读书笔记

笔记本:	My Notebook		
创建时间:	2021/1/18 21:06	**更新时间:**	2021/1/29 18:00
作者:	L.11_1.15		
标签:	0118		

第 1 章 什么是JavaScript

JavaScript 与 Java 和 C++等传统面向对象

DOM

文档对象模型（DOM，Document Object Model）是一个应用编程 接口（API），用于在HTML中使用扩展的XML。DOM将整个页面抽 象为一组分层节点。HTML或XML页面的每个组成部分都是一种节 点，包含不同的数据

DOM视图：描述追踪文档不同视图（如应用CSS样式前后的 文档）的接口。

DOM事件：描述事件及事件处理的接口。

DOM样式：描述处理元素CSS样式的接口。

DOM遍历和范围：描述遍历和操作DOM树的接口。

BOM

浏览器对象模型

用于支持访问和操作浏览器的窗口。使用BOM，开发者可以操控浏览 器显示页面之外的部分。而BOM真正独一无二的地方，当然也是问题 最多的地方，就是它是唯一一个没有相关标准的JavaScript实现。

小结

JavaScript是一门用来与网页交互的脚本语言，包含以下三个组成 部分。 ECMAScript：由ECMA-262定义并提供核心功能。 文档对象模型（DOM）：提供与网页内容交互的方法和接口。 浏览器对象模型（BOM）：提供与浏览器交互的方法和接口。

第 2 章 HTML中的JavaScript

推迟执行脚本

HTML 4.01为<script> 元素定义了一个叫 defer 的属性 这 个属性表示脚本在执行的时候不会改变页面的结构

完全可以在整个页面解析完之后再运行。在<script> 元素上设置 defer 属性，会告诉浏览器应该立即开始下载，但执行应该推迟

defer 属性只对外部脚本文件才有效

异步执行脚本

HTML5为<script>元素定义了 async 属性。从改变脚本处 理方式上看， async 属性与 defer 类似。当然，它们两者也都只 适用于外部脚本，都会告诉浏览器立即开始下载。不过，与 defer 不同的是，标记为 async 的脚本并不保证能按照它们出现的次序执 行

2.2 行内代码与外部文件

虽然可以直接在HTML文件中嵌入JavaScript代码，但通常认为最 佳实践是尽可能将JavaScript代码放在外部文件中 不过这个最佳实践 并不是明确的强制性规则。推荐使用外部文件的理由如下

可维护性。JavaScript代码如果分散到很多HTML页面，会导致维 护困难。而用一个目录保存所有JavaScript文件，则更容易维护， 这样开发者就可以独立于使用它们的HTML页面来编辑代码。

缓存。浏览器会根据特定的设置缓存所有外部链接的JavaScript文 件，这意味着如果两个页面都用到同一个文件，则该文件只需下 载一次。这最终意味着页面加载更快。

适应未来。通过把JavaScript放到外部文件中，就不必考虑用 XHTML或前面提到的注释黑科技。包含外部JavaScript文件的语 法在HTML和XHTML中是一样的。 在配置浏览器请求外部文件时，要重点考虑的一点是它们会占用 多少带宽。在SPDY/HTTP2中，预请求的消耗已显著降低，以轻量、 独立JavaScript组件形式向客户端送达脚本更具优势。

2.3 文档模式

即可以使用 doctype 切换文档模 式。最初的文档模式有两种：混杂模式（quirks mode）和标准模式 （standards mode）。前者让IE像IE5一样（支持一些非标准的特 性），后者让IE具有兼容标准的行为。虽然这两种模式的主要区别只 体现在通过CSS渲染的内容方面，但对JavaScript也有一些关联影响， 或称为副作用

小结

要包含外部JavaScript文件，必须将 src 属性设置为要包含文件 的URL。文件可以跟网页在同一台服务器上，也可以位于完全不 同的域.

所有<script>元素在不使用 defer 和 async 属性的情况下，包含在<script> 元素 中的代码必须严格按次序解释

对不推迟执行的脚本，浏览器必须解释完位于<script> 元素 中的代码，然后才能继续渲染页面的剩余部分。为此，通常应该 把<script> 元素放到页面末尾，介于主内容之后及</body> 标签之前

可以使用 defer 属性把脚本推迟到文档渲染完毕后再执行。推 迟的脚本总是按照它们被列出的次序执行。

可以使用 async 属性表示脚本不需要等待其他脚本，同时也不 阻塞文档渲染，即异步加载。异步脚本不能保证按照它们在页面 中出现的次序执行。

通过使用<noscript> 元素，可以指定在浏览器不支持脚本时 显示的内容。如果浏览器支持并启用脚本，则<noscript>元素中的任何内容都不回被渲染。

第 3 章 语言基础

本章内容：

- 语法
- 数据类型
- 流控制语句
- 理解函数

暂时性死区

let 与 var 的另一个重要的区别，就是 let 声明的变量不会 在作用域中被提升。

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 成为 window 对象的属性（ var 声明的变量则会）

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，由于声明会被提升，JavaScript引擎会 自动将多余的声明在作用域顶部合并为一个声明。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 声明过同名变量，同时也就不可能在没有声明的情况下声明它。

不能使用 let 进行条件式声明是件好事，因为条件声 明是一种反模式，它让程序变得更难理解。如果你发现自己在 使用这个模式，那一定有更好的替代方式。

const 声明

const 的行为与 let 基本相同，唯一一个重要的区别是用它声 明变量时必须同时初始化变量，且尝试修改 const 声明的变量会导 致运行时错误

3.4 数据类型

ECMAScript有6种简单数据类型（也称为原始类型）： Undefined 、 Null 、 Boolean 、 Number 、 String 和 Symbol 。 Symbol （符号）是ECMAScript 6新增的。还有一种复

杂数据类型叫 Object （对象）。 Object 是一种无序名值对的集 合。因为在ECMAScript中不能定义自己的数据类型，所有值都可以用 上述7种数据类型之一来表示 只有7种数据类型似乎不足以表示全部 数据。但ECMAScript的数据类型很灵活，一种数据类型可以当作多种 数据类型来使用。

typeof 操作符

typeof 操作符就是为此而生的。对一个值使 用 typeof 操作符会返回下列字符串之一：
"undefined" 表示值未定义；
"boolean" 表示值为布尔值；
"string" 表示值为字符串；
"number" 表示值为数值；
"object" 表示值为对象（而不是函数）或 null ；
"function" 表示值为函数； "symbol" 表示值为符号。

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

Null 类型

Null 类型同样只有一个值，即特殊值 null 。逻辑上讲， null 值表示一个空对象指针

Boolean 类型

Boolean （布尔值）类型是ECMAScript中使用最频繁的类型之 一，有两个字面值： true 和 false 。这两个布尔值不同于数值， 因此 true 不等于1， false 不等于0。

Number 类型

ECMAScript中最有意思的数据类型或许就是 Number 了。 Number 类型使用IEEE 754格式表示整数和浮点值（在某些语言中也 叫双精度值）。不同的数值类型相应地也有不同的数值字面量格式。

String 类型

String （字符串）数据类型表示零或多个16位Unicode字符序 列。字符串可以使用双引号（"）、单引号（'）或反引号（`）标示

转换为字符串

有两种方式把一个值转换为字符串。首先是使用几乎所有值都有 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

Symbol 类型

符号需要使用 Symbol() 函数初始化。因为符号本身是原始类 型，所以 typeof 操作符对符号返回 symbol
符号没有字面量语法，这也是它们发挥作用的关键。按照规范， 你只要创建 Symbol() 实例并将其用作对象的新属性，就可以 保证它不会覆盖已有的对象属性，无论是符号属性还是字符串属 性最重要的是， Symbol() 函数不能用作构造函数，与 new 关键 字一起使用。这样做是为了避免创建符号包装对象，像使用 Boolean 、 String 或 Number 那样，它们都支持构造函数 且可用于初始化包含原始值的包装对象：

Object 类型

ECMAScript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

布尔操作符

布尔操作符跟相等操作符几乎同样重要。如 果没有能力测试两个值的关系，那么像 if-else 和循环这样的语句 也没什么用了。布尔操作符一共有3个：逻辑非、逻辑与和逻辑或。

逻辑非

逻辑非操作符由一个叹号（！）表示
如果操作数是对象，则返回 false 。
如果操作数是空字符串，则返回 true 。 如果操作数是非空字符串，则返回 false 。

如果操作数是数值0，则返回 true 。
如果操作数是非0数值（包括 Infinity ），则返回 false 。
如果操作数是 null ，则返回 true 。
如果操作数是 NaN ，则返回 true 。
如果操作数是 undefined ，则返回 true 。

逻辑与

逻辑与操作符由两个和号（ && ）表示
逻辑与操作符可用于任何类型的操作数，不限于布尔值。
如果有 操作数不是布尔值，则逻辑与并不一定会返回布尔值，而是遵循 如下规则。
如果第一个操作数是对象，则返回第二个操作数。
如果第二个操作数是对象，则只有第一个操作数求值为 true 才会返回该对象。
如果两个操作数都是对象，则返回第二个操作数。
如果有一个操作数是 null ，则返回 null 。
如果有一个操作数是 NaN ，则返回 NaN 。
如果有一个操作数是 undefined ，则返回 undefined 。

逻辑或

逻辑或操作符由两个管道符（ || ）表示，
与逻辑与类似，如果有一个操作数不是布尔值，那么逻辑或操作 符也不一定返回布尔值。它遵循如下规则。
如果第一个操作数是对象，则返回第一个操作数。
如果第一个操作数求值为 false ，则返回第二个操作数。 如果两个操作数都是对象，则返回第一个操作数。
如果两个操作数都是 null ，则返回 null 。
如果两个操作数都是 NaN ，则返回 NaN 。
如果两个操作数都是 undefined ，则返回 undefined 。

关系操作符

关系操作符执行比较两个值的操作，包括小于（ < ）、大于 （ > ）、小于等于（ <= ）和大于等于（ >= ），用法跟数学课上学 的一样。这几个操作符都返回布尔值，
如果操作数都是数值，则执行数值比较。
如果操作数都是字符串，则逐个比较字符串中对应字符的编码。
如果有任一操作数是数值，则将另一个操作数转换为数值，执行 数值比较。
如果有任一操作数是对象，则调用其 valueOf() 方法，取得结 果后再根据前面的规则执行比较。
如果没有 valueOf() 操作 符，则调用 toString() 方法，取得结果后再根据前面的规则 执行比较。
如果有任一操作数是布尔值，则将其转换为数值再执行比较。

if 语句

if 语句是使用最频繁的语句之一
if (condition) statement1 else statement2
这里的条件（ condition ）可以是任何表达式，并且求值结果 不一定是布尔值。ECMAScript会自动调用 Boolean() 函数将这个表 达式的值转换为布尔值。如果条件求值为 true ，则执行语句 statement1 ； 如果条件求值为 false ，则执行语句 statement2 。这里的语句可能是一行代码，也可能是一个代码块 （即包含在一对花括号中的多行代码）。

```
if (i > 25)
   console.log("Greater than 25."); // 只有一行代码 的语句
else {
   console.log("Less than or equal to 25."); // 一 个语句块 }
```

do-while 语句

do-while 语句是一种后测试循环语句，即循环体中的代码执 行后才会对退出条件进行求值

```
let i = 0;
do {
 i += 2;
} while (i < 10);
```

在这个例子中，只要 i 小于10，循环就会重复执行。 i 从0开 始，每次循环递增2

while 语句

while 语句是一种先测试循环语句，即先检测退出条件，再执 行循环体内的代码。因此，while 循环体内的代码有可能不会执 行。

```
let i = 0;
while (i < 10) {
 i += 2;
}
```

在这个例子中，变量 i 从0开始，每次循环递增2。只要 i 小于 10，循环就会继续。

for 语句

for 语句也是先测试语句，只不过增加了进入循环之前的初始 化代码，以及循环执行后要执行的表达式

```
let count = 10;
for (let i = 0; i < count; i++) {     console.log(i);
}
```

以上代码在循环开始前定义了变量 i 的初始值为0。然后求值条 件表达式，如果求值结果为 true （i < count），则执行循环 体。因此循环体也可能不会被执行。如果循环体被执行了，则循环后 表达式也会执行，以便递增变量 i 。

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

```
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。每次执行循环，都会给变量 propName 赋予一个 window 对象 的属性作为值，直到 window 的所有属性都被枚举一遍。与 for 循 环一样，这里控制语句中的 const 也不是必需的。但为了确保这个 局部变量不被修改，推荐使用 const 。

如果 for-in 循环要迭代的变量是 null 或 undefined ，则 不执行循环体

for-of 语句

for-of 语句是一种严格的迭代语句，用于遍历可迭代对象的元 素，

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

在这个例子中，我们使用 for-of 语句显示了一个包含4个元素 的数组中的所有元素。循环会一直持续到将所有元素都迭代完。与 for 循环一样，这里控制语句中的 const 也不是必需的。但为了确 保这个局部变量不被修改，推荐使用 const 。

for-of 循环会按照可迭代对象的 next() 方法产生值的顺序 迭代元素。关于可迭代对象，本书将在第7章详细介绍。

如果尝试迭代的变量不支持迭代，则 for-of 语句会抛出错误

break 和 continue 语句

break 和 continue 语句为执行循环代码提供了更严格的控制 手段。其中， break 语句用于立即退出循环，强制执行循环后的下 一条语句。而 continue 语句也用于立即退出循环，但会再次从循 环顶部开始执行。

```
let num = 0;
for (let i = 1; i < 10; i++) {
 if (i % 5 == 0) {
 break;
} num++;
} console.log(num); // 4
```

在上面的代码中， for 循环会将变量 i 由1递增到10。而在循环 体内，有一个 if 语句用于检查 i 能否被5整除（使用取模操作 符）。如果是，则执行 break 语句，退出循环。变量 num 的初始值 为0，表示循环在退出前执行了多少次。当 break 语句执行后，下一 行执行的代码是 console.log(num) ，显示4。之所以循环执行了4 次，是因为当 i 等于5时， break 语句会导致循环退出，该次循环 不会执行递增 num 的代码。如果将 break 换成 continue

with 语句

with 语句的用途是将代码作用域设置为特定的对象

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

这里， with 语句用于连接 location 对象。这意味着在这个 语句内部，每个变量首先会被认为是一个局部变量。如果没有找到该 局部变量，则会搜索 location 对象，看它是否有一个同名的属性。如果有，则该变量会被求值为 location 对象的属性。 严格模式不允许使用 with 语句，否则会抛出错误。

switch 语句

switch 语句是与 if 语句紧密相关的一种流控制语句，从其他 语言借鉴而来。ECMAScript中 switch 语句跟C语言中 switch 语 句的语法非常相似

```
switch (expression) {
case value1:
 statement break;
case value2:
 statement break;
case value3:
 statement break;
case value4:
 statement break;
default:
 statement
}
```

这里的每个 case （条件/分支）相当于： “如果表达式等于后面 的值，则执行下面的语句。” break 关键字会导致代码执行跳出 switch 语句。如果没有 break ，则代码会继续匹配下一个条件。 default 关键字用于在任何条件都没有满足时指定默认执行的 语句（相当于 else 语句）

3.7 函数

函数对任何语言来说都是核心组件，因为它们可以封装语句，然 后在任何地方、任何时间执行。

function 关键字声明，后跟一组参数，然后是函数体。

小结

ECMAScript中的基本数据类型包括 Undefined 、 Null 、 Boolean 、 Number 、 String 和 Symbol

与其他语言不同，ECMAScript不区分整数和浮点值，只有 Number 一种数值数据类型。

Object 是一种复杂数据类型，它是这门语言中所有对象的基 类。

严格模式为这门语言中某些容易出错的部分施加了限制。

ECMAScript提供了C语言和类C语言中常见的很多基本操作符， 包括数学操作符、布尔操作符、关系操作符、相等操作符和赋值 操作符等。

这门语言中的流控制语句大多是从其他语言中借鉴而来的，比如 if 语句、 for 语句和 switch 语句等。 ECMAScript中的函数与其他语言中的函数不一样。

不需要指定函数的返回值，因为任何函数可以在任何时候返回任 何值。

不指定返回值的函数实际上会返回特殊值 undefined 。

第 4 章 变量、作用域与内存

本章内容

通过变量使用原始值与引用值

理解执行上下文

理解垃圾回收

原始值与引用值

ECMAScript变量可以包含两种不同类型的数据：原始值和引用 值。原始值（primitive value）就是最简单的数据，引用值（reference value）则是由多个值构成的对象。

动态属性

原始值和引用值的定义方式很类似，都是创建一个变量，然后给 它赋一个值。不过，在变量保存了这个值之后，可以对这个值做什 么，则大有不同。对于引用值而言，可以随时添加、修改和删除其属 性和方法。

首先创建了一个对象，并把它保存在变量 person 中。然 后，给这个对象添加了一个名为 name 的属性，并给这个属性赋值了 一个字符串 "Nicholas" 。在此之后，就可以访问这个新属性，直到对象被销毁或属性被显式地删除。 原始值不能有属性，尽管尝试给原始值添加属性不会报错。

```
let name = "Nicholas";
name.age = 27;
console.log(name.age); // undefined
```

作用域链增强

虽然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，这个上下文在代码 执行后会被删除。通常在两种情况下会出现这个现象，即代码执行到 下面任意一种情况时

try / catch 语句的 catch 块 with 语句 这两种情况下，都会在作用域链前端添加一个变量对象。对 with 语句来说，会向作用域链前端添加指定的对象；对 catch 语 句而言，则会创建一个新的变量对象，这个变量对象会包含要抛出的 错误对象的声明。

```
function buildUrl() {
    let qs = "?debug=true";
        with(location){
             let url = href + qs;
        }
    return url;
}
```

变量声明

ES6之后，JavaScript的变量声明经历了翻天覆地的变化。直到 ECMAScript 5.1， var 都是声明变量的唯一关键字。ES6不仅增加了 let 和 const 两个关键字，而且还让这两个关键字压倒性地超越 var 成为首选。

使用 var 的函数作用域声明

在使用 var 声明变量时，变量会被自动添加到最接近的上下 文。在函数中，最接近的上下文就是函数的局部上下文。在 with 语句中，最接近的上下文也是函数上下文。如果变量未经 声明就被初始化了，那么它就会自动被添加到全局上下文，如下 面的例子所示

```
function add(num1, num2) {
    var sum = num1 + num2;
    return sum;
}
let result = add(10, 20); // 30
console.log(sum); // 报错：sum在这里不是 有效变量
```

注意 未经声明而初始化变量是JavaScript编程中一个非常常见 的错误，会导致很多问题。为此，读者在初始化变量之前一定 要先声明变量。在严格模式下，未经声明就初始化变量会报错。

使用 let 的块级作用域声明

ES6新增的 let 关键字跟 var 很相似，但它的作用域是块级 的，这也是JavaScript中的新概念。块级作用域由最近的一对包含 花括号 {} 界定。换句话说， if 块、 while 块、 function 块，甚至连单独的块也是 let 声明变量的作用域

```
if (true) {
 let a;
} console.log(a);// ReferenceError: a没有定义
while (true) {
 let b;
} console.log(b); // ReferenceError: b没有定义
function foo() {
 let c;
```

```
} console.log(c); // ReferenceError: c没有定义
// 这没什么可奇怪的
// var声明也会导致报错
// 这不是对象字面量，而是一个独立的块
// JavaScript解释器会根据其中内容识别出它来
{
    let d;
} console.log(d); // ReferenceError: d没有定义
```

let 与 var 的另一个不同之处是在同一作用域内不能声明两 次。重复的 var 声明会被忽略，而重复的 let 声明会抛出 SyntaxError

使用 const 的常量声明

除了 let ， ES6同时还增加了 const 关键字。使用 const 声 明的变量必须同时初始化为某个值。一经声明，在其生命周期的 任何时候都不能再重新赋予新值

```
const a; // SyntaxError: 常量声明时没有初始化
const b = 3;
console.log(b); // 3
b = 4; // TypeError: 给常量赋值
```

const 除了要遵循以上规则，其他方面与 let 声明是一样的
const 声明只应用到顶级原语或者对象。换句话说，赋值为对 象的 const 变量不能再被重新赋值为其他引用值，但对象的键 则不受限制。

标识符查找

当在特定上下文中为读取或写入而引用一个标识符时，必须通过 搜索确定这个标识符表示什么。搜索开始于作用域链前端，以给 定的名称搜索对应的标识符。如果在局部上下文中找到该标识 符，则搜索停止，变量确定；如果没有找到变量名，则继续沿作 用域链搜索。（注意，作用域链中的对象也有一个原型链，因此 搜索可能涉及每个对象的原型链。）这个过程一直持续到搜索至 全局上下文的变量对象。如果仍然没有找到标识符，则说明其未读书笔记。

```
var color = 'blue';
function getColor() {
 return color;
}
console.log(getColor()); // 'blue'
```

标识符查找并非没有代价。访问局部变量比访问全局变 量要快，因为不用切换作用域。不过，JavaScript引擎在优化标 识符查找上做了很多工作，将来这个差异可能就微不足道了。

通过 const 和 let 声明提升性能

ES6增加这两个关键字不仅有助于改善代码风格，而且同样有助 于改进垃圾回收的过程。因为 const 和 let 都以块（而非函 数）为作用域，所以相比于使用 var ，使用这两个新关键字可 能会更早地让垃圾回收程序介入，尽早回收应该回收的内存。在 块作用域比函数作用域更早终止的情况下，这就有可能发生。

小结

JavaScript变量可以保存两种类型的值：原始值和引用值。原始值 可能是以下6种原始数据类型之一： Undefined 、 Null 、 Boolean 、 Number 、 String 和 Symbol 。原始值和引用值有 以下特点。
原始值大小固定，因此保存在栈内存上。
从一个变量到另一个变量复制原始值会创建该值的第二个副本。
引用值是对象，存储在堆内存上。
包含引用值的变量实际上只包含指向相应对象的一个指针，而不 是对象本身。
从一个变量到另一个变量复制引用值只会复制指针，因此结果是 两个变量都指向同一个对象。
typeof 操作符可以确定值的原始类型，而 instanceof 操作 符用于确保值的引用类型。
任何变量（不管包含的是原始值还是引用值）都存在于某个执行 上下文中（也称为作用域）。这个上下文（作用域）决定了变量的生 命周期，以及它们可以访问代码的哪些部分。
执行上下文可以总结如 下。 执行上下文分全局上下文、函数上下文和块级上下文。
代码执行流每进入一个新上下文，都会创建一个作用域链，用于 搜索变量和函数。
函数或块的局部上下文不仅可以访问自己作用域内的变量，而且 也可以访问任何包含上下文乃至全局上下文中的变量。 全局上下文只能访问全局上下文中的变量和函数，不能直接访问 局部上下文中的任何数据。 变量的执行上下文用于确定什么时候释放内存。 JavaScript是使用垃圾回收的编程语言，开发者不需要操心内存分 配和回收。JavaScript的垃圾回收程序可以总结如下。

第 5 章 基本引用类型

本章内容：
理解对象
基本JavaScript数据类型
原始值与原始值包装类型

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();
```

在不给 Date 构造函数传参数的情况下，创建的对象将保存当前日期和时间
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。

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

Date.UTC() 方法也返回日期的毫秒表示，但使用的是跟 Date.parse() 不同的信息来生成这个值。

```
// GMT时间2000年1月1日零点
let y2k = new Date(Date.UTC(2000, 0));

// GMT时间2005年5月5日下午5点55分55秒
let allFives = new Date(Date.UTC(2005, 4, 5, 17, 55, 55));
```

继承的方法

与其他类型一样， Date 类型重写了 toLocaleString() 、 toString() 和 valueOf() 方法。但与其他类型不同，重写后这些方法 的返回值不一样。

日期格式化方法

Date 类型有几个专门用于格式化日期的方法，它们都会返回字符 串：
toDateString() 显示日期中的周几、月、日、年（格式特定于实 现）；
toTimeString() 显示日期中的时、分、秒和时区（格式特定于实 现）；
toLocaleDateString() 显示日期中的周几、月、日、年（格式特 定于实现和地区）；
toLocaleTimeString() 显示日期中的时、分、秒（格式特定于实 现）；
toUTCString() 显示完整的UTC日期（格式特定于实现）。
这些方法的输出与 toLocaleString() 和 toString() 一样，会 因浏览器而异。因此不能用于在用户界面上一致地显示日期。
注意 还有一个方法叫 toGMTString()，这个方法跟 toUTCString() 是一样的，目的是为了向后兼容。不过，规范建议新 代码使用 toUTCString()。

日期/时间组件方法

Date 类型剩下的方法（见下表）直接涉及取得或设置日期值的特定 部分。注意表中“UTC日期”，指的是没有时区偏移（将日期转换为GMT）时 的日期。

getTime()	返回日期的毫秒表示；与 valueOf() 相同
setTime(*milliseconds*)	设置日期的毫秒表示，从而修改整个日期
getFullYear()	返回4位数年（即2019而不是19）
getUTCFullYear()	返回UTC日期的4位数年
setFullYear(*year*)	设置日期的年（*year* 必须是4位数）
setUTCFullYear(*year*)	设置UTC日期的年（*year* 必须是4位数）
getMonth()	返回日期的月（0表示1月，11表

	示12月）
`getUTCMonth()`	返回UTC日期的月（0表示1月，11表示12月）

方法	说明
`setMonth(month)`	设置日期的月（ *month* 为大于0的数值，大于11加年）
`setUTCMonth(month)`	设置UTC日期的月（ *month* 为大于0的数值，大于11加年）
`getDate()`	返回日期中的日（1~31）
`getUTCDate()`	返回UTC日期中的日（1~31）
`setDate(date)`	设置日期中的日（如果 *date* 大于该月天数，则加月）
`setUTCDate(date)`	设置UTC日期中的日（如果 *date* 大于该月天数，则加月）
`getDay()`	返回日期中表示周几的数值（0表示周日，6表示周六）

方法	说明
`getUTCDay()`	返回UTC日期中表示周几的数值（0表示周日，6表示周六）
`getHours()`	返回日期中的时（0~23）
`getUTCHours()`	返回UTC日期中的时（0~23）
`setHours(hours)`	设置日期中的时（如果 *hours* 大于23，则加日）
`setUTCHours(hours)`	设置UTC日期中的时（如果 *hours* 大于23，则加日）
`getMinutes()`	返回日期中的分（0~59）
`getUTCMinutes()`	返回UTC日期中的分（0~59）
`setMinutes(minutes)`	设置日期中的分（如果 *minutes* 大于59，则加时）
`setUTCMinutes(minutes)`	设置UTC日期中的分（如果 *minutes* 大于59，则加时）

方法	说明
`getSeconds()`	返回日期中的秒（0~59）
`getUTCSeconds()`	返回UTC日期中的秒（0~59）
`setSeconds(seconds)`	设置日期中的秒（如果 *seconds* 大于59，则加分）

setUTCSeconds(seconds)	设置UTC日期中的秒（如果 seconds 大于59，则加分）
getMilliseconds()	返回日期中的毫秒
getUTCMilliseconds()	返回UTC日期中的毫秒
setMilliseconds(milliseconds)	设置日期中的毫秒
setUTCMilliseconds(milliseconds) setMilliseconds(milliseconds)	设置UTC日期中的毫秒 毫秒
setUTCMilliseconds(milliseconds)	设置UTC日期中的毫秒
getTimezoneOffset()	返回以分钟计的UTC与本地时区的偏移量（如美国EST即"东部标准时间"返回300，进入夏令时的地区可能有所差异）

5.2 RegExp

ECMAScript通过 RegExp 类型支持正则表达式。正则表达式使用类似Perl的简洁语法来创建：

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

这个正则表达式的 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。每个正则表达式

RegExp

ECMAScript通过 RegExp 类型支持正则表达式。正则表达式使用类似 Perl的简洁语法来创建

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

这个正则表达式的 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。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。下面给 出了表示匹配模式的标记。

g： 全局模式，表示查找字符串的全部内容，而不是找到第一个匹配 的内容就结束。 i： 不区分大小写，表示在查找匹配时忽略 pattern 和字符串的大 小写。 m： 多行模式，表示查找到一行文本末尾时会继续查找。 y： 粘附模式，表示只查找从 lastIndex 开始及之后的字符串。 u： Unicode模式，启用Unicode匹配。 s： dotAll 模式，表示元字符 . 匹配任何字符（包括 \n 或 \r ）。

```
// 匹配字符串中的所有"at"
let pattern1 = /at/g;
// 匹配第一个"bat"或"cat"，忽略大小写
let pattern2 = /[bc]at/i;
// 匹配所有以"at"结尾的三字符组合，忽略大小写
let pattern3 = /.at/gi;
```

与其他语言中的正则表达式类似，所有元字符在模式中也必须转义，包 括：

([{ \ ^ $ |)] } ? * + .

元字符在正则表达式中都有一种或多种特殊功能，所以要匹配上面这些 字符本身，就必须使用反斜杠来转义。

字面量模式	对应的字符串
/\[bc\]at/	"\\[bc\\]at"
/\.at/	"\\.at"
/name\/age/	"name\\/age"
/\d.\d{1,2}/	"\\d.\\d{1,2}"
/\w\\hello\\123/	"\\w\\\\hello\\\\123"

此外，使用 RegExp 也可以基于已有的正则表达式实例

RegExp 实例属性

每个 RegExp 实例都有下列属性，提供有关模式的各方面信息。
global： 布尔值，表示是否设置了 g 标记。
ignoreCase： 布尔值，表示是否设置了 i 标记。
unicode： 布尔值，表示是否设置了 u 标记。
sticky： 布尔值，表示是否设置了 y 标记。
lastIndex： 整数，表示在源字符串中下一次搜索的开始位置，始 终从0开始。 multiline： 布尔值，表示是否设置了 m 标记。
dotAll： 布尔值，表示是否设置了 s 标记。
source： 正则表达式的字面量字符串（不是传给构造函数的模式字 符串），没有开头和结尾的斜杠。
flags： 正则表达式的标记字符串。始终以字面量而非传入构造函数 的字符串模式形式返回（没有前后斜杠）。
source 和 flags 属性返回的是规范化之后可以在字面量中使用的 形式。

RegExp 实例方法

RegExp 实例的主要方法是 exec()，主要用于配合捕获组使用。这 个方法只接收一个参数，即要应用模式的字符串。如果找到了匹配项，则返 回包含第一个匹配信息的数组；如果没找到匹配项，则返回 null 。返回 的数组虽然是 Array 的实例，但包含两个额外的属性： index 和 input 。 index 是字符串中匹配模式的起始位置， input 是要查找的 字符串。这个数组的第一个元素是匹配整个模式的字符串，其他元素是与表 达式中的捕获组匹配的字符串。如果模式中没有捕获组，则数组只包含一个 元素。

如果模式设置了全局标记，则每次调用 exec() 方法会返回一个匹配 的信息。如果没有设置全局标记，则无论对同一个字符串调用多少次 exec()，也只会返回第一个匹配的信息。
没有设置全局标记，调用 exec() 只返回第一 个匹配项（"cat"）。
lastIndex 在非全局模式下始终不变。

如果在这个模式上设置了 g 标记，则每次调用 exec() 都会在字符串 中向前搜索下一个匹配项，
这次模式设置了全局标记，因此每次调用 exec() 都会返回字符串中 的下一个匹配项，直到搜索到字符串末尾。注意模式的 lastIndex 属性 每次都会变化。在全局匹配模式下，每次调用 exec() 都会更新 lastIndex 值，以反映上次匹配的最后一个字符的索引。 如果模式设置了粘附标记 y，则每次调用 exec() 就只会在 lastIndex 的位置上寻找匹配项。粘附标记覆盖全局标记。

正则表达式的另一个方法是 test()，接收一个字符串参数。如果输 入的文本与模式匹配，则参数返回 true，否则返回 false 。这个方法适 用于只想测试模式是否匹配，而不需要实际匹配内容的情况。 test() 经 常用在 if 语句中

这里的模式是通过 RegExp 构造函数创建的，但 toLocaleString() 和 toString() 返回的都是其字面量的形式。
注意 正则表达式的 valueOf() 方法返回正则表达式本身。

RegExp 构造函数属性

RegExp 构造函数本身也有几个属性。（在其他语言中，这种属性被 称为静态属性。）这些属性适用于作用域中的所有正则表达式，而且会根据 最后执行的正则表达式操作而变化。这些属性还有一个特点，就是可以通过 两种不同的方式访问它们。换句话说，每个属性都有一个全名和一个简写。 下表列出了 RegExp 构造函数的属性

全名	简写	说明
`input`	`$_`	最后搜索的字符串
`lastMatch`	`$&`	最后匹配的文本
`lastParen`	`$+`	最后匹配的捕获组
`leftContext`	`$``	`input` 字符串中出现在 `lastMatch` 前面的文本

全名	简写	说明
`rightContext`	`$'`	`input` 字符串中出现在 `lastMatch` 后面的文本

通过这些属性可以提取出与 exec() 和 test() 执行的操作相关的信 息。
RegExp 还有其他几个构造函数属性，可以存储最多9个捕获组的匹配 项。这些属性通过 RegExp.$1~RegExp.$9 来访问，分别包含第1~9个捕 获组的匹配项。
RegExp 构造函数的所有属性都没有任何Web标准出处，因此不 要在生产环境中使用它们。

模式局限

虽然ECMAScript对正则表达式的支持有了长足的进步，但仍然缺少Perl 语言中的一些高级特性。
下列特性目前还没有得到ECMAScript的支持（想 要了解更多信息，可以参考Regular-Expressions.info网站）：
\A 和 \Z 锚（分别匹配字符串的开始和末尾）
联合及交叉类
原子组
x （忽略空格）匹配模式 条件式匹配
正则表达式注释
虽然还有这些局限，但ECMAScript的正则表达式已经非常强大，可以 用于大多数模式匹配任务。

原始值包装类型

为了方便操作原始值，ECMAScript提供了3种特殊的引用类型： Boolean 、 Number 和 String 。这些类型具有本章介绍的其他引用类 型一样的特点，但也具有与各自原始类型对应的特殊行为。每当用到某个原 始值的方法或属性时，后台都会创建一个相应原始包装类型的对象，从而暴 露出操作原始值的各种方法。

Boolean

Boolean 是对应布尔值的引用类型。要创建一个 Boolean 对象，就 使用 Boolean 构造函数并传入 true 或 false

Boolean 的实例会重写 valueOf() 方法，返回一个原始值 true 或 false 。 toString() 方法被调用时也会被覆盖，返回字符 串 "true" 或 "false" 。不过， Boolean 对象在ECMAScript中用得很 少。

Number

Number 是对应数值的引用类型。要创建一个 Number 对象，就使 用 Number 构造函数并传入一个数值

与 Boolean 类型一样， Number 类型重写了 valueOf() 、 toLocaleString() 和 toString() 方法。 valueOf() 方法返回 Number 对象表示的原始数值，另外两个方法返回数值字符串。
toString() 方法可选地接收一个表示基数的参数，并返回相应基数形式 的数值字符串，
toFixed() 方法返回包含指定小数点位数的数值字符串
toExponential() 也接收一个参数，表示结果中小数的位数。
注意: toPrecision() 方法可以表示带1~21个小数位的数值。某些浏 览器可能支持更大的范围，但这是通常被支持的范围
与 Boolean 对象类似， Number 对象也为数值提供了重要能力。但 是，考虑到两者存在同样的潜在问题，因此并不建议直接实例化 Number 对象。在处理原始数值和引用数值时， typeof 和 instacnceof 操作符 会返回不同的结果

String

String 是对应字符串的引用类型。要创建一个 String 对象，使 用 String 构造函数并传入一个数值，
String 对象的方法可以在所有字符串原始值上调用。3个继承的方 法 valueOf() 、toLcaleString() 和 toString() 都返回对象的原 始字符串值。 每个 String 对象都有一个 length 属性，表示字符串中字符的数 量。
charAt() 方法返回给定索引位置的字符，由传给方法的整数 参数指定。
charCodeAt() 方法可以查看指定码元的字符编码
fromCharCode() 方法用于根据给定的UTF-16码元创建字符串中的 字符。这个方法可以接受任意多个数值，并返回将所有数值对应的字符 拼接起来的字符串

字符串操作方法

本节介绍几个操作字符串值的方法。首先是 concat() ，用于将一个 或多个字符串拼接成一个新字符串。

```
let stringValue = "hello ";
let result = stringValue.concat("world");
console.log(result); // "hello world"
console.log(stringValue); // "hello"
```

slice() 、 substr() 和 substring() 。这3个方法都返回调用 它们的字符串的一个子字符串，而且都接收一或两个参数。第一个参数 表示子字符串开始的位置，第二个参数表示子字符串结束的位置。
substring() 方法会将所有负参数值都转换 为0

字符串位置方法

有两个方法用于在字符串中定位子字符串： indexOf() 和 lastIndexOf() 。
这两个方法从字符串中搜索传入的字符串，并返 回位置（如果没找到，则返回 -1 ）。
两者的区别在于： indexOf() 方法从字符串开头开始查找子字符串，而 lastIndexOf() 方法从字符串末尾开始查找子字符串。

字符串包含方法

startsWith() 、 endsWith() 和 includes() 。这些方法 都会从字符串中搜索传入的字符串，并返回一个表示是否包含的布尔 值。它们的区别在于， startsWith() 检查开始于索引0的匹配项， endsWith() 检查开始于索引

(string.length - substring.length) 的匹配项，而 includes() 检查整个字符 串
startsWith() 和 includes() 方法接收可选的第二个参数，表示 开始搜索的位置。如果传入第二个参数，则意味着这两个方法会从指定 位置向着字符串末尾搜索，忽略该位置之前的所有字符。

endsWith() 方法接收可选的第二个参数，表示应该当作字符串末尾 的位置。如果不提供这个参数，那么默认就是字符串长度。如果提供这 个参数，那么就好像字符串只有那么多字符一样

trim() 方法
ECMAScript在所有字符串上都提供了 trim() 方法。这个方法会创建 字符串的一个副本，删除前、后所有空格符，再返回结果。
repeat() 方法
ECMAScript在所有字符串上都提供了 repeat() 方法。这个方法接收 一个整数参数，表示要将字符串复制多少次，然后返回拼接所有副本后 的结果。

```
let stringValue = "na ";
console.log(stringValue.repeat(16) + "batman");
// na na na na na na na na na na na na na na na na batman
```

padStart() 和 padEnd() 方法
padStart() 和 padEnd() 方法会复制字符串，如果小于指定长 度，则在相应一边填充字符，直至满足长度条件。这两个方法的第一个 参数是长度，第二个参数是可选的填充字符串，默认为空格

可选的第二个参数并不限于一个字符。如果提供了多个字符的字符串， 则会将其拼接并截断以匹配指定长度。此外，如果长度小于或等于字符 串长度，则会返回原始字符串
字符串迭代与解构
字符串的原型上暴露了一个 @@iterator 方法，表示可以迭代字符串 的每个字符。
在 for-of 循环中可以通过这个迭代器按序访问每个字符
有了这个迭代器之后，字符串就可以通过解构操作符来解构了。比如， 可以更方便地把字符串分割为字符数组

字符串大小写转换
大小写转换，包括4个方法：
toLowerCase() 、 toLocaleLowerCase() 、 toUpperCase() 和 toLocaleUpperCase() 。
toLowerCase() 和 toUpperCase() 方法是原来就有的方法，与 java.lang.String 中的方法同名。
toLocaleLowerCase() 和 toLocaleUpperCase() 方法旨在基于特定地区实现
toLowerCase() 和 toLocaleLowerCase() 都返回 hello world ，而 toUpperCase() 和 toLocaleUpperCase() 都返回 HELLO WORLD 。通常，如果不知 道代码涉及什么语言，则最好使用地区特定的转换方法

字符串模式匹配方法
String 类型专门为在字符串中实现模式匹配设计了几个方法。第一 个就是 match() 方法，这个方法本质上跟 RegExp 对象的 exec() 方法相同。 match() 方法接收一个参数，可以是一个正则表达式字符 串，也可以是一个 RegExp 对象。

```
let text = "cat, bat, sat, fat";
let pattern = /.at/;
// 等价于pattern.exec(text)
let matches = text.match(pattern);
console.log(matches.index); // 0
console.log(matches[0]); // "cat"
console.log(pattern.lastIndex); // 0
```

match() 方法返回的数组与 RegExp 对象的 exec() 方法返回的数 组是一样的：第一个元素是与整个模式匹配的字符串，其余元素则是与 表达式中的捕获组匹配的字符串（如果有的话）
search()： 这个方法唯一的参数与 match() 方法一样：正则表达式字符串或 RegExp 对象。这个方法 返回模式第一个匹配的位置索引，如果没找到则返回-1。
localeCompare() 方法
最后一个方法是 localeCompare() ，这个方法比较两个字符串，返 回如下3个值中的一个
如果按照字母表顺序，字符串应该排在字符串参数前头，则返回负 值。（通常是 -1 ，具体还要看与实际值相关的实现。）
如果字符串与字符串参数相等，则返回 0 。

如果按照字母表顺序，字符串应该排在字符串参数后头，则返回正 值。（通常是 1 ，具体还要看与实际值相关的实现。）

HTML方法

早期的浏览器开发商认为使用JavaScript动态生成HTML标签是一个需 求。因此，早期浏览器扩展了规范，增加了辅助生成HTML标签的方 法。下表总结了这些HTML方法。不过，这些方法基本上已经没有人使 用了，因为结果通常不是语义化的标记。

方法	输出
anchor(*name*)	*string*
big()	<big>*string*</big>
bold()	*string*
fixed()	<tt>*string*</tt>

fontcolor(*color*)	*string*
fontsize(*size*)	*string*
italics()	*string*
link(url)	*string*
small()	*string*
strike()	<strike>*string*</strike>
sub()	_{*string*}
sup()	^{*string*}

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。除了这些， Global 对象上还有另外一些方法。

eval() 方法

最后一个方法可能是整个ECMAScript语言中最强大的了，它就是 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（JavaScript）字符串。来看一个例 子

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

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);
}
window.sayColor(); // "red"
```

这里定义了一个名为 color 的全局变量和一个名为 sayColor() 的 全局函数。在 sayColor() 内部，通过 window.color 访问了 color 变量，说明全局变量变成了 window 的属性。接着，又通过 window 对象直接调用了 window.sayColor() 函数，从而输出字 符串。

Math

Math 对象有一些属性，主要用于保存数学中的一些特殊值。下表列 出了这些属性。

属性	说明
Math.E	自然对数的基数e的值
Math.LN10	10为底的自然对数
Math.LN2	2为底的自然对数
Math.LOG2E	以2为底e的对数
Math.LOG10E	以10为底e的对数
Math.PI	π的值
Math.SQRT1_2	1/2的平方根
Math.SQRT2	2的平方根

这些值的含义和用法超出了本书的范畴，但都是ECMAScript规范定义 的，并可以在你需要时使用。

min() 和 max() 方法

Math 对象也提供了很多辅助执行简单或复杂数学计算的方法。 min() 和 max() 方法用于确定一组数值中的最小值和最大值。这两 个方法都接收任意多个参数

```
let max = Math.max(3, 54, 32, 16);
 console.log(max); // 54
let min = Math.min(3, 54, 32, 16);
 console.log(min); // 3
```

要知道数组中的最大值和最小值，可以像下面这样使用扩展操作符

```
let values = [1, 2, 3, 4, 5, 6, 7, 8];
let max = Math.max(...values);
```

舍入方法

把小数值舍入为整数的4个方法： Math.ceil() 、 Math.floor() 、 Math.round() 和 Math.fround() 。这几个 方法处理舍入的方式如下所述。

```
Math.ceil() 方法始终向上舍入为最接近的整数。
Math.floor() 方法始终向下舍入为最接近的整数。
Math.round() 方法执行四舍五入。
Math.fround() 方法返回数值最接近的单精度（32位）浮点值 表示。

console.log(Math.ceil(25.9)); // 26
console.log(Math.ceil(25.5)); // 26
console.log(Math.ceil(25.1)); // 26
console.log(Math.round(25.9)); // 26
console.log(Math.round(25.5)); // 26
```

```
console.log(Math.round(25.1)); // 25
console.log(Math.fround(0.4)); // 0.4000000059604645
console.log(Math.fround(0.5)); // 0.5
console.log(Math.fround(25.9)); // 25.899999618530273
console.log(Math.floor(25.9)); // 25
console.log(Math.floor(25.5)); // 25
console.log(Math.floor(25.1)); // 25
```

random() 方法

Math.random() 方法返回一个0~1范围内的随机数，其中包含0但不 包含1。对于希望显示随机名言或随机新闻的网页，这个方法是非常方 便的。可以基于如下公式使用 Math.random() 从一组整数中随机选 择一个数

```
number = Math.floor(Math.random() * total_number_of_choices +
first_possible_value)
```

这里使用了 Math.floor() 方法，因为 Math.random() 始终返回 小数，即便乘以一个数再加上一个数也是小数。因此，如果想从1~10范 围内随机选择一个数

```
let num = Math.floor(Math.random() * 10 + 1);
```

小结

JavaScript中的对象称为引用值，几种内置的引用类型可用于创建特定 类型的对象。 引用值与传统面向对象编程语言中的类相似，但实现不同。 Date 类型提供关于日期和时间的信息，包括当前日期、时间及相关 计算。 RegExp 类型是ECMAScript支持正则表达式的接口，提供了大多数基 础的和部分高级的正则表达式功能。 JavaScript比较独特的一点是，函数实际上是 Function 类型的实例， 也就是说函数也是对象。因为函数也是对象，所以函数也有方法，可以用于 增强其能力。 由于原始值包装类型的存在，JavaScript中的原始值可以被当成对象来 使用。有3种原始值包装类型： Boolean 、 Number 和 String 。它们 都具备如下特点。 每种包装类型都映射到同名的原始类型。 以读模式访问原始值时，后台会实例化一个原始值包装类型的对象，借 助这个对象可以操作相应的数据。 涉及原始值的语句执行完毕后，包装对象就会被销毁。 当代码开始执行时，全局上下文中会存在两个内置对象： Global 和 Math 。其中，Global 对象在大多数ECMAScript实现中无法直接访问。 不过，浏览器将其实现为 window 对象。所有全局变量和函数都是 Global 对象的属性。 Math 对象包含辅助完成复杂计算的属性和方法。

第 6 章 集合引用类型

本章内容：

对象

数组与定型数组

Map 、 WeakMap 、 Set 以及 WeakSet 类型

Object

到目前为止，大多数引用值的示例使用的是 Object 类型。 Object 是ECMAScript中最常用的类型之一。虽然 Object 的实例没有 多少功能，但很适合存储和在应用程序间交换数据。 显式地创建 Object 的实例有两种方式。第一种是使用 new 操作符 和 Object 构造函数

另一种方式是使用对象字面量（object literal）表示法。对象字面量是 对象定义的简写形式，目的是为了简化包含大量属性的对象的创建。比 如，下面的代码定义了与前面示例相同的 person 对象，但使用的是对象 字面量表示法

```
let person = {
    name: "Nicholas" ,
    age: 29
};
```

注意： 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。

Array

Array 也是ECMAScript中最常用的类型了。
有几种基本的方式可以创建数组。一种是使用 Array 构造函数，比 如

```
let colors = new Array();
```

另一种创建数组的方式是使用数组字面量（array literal）表示法。数 组字面量是在中括号中包含以逗号分隔的元素列表

注意 与对象一样，在使用数组字面量表示法创建数组不会调用 Array 构造函数。

Array 构造函数还有两个ES6新增的用于创建数组的静态方法： from() 和 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。

数组索引

要取得或设置数组的值，需要使用中括号并提供相应值的数字索引
数组中元素的数量保存在 length 属性中，这个属性始终返回0或大 于0的值
注意 数组最多可以包含4 294 967 295个元素，这对于大多数编程任务 应该足够了。如果尝试添加更多项，则会导致抛出错误。以这个最大值 作为初始值创建数组，可能导致脚本运行时间过长的错误。

检测数组

一个经典的ECMAScript问题是判断一个对象是不是数组。在只有一个 网页（因而只有一个全局作用域）的情况下，使用 instanceof 操作符 就足矣

```
if (value instanceof Array){
 // 操作数组
}
```

使用 instanceof 的问题是假定只有一个全局执行上下文。如果网 页里有多个框架，则可能涉及两个不同的全局执行上下文，因此就会有两 个不同版本的 Array 构造函数。如果要把数组从一个框架传给另一个框 架，则这个数组的构造函数将有别于在第二个框架内本地创建的数组
为解决这个问题，ECMAScript提供了 Array.isArray() 方法。这 个方法的目的就是确定一个值是否为数组，而不用管它是在哪个全局执行 上下文中创建的。来看下面的例子

```
if (Array.isArray(value)){
 // 操作数组
}
```

迭代器方法

ECMAScript为数组定义了5个迭代方法。每个方法接收两个参数：以 每一项为参数运行的函数，以及可选的作为函数运行上下文的作用域对象 （影响函数中 this 的值）。
传给每个方法的函数接收3个参数：数组元 素、元素索引和数组本身。因具体方法而异，这个函数的执行结果可能会 也可能不会影响方法的返回值。数组的5个迭代方法如下。
every()： 对数组每一项都运行传入的函数，如果对每一项函数都 返回 true ， 则这个方法返回 true 。
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。
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。 在这些方法中， every() 和 some() 是最相似的，都是从数组中搜 索符合某个条件的元素。对 every() 来说，传入的函数必须对每一项都 返回 true ，它才会返回 true ； 否则，它就返回 false 。而对 some() 来说，只要有一项让传入的函数返回 true ， 它就会返回 true 。

复制和填充方法

ES6新增了两个方法：批量复制方法 fill() ，以及填充数组方法 copyWithin() 。这两个方法的函数签名类似，都需要指定既有数组实 例上的一个范围，包含开始索引，不包含结束索引。使用这个方法创建的 数组不能缩放。

使用 fill() 方法可以向一个已有的数组中插入全部或部分相同的 值。开始索引用于指定开始填充的位置，它是可选的。如果不提供结束索 引，则一直填充到数组末尾。负值索引从数组末尾开始计算。也可以将负 索引想象成数组长度加上它得到的一个正索引
fill() 静默忽略超出数组边界、零长度及方向相反的索引范围
与 fill() 不同， copyWithin() 会按照指定范围浅复制数组中的 部分内容，然后将它们插入到指定索引开始的位置。开始索引和结束索引 则与 fill() 使用同样的计算方法

转换方法

前面提到过，所有对象都有 toLocaleString() 、 toString() 和 valueOf() 方法。其中，valueOf() 返回的还是数组本身。而 toString() 返回由数组中每个值的等效字符串拼接而成的一个逗号分 隔的字符串。也就是说，对数组的每个值都会调用其 toString() 方 法，以得到最终的字符串。
首先是被显式调用的 toString() 和 valueOf() 方法，它们分别 返回了数组的字符串表示，即将所有字符串组合起来，以逗号分隔。最后 一行代码直接用 alert() 显示数组，因为 alert() 期待字符串，所以 会在后台调用数组的 toString() 方法，从而得到跟前面一样的结果。
toLocaleString() 方法也可能返回跟 toString() 和 valueOf() 相同的结果，但也不一定。在调用数组的 toLocaleString() 方法时，会得到一个逗号分隔的数组值的字符串。 它与另外两个方法唯一的区别是，为了得到最终的字符串，会调用数组每 个值的 toLocaleString() 方法，而不是 toString() 方法。

栈方法

ECMAScript给数组提供几个方法，让它看起来像是另外一种数据结 构。数组对象可以像栈一样，也就是一种限制插入和删除项的数据结构。 栈是一种后进先出（LIFO，Last-In-First-Out）的结构，也就是最近添加的 项先被删除。数据项的插入（称为推入，push）和删除（称为弹出，pop） 只在栈的一个地方发生，即栈顶。ECMAScript数组提供了 push() 和 pop() 方法，以实现类似栈的行为。
push() 方法接收任意数量的参数，并将它们添加到数组末尾，返回 数组的最新长度。 pop() 方法则用于删除数组的最后一项，同时减少数 组的 length 值，返回被删除的项。
这里创建了一个当作栈来使用的数组（注意不需要任何额外的代码， push() 和 pop() 都是数组的默认方法）。首先，使用 push() 方法 把两个字符串推入数组末尾，将结果保存在变量 count 中（结果为 2 ）
然后，再推入另一个值，再把结果保存在 count 中。因为现在数组 中有3个元素，所以 push() 返回 3 。在调用 pop() 时，会返回数组的 最后一项，即字符串 "black" 。此时数组还有两个元素
这里先初始化了包含两个字符串的数组，然后通过 push() 添加了第 三个值，第四个值是通过直接在位置3上赋值添加的。调用 pop() 时，返 回了字符串 "black" ，也就是最后添加到数组的字符串。

这里先初始化了包含两个字符串的数组，然后通过 push() 添加了第 三个值，第四个值是通过直接在位置3上赋值添加的。调用 pop() 时，返 回了字符串 "black" ，也就是最后添加到数组的字符串。

队列方法

就像栈是以LIFO形式限制访问的数据结构一样，队列以先进先出 （FIFO，First-In-First-Out）形式限制访问。队列在列表末尾添加数据，但 从列表开头获取数据。因为有了在数据末尾添加数据的 push() 方法，所 以要模拟队列就差一个从数组开头取得数据的方法了。这个数组方法叫 shift() ，它会删除数组的第一项并返回它，然后数组长度减1。使用 shift() 和 push()
ECMAScript也为数组提供了 unshift() 方法。顾名思义， unshift() 就是执行跟 shift() 相反的操作：在数组开头添加任意多 个值，然后返回新的数组长度。通过使用 unshift() 和 pop() ，可以在相反方向上模拟队列，即在数组开头添加新数据，在数组末尾取得数 据，
就像栈是以LIFO形式限制访问的数据结构一样，队列以先进先出 （FIFO，First-In-First-Out）形式限制访问。队列在列表末尾添加数据，但 从列表开头获取数据。因为有了在数据末尾添加数据的 push() 方法，所 以要模拟队列就差一个从数组开头取得数据的方法了。这个数组方法叫 shift() ，它会删除数组的第一项并返回它，然后数组长度减1。使用 shift() 和 push()

```
let colors = new Array(); // 创建
    一个数组
    let count = colors.unshift("red"
        ,
        "green"); // 从数
    组开头推入两项
```

```
    alert(count); // 2
    count = colors.unshift("black"); // 再推入一项
    alert(count); // 3
    let item = colors.pop(); // 取得最后一项
    alert(item); // green
    alert(colors.length); // 2
```

排序方法

数组有两个方法可以用来对元素重新排序： reverse() 和 sort() 。顾名思义， reverse() 方法就是将数组元素反向排列。
这里，数组 values 的初始状态为 [1,2,3,4,5] 。通过调用 reverse() 反向排序，得到了 [5,4,3,2,1] 。这个方法很直观，但 不够灵活，所以才有了 sort() 方法。
比较函数接收两个参数，如果第一个参数应该排在第二个参数前面， 就返回负值；如果两个参数相等，就返回0；如果第一个参数应该排在第 二个参数后面，就返回正值。
这个比较函数可以适用于大多数数据类型，可以把它当作参数传给 sort() 方法
在给 sort() 方法传入比较函数后，数组中的数值在排序后保持了正 确的顺序。当然，比较函数也可以产生降序效果，只要把返回值交换一下 即可
在这个修改版函数中，如果第一个值应该排在第二个值后面则返回 1，如果第一个值应该排在第二个值前面则返回-1。交换这两个返回值之 后，较大的值就会排在前头，数组就会按照降序排序。当然，如果只是想 反转数组的顺序， reverse() 更简单也更快
注意 reverse() 和 sort() 都返回调用它们的数组的引用。

操作方法

对于数组中的元素，我们有很多操作方法。比如， concat() 方法 可以在现有数组全部元素基础上创建一个新数组。它首先会创建一个当前 数组的副本，然后再把它的参数添加到副本末尾，最后返回这个新构建的 数组。如果传入一个或多个数组，则 concat() 会把这些数组的每一项都添加到结果数组。如果参数不是数组，则直接把它们添加到结果数组末 尾。
方法 slice() 用于创建一个包含原有数组中一个或多个元 素的新数组。 slice() 方法可以接收一个或两个参数：返回元素的开始 索引和结束索引。如果只有一个参数，则 slice() 会返回该索引到数组 末尾的所有元素。如果有两个参数，则 slice() 返回从开始索引到结束 索引对应的所有元素，其中不包含结束索引对应的元素。记住，这个操作 不影响原始数组。

```
let colors = ["red"
        ,
        "green"
        ,
        "blue"
        ,
        "yellow"
        ,
        "purple"];
    let colors2 = colors.slice(1);
    let colors3 = colors.slice(1, 4);
    alert(colors2); // green,blue,yellow,purple
    alert(colors3); // green,blue,yellow
```

注意：如果 slice() 的参数有负值，那么就以数值长度加上这个负值的结果确定位置。比如，在包含5个元素的数组上调用 slice(-2,-1) ，就相当于调用 slice(3,4) 。如果结束位置小于 开始位置，则返回空数组。

删除。需要给 splice() 传2个参数：要删除的第一个元素的位置和 要删除的元素数量。可以从数组中删除任意多个元素，比如 splice(0, 2) 会删除前两个元素。
插入。需要给 splice() 传3个参数：开始位置、0（要删除的元素 数量）和要插入的元素，可以在数组中指定的位置插入元素。第三个 参数之后还可以传第四个、第五个参数，乃至任意多个要插入的元 素。比如， splice(2, 0, "red" , "green") 会从数组位置2开 始插入字符串 "red" 和 "green" 。
替换。 splice() 在删除元素的同时可以在指定位置插入新元素， 同样要传入3个参数：开始位置、要删除元素的数量和要插入的任意 多个元素。要插入的元素数量不一定跟删除的元素数量一致。比如， splice(2, 1, "red" , "green") 会在位置2删除一个元素，然 后从该位置开始向数组中插入 "red" 和 "green" 。

splice() 方法始终返回这样一个数组，它包含从数组中被删除的 元素（如果没有删除元素，则返回空数组）。以下示例展示了上述3种使 用方式。

搜索和位置方法

ECMAScript提供两类搜索数组的方法：按严格相等搜索和按断言函数 搜索。

严格相等

ECMAScript提供了3个严格相等的搜索方法： indexOf() 、 lastIndexOf() 和 includes() 。其中，前两个方法在所有版本 中都可用，而第三个方法是ECMAScript 7新增的。

这些方法都接收两 个参数：要查找的元素和一个可选的起始搜索位置。

indexOf() 和 includes() 方法从数组前头（第一项）开始向后搜索，而 lastIndexOf() 从数组末尾（最后一项）开始向前搜索。

indexOf() 和 lastIndexOf() 都返回要查找的元素在数组中的 位置，如果没找到则返回-1。

includes() 返回布尔值，表示是否 至少找到一个与指定元素匹配的项。在比较第一个参数跟数组每一项 时，会使用全等（ === ）比较，也就是说两项必须严格相等。

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
alert(numbers.indexOf(4)); // 3
alert(numbers.lastIndexOf(4)); // 5
alert(numbers.includes(4)); // true
alert(numbers.indexOf(4, 4)); // 5
alert(numbers.lastIndexOf(4, 4)); // 3
alert(numbers.includes(4, 7)); // false
let person = { name: "Nicholas" };
let people = [{ name: "Nicholas" }];
let morePeople = [person];
alert(people.indexOf(person)); // -1
alert(morePeople.indexOf(person)); // 0
alert(people.includes(person)); // false
alert(morePeople.includes(person)); // true
```

断言函数

断言函数接收3个参数：元素、索引和数组本身。其中元素是数组中 当前搜索的元素，索引是当前元素的索引，而数组就是正在搜索的数 组。断言函数返回真值，表示是否匹配。 find() 和 findIndex() 方法使用了断言函数。这两个方法都从 数组的最小索引开始。

find() 返回第一个匹配的元素， findIndex() 返回第一个匹配元素的索引。这两个方法也都接收第 二个可选的参数，用于指定断言函数内部 this 的值。

迭代方法

ECMAScript为数组提供了两个归并方法 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。 reduce() 方法从数组第一项开始遍历到最后一 项。而 reduceRight() 从最后一项开始遍历至第一项。

可以使用 reduce() 函数执行累加数组中所有数值的操作

```
let values = [1, 2, 3, 4, 5];
let sum = values.reduce((prev, cur, index, array) =>
    prev + cur);
alert(sum); // 15
```

第一次执行归并函数时， prev 是1， cur 是2。第二次执行时， prev 是3（1 + 2）， cur 是3（数组第三项）。如此递进，直到把所有 项都遍历一次，最后返回归并结果。

reduceRight() 方法与之类似，只是方向相反。来看下面的例 子：

```
let values = [1, 2, 3, 4, 5];
    let sum = values.reduceRight(function (prev, cur,
        index, array) {
        return prev + cur;
    });
    alert(sum); // 15
```

在这里，第一次调用归并函数时 prev 是5，而 cur 是4。当然，最 终结果相同，因为归并操作都是简单的加法。 究竟是使用 reduce() 还是 reduceRight() ，只取决于遍历数组 元素的方向。除此之外，这两个方法没什么区别。

定型数组

定型数组行为 从很多方面看，定型数组与普通数组都很相似。定型数组支持如下操 作符、方法和属性： [] copyWithin()
entries()
every()
fill()
filter()
find()
findIndex()
forEach()
indexOf()
join()
keys()
lastIndexOf()
length
map()
reduce()
reduceRight()
reverse()
slice()
some()
sort()
toLocaleString()
toString()
values()

小结

JavaScript中的对象是引用值，可以通过几种内置引用类型创建特定类 型的对象。 引用类型与传统面向对象编程语言中的类相似，但实现不同。 Object 类型是一个基础类型，所有引用类型都从它继承了基本的行 为。 Array 类型表示一组有序的值，并提供了操作和转换值的能力。定型数组包含一套不同的引用类型，用于管理数值在内存中的类型。 Date 类型提供了关于日期和时间的信息，包括当前日期和时间以及 计算。 RegExp 类型是ECMAScript支持的正则表达式的接口，提供了大多 数基本正则表达式以及一些高级正则表达式的能力。 JavaScript比较独特的一点是，函数其实是 Function 类型的实例， 这意味着函数也是对象。由于函数是对象，因此也就具有能够增强自身行 为的方法。 因为原始值包装类型的存在，所以JavaScript中的原始值可以拥有类似 对象的行为。有3种原始值包装类型： Boolean 、 Number 和 String 。它们都具有如下特点。

第 8 章 对象、类与面向对象编程

8.1 理解对象

8.1.1 创建自定义对象

```
// 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。
let person = new Object();
person.name = "Kobe";
person.age = 40;
person.sayName = function() {
console.log(this.name);
};
```

```
// 对象字面量
let person = {
name: "Kobe",
age: 40,
sayName() {
console.log(this.name);
}
};
```

8.1.2 属性分两种

```
// 数据属性
[[Configurable]]: 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 。
[[Enumberable]]: 表示属性是否可以通过 for-in 循环返回。默认为 true 。
[[Writable]]: 表示属性的值是否可以被修改。默认为 true 。
[[Value]]: 包含属性实际的值。默认为 undefined 。
```

```
// 访问器属性
[[Configurable]]: 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 。
[[Enumberable]]: 表示属性是否可以通过 for-in 循环返回。默认为 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 。
```

8.1.3 Object.defineProperty()

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

// 访问器属性是不能直接定义的，必须使用 Object.defineProperty() 。
// year_ 中的下划线常用来表示该属性并不希望在对象方法的外部被访问。
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

// 在一个对象上同时定义多个属性
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;
```

```
}
},
},
});
```

8.1.4 Object.getOwnPropertyDescriptor()

```
// 读取属性的特性
// 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
let book = {};
Object.defineProperties(book, {
year_: {
value: 2017,
},
edition: {
value: 1,
},
year: {
get: function () {
return this.year_;
},
set: function (newValue) {
if (newValue > 2017) {
this.year_ = newValue;
this.edition += newValue - 2017;
},
},
});

let descriptor = Object.getOwnPropertyDescriptor(book, "year_");
console.log(descriptor.value); // 2017
console.log(descriptor.configurable); // false
console.log(typeof descriptor.get); // "undefined"
let descriptor = Object.getOwnPropertyDescriptor(book, "year");
console.log(descriptor.value); // undefined
console.log(descriptor.enumerable); // false
console.log(typeof descriptor.get); // "function"
```

8.1.5 Object.getOwnPropertyDescriptors()

```
// 这个方法实际上会在每个自有属性上调用 Object.defineProperties()并在一个新对象中返回它们。
let book = {};
Object.defineProperties(book, {
year_: {
value: 2017,
},
edition: {
value: 1,
},
year: {
get: function () {
return this.year_;
},
set: function (newValue) {
if (newValue > 2017) {
this.year_ = newValue;
this.edition += newValue - 2017;
}
},
},
});
console.log(Object.getOwnPropertyDescriptors(book));
/* {
edition: {
configurable: false,
enumerable: false,
value: 1,
writable: false
},
year: {
configurable: false,
enumerable: false,
get: f(),
set: f(newValue),
},
year_: {
configurable: false,
enumerable: false,
value: 2019,
writable: false
}
} */
```

8.1.6 Object.assign() => 合并对象

```
// 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。
// 实际上对每个源对象执行的是浅复制, 浅复制意味着只会复制对象的引用。
// 不能在两个对象间转移获取函数和设置函数。
// 如果多个源对象都有相同的属性，则使用最后一个复制的值。
// Object.assign()没办法回滚已经完成的修改，因此在抛出错误之前，目标对象上已经完成的修改会继续存在。
let dest, src, result;
// 简单复制
dest = {};
src = { id: "src" };
result = Object.assign(dest, src);
// Object.assign修改目标对象
// 也会返回修改后的目标对象
console.log(dest === result); // true
console.log(dest !== src); // true
console.log(result); // { id: src }
console.log(dest); // { id: src }

// 多个源对象
dest = {};
result = Object.assign(dest, { a: "foo" }, { b: "bar" });
console.log(result); // { a: foo, b: bar }

// 获取函数与设置函数
```

8.1.7 对象标识及相等判定

```
// 要确定NaN的相等性，必须使用极为讨厌的isNaN()
console.log(NaN === NaN); // false
console.log(isNaN(NaN)); // true
```

Object.is()

```
// 必须接收两个参数
// 正确的NaN相等判定
console.log(Object.is(NaN, NaN)); // true
```

8.1.8 增强对象语法

属性值简写

```
// 只要使用变量名（不用再写冒号）就会自动被解释为同名的属性键。如果没有找到同名变量，则会抛出 ReferenceError 。
let name = 'Matt';
let person = {
// name: name
name
};
console.log(person); // { name: 'Matt' }
```

可计算属性

```
// 可以在对象字面量中完成动态属性赋值。
// 中括号包围的对象属性键告诉运行时将其作为JavaScript表达式而不是字符串来求值。
const nameKey = "name";
const ageKey = "age";
const jobKey = "job";
let person = {
[nameKey]: "Matt",
[ageKey]: 27,
[jobKey]: "Software engineer",
};
console.log(person); // { name: 'Matt', age:27, job: 'Software engineer' }
```

简写方法名

```
let person = {
sayName(name) {
console.log(&amp;quot;My name is ${name}&amp;quot;);
},
};
person.sayName(&amp;quot;Matt&amp;quot;); // My name is Matt8
```

8.1.9 对象解构

```
// 使用与对象匹配的结构来实现对象属性赋值。
// 如果引用的属性不存在，则该变量的值就是 undefined
let person = {
name: 'Matt',
age: 27
};
let { name: personName, age: personAge } = person;
// 可以让变量直接使用属性的名称，也可以在解构赋值的同时定义默认值。
let { name, age, job = 'Software engineer' } = person;
console.log(personName); // Matt
console.log(personAge); // 27
console.log(name); // Matt
console.log(age); // 27
console.log(job); // Software engineer
```

嵌套解构

部分解构

参数上下文匹配

8.2 创建对象

8.2.1 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。
```

8.2.2 原型和 in 操作符

```
// 有两种方式使用 in 操作符：单独使用和在 for-in 循环中使用。
// 在单独使用时， in 操作符会在可以通过对象访问指定属性时返回 true ，无论该属性是在实例上还是在原型上。
// 在 for-in 循环中使用 in 操作符时，可以通过对象访问且可以被枚举的属性都会返回，包括实例属性和原型属性。
```

8.3 继承

8.3.1 原型链继承

```
/* 原型链的基本构想:
每个构造函数都有一个原型对象，原型有一个属性指回构造函数，而实例有一个内部指针指向原型。如果原型是另一个类型的实例呢？那就意味着这个原型本身有一个内部指针指向另一个原型，相应地另一个原型也有一个指针指向另一个构造函数。这样就在实例和原型之间构造了一条原型链。这就是原型链的基本构想。*/

function SuperType() {
this.property = true;
}
SuperType.prototype.getSuperValue = function () {
return this.property;
};
function SubType() {
this.subproperty = false;
}
// 继承SuperType
SubType.prototype = new SuperType();
SubType.prototype.getSubValue = function () {
return this.subproperty;
};
let instance = new SubType();
console.log(instance.getSuperValue()); // true
```

默认原型

```
// 所有引用类型都继承自 Object ,
// 任何函数的默认原型都是一个 Object 的实例，这意味着这个实例有一个内部指针指向 Object.prototype
```

原型与继承关系

```
// 原型与实例的关系可以通过两种方式来确定:
// 第一种方式是使用instanceof 操作符，如果一个实例的原型链中出现过相应的构造函数, 则 instanceof 返回 true 。
console.log(instance instanceof Object); // true

// 第二种方式是使用 isPrototypeOf() 方法。只要原型链中包含这个原型，这个方法就返回 true 。
console.log(Object.prototype.isPrototypeOf(instance)); // true
```

8.3.2 盗用构造函数

```
// 可以使用 apply() 和 call() 方法以新创建的对象为上下文执行构造函数。
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");
```

8.3.3 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。
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);
};
let instance1 = new SubType("Nicholas", 29);
instance1.colors.push("black");
console.log(instance1.colors); //
("red,blue,green,black");
instance1.sayName(); // "Nicholas";
instance1.sayAge(); // 29
let instance2 = new SubType("Greg", 27);
console.log(instance2.colors); //
("red,blue,green");
instance2.sayName(); // "Greg";
instance2.sayAge(); // 27
```

8.3.4 原型式继承

```
// 原型式继承非常适合不需要单独创建构造函数，但仍然需要在对象间共享信息的场合。但要记
住，属性中包含的引用值始终会在相关对象间共享，跟使用原型模式是一样的。
let person = {
name: "Nicholas",
friends: ["Shelby", "Court", "Van"],
};
let anotherPerson = object(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");
let yetAnotherPerson = object(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");
console.log(person.friends); //
("Shelby,Court,Van,Rob,Barbie");
```

8.3.5 寄生式继承

```
// 创建一个实现继承的函数，以某种方式增强对象，然后返回这个对象。
// 同样适合主要关注对象，而不在乎类型和构造函数的场景。
// 通过寄生式继承给对象添加函数会导致函数难以重用，与构造函数模式类似。

function createAnother(original) {
// 通过调用函数创建一个新对象;
let clone = object(original);
clone.sayHi = function () {
// 以某种方式增强这个对象;
console.log("hi");
};
return clone; // 返回这个对象
}
```

8.3.6 寄生式组合继承

```
// 寄生式组合继承通过盗用构造函数继承属性，但使用混合式原型链继承方法。
// 基本思路: 不通过调用父类构造函数给子类原型赋值，而是取得父类原型的一个副本。就是使用寄生式继承来继承父类原型，然后将返回的新对象赋值给子类原型。
// 最主要的效率问题就是父类构造函数始终会被调用两次：一次在是创建子类原型时调用，另一次是在子类构造函数中调用。
// 寄生式组合继承可以算是引用类型继承的最佳模式。
function inheritPrototype(subType, superType) {
let prototype = object(superType.prototype);
// 创建对象
prototype.constructor = subType;
// 增强对象
subType.prototype = prototype;
// 赋值对象
}
```

8.4 类

8.4.1 定义类

两种主要方式：

```
// 类声明
class Person {}
// 类表达式
const Animal = class {};
```

8.4.2 与函数表达式的异同

同：被求值前也不能引用。

异：

虽然函数声明可以提升，但类定义不能。

函数受函数作用域限制，而类受块作用域限制。

8.4.3 类的构成

```
// 类可以包含构造函数方法、实例方法、获取函数、设置函数和静态类方法，但这些都不是必需的。空的类定义照样有效。
// 默认情况下，类定义中的代码都在严格模式下执行。
// 首字母要大写
// 类表达式的名称是可选的。在把类表达式赋值给变量后，可以通过 name 属性取得类表达式的名称字符串。但不能在类表达式作用域外部访问这个标识符。

// 空类定义，有效
class Foo {}
// 有构造函数的类，有效
class Bar {
constructor() {}
}
// 有获取函数的类，有效
class Baz {
get myBaz() {}
}
// 有静态方法的类，有效
class Qux {
static myQux() {}
}
```

8.4.4 类构造函数

```
// constructor 关键字用于在类定义块内部创建类的构造函数。
// 构造函数的定义不是必需的，不定义构造函数相当于将构造函数定义为空函数。
// 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。
```

8.4.5 静态类方法

可以在类上定义静态方法。这些方法通常用于执行不特定于实例的操作，也不要求存在类的实例。与原型成员类似，每个类上只能有一个静态成员

静态类成员在类定义中使用 static 关键字作为前缀。在静态成员中， this 引用类自身。