

JavaScript 高级程序设计笔记

Pionpill¹

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 13 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 语法非常灵活 (乱), 有很多没必要记的东西, 也有很多不明所以的设计方案, 这和他的历史有关。现在主流的做法是使用 TypeScript 编写前端脚本, 因此本人增加了 TypeScript 这一部分。

《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。
- DOM 与 BOM: 与 Web 开发相关的内容。
- 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 或其他讨论的文章, 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, 更多写法请读者用现代编辑器代码提示功能查询。

2022 年 12 月 13 日

目录

第一部分 ECMAScript	1
I 基础语法	
1 JavaScript 介绍	2
1.1 历史回顾	2
1.2 HTML 中的 JavaScript	2
2 语言基础	7
2.1 语法	7
2.2 关键字与保留字	8
2.3 变量	8
2.3.1 var 关键字	8
2.3.2 let 声明	9
2.3.3 const 声明	11
2.4 操作符	12
2.5 语句	13
2.5.1 for-in 语句	13
2.5.2 for-of 语句	14
2.5.3 标签语句	14
2.5.4 with 语句	14
3 变量，作用域与内存	16
3.1 原始值与引用值	16
3.2 上下文与作用域	17
3.3 垃圾回收	18
II 数据类型	
4 简单数据类型	20
4.1 typeof 操作符	20
4.2 Undefined 类型	20
4.3 Null 类型	21
4.4 Boolean 类型	22
4.5 Number 类型	22

4.6	String 类型	25
4.7	Symbol 符号类型	28
5	基本引用类型	30
5.1	Object	30
5.2	Date	31
5.3	RegExp	31
5.4	包装类型	32
5.5	单例内置对象	33
5.5.1	Global	33
5.5.2	Math	35
6	集合引用类型	36
6.1	Array 数组	36
6.1.1	数组表现	36
6.1.2	数组作为映射, 栈, 队列	36
6.1.3	谓词, 迭代与归并	37
6.1.4	ArrayBuffer	38
6.2	Map 映射	38
6.2.1	Map	38
6.2.2	WeakMap	39
6.3	Set 集合	39
III	函数与类	
7	函数	40
7.1	定义函数的多种方式	40
7.2	ECMAScript 的函数	41
7.3	函数内部	43
8	迭代器与生成器	45
8.1	迭代器	45
8.2	生成器	45
9	对象, 类	47
9.1	对象	47
9.1.1	属性的类型	47
9.1.2	对象操作	49
9.2	创建对象	50
9.2.1	工厂模式鱼构造函数模式	50

9.2.2	原型模式	52
9.3	继承	53
9.3.1	原型链继承	53
9.3.2	经典继承	54
9.3.3	组合继承	54
9.4	类	55

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 才渐渐得到统一。

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，其版本并不是由浏览器决定的，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。

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="anonymous"` 配置文件请求不必设置凭据标志。
- `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：

```
1 <script>
2     function sayHi() {
3         console.log("Hi!");
4     }
5 </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 `"\"`。

```
1 <script>
2     function sayHi() {
3         console.log("<\script>");
4     }
5 </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。

```
1 <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。

```
1 | <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。

```
1 | <!DOCTYPE html>
2 | <html lang="en">
3 |   <head>
4 |     <title> Example </title>
5 |     <script defer src="example1.js"> </script>
6 |     <script defer src="example2.js"> </script>
7 |   </head>
8 |   <body></body>
9 | </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。

```
1 let script = document.createElement('script');
2 script.src = 'gibberish.js';
3 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。

```
1 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。

```
1 <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：

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <script defer="defer" src="example1.js"></script>
6     <script defer="defer" src="example2.js"></script>
7     <title>Document</title>
8   </head>
9   <body>
10    <noscript>
11      <p>This page needs a JavaScript-enabled browser.</p>
12    </noscript>
13  </body>
```

14 | `</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:

```
1 | // 当行注释
```

- 多行注释 (块注释):

```
1 | /*多行
2 | 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：

```
1 | "use strict"
```

也可以单独在一个函数中执行严格模式：

```
1 | function doSomething() {
2 |     "use strict";
3 |     // 函数体
4 | }
```

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。

```
1 function test() {
2     var message = "hi";
3 }
4 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。

```
1 function test() {
2     message = "hi";
3 }
4 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：

```
1 function foo() {
2     console.log(age);
3     var age = 26;
4 }
```

上述代码等价于：

```
1 function foo() {
2     var age;
3     console.log(age);
4     age = 26;
5 }
```

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。

```
1 function foo() {
2     var age = 16;
3     var age = 26;
4     var age = 36;
5     console.log(age);
6 }
```

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辑。

⁶这种做法也不建议使用。

辑。

```
1 if (true) {  
2   var name = 'Matt';  
3   console.log(name);  
4 }  
5 console.log(name); // 正确
```

```
1 if (true) {  
2   let name = 'Matt';  
3   console.log(name);  
4 }  
5 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：

```
1 var name;  
2 var name; // 出现两次，无误  
3  
4 let age;  
5 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)。

```
1 var name = 'Matt';  
2 console.log(window.name); // 'Matt'  
3  
4 let age = 26;  
5 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。

```
1 <script>
2   var name = 'Pionpill';
3   let age = 22;
4 </script>
5
6 <script>
7   var name = 'Matt';
8   // 没有问题，同时被作为一个提升声明来处理
9   // 不需要检查之前是否声明过同名变量
10  let age = 23;
11  // age 之前声明过，报错
12 </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：

```
1 for(var i=0; i<5; i++) {
2   // 循环逻辑
3 }
4 console.log(i); // 5
```

改用 `let` 后就不会出现这个问题：

```
1 for(let i=0; i<5; i++) {
2   // 循环逻辑
3 }
4 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。

```
1 const name = "Pionpill";
2 name = "Matt"; // 报错
3
4 const person = {};
5 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)：

```
1 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：

```
1 let i = 0;
2 for(const j=7; i<5; ++i) {
3   console.log(j);
4 }
5 // 7,7,7,7,7
6
7 for(const key in {a:1,b:2}) {
8   console.log(key);
9 }
10 // a,b
11
12 for (const value of [1,2,3,4,5]) {
13   console.log(value);
14 }
15 // 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。

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

不止如此，ECMA 还提供了指数赋值操作符 `**=`：

```
1 let a = 3;
2 a **= 2;
3 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。

```
1 console.log("55" == 55); // true
2 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：

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

下面是一个例子：

```
1 for (const propName in window) {
2     document.write(propName);
3 }
```

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：

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

下面是示例：

```
1 | for (const el of [2,4,6,8]) {  
2 |     document.write(el);  
3 | }
```

2.5.3 标签语句

标签语句用于给语句加标签，语法如下：

```
1 | label: statement
```

下面是一个例子：

```
1 | start: for(let i=0; i<count; i++) {  
2 |     console.log(i);  
3 | }
```

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：

```
1 | 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。

```
1 | // 不使用 with 语句  
2 | let qs = location.search.substring(1);  
3 | let hostname = location.hostname;  
4 | let url = location.href;  
5 |  
6 | // 使用 with 语句  
7 | with(location) {  
8 |     let qs = search.substring(1);  
9 |     let hostName = hostname;  
10 |    let url = href;  
11 | }
```

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，对应变量的保存数据本身。

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。

```
1 let person = new Object();  
2 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。

```
1 function setName(obj) {  
2     obj.name = "Pionpill";  
3     obj = new Object();  
4     obj.name = "Brandon";  
5 }  
6  
7 let person = new Object();  
8 setName(person)  
9 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` 调用对应变量的。

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。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。

```
1 let person = new Object();
2 person.name = "Pionpill";
3
4 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）。

```
1 function Article(opt) {  
2     this.title = "ababababa";  
3     this.author = opt;  
4 }  
5  
6 let a1 = new Article();  
7 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`。

```
1 let a1 = new Article();  
2 let a2 = new Article();  
3  
4 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的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。

```
1 let message;  
2 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。

```
1 let message = undefined;
```

```
2 | 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。

```
1 | let message;
2 |
3 | console.log(message); // "undefined"
4 | 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"。

```
1 | let message
2 |
3 | console.log(typeof message); // "undefined"
4 | 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。

```
1 | let car = null;
2 | 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。

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

有别于 *undefined*，如果我们要保存未来的对象，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 <code>true</code>	转换为 <code>false</code>
Boolean	<code>true</code>	<code>false</code>
String	非空串	空串
Number	非零数值	0, NaN
Object	任意对象	<code>null</code>
Undefined	N/A	<code>undefined</code>

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：

```
1 let intNum = 55; // 十进制
2 let octNum = 070; // 八进制 56
3 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。

```
1 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价的。

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。

```
1 let floatNum1 = 1.1;
2 let floatNum2 = 0.1;
3 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。如果值本身是整数，那么它会被自动转换为整型值。

²严格模式下无效。

```
1 | let floatNum1 = 1.; // 小数点后没有数字, 当作整型1
2 | 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。

```
1 | 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。

```
1 | let result = Number.MAX_VALUE + Number.MAX_VALUE;
2 | 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。

```
1 | 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`:

```
1 | console.log(isNaN(NaN)); // true
2 | console.log(isNaN(10)); // false, 10是数值
3 | console.log(isNaN("10")); // false, "10"可以转换为数值
4 | console.log(isNaN("blue")); // true, 不可以转换为数值
5 | 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。

```
1 | parseInt("1234blue"); // 1234
2 | 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。

```
1 | parseInt("0xA");      // 10
2 | 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:

```
1 | parseInt("0xAF",16); // 175
2 | parseInt("AF",16);   // 175
3 | 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:

```
1 | parseInt("10",2);    // 二进制
2 | parseInt("10",8);    // 八进制
3 | parseInt("10",10);   // 十进制
4 | 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。

```
1 let age = 11;
2 let ageString = age.toString();    // 字符串 "11"
3 let found = true;
4 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)。

```
1 let num = 10;
2 console.log(num.toString()); // "10"
3 console.log(num.toString(2)); // "1010"
4 console.log(num.toString(8)); // "12"
5 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。

```
1 let multiline1 = "first lin\nsecond line";
2 let multiline1 = "first line
3 second line";
4
5 console.log(multiline1);
```

```

6 // first line
7 // second line
8
9 console.log(multiline2);
10 // first line
11 // 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：

```

1 let value = 5;
2 let exponent = 'second';
3 // 之前的写法
4 let interpolatedString = value + ' to the ' + exponent + ' power is ' + (value * value);
5 // 现在的写法
6 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：

```

1 function capitalize(word) {
2   return `${word[0].toUpperCase()} ${word.slice(1)}`
3 }
4 console.log(`${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。

```

1 let a = 6;
2 let b = 9;
3
4 function simpleTag(strings, aValExpression, bValExpression, sumExpression) {
5   console.log(strings);
6   console.log(aValExpression);
7   console.log(bValExpression);
8   console.log(sumExpression);
9   return 'foobar';
10 }
11
12 let untaggedResult = `${a} + ${b} == ${a+b}`;

```

```

13 let taggedResult = simpleTag`${a} + ${b} == ${a+b}`;
14 // ["", "+", "=", ""]
15 // 6
16 // 9
17 // 15
18
19 console.log(untaggedResult); // "6 + 9 = 15"
20 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：

```

1  let a = 6;
2  let b = 9;
3
4  function simpleTag(strings, ...expressions) {
5      console.log(strings);
6      for (const expression of expressions) {
7          console.log(expression);
8      }
9      return 'foobar';
10 }
11
12 let taggedResult = simpleTag`${a} + ${b} == ${a+b}`;
13 // ["", "+", "=", ""]
14 // 6
15 // 9
16 // 15
17
18 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：

```

1  let a = 6;
2  let b = 9;
3
4  function zipTag(strings, ...expressions) {
5      return strings[0] + expressions.map((e,i) => `${e}${strings[i+1]}`).join('');
6  }
7
8  let taggedResult = zipTag`${a} + ${b} == ${a+b}`;
9
10 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：

```

1 console.log(`\u00A9`); // ©

```



```

2 console.log(String.raw'\u00A9'); // \u00A9
3
4 console.log(String.raw'first line
5 second line')
6 // first line
7 // 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：

```

1 let firstSymbol = Symbol();
2 let secondSymbol = Symbol();
3
4 let thirdSymbol = Symbol("foo");
5 let forthSymbol = Symbol("foo");
6
7 console.log(firstSymbol == secondSymbol); // false
8 console.log(thirdSymbol == forthSymbol); // false
9
10 console.log(firstSymbol); // Symbol()
11 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：

```

1 let mySymbol = Symbol();
2 let myWrappedSymbol = Object(mySymbol);
3 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：

```

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

```

`Symbol.for()` 对每个字符串键都执行等幂操作。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。

```
1 let fooGlobalSymbol = Symbol.for('foo'); // 创建新符号
2 let otherFooGlobalSymbol = Symbol.for('foo'); // 重用已有符号
3
4 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:

```
1 let localSymbol = Symbol('foo');
2 let globalSymbol = Symbol.for('foo');
3
4 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。

```
1 let emptyGlobalSymbol = Symbol.for();
2 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`。

```
1 let s = Symbol.for('foo');
2 console.log(Symbol.keyFor(s)); // foo
3
4 let s2 = Symbol('bar');
5 console.log(Symbol.keyFor(s2)); // undefined
6
7 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。

```
1 let o = new Object();
2 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。
- `toLocaleString()`: 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。

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

- 使用对象字面量，直观方便。

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

```
3 |   age: 29
4 | };
```

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：

```
1 | person.name = "Pionpill";
2 | 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：

```
1 | 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

```
1 | let someDate = new Date(Date.parse("May 23, 2019"));
2 | 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：

```
1 | 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：

- `toLocaleString()`：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。

```
1 | 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。

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

上述写法是一种语法糖, 也可以使用构造函数:

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

在构造函数中, 元字符必须经过两次转义:

```
1 const re1 = /cat/g;
2 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:

```
1 let s1 = "some text";
2 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。

```
1 let s1 = "some text";
2 s1.color = "red";
3 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：

```
1 let obj = new Object("some text");
2 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：

```
1 let value = "25";
2 let number = Number(value); // 转型函数
3 console.log(typeof number); // "number"
4 let obj = new Number(value); // 构造函数
5 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：

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

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

同时也提供对应的解码方法:

```
1 let uri = "http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.js%23start";
2 // http%3A%2F%2Fwww.wrox.com%2Fillegal value.js%23start
3 console.log(decodeURI(uri));
4 // http:// www.wrox.com/illegal value.js#start
5 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:

```
1 eval("console.log('hi')");
2 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。

```
1 var color = "red";
2 function sayColor() {
3     console.log(window.color);
4     return this; // this 表示 Global 对象
5 }
6 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：

```
1 // new 创建空数组
2 let colors = new Array();
3 // new 创建指定大小的数组
4 let colors = new Array(3);
5 // new 创建指定元素的数组
6 let colors = new Array("red", "blue", "yellow");
7 // 字面量创建数组
8 let colors = ["red", "blue", "yellow"];
9 // 字面量创建有空位的数组，空位被视作 undefined
10 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()`

```
1 if (Array.isArray(value)) {
2     .....
3 }
```

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：

```
1 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
2 // 通过Array.from()直接转换为数组实例
3 const aKeys = Array.from(a.keys()); // [0, 1, 2, 3]
4 const aValues = Array.from(a.values()); // ["foo", "bar", "baz", "qux"]
5 const aEntries = Array.from(a.entries()); // [[0, "foo"], [1, "bar"], [2, "baz"], [3, "qux"]]
```

因此，可以这样写：

```

1  const a = ["foo", "bar", "baz", "qux"];
2  for (const [idx, element] of a.entries()) {
3      alert(idx);
4      alert(element);
5  }

```

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

```

1  // 栈操作
2  let colors = new Array(); // 创建一个数组
3  let count = colors.push("red", "green"); // 压入两项
4  let item = colors.pop();    // 弹出一项
5  // 队列操作
6  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。

```

1  const people = [
2      {
3          name: "Matt",
4          age: 27
5      },
6      {
7          name: "Nicholas",
8          age: 29
9      }
10 ];
11 alert(people.find((element, index, array) => element.age < 28));
12 // {name: "Matt", age: 27}
13 alert(people.findIndex((element, index, array) => element.age < 28));
14 // 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。

```
1 | const buf = new ArrayBuffer(16); // 在内存中分配16字节
2 | 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：

```
1 | const buf1 = new ArrayBuffer(16);
2 | const buf2 = buf1.slice(4, 12);
3 | 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 $\text{Number.MAX_SAFE_INTEGER}^{53} - 1$ 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：

```
1 | const m = new Map([
2 |   ["key1", "val1"],
3 |   ["key2", "val2"],
4 |   ["key3", "val3"]
5 | ]);
```

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 是 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。

```
1 | const wm = new WeakMap();  
2 | 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。

```
1 | 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：

```
1 function sum(num1, num2) {  
2     return num1 + num2;  
3 }
```

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：

```
1 let sum = function(num1, num2) {  
2     return num1 + num2;  
3 };
```

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：

```
1 (function() {  
2     // 块级作用域  
3 })();
```

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)：

```
1 let sum = (num1, num2) => {  
2     return num1 + num2;  
3 };
```

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：

```
1 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:

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

```
1 let double = (x) => {return 2*x;};
2 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)。

```
1 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”。

```
1 function foo() {}
2 let bar = function() {};
3 let baz = () => {};
4
5 console.log(foo.name); // foo
6 console.log(bar.name); // bar
7 console.log(baz.name); // baz
8 console.log(() => {}.name); // 空字符串
9 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。

```
1 function sayHi(name, message) {  
2     console.log("Hello" + name + "," + message);  
3 }  
4  
5 // 等价写法  
6 function sayHi() {  
7     console.log("Hello" + arguments[0] + "," + arguments[1]);  
8 }
```

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。

```
1 function makeKing(name = 'Henry') {  
2     return name;  
3 }
```

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：

```
1 function getSum() {  
2     let sum = 0;  
3     for (let i = 0; i < arguments.length; ++i) {  
4         sum += arguments[i];  
5     }  
6     return sum;  
7 }  
8 let values = [1,2,3,4];  
9 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：

```
1 function ignoreFirst(firstValue, ...values) {  
2     console.log(values);  
3 }
```

```
4 | 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`)

```
1 | window.color = 'red';
2 | let o = { color: 'blue' };
3 | function sayColor() {
4 |     console.log(this.color);
5 | }
6 | sayColor(); // 'red'
7 | o.sayColor = sayColor;
8 | 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。

```
1 | function King() {
2 |     if (!new.target) {
3 |         throw 'King must be instantiated using "new"';
4 |     }
5 |     console.log('King instantiated using "new"');
6 | }
7 | new King(); // King instantiated using "new"
8 | 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。

```
1 function sum(num1, num2) {  
2     return num1 + num2;  
3 }  
4 function callSum1(num1, num2) {  
5     return sum.apply(this, arguments);  
6     // 传入arguments对象  
7 }  
8 function callSum2(num1, num2) {  
9     return sum.apply(this, [num1, num2]); // 传入数组  
10 }  
11 console.log(callSum1(10, 10)); // 20  
12 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。

```
1 // 生成器函数声明
2 function * generatorFn() {...}
3 // 生成器哈桑农户表达式
4 let generationFn = function*() {...}
5 // 作为对象字面量方法的生成器函数
6 let foo = {
7   * generatorFn() {}
8 }
9 // 作为类实例方法的生成器函数
10 class Foo {
11   * generatorFn() {}
12 }
13 // 作为类静态方法的生成器函数
14 class Bar {
15   static * generatorFn() {}
16 }
```

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。

```
1 function* generatorFn() {}
2 const g = generatorFn();
3 console.log(g); // generatorFn {<suspended>}
4 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。

```
1 function* generatorFn() {
2   yield 'foo'; yield 'bar';
3   return 'baz';
4 }
5 let generatorObject = generatorFn();
6 console.log(generatorObject.next()); // { done: false, value: 'foo' }
7 console.log(generatorObject.next()); // { done: false, value: 'bar' }
8 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：

```
1 function* generatorFn() {
2   yield 'foo';
3   yield 'bar';
4   return 'baz';
5 }
6 let generatorObject1 = generatorFn();
7 let generatorObject2 = generatorFn();
8 console.log(generatorObject1.next()); // { done: false, value: 'foo' }
9 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：

```
1 // 写法一
2 let person = new Object();
3 person.name = "Nicholas";
4 person.age = 29;
5 person.job = "Software Engineer";
6 person.sayName = function() {
7     console.log(this.name);
8 };
9 // 写法二
10 let person = {
11     name: "Nicholas",
12     age: 29,
13     job: "Software Engineer",
14     sayName() {
15         console.log(this.name);
16     }
17 };
```

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`。
- `[[Enumerable]]`: 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：要为其添加属性的对象、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：

```
1 let person = {};  
2 Object.defineProperty(person, "name", {  
3   writable: false,  
4   value: "Nicholas"  
5 });  
6 console.log(person.name); // "Nicholas"  
7 person.name = "Greg";  
8 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()`。

```
1 let book = {  
2   year_: 2017,  
3   edition: 1  
4 };  
5  
6 Object.defineProperty(book, "year", {  
7   get() {  
8     return this.year_;  
9   },  
10  set(newValue) {  
11    if (newValue > 2017) {  
12      this.year_ = newValue;  
13      this.edition += newValue - 2017;  
14    }  
15  }  
16 });  
17 book.year = 2018;  
18 console.log(book.edition); // 2
```

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

```

1 let book = {};
2 Object.defineProperties(book, {
3   year_: {
4     value: 2017
5   },
6   edition: {
7     value: 1
8   },
9   year: {
10    get() {
11      return this.year_;
12    },
13    set(newValue) {
14      if (newValue > 2017) {
15        this.year_ = newValue;
16        this.edition += newValue - 2017;
17      }
18    }
19  }
20 });

```

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。

```

1 let person = {
2   name: 'Matt',
3   age: 27
4 };
5 let { name: personName, age: personAge } = person;

```

```

6 console.log(personName); // Matt
7 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：

```

1 let person = {
2   name: 'Matt',
3   age: 27
4 };
5 let { name, age, job } = person;
6 console.log(name); // Matt
7 console.log(age); // 27
8 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：

```

1 let person = {
2   name: 'Matt',
3   age: 27,
4   job: {
5     title: 'Software engineer'
6   }
7 };
8 let personCopy = {};
9
10 ({
11   name: personCopy.name,
12   age: personCopy.age,
13   job: personCopy.job
14 } = person);
15
16 person.job.title = 'Hacker'
17 console.log(person); // { name: 'Matt', age: 27, job: { title: 'Hacker' } }
18 console.log(personCopy); // { name: 'Matt', age: 27, job: { title: 'Hacker' } }

```

9.2 创建对象

9.2.1 工厂模式/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

```

1 function createPerson(name, age, job) {

```

```

2   let o = new Object();
3   o.name = name;
4   o.age = age;
5   o.job = job;
6   o.sayName = function() {
7       console.log(this.name);
8   };
9   return o;
10 }
11 let person1 = createPerson("Nicholas", 29, "Software Engineer");
12 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：

```

1  function Person(name, age, job){
2      this.name = name;
3      this.age = age;
4      this.job = job;
5      this.sayName = function() {
6          console.log(this.name);
7      };
8  }
9  let person1 = new Person("Nicholas", 29, "Software Engineer");
10 let person2 = new Person("Greg", 27, "Doctor");
11 person1.sayName(); // Nicholas
12 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：

```
1 console.log(person1.constructor == Person); // true
2 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原型的对象。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：

```
1 function Person() {}
2 Person.prototype.name = "Nicholas";
3 Person.prototype.age = 29;
4 Person.prototype.job = "Software Engineer";
5 Person.prototype.sayName = function() {
6     console.log(this.name);
7 };
8 let person1 = new Person();
9 person1.sayName(); // "Nicholas"
10 let person2 = new Person();
11 person2.sayName(); // "Nicholas"
12 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，指向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:

```
1 function SuperType() {
2     this.property = true;
3 }
4 SuperType.prototype.getSuperValue = function() {
5     return this.property;
6 };
7 function SubType() {
8     this.subproperty = false;
9 }
10
11 // 继承SuperType
12 SubType.prototype = new SuperType();
13
14 // 新方法
```

```

15 SubType.prototype.getSubValue = function () {
16     return this.subproperty;
17 };
18
19 // 覆盖已有的方法
20 SubType.prototype.getSuperValue = function () {
21     return false;
22 };
23 let instance = new SubType();
24 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”)。

```

1 function SuperType() {
2     this.colors = ["red", "blue", "green"];
3 }
4
5 function SubType() {
6     // 继承SuperType
7     SuperType.call(this);
8 }
9
10 let instance1 = new SubType();
11 instance1.colors.push("black");
12 console.log(instance1.colors); // "red,blue,green,black"
13 let instance2 = new SubType();
14 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：

```

1 function SuperType (name){

```

```

2   this.name = name;
3   this.colors ["red","blue","green"];
4 }
5
6 SuperType.prototype.sayName = function(){
7   console.log(this.name);
8 };
9
10 function SubType(name,age){
11   //继承属性
12   SuperType.call(this,name);
13   this.age = age;
14 }
15
16 //继承方法
17 SubType.prototype = new SuperType();
18 SubType.prototype.sayAge = function () {
19   console.log(this.age);
20 }

```

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：

```

1 class Person {};
2 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：

```
1 class Person {
2   constructor(name) {
3     console.log(arguments.length);
4     this.name = name || null;
5   }
6 }
7 let p1 = new Person;           // 0
8 console.log(p1.name);         // null
9 let p2 = new Person();        // 0
10 console.log(p2.name);         // null
11 let p3 = new Person('Jake');  // 1
12 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：

```
1 class Person {
2   constructor() { // 添加到this的所有内容都会存在于不同的实例上
3     this.locate = () => console.log('instance');
4   }
5   // 在类块中定义的所有内容都会定义在类的原型上
6   locate() { console.log('prototype'); }
7 } let p = new Person();
8 p.locate();           // instance
9 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：

```
1 class Person {
2   constructor() { // 添加到this的所有内容都会存在于不同的实例上
3     this.locate = () => console.log('instance');
4   }
5 }
```

```

5 // 在类块中定义的所有内容都会定义在类的原型上
6 locate() {
7     console.log('prototype');
8 }
9 // 定义在类本身上
10 static locate() {
11     console.log('class', this);
12 }
13 }

```

此外，也可以像函数一样在类定义外增加函数：

```

1 // 在类上定义独享的函数
2 Person.greeting = 'My name is';
3 // 在原型上定义共享的函数
4 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。

```

1 class Vehicle {
2     constructor() {
3         console.log(new.target);
4         if (new.target === Vehicle) {
5             throw new Error('Vehicle cannot be directly instantiated');
6         }
7     }
8 }

```