3.4.8　Object 类型

ECMAScript 中的对象其实就是一组数据和功能的集合。对象通过 new 操作符后跟对象类型的名称来创建。开发者可以通过创建 Object 类型的实例来创建自己的对象，然后再给对象添加属性和方法：

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

这个语法类似 Java，但 ECMAScript 只要求在给构造函数提供参数时使用括号。如果没有参数，如上面的例子所示，那么完全可以省略括号（不推荐）：

```
let o = new Object;  // 合法，但不推荐
```

Object 的实例本身并不是很有用，但理解与它相关的概念非常重要。类似 Java 中的 java.lang.Object，ECMAScript 中的 Object 也是派生其他对象的基类。Object 类型的所有属性和方法在派生的对象上同样存在。

每个 Object 实例都有如下属性和方法。

- constructor：用于创建当前对象的函数。在前面的例子中，这个属性的值就是 Object()函数。
- hasOwnProperty(*propertyName*)：用于判断当前对象实例（不是原型）上是否存在给定的属性。要检查的属性名必须是字符串（如 o.hasOwnProperty("name")）或符号。
- isPrototypeOf(*object*)：用于判断当前对象是否为另一个对象的原型。（第 8 章将详细介绍原型。）
- propertyIsEnumerable(*propertyName*)：用于判断给定的属性是否可以使用（本章稍后讨论的）for-in 语句枚举。与 hasOwnProperty()一样，属性名必须是字符串。
- toLocaleString()：返回对象的字符串表示，该字符串反映对象所在的本地化执行环境。
- toString()：返回对象的字符串表示。
- valueOf()：返回对象对应的字符串、数值或布尔值表示。通常与 toString()的返回值相同。

因为在 ECMAScript 中 Object 是所有对象的基类，所以任何对象都有这些属性和方法。第 8 章将介绍对象间的继承机制。

> **注意**　严格来讲，ECMA-262 中对象的行为不一定适合 JavaScript 中的其他对象。比如浏览器环境中的 BOM 和 DOM 对象，都是由宿主环境定义和提供的宿主对象。而宿主对象不受 ECMA-262 约束，所以它们可能会也可能不会继承 Object。

3.5　操作符

ECMA-262 描述了一组可用于操作数据值的**操作符**，包括数学操作符（如加、减）、位操作符、关系操作符和相等操作符等。ECMAScript 中的操作符是独特的，因为它们可用于各种值，包括字符串、数值、布尔值，甚至还有对象。在应用给对象时，操作符通常会调用 valueOf()和/或 toString()方法来取得可以计算的值。

3.5.1　一元操作符

只操作一个值的操作符叫**一元操作符**（unary operator）。一元操作符是 ECMAScript 中最简单的操作符。

1. 递增/递减操作符

递增和递减操作符直接照搬自 C 语言，但有两个版本：前缀版和后缀版。顾名思义，前缀版就是位于要操作的变量前头，后缀版就是位于要操作的变量后头。前缀递增操作符会给数值加 1，把两个加号（++）放到变量前头即可：

```
let age = 29;
++age;
```

在这个例子中，前缀递增操作符把 age 的值变成了 30（给之前的值 29 加 1）。因此，它实际上等于如下表达式：

```
let age = 29;
age = age + 1;
```

前缀递减操作符也类似，只不过是从一个数值减 1。使用前缀递减操作符，只要把两个减号（--）放到变量前头即可：

```
let age = 29;
--age;
```

执行操作后，变量 age 的值变成了 28（从 29 减 1）。

无论使用前缀递增还是前缀递减操作符，变量的值都会在语句被求值之前改变。（在计算机科学中，这通常被称为具有**副作用**。）请看下面的例子：

```
let age = 29;
let anotherAge = --age + 2;

console.log(age);         // 28
console.log(anotherAge);  // 30
```

在这个例子中，变量 anotherAge 以 age 减 1 后的值再加 2 进行初始化。因为递减操作先发生，所以 age 的值先变成 28，然后再加 2，结果是 30。

前缀递增和递减在语句中的优先级是相等的，因此会从左到右依次求值。比如：

```
let num1 = 2;
let num2 = 20;
let num3 = --num1 + num2;
let num4 = num1 + num2;
console.log(num3);  // 21
console.log(num4);  // 21
```

这里，num3 等于 21 是因为 num1 先减 1 之后才加 num2。变量 num4 也是 21，那是因为加法使用的也是递减后的值。

递增和递减的后缀版语法一样（分别是++和--），只不过要放在变量后面。后缀版与前缀版的主要区别在于，后缀版递增和递减在语句被求值后才发生。在某些情况下，这种差异没什么影响，比如：

```
let age = 29;
age++;
```

把递增操作符放到变量后面不会改变语句执行的结果，因为递增是唯一的操作。可是，在跟其他操作混合时，差异就会变明显，比如：

```
let num1 = 2;
let num2 = 20;
let num3 = num1-- + num2;
let num4 = num1 + num2;
```

```
console.log(num3);  // 22
console.log(num4);  // 21
```

这个例子跟前面的那个一样，只是把前缀递减改成了后缀递减，区别很明显。在使用前缀版的例子中，num3 和 num4 的值都是 21。而在这个例子中，num3 的值是 22，num4 的值是 21。这里的不同之处在于，计算 num3 时使用的是 num1 的原始值（2），而计算 num4 时使用的是 num1 递减后的值（1）。

这 4 个操作符可以作用于任何值，意思是不限于整数——字符串、布尔值、浮点值，甚至对象都可以。递增和递减操作符遵循如下规则。

- 对于字符串，如果是有效的数值形式，则转换为数值再应用改变。变量类型从字符串变成数值。
- 对于字符串，如果不是有效的数值形式，则将变量的值设置为 NaN 。变量类型从字符串变成数值。
- 对于布尔值，如果是 false，则转换为 0 再应用改变。变量类型从布尔值变成数值。
- 对于布尔值，如果是 true，则转换为 1 再应用改变。变量类型从布尔值变成数值。
- 对于浮点值，加 1 或减 1。
- 如果是对象，则调用其（第 5 章会详细介绍的）valueOf()方法取得可以操作的值。对得到的值应用上述规则。如果是 NaN，则调用 toString()并再次应用其他规则。变量类型从对象变成数值。

下面的例子演示了这些规则：

```
let s1 = "2";
let s2 = "z";
let b = false;
let f = 1.1;
let o = {
  valueOf() {
    return -1;
  }
};

s1++;  // 值变成数值 3
s2++;  // 值变成 NaN
b++;   // 值变成数值 1
f--;   // 值变成 0.10000000000000009（因为浮点数不精确）
o--;   // 值变成-2
```

2. 一元加和减

一元加和减操作符对大多数开发者来说并不陌生，它们在 ECMAScript 中跟在高中数学中的用途一样。一元加由一个加号（+）表示，放在变量前头，对数值没有任何影响：

```
let num = 25;
num = +num;
console.log(num); // 25
```

如果将一元加应用到非数值，则会执行与使用 Number()转型函数一样的类型转换：布尔值 false 和 true 转换为 0 和 1，字符串根据特殊规则进行解析，对象会调用它们的 valueOf()和/或 toString()方法以得到可以转换的值。

下面的例子演示了一元加在应用到不同数据类型时的行为：

```
let s1 = "01";
let s2 = "1.1";
```

```
let s3 = "z";
let b = false;
let f = 1.1;
let o = {
  valueOf() {
    return -1;
  }
};

s1 = +s1;  // 值变成数值 1
s2 = +s2;  // 值变成数值 1.1
s3 = +s3;  // 值变成 NaN
b = +b;    // 值变成数值 0
f = +f;    // 不变，还是 1.1
o = +o;    // 值变成数值-1
```

一元减由一个减号（-）表示，放在变量前头，主要用于把数值变成负值，如把 1 转换为–1。示例如下：

```
let num = 25;
num = -num;
console.log(num);  // -25
```

对数值使用一元减会将其变成相应的负值（如上面的例子所示）。在应用到非数值时，一元减会遵循与一元加同样的规则，先对它们进行转换，然后再取负值：

```
let s1 = "01";
let s2 = "1.1";
let s3 = "z";
let b = false;
let f = 1.1;
let o = {
  valueOf() {
    return -1;
  }
};

s1 = -s1;  // 值变成数值-1
s2 = -s2;  // 值变成数值-1.1
s3 = -s3;  // 值变成 NaN
b = -b;    // 值变成数值 0
f = -f;    // 变成-1.1
o = -o;    // 值变成数值 1
```

一元加和减操作符主要用于基本的算术，但也可以像上面的例子那样，用于数据类型转换。

3.5.2 位操作符

接下来要介绍的操作符用于数值的底层操作，也就是操作内存中表示数据的比特（位）。ECMAScript 中的所有数值都以 IEEE 754 64 位格式存储，但位操作并不直接应用到 64 位表示，而是先把值转换为 32 位整数，再进行位操作，之后再把结果转换为 64 位。对开发者而言，就好像只有 32 位整数一样，因为 64 位整数存储格式是不可见的。既然知道了这些，就只需要考虑 32 位整数即可。

有符号整数使用 32 位的前 31 位表示整数值。第 32 位表示数值的符号，如 0 表示正，1 表示负。这一位称为**符号位**（sign bit），它的值决定了数值其余部分的格式。正值以真正的二进制格式存储，即 31 位中的每一位都代表 2 的幂。第一位（称为第 0 位）表示 2^0，第二位表示 2^1，依此类推。如果一个位是

空的，则以 0 填充，相当于忽略不计。比如，数值 18 的二进制格式为 00000000000000000000000000010010，或更精简的 10010。后者是用到的 5 个有效位，决定了实际的值（如图 3-1 所示）。

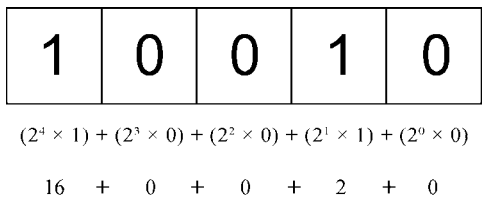

图　3-1

负值以一种称为**二补数**（或补码）的二进制编码存储。一个数值的二补数通过如下 3 个步骤计算得到：

(1) 确定绝对值的二进制表示（如，对于–18，先确定 18 的二进制表示）；

(2) 找到数值的一补数（或反码），换句话说，就是每个 0 都变成 1，每个 1 都变成 0；

(3) 给结果加 1。

基于上述步骤确定–18 的二进制表示，首先从 18 的二进制表示开始：

```
0000  0000  0000  0000  0000  0000  0001  0010
```

然后，计算一补数，即反转每一位的二进制值：

```
1111  1111  1111  1111  1111  1111  1110  1101
```

最后，给一补数加 1：

```
1111  1111  1111  1111  1111  1111  1110  1101
                        1
-----------------------------------------------
1111  1111  1111  1111  1111  1111  1110  1110
```

那么，–18 的二进制表示就是 11111111111111111111111111101110。要注意的是，在处理有符号整数时，我们无法访问第 31 位。

ECMAScript 会帮我们记录这些信息。在把负值输出为一个二进制字符串时，我们会得到一个前面加了减号的绝对值，如下所示：

```
let num = -18;
console.log(num.toString(2)); // "-10010"
```

在将–18 转换为二进制字符串时，结果得到–10010。转换过程会求得二补数，然后再以更符合逻辑的形式表示出来。

> **注意**　默认情况下，ECMAScript 中的所有整数都表示为有符号数。不过，确实存在无符号整数。对无符号整数来说，第 32 位不表示符号，因为只有正值。无符号整数比有符号整数的范围更大，因为符号位被用来表示数值了。

在对 ECMAScript 中的数值应用位操作符时，后台会发生转换：64 位数值会转换为 32 位数值，然后执行位操作，最后再把结果从 32 位转换为 64 位存储起来。整个过程就像处理 32 位数值一样，这让

二进制操作变得与其他语言中类似。但这个转换也导致了一个奇特的副作用，即特殊值 NaN 和 Infinity 在位操作中都会被当成 0 处理。

如果将位操作符应用到非数值，那么首先会使用 Number() 函数将该值转换为数值（这个过程是自动的），然后再应用位操作。最终结果是数值。

1. 按位非

按位非操作符用波浪符（~）表示，它的作用是返回数值的一补数。按位非是 ECMAScript 中为数不多的几个二进制数学操作符之一。看下面的例子：

```
let num1 = 25;       // 二进制 00000000000000000000000000011001
let num2 = ~num1;    // 二进制 11111111111111111111111111100110
console.log(num2);  // -26
```

这里，按位非操作符作用到了数值 25，得到的结果是–26。由此可以看出，按位非的最终效果是对数值取反并减 1，就像执行如下操作的结果一样：

```
let num1 = 25;
let num2 = -num1 - 1;
console.log(num2);   // "-26"
```

实际上，尽管两者返回的结果一样，但位操作的速度快得多。这是因为位操作是在数值的底层表示上完成的。

2. 按位与

按位与操作符用和号（&）表示，有两个操作数。本质上，按位与就是将两个数的每一个位对齐，然后基于真值表中的规则，对每一位执行相应的与操作。

第一个数值的位	第二个数值的位	结　果
1	1	1
1	0	0
0	1	0
0	0	0

按位与操作在两个位都是 1 时返回 1，在任何一位是 0 时返回 0。

下面看一个例子，我们对数值 25 和 3 求与操作，如下所示：

```
let result = 25 & 3;
console.log(result); // 1
```

25 和 3 的按位与操作的结果是 1。为什么呢？看下面的二进制计算过程：

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
-------------------------------------------
AND = 0000 0000 0000 0000 0000 0000 0000 0001
```

如上所示，25 和 3 的二进制表示中，只有第 0 位上的两个数都是 1。于是结果数值的所有其他位都会以 0 填充，因此结果就是 1。

3. 按位或

按位或操作符用管道符（|）表示，同样有两个操作数。按位或遵循如下真值表：

第一个数值的位	第二个数值的位	结　果
1	1	1
1	0	1
0	1	1
0	0	0

按位或操作在至少一位是 1 时返回 1，两位都是 0 时返回 0。

仍然用按位与的示例，如果对 25 和 3 执行按位或，代码如下所示：

```
let result = 25 | 3;
console.log(result); // 27
```

可见 25 和 3 的按位或操作的结果是 27：

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
--------------------------------------------
 OR = 0000 0000 0000 0000 0000 0000 0001 1011
```

在参与计算的两个数中，有 4 位都是 1，因此它们直接对应到结果上。二进制码 11011 等于 27。

4. 按位异或

按位异或用脱字符（^）表示，同样有两个操作数。下面是按位异或的真值表：

第一个数的位	第二个数的位	结　果
1	1	0
1	0	1
0	1	1
0	0	0

按位异或与按位或的区别是，它只在一位上是 1 的时候返回 1（两位都是 1 或 0，则返回 0）。

对数值 25 和 3 执行按位异或操作：

```
let result = 25 ^ 3;
console.log(result); // 26
```

可见，25 和 3 的按位异或操作结果为 26，如下所示：

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
--------------------------------------------
XOR = 0000 0000 0000 0000 0000 0000 0001 1010
```

两个数在 4 位上都是 1，但两个数的第 0 位都是 1，因此那一位在结果中就变成了 0。其余位上的 1 在另一个数上没有对应的 1，因此会直接传递到结果中。二进制码 11010 等于 26。（注意，这比对同样两个值执行按位或操作得到的结果小 1。）

5. 左移

左移操作符用两个小于号（<<）表示，会按照指定的位数将数值的所有位向左移动。比如，如果数值 2（二进制 10）向左移 5 位，就会得到 64（二进制 1000000），如下所示：

```
let oldValue = 2;              // 等于二进制 10
let newValue = oldValue << 5;  // 等于二进制 1000000，即十进制 64
```

注意在移位后，数值右端会空出 5 位。左移会以 0 填充这些空位，让结果是完整的 32 位数值（见图 3-2）。

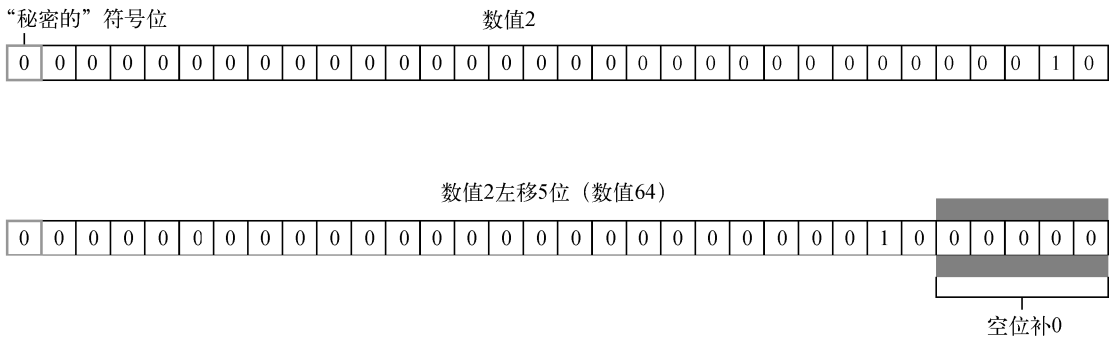

图 3-2

注意，左移会保留它所操作数值的符号。比如，如果–2 左移 5 位，将得到–64，而不是正 64。

6. 有符号右移

有符号右移由两个大于号（>>）表示，会将数值的所有 32 位都向右移，同时保留符号（正或负）。有符号右移实际上是左移的逆运算。比如，如果将 64 右移 5 位，那就是 2：

```
let oldValue = 64;              // 等于二进制 1000000
let newValue = oldValue >> 5;  // 等于二进制 10，即十进制 2
```

同样，移位后就会出现空位。不过，右移后空位会出现在左侧，且在符号位之后（见图 3-3）。ECMAScript 会用符号位的值来填充这些空位，以得到完整的数值。

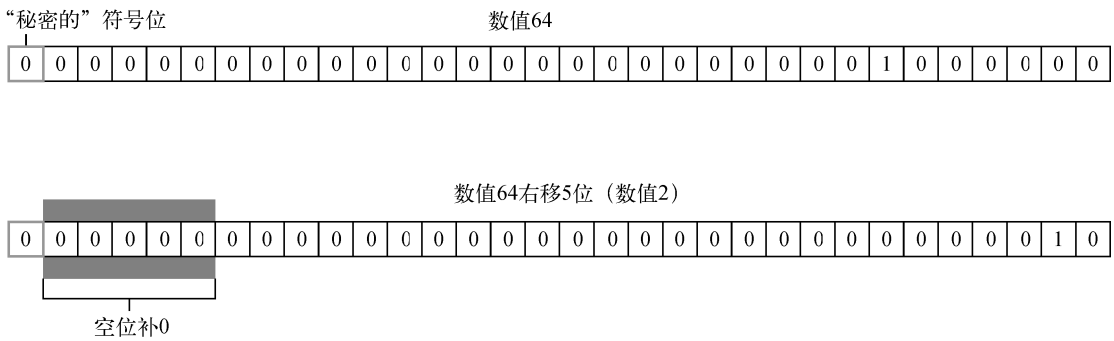

图 3-3

7. 无符号右移

无符号右移用 3 个大于号表示（>>>），会将数值的所有 32 位都向右移。对于正数，无符号右移与有符号右移结果相同。仍然以前面有符号右移的例子为例，64 向右移动 5 位，会变成 2：

```
let oldValue = 64;              // 等于二进制 1000000
let newValue = oldValue >>> 5;  // 等于二进制 10，即十进制 2
```

对于负数，有时候差异会非常大。与有符号右移不同，无符号右移会给空位补 0，而不管符号位是什么。对正数来说，这跟有符号右移效果相同。但对负数来说，结果就差太多了。无符号右移操作符将

负数的二进制表示当成正数的二进制表示来处理。因为负数是其绝对值的二补数，所以右移之后结果变得非常之大，如下面的例子所示：

```
let oldValue = -64;              // 等于二进制 11111111111111111111111111000000
let newValue = oldValue >>> 5;   // 等于十进制 134217726
```

在对–64 无符号右移 5 位后，结果是 134 217 726。这是因为–64 的二进制表示是 11111111111111111111111111000000，无符号右移却将它当成正值，也就是 4 294 967 232。把这个值右移 5 位后，结果是 00000111111111111111111111111110，即 134 217 726。

3.5.3　布尔操作符

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

1. 逻辑非

逻辑非操作符由一个叹号（`!`）表示，可应用给 ECMAScript 中的任何值。这个操作符始终返回布尔值，无论应用到的是什么数据类型。逻辑非操作符首先将操作数转换为布尔值，然后再对其取反。换句话说，逻辑非操作符会遵循如下规则。

- 如果操作数是对象，则返回 `false`。
- 如果操作数是空字符串，则返回 `true`。
- 如果操作数是非空字符串，则返回 `false`。
- 如果操作数是数值 0，则返回 `true`。
- 如果操作数是非 0 数值（包括 `Infinity`），则返回 `false`。
- 如果操作数是 `null`，则返回 `true`。
- 如果操作数是 `NaN`，则返回 `true`。
- 如果操作数是 `undefined`，则返回 `true`。

以下示例验证了上述行为：

```
console.log(!false);   // true
console.log(!"blue");  // false
console.log(!0);       // true
console.log(!NaN);     // true
console.log(!"");      // true
console.log(!12345);   // false
```

逻辑非操作符也可以用于把任意值转换为布尔值。同时使用两个叹号（`!!`），相当于调用了转型函数 `Boolean()`。无论操作数是什么类型，第一个叹号总会返回布尔值。第二个叹号对该布尔值取反，从而给出变量真正对应的布尔值。结果与对同一个值使用 `Boolean()`函数是一样的：

```
console.log(!!"blue"); // true
console.log(!!0);      // false
console.log(!!NaN);    // false
console.log(!!"");     // false
console.log(!!12345);  // true
```

2. 逻辑与

逻辑与操作符由两个和号（`&&`）表示，应用到两个值，如下所示：

```
let result = true && false;
```

逻辑与操作符遵循如下真值表：

第一个操作数	第二个操作数	结　果
true	true	true
true	false	false
false	true	false
false	false	false

逻辑与操作符可用于任何类型的操作数，不限于布尔值。如果有操作数不是布尔值，则逻辑与并不一定会返回布尔值，而是遵循如下规则。

- 如果第一个操作数是对象，则返回第二个操作数。
- 如果第二个操作数是对象，则只有第一个操作数求值为 true 才会返回该对象。
- 如果两个操作数都是对象，则返回第二个操作数。
- 如果有一个操作数是 null，则返回 null。
- 如果有一个操作数是 NaN，则返回 NaN。
- 如果有一个操作数是 undefined，则返回 undefined。

逻辑与操作符是一种短路操作符，意思就是如果第一个操作数决定了结果，那么永远不会对第二个操作数求值。对逻辑与操作符来说，如果第一个操作数是 false，那么无论第二个操作数是什么值，结果也不可能等于 true。看下面的例子：

```
let found = true;
let result = (found && someUndeclaredVariable); // 这里会出错
console.log(result); // 不会执行这一行
```

上面的代码之所以会出错，是因为 someUndeclaredVariable 没有事先声明，所以当逻辑与操作符对它求值时就会报错。变量 found 的值是 true，逻辑与操作符会继续求值变量 someUndeclaredVariable。但是由于 someUndeclaredVariable 没有定义，不能对它应用逻辑与操作符，因此就报错了。假如变量 found 的值是 false，那么就不会报错了：

```
let found = false;
let result = (found && someUndeclaredVariable);  // 不会出错
console.log(result);  // 会执行
```

这里，console.log 会成功执行。即使变量 someUndeclaredVariable 没有定义，由于第一个操作数是 false，逻辑与操作符也不会对它求值，因为此时对&&右边的操作数求值是没有意义的。在使用逻辑与操作符时，一定别忘了它的这个短路的特性。

3. 逻辑或

逻辑或操作符由两个管道符（||）表示，比如：

```
let result = true || false;
```

逻辑或操作符遵循如下真值表：

第一个操作数	第二个操作数	结　果
true	true	true
true	false	true
false	true	true
false	false	false