第 3 章

运　算　符

编程语言的发明有一个重要原因是为了代替人工进行复杂的数据计算，而为了进行这些计算，需要使用各种各样的运算符(Operator)进行操作。在 JavaScript 中，除了有熟知的数学、赋值、比较和逻辑等运算符外，还有其他许多专门为编程提供便利的运算符，例如位运算符，利用多种运算符可以编写更复杂的表达式。在运算符中参与运算的表达式一般称为操作数(Operand)，按照操作数数量的不同，运算符又分为一元运算符、二元运算符等。操作数在 JavaScript 中也是表达式，所以在本章中，这两种名词会交替使用。

当同时使用多个运算符时，不同的运算符还有不同的计算顺序，有的是从左到右进行计算，有的则是从右到左进行计算。另外，运算符的优先级也不相同，例如乘法和除法的优先级比加法和减法的优先级高，逻辑与和逻辑或运算符优先级比数学运算符低，这些都是在实际开发中要注意的，本章在结束的时候也会列出优先级表供查阅，接下来先介绍不同运算符的作用和用法。

3.1　赋值运算符

赋值运算符使用＝表示，在之前的章节已经多次使用过它了，给变量、数组元素、对象的属性等赋值时都使用了赋值运算符，它所做的操作是把右边操作数的值赋给左边的操作数，代码如下：

```
let a = 10;              //把 10 赋值给 a,a 是左侧表达式
let obj = {};
obj.prop = "value";      //把"value"值赋给 obj 对象的 prop 属性, obj 是左侧表达式
```

赋值运算符的计算顺序是从右向左的，如果使用了多个赋值运算符，则会从最右边开始，把值依次赋给左侧表达式。这里需要注意的是，因为只有左侧表达式才能被赋值，所以在使用多个赋值运算符时，只有最右边的可以是任意表达式，中间及最左边的所有表达式必须为能够被赋值的左侧表达式，代码如下：

```
let b = 10;
let a = b = 5;              //b = 5,a = b,先把 b 的值改成 5,再把 b 的值赋给 a
//Uncaught SyntaxError: Invalid left-hand side in assignment
let c = 6 = 7;              //6 不是左侧表达式,不能被赋值
```

3.2 数学运算符

数学运算符,顾名思义,就是用于进行数学计算的,跟数学课中所学的用法基本保持一致,不过有一些数学运算符有特殊的用法。JavaScript 中的数学运算符有+、-、*、/、%、++、--这几种,按照操作数数量的不同,有些运算符既可以是一元运算符,又可以是二元运算符,这里按操作数的数量分别看一下数学运算符的用法。

3.2.1 一元数学运算符

这类运算符包括+、-、++和-- 4 种,可用于单一操作数。

+既可以是一元运算符,也可以是二元的,这里看一下它作为一元运算符时的作用。一元+表示返回一个数字的正数表示,从数学课中可以知道,一个数字的正数还是它本身,所以+一般可以省略,代码如下:

```
+5;                 //5
+-5;                //-5
+1.23;              //1.23
0;                  //0
```

一元+有一个使用技巧,可以把非数字类型的值转换为数字,作为快速显式类型转换的方式之一,代码如下:

```
+"7";               //7
+true;              //1
```

-表示的含义是取一个数字的相反数,即正变负,负变正,用法跟一元+一样,代码如下:

```
-6;                 //-6
-(-7);              //7,因为 -- 是自减运算符,所以 -7 需要用括号
-25.6;              //-25.6
-0;                 //-0
```

++自增,表示对数值自身进行加 1,它的操作数不能是字面值,而应为变量、对象的属性、数组中的元素等,并且操作数必须能够转换成数字类型。另外,++既可以放在操作数

的前方，也可以放在后方，前者称为前缀＋＋，后者称为后缀＋＋，它们的区别是，前缀＋＋会先把数值加 1，再返回加 1 后的结果，后缀则是先返回当前的数值，再对数值加 1，代码如下：

```
let i = 0;
i++;                    //先返回 0,i 再变成 1
++i;                    //直接返回 i + 1 后的值,此处为 2
```

－－自减，表示对数值减 1，规则和使用方法与＋＋一致，代码如下：

```
let i = 10;
i--;                        //10
--i;                        //8
```

＋＋和－－经常在循环中使用，对计数变量进行加 1 或减 1 操作，当满足一定条件的时候循环就会停止，这个在后边讲到循环的时候再详细介绍。

3.2.2 二元数学运算符

二元数学运算符有＋、－、*、/、%、**（ES2020），分别代表加、减、乘、除、取模（求余数）和幂运算。

1. 加法

加法首先可以用来对数字操作数求和，代码如下：

```
1 + 2;                  //3
0.2 + 0.5;              //0.7
```

加法除了用于数字相加，还能用于字符串拼接，它会把进行加法操作的字符串合并成一个，例如"hello"＋"world"会返回"hello world"。如果将数字和字符串相加，则会先把数字转换为字符串，然后进行拼接，例如 1＋"2"会返回"12"。

对于其他类型遵循以下规则：

（1）如果数字与布尔类型相加，则会把 true 转换为 1，把 false 转换为 0。

（2）如果与 null 相加，则会把 null 转换成 0。

（3）如果与 undefined 相加，则会把 undefined 转换为 NaN，任何数字与 NaN 进行计算都会返回 NaN。

（4）如果与对象相加，对象实现了 valueOf()方法，并且返回数字，则会使用它进行相加，否则会把对象转换为字符串然后进行拼接。

（5）如果与数组相加，则会把数字转换为字符串，然后进行拼接。

来看一些例子，代码如下：

```
//chapter3/addition1.js
5 + true;                          //6
5 + null;                          //5
5 + undefined;                     //NaN
5 + {};                            //"5[object Object]"
5 + {valueOf() {return 10}};       //15
5 + [];                            //"5"
```

2. 减法

减法使用－符号。减法及后面要介绍的数学运算符不会把操作数转换为字符串(字符串只能使用＋进行拼接),而是都转换为数字类型,不能转换的则直接返回 NaN。下边的例子展示了减法运算符的用法,代码如下:

```
//chapter3/substraction1.js
10 - 1;                  //9
10 - true;               //9
10 - null;               //10
10 - "5";                //5
10 -  - 5;               //15,含义为 10 减去 - 5,两个减号中间需要有空格,否则会被认为是自减
10 - "str";              //NaN
```

3. 乘法

乘法使用 * 符号,对两个操作数进行乘法运算,代码如下:

```
3 * 2;                  //6
5 * "6";                //30
5 * "a";                //NaN
```

4. 除法

除法使用/符号,与 Java 等编程语言不同的是,JavaScript 的除法得到的商并不是取整数部分,而是包含完整的浮点数部分,代码如下:

```
6 / 3;                  //2 整除
3 / 2;                  //1.5 包括小数
```

如果除数为 0,则任何数与 0 相除会得到 Infinity,代码如下:

```
1 / 0;                  //Infinity
3.5 / 0;                //Infinity
```

需要注意的是,Number 类型因为都有正数和负数表示法,0 也不例外,虽然＋0 和－0 是相同的,但是在作为除数时,任何数与－0 相除都会得到－Infinity,代码如下:

```
-10 / 0;                    // - Infinity
-2.33 / 0;                  // - Infinity
```

5. 取模

取模与除法类似，只是它的结果是余数，使用%符号，代码如下：

```
10 % 3;                     //1,商为 3,余数为 1
10 % 2;                     //0
-9 % 2;                     // - 1
```

取模也可以用在浮点数上，商是整数部分，余数则包括剩余的部分，代码如下：

```
10.25 % 3;                    //1.25
```

利用取模运算可以把取值限定在一个范围内，例如访问数组中的元素，可以对数组的长度进行取模运算，这样就不会超过数组的最大索引了，代码如下：

```
let arr = [1, 2, 3];
let i = 0;
arr[(i++) % arr.length];              //1,其中 i = 0
arr[(i++) % arr.length];              //2,其中 i = 1
arr[(i++) % arr.length];              //3,其中 i = 2
arr[(i++) % arr.length];              //1,其中 i = 0
```

取模运算的用途非常广泛，例如在 Diffie-Hellman 密钥交换中就使用了取模运算，关于算法的内容已经超出本书所讨论的范畴，读者可自行研究。

6. 幂运算

幂运算是在 ES2020 推出的新语法，使用 ** 符号表示，左边为底数，右边为指数（幂），然后对底数进行幂运算，代码如下：

```
3 ** 2;                     //9
6 ** 3;                     //216
(-3) ** 3;                  // - 27
(-4) ** 2;                  //16
10 **  -2;                  //0.01
5 ** 0;                     //1
4 ** 0.5;                   //2
```

可以看到幂运算与所学到的数学中的幂运算的规则是一样的。

（1）如果底数为负数，并且幂为奇数，则结果仍然是负数。

（2）如果底数为负数，并且幂为偶数，则结果为正数。

(3) 如果幂为负数,则把幂当作正数,然后对底数进行幂运算,最后取结果的倒数。

(4) 如果幂为小数,则进行开方运算。

这里需要注意的是,如果底数为负数,需要使用小括号,否则会引起歧义,不知究竟应把底数作为负数还是应把结果转换为负数,同时 JavaScript 也会给出错误提示。

3.2.3　计算顺序与优先级

数学运算符的计算顺序一般是从左到右的,即在有多个操作数的情况下,会先从左到右两两进行计算,最后得出结果,但是幂运算除外,它是自右向左进行计算的,代码如下:

```
2 ** 3 ** 2;                    //结果是 512,而不是 64
```

如果一个表达式中有多种数学计算,则不同的运算符有不同的优先级,优先级高的先进行计算,优先级低的后进行计算,相同优先级的,则按运算符的计算顺序依次进行计算,数学运算符的优先级从高到低分别为:

(1) ++(自增)和--(自减)。

(2) **(幂运算)。

(3) *(乘)、/(除)、%(取模)。

(4) +(加)、-(减)。

例如在下方代码中,首先 a 进行自增,得到 3,3 乘以 5 得到 15。接着,计算 4 ** 2 得到 16,18 对 16 取模,得到 2。最后,15 加上 2 得到 17,代码如下:

```
let a = 2;
++a * 5 + 18 % 4 ** 2;              //17
```

不过这样的代码看起来非常难以理解,这时可以使用小括号()来对表达式进行分组,括号内部的表达式的优先级最高,如果有嵌套的括号,则从里向外进行计算,如果是同级的括号,则按计算顺序进行计算。这样上边的表达式可以写成使用小括号的形式,并且运算结果也是相同的,代码如下:

```
let a = 2;
((++a) * 5) + (18 % (4 ** 2));              //17
```

3.3　比较运算符

比较运算符用来判断两个操作数之间的关系,所以又称为关系运算符。根据操作数关系的成立与否,比较运算符会返回布尔类型的结果:true 代表关系成立,false 代表关系不成立。比较运算符既可以用于变量之间也可以用于字面值之间的比较,常用的比较运算符有

>、<、>=、<=、==、===、!=、!==，下边分别看一下它们的用法和注意事项。

>、<、>=、<=最常见的用法是对数字进行比较，即左边的操作数是否大于、小于、大于或等于、小于或等于右边的操作数，如果成立，则返回 true，如果不成立，则返回 false，代码如下：

```
//chapter3/comparison1.js
5 > 6;                  //false
let a = 10;
a >= 10;                //true
a < 4;                  //false
a < true;               //false,这里 true 被转换成了数字 1,10 不大于 1,所以返回 false
99999999n < 100000      //false, BigInt 和 Number 类型可以进行混合比较
```

如果操作数为 boolean、null、undefined 或字符串类型，则会先把它们转换为数字再进行比较：boolean 类型的 true 和 false 分别会被转换为 1 和 0，null 会被转换为 0，undefined 会被转换为 NaN，任何数字与 NaN 比较都会返回 false。至于字符串类型，如果字符串中的内容是数字则会转换为数字，非数字则会转换为 NaN，代码如下：

```
5 > "3";                //true
5 > "a";                //false
5 <= "a";               //false
```

如果操作数两边同时为字符串类型，则会按字符串比较规则进行比较。由于字符串底层是使用 UTF-16 编码进行表示的，实际上是对 UTF-16 代码点逐位进行比较，且每个代码点在 Unicode 字符中都有不同的顺序，所以可以用于比较大小。字符串比较的方式如下：

(1) 如果第 1 个字符串的第 1 个字符和第 2 个字符串的第 1 个字符不同，则直接把这两个字符比较的结果作为整体结果返回。

(2) 如果相同，则比较第二位，以此类推，直到有不同的字符。

(3) 若每位都相同，且两个字符串长度也相同，则它们是相等的。

(4) 若每位都相同，但是字符串长度不同，则长度大的比长度小的大。

来看一些例子，代码如下：

```
"a" < "b";              //true, a 为 97, b 为 98
"abcd" < "abbb";        //false,长度相同,c 比 b 大,所以"abcd" 应大于 "abbb"
"abcd" < "ab";          //false, ab 字符都相同,长者为大,所以"abcd"应大于 "ab"
```

==、===是用来比较两个操作数的值是否相等的，如果两个操作数都是基本类型，且内容相同，则这两个运算符都会返回 true，否则会返回 false，代码如下：

```
//chapter3/comparison2.js
10 == 10;                    //true
12 === 12;                   //true
let a = 5, b = 5;
a == b;                      //true
a === b;                     //true
```

它们两个虽然用法是一样的，但是对于比较的方式有一定的区别。==是松散的比较方式，在比较的过程中，如果两边的操作数类型不同，会先把类型转换为相同的之后再进行比较。使用==比较两个值时，发生的类型转换可能与预想的不一样，遇到这种情况需要注意，代码如下：

```
null == undefined;           //true
1 == true;                   //true
0 == false;                  //true
1 == "1";                    //true
```

这里可以看到，null 和 undefined 被认为是相等的，当布尔类型和字符串类型分别与数字类型进行比较时，会先把布尔类型或字符串类型转换为数字类型再进行比较。对于数字类型的比较还要注意一点：比较 NaN，无论是使用==还是使用===都会返回 false。

===是严格的比较方式，因此也称为严格相等(Strict Equality)，它不会发生类型转换，只有类型和值完全相同时才返回 true，其他情况无论是值还是类型不同，都会返回 false。为了减少 Bug，建议全部使用===，除非有特别的业务需要再使用==，代码如下：

```
//chapter3/comparison3.js
null === undefined;          //false
1 === true;                  //false
0 === false;                 //false
1 === "1";                   //false
let a = 10, b = 10;
a === b;                     //true
```

上边的例子是对基本数据类型进行比较。如果比较的是对象类型，例如数组、对象、函数等，则比较的是它们的引用。对象类型在创建的时候，会返回它们在内存地址中的引用，例如比较两个对象时，是比较保存了对象引用的变量是否指向了同一块内存地址。如果两个对象的内容相同，但是指向的内存地址不同，则它们是不同的对象。例如使用==和===判断两个对象是否相同，代码如下：

```
//chapter3/comparison4.js
let a = {x : 1}, b = {x : 1};
a === b;                     //false
```

```
a == b;                                    //false
let c = a;
a === c;                                   //true
a == c;                                    //true
```

!=和 !==的结果与==和===的相反,其余的用法一样。

3.4 逻辑运算符

逻辑运算符可以对表达式进行与(&&)、或(||)、非(!)运算。一般地,进行与运算时,只有所有表达式结果都为 true 时才成立。进行或运算时,只要有一个表达式结果为 true 就会成立。非运算则是对自身进行取反,为真时返回假,为假时返回真。一般逻辑运算符会跟比较运算符结合使用,来形成更复杂的逻辑判断表达式,代码如下:

```
//chapter3/logical1.js
let min = 10;
let max = 100;
let value = 25;
value >= min && value <= max;              //true
value = 125;
value >= min || value <= max;              //true
!(value >= min);                           //false
```

不过在 JavaScript 中,逻辑运算符不严格要求两边的表达式返回布尔类型的 true 和 false,而是任何表达式均可以进行逻辑运算,在运算过程中,是对表达式逻辑意义上的真值(Truthy Value)和假值(Falsy Value)进行计算。转换规则见表 2-2。简单来讲,0、NaN、null、undefined、""、false 都是逻辑意义上的假值,其他都是真值,例如!6 会返回 false,!null 会返回 true。

JavaScript 的逻辑与、逻辑或与其他编程语言(如 Java)中的不同,它的结果并非总是布尔类型的值,代码如下:

```
let value = null;
value && 3;                                //null
```

这段代码涉及了几个问题。代码中 value 的值为 null,在逻辑意义上,也就是当转换为布尔类型时,它是 false,而 3 在逻辑意义上是 true,false 和 true 进行与的计算结果应该是 false,但是这里结果是 null,为什么呢?

这是因为,逻辑运算符返回的并不是布尔类型的值,而是返回最后一个参与计算的表达式的值。另外,逻辑与、逻辑或有短路(Short-Circuit)特性,当进行逻辑与计算时,如果遇到一个逻辑意义上的假值,则这时整个逻辑表达式的结果就确定为不成立了,所以就会直接返

回假值表达式的执行结果。

代码中 value && 3 这个运算在计算 value 的值时得到了 null，因为是假值，所以逻辑与的结果已经能确定为假了，之后它会返回 value 的计算结果：null，所以 value && 3 直接就返回了 null。同时，使用逻辑或把上述表达式改为 value||3，则会返回 3，这是因为在计算 value 的值时，得到的是假值，但程序仍然需要判断第 2 个表达式是否为真值才能得出最终结果，所以会返回最后一个表达式的值。

利用这些特性，可以使用逻辑与来避免因为使用未定义的值而报错，代码如下：

```
let obj;
obj.a;                          //类型错误:不能访问 undefined 中的 a 属性
obj && obj.a;                   //无报错,返回 undefined
```

还可以使用或运算来给一些空值设置默认值，代码如下：

```
let data = null;
data || "无数据";               //"无数据"
```

非运算会把结果转换为布尔类型，如果是真值则返回 false，如果是假值则返回 true，代码如下：

```
!false;                         //true
!"";                            //true
!3;                             //false
```

如果使用两个非运算符，则可以把任何值转换为 boolean 类型，即将真值转换为 true，将假值转换为 false，代码如下：

```
!!10;                           //true
!!"";                           //false
```

3.5 Nullish Coalescing 运算符

Nullish Coalescing(空值合并)运算符是在 ES2020 规范中定义的，使用??表示。它的含义与使用逻辑或设置默认值类似，只是在进行运算时，只有当左侧操作数为 null 或 undefined 的时候，才会使用右侧操作数的值，否则返回左侧操作数的值，代码如下：

```
"" ?? 10;                       //""
false ?? "no";                  //false
null ?? 100;                    //100
undefined ?? "empty";           //"empty"
```

Nullish Coalescing 运算符的应用场景可以是：把""、false、0 等值当作正常的值，只有遇到 null 和 undefined 时，才设置默认值。

3.6 三目运算符

三目运算符(Tenary Operator)是一个需要 3 个操作数的运算符，它可以根据条件的成立与否执行不同的表达式。

三目运算符使用 ？：表示，问号前边是要判断的条件，问号与冒号中间是条件成立时要执行的表达式，冒号后边是条件不成立时，要执行的表达式。这里的条件判断与逻辑运算符一样，会使用逻辑意义上的真、假值进行判断，代码如下：

```
let show = false;
show ? "visible" : "hidden";          //hidden,show 为 false,所以执行冒号后边的表达式
let a = 10;
a ? a + 5 : 0;                        //15, a 为 10,是真值,所以会执行冒号前边的表达式
```

三目运算符也可以进行嵌套，也就是说冒号前后的表达式也可以是另一个三目运算符，不过不建议这样编写代码，因为这样会影响代码的阅读，代码如下：

```
let a = 10;
a > 5 ? a > 8 ? a + 2 : a + 4 : a + 6;          //12
```

这里可以看到，要找出对应的条件表达式很难。这个表达式所进行的计算是：

如果 a 大于 5，会执行 a＞8？ a＋2：a＋4 这个表达式，否则执行 a＋6，这里的 a 是大于 5 的，所以会执行第 1 个表达式，第 1 个表达式又是一个三目运算，判断如果 a 大于 8，则 a 就加 2，否则 a 加 4。这里的 a 是大于 8 的，所以最后的结果是 a 加 2，即 12。

在 JavaScript 开发中，尤其是前端开发，很容易在代码里使用嵌套的三目运算符，但是这样非常难以阅读，所以推荐使用 if...else 判断条件并返回相应的结果。三目运算符可以看作简化版的 if...else 语句，第 4 章将会讲到它。

3.7 位运算符

位运算符是用来操作数字的二进制位的，在一般的编程需求里很少用到，但是在数据结构和算法中，还是经常会遇到，所以这里简单地介绍一下它们的用法。

位运算的操作数是 32 位的整数，而不是 64 位的浮点数，如果使用浮点数，则它会把小数部分截掉，然后把它转换成 32 位之后再进行计算。位运算符有与(&)、或(|)、异或(^)、取反(～)、左移(<<)、右移(>>)、补 0 右移(>>>)这几种。与、或、异或、取反这几种操作跟逻辑运算符类似，只是它们比较的是 1 和 0，然后返回逐位运算的结果。

3.7.1 与运算

与运算的计算方法是：当两个操作数位都是 1 时，返回 1，其他情况返回 0。例如计算 3&5，会分别先把 3 和 5 转换成二进制形式，二进制位数不同的，在长度短的一方的前边用 0 补齐，例如下方展示了 3&5 的计算过程，代码如下：

```
011                    //3
101                    //5
----------
001                    //1
```

这样，进行与运算后的结果为 001，即十进制的 1。另外，在进行与运算时，任何数字跟它本身做与运算，返回的都是它自己，跟 0 进行与运算则都返回 0，代码如下：

```
10 & 10;               //10
999 & 999;             //999
10 & 0;                //0
```

3.7.2 或运算

或运算的计算方法是，只有当两个操作数位都是 0 时才返回 0，其余情况返回 1。例如计算 3|5 的过程，代码如下：

```
011                    //3
101                    //5
----------
111                    //7
```

在进行或运算时，任何数字跟它本身和 0 进行运算时，都会返回数字本身，代码如下：

```
10 | 10;               //10
10 | 0;                //10
-5 | -5;               //-5
-5 | 0;                //-5
```

3.7.3 异或运算

异或运算的计算方法是，当两个操作数位不相同时，返回 1，相同则返回 0。例如计算 3^5 的过程如下：

```
011                 //3
101                 //5
 ----------
110                 //6
```

3.7.4 取反运算

取反运算会把一个操作数的所有二进制位进行取反操作，1 变为 0，0 变为 1。这样得出来的结果是数字本身加一之后再取相反数，代码如下：

```
~3;                  //-4, 即 -(3+1)
~-9;                 //8, 即 -(-9+1)
```

3.7.5 左移运算

左移会把二进制位向左移动若干位，右边用 0 补足空白位置。例如 2 << 3 结果为 16，计算过程如下：

```
00010;                  //2
//把 1 左移 3 位
10000;                  //16
```

如果左移后，有二进制位超过了 32 位，则超出部分将直接被舍弃，例如计算 25 << 30 的结果为 1073741824，计算过程如下：

```
0000 0000 0000 0000 0000 0000 0001 1001             //25
0100 0000 0000 0000 0000 0000 0000 0000             //1073741824
```

25 的二进制位表示为 00011001，在左移 30 位之后，只剩下最后一个 1 还在有效范围内，所以结果会变成上方计算过程所示。需要注意的是，左移的位数只能取 0～31 的数字，如果大于 31，则会取对 32 进行取模之后得到的余数，例如左移 35 位会变为左移 3 位。

3.7.6 右移运算

右移会进行与左移相反的运算，并且超过 32 位的部分也会被舍弃。在移动时，左侧空位会复制最左侧符号位的数值，如果是负数则为 1，如果是正数则为 0。例如计算 -5 >> 1 的结果为 -3，计算过程如下：

```
1111 1111 1111 1111 1111 1111 1111 1011       //-5,最左侧为符号位,-5 的二进制表示其实
//是对 4 的二进制表示进行取反运算
1111 1111 1111 1111 1111 1111 1111 1101       //-3
```

3.7.7 补零右移运算

补零右移(Zero-Fill Right Shift)又称为无符号右移,与右移不同的地方在于,右移后左侧的空位会用0补齐,这样只要移动的位数大于或等于1,那么它的符号位就始终是0,例如计算-5 >>> 1的结果是2147483645,计算过程如下:

```
1111 1111 1111 1111 1111 1111 1111 1011          //-5
0111 1111 1111 1111 1111 1111 1111 1101          //2147483645
```

3.8 组合运算符

组合运算符指的是把赋值运算符和其他运算符组合起来进行使用,是先计算后赋值的一种简写形式,代码如下:

```
let speed = 10, delta = 0.5;
speed += delta;             //10.5
```

代码中的speed+=delta可以写作speed=speed+delta,即把speed+delta的结果赋值给speed变量,这样可以少写一次speed。

其他组合运算符还有-=、*=、/=、%=、**=、&=、^=、~=、<<=、>>=、>>>=、&&=、||=、??=。

3.9 其他运算符

有一些运算符在之前的章节里已经介绍过了,例如访问数组元素的[]、访问对象属性的.和获取数据类型的typeof。JavaScript中还有其他比较常用的运算符,因为需要涉及更复杂的结构类型,所以将在后面各章中进行介绍。这些运算符有可选链(Optional Chaining)、解构赋值(Destructuring Assignment)、rest运算符、instanceof、delete和await。

3.10 优先级表

表3-1列出了运算的优先级和计算顺序,由高到低,方便进行参考,同一个单元格的运算符优先级相同。

表3-1 运算符优先级和计算顺序

运算符	含义	计算顺序
()	分组	—
.,[], new,	访问对象属性,访问数组元素 创建对象	从左到右 —

续表

运算符	含义	计算顺序
(),?.	调用函数,可选链	从左到右
++,--	后缀自增,后缀自减	—
!,~ +,-,++,-- typeof,delete,await	逻辑非,取反 一元加,一元减,前缀自增,前缀自减 获取类型,删除对象属性,异步等待	从右到左
**	幂运算	从右到左
*, /, %	乘法,除法,取模	从左到右
+, -	加法,减法	从左到右
<<, >>, >>>	左移,右移,补零右移	从左到右
<, <=, >, >= instanceof	小于,小于或等于,大于,大于或等于 类型检测	从左到右
==, ===, !=, !==	相等,严格相等,不相等,严格不相等	从左到右
&	按位与	从左到右
^	按位异或	从左到右
|	按位或	从左到右
&&	逻辑与	从左到右
||	逻辑或	从左到右
??	空值合并	从左到右
?:	三目运算符	从右到左
=,+=,-=,**=,*=,/=,%= <<=, >>=, >>>= &=, ^=, |= &&=, ||=, ??=	赋值,加等于,减等于,幂等于,乘等于,除等于,取模等于 左移等于,右移等于,补零右移等于 按位与等于,按位异或等于,按位或等于 逻辑与等于,逻辑或等于,空值合并等于	从右到左

3.11 小结

使用运算符可以编写更复杂的 JavaScript 表达式,以满足不同的业务需求。虽然运算符的种类众多,但是有一些是常用的,而另一些则并不常用,可以随时通过查阅本书来复习运算符的用法。本章的重点内容有以下几点。

(1) JavaScript 的运算符包括赋值、数学、比较、逻辑、三目、位和组合等。

(2) 运算符有不同的优先级和计算顺序。

(3) 小括号可以改变优先级。

(4) +和-既可以是一元也可以是二元运算符。

(5) +可以做加法,也可以做字符串拼接操作,还能够将非数字类型转换为数字类型。

(6) 逻辑与、逻辑或、空值合并运算符返回的是表达式计算的结果,并且有短路特性。