodeList 的“实时”集合。元素的每个属性都表示为一个 Attr 节点，并保存在这个 NamedNodeMap 对象中。NamedNodeMap 对象包含下列方法：

- getNamedItem(name): 返回 nodeName 属性等于 name 的节点;
- removeNamedItem(name)，删除 nodeName 属性等于 name 的节点;
- setNamedItem(node)，向列表中添加 node 节点，以其 nodeName 为索引;
- item(pos)，返回索引位置 pos 处的节点。

创建元素

可以使用 document.createElement() 方法创建新元素。这个方法接收一个擦拭农户，即要创建元素的标签名。使用 createElement() 方法创建新元素的同时也会将其 ownerDocument 属性设置为 document。此时，可以再为其添加属性、添加更多子元素:

```
let div = document.createElement("div");
div.id = "myNewDiv";
div.className = "box";
```

在新元素上设置这些属性只会附加信息。因为这个元素还没有添加到文档树，所以不会影响浏览器显示。

元素可以拥有任意多个子元素和后代元素，因为元素本身也可以是其他元素的子元素。childNodes 属性包含元素所有的子节点，这些子节点可能是其他元素、文本节点、注释或处理指令。不同浏览器在识别这些节点时的表现有明显不同。比如下面的代码：

```
<ul id="myList">
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ul>
```

在解析以上代码时， 元素会包含 7 个子元素，其中 3 个是 <li> 元素，还有 4 个 Text 节点（表示 <li> 元素周围的空格）。

11.1.4 Text 类型

Text 节点由 Text 类型表示，包含按字面解释的纯文本，也可能包含转义后的 HTML 字符，但不含 HTML 代码。Text 类型的节点具有以下特征：

- nodeType 为 Node.Text_NODE;
- nodeName 值为 ``#text'';
- nodeValue 值为节点中包含的文本;
- parentNode 值为 Element 对象;
- 不支持子节点。

Text 节点中包含的文本可以通过 nodeValue 属性访问，也可以通过 data 属性访问，这两个属性包含相同的值。修改 nodeValue 或 data 的值，也会在另一个属性反映出来。文本节点暴露了以下操作文本的方法：

- appendData(text): 向节点末尾添加文本 text。
- deleteData(offset,count): 从位置 offset 开始删除 count 个字符。
- insertData(offset,text): 从位置 offset 插入 text。
- replaceData(offset,count,text): 用 text 替换从位置 offset 到 offset+count 的文本；
- splitText(offset): 在位置 offset 将当前文本节点拆分为两个文本节点。
- substringData(offset,count): 提取从位置 offset 到 offset + count 的文本。

默认情况下，包含文本内容的每个元素最多只能有一个文本节点。只要开始标签和结束标签之间有内容，就会创建一个文本节点。

```
<div>Hello World!</div>
```

下列代码可以用来访问并修改这个文本节点：

```
let textNode = div.firstChild;
div.firstChild.nodeValue = "Some other message";
```

修改文本节点有一点要注意，就是 HTML 或 XML 代码（取决于文档类型）会被转换成实体编码，即小于号、大于号或引号会被转义，如下所示：

```
// 输出为"Some &lt;strong&gt;other&lt;/strong&gt; message"
div.firstChild.nodeValue = "Some <strong>other</strong> message";
```

操作文本

`document.createTextNode()` 可以用来创建新文本节点，它接收一个参数，即要插入节点的文本。创建新文本节点后，其 `ownerDocument` 属性会被设置为 `document`。在把这个节点添加到文档树之前，我们不会在浏览器中看到它。

一般来说一个元素只包含一个文本子节点，也可以添加多个文本子节点，不过这样做没有实际意义。在将一个文本节点作为另一个文本节点的同胞插入后，两个文本节点的文本之间不会包含空格。

DOM 文档中的同胞文本节点可能导致困惑，因为一个文本节点足以表示一个文本字符串。同样，DOM 文档中也经常会出现两个相邻文本节点。为此，有一个方法可以合并相邻的文本节点: `normalize()`。是在 `Node` 类型中定义的（因此所有类型的节点上都有这个方法）。在包含两个或多个相邻文本节点的父节点上调用 `normalize()` 时，所有同胞文本节点会被合并为一个文本节点，这个文本节点的 `nodeValue` 就等于之前所有同胞节点 `nodeValue` 拼接在一起得到的字符串。

`Text` 类型定义了一个与 `normalize()` 相反的方法：`splitText()`。这个方法可以在指定的偏移位置拆分 `nodeValue`，将一个文本节点拆分成两个文本节点。拆分之后，原来的文本节点包含开头到偏移位置前的文本，新文本节点包含剩下的文本。这个方法返回新的文本节点，具有与原来的文本节点相同的 parentNode。

11.1.5 其他节点类型

Comment 类型

DOM 中的注释通过 `Comment` 类型表示。`Comment` 类型的节点具有以下特征：

- `nodeType` 为 `Node.Comment_NODE`;
- `nodeName` 值为 ``#comment'';
- `nodeValue` 值为节注释的内容;
- `parentNode` 值为 `Document, Element` 对象;
- 不支持子节点。

`Comment` 类型与 `Text` 类型继承同一个基类（`CharacterData`），因此拥有除 `splitText()` 之外 `Text` 节点所有的字符串操作方法。

注释节点可以作为父节点的子节点来访问。比如下面的 HTML 代码：

```
<div id="myDiv"><!-- A comment --></div>
```

这里的注释是 `<div>` 元素的子节点，这意味着可以像下面这样访问它：

```
let div = document.getElementById("myDiv");
let comment = div.firstChild;
alert(comment.data); // "A comment"
```

可以使用 `document.createComment()` 方法创建注释节点，参数为注释文本。

```
let comment = document.createComment("A comment");
```

注释节点很少使用，浏览器不承认 </html> 标签之后的注释。

CDATASection 类型

CDATASection 类型表示 XML 中特有的 CDATA 区块。CDATASection 类型继承 Text 类型，因此拥有包括 splitText() 在内的所有字符串操作方法。

- nodeType 为 Node.CDATA_SECTION_NODE;
- nodeName 值为 ``#cdata-section'';
- nodeValue 值为 CDATA 区块的内容;
- parentNode 值为 Document, Element 对象;
- 不支持子节点。

CDATA 区块只在 XML 文档中有效，因此某些浏览器比较陈旧的版本会错误地将 CDATA 区块解析为 Comment 或 Element。比如下面这行代码：

```
<div id="myDiv"><![CDATA[This is some content.]]></div>
```

在真正的 XML 文档中，可以使用 document.createCDataSection() 并传入节点内容来创建 CDATA 区块。

DocumentType 类型

DocumentType 类型的节点包含文档的文档类型（doctype）信息，具有以下特征：

- nodeType 为 Node.DOCUMENT_TYPE_NODE;
- nodeName 值为文档类型的名称;
- nodeValue 值为 null;
- parentNode 值为 Document 对象;
- 不支持子节点。

DocumentType 对象在 DOM Level 1 中不支持动态创建，只能在解析文档代码时创建。对于支持这个类型的浏览器，DocumentType 对象保存在 document.doctype 属性中。DOM Level 1 规定了 DocumentType 对象的 3 个属性：name, entities, notations:

- name: 文档类型的名称。
- entities: 文档类型描述的实体的 NamedNodeMap。
- notations: 文档类型描述的表示法的 NamedNodeMap。

因为浏览器中的文档通常是 HTML 或 XHTML 文档类型，所以 entities 和 notations 列表为空。（这个对象只包含行内声明的文档类型。）无论如何，只有 name 属性是有用的。这个属性包含文档类型的名称，即紧跟在 <!DOCTYPE 后面的那串文本。比如下面的 HTML 4.01 严格文档类型：

```
<!DOCTYPE HTML PUBLIC "-// W3C// DTD HTML 4.01// EN"
    "http:// www.w3.org/TR/html4/strict.dtd">
```

```
alert(document.doctype.name); // "html"
```

DocumentFragment 类型

在所有节点类型中，DocumentFragment 类型是唯一一个在标记中没有对应表示的类型。DOM 将文档片段定义为“轻量级”文档，能够包含和操作节点，却没有完整文档那样额外的消耗。DocumentFragment 节点具有以下特征：

- nodeType 为 Node.DOCUMENT_FRAGMENT_NODE;
- nodeName 值为 ``#document-fragment'';
- nodeValue 值为 null;
- parentNode 值为 null;
- 子节点可以是 Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference。

不能直接把文档片段添加到文档。相反，文档片段的作用是充当其他要被添加到文档的节点的仓库。可以使用 document.createDocumentFragment() 方法像下面这样创建文档片段：

```
let fragment = document.createDocumentFragment();
```

文档片段从 Node 类型继承了所有文档类型具备的可以执行 DOM 操作的方法。如果文档中的一个节点被添加到一个文档片段，则该节点会从文档树中移除，不会再被浏览器渲染。添加到文档片段的新节点同样不属于文档树，不会被浏览器渲染。可以通过 appendChild() 或 insertBefore() 方法将文档片段的内容添加到文档。在把文档片段作为参数传给这些方法时，这个文档片段的所有子节点会被添加到文档中相应的位置。文档片段本身永远不会被添加到文档树。

Attr 类型

元素数据在 DOM 中通过 Attr 类型表示。Attr 类型构造函数和原型在所有浏览器中都可以直接访问。技术上讲，属性是存在于元素 attributes 属性中的节点。Attr 节点具有以下特征：

- nodeType 为 Node.ATTRIBUTE_NODE;
- nodeName 值为属性名;
- nodeValue 值为属性值;
- parentNode 值为 null;
- 在 HTML 中不支持子节点，在 XML 中子节点可以是: Text, EntityReference。

属性节点尽管是节点，却不被认为是 DOM 文档树的一部分。Attr 节点很少直接被引用，

通常开发者更喜欢使用 `getAttribute()`、`removeAttribute()`、`setAttribute()` 方法操作属性。

Attr 对象上有 3 个属性：`name`、`value`、`specified`。`specified` 是一个布尔值，表示属性使用的是默认值还是被指定的值。

可以使用 `document.createAttribute()` 方法创建新的 `Attr` 节点，参数为属性名。

11.2 DOM 编程

动态加载

动态脚本就是在页面初始加载时不存在，之后又通过 DOM 包含的脚本。与之对应的，在 HTML 中插入 JavaScript 脚本有两种方式: 外部文件和直接引用。

动态加载外部文件很容易实现，比如:

```
<script src="foo.js"></script>
```

```
let script = document.createElement("script");
script.src = "foo.js";
document.body.appendChild(script);
```

另一个动态插入 JavaScript 的方式是嵌入源代码，如下面的例子所示：

```
<script>
    function sayHi() {
        alert("hi");
    }
</script>
```

```
let script = document.createElement("script");
script.appendChild(document.createTextNode("function sayHi(){alert('hi');}"));
document.body.appendChild(script);
```

动态样式

CSS 样式在 HTML 页面中可以通过两个元素加载。`<link>` 元素用于包含 CSS 外部文件，而 <style> 元素用于添加嵌入样式。与动态脚本类似，动态样式也是页面初始加载时并不存在，而是在之后才添加到页面中的。

动态加载外部文件很简单, 也是主流的方案:

```
<link rel="stylesheet" type="text/css" href="styles.css">
```

```
let link = document.createElement("link");
link.rel = "stylesheet";
link.type = "text/css";
link.href = "styles.css";
```

```
let head = document.getElementsByTagName("head")[0];
head.appendChild(link);
```

当然也可以嵌入源代码，但是比较繁琐:

```
<style type="text/css">
body {
    background-color: red;
}
</style>
```

```
let style = document.createElement("style");
style.type = "text/css";
style.appendChild(document.createTextNode("body{background-color:red}"));
let head = document.getElementsByTagName("head")[0];
head.appendChild(style);
```

不同的浏览器可能会不支持部分操作，我们可以使用 `try...catch` 对有问题的语句捕捉。

操作表格

表格是 HTML 最复杂的结构之一，用 DOM 操作表格往往十分繁琐，为此 DOM 给 `<table>`, `<tbody>`, `<tr>` 元素增加了一些属性和方法，下面展示 `<table>` 元素部分属性方法，其余几个元素方法类似:

- `tBodies`: 指向 `<tbody>` 元素的 `HTMLCollection`。
- `tHead`: 指向 `thead` 元素 (如果存在)。
- `createTHead()`，创建 `<thead>` 元素，放到表格中，返回引用。
- `insertRow(pos)`，在行集合中给定位置插入一行。

NodeList

DOM 中有三个关键的集合对象: `NodeList`, `NamedNodeMap`, `HTMLCollection`，后两个继承自 `NodeList`。这 3 个集合类型都是“实时的”，意味着文档结构的变化会实时地在它们身上反映出来。

任何时候要迭代 NodeList，最好再初始化一个变量保存当时查询时的长度，然后用循环变量与这个变量进行比较，因为往 `NodeList` 中添加元素会实时改变它的长度，最好的方式如下:

```
let divs = document.getElementsByTagName("div");
for (let i = 0, len = divs.length; i < len; ++i) {
    let div = document.createElement("div");
    document.body.appendChild(div);
}
```

一般来说，最好限制操作 `NodeList` 的次数。因为每次查询都会搜索整个文档，所以最

好把查询到的 NodeList 缓存起来。

11.3 MutationObserver 接口

MutationObserver 接口可以在 DOM 被修改时异步执行回调。使用 MutationObserver 可以观察整个文档、DOM 树的一部分，或某个元素。此外还可以观察元素属性、子节点、文本，或者前三者任意组合的变化。

MutationObserver 的实例要通过调用构造函数并传入一个回调函数来创建：

```
let observer = new MutationObserver(() => console.log('DOM was mutated!'));
```

observe() 方法

新创建的 MutationObserver 实例不会关联 DOM 的任何部分。要把这个 observer 与 DOM 关联起来，需要使用 observe() 方法。这个方法接收两个必需的参数：要观察其变化的 DOM 节点，以及一个 MutationObserverInit 对象。

MutationObserverInit 对象用于控制观察哪些方面的变化，是一个键/值对形式配置选项的字典。例如，下面的代码会创建一个观察者（observer）并配置它观察 <body> 元素上的属性变化：

```
let observer = new MutationObserver(() => console.log('<body> attributes changed'));
observer.observe(document.body, { attributes: true });
```

执行以上代码后，<body> 元素上任何属性发生变化都会被这个 MutationObserver 实例发现，然后就会异步执行注册的回调函数。

回调与 MutationRecord

每个回调都会收到一个 MutationRecord 实例的数组。MutationRecord 实例包含的信息包括发生了什么变化，以及 DOM 的哪一部分受到了影响。因为回调执行之前可能同时发生多个满足观察条件的事件，所以每次执行回调都会传入一个包含按顺序入队的 MutationRecord 实例的数组。

```
let observer = new MutationObserver((mutationRecords) => console.log(mutationRecords));
```

mutationRecords 有很多属性，例如 target，type，oldValue 等，开发者可以调用这些属性自定义回调函数。

默认情况下，只要被观察的元素不被垃圾回收，MutationObserver 的回调就会响应 DOM 变化事件，从而被执行。要提前终止执行回调，可以调用 disconnect() 方法。

调用 disconnect() 并不会结束 MutationObserver 的生命。还可以重新使用这个观察者，再用 observer() 将它关联到新的目标节点。

异步回调与记录队列

MutationObserver 接口是出于性能考虑而设计的，其核心是异步回调与记录队列模型。为了在大量变化事件发生时不影响性能，每次变化的信息（由观察者实例决定）会保存在 MutationRecord 实例中，然后添加到记录队列。这个队列对每个 MutationObserver 实例都是唯一的，是所有 DOM 变化事件的有序列表。

每次 MutationRecord 被添加到 MutationObserver 的记录队列时，仅当之前没有已排期的微任务回调时（队列中微任务长度为 0），才会将观察者注册的回调（在初始化 MutationObserver 时传入）作为微任务调度到任务队列上。这样可以保证记录队列的内容不会被回调处理两次。

不过在回调的微任务异步执行期间，有可能又会发生更多变化事件。因此被调用的回调会接收到一个 MutationRecord 实例的数组，顺序为它们进入记录队列的顺序。回调要负责处理这个数组的每一个实例，因为函数退出之后这些实现就不存在了。回调执行后，这些 MutationRecord 就用不着了，因此记录队列会被清空，其内容会被丢弃。

调用 MutationObserver 实例的 takeRecords() 方法可以清空记录队列，取出并返回其中的所有 MutationRecord 实例。

性能，内存与垃圾回收

DOM Level 2 规范中描述的 MutationEvent 定义了一组会在各种 DOM 变化时触发的事件。由于浏览器事件的实现机制，这个接口出现了严重的性能问题。因此，DOM Level 3 规定废弃了这些事件。MutationObserver 接口就是为替代这些事件而设计的更实用、性能更好的方案。

无论如何，使用 MutationObserver 仍然不是没有代价的。因此理解什么时候避免出现这种情况就很重要了。

MutationObserver 实例与目标节点之间的引用关系是非对称的。MutationObserver 拥有对要观察的目标节点的弱引用。因为是弱引用，所以不会妨碍垃圾回收程序回收目标节点。然而，目标节点却拥有对 MutationObserver 的强引用。如果目标节点从 DOM 中被移除，随后被垃圾回收，则关联的 MutationObserver 也会被垃圾回收。

记录队列中的每个 MutationRecord 实例至少包含对已有 DOM 节点的一个引用。如果变化是 childList 类型，则会包含多个节点的引用。记录队列和回调处理的默认行为是耗尽这个队列，处理每个 MutationRecord，然后让它们超出作用域并被垃圾回收。

有时候可能需要保存某个观察者的完整变化记录。保存这些 MutationRecord 实例，也就会保存它们引用的节点，因而会妨碍这些节点被回收。如果需要尽快地释放内存，建议从每个 MutationRecord 中抽取出最有用的信息，然后保存到一个新对象中，最后抛弃 MutationRecord。

12 DOM 扩展

尽管 DOM API 已经相当不错，但各个浏览器社区开发出了自己专有的 DOM 扩展。W3C 着手将这些已成为事实标准的专有扩展编制成正式规范。由此诞生了描述 DOM 扩展的两个标准：Selectors API 与 HTML5。另外还有较小的 Element Traversal 规范，增加了一些 DOM 属性。

12.1 Selectors API

JavaScript 库中最流行的一种能力就是根据 CSS 选择符的模式匹配 DOM 元素。比如，jQuery 就完全以 CSS 选择符查询 DOM 获取元素引用，而不是使用 `getElementById()` 和 `getElementsByTagName()`。

Selectors API 是 W3C 推荐标准，规定了浏览器原生支持的 CSS 查询 API。支持这一特性的所有 JavaScript 库都会实现一个基本的 CSS 解析器，然后使用已有的 DOM 方法搜索文档并匹配目标节点。虽然库开发者在不断改进其性能，但 JavaScript 代码能做到的毕竟有限。通过浏览器原生支持这个 API，解析和遍历 DOM 树可以通过底层编译语言实现，性能也有了数量级的提升。

Selectors API Level 1 的核心是两个方法：`querySelector()` 和 `querySelectorAll()`。在兼容浏览器中，`Document` 类型和 `Element` 类型的实例上都会暴露这两个方法。

Selectors API Level 2 规范在 Element 类型上新增了更多方法，比如 `matches()`、`find()` 和 `findAll()`。不过，目前还没有浏览器实现或宣称实现 `find()` 和 `findAll()`。

总而言之，现在通用的有三个 Selectors API: `querySelector()`, `querySelectorAll()`, `matches()`。

- `querySelector()`: 接收 CSS 选择符参数，返回匹配该模式的第一个后代元素，如果没有匹配项则返回 null。在 `Document` 上使用 `querySelector()` 方法时，会从文档元素开始搜索；在 `Element` 上使用 `querySelector()` 方法时，则只会从当前元素的后代中查询。

```
let body = document.querySelector("body");
```

- `querySelectorAll()`: 和 `querySelector()` 类似，返回的是 `NodeList` 的静态实例。
- `matches()`: 接收一个 CSS 选择符参数，如果元素匹配则该选择符返回 `true`，否则返回 `false`。使用这个方法可以方便地检测某个元素会不会被上面两个方法返回。

12.2 HTML5

HTML5 代表着与以前的 HTML 截然不同的方向。在所有以前的 HTML 规范中，从未出现过描述 JavaScript 接口的情形，HTML 就是一个纯标记语言。JavaScript 绑定的事，一概交给 DOM 规范去定义。然而，HTML5 规范却包含了与标记相关的大量 JavaScript API 定义。其

中有的 API 与 DOM 重合，定义了浏览器应该提供的 DOM 扩展。

CSS 类扩展

自 HTML4 被广泛采用以来，Web 开发中一个主要的变化是 `class` 属性用得越来越多，其用处是为元素添加样式以及语义信息。自然地，JavaScript 与 CSS 类的交互就增多了，包括动态修改类名，以及根据给定的一个或一组类名查询元素，等等。为了适应开发者和他们对 `class` 属性的认可，HTML5 增加了一些特性以方便使用 CSS 类。

- `getElementsByClassName`: 暴露在 `document` 对象和所有 HTML 元素上。这个方法脱胎于基于原有 DOM 特性实现该功能的 JavaScript 库，提供了性能高好的原生实现。
  - 接收一个参数，即包含一个或多个类名的字符串，返回类名中包含相应类的元素的 `NodeList`。如果提供了多个类名，则顺序无关紧要。

```
// 取得所有类名中包含"username"和"current"元素
// 这两个类名的顺序无关紧要
let allCurrentUsernames = document.getElementsByClassName("username current");
```

- `classList` 属性: 用于方便快捷地操作元素地多个 `class` 属性,可以完全取代 `className` 属性。它包含了以下方法:
  - `add(value)`: 向类名列表中添加指定的字符串值 `value`。
  - `contains(value)`: 返回布尔值，表示 `value` 是否存在。
  - `remove(value)`: 向类名列表中删除指定的字符串值 `value`。
  - `toggle(value)`: 如果类名列表中已经存在指定的 `value`，则删除；如果不存在，则添加。

焦点管理

HTML5 增加了辅助 DOM 焦点管理的功能。首先是 `document.activeElement`，始终包含当前拥有焦点的 DOM 元素。页面加载时，可以通过用户输入（按 Tab 键或代码中使用 `focus()` 方法）让某个元素自动获得焦点。例如：

```
let button = document.getElementById("myButton");
button.focus();
console.log(document.activeElement === button); // true
```

默认情况下，`document.activeElement` 在页面刚加载完之后会设置为 `document.body`。而在页面完全加载之前，`document.activeElement` 的值为 `null`。

还有一个 `document.hasFocus()` 方法，返回布尔值表示文档是否有焦点。

第一个方法可以用来查询文档，确定哪个元素拥有焦点，第二个方法可以查询文档是否获得了焦点。

HTMLDocument 扩展

HTML5 扩展了 HTMLDocument 类型，增加了更多功能。与其他 HTML5 定义的 DOM 扩展一样，这些变化同样基于所有浏览器事实上都已经支持的专有扩展。

- readyState 属性: 这个属性表示文档加载的状态，有两个值: loading 与 complete 分别表示文档正在加载和加载完成。实际开发中，最好是把 document.readState 当成一个指示器，以判断文档是否加载完毕。
- compatMode 属性: 指示浏览器当前处于什么渲染模式，一般有两个值: CSS1Compat 和 BackCompat 分别表示标准模式和混杂模式。
- head: 指向 <head> 元素。

此外，还增加了一个 characterSet 属性表示文档实际使用的字符集。默认是 “UTF-16”。

自定义数据属性

HTML5 允许给元素指定非标准的属性，但要使用前缀 data-以便告诉浏览器，这些属性既不包含与渲染有关的信息，也不包含元素的语义信息。除了前缀，自定义属性对命名是没有限制的，data-后面跟什么都可以。

定义了自定义数据属性后，可以通过元素的 dataset 属性来访问。dataset 属性是一个 DOMStringMap 的实例，包含一组键/值对映射。元素的每个 data-name 属性在 dataset 中都可以通过 data-后面的字符串作为键来访问。

插入标记

DOM 虽然已经为操纵节点提供了很多 API，但向文档中一次性插入大量 HTML 时还是比较麻烦。相比先创建一堆节点，再把它们以正确的顺序连接起来，直接插入一个 HTML 字符串要简单（快速）得多。

- innerHTML 属性: 会返回元素所有后代的 HTML 字符串，包括元素、注释和文本节点。而在写入 innerHTML 时，则会根据提供的字符串值以新的 DOM 子树替代元素中原来包含的所有节点，浏览器会解析字符串。
- outerHTML 属性: 读取 outerHTML 属性时，会返回调用它的元素（及所有后代元素）的 HTML 字符串。在写入 outerHTML 属性时，调用它的元素会被传入的 HTML 字符串经解释之后生成的 DOM 子树取代。
- insertAdjacentHTML(), insertAdjacentText(): 接收两个参数, 第二个为要插入标记的位置和要插入的 HTML 或文本。第一个参数是下列值中的一个 (不区分大小写):
  - "beforebegin": 插入当前元素前面，作为前一个同胞节点；
  - "afterbegin": 插入当前元素内部，作为新的子节点或放在第一个子节点前面；
  - "beforeend": 插入当前元素内部，作为新的子节点或放在最后一个子节点后面；
  - "afterend": 插入当前元素后面，作为下一个同胞节点。

一般情况下这样做会带来性能提升，但在不同的浏览器上也不一定；实际开发中更多地会使用 JSX 风格的代码。

scrollIntoView()

DOM 规范中没有涉及的一个问题是如何滚动页面中的某个区域。为填充这方面的缺失，不同浏览器实现了不同的控制滚动的方式。在所有这些专有方法中，HTML5 选择了标准化 scrollIntoView()。scrollIntoView() 方法存在于所有 HTML 元素上，可以滚动浏览器窗口或容器元素以便包含元素进入视口。这个方法的参数如下：

- alignToTop: 布尔值。
  - true: 窗口滚动后元素的顶部与视口顶部对齐。
  - false：窗口滚动后元素的底部与视口底部对齐。
- scrollIntoViewOptions: 选项对象。
  - behavior: 定义过渡动画，可取的值为"smooth" 和"auto"，默认为"auto"。
  - block: 定义垂直方向的对齐，可取的值为"start"、"center"、"end" 和"nearest"，默认为"start"。
  - inline: 定义水平方向的对齐，可取的值为"start"、"center"、"end" 和"nearest"，默认为"nearest"。

不传参数等同于 alignToTop 为 true。

除此之外，还有一个 scrollIntoViewIfNeeded() 方法，将不在浏览器窗口的可见区域内的元素滚动到浏览器窗口的可见区域。这个方法是非标准的，不建议使用。

12.3 其他扩展

Element Traversal API

Element Traversal API 为 DOM 元素添加了 5 个属性：

- childElementCount: 返回子元素数量（不包含文本节点和注释）;
- firstElementChild: 指向第一个 Element 类型的子元素;
- lastElementChild: 指向最后第一个 Element 类型的子元素;
- previousElementSibling: 指向前一个 Element 类型的同胞元素;
- nextElementSibling: 指向后一个 Element 类型的同胞元素。

在支持的浏览器中，所有 DOM 元素都会有这些属性，为遍历 DOM 元素提供便利。这样开发者就不用担心空白文本节点的问题了。

专有扩展

这些扩展大部分是由于历史原因造成的，比如最早由 IR 浏览器提供，后来被所有浏览器支持。

- `children` 属性: 一个 `HTMLCollection`，只包含元素的 Element 类型的子节点。如果元素的子节点类型全部是元素类型，那 `children` 和 `childNodes` 中包含的节点应该是一样的。
- `contains()`: 需要确定一个元素是不是另一个元素的后代。
- `innerText` 属性: 对应元素中包含的所有文本内容，无论文本在子树中哪个层级。在用于读取值时，`innerText` 会按照深度优先的顺序将子树中所有文本节点的值拼接起来。在用于写入值时，`innerText` 会移除元素的所有后代并插入一个包含该值的文本节点。
- `outerText` 属性: 与 `innerText` 类似，只不过作用范围包含调用它的节点。

V DOM 进阶

13 DOM 事件

JavaScript 与 HTML 的交互是通过事件实现的，事件代表文档或浏览器窗口中某个有意义的时刻。可以使用仅在事件发生时执行的监听器（也叫处理程序）订阅事件。

13.1 事件流

在第四代 Web 浏览器（IE4 和 Netscape Communicator 4）开始开发时，开发团队碰到了一个有意思的问题：页面哪个部分拥有特定的事件呢？要理解这个问题，可以在一张纸上画几个同心圆。把手指放到圆心上，则手指不仅是在一个圆圈里，而且是在所有的圆圈里。两家浏览器的开发团队都是以同样的方式看待浏览器事件的。当你点击一个按钮时，实际上不光点击了这个按钮，还点击了它的容器以及整个页面。

事件流描述了页面接收事件的顺序。结果非常有意思，IE 和 Netscape 开发团队提出了几乎完全相反的事件流方案。IE 将支持事件冒泡流，而 Netscape Communicator 将支持事件捕获流。

事件冒泡

IE 事件流被称为事件冒泡，这是因为事件被定义为从最具体的元素（文档树中最深的节点）开始触发，然后向上传播至没有那么具体的元素（文档）。

比如说，我们点击 `<body>` 标签中的一个 `<div>`, `click` 事件会由如下顺序发生: `<div>` → `<body>` → `<html>` → `document`。

也就是说，`<div>` 元素，即被点击的元素，最先触发 `click` 事件。然后，`click` 事件沿 DOM 树一路向上，在经过的每个节点上依次触发，直至到达 `document` 对象。

所有现代浏览器都支持事件冒泡，只是在实现方式上会有一些变化。现代浏览器中的事件会一直冒泡到 `window` 对象。

事件捕获

事件捕获事件流由网景公司提出，沿着 DOM 树向下传播和事件冒泡流顺序完全相反。`click` 事件会由如下顺序发生: `document` → `<html>` → `<body>` → `<div>`。

事件捕获得到了所有现代浏览器的支持。实际上，所有浏览器都是从 `window` 对象开始捕获事件，而 DOM2 Events 规范规定的是从 `document` 开始。由于旧版本浏览器不支持，因此实际当中几乎不会使用事件捕获。通常建议使用事件冒泡，特殊情况下可以使用事件捕获。

DOM 事件流

DOM2 Events 规范规定事件流分为 3 个阶段：事件捕获、到达目标和事件冒泡。事件捕获最先发生，为提前拦截事件提供了可能。然后，实际的目标元素接收到事件。最后一个阶段是冒泡，最迟要在这个阶段响应事件。

在 DOM 事件流中，实际的目标（`<div>` 元素）在捕获阶段不会接收到事件。这是因为捕获阶段从 `document` 到 `<html>` 再到 `<body>` 就结束了。下一阶段，即会在 `<div>` 元素上触发事件的“到达目标”阶段，通常在事件处理时被认为是冒泡阶段的一部分。然后，冒泡阶段开始，事件反向传播至文档。

虽然 DOM2 Events 规范明确捕获阶段不命中事件目标，但现代浏览器都会在捕获阶段在事件目标上触发事件。最终结果是在事件目标上有两个机会来处理事件。

13.2 事件处理程序

事件意味着用户或浏览器执行的某种动作。比如，单击（click）、加载（load）、鼠标悬停（mouseover）。为响应事件而调用的函数被称为事件处理程序（或事件监听器）。事件处理程序的名字以”on” 开头，因此 click 事件的处理程序叫作 `onclick`，而 load 事件的处理程序叫作 `onload`。有很多方式可以指定事件处理程序。

HTML 事件处理程序

特定元素支持的每个事件都可以使用事件处理程序的名字以 HTML 属性的形式来指定。此时属性的值必须是能够执行的 JavaScript 代码。例如，要在按钮被点击时执行某些 JavaScript 代码，可以使用以下 HTML 属性：

```
<input type="button" value="Click Me" onclick="console.log('Clicked')"/>
```

在 HTML 中定义的事件处理程序可以包含精确的动作指令，也可以调用在页面其他地方定义的脚本，比如：

```
<script>
    function showMessage() {
        console.log("Hello world!");
    }
</script>
<input type="button" value="Click Me" onclick="showMessage()"/>
```

以这种方式指定的事件处理程序有一些特殊的地方。首先，会创建一个函数来封装属性的值。这个函数有一个特殊的局部变量 `event`，其中保存的就是 `event` 对象 (`event` 具体的属性不讨论)。有了这个对象，就不用开发者另外定义其他变量，也不用从包装函数的参数列表中去取了。

这个动态创建的包装函数还有一个特别有意思的地方，就是其作用域链被扩展了。在这

个函数中，document 和元素自身的成员都可以被当成局部变量来访问。这意味着事件处理程序可以更方便地访问自己的属性。

在 HTML 中指定事件处理程序有一些问题。第一个问题是时机问题。有可能 HTML 元素已经显示在页面上，用户都与其交互了，而事件处理程序的代码还无法执行。为此，大多数 HTML 事件处理程序会封装在 try/catch 块中，以便在这种情况下静默失败。另一个问题是对事件处理程序作用域链的扩展在不同浏览器中可能导致不同的结果。不同 JavaScript 引擎中标识符解析的规则存在差异，因此访问无限定的对象成员可能导致错误。

使用 HTML 指定事件处理程序的最后一个问题是 HTML 与 JavaScript 强耦合。如果需要修改事件处理程序，则必须在两个地方，即 HTML 和 JavaScript 中，修改代码。这也是很多开发者不使用 HTML 事件处理程序，而使用 JavaScript 指定事件处理程序的主要原因。

DOM0 事件处理程序

在 JavaScript 中指定事件处理程序的传统方式是把一个函数赋值给（DOM 元素的）一个事件处理程序属性。使用 JavaScript 指定事件处理程序，必须先取得要操作对象的引用。

每个元素（包括 `window` 和 `document`）都有通常小写的事件处理程序属性，比如 `onclick`。只要把这个属性赋值为一个函数即可：

```
let btn = document.getElementById("myBtn");
btn.onclick = function() {
    console.log("Clicked");
};
```

像这样使用 DOM0 方式为事件处理程序赋值时，所赋函数被视为元素的方法。因此，事件处理程序会在元素的作用域中运行，即 this 等于元素。下面的例子演示了使用 this 引用元素本身：

```
let btn = document.getElementById("myBtn");
btn.onclick = function() {
    console.log(this.id); // "myBtn"
};
```

通过将事件处理程序属性的值设置为 null，可以移除通过 DOM0 方式添加的事件处理程序。

DOM2 事件处理程序

DOM2 Events 为事件处理程序的赋值和移除定义了两个方法：`addEventListener()` 和 `removeEventListener()`。这两个方法暴露在所有 DOM 节点上，它们接收 3 个参数：事件名、事件处理函数和一个布尔值，`true` 表示在捕获阶段调用事件处理程序，`false`（默认值）表示在冒泡阶段调用事件处理程序。

仍以给按钮添加 `click` 事件处理程序为例，可以这样写：

```
let btn = document.getElementById("myBtn");
```

```
btn.addEventListener("click", () => {
    console.log(this.id);
}, false);
```

与 DOM0 方式类似，这个事件处理程序同样在被附加到的元素的作用域中运行。使用 DOM2 方式的主要优势是可以为同一个事件添加多个事件处理程序。

通过 `addEventListener()` 添加的事件处理程序只能使用 `removeEventListener()` 并传入与添加时同样的参数来移除。这意味着使用 `addEventListener()` 添加的匿名函数无法移除。

```
let btn = document.getElementById("myBtn");
let handler = function () {
    console.log(this.id);
};
btn.addEventListener("click", handler, false);
btn.removeEventListener("click", handler, false);
```

大多数情况下，事件处理程序会被添加到事件流的冒泡阶段，主要原因是跨浏览器兼容性好。把事件处理程序注册到捕获阶段通常用于在事件到达其指定目标之前拦截事件。如果不需要拦截，则不要使用事件捕获。