第5章

JavaScript 基础

JavaScript 最早源自网景(Netscape)公司,1996 年,网景将 JavaScript 提交给 ECMA(前身为 European Computer Manufactures Association)进行标准化,1997 年 6 月,ECMA 以 JavaScript 语言为基础制定了 ECMAScript 标准规范 ECMA-262。

与 HTML 和 CSS 不同,前两者是 W3C 的推荐标准,而 JavaScript 是 ECMAScript 标准规范的实现。除 JavaScript 外,其他语言也实现了 ECMAScript 标准,如微软早期的 JScript,但因 JavaScript 最为著名,因此通常提到 ECMAScript 指的就是 JavaScript。

最初,JavaScript 更多是作为一种运行于浏览器中的小脚本语言,为程序员提供与浏览器交互的能力,但如今它已走出了浏览器的局限,也可运行于非浏览器环境,其应用领域远远超出了最初的范围。2015 年 6 月,ECMAScript 第 6 版发布(常简写作 ES6 或 ES2015),在此版本中有许多重大更新。目前多数情形下,若不特别声明,程序员口中的 JavaScript 指的即是 ES6 之后的版本,本书内容基于 ES6 讲解,也加入了一些 ES6 之后版本的内容。目前,ECMAScript 是一个相对活跃的标准,自 2015 年起,直至本书出版时,每年 ECMA 均发布了新的 ECMAScript 标准,目前通常使用年份作为版本号,如 2022 年 6 月正式发布的版本为 ECMAScript 2022。读者可随时关注新版本的变化,若运行环境支持,不妨大胆使用新的语法特性。

JavaScript 虽然在名称上与 Java 相似,但应注意 JavaScript 与 Java 语言有本质上的区别。其一,JavaScript 是一种解释型编程语言,简单地说,JavaScript 的代码会在程序运行时被 JavaScript 解释器(引擎)逐句解释为机器代码执行,Java 则是一种编译型的语言,它的源代码需要经编译器预编译才可执行。其二,JavaScript 是一种动态语言,同时也是一种弱类型语言,而 Java 是静态的强类型语言。当然,就程序设计语言本身来说,并无本质上的优劣之分,关键看应用场景。编译型语言因其预先编译往往带来更高的执行效率,静态语言可帮助程序员发现很多数据类型兼容问题。

如果读者有学习 C/C++/Java 的经验,会看到 JavaScript 的许多基本语法与它们相似,通常把这类语言称作“类 C 语言”。因此,本书并不打算按部就班地介绍 JavaScript,多数时候会引导读者换一个角度思考,从需求出发,从程序设计的理念来学习和理解。当然,即使读者从未接触过任何程序设计语言也不必担心,

本书将由浅入深逐步展开，配合精心准备的示例，让读者学有所得。

图灵奖获得者 Niklaus Emil Wirth 曾写过一本书 *Algorithms＋Data Structures＝Programs*《算法＋数据结构＝程序》，这句话也成了计算机科学的名言。我们至少可从这句话中读出以下两个信息。

- 计算机程序处理的核心内容是数据，因此首先要解决的问题即是如何合理地表达和存储数据。
- 计算机程序按照一定的算法对数据进行处理，最终输出结果。

因为不同类型的数据在计算机内部有着不同的表达、存储、运算方式，因此发展出了“数据类型”的概念。计算机程序设计语言中，一般都包括如下基本数据类型。

(1) 数值型：如 1、－2、3.14，因内部存储机制的不同，数值型又被细分为定点数(integer，整数)和浮点数(float，小数)。根据存储空间大小的不同，以及是否表达负数，又可再细分为更多子类型。

(2) 字符型：如字母、汉字，狭义地指单个的字符(char)，广义地包含字符串(string)。

(3) 布尔型：只表达真(true)或假(false)。

为了使用方便，许多程序设计语言中还会支持更高级的数据类型，但通常由以上基本数据类型来表达。

如果数据项之间没有关系，则称作离散型数据，而进入计算机世界的数据之间往往是有关联的，为表达这种数据之间的关系，发展出了“数据结构”的概念，即定义一定的存储结构来表达数据之间的关系，同时定义针对不同数据类型和数据结构的运算/操作规则。

读到这里，读者也许会有这样的疑问：这与我们要学习的知识有何关系？如果读者有学习任何一门程序设计语言的经验，不妨将教材翻开对照一下，通常第一部分介绍的就是以上内容在具体程序设计语言中的实现：数据类型、变量、常量、数组、运算符……若是面向对象语言，也许会涉及一些常用的高级数据类型，如 String、Date，以及它们所支持的操作。这些概念其实就可归入上述三个层次：数据类型、数据结构、运算/操作规则。

计算机算法通过将多条语句按一定规则组合在一起以实现特定的功能，这便涉及程序设计语言中的控制语句。当需要解决的问题变得复杂时，为便于管理和复用，同时保持程序的可维护性，将相对独立的功能模块进行封装便形成了函数。而在面向对象程序设计语言中，又更进一步地将数据和功能逻辑封装为一个整体，即对象。

本章内容将围绕上述主题逐层展开，以下是总体规划，希望读者在学习技术细节的同时，也能理解这些技术所要实现的宏观目标。

5.1 节：介绍常用的数据类型、数据结构和声明方式。

5.2 节：深入讨论数据的存储方式。

5.3 节：介绍基本运算与操作规则。

5.4 节：介绍控制逻辑的表达，即控制语句。

5.5 节：讨论功能逻辑的封装，即函数的定义和使用。

5.6 节：进一步探讨数据与功能逻辑的封装，即对象和类。

视频讲解

5.1 数据类型与数据声明

5.1.1 基本数据类型

基本数据类型也称原始数据类型,其值被称作**原始值**。JavaScript 的基本数据类型包括:number、bigint、string、boolean、null、undefined 和 symbol。

1. number

JavaScript 中,无论是整数还是小数均按 number 类型处理,始终使用 IEEE754 标准中的 64 位双精度浮点数来表示。

- 表达范围:$\pm 1.797\ 693\ 134\ 862\ 315\ 7\times 10^{308}$之间。
- 最小值:$\pm 5\times 10^{-324}$,可表达的最接近 0 的数。

对于整数,可安全表达的范围为 $\pm(2^{53}-1)$,超出此范围则不能精确表达,此时可使用 bigint 类型。

整数的字面量[①]可使用十进制、八进制(以 0 开头)或十六进制(以 0x 或 0X 开头)表示。例如,十进制数 14 可写作 14(十进制)、016(八进制)或 0xE(十六进制)。浮点数的字面量可直接使用十进制数表示,或使用科学记数法,例如,3.14、0.314e1、0.314E1。

以下为常用的 number 类型特殊值/常量。

- NaN:Not a Number,即从数据类型看该值属于 number 类型,但并非合法的数值。例如,执行 "A" * 2 的结果即为 NaN。
- Infinity/-Infinity:正/负无穷。
- Number.MAX_VALUE:$1.797\ 693\ 134\ 862\ 315\ 7\times 10^{308}$(number 类型最大值)。
- Number.MIN_VALUE:5×10^{-324},最接近 0 的正数。
- Number.MIN_SAFE_INTEGER:$-2^{53}+1$,可安全表达的最小整数。
- Number.MAX_SAFE_INTEGER:$+2^{53}-1$,可安全表达的最大整数。
- Number.NEGATIVE_INFINITY:-Infinity,负无穷。
- Number.POSITIVE_INFINITY:Infinity,正无穷。

2. bigint

bigint 是 ES2020 新加入的基本数据类型,可以表达任意大小的整数,且不限制其所占用的字节数。bigint 字面量以字母 n 结尾,例如,3n、-5n。适用于 number 类型的运算大部分也适用于 bigint 类型,但注意表达式中不可混用 number 和 bigint,例如,3+3n 是错误的表达式,应做显式类型转换,如 3+number(3n)或 bigint(3)+3n。

3. string

JavaScript 中无论单个字符还是多个字符(字符串)都按 string 类型处理。string 类型的字面量必须使用单引号(')或双引号(")为定界符。

字符串字面量中若出现特殊字符,应使用反斜杠(\)进行转义,例如,\n(换行,New Line)、\r(回车,Carriage Return)、\'(单引号)、\" (双引号)。也可使用 \xXX 或 \uXXXX

① 字面量(literal,直接量)即程序中直接书写的值,如 18、3.14。

指定 Latin-1 或 Unicode 字符，其中，“X”为十六进制数值，如 \u000D(回车符)。

在 ES6 中增加了反引号(`)可用于定义字符串模板字面量，此时特殊字符无须转义，也可用于定义多行字符串。下面的代码展示了 He's often called "Johnny" 这样一个句子，分别使用单引号、双引号、反引号定义字面量的写法。

```
He\'s often called "Johnny"'
"He's often called \"Johnny\""
`He's often called "Johnny"`
```

4. boolean

布尔型(boolean)数据的取值只能为 true(真)或 false(假)，称作布尔值或逻辑数。

在布尔上下文[①]中，false、0、－0、0n(bigint 类型的 0)、""(空字符串)、null、undefined、NaN 均被认定为 false，这些值被统称作 **falsy**(假值)，其余情况均认定为 true，统称作 **truthy**(真值)。

5. null

null 表示无效的对象或地址引用。null 和 undefined 有时被统称为 nullish。

6. undefined

undefined 表示未赋初始值的变量，或未获得值的形式参数(形参)。

7. symbol

ES6 中新增的 symbol 类型用于表示唯一的、不可更改的值，使用 Symbol()函数创建。常可用作对象属性的键(key)或数据对象的唯一标识(id)。

5.1.2　数据声明

1. 变量

程序中若需要空间存储数据，可使用如下代码进行声明(declare)。

```
let x
```

以上代码声明了一个**变量**(variable)，x 为变量名。此时变量 x 并没有具体的值，它的值即为 undefined。之所以称作“变量”，是因为其值可以被更改。

作为弱类型的动态程序设计语言，JavaScript 中声明变量时无须明确数据类型，而变量具体的数据类型视其值而定。可看到如下代码中，随着程序逐行向下执行时变量 x 的值在变化，其数据类型也随之改变。

```
let x=3                 //x 值为 3, number 类型
x='Hello'               //x 值变为'Hello', string 类型
x=true                  //x 值变为 true, boolean 类型
```

代码中的等号(=)意为**赋值**，即将等号右边的值赋给左边的变量，在声明变量时可以直接为其赋值(如上述代码第 1 行)。

“//…”为注释，若需要注释多行内容可使用“/*…*/”。此外，注意 JavaScript 是一种区分大小写的语言。

在 ES6 之前的版本中，声明变量使用关键词 var，例如，var x= 3，但 ES6 之后更建议使

① 布尔上下文(boolean context)：可理解为程序中需要使用布尔表达式的位置，如 if (…) 的“( )”内。

用 let,5.4.1 节将详细对比它们的区别。

接下来学习如何将 JavaScript 代码嵌入网页。请创建如下所示的 HTML 文件,并将 JavaScript 代码写入<script>…</script>标签内,浏览器加载页面时便会自动执行。

code-5.1.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>5.1</title>
    <script>
        let x=3
        console.log(x)                  //>>>3          (输出 x 的值)
        console.log(typeof x)           //>>>number     (typeof 取得变量 x 的数据类型)

        x='Hello'
        console.log(x)                  //>>>Hello
        console.log(typeof x)           //>>>string

        x=true
        console.log(x)                  //>>>true
        console.log(typeof x)           //>>>boolean
    </script>
</head>
<body>
    <p>请打开"开发者工具"切换至"Console"标签页</p>
</body>
</html>
```

以上代码中 console.log()用于向控制台输出信息,为便于理解,注释中以“>>>”表示程序在此行向控制台实际输出的内容。

在浏览器中打开 code-5.1.html,并切换至开发者工具的 Console 标签页(控制台),便可看如图 5.1 所示的程序运行结果。

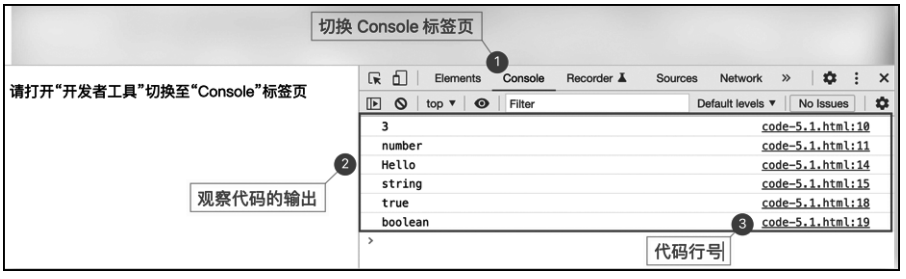

图 5.1 在开发者工具中查看控制台输出

更佳的做法是将 JavaScript 代码移至独立的 js 文件中(jscode.js),并使用<script>标签链入,代码如下。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <!--./jscode.js 为外部 js 文件的相对路径  -->
```

```
    <script src="./jscode.js"></script>
</head>
<body></body>
</html>
```

为节省篇幅，若不特别说明，下文中的 JavaScript 代码请读者自行创建上述 HTML 文档框架进行实验。为便于阅读，多数情况下本书将 HTML、CSS、JavaScript 代码都写在一个 HTML 文件中，实践中建议将三者分离。

2. 常量

在一些情况下，为避免变量的值被更改，可将其声明为常量(constant)。

对于常量，一旦赋值则不可以重新赋值。例如，下面的代码使用关键词 const 定义了常量 c 并将其赋值为 3，此后若再对常量 c 进行赋值程序将报错。

```
const c=3
c=4   //程序执行至此行将报错: Uncaught TypeError: Assignment to constant variable
```

因为常量不可重新赋值，所以声明常量时必须为其赋予初始值，如上述第 1 行代码。

常量与变量的区别仅在于其操作规则的限制(是否可重新赋值)，与数据类型以及数据的表达、存储方式无关。若无特别说明，本书关于变量的描述也同样适合于常量。

在程序设计语言中，**标识符**(identifier)用于为变量、变量、函数等命名，如前面示例中的 x、c。JavaScript 标识符命名规则与其他程序设计语言类似，可以由字母、数字、下画线(_)和 $ 组成，但第一个字符不可以是数字，此外应避免使用语言自身的保留字(关键字)作为标识符，如 let、const、if 等。这里所说的字母和数字包括 Unicode 字符集中的所有字符，因此也可使用汉字，但不建议这样做。

JavaScript 程序中变量名一般使用小驼峰命名风格，即第一个单词首字母小写，其余单词首字母大写，如 userName、lastModified；常量名一般使用全大写的蛇形命名风格，如 USER_KEY。

若读者有使用 C/C++/Java 等程序设计语言的经验，可能习惯于在每条语句末尾添加分号(;)，但 JavaScript 中，若语句置于不同行，则语句末尾的分号可以省略，当然有也无妨，这仅是风格问题，看团队要求或个人习惯而定。

多条 let/const 语句可合作一行，中间使用逗号分隔，例如：

```
let x=1, y, z=3.14
const c=3, d=4
```

5.1.3　常用引用类型

除 5.1.1 节介绍的基本数据类型外，JavaScript 中其余类型均属于**对象类型**，也称作**引用类型**，其值被称作**引用值**。我们将在 5.2 节进一步探讨基本数据类型与引用类型的区别，本节介绍常用的几种引用类型。

1. 数组

数组(Array)用于封装有序的成组元素。数组字面量中使用中括号括起 0 个或任意多个组成数组的元素，其间使用逗号分隔，数组元素可以是任何数据类型，其个数称作数组长度，可通过访问数组对象的 length 属性获得。与其他强类型语言不同，JavaScript 数组中各

元素的数据类型可以不相同,请参看如下代码。

```
const arr0=[]                          //空数组,没有任何元素,长度为 0
const arr1=[1, 2, 3]                   //3 个元素均为 number 类型
const arr2=['zhangsan', true, 60.5] //3 个元素依次为 string, boolean, number 类型
```

当需要访问数组中的某个元素时,可使用元素的索引值(index,序号)来找到它,例如:

```
const arr=['A', 'B', 'C']
let a1=arr[1]                  //程序运行至此,变量 a1 赋值为 'B'
arr[2]='X'                     //程序运行至此,arr 为 ['A', 'B', 'X']
arr[3]='Y'                     //程序运行至此,arr 为 ['A', 'B', 'X', 'Y'],
console.log(arr.length)        //>>>4(当前数组长度为 4)
arr=['D']                      //程序报错:arr 为常量,不可重新赋值
```

从上面几行代码至少可获得如下知识。

- 数组元素的索引值是从 0 开始的正整数,即例子中数组元素 'A'的索引值为 0。
- JavaScript 中的数组长度是可变的。
- 即便数组 arr 本身是常量,也可以改变数组中元素的值,但不可对 arr 本身重新赋值。

2. 对象

与数组不同,对象(Object)使用键值对(key-value pair)的形式封装数据,例如:

```
const boy={              //男孩对象
  name: '张三',          //姓名属性
  height: 1.7,           //身高属性
  weight: 65             //体重属性
}
```

上述 boy 即为对象,其封装了一个男孩的基本信息。大括号"{}"括住的部分描述了该对象特征:姓名、身高和体重。这些所谓的特征被称作对象的**属性**(property),其中,name、heigh、weight 为**属性名**,'张三'、1.7、65 为**属性值**,属性名与属性值之间使用冒号分隔,属性之间则使用逗号分隔。属性可以是任意数据类型,当然也包括对象类型。

不仅如此,还可以为对象 boy 增加一些功能逻辑,例如,计算并返回这个男孩的 BMI 值(体质指数=体重除以身高的平方)。请尝试运行如下代码,并观察其输出。

```
const boy={
  name: '张三',
  height: 1.7,
  weight: 65,
  //定义"方法",计算该男孩的 BMI 值
  getBMI() {
    //this 指代"这个男孩",this.weight 即这个男孩的体重
    return this.weight / (this.height * this.height)
  }
}

console.log( boy.name )          //>>>张三
console.log( boy.getBMI() )      //>>>22.49134948096886
```

上述代码中,getBMI() {…} 被称作"**方法**"(method),它封装了计算 BMI 的功能,getBMI 为方法名,方法名之后小括号"()"中的内容称作参数表(上例为空),再之后的{…}

部分称作方法体，是具体功能的实现。在对象的方法体中可使用 this 指代当前对象，因而 this.weight 即为当前男孩对象的体重，此处等价于 boy.weight。方法体中的 return 语句用于将计算结果返回。代码的最后两行分别通过访问 boy 对象的 name 属性和调用其 getBMI() 方法取得该男孩的姓名和 BMI 数值。

综上所述，对象使用“属性”对数据进行封装，而使用“方法”对功能进行封装，因此对象是数据和功能的封装体。

方法如果脱离了对象即是函数（5.5 节介绍），在许多概念和语法上方法与函数是相通的，JavaScript 中也可以使用如下方式定义方法。

```
const boy={
  ...
  getBMI: function() {…}
}
```

这样看起来方法更像是对象的一个属性，只是该属性的值是一个函数。

在其他一些面向对象程序设计语言中（如 C++/Java 等），若要构建对象必须先定义类（class），然后再创建该类的实例，即对象。而从上面的例子可以看到，JavaScript 允许直接使用对象字面量定义对象，而跳过定义类的步骤。对象与类是面向对象程序设计语言中一个重要的话题，将在 5.6 节重点讨论。

3. Set 和 Map

Set 和 Map 是 ES6 新增的数据类型，它们在一定程度上像是数组和普通对象的补充，Set 用于存储一组唯一的元素，而 Map 以键值对的形式存储数据（以前通常使用对象）。

下面的例子演示了 Set 和 Map 的简单用法。

```
//此处基于数组构建 set 对象，也可不带参数，使用 new Set()语句构建空的 set 对象
const set=new Set([1, 2])
set.add(3)                              //向 set 对象中添加新元素
console.log(set)                        //>>>{1, 2, 3}

//此处基于数组构建 map 对象，也可不带参数，使用 new Map()语句构建空的 map 对象
const map=new Map([['name', 'Johnny'], ['weight': 65]])
map.set('height', '1.7')                //设置 height 属性为 1.7
console.log(map.get('name'))            //>>>Johnny(取得 name 属性值)
console.log(map.get('height'))          //>>>1.7(取得 height 属性值)
```

与数组不同，Set 中的元素是唯一的，因此可利用其进行去重操作。

对象的键（属性名）只可使用 string 或 symbol 类型，而 Map 的键可以使用任意类型，除此之外，在进行遍历、计数等操作时 Map 有一定优势，因此若只是为封装数据，可使用 Map 替代普通对象。

5.2　基本类型与引用类型

5.1 节中基本建立了数据类型的概念，也初步认识了一些常用的数据结构。在继续学习之前，有必要深入地讨论一下前文划分基本数据类型（基本类型、原始类型）与引用类型（对象类型）的依据到底是什么？明白这些原理将有助于掌握后续内容。

本节涉及一些暂未介绍的概念,若读者感到晦涩难懂可略读,但本节内容值得精读,后续章节中会适时指引读者返回本节学习。

程序设计语言中,基本类型所占用的空间大小是固定的。例如,JavaScript 中 number 类型的数值在内存中始终占用 8B(64 个二进制位),无论是存储一个 1 还是 2^{10},并且数值被直接存储于变量所指向的内存单元。

但程序中也免不了存储一些占用空间大小不固定的数据,如数组、对象等,它们所占用的内存空间大小可能因其包含的元素/属性的数量而不同,并且有可能随时变化。对于这些占用空间大小不固定的数据类型,其值不便直接存储于变量所指向的内存单元,而是被存储于动态分配的内存空间内,变量指向的内存单元中存放的却是值所在的动态空间地址。一些程序设计语言中将内存地址称作"指针"(如 C/C++语言),而 JavaScript 中称作"引用",相应地,此类数据被称作引用类型数据。图 5.2 展示了基本类型与引用类型数据的不同存储方式。

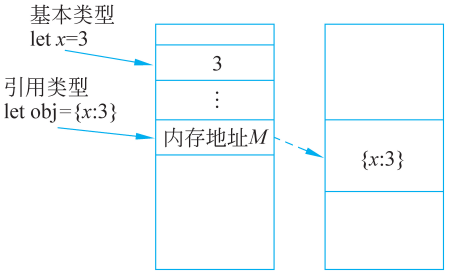

图 5.2 基本类型与引用类型数据的不同存储方式

赋值操作时,无论是基本类型或是引用类型,复制的都是变量所指向的内存单元内的值,做等值比较时,也是比较该值是否相等(即图 5.2 中左侧空间中的值)。请参看如下程序。

```
let x=3                        //基本类型 number
let obj={ x: 3 }               //引用类型 object

let x2=x                       //将 x 赋值给 x2
let obj2=obj                   //将 obj 赋值给 obj2

x2=4
obj2.x=4

console.log(x)                 //>>>3     (x 的值未发生变化)
console.log(obj.x)             //>>>4     (obj.x 的值变成了 4)

console.log(x==x2)             //>>>false  (二者指向的内存单元内的值不相等)
console.log(obj==obj2)         //>>>true   (二者指向的内存单元中存有相同的内存地址)
```

观察上述代码的运行结果可看到,虽然 x2= x、obj2= obj 两行赋值语句让 x2 和 obj2 复制了原变量的值,但对于引用类型复制的只是指向对象实际存储空间的内存地址,因而 obj2 和 obj 通过相同的地址间接地引用了同一对象,此后对 obj2.x 属性值的更改其实也是在更改 obj.x 属性值,最后 1 行的等值比较结果也证明了这一点。图 5.3 呈现了上述代码执行结束时内存中的状态。

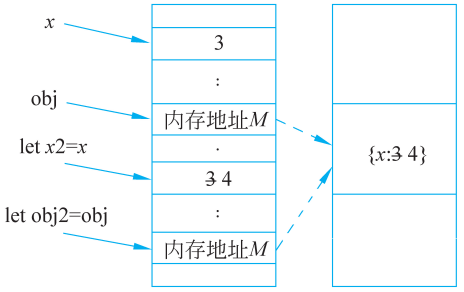

图 5.3　基本类型与引用类型赋值操作示意

在函数调用时，同样会因传递的参数类型不同（基本类型/引用类型），导致函数内部对值的更改所产生的影响也不同，请参看如下代码。

```
let x=3
let obj={ x: 3 }

function foo(x2, obj2) {        //函数内分别对 x2 和 obj2.x 赋值
  x2=4
  obj2.x=4
}

foo(x, obj)                     //调用函数 foo,并将 x 和 obj 作为实参传入

console.log(x)                  //>>>3  (foo 函数内对 x2 的更改未影响到函数外的 x)
console.log(obj.x)              //>>>4  (foo 函数内对 obj2.x 的更改同样影响到 obj.x)
```

从此段代码的运行结果可看出，作为参数传递时，引用类型因为传递的是内存地址，所以函数内的形参 obj2 事实上与函数外的实参 obj 引用了同一对象，因此对 obj2.x 的更改同样影响了 obj.x 的值。基本类型变量作为参数传递时，传递的是变量值，因此 x2 与 x 是完全独立的复本，函数内对 x2 的更改并未影响函数外 x 的值。原理与前一示例类似。简而言之，作为参数传递时，基本类型**传值**，而引用类型**传引用**（传址）。

下面的例子中同样涉及基本类型与引用类型数据的复制问题，请读者尝试使用前面所述的原理解释程序的输出结果。

```
let arr=[3, {x : 3}]
let arr2=[arr[0], arr[1]]       //将数组 arr 中的两个元素复制到 arr2

arr2[0]=4                       //arr2[0]重新赋值对 arr[0]无影响
arr2[1].x=4                     //arr2[1].x 重新赋值同时也改变了 arr[1].x 的值

console.log(arr[0])             //>>>3
console.log(arr[1])             //>>>{x: 4}
```

在上述数组元素的复制过程中，arr2[1] 和 arr[1] 事实上指向了同一个对象（仅复制了内存地址），因此对 arr2[1] 的操作其实也是在操作 arr[1]。

通常将类似上例这样的复制过程称作浅拷贝，换言之，若在赋值、复制、参数传递等过程中仅复制了对象的引用（地址），并未产生新对象，则称作**浅拷贝**，相对地，若产生了新的对象复本则称作**深拷贝**。有时这两个术语也用作名词，例如上例中将 arr2[1] 称作 arr[1] 的浅