第 1 章

ECMAScript 简介

本章内容主要介绍 ECMAScript 的历史、版本以及语法特性和多种集成开发环境，让读者对 ECMAScript 有一个初步的认识，以便为后续学习打好基础。

1.1 概　　述

成立于1961年之极具影响力的国际组织ECMA（European Computer Manufacturers Association，欧洲制造商协会），现今专门制定**信息**和**通信系统**的标准与技术报告，以促进和规范信息通信技术与消费电子产品。

ECMA 至今已经主动贡献了超过 400 个标准和 100 个技术报告，其中三分之二以上的部分，已在**国际上得到了广泛的使用**。其中，由 Ecma 制定的 ECMA-262（ISO/IEC 16262）的 ECMAScript，是具有商标的脚本编程语言（script programming language）的规范。

1.1.1 ECMAScript 各版本

从 1997 年 6 月的第 1 版，到 2018 年 6 月的第 9 版（ECMAScript 9），已问世的 ECMAScript 各版本，主要被用来标准化 JavaScript 脚本编程语言。目前业界的应用主要以 ECMAScript 5 / 6 为主、ECMAScript 7 / 8 / 9 为辅。从 ECMAScript 6 开始的各版本存在如表 1-1 所示的昵称。

表 1-1　ECMAScript 的各版本

正式名称	版本昵称	缩写昵称
ECMAScript 2015	ECMAScript 6	ES2015、ES6
ECMAScript 2016	ECMAScript 7	ES2016、ES7
ECMAScript 2017	ECMAScript 8	ES2017、ES8
ECMAScript 2018	ECMAScript 9	ES2018、ES9

1.1.2 关于 JavaScript

可简称为 JS 的 JavaScript，是一个动态高级解释脚本编程语言（dynamic high-level interpreting scripting programming language）。对于万维网而言，HTML、CSS 与 JavaScript 并列为网页制作的核心编程语言。

JavaScript 主要用来实现网页程序的用户交互机制（interactive mechanism），并通过 ECMAScript 规范，被创建和内置于网页浏览器（web browser）的 JavaScript 引擎中。

JavaScript 支持事件驱动（event-driven）、函数调用（function invocation / call）、面向对象编程（OOP，object-oriented programming）等特性，并具备处理文本、数组、日期、正则表达式（regular expression）和文档对象模型（DOM，document object model）的应用程序编程接口（API，application programming interface）。

JavaScript 引擎早期仅实现于网页浏览器当中，现今则被实现于其他类型的软件里，例如网页服务器、数据库服务器、文字处理器，以及用来编写移动或桌面应用程序的集成开发环境（IDE，integrated development environment）等软件里。

1.1.3 其他脚本语言

通过 ECMAScript 规范实现的编程语言，除了 JavaScript 之外，主要还有 Adobe 的 ActionScript 和 Microsoft 的 JScript，其语法非常相似于 JavaScript。

JScript 被用于 Internet Explorer 浏览器，ActionScript 主要被用于 Adobe Animate CC 集成开发环境中。

1.2 语法的实现

计算机编程语言的语法是一种规则，用来定义此编程语言表面形式的符号组合，以正确组成此语言的源代码（source code），进而堆砌成为片段或文件。

1.2.1 源代码

源代码（source code）通常以纯文本方式，存在于文档里，并作为输入数据，由汇编器（assembler）、编译器（compiler）或解释器（interpreter），转换成计算机可理解的二进制机器代码（machine code）。

通过经常阅读他人编写的源代码，计算机程序员可增进其编程技术的成熟度。因此，开发者互相分享源代码，可算是对彼此的一种贡献。

移植特定软件到其他计算机平台上，通常是异常困难的。然而，若有此软件的源代码，移植任务就会变得简单许多。

1.2.2　语句

在计算机编程当中，语句（statement）是命令式编程语言的语法单元，以表示欲指示计算机进行的一连串动作。

JavaScript 编程语言的语句大致分为简单语句（simple statement）和复合语句（compound statement）两种形式。

1. 简单语句

简单语句的末尾应该加上分号，例如：

（1）

```
let v01, v02 ;
```

变量 v01 与 v02 的声明。

（2）

```
var list01 = ['apple', 'banana', 'cherry'] ;
```

数组变量 list01 的声明与数据设置。

（3）

```
profile = {name: 'Alex', gender: 'male', age: '40'} ;
```

对象变量 profile 的数据设置。

（4）

```
document.writeln ('<h3>world peace</h3>') ;
```

写入带有文本“world peace”的 h3 元素实例到当前网页里。

（5）

```
confirm ('Are you sure to delete it?') ;
```

在网页上，显示带有“Are you sure to delete it?”信息并带有确认按钮和取消按钮的模式对话框。

（6）

```
username.style.color = 'RoyalBlue' ;
```

设置 id 为 username 元素实例的颜色样式为宝蓝色。

（7）

```
break ;
```

中断循环语句（loop statement），或确认切换语句（switch statement）的分支（branch）。

（8）

```
continue ;
```

在循环语句中，终止当前的循环，并立即进行下一次的循环。

（10）

```
debugger ;
```

设置调试**断点**（breakpoint），以暂停后续源代码的执行，并启动调试机制。

（11）

```
"use strict" ;
```

限制以严格模式执行源代码。

（12）

```
return result ;
```

终止函数的执行，并返回变量 result 的数据。

2. 复合语句

复合语句大都带有一对大括号，例如：

（1）

```
let result = 0 ;
alert ("You're welcome!") ;
{
  let num01 = 75, num02 = 64 ;
  result = num01 + num02 ;
}
```

在前述大括号中的源代码，连同大括号在内，可被称为复合语句。其中，变量 num01 与 num02 只有在大括号里才允许被访问。

（2）

```
with (send_button.style)
{
  color = 'Gold' ;
  backgroundColor = 'DodgerBlue' ;
  fontSize = '1.5em' ;
}
```

with 复合语句包含一对大括号，并且可以简化对象实例属性的访问语法。在此，【send_button.style.color = 'Gold' ;】的语句，在复合语句【with (send_button.style) 】的大括号里，可被简化为【color = 'Gold' ;】。

（3）

```
if (score >= 60)
{
  passer_count++ ;
  saying = 'You have passed!' ;
}
```

if 条件复合语句可包含一对大括号，当小括号里的条件【(score >= 60)】为真时，大括号里的各语句就会被执行。

（4）

```
for (let i = 0 ; i < 10 ; i++)
{
  count++ ;
  sum += count * i ** 2 ;
```

```
}
```

for 循环复合语句可包含一对大括号，并借助小括号【(let i = 0 ; i < 10 ; i++)】中的 3 个子语句，使得大括号中的各语句，可被迭代而执行 10 次。

（5）

```
switch (choice)
{
  case 1:
    grade = 'A+' ;
    break ;
  case 2:
    grade = 'A' ;
    break ;
  case 3:
    grade = 'B' ;
    break ;
  case 4:
    grade = 'C' ;
    break ;
  default:
    grade = '@_@' ;
}
```

switch 复合语句包含一对大括号，并通过小括号【(choice)】中变量 choice 的数值，来决定并执行大括号中的特定 case 或 default 分支的子语句。

1.2.3　表达式

编程语言中的表达式（expression）内含运算符（operator）以及代表操作数（operand）的常量（constant）、变量（variable）、函数返回值（function return value）与子表达式（subsidiary expression）。

一般情况下，表达式的结果数据通常是原始数据类型（primitive data type）之一，例如数字（number）、字符串（string）或布尔型（boolean）。

JavaScript 编程语言的表达式大致可分为以下几种。

1. 算术表达式（arithmetic expression）

例如：

（1）v01 = v02 + 5 ;

将变量 v02 的数值，加上 5 之后的结果值，赋给变量 v01。

（2）v03 = v02 ** 3 ;

将变量 v02 的数值，进行 3 次方运算的结果值，赋给变量 v03。

（3）v04 = 6 * (v03 + 10) ;

将 6 乘以【变量 v03 加上 10】之后的积，赋给变量 v04。

（4）v05 = v04 % 2 + v04 / 2 ;

将变量 v04 除以 2 的余数，加上变量 v04 除以 2 的结果值，赋给变量 v05。

（5）c01 = c01 + 1 ;

将变量 c01 的数值，加上 1 的结果值，赋给变量 c01 本身。

（6）c01 += 1 ;

将变量 c01 的数值**增加 1 / 递增**（increment）。

（7）c01++ ;

先返回 c01 的数值，再递增变量 c01。

（8）++c01 ;

先递增变量 c01 的数值，再返回 c01。

（9）c02 = c02 - 1 ;

将变量 c02 的数值，**减去 1 / 递减**（decrement）的结果值，赋给变量 c02 本身。

（10）c02 -= 1 ;

将变量 c02 的值**减去 1 / 递减**的结果值，赋给变量 c02 本身。

（11）c02-- ;

先返回 c02 的数值，再递减变量 c02。

（12）--c02 ;

先递减变量 c02 的数值，再返回 c02。

2. 字符串表达式（string expression）

例如：

（1）

```
let subject = 'Alex' ;
let object = 'Jasper' ;
let greeting = "'have a \"nice\" day'" ;
let message = '' ;

message = subject + ' said ' + greeting + ' to ' + object ;
message = `${subject} said ${greeting} to ${object}` ;
```

在一对单引号或双引号内的文本是代表一种常量（constant）的字符串字面量（string literal）。

在字符串字面量里的单引号或双引号应该冠上反斜杠（back slash）【\】，成为【\'】或【\"】。

多个字符串字面量可通过加法运算符【+】，结合成为新的字符串。

通过一对反引号（back quote）【`】，可创建模板字面量（template literal），并内含语法【${特定变量的名称}】，以动态解析特定变量的数据，成为新字符串的一部分。

在前述源代码里，最后两行具有相同的效果。

（2）

```
let sentence = 'Alice really \
lovingly loves \
lovely beloved of \
Jason very much.' ;
```

借助反斜杠【\】来分割较长的字符串字面量。

在各行中，字符串片段的最后一个字符必须就是反斜杠【\】，不可以再有包括空白字符在内的其他字符。

前述源代码被执行之后，变量 sentence 的数据会是字符串 "Alice really lovingly loves lovely beloved of Jason very much."。

3. 关系与逻辑表达式（relational and logical expression）

例如：

（1）

```
x > y
```

检验若变量 x 的数值大于 y 的数值，则返回布尔值 true。

（2）

```
x <= y && x <=z
```

检验若变量 x 的数值同时小于或等于变量 y 和 z 的数值，则返回布尔值 true。

（3）

```
a > 0 || b > 0 || c > 0
```

检验若变量 a 的数值大于 0、变量 b 的数值大于 0，或是变量 c 的数值大于 0，则返回布尔值 true。

（4）

```
! (v01 > 60 || v02 >= 90) && v03 >= 80
```

检验若【并非变量 v01 的数值大于 60，或是变量 v02 的数值大于或等于 90】，并且【变量 v03 的数值大于或等于 80】，则返回布尔值 true。

（5）

```
name == 'admin' && /^\w{6,}$/.test(password)
```

检验若变量 name 的数据等于字符串字面量 "admin"，并且变量 password 的数据是由【0～9】、【大小写 a～z】与【下画线_】所构成的**最少** 6 个字符的字符串，则返回布尔值 true。

4. 主要表达式（primary expression）

例如：

（1）

```
let circle_area = function (r)
{
  let result ;
  result = Math.PI * r ** 2
  console.log (`The circle area of radius ${r} = ${result}`) ;
  return result ;
} ;
```

在所谓的函数表达式（function expression）中，通过关键字 function，定义匿名函数（anonymous function），并以变量的名称 circle_area，作为匿名函数的别名（alias）。

（2）

```
let Cubic = class
```

```
{
  constructor (l, w, h)
  {
    this.length = l ;
    this.width = w ;
    this.height = h ;
  }

  volume (l = this.length, w = this.width, h = this.height)
  {
    return l * w * h ;
  }

  surface_area (l = this.length, w = this.width, h = this.height)
  {
    return 2 * (l * w + w * h + h * l) ;
  }
} ;
```

在所谓的类表达式（class expression）中，通过关键字 class，定义匿名类（anonymous class），并以变量的名称 Cubic 作为匿名类的别名。

（3）

```
let number_list = [15, 30, 75, 90, 180] ;
console.log(number_list[2]) ;
number_list[5] = 770 ;
```

这 3 个语句均可被视为主要表达式。在等号右侧，没有变量名称的一对中括号，是用来定义数组实例 [15, 30, 75, 90, 180]，并赋给等号左侧的数组变量 number_list。在等号左侧，变量名称衔接的一对中括号，则用来访问数组变量 number_list 中特定索引值（2 与 5）对应的元素值（75 与 770）。

（4）

```
var user_input = 'z1 = x1 ^ 2 + y1 * 3 + 6' ;
var pattern = /[a-zA-Z]\d/g ;
var matches = user_input.match (pattern) ;
console.log (matches) ;
```

这 4 个语句均可被视为主要表达式。在此，通过一对斜杠符号【/】，创建用来匹配特定模式【由字母开头，后接一个数字的文本】的**正则表达式字面量**（regular-expression literal）【/[a-zA-Z]\d/g】，并赋给等号左侧的变量 pattern。

5. 箭头函数表达式（arrow function expression）

箭头函数表达式存在如下几种形式。

（1）

```
(参数列) => { …} ;
```

通过带有参数列的一对小括号，衔接代表箭头符号的【=>】，与内含主体源代码的一对大括号，以定义没有名称、但可带有多个参数的箭头函数（arrow function）。

（2）

```
单一参数名称 => { …} ;
```

通过单一参数，衔接代表箭头符号的【=>】，与内含主体源代码的一对大括号，以定义没有名称、仅有一个参数的箭头函数。

（3）

```
单一参数名称 =>单一语句 ;
```

通过单一参数，衔接代表箭头符号的【=>】，和单一语句的主体源代码，以定义没有名称、仅有一个参数和语句的箭头函数。

（4）

```
() => { …} ;
```

通过不带任何参数的一对小括号，衔接代表箭头符号的【=>】，与内含主体源代码的一对大括号，以定义没有名称和参数的箭头函数。

（5）

```
() => 单一语句 ;
```

通过不带任何参数的一对小括号，衔接代表箭头符号的【=>】，和单一语句的主体源代码，以定义没有名称与参数、仅有单一语句的箭头函数。

6. 左侧表达式（left-hand expression）

例如：

（1）

```
profile.name = 'Jasper' ;
```

通过点号【.】，将字符串字面量 'Jasper'，赋给变量 profile 的实例属性 name。

（2）

```
profile['age'] = 28 ;
```

借助一对中括号，将整数值 28 赋给变量 profile 的实例属性 age。

（3）

```
super(10, 15) ;
```

通过关键字 super 与一对小括号，调用父类（parent class）的构造函数（constructor），并传入参数值 10 和 15。

（4）

```
super.cylinder_volume(10, 15) ;
```

借助关键字 super、点号与一对小括号，调用在父类中作为成员函数（member function）的函数 cylinder_volume()，并传入参数值 10 和 15。

（5）

```
x = x + 3 ;
```

在等号的左侧，设置变量 x，并将变量 x 的数值加上 3 的结果值，赋给变量 x 本身。

（6）

```
[a, b] = [7, 13] ;
```

在等号的左侧，编写内含变量 a 与 b 的一对中括号，并将右侧另一对中括号里的整数值 7 与 13，分别赋给变量 a 与 b。

（7）

```
[c, d] = [a + 3, b + 7] ;
```

在等号的左侧，编写内含变量 c 和 d 的一对中括号，并将右侧另一对中括号里的 a + 3 与 b + 7 的结果值，分别赋给变量 a 与 b。

1.2.4 子程序

在计算机编程中，子程序（subroutine）也可被称为函数（function）或方法（method），即是可重复被调用（call / invocation），以重新执行特定任务的一连串程序指令单元，进而节省开发与维护的成本，并提高质量与可靠性。

JavaScript 编程语言中的子程序，主要被称为函数，可通过以下语法来加以定义。

（1）

```
function sphere(r)
{
  values = {} ;
  values.volume = 4 / 3 * Math.PI * Math.pow(r, 3) ;
  values.surface_area = 4 * Math.PI * r * r ;

  return values ;
}
```

上述源代码是定义函数的标准语法。通过调用【sphere(10)】，可计算并返回【内含半径为 10 的球体积和球体表面积】相关数据的对象实例。

（2）

```
display = function ()
{
  console.log('Variable-type displaying.') ;
} ;
```

上述源代码是所谓的函数表达式，使得变量 display 具备函数的特征。通过调用【display()】，可在网页浏览器的调试工具【Console】面板中，显示【Variable-type displaying】的信息。

（3）

```
sphere_volume = r => { return 4 / 3 * Math.PI * Math.pow(r, 3) } ;
```

上述源代码中的简洁语法，等同于定义了函数 sphere_volume()。通过调用【sphere_volume(30)】，可计算并返回半径为 30 的球体积的结果值。

1.2.5　注释

在源代码里，注释（comment）是用来辅助程序员，加以理解各源代码片段的意义和用途，却不会被计算机加以执行的文本。

JavaScript 编程语言的注释方式有如下两种。

（1）在注释文本的行首，加上紧连的两个斜杠符号【//】。例如：

```
// calculate the summary values for bonus table.
```

（2）在注释文本的开头，加上紧连的斜杠符号与星号【/*】，并在注释文本的末尾，加上紧连的星号与斜杠符号【*/】。例如：

```
/*
  ca(r) calculates circle area of radius r.
  ssa(r) calculates sphere surface area of radius r.
  cv(r, h) calculates cylinder volume of radius r and height h.
*/
```

1.2.6　关键字

在计算机编程语言中，被称为保留字（reserved word）的关键字（keyword），不能或不应该作为常量、变量、属性、函数 / 方法的标识符（identifier）/ 名称（name）的词汇。JavaScript 编程语言的关键字如表 1-2 所示。

表 1-2　JavaScript 的关键字

关键字	关键字	关键字	关键字	关键字
abstract	arguments	await	boolean	break
byte	case	catch	char	class
const	continue	debugger	default	delete
do	double	else	enum	eval
export	extends	false	final	finally
float	for	function	goto	if
implements	import	in	Infinity	instanceof
int	interface	let	long	NaN
native	new	null	package	private
protected	public	return	short	static
super	switch	synchronized	this	throw
throws	transient	true	try	typeof
undefined	var	void	volatile	while
with	yield			

1.3 开发环境

目前，知名的 JavaScript 集成开发环境（IDE，integrated development environment）有如下几种。

1. 存在免费版本的

- Visual Studio Community 或 Visual Studio Code
  https://www.visualstudio.com/zh-hans/downloads
- Eclipse
  http://www.eclipse.org/downloads/eclipse-packages
- IntelliJ IDEA
  https://www.jetbrains.com/idea
- NetBeans IDE
  https://netbeans.org
- Sublime Text
  https://www.sublimetext.com
- Atom IDE
  https://ide.atom.io
- Brackets
  http://brackets.io
- Cloud9（在线系统）
  https://aws.amazon.com/cn/cloud9
- Codeanywhere（在线系统）
  https://codeanywhere.com

2. 仅有付费版本的

- WebStorm
  https://www.jetbrains.com/webstorm
- Komodo IDE
  https://www.activestate.com/komodo-ide

JavaScript 编程语言主要用于开发网页相关应用程序，所以，网页浏览器成为执行与调试 JavaScript 源代码的主要软件。

因此，上述各个集成开发环境所产生的内含 JavaScript 源代码的网页应用，最终仍然在网页浏览器或其他内置 JavaScript 引擎的软件上，被进行必要的调试与执行。

1.3.1　浏览器

网页浏览器（web browser）是用来检索与呈现网络上信息资源（文本信息、图像、音频、视频、动画）的软件。现今较流行的网页浏览器为 Google Chrome、Mozilla Firefox、Opera、Apple Safari、与 Microsoft Edge。

上述的网页浏览器皆内置 JavaScript 调试工具。以 Google Chrome 浏览器为例，启动 Chrome 并按下 Ctrl + Shift + I / J 快捷键之后，即可看到如图 1-1 所示的开发者工具调试窗格。

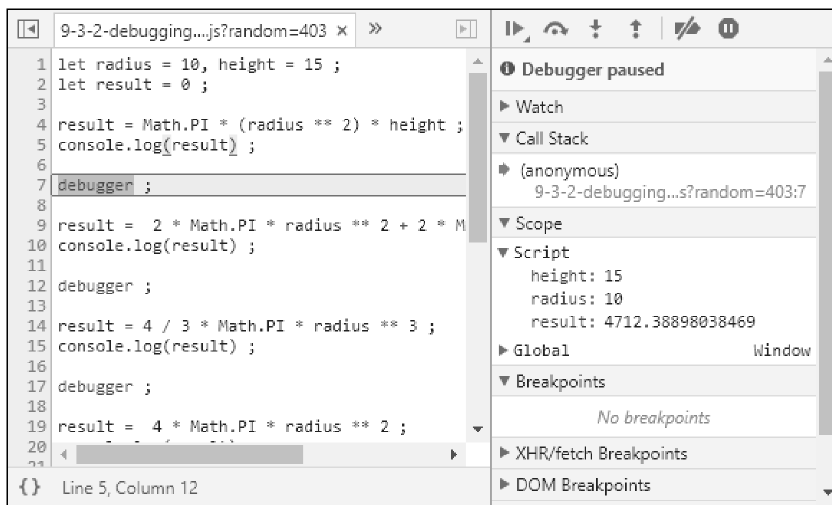


图 1-1

为了能顺利显示其 JavaScript 源代码于调试窗口中，可执行如下操作步骤。

步骤01 在 Chrome 浏览器，打开配书源代码示例文档中的网页文档【js_tester.html】。

选择文件 未选择任何文件

步骤02 按 Ctrl + Shift + I 快捷键，启动【开发者工具】，并可看到默认显示的 Console（控制台）窗口。

步骤03 请先按 Ctrl+ R 快捷键，以刷新页面，再单击网页中的【选择文件】按钮，并选择 JavaScript 源代码文档【js / 19-3-2-debugging- mechanism.js】之后，可在按钮右侧，明确看到被选定的文档名称和【Paused in debugger】信息。

选择文件 9-3-2-debugging-mechanism.js Paused in debugger

步骤04 观察调试窗口的变化，从【Console】面板，自动切换至【Source】面板中。

被读取的【19-3-2-debugging-mechanism.js】内含的源代码片段【debugger ;】用来设置调试断点（debugging breakpoint），使得浏览器启动调试机制，进而自动切换至【Source】面板中，并在第 1 个断点上，暂停 JavaScript 源代码的执行，如图 1-2 所示 。

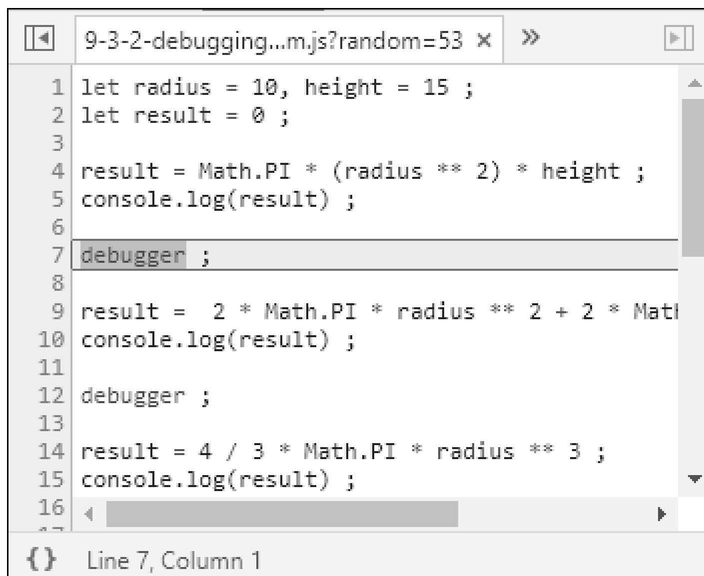


图 1-2

关于 JavaScript 源代码的相关调试，笔者将深入说明于本书 19.3 节。

关于操作扩展名亦为【.js】的【其他】示例文档，请完成至前述第 3 步即可。至于操作**扩展名**为【.html】的示例文档，则直接通过浏览器，**加载**并**浏览**其网页内容。

接着，观察特定扩展名为【.html】的示例文档，出现在浏览器中的网页内容，以及在**【Console】面板**里的对应信息，可进一步理解各个 HTML、CSS 和 JavaScript 源代码片段的工作原理。

1.3.2 Node.js

如同【ASP.NET】并非代表扩展名（file extension）为【.NET】的文档名称一样，Node.js 实际上亦不是代表 JavaScript 源代码的文档名称，而是开源且跨平台的**运行期环境**（run-time environment），并用于执行服务器端的 JavaScript 源代码，以生成动态的网页内容。

因此，对于主要借助 JavaScript 编程语言，整合前端（front-end）与后端（back-end）Web 应用程序开发的任务而言，Node.js 如今已然成为基本要素之一。此外，Node.js 具备事件驱动（event-driven）的运行架构，可实现数据异步的传递与接收，进而优化 Web 应用程序的处理能力和扩充灵活性。

1.3.3 其他 JavaScript Shell

通过特定 JavaScript Shell 的协助，可在不刷新特定网页内容的情况下，辅助开发与调试 JavaScript 源代码。较有名的 JavaScript Shell 有如下几种：

- Node.js
  https://nodejs.org
- JSDB
  http://www.jsdb.org
- JavaLikeScript
  http://javalikescript.free.fr
- GLUEscript
  http://gluescript.sourceforge.net
- jspl
  http://jspl.msg.mx
- ShellJS
  http://documentup.com/shelljs/shelljs

1.4 练　习　题

1. 截至 2018 年，ECMAScript 总共有几个版本？
2. 可被并列为网页制作的核心编程语言，除了 JavaScript 之外，还包括什么语言？
3. 网页浏览器相对应的英文短语是什么？
4. 除了 JavaScript 之外，通过 ECMAScript 规范实现的编程语言主要还有哪两个？
5. 依序翻译**在编程语言中**的专有名词：machine code、source code、programming language、syntax、subroutine、variable、constant 和 invocation。
6. 依序翻译**在编程语言中**的专有名词：assembler、compiler 和 interpreter。
7. 在如下的源代码片段里，有哪些**简单语句**？

```
with (send_button.style)
{
  color = 'Gold' ;
  backgroundColor = 'DodgerBlue' ;
  fontSize = '1.5em' ;
}
```

8. 在如下的源代码片段里，有哪些**简单语句**？

```
for (let i = 0 ; i < 10 ; i++)
{
  count++ ;
  sum += count * i ** 2 ;
}
```

9. 在如下的源代码片段里，哪些应该是**变量的名称**？

```
message = subject + ' said ' + greeting + ' to ' + object ;
```

10. 在源代码中的表达式里，哪些可以作为**操作数**？

11. 在如下源代码片段里，存在哪些不同的运算符？

```
! (v01 > 60 || v02 >= 90) && v03 >= 80
```

12. 在如下源代码片段里，存在哪些不同的常量？

```
let number_list = [15, 30, 75, 90, 180] ;
console.log (number_list[2]) ;
number_list[5] = 770 ;
```

13. 在如下源代码片段里，存在哪些不同的常量？

```
let user_input = 'z1 = x1 ^ 2 + y1 * 3 + 6' ;
let pattern = /[a-zA-Z]\d/g ;
let matches = user_input.match(pattern) ;
console.log(matches) ;
```

14. 在 JavaScript 语言里，如何调用如下被定义的函数，进而将半径为 50 的**球体积**和**球体表面积**，赋给变量 result？

```
function sphere(r)
{
  values = {} ;
  values.volume = 4 / 3 * Math.PI * Math.pow(r, 3) ;
  values.surface_area = 4 * Math.PI * r * r ;
  return values ;
}
```

15. 在 JavaScript 语言里，下列哪些可以作为**变量**的**名称**？

```
catchup、extends、finally、super、class、import、export 和 throw
```

16. 特定网页被浏览于 Google Chrome 浏览器窗口内时，有什么方式可以快速显示出浏览器**开发者工具**的 **Console 面板**？

第 2 章

表达式与运算符

本章内容主要介绍构成表达式的要素，包括常量、变量、子表达式、函数的返回值等形式的操作数，以及多种不同的运算符。

2.1 操 作 数

编程语言中的表达式主要由操作数（operand）和运算符（operator）构成。其中，操作数可以是常量（constant）、变量（variable）、子表达式（subsidiary expression）或是特定函数（function）的返回值（return value）。

2.1.1 常量（ES6）

在各种编程语言中，常量（constant）是指固定而明确的数据。例如：

（1）整数（integer）常量：25、770、-110。

（2）浮点数（floating-point number）常量：25.625、-23.71、Math.PI、Math.LN2、Math.SQRT2。

（3）字符串（string）的字面量（literal）：'Alex'、"Happily Ever After"、 '=.=+'。

（4）正则表达式（regular expression）的字面量：/\w+\s*.(\w\d+){1,3}/g、/^\w{2}\d{5}\s*$/。

（5）布尔（boolean）常量：true、false。

（6）原始（primitive）常量：NaN、null、undefined。

（7）编程人员在源代码中，自行声明（declare / declaration）的常量，可简称为自声明常量。例如在源代码片段【const num01 = 157, num02 = 268;】中，num01 和 num02 即是自声明常量。

关于自声明常量，可参考如下示例。

【2-1-1-constants.js】

```
const E = 299792458 ;

console.log('Speed of light is greater than 2.9E8?', E > 2.9e8) ;
```

【相关说明】

```
const E = 299792458 ;
```

声明数值为 299792458 的常量 E。

```
console.log('Speed of light is greater than 2.9E8?', E > 2.9e8) ;
```

在浏览器调试工具的【Console】面板里显示出【Speed of light is greater than 2.9E8? true】的信息。其中，2.9e8 代表 2.9×10^8，也就是 290000000。所以，【E > 2.9e8】等价于【E > 290000000】，会返回布尔值 true，这是因为自声明常量 E 的数值为 299792458，大于 290000000。

在上述简短的源代码中，常量 E 刻意被声明成整数常量 299792458，因此常量 E 在源代码的**运行期间**，不可被变更为其他数据。

2.1.2 变量（ES6）

在各种编程语言中，变量（variable）的标识符（identifier）/名称（name）用来识别特定存储位置中的可变数据。所谓的可变，是指在运行期间，其数据是可被变更的。

在 JavaScript 语言中，变量的作用范围（scope）大致可如下区分：

- 全局（global）范围
- 局部（local）范围
- 块（block）范围

下面通过示例，介绍如何运用全局范围和局部范围的变量。

【2-1-2-e1-global-and-local-scope-variables.js】

```
let start = 123 ;
const STEP = 3 ;
var result = 0 ;

function some_steps(step_count = 1)
{
  var output ; // or let output ;

  output = start + STEP * step_count ;

  console.log(`in function: start = ${start}, STEP = ${STEP}, result = ${result}`) ;
  console.log(`in function: output = ${output}\n\n`) ;

  return output ;
}

result = some_steps(5) ;
```

```
console.log(`outer: start = ${start}, step = ${STEP}, result = ${result}`) ;
console.log(`outer: output = ${output}`) ;
```

【相关说明】

```
let start = 123 ;
const STEP = 3 ;
var result = 0 ;
```

声明全局范围的变量 start 与 result，以及全局范围的自声明常量 STEP。既然是全局范围的变量和常量，即可让后续的任意表达式，加以访问。STEP 是一个常量，所以在后续的源代码里，无法被赋予新数据。

```
function some_steps(step_count = 1)
{
  var output ; // or let output ;

  output = start + STEP * step_count ;

  console.log(`in function: start = ${start}, step= ${STEP}, result = ${result}`) ;
  console.log(`in function: output = ${output}\n\n`) ;

  return output ;
}
```

定义函数 some_steps()，并在内部声明局部范围的变量 output。既然是局部范围的，变量 output 只能在函数 some_steps()内部被访问。变量 output 接着被赋予【带有全局范围变量 start 和常量 STEP】的表达式的结果值。这个函数内部的多条语句，访问了全局范围的变量 start、常量 STEP，以及局部范围的变量 output。

```
result = some_steps(5) ;
console.log(`outer: start = ${start}, step= ${STEP}, result = ${result}`) ;
```

访问全局范围的变量 start、result 与常量 STEP。

```
console.log(`outer: output = ${output}`) ;
```

因为访问了函数 some_steps()内的局部范围的变量 output，所以会显示出错误信息【Uncaught ReferenceError: output is not defined】。

关于块范围的理解，可参考如下示例。

【2-1-2-e2-block-scope-variables.js】

```
// different heights measured in meters.
{
  // height of the building.
  let height = 100 ;

  {
    // height of one floor.
    let height = 4 ;

    {
      // height of a room.
      let height = 3 ;
```

```
        {
          // height of a desk.
          let height = 1 ;

          console.log('height of a desk =', height) ;
        }

        console.log('height of a room =', height) ;
      }

      console.log('height of one floor =', height) ;
    }

    console.log('height of the building =', height) ;
  }

  console.log('') ;

  for (var m = 1; m < 10; m++)
  {
    for(var n = 1; n < 10; n++)
    {
      console.log(`kernel count = (${m},${n})`) ;
    }

    console.log(`inner count = (${m},${n})\n\n`) ;
  }

  console.log(`outer count = (${m},${n})`) ;
  console.log('') ;

  for (let i = 1; i < 10; i++)
  {
    for(let j = 1; j < 10; j++)
    {
      console.log(`kernel count = (${i},${j})`) ;
    }

    console.log(`inner count = (${i},${j})\n\n`) ;
  }

  // console.log(`outer count = (${i},${j})`) ;
```

【相关说明】

```
  // different heights measured in meters.
  {
    // height of the building.

    let height = 100 ;
```

此为第 1 层大括号内的块范围变量 height，所以第 1 层大括号之外的语句是访问不到的。声明块范围的变量，必须通过关键字 let 才行；由关键字 var 声明的变量，只会成为全局范围或局部范围的变量。在此，可将此层块范围的变量 height 视为**建筑物**的高度。

```
    {

      // height of one floor.
  let height = 4 ;
```

此为第 2 层大括号内的块范围变量 height，因此第 2 层大括号之外的语句是访问不到的。在此，可将此层块范围的变量 height，视为建筑物特定**楼层**的高度。

```
    {

      // height of a room.
      let height = 3 ;
```

此为第 3 层大括号内的块范围变量 height，因此第 3 层大括号之外的语句是访问不到的。在此，可将此层块范围的变量 height，视为特定楼层中特定**房间**的高度。

```
    {

      // height of a desk.
      let height = 1 ;
```

此为第 4 层大括号内的块范围变量 height，因此第 4 层大括号之外的语句是访问不到的。在此，可将此层块范围的变量 height，视为特定房间中特定**桌子**的高度。

```
        console.log('height of a desk =', height) ;
      }
      console.log('height of a room =', height) ;
    }
    console.log('height of one floor =', height) ;
  }
  console.log('height of the building =', height) ;
}
```

通过此源代码片段，可供读者调试出**每层**块范围变量 height 的对应值。

```
console.log('') ;

for (var m = 1; m < 10; m++)
{
  for(var n = 1; n < 10; n++)
  {
    console.log(`kernel count = (${m},${n})`) ;
  }
  console.log(`inner count = (${m},${n})\n\n`) ;
}
console.log(`outer count = (${m},${n})`) ;
```

测试此源代码片段，可看出通过关键字 var 声明的变量 m 与 n，在带有大括号的 for 循环语句结束之后，依然可以被访问到。这也就意味着，关键字 var 声明的，无法成为块范围的变量。

```
console.log('') ;

for (let i = 1; i < 10; i++)
{
  for(let j = 1; j < 10; j++)
  {
```

```
      console.log(`kernel count = (${i},${j})`) ;
    }
    console.log(`inner count = (${i},${j})\n\n`) ;
  }
  // console.log(`outer count = (${i},${j})`) ;
```

测试此源代码片段，可看出通过关键字 let 声明的变量 m 与 n，只能被访问于 for 循环语句大括号内的块范围里。这也就意味着，关键字 let 声明的，**可以**成为**块范围**的变量。复原最后一行成为注释的语句，并进行测试之后，可在网页浏览器的调试工具【Console】面板中，看到因为访问不到块范围变量 i 与 j 的错误信息。

关于特殊变量的理解，可参考如下示例。

【2-1-2-e3-special-variables.js】

```
document.body.style.backgroundColor = 'RoyalBlue' ;

console.log(document.body.style.backgroundColor) ;

document.body.style.fontSize = '3em' ;

console.log(document.body.style.fontSize) ;

file_selector.style.color = 'Gold' ;

console.log(file_selector.style.color) ;

file_selector.style.zoom = 2 ;
```

【相关说明】

```
file_selector.style.zoom = 2 ;
```

此语句中的子属性 zoom，因为可被设置新数据，所以算是**变量**的一种；然而，此语句中的子属性 style，则无法被设置新数据，也因此算是**常量**的一种。

通过 JavaScript 语法，可访问文档对象模型（DOM, document object model）中的特定元素实例（element instance）。元素实例可内含**常量**属性，只可被读取其数据，而不允许被写入新数据。下面举出 4 个元素实例中的常量属性：

- file_selector.style
- document.body.style
- document.body
- window.document（可被简写为 document）

以下可变更数据的是字符串（string）类型的属性，亦是网页浏览器内置的，所以可认为是网页程序在运行时的**特殊变量**：

- document.body.style.backgroundColor
- document.body.style.fontSize
- file_selector.style.color
- file_selector.style.zoom

2.1.3　子表达式

一般而言，特定表达式可再分割成为子表达式（subsidiary expression）。关于子表达式的运用，请参考如下示例。

【2-1-3-subsidiary-expressions.js】

```
var r = 10 ;
var volume = 4 / 3 * Math.PI * Math.pow(r, 3) ;

circumference = r => 2 * Math.PI * r ;

circle_area = r => Math.PI * Math.pow(r, 2) ;

sphere_volume = r => 4 / 3 * Math.PI * Math.pow(r, 3) ;

cylinder_volume = (r, h) => circle_area(r) * h ;

cylinder_surface_area = (r, h) => 2 * circle_area(r) + circumference(r) * h ;

rounded_circle_area = circle_area(10).toFixed(3) ;
rounded_cylinder_volume = cylinder_volume(10, 20).toFixed(3) ;
rounded_cylinder_surface_area = cylinder_surface_area(10, 20).toFixed(3) ;

console.log(rounded_circle_area) ;
console.log(rounded_cylinder_volume) ;
console.log(rounded_cylinder_surface_area) ;
```

【相关说明】

```
var r = 10 ;
```

此语句内含左侧表达式。

```
var volume = 4 / 3 * Math.PI * Math.pow(r, 3) ;
```

此主要表达式内含如下子表达式：

- 左侧表达式：【var volume =】和【Math.】
- 算术表达式：【4 / 3 * Math.PI * Math.pow(r, 3)】
- 主要表达式：【Math.PI】和【Math.pow(r, 3)】

```
circumference = r => 2 * Math.PI * r ;
```

此主要表达式内含如下子表达式：

- 左侧表达式：【circumference =】和【Math.】
- 箭头函数表达式：【r => 2 * Math.PI * r】
- 算术表达式：【2 * Math.PI * r】
- 主要表达式：【Math.PI】

```
circle_area = r => Math.PI * Math.pow(r, 2) ;
```

此表达式内含如下子表达式：

- 左侧表达式：【circle_area =】和【Math.】
- 箭头函数表达式：【r => Math.PI * Math.pow(r, 2)】
- 算术表达式：【Math.PI * Math.pow(r, 2)】
- 主要表达式：【Math.PI】和【Math.pow(r, 2)】

```
sphere_volume = r => 4 / 3 * Math.PI * Math.pow(r, 3) ;
```

此表达式内含如下子表达式：

- 左侧表达式：【sphere_volume =】和【Math.】
- 箭头函数表达式：【r => 4 / 3 * Math.PI * Math.pow(r, 3)】
- 算术表达式：【4 / 3 * Math.PI * Math.pow(r, 3)】
- 主要表达式：【Math.PI】和【Math.pow(r, 3)】

```
cylinder_volume = (r, h) => circle_area(r) * h ;
```

此表达式内含如下子表达式：

- 左侧表达式：【cylinder_volume =】
- 箭头函数表达式：【(r, h) => circle_area(r) * h】
- 算术表达式：【circle_area(r) * h】
- 主要表达式：【circle_area(r)】

```
cylinder_surface_area = (r, h) => 2 * circle_area(r) + circumference(r) * h ;
```

此表达式内含如下子表达式：

- 左侧表达式：【cylinder_surface_area =】
- 箭头函数表达式：【(r, h) => 2 * circle_area(r) + circumference(r) * h】
- 算术表达式：【2 * circle_area(r) + circumference(r) * h】
- 主要表达式：【circle_area(r)】和【circumference(r)】

```
rounded_circle_area = circle_area(10).toFixed(3) ;
```

此表达式内含如下子表达式：

- 左侧表达式：【rounded_circle_area =】和【circle_area(10).】
- 主要表达式：【circle_area(10).toFixed(3)】、【circle_area(10)】和【toFixed(3)】

```
rounded_cylinder_volume = cylinder_volume(10, 20).toFixed(3) ;
```

此表达式内含如下子表达式：

- 左侧表达式：【rounded_cylinder_volume =】和【cylinder_volume(10, 20).】
- 主要表达式：【cylinder_volume(10, 20).toFixed(3)】、【cylinder_volume(10, 20)】和【toFixed(3)】

```
rounded_cylinder_surface_area = cylinder_surface_area(10, 20).toFixed(3) ;
```

此表达式内含如下子表达式：

- 左侧表达式:【rounded_cylinder_surface_area =】和【cylinder_surface_area(10, 20).】
- 主要表达式:【cylinder_surface_area(10, 20).toFixed(3)】、【cylinder_surface_area(10, 20)】和【toFixed(3)】

```
console.log(rounded_circle_area) ;
console.log(rounded_cylinder_volume) ;
console.log(rounded_cylinder_surface_area) ;
```

如上 3 个语句排除分号【;】的其余部分，皆为主要表达式。

2.1.4　函数的返回值

在多种编程语言中，关键字 return 开头的语句，除了会终止当前函数内源代码的执行，并返回调用函数的子表达式之外，亦会使得前述子表达式，被取代成为返回值（return value），成为原始表达式的操作数。关于返回值的理解，可参考如下示例。

【2-1-4-function-return-values.js】

```
// cv stands for "cubic volume".
function cv01(l, w, h)
{
  let cubic_volume = l * w * h ;

  console.log(`cubic volume of (${l}, ${w}, ${h}) = ${cubic_volume}`) ;

  // return ;
}

function cv02(l, w, h)
{
  let cubic_volume = l * w * h ;

  return cubic_volume ;
}

let result01 = cv01(3, 5, 10) ;

let result02 = cv02(3, 5, 10) ;

let result03 = cv02(1, 3, 5) + cv02(2, 4, 6) ;

console.log(`result01 = ${result01}`) ;
console.log(`result02 = ${result02}`) ;
console.log(`result03 = ${result03}`) ;
```

【相关说明】

```
// cv stands for "cubic volume".
function cv01(l, w, h)
{
  let cubic_volume = l * w * h ;
  console.log(`cubic volume of (${l}, ${w}, ${h}) = ${cubic_volume}`) ;
```

```
  // return ;
}
```

- 在上述函数内的末尾处，并未放置 return 语句，或是 return 语句仅仅衔接分号【;】。所以，此段源代码定义了未带返回值的函数 cv01()。
- 函数 cv01()可在网页浏览器的调试工具【Console】面板里，显示出长、宽、高可能不同的立方体积。

```
function cv02(l, w, h)
{
  let cubic_volume = l * w * h ;
  return cubic_volume ;
}
```

- 此源代码片段定义了**具有返回值**的函数 cv02()，因为在上述函数内的末尾处，放置了语句【return cubic_volume ;】。
- 函数 cv02()亦可计算出长、宽、高可能不同的立方体积。

```
let result01 = cv01(3, 5, 10) ;
```

- 此语句调用了函数 cv01(3, 5, 10)，显示出长、宽、高各为 3、5、10 的立方体积。
- 因为函数 cv01()并无返回值，所以变量 result01 会被赋予原始常量 undefined。

```
let result02 = cv02(3, 5, 10) ;
```

此语句调用了函数 cv02(3, 5, 10)，并将长、宽、高各为 3、5、10 的立方体积结果值，返回到此语句，成为新的操作数 150，使得此语句等价于【let result02 = 150 ;】，进而让变量 result02 的数值变成 150。

```
let result03 = cv02(1, 3, 5) + cv02(2, 4, 6) ;
```

此语句调用了函数 cv02(1, 3, 5)和 cv02(2, 4, 6)，并将长、宽、高各为 1、3、5 与 2、4、6 的个别立方体积结果值，返回到此语句，成为新的操作数 15 与 48，使得此语句等价于【let result02 = 15 + 48 ;】，进而让变量 result02 的数值变成 63。

```
console.log(`result01 = ${result01}`) ;
```

此语句显示出变量 result01 的数据为 undefined。

```
console.log(`result02 = ${result02}`) ;
console.log(`result03 = ${result03}`) ;
```

这两个语句分别显示出变量 result02 的数值 150，以及变量 result03 的数值 63。

2.2 运 算 符

各种计算机编程语言均支持一系列的运算符，并和参与其中的操作数，构成特定形态的表达式，以满足计算机各种复杂的演算、分析与归纳。

2.2.1　算术运算符（ES7）

JavaScript 语言的算术运算符如表 2-1 所示。

表 2-1　JavaScript 的运算符

运算符分类	运算符
加法运算符（addition operator）或一元正号运算符（unary plus operator）	+
减法运算符（subtraction operator）或一元负号运算符（unary negation operator）	-
乘法运算符（multiplication operator）	*
除法运算符（division operator）	/
求余运算符（remainder operator）	%
求幂运算符（exponentiation operator）	**
递增运算符（increment operator）	++
递减运算符（decrement operator）	--

下面通过示例介绍各种算术运算符的运用。

【2-2-1-arithmetic-operators.js】

```
let num01 = 125, num02 = 10, num03 = 5 ;
let result = 0 ;

result = num01 % num02 ;
console.log(result) ;

result = num01 / num02 ;
console.log(result) ;

result = num01 * num02 ;
console.log(result) ;

result = num01 + num02 - num03 ;
console.log(result) ;

result = ++num01 ;
console.log(result, num01) ;

result = num01++ ;
console.log(result, num01) ;

++num01 ;
num01++ ;
console.log(num01) ;

num01 += 1 ;
console.log(num01) ;

num01 = num01 + 1 ;
console.log(num01) ;

result = --num02 ;
```

```
    console.log(result, num02) ;

    result = num02-- ;
    console.log(result, num02) ;

    --num02 ;
    num02-- ;
    console.log(num02) ;

    num02 -= 1 ;
    num02 = num02 - 1 ;
    console.log(num02) ;

    result = num03 ** 3 ** 2 ;
    console.log(result) ;

    result = num03 ** (3 ** 2) ;
    console.log(result) ;

    result = (num03 ** 3) ** 2 ;
    console.log(result) ;
```

【相关说明】

```
let num01 = 125, num02 = 10, num03 = 5 ;
let result = 0 ;
```

这两个语句声明了具有初始数据的变量 num01、num02、num03 与 result。

```
result = num01 % num02 ;
```

此语句使得变量 result，被赋予了**【变量 num01 的数值**除以**变量 num02 的数值】**的余数。

```
result = num01 / num02 ;
```

此语句使得变量 result，被赋予了**【变量 num01 的数值**除以**变量 num02 的数值】**的结果值。

```
result = num01 * num02 ;
```

此语句使得变量 result，被赋予了**【变量 num01 的数值**乘以**变量 num02 的数值】**的结果值。

```
result = num01 + num02 - num03 ;
```

此语句使得变量 result，被赋予了【变量 num01 的数值加上变量 num02 的数值，再减去变量 num03 的数值】的结果值。

```
result = ++num01 ;
```

此语句被执行之前，变量 num01 的数值为 125。

因为运算符 ++ 出现在变量 num01 的左侧，也就意味着 num01 的数值要先递增为 126，再执行【result = num01】。

所以，result 的数值最终为 126。

```
result = num01++ ;
```

此语句被执行之前，变量 num01 的数值已经变成 126。

因为运算符 ++ 出现在变量 num01 的右侧，也就意味着要先执行【result = num01】，之后 num01 的数值才递增为 127。

所以，result 的数值最终仍然为 126。

```
++num01 ;
num01++ ;
```

这两个语句被执行之前，变量 num01 的数值已经变成 127。因为表达式++num01 与 num01++ 均单独出现在语句中，所以可简单视为变量 num01 的数值，被进行了 2 次递增，变成 129。

```
num01 += 1 ;
```

此语句被执行之前，变量 num01 的数值已经变成 129。此语句等同于递增变量 num01 的数值，使得 num01 的数值成为 130。

```
num01 = num01 + 1 ;
```

此语句被执行之前，变量 num01 的数值已经变成 130。此语句亦等同于递增变量 num01 的数值，成为 131。

```
result = --num02 ;
```

此语句被执行之前，变量 num02 的数值为 10。因为运算符 -- 出现在变量 num02 的**左侧**，也就意味着 num02 的数值要先递减为 9，再执行【result = num02】。所以，result 的数值最终为 9。

```
result = num02-- ;
```

此语句被执行之前，变量 num02 的数值已经变成 9。因为运算符 -- 出现在变量 num02 的**右侧**，也就意味着要先执行【result = num02】，之后 num02 的数值才递减为 8。所以，result 的数值最终仍然为 9。

```
--num02 ;
num02-- ;
```

这两个语句被执行之前，变量 num02 的数值已经变成 8。因为运算符 --num02 与 num02--均单独出现在语句中，所以可简单视为变量 num01 的数值，被进行了两次递减，变成 6。

```
num02 -= 1 ;
```

此语句被执行之前，变量 num02 的数值已经变成 6。此语句等同于递减变量 num02 的数值，成为 5。

```
num02 = num02 - 1 ;
```

此语句被执行之前，变量 num02 的数值已经变成 5。此语句亦等同于递减变量 num02 的数值，成为 4。

```
result = num03 ** 3 ** 2 ;
```

在此，变量 num03 的数值是 5。因为求幂运算符 ** 具有右结合的特征，所以【3 ** 2】要先被评估成为 3^2，也就是 9，然后再评估【num03 ** 9】，结果值为 5^9，也就是 1953125。

```
result = num03 ** (3 ** 2) ;
```

在此，借助小括号运算符()的辅助，可明确得知【(3 ** 2)】会优先被评估成为 3^2，也就是 9，然后评估【num03 ** 9】的结果值为 5^9，也就是 1953125。

```
result = (num03 ** 3) ** 2 ;
```

在此，使用小括号运算符()的辅助，可明确得知【(num03 ** 3)】会优先被评估成为 5^3，也就是 125，然后再评估【125** 2】的结果值为 125^2，也就是 15625。

2.2.2 赋值运算符

JavaScript 编程语言的赋值运算符（assignment operator）存在基本形式的等号【=】，以及如表 2-2 所示的复合形式。

表 2-2 JavaScript 的赋值运算符

赋值运算符	用法	含义
=	a = b	
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
<<=	a <<= b	a = a << b
>>=	a >>= b	a = a >> b
>>>=	a >>>= b	a = a >>> b
&=	a &= b	a = a & b
^=	a ^= b	a = a ^ b
|=	a |= b	a = a | b

关于赋值运算符的运用，可参考如下示例。

【2-2-2-assignment-operators.js】

```
let a = 18, b = 5, c = 2 ;
let result = 0 ;

result = a + b + c ;
console.log(result) ;

// a = a + b ;
a += b ;
console.log(a) ;

// a = a - b ;
a -= b ;
console.log(a) ;

// a = a * b ;
a *= b ;
```

```
console.log(a) ;

// a = a / b ;
a /= b ;
console.log(a) ;

// a = a % b ;
a %= b ;
console.log(a) ;

// b = b ** c ;
b **= c ;
console.log(b) ;

// b = b << c ;
b <<= c ;
console.log(b) ;

// b = b >> c ;
b >>= c ;
console.log(b) ;

// b = b >>> c ;
b >>>= c ;
console.log(b) ;

// b = b & c ;
b &= c ;
console.log(b) ;

// b = b ^ c ;
b ^= c ;
console.log(b) ;

// b = b | c ;
b |= c ;
console.log(b) ;
```

【相关说明】

```
let a = 18, b = 5, c = 2 ;
let result = 0 ;
```

这两个语句声明了具有初始数据的变量 a、b、c 与 result。

```
result = a + b + c ;
```

此语句里的赋值运算符【=】，使得变量 result 的数值，成为变量 a、b、c 各数值的总和。

```
// a = a + b ;
a += b ;
```

此语句里的赋值运算符【+=】，使得变量 a 的数值，成为【变量 a 本身的数值加上变量 b 的数值】的结果值。

```
// a = a - b ;
```

```
a -= b ;
```

此语句里的赋值运算符【-=】，使得变量 a 的数值，成为【变量 a 本身的数值减去变量 b 的数值】的结果值。

```
// a = a * b ;
a *= b ;
```

此语句里的赋值运算符【*=】，使得变量 a 的数值，成为【变量 a 本身的数值乘以变量 b 的数值】的结果值。

```
// a = a / b ;
a /= b ;
```

此语句里的赋值运算符【/=】，使得变量 a 的数值，成为【变量 a 本身的数值除以变量 b 的数值】的结果值。

```
// a = a % b ;
a %= b ;
```

此语句里的赋值运算符【%=】，使得变量 a 的数值，成为【变量 a 本身的数值除以变量 b 的数值】的余数。

```
// b = b ** c ;
b **= c ;
```

此语句里的赋值运算符【**=】，使得变量 b 的数值，成为**【变量 b 本身数值**的 c 幂次**】**（b^c）的结果值。

```
// b = b << c ;
b <<= c ;
```

此语句里的赋值运算符【<<=】，使得变量 b 的二进制数值，向**左**偏移**变量 c** 所代表的**比特位（bit）个数**。

```
// b = b >> c ;
b >>= c ;
```

此语句里的赋值运算符【>>=】，使得变量 b 的二进制数值，在保留正负号的前提下，向右偏移变量 c 所代表的比特位（bit）个数。所谓的保留正负号就是：

- 若该二进制数值为负值，则其最左侧的符号位（sign bit）是 1，并在向右偏移的同时，保持其符号位为 1。
- 若该二进制数值为正值，则其最左侧的符号位是 0，并在向右偏移的同时，保持其符号位为 0。

```
// b = b >>> c ;
b >>>= c ;
```

此语句里的赋值运算符【>>>=】，使得变量 b 的数值，成为【变量 b 的二进制数值，向右偏移变量 c 所代表的比特位（bit）个数，并在其左侧补上相同个数的二进制 0】之后的结果值。

在其左侧补上相同个数的二进制 0，也就意味着，一开始无论其最左侧的符号位是 0（正值）或是 1（负值），后续皆在向右偏移的同时，保持符号位成为 0。

```
// b = b & c ;
```

```
b &= c ;
```

此语句里的赋值运算符【&=】，使得变量 b 的数值，成为【变量 b 本身的二进制数值与变量 c 的二进制数值，进行按位与（bitwise and）运算】之后的结果值。

```
// b = b ^ c ;
b ^= c ;
```

此语句里的赋值运算符【^=】，使得变量 b 的数值，成为【变量 b 的二进制数值与变量 c 的二进制数值，进行按位异或（bitwise exclusive or）运算】之后的结果值。

```
// b = b | c ;
b |= c ;
```

此语句里的赋值运算符【^=】，使得变量 b 的数值，成为【变量 b 的二进制数值与变量 c 的二进制数值。进行按位或（bitwise or）运算】之后的结果值。

2.2.3　比较运算符

在各编程语言中，比较运算符（comparison operator）是用来决定其**两侧**操作数相等或不相等的关系。JavaScript 编程语言的比较运算符如表 2-3 所示。

表 2-3　JavaScript 语言的比较运算符

<table>
<tr><th>比较运算符</th><th>用法</th><th>含义</th><th>判断后的返回值</th></tr>
<tr><td>==</td><td>a == b</td><td>判断变量 a 与 b 的数值，是否相等</td><td rowspan="8">• 为真，返回 true
• 为假，返回 false</td></tr>
<tr><td>===</td><td>a === b</td><td>判断变量 a 与 b 的数值，是否相等，以及其数据类型是否也相同</td></tr>
<tr><td>!=</td><td>a != b</td><td>判断变量 a 与 b 的数值，是否并不相等</td></tr>
<tr><td>!==</td><td>a !== b</td><td>判断变量 a 与 b 的数值，是否并不相等，或者其数据类型是否也不同</td></tr>
<tr><td>></td><td>a > b</td><td>判断变量 a 的数值，是否大于变量 b 的数值</td></tr>
<tr><td>>=</td><td>a >= b</td><td>判断变量 a 的数值，是否大于或者等于变量 b 的数值</td></tr>
<tr><td><</td><td>a < b</td><td>判断变量 a 的数值，是否小于变量 b 的数值</td></tr>
<tr><td><=</td><td>a <= b</td><td>判断变量 a 的数值，是否小于或者等于变量 b 的数值</td></tr>
</table>

关于比较运算符的综合运用，可参考如下示例。

【2-2-3-comparison-operators.js】

```
let v01 = 100 , v02 = 250, v03 = 500 ;
let s01 = '100', s02 = '250', s03 = '500' ;

console.log(v01 == s01) ;
console.log(v01 === s01) ;

console.log(v02 > s01) ;
console.log(v02 < s03) ;

console.log(v03 >= s03) ;
console.log(v03 <= s03) ;
```

【相关说明】

```
let v01 = 100 , v02 = 250, v03 = 500 ;
```

此语句声明了初始数值为整数常量的变量 v01、v02 和 v03。

```
let s01 = '100', s02 = '250', s03 = '500' ;
```

此语句声明了初始数据为字符串字面量的变量 s01、s02 与 s03。

```
console.log(v01 == s01) ;
```

判断变量 v01 与 s01 的数据是否相同，并显示判断为真的返回值 true，于浏览器的调试工具【Console】面板中。

```
console.log(v01 === s01) ;
```

判断变量 v01 与 s01 的数据是否相同，以及其数据类型是否也相同，并显示判断为假的返回值 false，于浏览器的调试工具【Console】面板中。

```
console.log(v02 > s01) ;
```

判断变量 v02 的数值是否大于变量 s01 的数据，并显示判断为真的返回值 true，于浏览器的调试工具【Console】面板中。

```
console.log(v02 < s03) ;
```

判断变量 v02 的数值是否小于变量 s03 的数据，并显示判断为真的返回值 true，于浏览器的调试工具【Console】面板中。

```
console.log(v03 >= s03) ;
```

判断变量 v03 的数值是否大于**或者**等于变量 s03 的数据，并显示判断为真的返回值 true 于浏览器调试工具的【Console】面板中。

```
console.log(v03 <= s03) ;
```

判断变量 v03 的数值是否小于**或者**等于变量 s03 的数据，并显示判断为真的返回值 true，于浏览器的调试工具【Console】面板中。

2.2.4 逻辑运算符

在各种编程语言中，逻辑运算符（logical operator）主要用来串联带有比较含义的表达式。JavaScript 编程语言的逻辑运算符如表 2-4 所示。

表 2-4 JavaScript 语言的逻辑运算符

逻辑运算符	用法	含义	判断后的返回值
&&	e1 && e2	判断表达式 e1 与 e2，是否皆为真	• 为真，返回 true • 为假，返回 false
||	e1 || e2	判断表达式 e1 或 e2，是否**其中之一**为真	
!	! e1	判断表达式 e1，是否**并非**为真	
in	e1 in e2	判断 e1 所代表的属性名称，是否存在于 e2 所代表的对象实例中	

关于逻辑运算符的综合运用，可参考如下示例。

【2-2-4-logical-operators.js】

```
let v01 = 10, v02 = 50, v03 = 60 ;
let values = [10, 20, 30, 40, 50] ;
let person = {name: 'Gary', gender: 'male', age: '25'} ;

// result = true && true ;
result = v01 < v02 && v02 < v03 ;
console.log(result) ;

// result = false || true ;
result = v01 > v02 || v03 > v02 ;
console.log(result) ;

// result = ! false ;
result = ! (v01 > v02) ;
console.log(result) ;

result = 2 in values ;
console.log(result) ;

result = 'length' in values ;
console.log(result) ;

result = 'round' in Math ;
console.log(result) ;

result = 'gender' in person ;
console.log(result) ;
```

【相关说明】

```
let v01 = 10, v02 = 50, v03 = 60 ;
```

此语句声明了初始数值为整数常量的变量 v01、v02 与 v03。

```
let values = [10, 20, 30, 40, 50] ;
```

此语句声明了初始数据为数组实例的变量 values。

```
let person = {name: 'Gary', gender: 'male', age: '25'} ;
```

此语句声明了初始数据为对象实例的变量 person。

```
// result = true && true ;
result = v01 < v02 && v02 < v03 ;
```

【v01 < v02】为真，所以返回 true；【v02 < v03】为真，所以返回 true。因此，【v01 < v02 && v02 < v03】亦为真而返回 true，使得变量 result 的数据成为布尔值 true。

```
// result = false || true ;
result = v01 > v02 || v03 > v02 ;
```

【v01 > v02】为假，所以返回 false；【v03 > v02】为真，所以返回 true。因此，【v01 > v02 || v03 > v02】亦为真而返回 true，使得变量 result 的数据成为布尔值 true。

```
// result = ! false ;
result = ! (v01 > v02) ;
```

【v01 > v02】为假，所以返回 false。因此，【! (v01 > v02)】为真而返回 true，使得变量 result 的数据成为布尔值 true。

```
result = 2 in values ;
```

变量 values 的数据是数组实例 [10, 20, 30, 40, 50]，所以 values[0]可访问到数组实例的元素值 10；values[4]可访问到数组实例的元素值 50。

在数组实例名称 values 右侧的中括号[]里面，例如 values[3]，存在作为索引值（index value）的整数值 3，以访问其索引值 3 所代表的特定元素 40。

在此，因为索引值 2（第 3 个）对应到 values[2]所代表的第 3 个元素值 30，所以【2 in values】为真而返回 true，使得变量 result 的数据成为布尔值 true。

```
result = 'length' in values ;
```

变量 values 的数据是数组实例，因此变量 values 具有属性 length，可用来获取 values.length 所代表元素个数的整数值 5。所以，【'length' in values】为真而返回 true，使得变量 result 的数据成为布尔值 true。

```
result = 'round' in Math ;
```

内置的对象 Math 具有函数 round()，所以【'round' in Math】为真而返回 true，进而使得变量 result 的数据成为布尔值 true。

```
result = 'gender' in person ;
```

变量 person 的数据是对象实例{name: 'Gary', gender: 'male', age: '25'}，所以属性 gender 可用来访问到 person.gender 所代表属性的数据'male'。所以，【'gender' in person】为真而返回 true，使得变量 result 的数据成为布尔值 true。

2.2.5　条件运算符

从 C 语言开始，许多后继的编程语言，均支持条件运算符（conditional operator）/三元运算符（ternary operator），并用来简化特定形式的 if 语句。关于条件运算符的运用，可参考如下示例。

【2-2-5-conditional-operator.js】

```
let score, passed ;

score = 58 ;

passed = score >= 60 ? 'yes' : 'no' ;
console.log(passed) ;

score = 77 ;

passed = score >= 60 ? 'yes' : 'no' ;
console.log(passed) ;
```

```
score = 60 ;

if (score >= 60) passed = 'yes' ;
else passed = 'no' ;

console.log(passed) ;
```

【相关说明】

```
let score, passed ;
```

此语句声明了变量 score 与 passed。

```
score = 58 ;
```

此语句使得变量 score，被赋予整数值 58。

```
passed = score >= 60 ? 'yes' : 'no' ;
```

对于表达式【score >= 60 ? 'yes' : 'no'】而言，若【score >= 60】为真，则返回字符串'yes'；否则返回字符串'no'。在此，其为假，所以返回字符串'no'，进而使得变量 passed 的数据，成为字符串'no'。

```
score = 77 ;
```

此语句使得变量 score，被赋予整数值 77。

```
passed = score >= 60 ? 'yes' : 'no' ;
```

在此，表达式【score >= 60 ? 'yes' : 'no'】返回字符串'yes'，进而使得变量 passed 的数据，成为字符串'yes'。

```
score = 60 ;
```

此语句使得变量 score，被赋予整数值 60。

```
if (score >= 60) passed = 'yes' ;
else passed = 'no' ;
```

对于此条件语句而言，若【score >= 60】为真，则变量 passed 会被赋予字符串'yes'；否则会被赋予字符串'no'。在此，其为真，所以返回字符串'yes'，进而使得变量 passed，被赋予字符串'yes'。

2.2.6　类型运算符

类型运算符（typeof operator）用来返回特定操作数（operand）的数据类型（data type）。关于类型运算符的综合运用，可参考如下示例。

【2-2-6-typeof-operator.js】

```
let num01 = 33, num02 = 1.414 ;

console.log(typeof num01) ;
console.log(typeof 33) ;
console.log(typeof num02) ;
console.log(typeof 1.414) ;
```

```
    console.log('') ;

    console.log(typeof Math.PI) ;
    console.log(typeof NaN) ;
    console.log(typeof Infinity) ;
    console.log('') ;

    console.log(typeof '') ;
    console.log(typeof "") ;
    console.log(typeof "Hello, Earth!") ;
    console.log('') ;

    console.log(typeof true) ;
    console.log(typeof false) ;
    console.log(typeof (num01 > num02)) ;
    console.log('') ;

    console.log(typeof undefined) ;
    console.log(typeof num03) ;
    console.log('') ;

    console.log(typeof function() {}) ;
    console.log(typeof Array.isArray) ;
    console.log('') ;

    console.log(typeof Object()) ;
    console.log(typeof new Object()) ;
    console.log(typeof {}) ;
    console.log('') ;

    console.log(typeof Array()) ;
    console.log(typeof new Array()) ;
    console.log(typeof []) ;
    console.log('') ;

    console.log(typeof null) ;
    console.log('') ;

    console.log(typeof String('test')) ;
    console.log(typeof new String('test')) ;
    console.log('') ;

    console.log(typeof Number(123)) ;
    console.log(typeof new Number(123)) ;
    console.log('') ;

    console.log(typeof Date()) ;
    console.log(typeof new Date()) ;
```

【相关说明】

```
let num01 = 33, num02 = 1.414 ;
```

此语句声明了具有初始数值的变量 num01 与 num02。

```
console.log(typeof num01) ;
```

表达式【typeof num01】会得出变量 num01 的数据类型名称，在此为字符串'number'。

```
console.log(typeof 33) ;
```

表达式【typeof 33】会得出常量 33 的数据类型名称，在此为字符串'number'。

```
console.log(typeof num02) ;
```

表达式【typeof num02】会得出变量 num02 的数据类型名称，在此为字符串'number'。

```
console.log(typeof 1.414) ;
```

表达式【typeof 1.414】会得出常量 1.414 的数据类型名称，在此为字符串'number'。

```
console.log(typeof Math.PI) ;
```

表达式【typeof Math.PI】会得出内置对象 Math 的常量属性 PI 的数据类型名称，在此为字符串'number'。

```
console.log(typeof NaN) ;
```

表达式【typeof NaN】会得出原始常量 NaN 的数据类型名称，在此为字符串'number'。

```
console.log(typeof Infinity) ;
```

表达式【typeof Infinity】会得出内置常量 Infinity 的数据类型名称，在此为字符串'number'。

```
console.log(typeof '') ;
```

表达式【typeof''】会得出**空**字符串''的数据类型名称，在此为字符串'string'。

```
console.log(typeof "") ;
```

表达式【typeof ""】会得出**空**字符串""的数据类型名称，在此为字符串'string'。

```
console.log(typeof "Hello, Earth!") ;
```

表达式【typeof "Hello, Earth!"】会得出字符串"Hello, Earth!"的数据类型名称，在此为字符串'string'。

```
console.log(typeof true) ;
```

表达式【typeof true】会得出布尔值 true 的数据类型名称，在此为字符串'boolean'。

```
console.log(typeof false) ;
```

表达式【typeof false】会得出布尔值 false 的数据类型名称，在此为字符串'boolean'。

```
console.log(typeof (num01 > num02)) ;
```

表达式【typeof (num01 > num02)】会得出子表达式【num01 > num02】返回值的数据类型名称，在此为字符串'boolean'。

```
console.log(typeof undefined) ;
```

值得关注的是，原始常量 undefined 的数据类型亦是 undefined。表达式【typeof undefined】会得出原始常量 undefined 的数据类型名称，在此为字符串'undefined'。

```
console.log(typeof num03) ;
```

在此，num03 未被声明为变量或函数名称，所以 num03 处于未定义（undefined）的状态。表达式【typeof num03】会得出未被声明的 num03 的数据类型名称，在此为字符串'undefined'。

```
console.log(typeof function() {}) ;
```

表达式【typeof function() {}】会得出匿名函数的数据类型名称，在此为字符串'function'。

```
console.log(typeof Array.isArray) ;
```

表达式【typeof Array.isArray】会得出内置对象 Array 的函数 isArray()的数据类型名称，在此为字符串'function'。

```
console.log(typeof Object()) ;
```

因为 Object 对象的构造函数 Object()，会返回 Object 对象的实例；也因此表达式【typeof Object()】会得出 Object 对象实例的数据类型名称，在此为字符串'object'。

```
console.log(typeof new Object()) ;
```

表达式【typeof new Object()】也会得出 Object 对象实例的数据类型名称，在此亦为字符串'object'。此外，JavaScript 语言尚未界定【Object()】与【new Object()】的明显区别。

```
console.log(typeof {}) ;
```

表达式【typeof {}】会得出空对象实例{}的数据类型名称，在此亦为字符串'object'。而在 JavaScript 编程语言中，如下 3 个语句是等价的：

- obj = {} ;
- obj = new Object() ;
- obj = Object() ;

```
console.log(typeof Array()) ;
```

因为 Array 对象的构造函数 Array()，会返回 Array 对象的实例；也因此表达式【typeof Array()】会得出 Array 对象实例的数据类型名称，在此竟然也为字符串'object'。

```
console.log(typeof new Array()) ;
```

表达式【typeof new Array()】亦会得出 Array 对象实例的数据类型名称，在此亦为字符串'object'。此外，JavaScript 编程语言尚未界定【Array()】与【new Array()】的明显区别。

```
console.log(typeof []) ;
```

表达式【typeof []】会得出空数组实例[]的数据类型名称，在此为字符串'object'。而在 JavaScript 语言中，如下 3 个语句是等价的：

- arr = [] ;
- arr = new Array() ;
- arr = Array() ;

在 JavaScript 语言里，将数组实例（array instance）的数据类型视为 object。

```
console.log(typeof null) ;
```

【typeof null】表达式会得出原始常量 null 的数据类型名称，在此为字符串'object'。

```
console.log(typeof String('test')) ;
```

内置对象 String 的构造函数 String('test')，只会返回字符串'test'。因此，表达式【typeof String('test')】等同于获得字符串'test'的数据类型名称，在此为字符串'string'。

```
console.log(typeof new String('test')) ;
```

内置对象 String 的构造函数 String('test')会返回字符串'test'。再经过 new 关键字的处理，会返回内含字符串'test'的 String 对象实例。因此，表达式【typeof new String('test')】等同于获得 String 对象实例的数据类型名称，在此为字符串'object'。

```
console.log(typeof Number(123)) ;
```

内置对象 Number 的构造函数 Number(123)，会返回整数常量 123。因此，【typeof Number(123)】表达式等同于获得整数常量 123 的数据类型名称，在此为字符串'number'。

```
console.log(typeof new Number(123)) ;
```

内置对象 Number 的构造函数 Number(123)，会返回整数常量 123。再经过 new 关键字的处理，会返回内含整数常量 123 的 Number 对象实例。因此，表达式【typeof new Number(123)】等同于获得 Number 对象**实例**的数据类型名称，在此为字符串'object'。

```
console.log(typeof Date()) ;
```

内置 Date 对象的构造函数 Date()，会返回当前日期与时间的字符串。因此，表达式【typeof Date()】等同于获得当前日期与时间的字符串的数据类型名称，在此为字符串'string'。

```
console.log(typeof new Date()) ;
```

内置 Date 对象的构造函数 Date()，会返回当前日期与时间的字符串。再经过 new 关键字的处理，会返回内含当前日期与时间的 Date 对象实例。因此，表达式【typeof new Date()】等同于获得 Date 对象实例的数据类型名称，在此为字符串'object'。

2.2.7　按位运算符

JavaScript 语言的按位运算符（bitwise operator），在被运算之前，其两侧操作数（operand）的数据，会被转换成为 32 个比特位的数据。关于按位运算符的综合运用，可参考如下示例。

【2-2-7-bitwise-operators.js】

```
let num01 = 56, num02 = 77 ;
let num03 = 124, num04 = -3 ;

console.log(num01.toString(2)) ;
console.log(num02.toString(2)) ;
console.log(num03.toString(2)) ;
console.log('') ;

console.log(num01 == 0b111000) ;
console.log(num01 == 0b00111000) ;
```

```
console.log('') ;

console.log(num02 == 0b1001101) ;
console.log(num02 == 0b01001101) ;
console.log('') ;

console.log(num03 == 0b1111100) ;
console.log(num03 == 0b01111100) ;
console.log('') ;

/*
 00111000
&01001101
----------
 00001000
*/
result = num01 & num02 ;

console.log(result) ;
console.log(result.toString(2)) ;
console.log('') ;

/*
 00111000
|01001101
----------
 01111101
*/
result = num01 | num02 ;

console.log(result) ;
console.log(result.toString(2)) ;
console.log('') ;

/*
 00111000
^01001101
----------
 01110101
*/
result = num01 ^ num02 ;

console.log(result) ;
console.log(result.toString(2)) ;
console.log('') ;

/*
 ~ 01111100
------------
   10000011 (negative value)
=
  -01111100
 + 00000001 (because of 2's complement)
------------
  -01111101
```

```
*/
result = ~ num03 ;

console.log(result) ;
console.log(result.toString(2)) ;
console.log('') ;

/*
<< 001111100
------------
   011111000
*/
result = num03 << 1 ;

console.log(result) ;
console.log(result.toString(2)) ;
console.log('') ;

/*
>> 001111100
------------
   000111110
*/
result = num03 >> 1 ;

console.log(result) ;
console.log(result.toString(2)) ;
console.log('') ;

/*
but -3 actually is
    11111111111111111111111111111101 in 32-bit 2's complement.
So,
>>> 11111111111111111111111111111101
--------------------------------------
    01111111111111111111111111111110
*/
result = num04 >>> 1 ;

console.log(num04.toString(2)) ;
console.log(result) ;
console.log((2 ** 31 - 1) - 1) ;
console.log(result.toString(2)) ;
```

【相关说明】

```
let num01 = 56, num02 = 77 ;
let num03 = 124, num04 = -3 ;
```

这两个语句声明了初始数据为整数值的变量 num01、num02、num03 与 num04。

```
console.log(num01.toString(2)) ;
console.log(num02.toString(2)) ;
console.log(num03.toString(2)) ;
```

这 3 个语句中的【.toString(2)】，可分别将变量 num01、num02 与 num03 的整数值，转换成

为字符串类型的**二进制**数码。

```
console.log(num01 == 0b111000) ;
```

【num01 == 0b111000】可用来判断变量 num01 的数值，是否等于二进制整数【111000】。在此，其返回 true。加上【0b】在二进制数码的左侧，可使得此代码被视为二进制数值。

在此亦可得知，浏览器中的 JavaScript 引擎，仍然将**最左比特位**（left-most bit）为 1 的数值，视为正值（positive value）。因此，若要变更成为负值（negative value），则修改为【- 0b111000】。

```
console.log(num01 == 0b00111000) ;
```

【num01 == 0b00111000】可用来判断变量 num01 的数值，是否等于二进制整数【00111000】。在此，其返回 true。

在此可得知，浏览器中的 JavaScript 引擎将【0b00111000】与【0b111000】，视为相同的二进制数值。

```
console.log(num02 == 0b1001101) ;
console.log(num02 == 0b01001101) ;
```

【num02 == 0b1001101】可用来判断变量 num02 的数值，是否等于二进制整数【1001101】。在此，其返回 true。

【num02 == 0b01001101】可用来判断变量 num02 的数值，是否等于二进制整数【01001101】。在此，其返回 true。

在此可得知，JavaScript 引擎将【0b1001101】与【0b01001101】，视为相同的二进制数值。

```
console.log(num03 == 0b1111100) ;
console.log(num03 == 0b01111100) ;
```

【num03 == 0b1111100】可用来判断变量 num03 的数值，是否等于二进制整数【0b1111100】。在此，其返回 true。

【num03 == 0b01111100】可用来判断变量 num03 的数值，是否等于二进制整数【0b01111100】。在此，其返回 true。

由此可得知，JavaScript 引擎将【0b1111100】与【0b01111100】，视为相同的二进制数值。

```
/*
 00111000
&01001101
----------
 00001000
*/
result = num01 & num02 ;
```

【num01 & num02】可得到【变量 num01 与 num02 的二进制数据，进行**按位和**（bitwise and）运算】之后的结果值。

```
/*
 00111000
|01001101
----------
 01111101
*/
```

```
result = num01 | num02 ;
```

【num01 | num02】可得到【变量 num01 与 num02 的二进制数据，进行**按位或**（bitwise or）运算】之后的结果值。

```
/*
 00111000
^01001101
----------
 01110101
*/
result = num01 ^ num02 ;
```

【num01 ^ num02】可得到【变量 num01 与 num02 的二进制数据，进行**按位异或**（bitwise exclusive or）运算】之后的结果值。

```
/*
 ~ 01111100
------------
   10000011 (negative value)
=
  -01111100
 + 00000001 (because of 2's complement)
------------
  -01111101
*/
result = ~ num03 ;
```

【~ num03】会得到【变量 num03 的二进制数据，进行 **2 的补码**（two's complement）】之后的结果值。

```
/*
<< 001111100
------------
    011111000
*/
result = num03 << 1 ;
```

【num03 << 1】会得出【变量 num03 的 2 进位数据，向**左**偏移 1 个比特位】之后的结果值。

```
/*
>> 001111100
------------
    000111110
*/
result = num03 >> 1 ;
```

【num03 >> 1】会得出【变量 num03 的 2 进位数据，向**右**偏移 1 个比特位】之后的结果值。

```
/*
but -3 actually is
    11111111111111111111111111111101 in 32-bit 2's complement.
So,
>>> 11111111111111111111111111111101
--------------------------------------
    01111111111111111111111111111110
```

```
*/
result = num04 >>> 1 ;
```

【num04 >>> 1】会得出【变量 num04 的 2 进位数据，向**右**偏移 1 个比特位的同时，在其最左侧的符号位（sign bit），填入 0】之后的结果值。

此语句被执行之前，变量 num04 的数值为负整数-3。此语句被执行之后，变量 num04 的数值却变成非常大的正整数 2147483646，可见其符号位被填入了 0。

```
console.log((2 ** 31 - 1) - 1) ;
```

【(2 ** 31 - 1) - 1】会评估出$(2^{31} - 1) - 1$，也就是正整数 2147483646。

2.2.8 括号运算符

括号运算符包含：

- 小括号 / 圆括号（parentheses, round brackets）运算符()：除了用于变更特定表达式的运算优先级之外，亦被用于 if、switch、for、while 等语句和函数的调用。
- 中括号 / 方括号（brackets, square brackets）运算符[]：主要用于数组（array）或字符串（string）相关的定义和访问。
- 大括号 / 花括号（braces, curly brackets）运算符{}：用于语句的分组（grouping）、函数的主体构造，以及对象（object）的定义。

关于括号运算符的综合运用，可参考如下示例。

【2-2-8-brackets-operators.js】

```
let a = 10, b = 5, c = 3 ;
let result = 0 ;

result = a - b * c ;
console.log(result) ;

result = (a - b) * c ;
console.log(result) ;

result = a ** c ** 2 ;
console.log(result) ;

result = (a ** c) ** 2 ;
console.log(result) ;
console.log('') ;

///
let now = new Date() ;

console.log(now) ;
console.log(now.toLocaleString()) ;
console.log('') ;
```

```
let obj = new Object() ;

obj.name = 'Jasper' ;
obj.gender = 'male' ;
obj.age = 28 ;

console.log(obj) ;
console.log('') ;

///
function display(choice, message)
{
  if (choice == 1) alert(message) ;
  else if (choice == 2) confirm(message) ;
}

display(1, 'Hello, Earth!') ;
display(2, 'Hello, are you human?') ;

///
let fruits = ['apple', 'banana', 'cherry', 'durian'] ;

console.log(fruits[1]) ;
```

【相关说明】

```
let a = 10, b = 5, c = 3 ;
let result = 0 ;
```

这两个语句声明了具有初始数值的变量 a、b、c 和 result。

```
result = a - b * c ;
```

表达式【a - b * c】会先被计算出【b * c】的结果值，再计算出整个表达式的结果值。

```
result = (a - b) * c ;
```

因为小括号运算符的缘故，表达式【(a - b) * c】会先被计算出【a - b】的结果值，再计算出整个表达式的结果值。

```
result = a ** c ** 2 ;
```

因为求幂运算符**具有**右结合**的特征，所以表达式【a ** c ** 2】会先被计算出【c ** 2】的结果值，再计算整个表达式的结果值。

```
result = (a ** c) ** 2 ;
```

因为小括号运算符的缘故，表达式【(a ** c) ** 2】会先被计算出【a ** c】的结果值，再被计算出整个表达式的结果值。

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

此语句使得变量 now，具有初始数据为【内含当前日期与时间】的 Date 对象实例。

```
console.log(now) ;
```

借助此语句，可在网页浏览器的调试工具【Console】面板中，显示出标准格式的日期与时间。例如：【Fri Dec 29 2017 01:16:16 GMT+XX00 (XX 标准时间)】。

```
console.log(now.toLocaleString()) ;
```

【now.toLocalString()】可将日期与时间，以精简的本地格式，显示出来。例如：【2017/12/29 上午 1:16:16】。

```
let obj = new Object() ;
```

因为小括号运算符的缘故，使得 Object 对象的构造函数 Object()被调用。此语句使得变量 obj 的初始数据，成为 Object 对象的**空**实例（empty instance）。

```
obj.name = 'Jasper' ;
obj.gender = 'male' ;
obj.age = 28 ;
```

这 3 个语句分别设置了变量 obj 的新属性 name、gender、age 和其个别数据'Jasper'、'male'、28。

```
console.log(obj) ;
```

此语句可显示出变量 obj 的数据，也就是对象实例{name: "Jasper", gender: "male", age: 28}。

```
function display(choice, message)
```

因为小括号运算符的缘故，网页浏览器或其他软件中的 JavaScript 引擎，可认出关键字 function 开头的此源代码片段，即是用来定义【带有参数 choice 和 message，而且其名称为 display】的函数。

```
{
  if (choice == 1) alert(message) ;
  else if (choice == 2) confirm(message) ;
}
```

借助大括号运算符{}，JavaScript 引擎可认出大括号{ ... }里的源代码，即是函数 display()的主体结构。

在函数 display()的主体结构里，仍然可以看到小括号运算符，出现在 if 语句中。另外，亦可看到小括号运算符，出现在内置函数 alert()与 confirm()的调用语法中。

```
display(1, 'Hello, Earth!') ;
display(2, 'Hello, are you human?') ;
```

这两个语句，通过传入不同参数值，而重复调用了函数 display()。

```
let fruits = ['apple', 'banana', 'cherry', 'durian'] ;
```

借助中括号运算符[]，在此语句里的等号右侧，即是内含 4 个字符串元素的数组实例。

```
console.log(fruits[1]) ;
```

此语句通过数组变量的名称 fruits，以及中括号运算符里的元素索引值 1，访问到 fruits[1]所代表的字符串元素'banana'。

2.2.9　扩展运算符（ES6）

简单来说，由 3 个英文句点【...】构成的扩展运算符（spread operator），可被用来【卸除】特定数组的中括号或者特定对象的大括号。关于扩展运算符的综合运用，可参考如下示例。

【2-2-9-spread-operator.js】

```
var greetings = ['Hi', 'Howdy', 'Hey, man', 'G\'day mate'] ;

var extended_greetings = ['Long time no see', 'Nice to see you', 'Hiya' , ... greetings] ;

console.log(extended_greetings) ;
console.log('') ;

var number_texts = ['one', 'two', 'three']
var number_digits = '123' ;

var numbers = [... number_texts, ... number_digits]

console.log(numbers) ;
console.log('') ;

///
let birthday = new Date(1999, 11, 25, 20, 30) ;
let now = new Date() ;

console.log(birthday.toLocaleString()) ;
console.log(now.toLocaleString()) ;
console.log('') ;

///
let arr01 = [1, 2, 3] ;
let arr02 = [10, 20, 30] ;
let arr03 = [... arr01, ... arr02, 100, 200, 300] ;

let obj01 = {name: 'orange', amount: 10} ;
let obj02 = {name: 'durian', amount: 5, origin: 'Thai'} ;
let obj03 = {... obj01, ... obj02} ;

console.log(arr03) ;
console.log(obj03) ;
console.log('') ;
```

【相关说明】

```
var greetings = ['Hi', 'Howdy', 'Hey, man', 'G\'day mate'] ;
```

此语句声明了初始数据为数组实例的变量 greetings。

```
var extended_greetings = ['Long time no see', 'Nice to see you', 'Hiya' , ... greetings] ;
```

此语句声明了初始数据为另一数组实例的变量 extended_greetings。扩展运算符【...】使得变量 extended_greetings 的数组实例，带有变量 greetings 的数组实例中的所有元素。

```
var number_texts = ['one', 'two', 'three']
```

此语句亦声明了初始数据为另一数组实例的变量 number_texts。

```
var number_digits = '123' ;
```

此语句声明了初始数据为字符串'123'的变量 number_digits。

```
var numbers = [... number_texts, ... number_digits]
```

扩展运算符【...】使得变量 numbers 的初始数据，成为合并【变量 number_texts 与 number_digits 的个别数组实例】的新数组实例，进而使得变量 numbers 的数据，成为数组实例["one", "two", "three", "1", "2", "3"]。

```
let birthday = new Date(1999, 11, 25, 20, 30) ;
```

此语句声明了【初始数据为内含日期与时间 1999/12/25 20:30:00 的 Date 对象实例】的变量 birthday。

需留意的是，构造函数 Date()小括号中的第 2 个参数值为 11，其实是代表 12 月份的含义。换句话说，此参数值若为 0，则代表 1 月份。

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

此语句声明了【初始数据为**当前日期与时间**的 Date 对象实例】的变量 now。

```
console.log(birthday.toLocaleString()) ;
```

此语句会显示出【1999/12/25 下午 8:30:00】的信息。

```
console.log(now.toLocaleString()) ;
```

此语句显示出当前日期与时间的信息。

```
let arr01 = [1, 2, 3] ;
let arr02 = [10, 20, 30] ;
```

这两个语句声明了【初始数据为不同数组实例】的变量 arr01 与 arr02。

```
let arr03 = [... arr01, ... arr02, 100, 200, 300] ;
```

此语句声明了变量 arr03，并借助扩展运算符【...】，使得其【初始数据为合并变量 arr01 与 arr02 的数组实例，再衔接子数组实例[100, 200, 300]】的新数组实例[1, 2, 3, 10, 20, 30, 100, 200, 300]。

```
let obj01 = {name: 'orange', amount: 10} ;
let obj02 = {name: 'durian', amount: 5, origin: 'Thai'} ;
```

这两个语句声明了【初始数据为对象实例】的变量 obj01 与 obj02。

```
let obj03 = {... obj01, ... obj02} ;
```

此语句声明了变量 obj03，并借助扩展运算符【...】，使得其【初始数据为合并变量 obj01 与 obj02 对象实例】的新对象实例{name: "durian", amount: 5, origin: "Thai"}。

值得注意的是，变量 obj01 与 obj02 均存在属性 name 与 amount。经过扩展运算之后，变量 obj03 的属性 name 与 amount 却只有一个，而且其属性的数据，均个别与变量 obj02 的属性 name 和 amount

的数据，是相同的。

2.2.10　逗号运算符

逗号运算符（comma operator）主要用来并联多个赋值表达式或操作数。关于逗号运算符的运用，可参考如下示例。

【2-2-10-comma-operator.js】

```
let a = 11, b = 21, c = 31 ;
let d, e, f ;

d = e = a + b, f = b + c ;

console.log(d, e, f) ;
```

【相关说明】

```
let a = 11, b = 21, c = 31 ;
```

此语句声明了初始数据均为整数的变量 a、b 与 c。

```
let d, e, f ;
```

此语句声明了未设置初始数据的变量 d、e 与 f。

```
d = e = a + b, f = b + c ;
```

此语句主要由两个表达式【d = e = a + b】与【f = b + c】构成。

- 第 1 个表达式先被计算出【a + b】的结果值，再赋给变量 d 和 e。
- 第 2 个表达式先被计算出【b + c】的结果值，再赋给变量 f。

修改此语句中的逗号运算符【,】，成为代表语句结束的分号【;】，并不影响其结果值。只是，原本的单一语句，就变成在同一行的两个语句了。

```
console.log(d, e, f) ;
```

通过传入变量 d、e 和 f 的数值，作为函数 console.log()的参数值，可让这些变量的数值，显示在同一行的信息里，例如【32 32 52】。

2.2.11　删除运算符

JavaScript 语言中的删除运算符（delete operator），仅用来删除特定对象实例的特定属性。关于删除运算符的运用，可参考如下示例。

【2-2-11-delete-operator.js】

```
let person = {name: 'Ivory', gender: 'female', age: '30'} ;
let colors = ['RoyalBlue', 'GreenYellow', 'Gold', 'Cyan'] ;

var num01 = 123 ;
```

```
let num02 = 456 ;
num03 = 789 ;

console.log(num01, num02, num03) ;
console.log('') ;

delete person.name ;
delete colors[1] ;

delete num01 ;
delete num02 ;
delete num03 ;

console.log(person.name) ;
console.log(person) ;
console.log('') ;

console.log(colors[1]) ;
console.log(colors) ;
console.log('') ;

console.log(num01) ;
console.log(num02) ;
console.log(num03) ;
```

【相关说明】

```
let person = {name: 'Ivory', gender: 'female', age: '30'} ;
```

此语句声明了初始数据为对象实例的变量 person。

```
let colors = ['RoyalBlue', 'GreenYellow', 'Gold', 'Cyan'] ;
```

此语句声明了初始数据为数组实例的变量 colors。

```
var num01 = 123 ;
let num02 = 456 ;
num03 = 789 ;
```

无论有无通过关键字 var 或 let 进行声明，这 3 个语句分别声明了初始数据为不同整数的变量 num01、num02 与 num03。

```
delete person.name ;
```

此语句使用关键字 delete，并配合点运算符【.】，可删除变量 person 的对象实例的属性 name。

```
delete colors[1] ;
```

此语句使用关键字 delete，并配合中括号运算符[]，可在变量 colors 数组实例中，仅删除索引值为 1 的元素数据'GreenYellow'，但是此元素占用的缓存空间，仍然保留在其数组实例中。

```
delete num01 ;
delete num02 ;
delete num03 ;
```

通过这 3 个语句，试图删除变量 num01、num02 与 num03；然而实际上，仅有未通过 var 或 let 关键字，加以声明的变量 num03，可以被成功删除。

```
console.log(person.name) ;
```

【person.name】会返回 undefined，即代表变量 person 的属性 name，已被删除了。

```
console.log(person) ;
```

此语句只会产生{gender: "female", age: "30"}的信息。由此可见，属性 name 和其数据'Ivory'已被删除了。

```
console.log(colors[1]) ;
```

colors[1]在此返回 undefined，即代表变量 colors 中的索引值为 1 的数据，已经被清除了。

```
console.log(colors) ;
```

此语句会产生["RoyalBlue", empty, "Gold", "Cyan"]的信息，可看出 colors[1]对应的元素数据已经被清除了，才会被标记为 empty。

```
console.log(num01) ;
console.log(num02) ;
```

查看这两个语句所产生的信息，可得知变量 num01 与 num02 并未被删除！那是因为这两个变量，是借助 var 或 let 关键字，来加以声明的。

```
console.log(num03) ;
```

执行此语句时，会产生【num03 is not defined 参考错误（reference error）】的信息。由此可知，变量 num03 确实被删除了。

2.2.12　运算符的优先级（ES6）

在特定表达式中，运算符的优先级（operators precedence）可用来决定各子表达式的执行顺序。在相邻的子表达式中，其运算符优先级较高的子表达式，会先被执行。关于各运算符的优先级，如表 2-5 所示。

表 2-5　运算符的优先级

优先级	用途	关联性	运算符
20	变更优先级	无	(…)
19	访问属性或函数	由左至右	… . …
	访问子元素或定义数组实例的元素个数	由左至右	[…]
	创建特定对象【非】默认的实例	无	new … (…)
	调用特定函数	由左至右	… (…)
18	创建特定对象默认的实例	由右至左	new …
17	评估后再递增	无	… ++
	评估后再递减		… --

（续表）

<table>
<tr><th>优先级</th><th>用途</th><th>关联性</th><th>运算符</th></tr>
<tr><td rowspan="10">16</td><td>逻辑非运算</td><td rowspan="10">由右至左</td><td>! …</td></tr>
<tr><td>按位非运算</td><td>~ …</td></tr>
<tr><td>正值</td><td>+ …</td></tr>
<tr><td>负值</td><td>- …</td></tr>
<tr><td>先递增再评估</td><td>++ …</td></tr>
<tr><td>先递减再评估</td><td>-- …</td></tr>
<tr><td>返回数据类型</td><td>typeof …</td></tr>
<tr><td>立即调用匿名函数</td><td>void …</td></tr>
<tr><td>删除特定数据</td><td>delete …</td></tr>
<tr><td>等待评估后的数据</td><td>await …</td></tr>
<tr><td>15</td><td>求幂运算</td><td>由右至左</td><td>… ** …</td></tr>
<tr><td rowspan="3">14</td><td>乘法运算</td><td rowspan="3">由左至右</td><td>… * …</td></tr>
<tr><td>除法运算</td><td>… / …</td></tr>
<tr><td>求余运算</td><td>… % …</td></tr>
<tr><td rowspan="2">13</td><td>加法运算</td><td rowspan="2">由左至右</td><td>… + …</td></tr>
<tr><td>减法运算</td><td>… - …</td></tr>
<tr><td rowspan="3">12</td><td>按位左移运算</td><td rowspan="3">由左至右</td><td>… << …</td></tr>
<tr><td>按位右移运算</td><td>… >> …</td></tr>
<tr><td>按位无符号右移运算</td><td>… >>> …</td></tr>
<tr><td rowspan="6">11</td><td>比较小于的关系</td><td rowspan="6">由左至右</td><td>… < …</td></tr>
<tr><td>比较小于或等于的关系</td><td>… <= …</td></tr>
<tr><td>比较大于的关系</td><td>… > …</td></tr>
<tr><td>比较大于或等于的关系</td><td>… >= …</td></tr>
<tr><td>判断是否存在特定属性</td><td>… in …</td></tr>
<tr><td>判断是否为特定对象的实例</td><td>… instanceof …</td></tr>
<tr><td rowspan="4">10</td><td>判断数据是否相等</td><td rowspan="4">由左至右</td><td>… == …</td></tr>
<tr><td>判断数据是否不相等</td><td>… != …</td></tr>
<tr><td>判断数据【和】类型是否都相同</td><td>… === …</td></tr>
<tr><td>判断数据【或】类型是否不相同</td><td>… !== …</td></tr>
<tr><td>9</td><td>按位与运算</td><td>由左至右</td><td>… & …</td></tr>
<tr><td>8</td><td>按位异或运算</td><td>由左至右</td><td>… ^ …</td></tr>
<tr><td>7</td><td>按位或运算</td><td>由左至右</td><td>… | …</td></tr>
<tr><td>6</td><td>逻辑与运算</td><td>由左至右</td><td>… && …</td></tr>
<tr><td>5</td><td>逻辑或运算</td><td>由左至右</td><td>… || …</td></tr>
<tr><td>4</td><td>条件运算</td><td>由右至左</td><td>… ? … : …</td></tr>
</table>

（续表）

优先级	用途	关联性	运算符
3	赋值运算	由右至左	… = …
			… += …
			… -= …
			… *= …
			… /= …
			… %= …
			… <<= …
			… >>= …
			… >>>= …
			… &= …
			… ^= …
			… |= …
2	返回生成器中的特定数据	由右至左	yield …
	返回子生成器中的特定数据		yield * …
1	扩展运算	无	... …
0	逗号运算	由左至右	… , …

2.3　练　习　题

1. 在 JavaScript 语言里，下列哪些项目是常量？

770、Math.PI、'Nice Day'、"good"、/\w\s\d/g、/.\w.\d/、TRUE、FALSE、Undefined、Null。

2. 在 JavaScript 语言里，应该通过什么语法声明名称为 love_you_forever 而代表着整数值 201314 的常量？

3. 在 JavaScript 语言里，应该通过什么语法声明名称为 love_me_longer 而代表着浮点数值 2591.8 的常量？

4. 在如下 JavaScript 源代码片段里，哪些是**全局**变量？哪些是**局部**变量？

```
let value01 = 10 ;
var value02 = 30 ;

function func01(data, identity)
{
  let result ;

  result = 21 + data + 2 * identity ;
  return result ;
}

var str01 = 'Finished', str02 = 'Error' ;

function func02(amount, price)
```

```
{
  let output ;

  output = (value01 + value02 + price) * amount ;
  return output ;
}
```

5. 在如下 JavaScript 源代码片段里，请**至少**列举出其中的 **5 个**表达式。

```
sphere_volume = r => 4 / 3 * Math.PI * Math.pow(r, 3) ;
```

6. 编写带有上底宽度、下底宽度和高度 3 个参数，可计算并返回**梯形面积**的**函数**定义。
7. 编写带有**长轴**长度、**短轴**长度 2 个参数，可计算并返回**椭圆面积**的**函数**定义。
8. 如下源代码被执行之后，变量 result 的结果数据是什么？

```
let result, n01 = 10, n02 = 20 ;
result = n01++ + ++n02 % 6 ;
```

9. 如下源代码被执行之后，变量 result 的结果数据是什么？

```
let result, n03 = 30, n04 = 40 ;
result = n03-- - --n04 % 6 ;
```

10. 如下源代码被执行之后，变量 result 的结果数据是什么？

```
let result, n05 = 50, n06 = 60 ;
result = (n05 / 5) ** 3 + (n06 / 10) ** 2 ;
```

11. 如下源代码被执行之后，变量 result 的结果数据是什么？

```
let result, n07 = 70, n08 = 80 ;
result = (n07 / 10) ** 2 + (n08 / 20) ** 0.5  ;
```

12. 通过编写条件运算符【? ... :】相关的源代码来实现下述功能。

已知两个变量 score 和 rating。当 score 的数值低于 60 时，变量 rating 的数据为字符串'failed'；当 score 的数值大于或等于 60 而小于 80 时，变量 rating 的数据为字符串'passed'；当 score 的数值大于或等于 80 而小于或等于 100 时，变量 rating 的数据为字符串'nice'；当 score 的数值超过 100 时，变量 rating 的数据为字符串'error'。

13. 至少列举类型运算符 typeof 表达式可返回的 5 种数据类型。

第 3 章

数 据 类 型

不同的编程语言存在不同的数据类型（data type），各种数据类型可用来描述容纳特不同数据的数据结构（data structure）。特定表达式被评估时，可返回特定类型的数据。

3.1　数 值 类 型

JavaScript 语言的数值，不仅包括整数（integer），也包含带有小数部分的浮点数（floating-point number）。

3.1.1　2^n 进制的字面量（ES6）

2^n 进制主要是指二进制、八进制与十六进制。在 JavaScript 语言中，可通过 2^n 进制的字面量（literal），来表示 2^n 进制的**数字常量**（number constant）/ **数值**（number value）。

【3-1-1-nth-power-of-2-based-literals.js】

```
var number01 = 111 ;
var number02 = parseInt('111', 16) ;
var number03 = 0x111 ;
var number04 = 0111 ; // cannot use in strict mode.
var number05 = parseInt('111', 8) ;
var number06 = 0o111 ; // new literal in ES6
var number07 = parseInt('111', 2) ;
var number08 = 0b111 ; // new literal in ES6

console.log(number01) ;
console.log(number02) ;
console.log(number03) ;
```

```
console.log(number04) ;
console.log(number05) ;
console.log(number06) ;
console.log(number07) ;
console.log(number08) ;
```

【相关说明】

```
var number01 = 111 ;
```

111 代表十进制的数值 111。

```
var number02 = parseInt('111', 16) ;
```

parseInt('111', 16) 返回十六进制数值 111 所对应的十进制数值 273。

```
var number03 = 0x111 ;
```

0x111 代表十六进制的数值 111。因此，变量 number03 的初始数值，会成为十六进制数值 111 所对应的十进制数值 273。

```
var number04 = 0111 ; // cannot use in strict mode.
```

0111 代表八进制的数值 111。因此，变量 number04 的初始数值，会成为八进制数值 111 所对应的十进制数值 73。

```
var number05 = parseInt('111', 8) ;
```

parseInt('111', 8) 会返回八进制数值 111 所对应的十进制数值 73。

```
var number06 = 0o111 ; // new literal in ES6
```

0o111 代表八进制的数值 111。因此，变量 number06 的初始数值，会成为八进制数值 111 所对应的十进制数值 73。

```
var number07 = parseInt('111', 2) ;
```

parseInt('111', 2) 会返回二进制数值 111 所对应的十进制数值 7。

```
var number08 = 0b111 ; // new literal in ES6
```

0b111 代表二进制的数值 111。因此，变量 number08 的初始数值，会成为二进制数值 111 所对应的十进制数值 7。

3.1.2 数值的比较（ES6）

要比较两个数值类型的操作数（operand）的数值（number value）是相等、不相等、大于或小于的关系，必须通过带有特定比较运算符的表达式来实现。关于数值的比较，可参考如下示例。

【3-1-2-number-comparisons.js】

```
console.log(3 + 0.1416 === 3.1416) ;
console.log(0.5 + 0.25 === 0.75) ;
console.log('') ;

console.log(0.1 + 0.02 === 0.12) ;
```

```
a = 0.1 ;
b = 0.02 ;
c = 0.12 ;
console.log(a + b === c) ;
console.log(Math.abs(a + b - c) < Number.EPSILON) ;
```

【相关说明】

```
console.log(3 + 0.1416 === 3.1416) ;
console.log(0.5 + 0.25 === 0.75) ;
```

在这两个语句中的比较表达式里，比较运算符【===】两侧操作数的数值，被判断为均相等，以及其数据类型均为浮点数（floating-point number）。

```
console.log('') ;

console.log(0.1 + 0.02 === 0.12) ;
```

在这个语句中，虽然比较运算符【===】两侧的数值，看起来是相等的，但是判断的结果却为假而返回布尔值 false。

计算机的浮点数表示法，无法完整表示大部分的浮点数，所以计算机进行浮点数的运算时，常常会出现小数部分的截断误差（truncation error）。看似简单浮点数的比较运算，对计算机来说，反而是困难的任务。

```
a = 0.1 ;
b = 0.02 ;
c = 0.12 ;
```

这 3 个语句声明了初始数值为不同浮点数的变量 a、b 与 c。

```
console.log(a + b === c) ;
```

虽然比较运算符【===】两侧的数值，看起来是相等的，但是判断的结果却为假而返回布尔值 false。

```
console.log(Math.abs(a + b - c) < Number.EPSILON) ;
```

【Number.EPSILON】是内置对象 Number 中的常量属性 EPSILON，用来表示浮点数运算之后，可被容忍的截断误差值。因此，欲正确比较浮点数，表达式【a + b === c】应该被改写成为【Math.abs(a + b - c) < Number.EPSILON】。

3.1.3 数值的正负符号（ES6）

在 JavaScript 语言中，数值可以是负数、正数或零值，而且零值还可以分为正零和负零。关于数值的正负符号的判断与诠释，可参考如下示例。

【3-1-3-number-sign-deteminations.js】

```
console.log(Math.sign(0)) ;
console.log(Math.sign(-0)) ;

console.log(Math.sign(123)) ;
```

```
console.log(Math.sign(-123)) ;

console.log(Math.sign(null)) ;

console.log(Math.sign(NaN)) ;
console.log(Math.sign('Hello')) ;
console.log(Math.sign(undefined)) ;
```

【相关说明】

函数 Math.sign(特定变量名称)的返回值，可用来判断特定变量的如下特征：

- 其返回值为 0，代表特定变量的数值亦为 0。
- 其返回值为-0，代表特定变量的数值亦为-0。
- 其返回值为 1，代表特定变量的数值为正数，例如 123、23.8、1250.75。
- 其返回值为-1，代表特定变量的数值为负数，例如-123、-29.15、-2500.88。
- 其返回值为 NaN（not a number），代表特定变量的数据，无法被转换成为数值，例如 NaN、undefined、'Hello'、'125a'。

```
console.log(Math.sign(0)) ;
```

Math.sign(0)的返回值为 0。

```
console.log(Math.sign(-0)) ;
```

Math.sign(-0)的返回值为-0。

```
console.log(Math.sign(123)) ;
```

Math.sign(123)的返回值为 1。

```
console.log(Math.sign(-123)) ;
```

Math.sign(-123)的返回值为-1。

```
console.log(Math.sign(null)) ;
```

Math.sign(null)的返回值为 0。

```
console.log(Math.sign(NaN)) ;
```

Math.sign(NaN)的返回值为 NaN。其中，NaN 的含义表示**并非一个数值**（not a number）。

```
console.log(Math.sign('Hello')) ;
```

Math.sign('Hello')的返回值为 NaN。

```
console.log(Math.sign(undefined)) ;
```

Math.sign(undefined)的返回值为 NaN。

3.1.4 数值的截断（ES6）

数值的截断（truncation），是指对于特定浮点数，先进行**无条件**或**四舍五入**的**数值修约**，再

去除其小数，转换成为整数的过程。关于数值的截断的综合运用，可参考如下示例。

【3-1-4-number-truncations.js】

```
v01 = 12.5 ;
v02 = 12.3 ;
v03 = 0.56 ;
v04 = -0.83 ;

console.log(parseInt(v01)) ;
console.log(Math.trunc(v01)) ;
console.log(Math.floor(v01)) ;
console.log(Math.round(v01)) ;
console.log(Math.ceil(v01)) ;
console.log('') ;

console.log(parseInt(v02)) ;
console.log(Math.trunc(v02)) ;
console.log(Math.floor(v02)) ;
console.log(Math.round(v02)) ;
console.log(Math.ceil(v02)) ;
console.log('') ;

console.log(parseInt(v03)) ;
console.log(Math.trunc(v03)) ;
console.log(Math.floor(v03)) ;
console.log(Math.round(v03)) ;
console.log(Math.ceil(v03)) ;
console.log('') ;

console.log(parseInt(v04)) ;
console.log(Math.trunc(v04)) ;
console.log(Math.floor(v04)) ;
console.log(Math.round(v04)) ;
console.log(Math.ceil(v04)) ;
console.log('') ;

console.log(0 === -0) ;
console.log(+0 === -0) ;
```

【相关说明】

```
v01 = 12.5 ;
v02 = 12.3 ;
v03 = 0.56 ;
v04 = -0.83 ;
```

这 4 个语句声明了初始数据为不同浮点数的变量 v01、v02、v03 与 v04。

```
console.log(parseInt(v01)) ;
```

调用内置函数 parseInt(v01)，会返回变量 v01 的数值 12.5 的整数部分 12。

```
console.log(Math.trunc(v01)) ;
```

调用内置函数 Math.trunc(v01)，会返回变量 v01 的数值 12.5 的整数部分 12。

```
console.log(Math.floor(v01)) ;
```

调用内置函数 Math.floor(v01)，会返回**小于**但**接近**变量 v01 的数值 12.5 的整数值 12。

```
console.log(Math.round(v01)) ;
```

调用内置函数 Math.round(v01)，会返回对变量 v01 的数值 12.5 进行**四舍五入**之后的整数值 13。

```
console.log(Math.ceil(v01)) ;
```

调用内置函数 Math.ceil(v01)，会返回**大于**但**接近**变量 v01 数值 12.5 的整数值 13。

```
console.log(parseInt(v02)) ;
```

调用内置函数 parseInt(v02)，会返回变量 v02 的数值 12.3 的整数部分 12。

```
console.log(Math.trunc(v02)) ;
```

调用内置函数 Math.trunc(v02)，会返回变量 v02 的数值 12.3 的整数部分 12。

```
console.log(Math.floor(v02)) ;
```

调用内置函数 Math.floor(v02)，会返回**小于**但**接近**变量 v02 的数值 12.3 的整数值 12。

```
console.log(Math.round(v02)) ;
```

调用内置函数 Math.round(v02)，会返回对变量 v02 的数值 12.3 进行**四舍五入**之后的整数值 12。

```
console.log(Math.ceil(v02)) ;
```

调用内置函数 Math.ceil(v02)，会返回**大于**但**接近**变量 v02 的数值 12.3 的整数值 13。

```
console.log(parseInt(v03)) ;
```

调用内置函数 parseInt(v03)，会返回变量 v03 的数值 0.56 的整数部分 0。

```
console.log(Math.trunc(v03)) ;
```

调用内置函数 Math.trunc(v03)，会返回变量 v03 的数值 0.56 的整数部分 0。

```
console.log(Math.floor(v03)) ;
```

调用内置函数 Math.floor(v03)，会返回**小于**但**接近**变量 v03 的数值 0.56 的整数值 0。

```
console.log(Math.round(v03)) ;
```

调用内置函数 Math.round(v03)，会返回对变量 v03 的数值 0.56 进行**四舍五入**之后的整数值 1。

```
console.log(Math.ceil(v03)) ;
```

调用内置函数 Math.ceil(v03)，会返回**大于**但**接近**变量 v03 的数值 0.56 的整数值 1。

```
console.log(parseInt(v04)) ;
```

调用内置函数 parseInt(v04)，会返回变量 v04 的数值-0.83 的整数部分 0。

```
console.log(Math.trunc(v04)) ;
```

调用内置函数 Math.trunc(v04)，会返回变量 v04 的数值-0.83 的**负**整数部分-0。

```
console.log(Math.floor(v04)) ;
```

调用内置函数 Math.floor(v04)，会返回**小于**但**接近**变量 v04 的数值-0.83 的整数值-1。

```
console.log(Math.round(v04)) ;
```

调用内置函数 Math.round(v04)，会返回对变量 v04 的数值-0.83 进行**四舍五入**之后的整数值-1。

```
console.log(Math.ceil(v04)) ;
```

调用内置函数 Math.ceil(v04)，会返回大于但接近变量 v04 的数值-0.83 的负整数值-0。

```
console.log(0 === -0) ;
```

【0 === -0】返回布尔值 true，所以 0 与 -0 的数值相等、数据类型相同。

```
console.log(+0 === -0) ;
```

【+0 === -0】返回布尔值 true，所以+0 与 -0 的数值相等、数据类型相同。

3.1.5　数值的特殊格式（ECMA-402）

在此，数值的特殊格式，是指千分位（thousands）或者货币（currency）形式的数值表示方式。关于数值的特殊格式的综合运用，可参考如下示例。

【3-1-5-number-formattings.js】

```
var nf_en = Intl.NumberFormat("en") ;
var nf_de = Intl.NumberFormat("de") ;

var number01 = 2533591.8 ;

console.log(nf_en.format(number01)) ;
console.log(nf_de.format(number01)) ;
console.log('') ;

console.log(number01.toString()) ;
console.log(number01.toLocaleString()) ;
console.log('') ;

console.log(number01.toLocaleString('en')) ;
console.log(number01.toLocaleString('de')) ;
console.log('') ;

// currency number format
var cnf_cn = Intl.NumberFormat('cn', {style: 'currency', currency: 'cny'}) ;
var cnf_jp = Intl.NumberFormat('jp', {style: 'currency', currency: 'jpy'}) ;
var cnf_en = Intl.NumberFormat('en', {style: 'currency', currency: 'usd'}) ;
var cnf_uk = Intl.NumberFormat('gb', {style: 'currency', currency: 'gbp'}) ; // Great Britain
 Pound
var cnf_de = Intl.NumberFormat('de', {style: 'currency', currency: 'eur'}) ;

var price01 = 25324700.56 ;

console.log(cnf_cn.format(price01)) ;
console.log(cnf_jp.format(price01)) ;
console.log(cnf_en.format(price01)) ;
```

```
    console.log(cnf_uk.format(price01)) ;
    console.log(cnf_de.format(price01)) ;
```

【相关说明】

```
    var nf_en = Intl.NumberFormat("en") ;
```

Intl.NumberFormat 对象的构造函数 Intl.NumberFormat("en")，会返回【以字符串形式，表示英文（en, English）数值格式（number format）】的 Intl.NumberFormat 对象实例。因此，后续源代码，可借助其数据为 Intl.NumberFormat 对象实例的变量 nf_en，将特定数值，表示成为**英文**数值格式化之后的字符串。

```
    var nf_de = Intl.NumberFormat("de") ;
```

Intl.NumberFormat("de")会返回【以字符串形式，表示德文 / 德国（de, DENIC eG for Germany）数值格式】的 Intl.NumberFormat 对象实例。因此，后续源代码，可借助其数据为 Intl.NumberFormat 对象实例的变量 nf_de，将特定数值，表示成为**德文**数值格式化之后的字符串。

```
    var number01 = 2533591.8 ;
```

此语句声明了初始数据为 2533591.8 的变量 number01。

```
    console.log(nf_en.format(number01)) ;
```

因为变量 nf_en 的数据为 Intl.NumberFormat 对象实例，所以存在可调用的函数 format()。nf_en.format(number01)会返回【将变量 number01 的数值 2533591.8，表示成为**英文**数值格式化】之后的字符串"2,533,591.8"。

```
    console.log(nf_de.format(number01)) ;
```

nf_en.format(number01)会返回【将变量 number01 的数值 2533591.8，表示成为**德文**数值格式化】之后的字符串"2.533.591,8"。

```
    console.log('') ;

    console.log(number01.toString()) ;
```

number01.toString()仅会返回【将变量 number01 的数值 2533591.8，转换成为字符串】之后的"2533591.8"。

```
    console.log(number01.toLocaleString()) ;
```

number01.toString()仅会返回【将变量 number01 的数值 2533591.8，表示成为**本地语言**的数值格式化】之后的"2,533,591.8"。在此，本地语言为中文（cn, China）。

```
    console.log('') ;

    console.log(number01.toLocaleString('en')) ;
```

console.log(number01.toLocaleString('en'))和如下源代码片段之一，存在着异曲同工之妙：

- var nf_en = Intl.NumberFormat("en") ;
  console.log(nf_en.format(number01));
- console.log(Intl.NumberFormat("en").format(number01)) ;

```
    console.log(number01.toLocaleString('de')) ;
```

number01.toLocaleString('de')和如下源代码片段之一，存在着异曲同工之妙：

- var nf_de = Intl.NumberFormat("de") ;
  console.log(nf_de.format(number01)) ;
- console.log(Intl.NumberFormat("de").format(number01)) ;

```
    console.log('') ;

    // currency number format
    var cnf_cn = Intl.NumberFormat('cn', {style: 'currency', currency: 'cny'}) ;
```

Intl.NumberFormat 对象的构造函数 Intl.NumberFormat('cn', {style: 'currency', currency: 'cny'})，会返回【以字符串形式，表示中文 / 中国（cn, China）与人民币（cny, China Yuan）的数值格式】的 Intl.NumberForma 对象实例。后续源代码，可借助其数据为 Intl.NumberFormat 对象实例的变量 cnf_cn，将特定数值，表示成为**中文**与人民币的数值格式化之后的字符串。

```
    var cnf_jp = Intl.NumberFormat('jp', {style: 'currency', currency: 'jpy'}) ;
```

Intl.NumberFormat('jp', {style: 'currency', currency: 'jpy'})，会返回【以字符串形式，表示日文（jp, Japan）与日币（jpy, Japan Yuan）的数值格式】的 Intl.NumberFormat 对象实例。后续源代码，可借助其数据为 Intl.NumberFormat 对象实例的变量 cnf_jp，将特定数值，表示成为**日文**与日币的数值格式化之后的字符串。

```
    var cnf_en = Intl.NumberFormat('en', {style: 'currency', currency: 'usd'}) ;
```

Intl.NumberFormat('en', {style: 'currency', currency: 'usd'})，会返回【以字符串形式，表示英文（en, English）与美元（usd, United States Dollar）的数值格式的 Intl.NumberFormat 对象实例。后续源代码可借助其数据为 Intl.NumberFormat 对象实例的变量 cnf_en，将特定数值，表示成为**英文**与美元的数值格式化之后的字符串。

```
    var cnf_uk = Intl.NumberFormat('gb', {style: 'currency', currency: 'gbp'}) ; // Great Britain
Pound
```

Intl.NumberFormat('gb', {style: 'currency', currency: 'gbp'})，会返回【以字符串形式，表示英式英文（gb, Great Britain）与英镑（gbp, Great Britain Pound）的数值格式】的 Intl.NumberFormat 对象实例。后续源代码可借助其数据为 Intl.NumberFormat 对象实例的变量 cnf_uk，将特定数值，表示成为**英式英文**与英镑的数值格式化之后的字符串。

```
    var cnf_de = Intl.NumberFormat('de', {style: 'currency', currency: 'eur'}) ;
```

Intl.NumberFormat('de', {style: 'currency', currency: 'eur'})会返回【以字符串形式，表示德文 /德国（de, DENIC eG for Germany）与欧元（eur, Euro）的数值格式】的对象实例。后续源代码可借助其数据为 Intl.NumberFormat 对象实例的变量 cnf_de，将特定数值，表示成为**德文**与欧元的数值格式化之后的字符串。

```
    var price01 = 25324700.56 ;
```

此语句声明了初始数据为 25324700.56 的变量 price01。

```
console.log(cnf_cn.format(price01)) ;
```

因为变量 cnf_cn 是 Intl.NumberFormat 对象实例，所以存在可调用的函数 format()。cnf_cn.format(price01)会返回【将变量 price01 的数值 25324700.56，表示成为**人民币**数值格式化】之后的字符串"CN¥25,324,700.56"。

```
console.log(cnf_jp.format(price01)) ;
```

cnf_jp.format(price01)会返回【将变量 price01 的数值 25324700.56，表示成为**日币**数值格式化】之后的字符串"JP¥25,324,701"。

```
console.log(cnf_en.format(price01)) ;
```

cnf_en.format(price01)会返回【将变量 price01 的数值 25324700.56，表示成为**美元**数值格式化】之后的字符串"$25,324,700.56"。

```
console.log(cnf_uk.format(price01)) ;
```

cnf_uk.format(price01)会返回【将变量 price01 的数值 25324700.56，表示成为**英镑**数值格式化】之后的字符串"£25,324,700.56"。

```
console.log(cnf_de.format(price01)) ;
```

cnf_de.format(price01)会返回【将变量 price01 的数值 25324700.56，表示成为**德文**与**欧元**的数值格式化】之后的字符串"£25,324,700.56"。

关于语言代码（language code），例如 cn、en、gb 等，以及货币代码（currency code），例如 cny、usd、gbp 等，可参考如下网址的内容：

- 语言代码: en.wikipedia.org/wiki/Language_code
- 货币代码: en.wikipedia.org/wiki/ISO_4217

3.1.6 整数值的安全范围（ES6）

简单来说，JavaScript 语言的整数值的安全范围是$-(2^{53} - 1) \sim +(2^{53} - 1)$。关于整数值的安全范围的理解，可参考如下示例。

【3-1-6-number-safe-ranges.js】

```
console.log(Number.isNaN(NaN)) ;
console.log(Number.isNaN(123)) ;

console.log(Number.isFinite(456)) ;
console.log(Number.isFinite(NaN)) ;

console.log(Number.isFinite(Infinity)) ;
console.log(Number.isFinite(-Infinity)) ;

console.log('') ;

console.log(Number.isSafeInteger(Math.pow(2, 48) + 100)) ;
console.log(Number.isSafeInteger(Math.pow(2, 53) - 1)) ;
console.log(Number.isSafeInteger(Math.pow(2, 53))) ;
```

【相关说明】

函数 Number.isNaN(特定变量名称)的返回值，可用来判断特定变量的数值，是否并非一个数值（not a number）：

- 其返回值为 true，代表特定变量的数值，并不是一个数值。
- 其返回值为 false，代表特定变量的数值，是一个数值。

```
console.log(Number.isNaN(NaN)) ;
```

Number.isNaN(NaN)的返回值为 true。

```
console.log(Number.isNaN(123)) ;
```

Number.isNaN(123)的返回值为 false。函数 Number.isFinite(特定变量名称)的返回值，可用来判断特定变量的数值，是否为**有限**（finite）的数值：

- 其返回值为 true，代表特定变量的数值，是计算机可明确表示的数值。
- 其返回值为 false，代表特定变量的数值，**并不是**计算机可明确表示的数值。

```
console.log(Number.isFinite(456)) ;
```

Number.isFinite(456)的返回值为 true。

```
console.log(Number.isFinite(NaN)) ;
```

Number.isFinite(NaN)的返回值为 false。

```
console.log(Number.isFinite(Infinity)) ;
```

Number.isFinite(Infinity)的返回值为 false。

```
console.log(Number.isFinite(-Infinity)) ;
```

Number.isFinite(-Infinity)的返回值为 false。

函数 Number.isSafeInteger(特定变量名称)的返回值，可用来判断特定变量的数值，是否为 IEEE-754 所规范的安全范围$-(2^{53} - 1) \sim +(2^{53} - 1)$里的整数值：

- 返回值为 true，代表特定变量的数值，是安全范围内的整数值。
- 返回值为 false，代表特定变量的数值，并不是安全范围内的整数值。

```
console.log(Number.isSafeInteger(Math.pow(2, 48) + 100)) ;
```

Math.pow(2, 48) + 100 的结果值为 $2^{48} + 100$。

Number.isSafeInteger(Math.pow(2, 48) + 100)的返回值为 true。

```
console.log(Number.isSafeInteger(Math.pow(2, 53) - 1)) ;
```

Math.pow(2, 53) - 1 的结果值为 $2^{53} - 1$。

Number.isSafeInteger(Math.pow(2, 53) - 1)的返回值为 true。

```
console.log(Number.isSafeInteger(Math.pow(2, 53))) ;
```

Math.pow(2, 53)的返回值为 2^{53}。

Number.isSafeInteger(Math.pow(2, 53))的返回值为 false。

```
console.log(Math.pow(2, 64)) ;
```

Math.pow(2, 64)的返回值，理应为 2^{64}，也就是 1844674407370955**1616**；但是，其真实的返回值却为 1844674407370955**2000**。所以，其最右侧的 4 位数，明显存在误差。有误差的缘故，就是因为 2^{64} 已经是超越安全范围的整数值了。

3.2 布尔类型

JavaScript 布尔（boolean）类型的数据，只存在布尔值 false（假）和布尔值 true（真）两个组合。其中，false 代表假、否、非、不对、不是等**不成立**的含义；true 则代表真、对、是等**成立**的含义。

【3-2---Boolean-data-type.js】

```
let passed = false, score = 0 ;

score = 80 ;

passed = score > 60 ;

console.log(passed) ;

score = 55 ;

passed = score > 60 ;

console.log(passed) ;
console.log('') ;

///
let conversion ;

conversion = Boolean(-10.8) ;

console.log(conversion) ;

conversion = Boolean(15.6) ;

console.log(conversion) ;
console.log('') ;

///
conversion = Boolean(0) ;

console.log(conversion) ;

conversion = Boolean(null) ;

console.log(conversion) ;
```

```
conversion = Boolean(false) ;

console.log(conversion) ;

conversion = Boolean('') ;

console.log(conversion) ;

conversion = Boolean(undefined) ;

console.log(conversion) ;
console.log('') ;

///
conversion = new Boolean(-10.8) ;

console.log(conversion.valueOf()) ;

conversion = new Boolean(15.6) ;

console.log(conversion.valueOf()) ;

conversion = new Boolean(0) ;

console.log(conversion.valueOf()) ;
```

【相关说明】

```
let passed = false, score = 0 ;
```

此语句声明了初始数据为 false 的变量 passed，以及初始数据为 0 的变量 score。

```
score = 80 ;
```

此语句赋予整数值 80 给变量 score。

```
passed = score > 60 ;
```

在此，【score > 60】是比较运算式，得出来的结果值为布尔值 true，并被赋给变量 passed。

```
console.log(passed) ;

score = 55 ;
```

此语句赋予整数值 55 给变量 score。

```
passed = score > 60 ;
```

此语句将【score > 60】的结果值 false，赋给变量 passed。

```
console.log(passed) ;
console.log('') ;

///
let conversion ;
```

此语句声明了变量 conversion。

```
conversion = Boolean(-10.8) ;
```

-10.8 是一个非 0 的数值，因此，经过 Boolean 对象的构造函数 Boolean()处理之后，会返回 true。

```
console.log(conversion) ;

conversion = Boolean(15.6) ;
```

15.6 是一个非 0 的数值，因此，经过 Boolean 对象的构造函数 Boolean()处理之后，会返回 true。

```
console.log(conversion) ;
console.log('') ;

///
conversion = Boolean(0) ;

conversion = Boolean(null) ;

conversion = Boolean(false) ;

conversion = Boolean('') ;

conversion = Boolean(undefined) ;
```

在前段各语句中，无论是 0、null、false、空字符串''或 undefined，均是具有零值意义的常量，经过构造函数 Boolean()的处理之后，均会返回 false。

```
console.log(conversion) ;
console.log('') ;

conversion = new Boolean(-10.8) ;

conversion = new Boolean(15.6) ;

conversion = new Boolean(0) ;
```

在前段各语句中，构造函数 Boolean()左侧被衔接了关键字 new，使得变量 conversion 的数据，成为内含布尔值 true 或 false 的 Boolean 对象实例。

```
console.log(conversion.valueOf()) ;
```

调用 conversion.valueOf()函数，可返回变量 conversion 内含的布尔值 true 或 false。

3.3 数 组 类 型

数组（array）是用来表示一组【具有**各自数据**】的元素。然而，就 JavaScript 语言来说，其数组实例的数据类型（data type）为**对象**类型。数组实例中各**元素**的数据，可以同时存在**数值**、**布尔值**、**字符串**、**对象**，甚至是子**数组**！关于数组类型的综合运用，可参考如下示例。

【3-3---Array-data-type.js】

```
let numbers = [520, 530, 1314, 2013, 2014] ;
let profile = ['Tommy', 'male', 33, [180, 72]] ;
let newone = new Array(6) ;

console.log(numbers) ;
console.log(numbers[2]) ;

numbers[1] = 2591.8 ;

console.log(numbers) ;
console.log(numbers.length) ;
console.log('') ;

///
console.log(profile) ;
console.log(profile[0]) ;
console.log(profile[2]) ;
console.log(profile[3][0]) ;
console.log(profile[3][1]) ;

profile[3][1] = 70 ;

console.log(profile) ;
console.log(profile.length) ;
console.log(profile[3].length) ;

///
console.log(numbers[numbers.length]) ;
console.log(profile[profile.length]) ;
console.log('') ;
console.log(newone[0]) ;
console.log(newone[newone.length]) ;
```

【相关说明】

```
let numbers = [520, 530, 1314, 2013, 2014] ;
```

此语句声明了初始数据为内含多个**整数元素**的数组实例[520, 530, 1314, 2013, 2014]的变量numbers。

```
let profile = ['Tommy', 'male', 33, [180, 72]] ;
```

此语句声明了初始数据为数组实例 ['Tommy', 'male', 33, [180, 72]] 的变量 profile。

```
let newone = new Array(6) ;
```

new Array(6)会返回内含 6 个空元素的数组实例。此语句声明了初始数据为内含 6 个空元素的数组实例的变量 newone。

```
console.log(numbers) ;
```

在网页浏览器的调试工具【Console】面板中，显示出[520, 530, 1314, 2013, 2014]的信息。

```
console.log(numbers[2]) ;
```

numbers[2]会返回在变量 numbers 的数组实例中，其索引值为 2（第 3 个）的元素值 1314。

```
numbers[1] = 2591.8 ;
```

数值 2591.8 被存为在变量 numbers 数组实例中，其索引值为 1（第 2 个）的元素值。

```
console.log(numbers) ;
```

显示出[520, 2591.8, 1314, 2013, 2014]的信息。

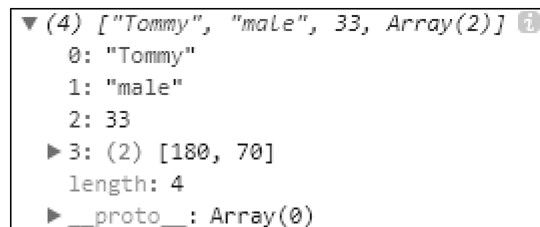
```
console.log(numbers.length) ;
```

numbers.length 会返回变量 numbers 数组实例中的元素个数。

```
console.log(profile) ;
```

在网页浏览器的调试工具【Console】面板中，显示出**可展开**的["Tommy", "male", 33, Array(2)]信息。

展开上述信息之后，可看见 Array(2)代表子数组实例[180, 70]。在此，应该为整数值 72 的，却被显示为如下屏幕快照里的整数值 70，是因为调试工具的运行机制，即时响应了随后源代码的执行结果。




```
console.log(profile[0]) ;
```

profile[0]会返回在变量 profile 的数组实例中，其索引值为 0（第 1 个）的元素的数据"Tommy"。

```
console.log(profile[2]) ;
```

profile[2]会返回在变量 profile 的数组实例中，其索引值为 2（第 3 个）的元素的数值 33。

```
console.log(profile[3][0]) ;
console.log(profile[3][1]) ;
```

profile[3]会返回在变量 profile 的数组实例中，其索引值为 3（第 4 个）的元素的数据，也就是子数组实例[180,72]。

所以，profile[3][0]会返回在子数组实例[180, 72]中，其索引值为 0（第 1 个）的元素的数值 180。

profile[3][1]则会返回在子数组实例[180, 72]中，其索引值为 1（第 2 个）的元素的数值 72。

```
profile[3][1] = 70 ;
```

此语句使得在 profile[3]所代表的子数组实例[180, **72**]中，其索引值为 1（第 2 个）的元素的数值，被修改成为 70。因此，子数组实例变更为[180, **70**]。

```
console.log(profile.length) ;
```

profile.length 会返回在变量 profile 的数组实例中，其**第 1 层**元素的个数 4。

```
console.log(profile[3].length) ;
```

profile[3]代表子数组实例[180, 72]。profile[3].length 会返回子数组实例[180,72]的元素个数 2。

```
console.log(numbers[numbers.length]) ;
```

numbers.length 会返回变量 numbers 的数组实例中的元素个数 5。

在变量 numbers 的数组实例中，若其索引值为 5，则代表第 6 个元素的索引值。

在变量 numbers 的数组实例中，一开始并无第 6 个元素，所以，numbers[numbers.length]如同 numbers[5]的语法，会返回 undefined。

```
console.log(profile[profile.length]) ;
```

profile.length 会返回变量 profile 的数组实例中的元素个数 4。在变量 profile 的数组实例中，若其索引值为 4，则代表第 5 个元素的索引值。在变量 profile 的数组实例中，一开始并无第 5 个元素，所以 profile[profile.length]如同是 profile[4]的语法，会返回 undefined。

```
console.log(newone[0]) ;
console.log(newone[newone.length]) ;
```

因为变量 newone 一开始为内含 6 个空元素的数组实例，所以 newone[0]会返回 undefined，newone.length 会返回变量 newone 的数组实例中的元素个数 6。在变量 newone 的数组实例中，若其索引值为 6，则代表第 7 个元素的索引值。然而，一开始并无第 7 个元素，所以 newone[newone.length]如同是 newone[7]的语法，会返回 undefined。

3.4　对象类型

对象（object）通过属性（property）和方法（method）/函数（function），模拟在真实世界中，具有身份数据（identity data）和表现**特征行为**（characteristic behavior）的物体（object）。关于对象类型的理解，可参考如下示例。

【3-4---Object-data-type.js】

```
// let item01 = {} ;
let item01 = new Object() ;

item01.name = 'Tablet PC' ;
item01.price = 1000 ;
item01.origin = 'China' ;
item01['manufacture date'] = '2018/12/15' ;
item01['color'] = 'RoyalBlue' ;
item01[''] = 'secret data...' ; // empty property name

console.log(item01) ;
console.log(item01['']) ;
console.log(item01.color) ;
console.log(item01['manufacture date']) ;
console.log(item01.price) ;

item01.price = 900 ;
```

```
item01['color'] = 'Gold' ;

console.log(item01) ;
```

【相关说明】

```
// let item01 = {} ;
let item01 = new Object() ;
```

通过关键字 new 和 Object 对象的构造函数 Object()的语法，可返回 Object 对象的空实例（empty instance），或称为空的 Object 对象实例。在此，【= new Object()】等同于【= {}】的语法；而且此语句声明了初始数据为空的 Object 对象实例的变量 item01。

```
item01.name = 'Tablet PC' ;
item01.price = 1000 ;
item01.origin = 'China' ;
```

通过变量 item01 衔接点运算符【.】，可动态创建变量 item01 的 Object 对象实例的 3 个新属性 name、price、origin 及其个别的数据'Table PC'、1000、'China'。

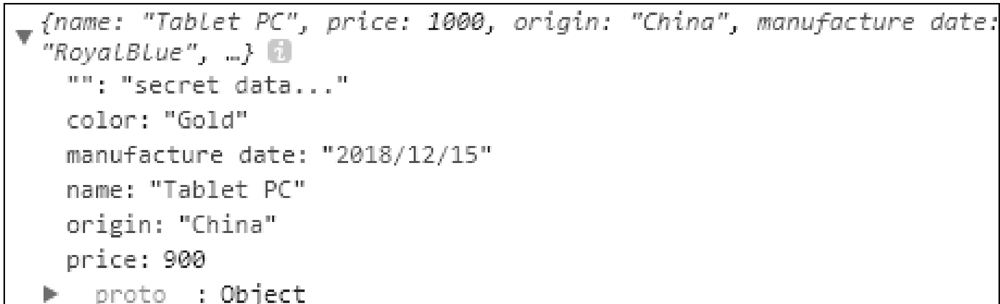
```
item01['manufacture date'] = '2018/12/15' ;
item01['color'] = 'RoyalBlue' ;
item01[''] = 'secret data...' ;  // empty property name
```

通过变量 item01 衔接中括号运算符[]，亦可动态创建变量 item01 的 Object 对象实例的 3 个新属性 manufacture date、color、''，以及其个别的数据'2018/12/15'、'RoyalBlue'、'secret data...'。值得注意的是，和衔接点运算符【.】比起来，衔接中括号运算符[]的方式，可创建具有如下特殊名称的属性：

- 带有空格（space）字符的属性名称，例如【manufacture date】。
- 空（empty）的属性名称，例如【''】。

```
console.log(item01) ;
```

此语句使得网页浏览器在调试工具【Console】面板中，显示出如下信息：




```
console.log(item01['']) ;
```

item01['']返回空属性对应的数据"secret data..."。

```
console.log(item01.color) ;
```

item01.color 返回属性 color 对应的数据"RoyalBlue"。

```
console.log(item01['manufacture date']) ;
```

item01['manufacture date']返回属性 manufacture date 对应的数据"2018/12/15"。

```
console.log(item01.price) ;
```

item01.price 返回属性 price 对应的数值 1000。

```
item01.price = 900 ;
```

此语句使得变量 item01 的属性 price，重新被赋予整数值 900。

```
item01['color'] = 'Gold' ;
```

此语句使得变量 item01 的属性 color，重新被赋予字符串'Gold'。

```
console.log(item01) ;
```

再次通过此语句，查看变量 item01 的数据是否被变更了。

3.5　字符串类型

字符串（string）是具有一串字符（character）的文本数据。在各种应用程序的画面当中，显示给用户观看的文本，即是持续由多个字符串，拼凑而成的产物。

3.5.1　一般字符串

在 JavaScript 语言中，字符串的字面量（literal）均以【内含一串字符】的一对单引号或双引号来表示。关于一般字符串的运用，请看下面的示例。

【3-5-1-common-strings.js】

```
let sentence = 'Hi,\n\tlong time no see!\nhow are you today?\n\nBest Regards,\nAlex' ;

console.log(sentence) ;

sentence = "Limit '3' days and \"5\" persons." ;

console.log(sentence) ;

sentence = '\\It costs \'370\' dollars.\\' ;

console.log(sentence) ;

sentence = 'Alex lovingly loves ' +
'lovely beloved of ' +
'Daisy.' ;

console.log(sentence) ;

sentence = 'Alex lovingly loves \
```

```
lovely beloved of \
Daisy.' ;

console.log(sentence) ;
console.log(sentence[0]) ;
console.log(sentence[1]) ;
console.log(sentence[2]) ;
console.log(sentence[3]) ;
console.log('') ;

console.log(sentence.charAt(0)) ;
console.log(sentence.charAt(1)) ;
console.log(sentence.charAt(2)) ;
console.log(sentence.charAt(3)) ;
```

【相关说明】

```
let sentence = 'Hi,\n\tlong time no see!\nhow are you today?\n\nBest Regards,\nAlex' ;
```

此语句声明了初始数据为字符串的变量 sentence。在此字符串中，存在多个换行（new line）字符'\n'，以及间隔至多如同 8 个空格的制表（tab）字符'\t'。

```
console.log(sentence) ;
```

此语句显示出变量 sentence 的如下数据字符串：

Hi,

        long time no see!

how are you today?

Best Regards,

Alex

```
sentence = "Limit '3' days and \"5\" persons." ;
```

此语句赋予新的字符串字面量，给变量 sentence。在此语句中，通过一对双引号【""】，容纳字符串。然而，在此双引号之内，还存在其他单引号与双引号。

容纳字符串的一对双引号【""】或单引号【''】，不可以冲突于其内部的双引号或单引号！其内部发生冲突的引号左侧，必须衔接反斜杠（back slash）字符才行，例如：

- "...\".....\"....'..'...."
- '..."....."....\'..\'....'

```
console.log(sentence) ;
```

此语句显示出变量 sentence 的数据字符串【Limit '3' days and "5" persons.】。

```
sentence = '\\It costs \'370\' dollars.\\' ;
```

此语句将新的字符串字面量，赋给变量 sentence。在此语句中，通过一对单引号【''】容纳字符串。然而，在单引号之内，还存在反斜杠和其他单引号字符。

在字符串内，单引号、反斜杠均是特殊字符，所以必须在其左侧，衔接额外的反斜杠字符，成为【\\】或【\'】，才能正常呈现在信息里。

```
console.log(sentence) ;
```

此语句显示出变量 sentence 的数据字符串【\It costs '370' dollars.\】。

```
sentence = 'Alex lovingly loves ' +
'lovely beloved of ' +
'Daisy.' ;
```

此语句被分成 3 行，在等号右侧，借助加法运算符【+】，进行 3 个字符串的合并运算。

```
console.log(sentence) ;
```

此语句显示出多个字符串被合并之后的新字符串'Alex lovingly loves lovely beloved of Daisy.'。

```
sentence = 'Alex lovingly loves \
lovely beloved of \
Daisy.' ;
```

此语句亦被分成 3 行，在等号右侧，借助反斜杠字符，串接字符串的各个片段。

欲使用此法，在各个反斜杠字符的右侧，不可再加上包括空格（space）在内的任何字符。

```
console.log(sentence) ;
```

此语句亦显示出多个字符串被合并之后的新字符串'Alex lovingly loves lovely beloved of Daisy.'。

```
console.log(sentence[0]) ;
console.log(sentence[1]) ;
console.log(sentence[2]) ;
console.log(sentence[3]) ;
```

变量 sentence 目前的数据为字符串'Alex lovingly loves lovely beloved of Daisy.'。所以，将变量 sentence 的数据字符串，视为内含多个字符的数组实例时，每个元素的数据，即为各个单一字符。这 4 个语句分别显示出在变量 sentence 的数据字符串里，其第 1 个至第 4 个元素的数据字符'A'、'l'、'e'与'x'。

```
console.log(sentence.charAt(0)) ;
console.log(sentence.charAt(1)) ;
console.log(sentence.charAt(2)) ;
console.log(sentence.charAt(3)) ;
```

这 4 个语句较为正统，和前面 4 个语句有异曲同工之妙，亦可显示出在变量 sentence 的数据字符串里，其第 1 个至第 4 个元素的数据字符'A'、'l'、'e'与'x'。

3.5.2　格式化字符串（ES6）

在 JavaScript 语言中，所谓的格式化字符串（formatting string）的正式名称为模板字面量（template literal），可用来嵌入待评估的表达式。当这些表达式被评估完成而返回特定数据时，格式化字符串才能被确认最终的模样。关于格式化字符串的运用，可参考如下示例。

【3-5-2-string-interpolations.js】

```
var users =
[
  {name: 'John', age: '33', gender: 'male'},
  {name: 'Jessica', age: '27', gender: 'female'},
  {name: 'Daisy', age: '33', gender: 'female'},
  {name: 'Sean', age: '24', gender: 'male'}
] ;

var nations = [ 'China', 'Canada', 'America', 'New Zeland'] ;

var days_amount = 5 ;

var flight_message = `${users[1].name} decides to flight to ${nations[0]} after ${days_amount}
  days.` ;

console.log(flight_message) ;

flight_message = `${users[3].name} decides to flight to ${nations[2]} after ${days_amount}
  days.` ;

console.log(flight_message) ;

var items =
[
  {product_id: 15023, price: 330},
  {product_id: 16002, price: 500}
]

var checkout_message = `This product costs ${items[1].price * 0.8}.` ;

console.log(checkout_message) ;

var string01 = 'Hello\nEarth!' ;
var string02 = `Hello\nEarth!` ;
var string03 = String.raw `Hello\nEarth!` ;

console.log(string01) ;
console.log(string02) ;
console.log(string03) ;

number_digits = '1 2 3 4 5 6 7' ;

// string04 = String.raw({raw: 'a b c d e f g '}, '1',' ','2',' ','3',' ','4',' ','5',' ','6','
  ','7') ;
string04 = String.raw({raw: 'a b c d e f g '}, ... number_digits) ;


console.log(string04) ;
```

【相关说明】

```
var users =
[
  {name: 'John', age: '33', gender: 'male'},
```

```
    {name: 'Jessica', age: '27', gender: 'female'},
    {name: 'Daisy', age: '33', gender: 'female'},
    {name: 'Sean', age: '24', gender: 'male'}
  ] ;
```

此语句声明了初始数据为【内含 4 个对象实例的数组实例】的变量 users。

```
var nations = ['China', 'Canada', 'America', 'New Zeland'] ;
```

此语句声明了初始数据为【内含多个字符串的数组实例】的变量 nations。

```
var days_amount = 5 ;
```

此语句声明了初始数值为整数 5 的变量 days_amount。

```
var flight_message = `${users[1].name} decides to flight to ${nations[0]} after ${days_amount}
  days.` ;
```

此语句声明了初始数据为格式化字符串的变量 flight_message。格式化字符串必须放入一对反引号（back quote）【``】中。在格式化字符串里，可通过${变量名称}或${可评估的表达式}的语法，将特定变量的数据或者表达式被评估之后的数据，放置于格式化字符串里。

```
console.log(flight_message) ;
```

此语句显示出变量 flight_message 的数据，也就是已经被转换完成的字符串'Jessica decides to flight to China after 5 days.'。

```
flight_message = `${users[3].name} decides to flight to ${nations[2]} after ${days_amount}
  days.` ;
```

此语句使得变量 flight_message，被赋予另一个格式化字符串。

```
console.log(flight_message) ;
```

此语句显示出变量 flight_message 的数据，也就是已经被转换完成的字符串'Sean decides to flight to America after 5 days.'。

```
var items =
[
  {product_id: 15023, price: 330},
  {product_id: 16002, price: 500}
]
```

此语句声明了初始数据为【内含对象实例的数组实例】的变量 items。

```
var checkout_message = `This product costs ${items[1].price * 0.8}.` ;
```

此语句声明了初始数据为格式化字符串的变量 checkout_message。

```
console.log(checkout_message) ;
```

此语句显示出变量 checkout_message 的数据，也就是已经被转换完成的字符串'This product costs 400.'。

```
var string01 = 'Hello\nEarth!' ;
```

此语句通过单引号【''】，声明了初始数据为一般字符串的变量 string01。

```
var string02 = `Hello\nEarth!` ;
```

此语句通过反引号【``】，声明了初始数据为格式化字符串的变量 string02。

```
var string03 = String.raw `Hello\nEarth!` ;
```

此语句通过内置的 String 对象的函数 raw()，保留其中各字符的原始编码，并声明了初始数据为原始（raw）字符串的变量 string03。也就是说，在字符串里的【\n】，已经被视为一般的字符【\】与【n】，而不再具有换行（new line）的特征。

值得注意的是，raw()明明是一个函数，在此却不能衔接一对小括号，而是衔接一对反引号，以实现出一个原始字符串。

```
console.log(string01) ;
console.log(string02) ;
```

这两个语句显示的信息相同，具体如下：

Hello
Earth!

```
console.log(string03) ;
```

此语句显示的信息是【Hello\nEarth!】，【\n】的两个字符【\】与【n】均被保留了下来。

```
number_digits = '1 2 3 4 5 6 7' ;
```

此语句声明了初始数据为字符串的变量 number_digits。

```
// string04 = String.raw({raw: 'a b c d e f g '}, '1',' ','2',' ','3',' ','4',' ','5',' ','6','
  ','7') ;
string04 = String.raw({raw: 'a b c d e f g '}, ... number_digits) ;
```

等号右侧的 String.raw(),会返回**交叉分组**之后的字符串"a1 b2 c3 d4 e5 f6 g7 "。在此，如下语法是等价的：

- String.raw({raw: 'a b c d e f g '}, ... number_digits)
- String.raw({raw: 'a b c d e f g '}, '1',' ','2',' ','3',' ','4',' ','5',' ','6',' ','7')

```
console.log(string04) ;
```

此语句显示出变量 string04 的数据，也就是被**交叉分组**之后的字符串"a1 b2 c3 d4 e5 f6 g7 "。

3.5.3 日期与时间格式的字符串（ES6）

内含日期（date）与时间（time）相关数据的字符串，可被称为日期与时间格式的字符串。关于日期与时间格式的字符串的运用，可参考如下示例。

【3-5-3-date-and-time-strings.js】

```
var dt_cn = new Intl.DateTimeFormat('cn') ;
var dt_en = new Intl.DateTimeFormat('en') ;
var dt_de = new Intl.DateTimeFormat('de') ;
```

```
origin_datetime = new Date('2018-01-23') ;

console.log(origin_datetime.toDateString()) ;
console.log(origin_datetime.toLocaleDateString()) ;

console.log('') ;

dt01 = dt_cn.format(origin_datetime) ;
dt02 = dt_en.format(origin_datetime) ;
dt03 = dt_de.format(origin_datetime) ;

console.log(dt01) ;
console.log(dt02) ;
console.log(dt03) ;
```

【相关说明】

```
var dt_cn = new Intl.DateTimeFormat('cn') ;
```

Intl.DateTimeFormat 对象的构造函数 Intl.DateTimeFormat("cn")，会返回中文 / 中国（cn, China）格式的 Intl.DateTimeFormat 对象实例。后续源代码可借助变量 dt_cn 中的 Intl.DateTimeFormat 对象实例，将其他格式的日期与时间，表示成为中文格式的日期与时间。

```
var dt_en = new Intl.DateTimeFormat('en') ;
```

Intl.DateTimeFormat 对象的构造函数 Intl.DateTimeFormat("en")会返回英文（en, English）格式的 Intl.DateTimeFormat 对象实例。后续源代码可借助变量 dt_en 中的 Intl.DateTimeFormat 对象实例，将其他格式的日期与时间，表示成为英文格式的日期与时间。

```
var dt_de = new Intl.DateTimeFormat('de') ;
```

Intl.DateTimeFormat 对象的构造函数 Intl.DateTimeFormat("de")会返回德文 / 德国（de, DENIC eG for Germany）格式的 Intl.DateTimeFormat 对象实例。后续源代码可借助变量 dt_de 中的 Intl.DateTimeFormat 对象实例，将其他格式的日期与时间，表示成为德文格式的日期与时间。

```
origin_datetime = new Date('2018-01-23') ;
```

此语句声明了变量 origin_datetime，并被初始化为【日期是 2018/01/23】的 Date 对象实例。在此，Date('2018-01-23')或 Date('2018/01/23')均可被解读成功。

```
console.log(origin_datetime.toDateString()) ;
```

origin_datetime.toDateString()会返回字符串'Tue Jan 23 2018'。

```
console.log(origin_datetime.toLocaleDateString()) ;
```

origin_datetime.toLocaleDateString()会返回字符串'2018/1/23'。

```
dt01 = dt_cn.format(origin_datetime) ;
```

因为变量 dt_cn 的数据是 Intl.DateTimeFormat 对象实例，所以存在可调用的函数 format()。dt_cn.format(origin_datetime)会返回【在变量 origin_datetime 的 Date 对象实例中，其内含的日期与时间，被表示成为中文格式】之后的字符串'2018/1/23'。

```
dt02 = dt_en.format(origin_datetime) ;
```

因为变量 dt_en 的数据是 Intl.DateTimeFormat 对象实例，所以存在可调用的函数 format()。dt_en.format(origin_datetime)会返回【在变量 origin_datetime 的 Date 对象实例中，其内含的日期与时间，被表示成为英文格式】之后的字符串'1/23/2018'。

```
dt03 = dt_de.format(origin_datetime) ;
```

因为变量 dt_de 的数据是 Intl.DateTimeFormat 对象的实例，所以存在可调用的函数 format()。dt_de.format(origin_datetime)会返回【在变量 origin_datetime 的 Date 对象实例中，其内含的日期与时间，被表示成为德文格式】之后的字符串'23.1.2018'。

3.6 集合与地图类型

对 JavaScript 语言来说，集合（set）与地图（map）主要可用来简化编程的负担，是从 ECMAScript 2015（ES6）版本开始，才具有的数据类型。

3.6.1 集合类型（ES6）

集合（set）内含其数据均**不重复**的元素，和数学理论中的集合，有着非常相似的含义与原理。换句话说，在特定集合内，只存在其数据不相同的元素。关于集合的运用，可参考如下示例。

【3-6-1-Set-data-type.js】

```
let actions = new Set() ;

actions.add('read') ;
actions.add('write').add('update') ;
actions.add('delete') ;

actions.add('read').add('read').add('delete').add('write').add('update') ;

console.log(actions) ;

console.log(actions.entries()) ;
console.log(actions.keys()) ;
console.log(actions.values()) ;
console.log('') ;

for (let element of actions)
{
  console.log(element) ;
}

console.log('') ;
console.log(actions.size) ;

console.log(actions.has('hide')) ;
```

```
console.log(actions.has('write')) ;
```

【相关说明】

```
let actions = new Set() ;
```

此语句声明了初始数据为【空的 Set 对象实例】的变量 actions。

```
actions.add('read') ;
```

当前变量 actions 的数据为一个 Set 对象实例，因此存在可调用的函数 add()。在变量 actions 的 Set 对象实例中，添加数据为字符串'read'的新元素。

```
actions.add('write').add('update') ;
```

此语句使得在变量 actions 的 Set 对象实例中，连续添加数据为字符串'write'与'update'的两个新元素。

```
actions.add('delete') ;
```

此语句使得在变量 actions 的 Set 对象实例中，添加数据为字符串'delete'的新元素。

```
actions.add('read').add('read').add('delete').add('write').add('update') ;
```

在变量 actions 的 Set 对象实例中，连续添加数据为字符串'read'、'read'、'delete'、'write'与'update'的 5 个新元素。在这 5 个新元素里，存在重复的数据字符串；所以，被添加至变量 actions 的 Set 对象实例中的时候，重复的元素会自动被排除在外。

```
console.log(actions) ;
```

此语句显示出变量 actions 的 Set 对象实例【Set(4) {"read", "write", "update", "delete"}】的信息。在此，亦可看出在这个 Set 对象实例中，的确没有数据相同的元素。

```
console.log(actions.entries()) ;
console.log(actions.keys()) ;
console.log(actions.values()) ;
```

这 3 个语句皆显示出相同的信息【SetIterator {"read", "write", "update", "delete"}】。其中，Set Iterator 具有集合迭代器的含义。

```
for (let element of actions)
{
  console.log(element) ;
}
```

通过循环语句 for ... of 的迭代处理，可在每次循环中，显示出在变量 actions 的 Set 对象实例中，其特定元素的如下数据字符串：

- read
- write
- update
- delete

```
console.log(actions.size) ;
```

actions.size 会返回 Set 对象实例中的元素个数 4。

```
console.log(actions.has('hide')) ;
```

actions.has('hide')会返回 false，意味着在 Set 对象实例中，并无数据字符串为'hide'的元素。

```
console.log(actions.has('write')) ;
```

actions.has('write')会返回 true，代表着在 Set 对象实例中，存在数据字符串为'write'的元素。

3.6.2 地图类型（ES6）

地图（map）内含【**键名（key name）**对应到**值（value）/ 数据（data）**】的组合。地图与对象（object）极为相似，只是节省了对象的累赘和限制。关于地图的运用，可参考如下示例。

【3-6-2-Map-data-type.js】

```
let items = new Map() ;

items.set('slipper', 50) ;
items.set('shoes', 200) ;
items.set('pants', 100).set('shirt', 150) ;

console.log(items) ;

console.log(items.size) ;

console.log(items.entries()) ;
console.log(items.keys()) ;
console.log(items.values()) ;

for (let [product, price] of items)
{
  console.log(`One ${product} costs ${price}.`) ;
}
```

【相关说明】

```
let items = new Map() ;
```

此语句声明了初始数据为【空的 Map 对象实例】的变量 items。

```
items.set('slipper', 50) ;
```

变量 items 的数据，当前为一个 Map 对象实例，因此存在可调用的函数 set()。此语句使得在变量 items 的 Map 对象实例中，被添加键名为 slipper、值为 50 的新元素。

```
items.set('shoes', 200) ;
```

此语句使得在变量 items 的 Map 对象实例中，被添加键名为 shoes、值为 200 的新元素。

```
items.set('pants', 100).set('shirt', 150) ;
```

此语句使得在变量 items 的 Map 对象实例中，连续添加键名分别为 pants 与 shirt、值分别为 100 与 150 的两个新元素。

```
console.log(items) ;
```

此语句显示出在变量items的Map对象实例【Map(4) {"slipper" => 50, "shoes" => 200, "pants" => 100, "shirt" => 150}】的信息。

```
console.log(items.size) ;
```

items.size 会返回 Map 对象实例中的元素个数 4。

```
console.log(items.entries()) ;
```

此语句显示出 MapIterator {"slipper" => 50, "shoes" => 200, "pants" => 100, "shirt" => 150}的信息。其中，Map Iterator 具有地图迭代器的含义。

```
console.log(items.keys()) ;
```

此语句显示出 MapIterator {"slipper", "shoes", "pants", "shirt"}的信息。

```
console.log(items.values()) ;
```

显示出 MapIterator {50, 200, 100, 150}的信息。

```
for (let [product, price] of items)
{
  console.log(`One ${product} costs ${price}.`) ;
}
```

通过此循环语句 for ... of 的迭代处理，可在每次循环中，显示出在变量 items 的 Set 对象实例中，特定元素的如下数据字符串：

- One slipper costs 50.
- One shoes costs 200.
- One pants costs 100.
- One shirt costs 150.

值得注意的是，在上述循环语句的小括号里，【let [product, price] of items】的语句，使得在每次循环中：

- 变量 product 的数据，成为在变量 items 的 Map 对象实例中，特定元素的键名，例如 slipper。
- 变量 price 的数据，成为在变量 items 的 Map 对象实例中，特定元素的值，例如 50。

3.7　数据类型的转换（ES6）

对于各编程语言来说，特定变量在不同的时间点上，可被赋予不同类型的数据！在 JavaScript 引擎中，通过自动转换机制或者一些内置函数，可使得特定变量的数据，从原来的数据类型，转换成为新的数据类型。

最常见的数据类型的转换，莫过于【其他数据→字符串】和【字符串→数值】。关于数据类型的转换的综合运用，可参考如下示例。

【3-7---data-type-conversions.js】

```
let digital_string = '  123  ' ;
result = Number(digital_string) ;

console.log(result) ;

digital_string = '  0o123' ;
result = Number(digital_string) ;

console.log(result) ;

digital_string = '0x123  ' ;
result = Number(digital_string) ;

console.log(result) ;

result = parseInt(digital_string) ;

console.log(result) ;

result = Math.round(digital_string) ;

console.log(result) ;
console.log('') ;

///
digital_string = '35.62' ;
result = Math.floor(digital_string) ;

console.log(result) ;

digital_string = '28.2' ;
result = Math.ceil(digital_string) ;

console.log(result) ;

digital_string = '12.5' ;
result = Math.round(digital_string) ;

console.log(result) ;

///
let value = 53.8125 ;
result = value.toString(2) ;

console.log(result) ;

result = value.toString(8) ;

console.log(result) ;

result = value.toString(16) ;
```

```
console.log(result) ;
```

【相关说明】

```
let digital_string = '  123  ' ;
```

此语句声明了初始数据为字符串'　123　'的变量 digital_string。

```
result = Number(digital_string) ;
```

通过内置的 Number 对象的构造函数 Number()，将变量 digital_string 的数据字符串'　123 '，转换成十进制整数值 123。在转换过程中，所有空格（space）字符皆会被过滤掉。

```
digital_string = '  0o123' ;
```

将字符串'　0o123'，赋给变量 digital_string。

```
result = Number(digital_string) ;
```

通过内置的 Number 对象的构造函数 Number()，将变量 digital_string 的数据字符串'　0o123'，先视为成八进制整数码 123，再转换成为等价的十进制整数值 83。

```
digital_string = '0x123  ' ;
```

将字符串'0x123　'赋给变量 digital_string。

```
result = Number(digital_string) ;
```

通过内置的 Number 对象的构造函数 Number()，将变量 digital_string 的数据字符串'0x123　'，先视为十六进制整数码 123，再转换成为等价的十进制整数值 291。

```
result = parseInt(digital_string) ;
result = Math.round(digital_string) ;
```

这两个语句皆可达成【将**字符串**'0x123　'，转换成为等价的十进制**整数值** 291】的任务。

```
digital_string = '35.62' ;
result = Math.floor(digital_string) ;
```

这两个语句将字符串'35.62'，转换成为十进制整数值 35。

```
digital_string = '28.2' ;
result = Math.ceil(digital_string) ;
```

这两个语句将字符串'28.2'，转换成为十进制整数值 29。

```
digital_string = '12.5' ;
result = Math.round(digital_string) ;
```

这两个语句将字符串'12.5'，转换成为十进制整数值 13。

```
let value = 53.8125 ;
```

此语句声明了初始数值为 53.8125 的变量 value。

```
result = value.toString(2) ;
```

value.toString(2)会返回转换之后的二进制数码 110101.1101。

```
result = value.toString(8) ;
```

value.toString(8)会返回转换后的八进制数码 65.64。

```
result = value.toString(16) ;
```

value.toString(16)会返回转换后的十六进制数码 35.d。

3.8 练 习 题

1. 在 JavaScript 语言里，应该使用什么语句，才能将浮点数 25.75，分别表示成为二进制、八进制和十六进制数码的字符串？

2. 在 JavaScript 语言里，应该使用什么语句，才能将二进制数码 110111011，**直接**转换成为十六进制数码？

3. 在 JavaScript 语言里，应该使用什么语句，才能直接将二进制数码 101100011100、八进制数码 1275、十六进制数码 51cf 的总和，转换成**二进制**数码？

4. 已知变量 x 与 y，请将 $3x^2 + 2(x-1)^2y + 2xy^2 + 5y^3$，编写成为 JavaScript 语言中的**算术表达式**。

5. 已知有如下主要表达式：

```
let product = {item01: ['fruit_set', 100], item02: ['sticker_set', 250], item03: ['magnet_set',
  350], item04: ['drink_set', 150], item05: ['pizza_set', 300]} ;
```

请编写【将上述**各个整数值**的**总和，**赋给变量 result】的语句。

6. 说明原始常量 NaN、Infinity 和 undefined 的含义。

7. 执行如下 JavaScript 源代码片段之后，变量 result 的结果值是什么？

```
let num01 = 28.56, num02 = 32.47 ;
let result ;

result = parseInt(num01) + Math.trunc(num01) + Math.floor(num01) + Math.round(num01) +
  Math.ceil(num01) ;

result += parseInt(num02) + Math.trunc(num02) + Math.floor(num02) + Math.round(num02) +
  Math.ceil(num02) ;
```

8. 请编写 JavaScript 源代码，使得变量 price 的数值 72583000，显示成为**货币数值格式**的**字符串**' CN￥72,583,000.00'。

9. 列举两种 JavaScript 语法，来计算并显示【$\sqrt[3]{x}$, where $x = 768$】的数值。

10. 列举两个等价于【price >= 300 && amount < 10】的**比较与逻辑**表达式。

11. 已知如下数组实例：

```
let arr = [[[5, 10], [15, 20]], [[25, 30], [35, 40]], [[45, 50], [55, 60]]] ;
```

访问上述整数值 25、40 和 60 的语法是什么？

12. 已知如下对象实例：

```
let obj = {product: {en: 'browser', cn: '浏览器' }, developer: {en: 'Google', cn: '谷歌'}, price:
  {en: 'free', cn: '免费'}} ;
```

访问字符串'browser'、'谷歌'与'free'的语法是什么？

13. 已知数据为对象实例的变量 profile 和如下主要表达式：

```
let message = profile.name + ' now lives on ' + profile.planet + '.' ;
```

让变量 message 被设置为相同数据字符串的**等价语法**是什么？

14. 欲显示如下**分行**的信息：

```
Apple: 11
Banana: 15
Guava: 23
```

其较为简短的 JavaScript 语法应该是什么？

15. 欲显示当前的日期与时间，其较为简短的 JavaScript 语法应该是什么？

16. 欲声明其数据为如下**集合**实例的变量 components：

```
Set(5) {"window", "pane", "dialogue", "button", "scrollbar"}
```

其较为简短的 JavaScript 语法应该是什么？

17. 欲声明其数据为如下**地图**实例的变量 devices：

```
Map(4) {"mobile phone" => 10, "tablet PC" => 7, "notebook PC" => 3, "desktop PC" => 20}
```

其较为简短的 JavaScript 源代码应该是什么？

18. 已知变量 num 的数值为十进制整数值 201314，请编写【带有**置入变量名称**的模板字面量，并且显示如下信息】的 JavaScript 源代码：

```
变量 num 的：
  十进制数值 = 201314
  二进制数码 = 110001001001100010
  八进制数码 = 611142
  十六进制数码 = 31262
```